# Add Fault Tolerance – order & time

**Time, Clocks, and the Ordering of Events in a Distributed System**

Leslie Lamport

**Optimal Clock Synchronization**

T.K. Srikanth and Sam Toueg

Presenter: Feng Shao

(Some slides borrowed from Lamport)

# Why do we care about the “Time” in a distributed system?

- May need to know the time of day at which some event happens on a specific computer
  - external clock synchronization
- For two events that happened on different computers
  - May need to know the relative order
  - May need to know time interval
  - internal clock synchronization

# Physical Clocks

- Every computer contains a physical clock
- A clock is an electronic device that counts oscillations in a crystal at a particular frequency
- Count is typically divided and stored in a computer register
- Clock can be programmed to generate interrupts at regular intervals.
- This value can be used to timestamp an event on that computer
- Two events will have different timestamps only if clock resolution is sufficiently small
- Many applications are interested only in the order of events, not the exact time of day at which they occurred.

# Physical Clocks in Distributed Systems

- Does this work?
  - Synchronize all the clocks to some known high degree of accuracy, and then
  - Measure time relative to each local clock to determine order between two events

- Well, there are some problems…
  - It's difficult to synchronize the clocks
  - Crystal-based clocks tend to drift over time-count time at different rates, and diverge from each other
    - Physical variations in the crystals, temperature variations, etc.
    - Drift is small, but adds up over time
    - For quartz crystal time, typical drift rate is about one second every $10^6$ seconds=11.6days
    - Best atomic clocks have drift rate of one second in $10^{13}$ seconds = 300,000 years

# Logical Clocks

- Idea — abandon idea of physical time
- For many purposes, it is sufficient to know the **order** in which events occurred
- Lamport (1978) — introduce logical (*virtual*) *time*, to provide consistent event ordering

# TIME, CLOCKS AND THE ORDERING OF EVENTS IN A DISTRIBUTED SYSTEM

Leslie Lamport

# THE PAPER

- Handles the problem of clock drift in distributed systems
- Identify main function of computer clocks
- How to order events
  - Indicates which conditions clocks must satisfy to fulfill their role
- Introduces logical clocks

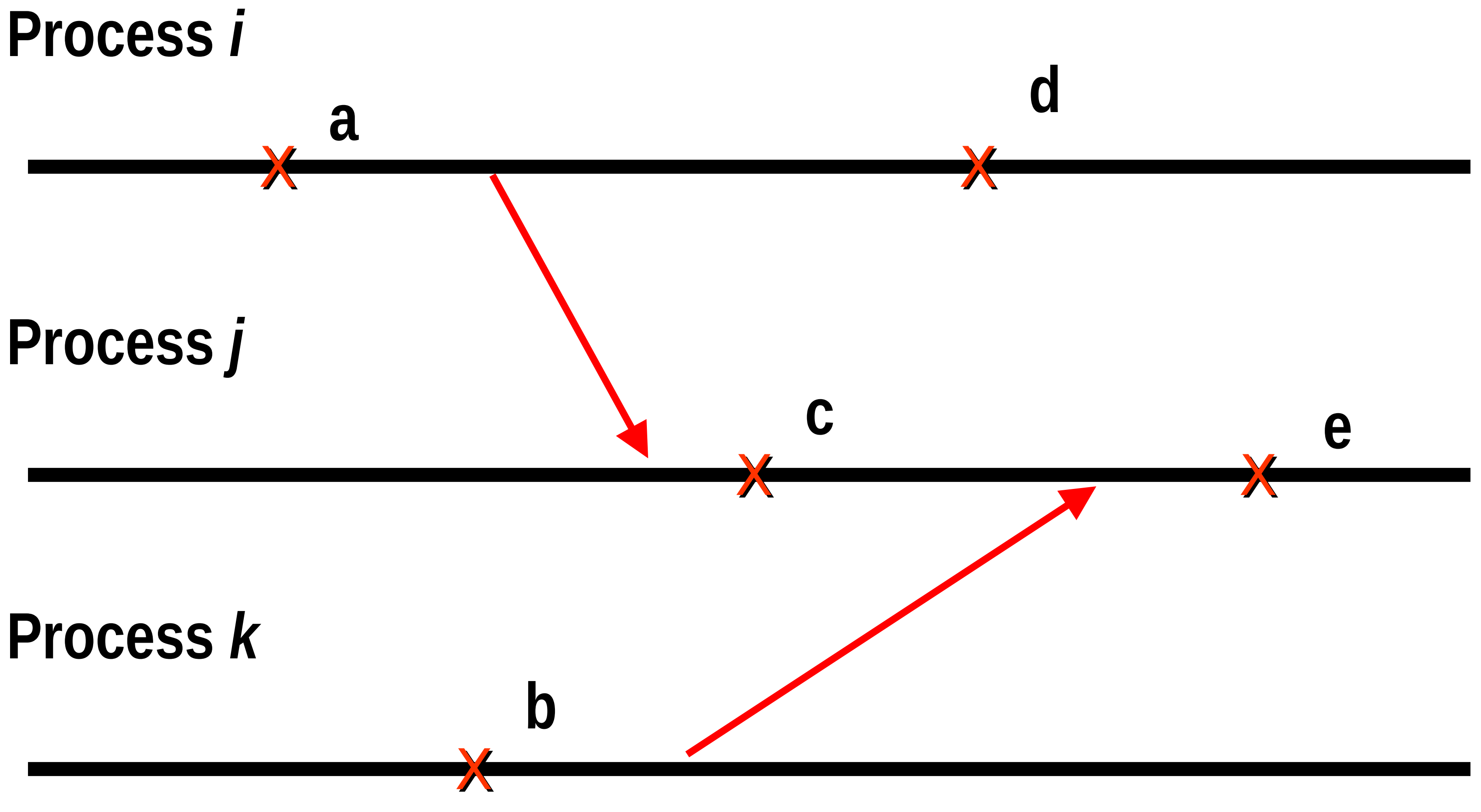
# ORDERING EVENTS

- Event ordering linked with concept of **causality**:
  - Saying that event *a* happened before event *b* is same as saying that event *a* could have affected the outcome of event *b*
  - If events *a* and *b* happen on processes that do not exchange any data, their exact ordering is not important

# Relation “has happened before” (I)

- Smallest relation satisfying the three conditions:
  - If *a* and *b* are events in the same process and *a* comes before *b*, then $a \rightarrow b$
  - If *a* is the sending of a message by a process and *b* its receipt by another process then $a \rightarrow b$
  - If $a \rightarrow b$ and b $\rightarrow c$ then $a \rightarrow c$.

# Example (I)



# Example (II)

- From first condition
  - $a \rightarrow d$
  - $c \rightarrow e$
- From second condition
  - $a \rightarrow c$
  - $b \rightarrow e$
- From third condition
  - $a \rightarrow e$

# Relation “has happened before” (II)

- We cannot always order events: relation “has happened before” is only a **partial order**
- If *a* did not happen before *b*, it cannot causally affect *b*.

# Logical clocks

- Verify the *clock condition*:
  - if $a \rightarrow b$ then $C\langle a\rangle \mathbf{<} C\langle b\rangle$

  and the two sub-conditions:

  - if *a* and *b* are events in process $P_i$ and *a* comes before *b*, then $C_i\langle a\rangle \mathbf{<} C_i\langle b\rangle$,
  - if *a* is the sending of a message by $P_i$ and *b* its receipt by $P_j$ then

    $C_i\langle a\rangle \mathbf{<} C_j\langle b\rangle$,

# Implementation rules

- Each process $P_i$ increments its clock $C_i$ between two consecutive events,

- If $a$ is the sending of a message $m$ by $P_i$ then $m$ includes a timestamp $T_m = C_i<a>$

  when $P_j$ receives $m$, it sets its clock to a value greater than or equal to its present value and greater than $T_m$.

# Defining a total order

- We can define a total ordering on the set of all system events

  $a \Rightarrow b$ if either $C_i\langle a\rangle < C_j\langle b\rangle$

  or

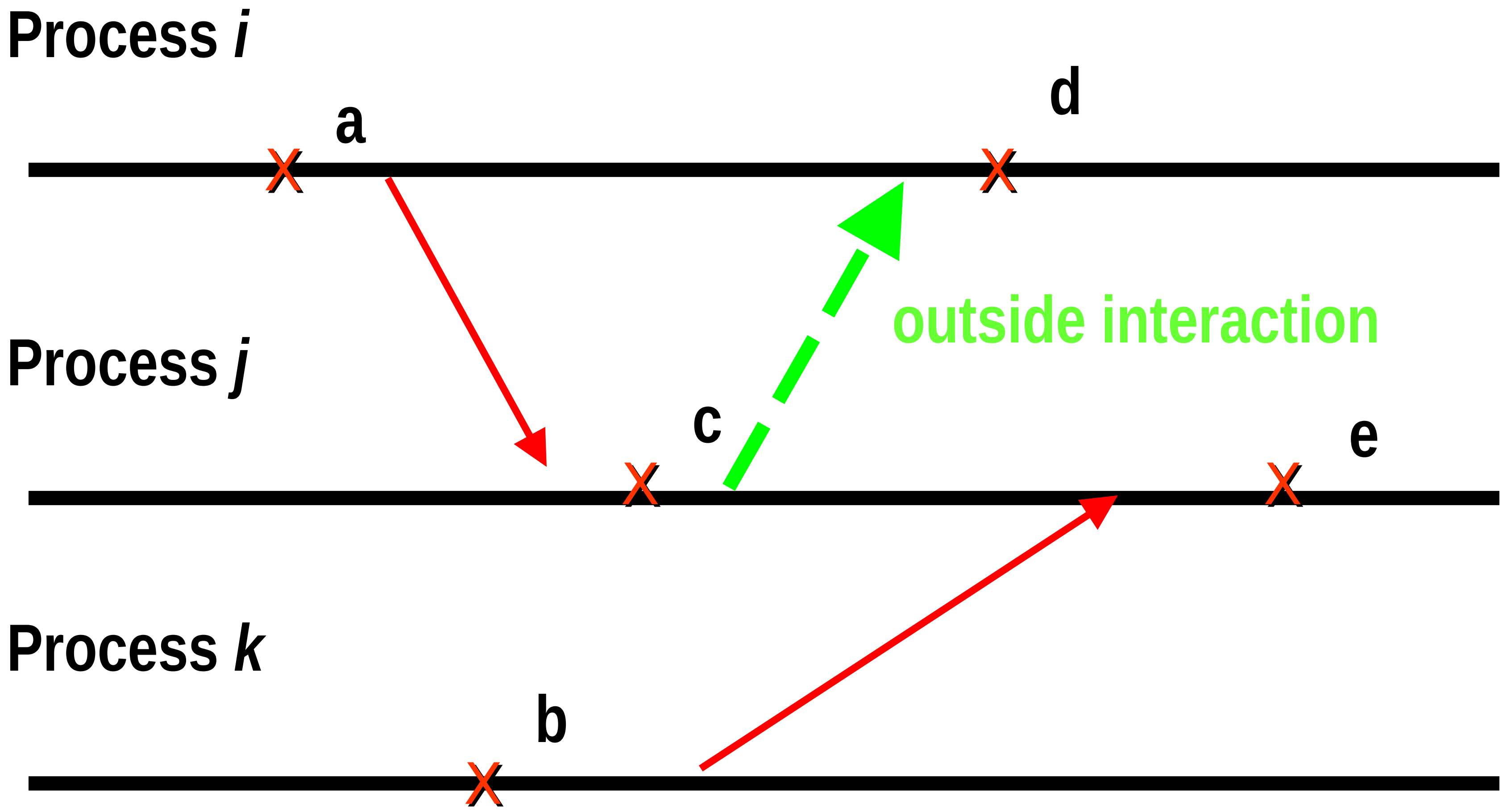
  $C_i\langle a\rangle = C_j\langle b\rangle$ and $P_i < P_j$.

- This ordering is *not unique*

# Anomalous behaviors

- Logical clocks have anomalous behaviors in the presence of outside interactions
  - carrying a diskette from one machine to another
  - dictating file changes over the phone
- Must use physical clocks

# Example



# Strong clock condition

- Let $S$ be set of all systems events plus the relevant external events
- For any events $a$, $b$ in S,
  if $a \rightarrow b$ then $C\langle a\rangle < C\langle b\rangle$

# Physical clock conditions

- There is a constant $k << 1$ such that for all $i$:

$$|d\, C_i(t)/dt - 1| < k$$

*The clock is neither too fast nor too slow*

- There is a constant $\varepsilon$ such that for all $i$, $j$:

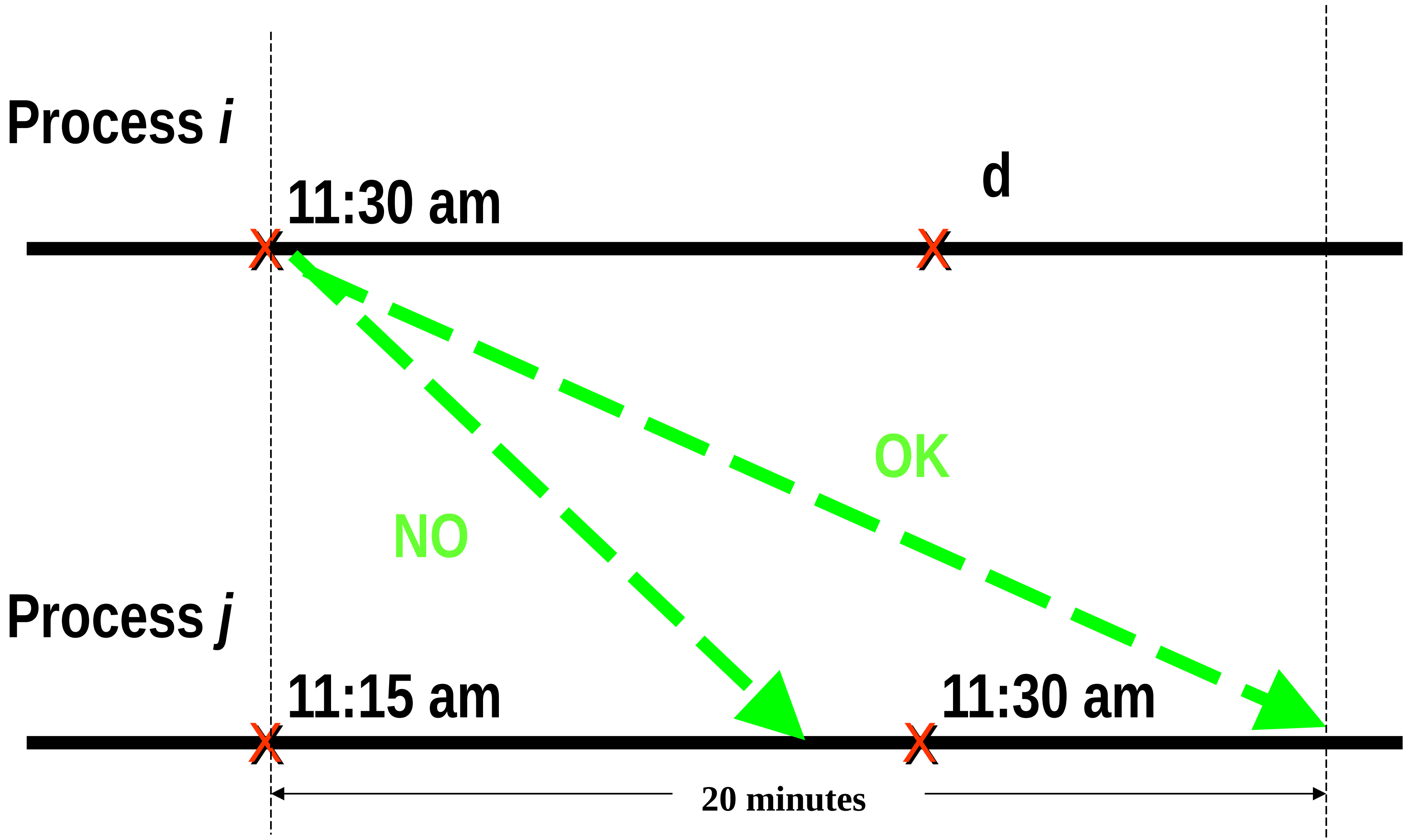
  - $$|C_i(t) - C_j(t)| < \varepsilon$$

*The clocks are more or less synchronized*

# Observations

- Like logical clocks, physical clocks cannot be rolled back
- Required accuracy of a physical clock depends on the minimum transmission delay of outside interactions
  - If it takes 20 minutes to carry a diskette between two machines their clocks can be off by up to 20 minutes

# Example



# Optimal Clock Synchronization

T. K. Srikanth and Sam Toueg

# Why do clock synchronization?

- Time-based computations on multiple machines
  - Applications that measure elapsed time
  - Agreeing on deadlines
  - Real time processes may need accurate timestamps

- Many applications require that clocks advance at similar rates
  - Real time scheduling events based on processor clock
  - Setting timeouts and measuring latencies
  - Ability to infer potential causality from timestamps

# Famous example

- Scud rockets launched by Iraq towards Israel
- Ground-based Patriot missiles fire back
- But missiles always missed the warhead!
- Why?

# Famous example

- Scud rockets launched by Iraq towards Israel
- Ground-based Patriot missiles fire back
- But missiles always missed the warhead!
- Why?
  - After 72 hours of waiting control system was out of sync relative to Patriot guidance system
  - “be at (x,y,z) at time t” was misinterpreted!

# Synchronization with failures

- A process is *faulty* if its behavior deviates from that prescribed by the algorithm it is running.

1. *Crash:* The process stops and does nothing from that point.
2. *Send omission:* The process crashes or omits to send messages that it is supposed to send.
3. *Receive omission:* The process crashes or does not receive messages sent to it.
4. *General omission:* The faulty process is subject to send omissions, receive omissions, or both.
5. *Arbitrary* (sometimes called *Byzantine*): The faulty process can exhibit any behavior, including malicious actions that will cause the system to fail.

# The System Model

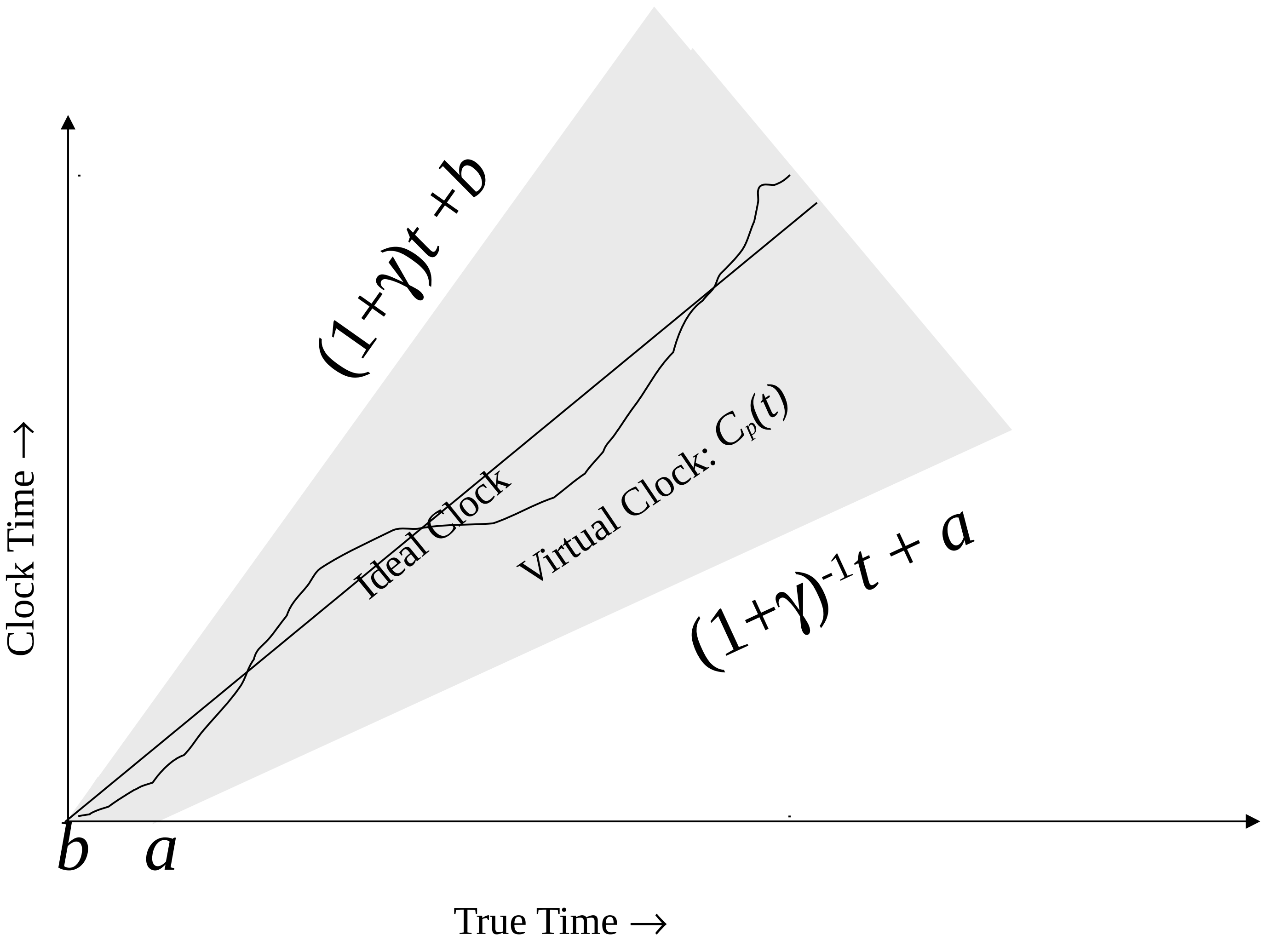
- Hardware clocks
  - Physical clock of process *q* designated $R_q(t)$
  - Clocks have a drift rate $\rho$:
    - $(1+\rho)^{-1}(t_2-t_1) \leq R_p(t_2) - R_p(t_1) \leq (1+\rho)(t_2-t_1)$
    - Implies that rate of drift is bounded by $d_r = \rho(2+\rho)/(1+\rho)$
    - For time t, general bounds:
      - $(1-\rho)t \leq (1+\rho)^{-1} t \leq R(t) \leq (1+\rho)t \leq (1-\rho)^{-1}t$

- There is a limit $t_{del}$ on message latency

# Clock synchronization goals

- A clock synchronization protocol implements a virtual clock function mapping real time $t$ to $C_p(t)$
- *Agreement* condition:
  - $|C_p(t) - C_q(t)| \leq D_{max}$ for all correct $p$, $q$
  - $D_{max}$ bounds the difference between two virtual clocks running on different processors
- *Accuracy* condition:
  - $(1+\gamma)^{-1}t + a \leq C_p(t) \leq (1+\gamma)t + b$, for constants a, b, $\gamma$
  - Says that *p*'s clock must be within a linear envelope of "real time"

# Clocks and True Time



# Authenticated Algorithm

- Solution for system of n processes, at most f of which are faulty

**cobegin**

**if** $C^{k-1}(t) = kP$

→sign and broadcast (*round k*) **fi**

//(not a sequential program)

**if** received f+1 signed messages (*round k*) ("accept")

→$C^{k}(t) := kP + a$;

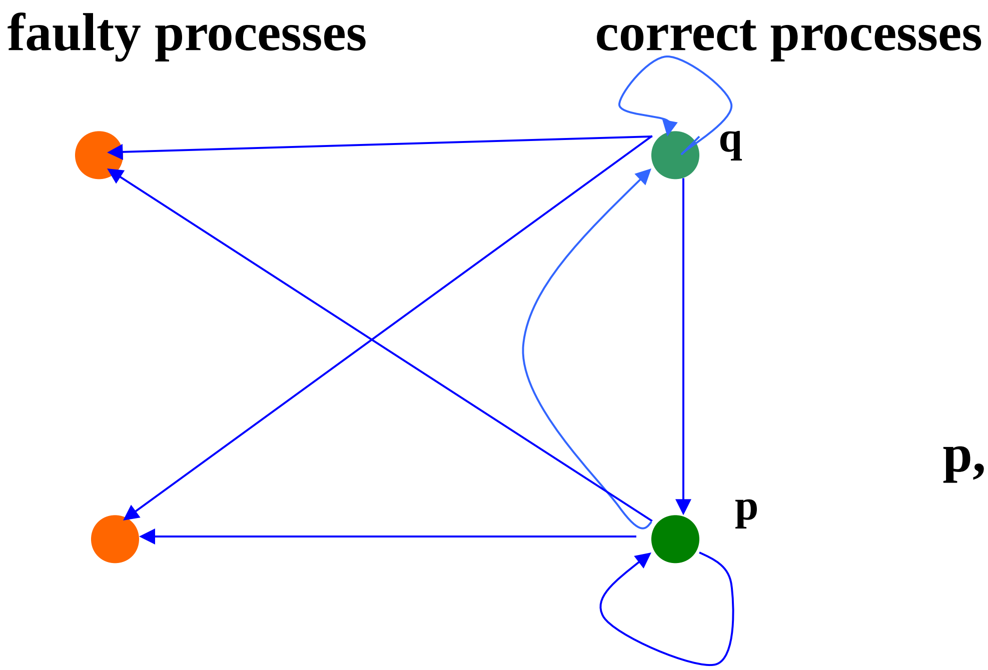
relay all $f$+1 signed messages to all **fi**

**coend**

# Observations

- Why relay?
  - Faulty processes do not necessarily broadcast.
- Why N > 2f?

N = 4, f = 2, suppose faulty processes get stuck and p, q want to resynchronize




**p, q cannot resynchronize !**

# Achieving Optimal Accuracy

- Bound on accuracy:

  for any synchronization, even in the absence of faults, accuracy cannot exceed that of the underlying hardware clocks

- Why algorithm 1 is not optimal?
  - Uncertainty of $t_{del}$ introduces a difference in the logical time between resyn.

# Optimality (informal description)

- Solution: compensate for the uncertainty of $t_{del}$:

  If a process accepts a (round k) message early, it delays the starting of the kth clock by $t_{del}/2(1+ \rho)$.

  If it accepts the message late, it advances the starting of kth clock by $t_{del}/2(1+ \rho)$.

- Suppose process i accepts (*round k*) message at time t, and let $T=C^{k-1}(t)$, $ß = t_{del}/2(1+ \rho)$
  - early: $T <= kP + ß$
  - late: $T > kP + ß$

- Proof of correctness: remarkably tricky, ignored here

# Unauthenticated algorithm

- The authenticated algorithm relies on properties of the message system:
  - *Correctness:* If at least $f$+1 correct processes broadcast ***round k*** messages by time $t$, then every correct process accepts a message by time $t+t_{del}$
  - *Unforgeability:* If no correct process broadcasts a ***round k*** message by time $t$, then no correct process accepts the message by time $t$ or earlier
  - *Relay*: If a correct process accepts the message ***round k*** at time $t$, then every correct process does so by time $t+t_{del}$

# Unauthenticated algorithm (II)

- A broadcast primitive which has the three properties

  To broadcast a (round k) message, a correct process sends (init, round k) to all.

  **for** each correct process:

  **if** received (init, round k) from at least f+ 1 distinct processes

  →send (echo, round k) to all;

  received (echo, round k) from at least f+ 1 distinct processes

  → send (echo, round k) to all;

  **fi**

  **if** received (echo, round k) from at least 2f+ 1 distinct processes

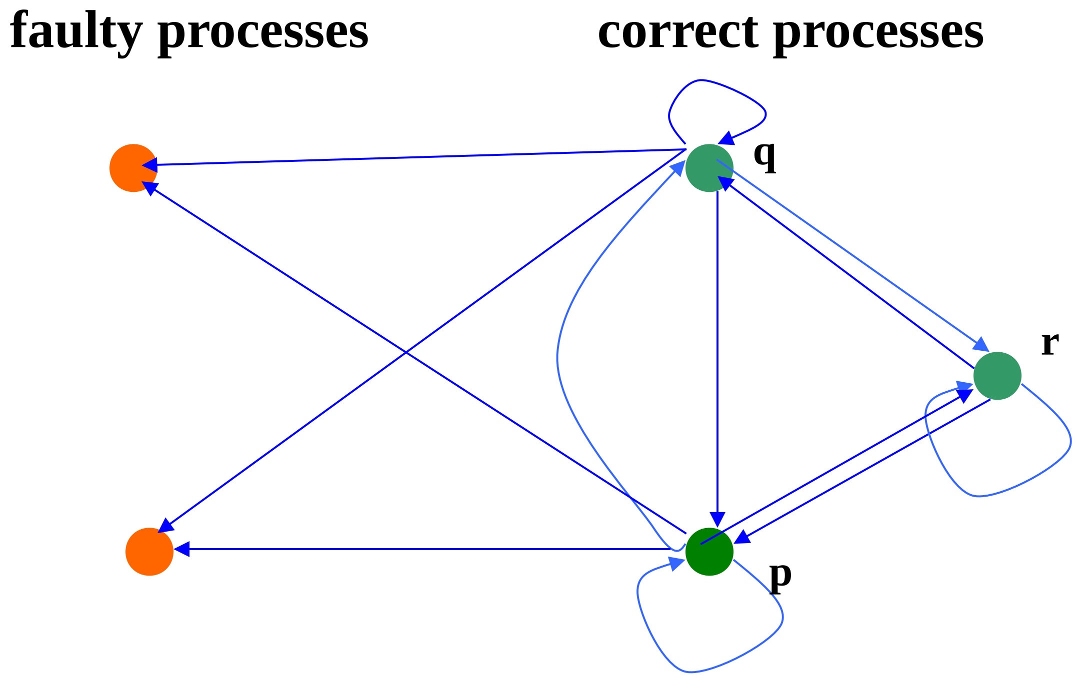
  →accept (round k)

  **fi**

- Requires n > 3f+1, in order to accept

# N > 3f +1

**N = 5, f = 2, suppose faulty processes get stuck, all three correct processes want to resynchronize**




**p, q, r never receive 2f +1 ( echo, round k), thus not accept**

# Simulating Authentication

- Nonauthenticated algorithm for clock synchronization for process p for round k

  **cobegin**

  **if** $C^{k-1}(t)$ = kP /* ready to start Ck */

  →broadcast (round k) **fi** /* using the broadcast primitive*/

  //

  **if** accepted the message (round k) /* according to the primitive */

  → $C^{k}(t)$ := kP + a **fi** /* start Ck */

  **coend**

- Message overhead: $O(n^2)$

# Restricted Models of failure

- Now assume arbitrary failure
- For other types of failures, including crash, sr-omission, the algorithm can be easily modified to achieve the optimality in the number of fault processes.

# Summary

- A unified solution for synchronizing clocks.
- In practice, quality of synchronization remains relatively poor
- At best synchronization will be limited by quality of physical clocks, rates of physical clock drift, and uncertainty in latencies

??? //