Distributed Systems 1: Synchronization

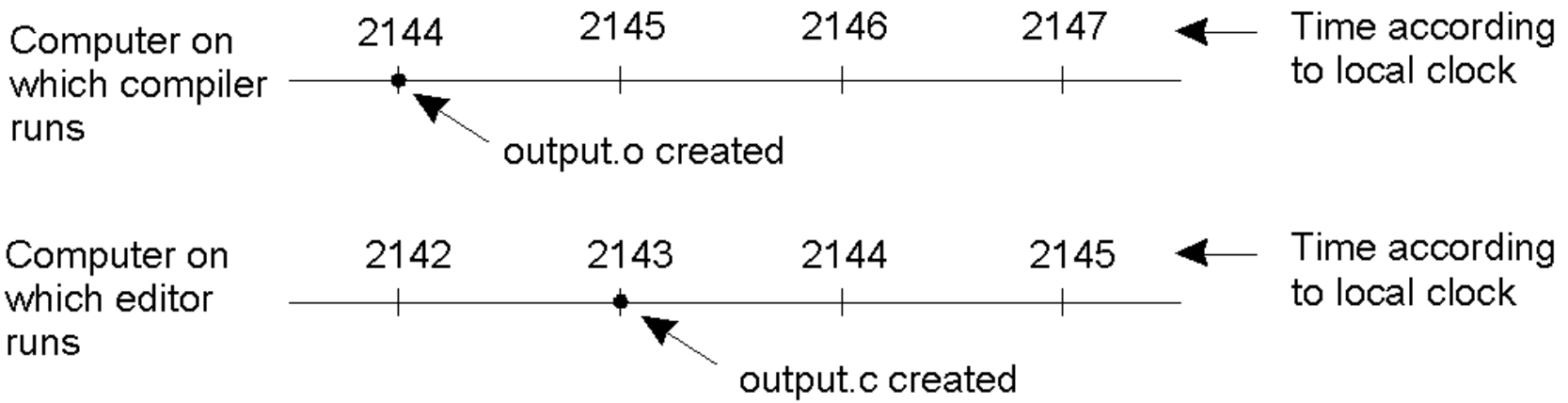
Gian Pietro Picco
Dipartimento di Ingegneria e Scienza dell'Informazione
University of Trento, Italy
gianpietro.picco@unitn.it
http://disi.unitn.it/~picco

Synchronization in Distributed Systems

- The problem of synchronizing concurrent activities arises also in non-distributed systems
- However, distribution complicates matters:
  - Absence of a global physical clock
  - Absence of globally shared memory
  - Partial failures
- In these lectures, we study distributed algorithms for:
  - Synchronize (physical) clocks to approximate global time
  - Abstract time with logical clocks capturing (causal) ordering
  - Collect global state
  - Coordinate access to shared resources

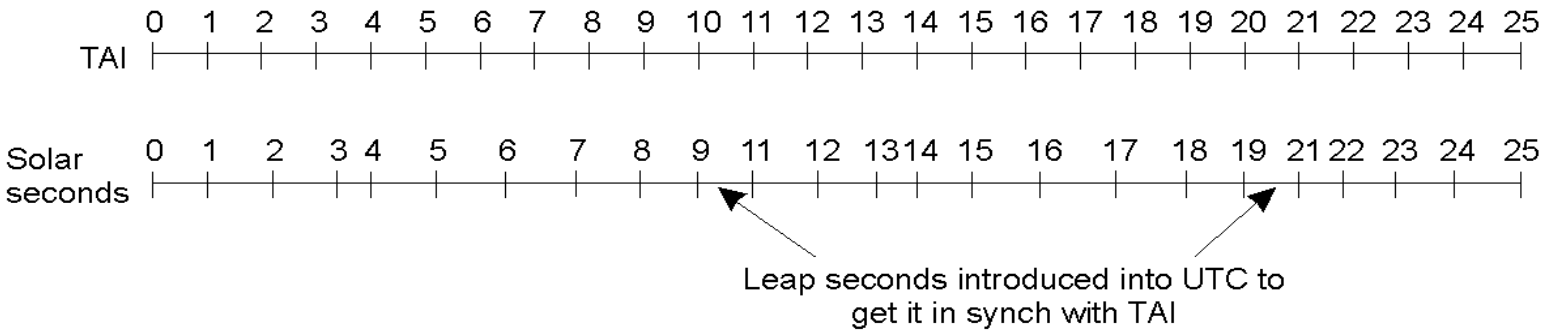
Time and Distributed Systems

- Time plays a fundamental role in many applications:
  - Execute a given action at a given time
  - Timestamping objects/data/messages enables reconstruction of event ordering
    - File versioning
    - Distributed debugging
    - Security algorithms
- Problem: ensure all machines “see” the same global time



Time

- Clocks vs. timers
- Time is a tricky issue per se:
  - Up to 1940, time is measured astronomically
    - 1 second = 1/86400th of a solar day
    - Earth is slowing down, making measures “inaccurate”
  - Since 1948, atomic clocks (International Atomic Time)
    - 1 second = 9,192,631,770 transitions of an atom of Cesium 133
    - Collected and averaged in Paris from 50 labs around the world
  - Skew between TAI and solar days accommodated by UTC (Universal Time Coordinates) when greater than 800ms
    - Greenwich Mean Time is only astronomical
    - About 30 leap seconds from 1958 to now
    - UTC disseminate via radio stations (DCF77 in Europe, WWV in US), GPS and GEOS satellite systems

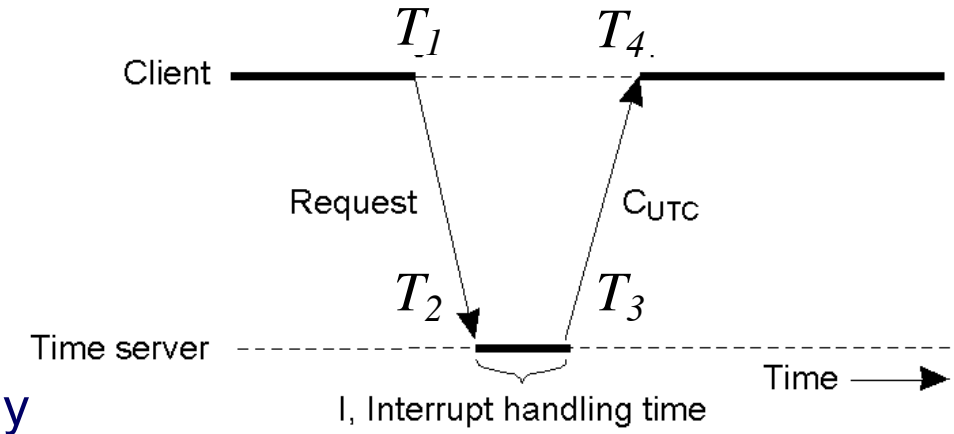


Synchronizing Physical Clocks

- To guarantee synchronization:
  - Maximum ***clock drift*** rate ρ is a constant of the timer
    - For ordinary quartz crystals, $\rho=10^{-6}$ s/s, i.e., 1s every 11.6 days
  - Maximum allowed ***clock skew*** δ is an engineering parameter
  - If two clocks are drifting in opposite directions, during a time interval Δt they accumulate a skew of $2\rho\Delta t$
    ➔ resynch needed at least every $\delta/2\rho$ seconds
- The problem is either:
  - Synchronize all clocks against a single one, usually the one with external, accurate time information (*accuracy*)
  - Synchronize all clocks among themselves (*agreement*)
- At least time monotonicity must be preserved
  - i.e., time cannot run backwards!
- Several protocols have been devised

Server-based Solutions

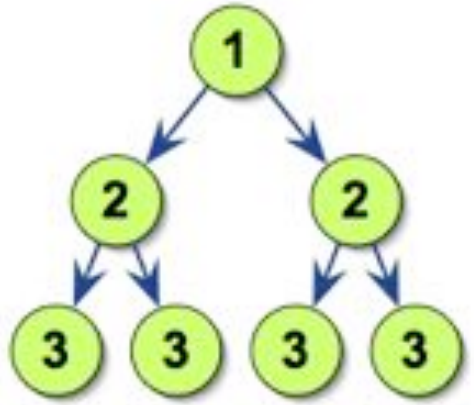
- Periodically, each client
  - sends a request to the time server, which replies with its clock
  - computes the offset θ, and
  - sets its local clock accordingly
- Assumption:
  - messages travel fast w.r.t. time accuracy
- Problems:
  - Major: time might run ***backwards*** on the client machine; therefore, introduce change gradually
    - e.g., advance clock 9ms instead of 10ms on each clock tick
  - Minor: it takes a non-zero amount of time to get the message to the server and back
    - measure round-trip time by encoding more timestamps (see below)
    - average over several measurements, …



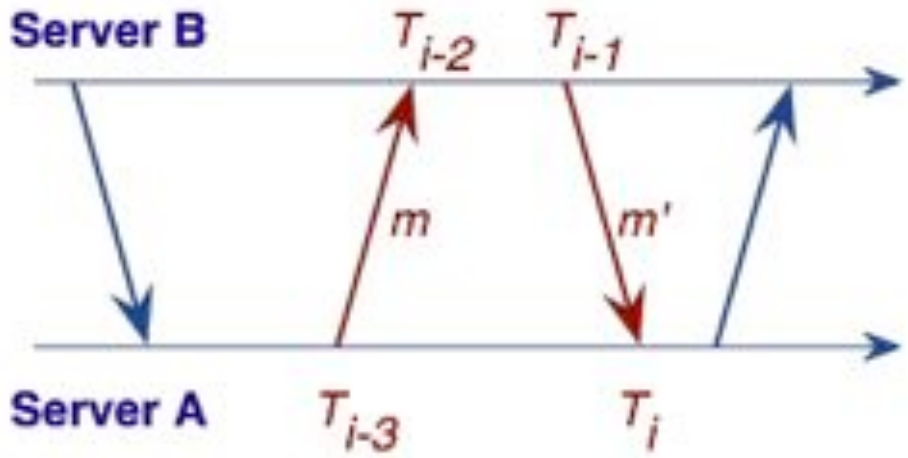
$$\theta = T_3 + \frac{(T_2 - T_1) + (T_4 - T_3)}{2} - T_4 = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

Network Time Protocol (NTP)

- Designed for UTC synch over ***large-scale*** networks
  - Used in practice over the Internet, on top of UDP
  - Estimate of 10-20 million NTP clients and servers
  - Widely available on all OS
  - ~1ms over LANs, 1-50ms over the Internet
  - More info at **`www.ntp.org`**
- Hierarchical *synchronization subnet* organized in *strata*
  - Servers in stratum 1 are directly connected to a UTC source
  - Lower strata (higher levels) provide more accurate information
  - Connections and strata membership change over time
  - Servers in a given stratum synchronize among themselves
- Synchronization mechanisms
  - Multicast (over LAN)
  - Procedure-call mode (server-based)
  - Symmetric mode (for higher levels)



NTP: Symmetric Mode



- Similar to server-based, but the two nodes involved continuously exchange their roles, in a continuous message "ping-pong"
- The offset estimate is complemented by a delay estimate δ

$$\delta = \frac{(T_4 - T_1) - (T_3 - T_2)}{2}$$

- The pairs $<\theta, \delta>$ are stored and processed through a statistical filter, to obtain a measure of reliability
- Each node communicates with several others
  - Only the most reliable data is returned to applications
  - Allows to select the best primary data source

Some Observations

- In many applications it is sufficient to agree on a time, even if it is not accurate w.r.t. the absolute time
- What matters is often the ***ordering*** and ***causality*** relationships of events, rather than the timestamp itself
  - i.e., the fact that one event occurred after another, and that it may have been "caused" (or may contain information about) the previous event
- If two processes do not interact, it is not necessary that their clocks be synchronized

Modeling Distributed Executions

- A ***distributed algorithm*** is a collection of distributed automata, one per process
- The ***execution*** of a distributed algorithm (on a process) is a sequence of ***events*** executed by the process
- In distributed algorithms, relevant ***events*** are:
  - ***send*** event: the process sends a message to another process, *send(m, p)*
  - ***receive*** event: the process receives a message, *receive(m)*
  - ***local*** event: anything that changes the application state of the process
    - (e.g., setting a variable, writing a file)

Histories

- The ***local history*** of a process p_i is a (possibly infinite) sequence of events

$$h_i = e_i^0 e_i^1 e_i^2 \ldots e_i^{m_i}$$

- The ***partial history*** up to event e_i^k is denoted h_i^k and is the prefix of the first k events in h_i
- Histories are useful to formalize distributed algorithms…
- … but they do not specify any relative timing/ordering between events in different processes…
- ... that would help us determine whether they occurred one after the other, or concurrently

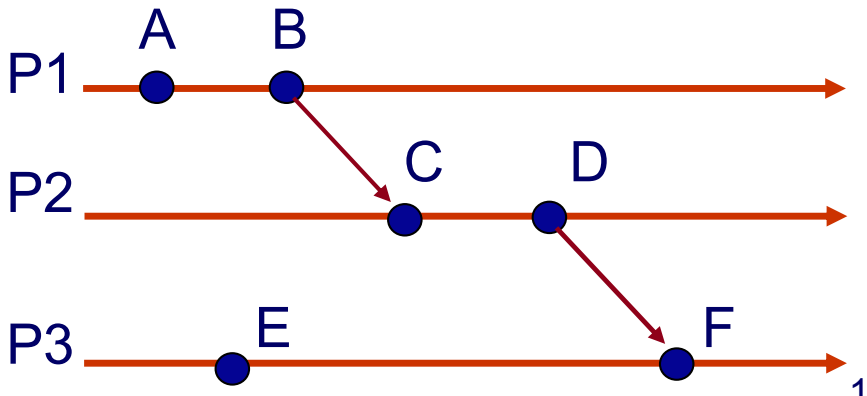
Happens-Before

Definition (Happen-before)

We say that an event e happens-before an event e', and write $e \rightarrow e'$, if one of the following three cases is true:

1. $\exists p_i \in \Pi : e = e_i^r, \quad e' = e_i^s, \quad r < s$
   (e and e' are executed by the same process, e before e')
2. $e = send(m, *) \wedge e' = receive(m)$
   (e is the send event of a message m and e' is the corresponding receive event)
3. $\exists e'' : e \rightarrow e'' \rightarrow e')$
   (in other words, $\rightarrow$ is transitive)

Two events e, e' for which happens-before does not hold $(e \not\rightarrow e' \wedge e' \not\rightarrow e)$ are called ***concurrent***, $e||e'$



What Does It Mean?

If $e \rightarrow e'$, this means that we can find a series of events $e^1 e^2 e^3 \ldots e^n$, where $e^1 = e$ and $e^n = e'$, such that for each pair of consecutive events e^i and e^{i+1}:

1. e^i and e^{i+1} are executed on the same process, in this order, or
2. $e^i = send(m, *)$ and $e^{i+1} = receive(m)$

- The happens-before relationship captures:
  - a flow of data between two events
  - the notion of (partial) ***causal ordering*** of events
- It is a key concept in concurrent systems
  - E.g., the multi-threaded memory model of Java is defined based on happens-before, where "events" are read/write of variables and acquisition/release of locks
    `http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html`

Logical Clocks

- Enable coordination among processes without synchronization of physical clocks
  - which can be difficult and/or costly to achieve
- Every process has a logical clock
  - essentially a counter, which is incremented using a well-defined set of rules…
  - … and whose value has no relationship with a physical clock whatsoever
- Distributed algorithms exploit logical clocks to timestamp events
  - The causality relation between events is then inferred from their timestamps
- Different notions of logical clocks exist, with different rules and expressiveness

Logical Clocks: Definitions

Definition (Logical clock)

A logical clock LC is a function that maps an event e from the history H of a distributed system execution to an element of a time domain T:

$$LC : H \rightarrow T$$

This mapping enables timestamping of events, if the following holds…

Definition (Clock Consistency)

$$e \rightarrow e' \Rightarrow LC(e) < LC(e')$$

… which states that if two events happen one after the other, their timestamps respect time monotonicity

Definition (Strong Clock Consistency)

$$e \rightarrow e' \Leftrightarrow LC(e) < LC(e')$$

If this stronger property holds, we can also reconstruct the ordering of events given the clocks

Scalar Clocks

L. Lamport. "Time, clocks, and the ordering of events in a distributed system". *Communications of the ACM*, 21(7):558-565, July 1978.

- The question then becomes: how to assign logical clocks in a way that guarantees clock consistency?

Definition (Scalar logical clocks)

- A Lamport's scalar logical clock is a monotonically increasing software counter
- Each process p_i keeps its own logical clock LC_i
- The timestamp of event e executed by process p_i is denoted $LC_i(e)$
- Messages carry the timestamp of their *send* event
- Logical clocks are initialized to 0

guarantees clock consistency by design

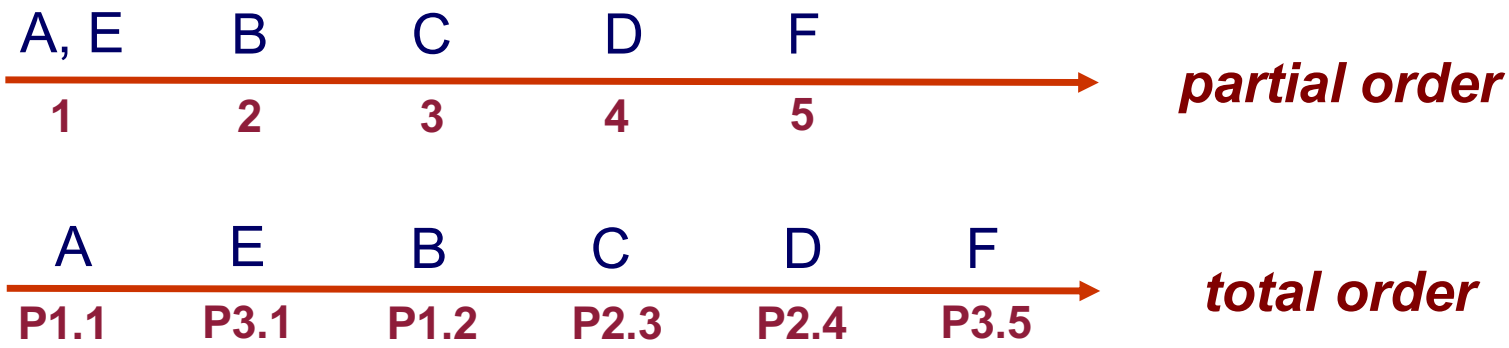
Update rule

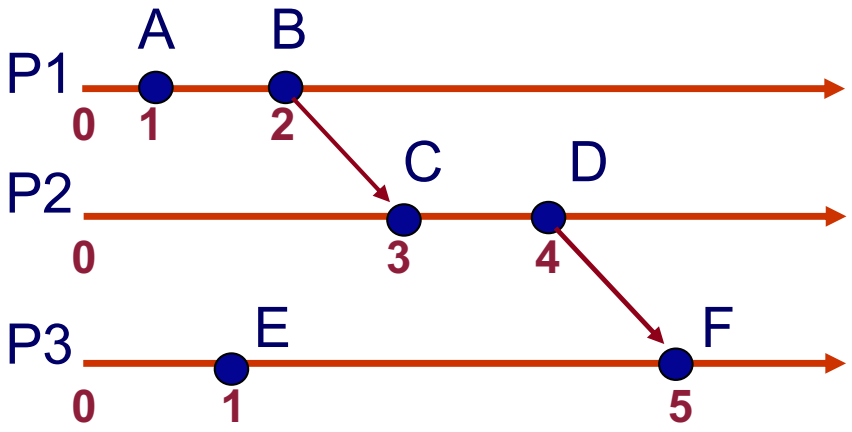
Whenever an event e is executed by process p_i, its local logical clock is updated as follows:

$$LC_i = \begin{cases} LC_i + 1 & \text{If } e_i \text{ is an internal or } send \text{ event} \\ \max\{LC_i, TS(m)\} + 1 & \text{If } e_i = receive(m) \end{cases}$$

Partial vs. Total Order

- The previous strategy achieves only partial ordering
- ***Total ordering*** can be obtained trivially by attaching process IDs to clocks
  - Assuming an ordering (e.g., lexicographic) holds among the IDs





Communication from UNITN

Dear all,

in the hope that you have read the UNITN communication sent on February 28, I remind you that those who are in one of these situations:

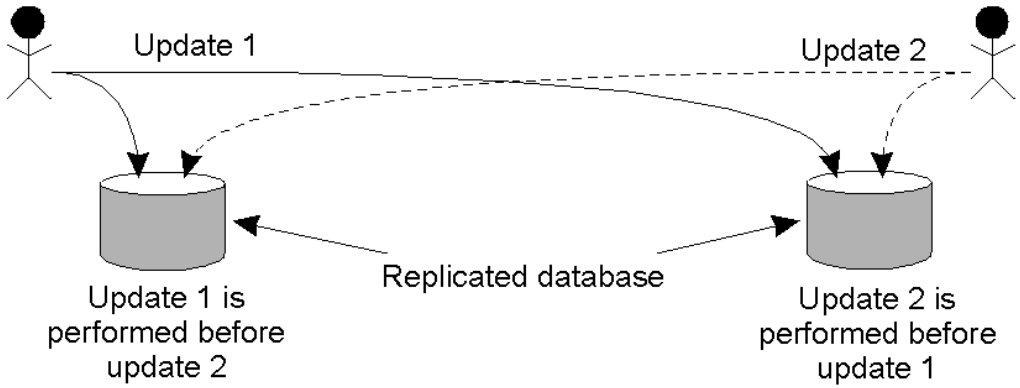
- have flu-like symptoms like fever, cough, shortness of breath
- have been in contact with someone who has been found positive of Covid-19
- have stayed in the last 15 days (excluding transit) in one of the Italian cities
- subject to fiduciary isolation (DPCM February 23, 2020): Bertonico,
- Casalpusterlengo, Castelgerundo, Castiglione d'Adda, Codogno, Fombio, Maleo, San Fiorano, Somaglia , Terranova dei Passerini, Vo' Euganeo
- have travelled from China in the last 15 days

should leave the classroom and contact the number **800867388**

Schedule

Date	#hrs	Type	Topic	Instructor
17 Feb, Mon	2	lecture	Course intro, administrivia	Picco
18 Feb, Tue	2	lecture	Synchronization: time synchronization, logical clocks	Picco
24 Feb, Mon			**NO CLASS (university closed)**	
25 Feb, Tue			**NO CLASS (university closed)**	
2 Mar, Mon	2	lecture	Synchronization: logical clocks, vector clocks	Picco
3 Mar, Tue	2	lecture	Synchronization: vector clocks & exercises	Picco
9 Mar, Mon	2	lab	Akka: introduction and simple examples	Vecchia
10 Mar, Tue	2	lecture	Synchronization: global snapshot & exercises	Picco
16 Mar, Mon			**NO CLASS (Lauree)**	
17 Mar, Tue			**NO CLASS (Lauree)**	

Example:Totally Ordered Multicast

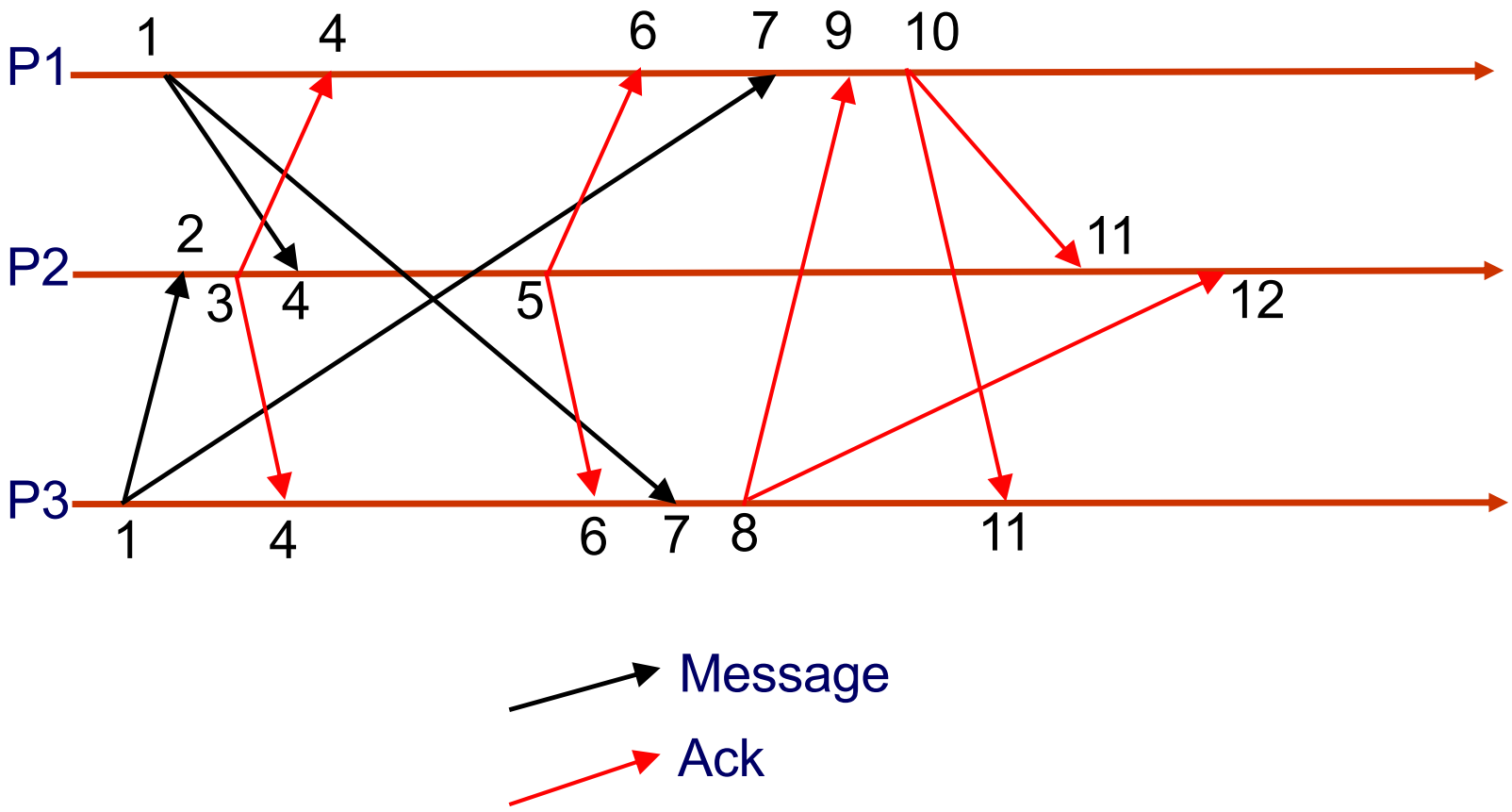


- Updates in sequence:
  - Customer deposits $100
  - bank adds 1% interest
- Updates are propagated to all locations:
  - If updates arrive in the same order at each copy, consistent result: the same value (e.g., $1110 or $1111) at both replicas
  - If updates arrive in a different order, inconsistent result: one replica holds $1110, the other $1111
- To solve the problem we need to know neither when the updates occur, nor in which order w.r.t. physical time
  - we need to make sure that everybody agrees ***on the same order***
- ***Totally ordered multicast*** delivers messages in the same global order
  - Unlike standard multicast, where packets can be arbitrarily reordered

Totally Ordered Multicast with Scalar Clocks

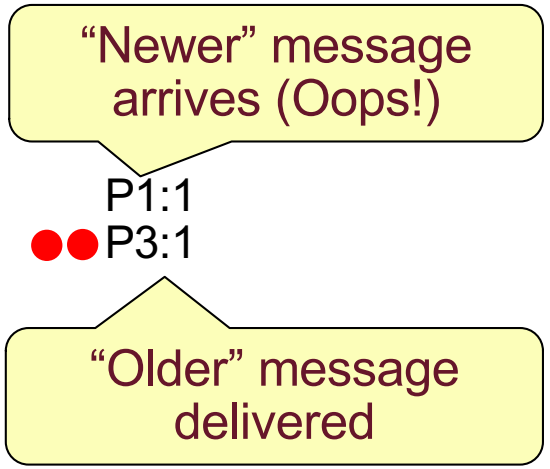
- Assumptions: ***reliable*** nodes/links and ***FIFO*** links
  - TCP fits the bill
- Algorithm outline:
  - Messages are sent (and acknowledged) simultaneously to all nodes
  - All messages (including acknowledgments) carry a timestamp with the sender's (scalar) clock
  - Receivers store all messages in a queue, ordered according to its (logical) timestamp
  - A message is delivered to the application only when it is the highest in the queue and all of its acknowledgments have been received

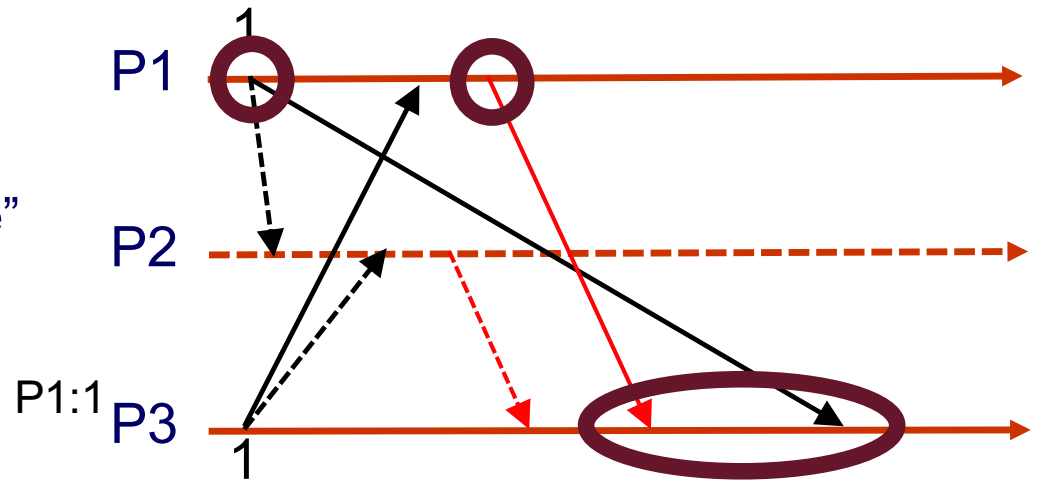
Example



Does It Really Work?

- The algorithm works because, ***eventually***, all processes see the same messages in the same order in their queue
  - … but why?
  - Logical clocks effectively provide a consistent timestamping across all nodes
- ***Question:*** Can it happen, at some node, that an "older" message is delivered and, later, a "newer" message shows up?
  - this would break the assumption above, and break also the total ordering
- ***Answer:*** It is not possible because of the FIFO assumption and the algorithm operation
  - The sender of the "late" message must ack the "older" message





What If …

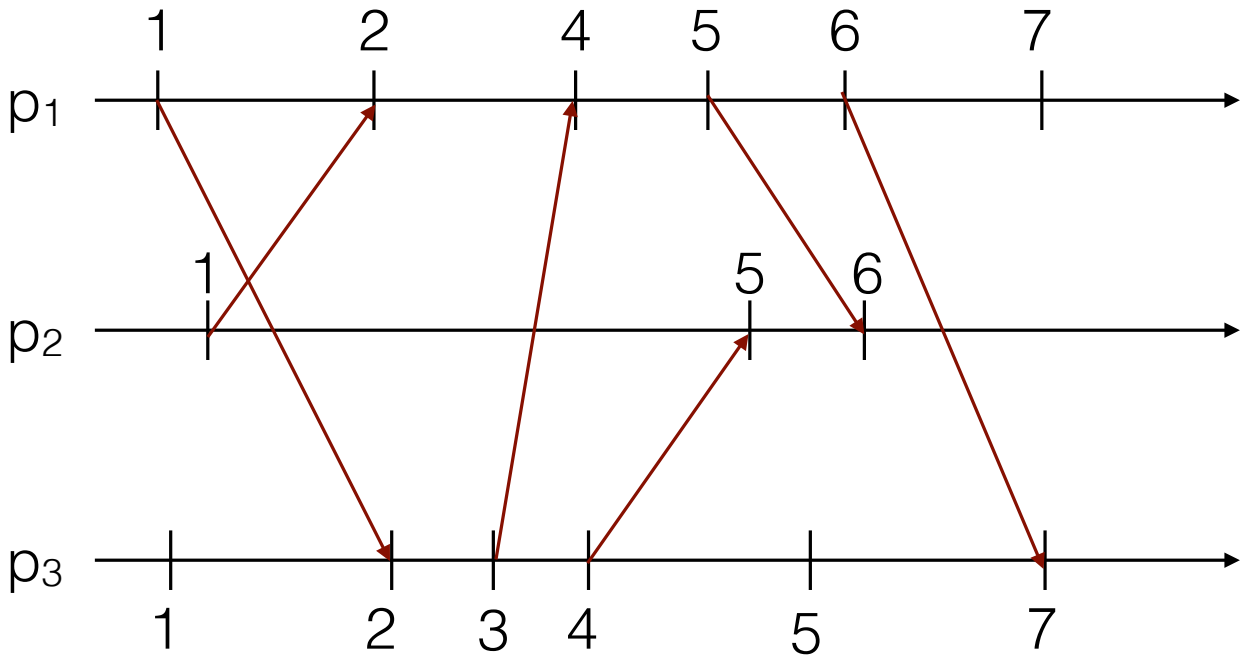
- ***… we remove the assumption about reliable nodes/links?***
- Message reception would not be guaranteed: a node could wait for a message (or ack) forever
- ***… we use logical clocks with partial ordering instead of total ordering?***
  - i.e., no disambiguation with process IDs?
- How would we deal with messages with the same logical clock?
  - e.g., those originating at P1 and P3
  - If we just inserted them in the local queue, P1 and P3 would each insert their own message first, resulting in different queue ordering, and therefore breaking the total order requirement

On Strong Clock Consistency

Theorem

Scalar logical clocks do not satisfy Strong clock consistency, i.e.

$$LC(e) < LC(e') \not\Rightarrow e \rightarrow e'$$



- Scalar clocks do not capture all possible ***causality*** relationships

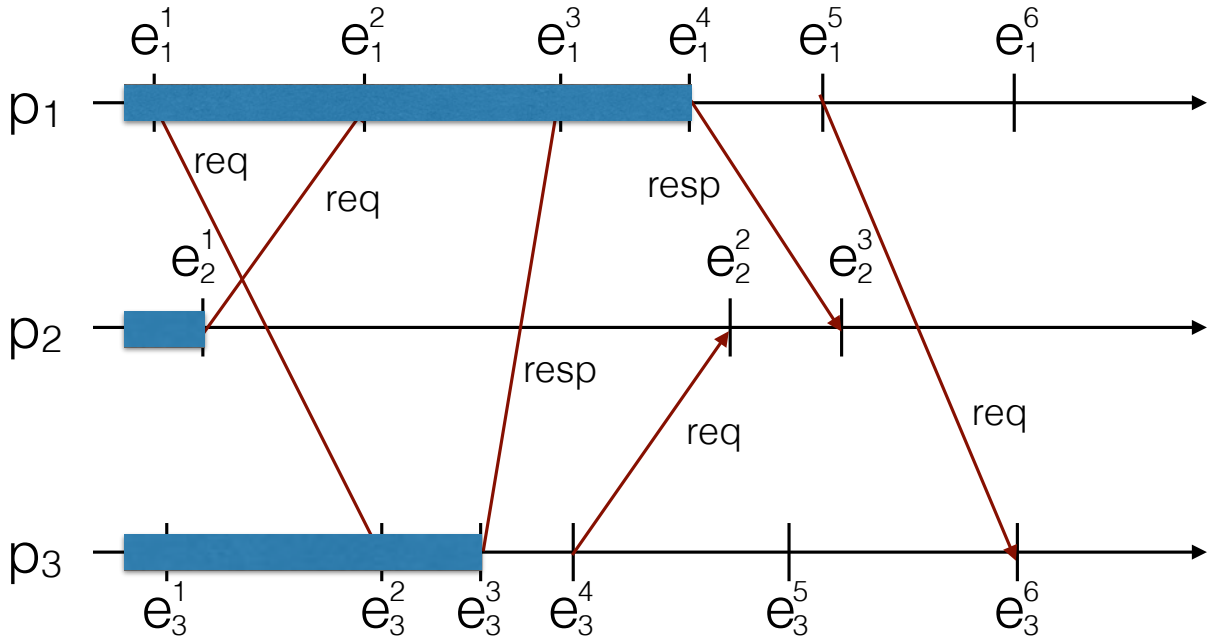
Causal Histories & Clocks

Definition (Causal History)

The causal history of an event e is the set of events that happen-before e, plus e itself.

$$\theta(e) = \{e' \in H \mid e' \rightarrow e\} \cup \{e\}$$

- In principle, the timestamp of an event could be represented by its causal history



In practice, they grow too much. But do we really need the entire causal history?

Vector Clocks

if *n* is the number of processes

Definition

The vector clock associated to event e is a n-dimensional vector $VC(e)$ such that

$$VC(e)[i] = c_i \qquad \text{where } \theta_i(e) = h_i^{c_i}$$

$\theta_i(e) = \theta(e) \cap h_i = h_i^{c_i}$ is the causal history projection over process p_i

Remember that h_i^k is the partial history of event e_i^k

- In practice, *VC(e)* contains the "most recent clock" of all processes that are in a causality relationship with event *e*

Vector Clocks: Implementation

- Each process p_i maintains a vector VC_i of n elements:
  - $VC_i[i]$ is the number of events that have occurred at p_i
  - If $VC_i[j]=k$, p_i knows that k events have occurred at p_j
- Initially, $VC_i[j]=0$ for all i,j
- A message m carries the timestamp $TS(m)$ of its *send* event (as in scalar clocks, but here is a vector)

Update rule

When event e_i is executed by process p_i, VC_i is updated as follows:

- If e_i is an internal or *send* event:

$$VC_i[i] = VC_i[i] + 1$$

- If $e_i = receive(m)$:

$$VC_i[j] = \max\{VC_i[j], TS(m)[j]\} \quad \forall j \neq i$$
$$VC_i[i] = VC_i[i] + 1$$

Vector Clocks: Properties

"Less than" relation for Vector clocks

$$VC < VC' \Leftrightarrow (VC \neq VC') \wedge (\forall k : 1 \leq k \leq n : VC[k] \leq VC'[k])$$

Strong Clock Condition

$$e \rightarrow e' \Leftrightarrow VC(e) < VC(e') \Leftrightarrow \theta(e) \subset \theta(e')$$

"Concurrent" clocks

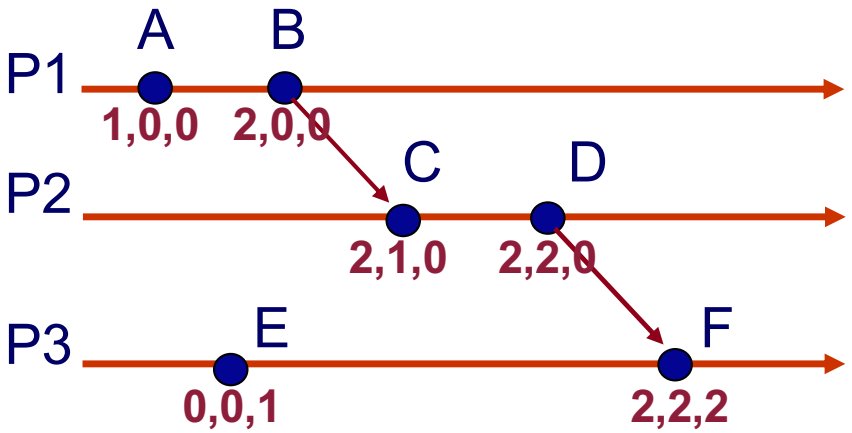
$$VC \,||\, VC' \Leftrightarrow \neg(VC < VC') \wedge \neg(VC > VC')$$

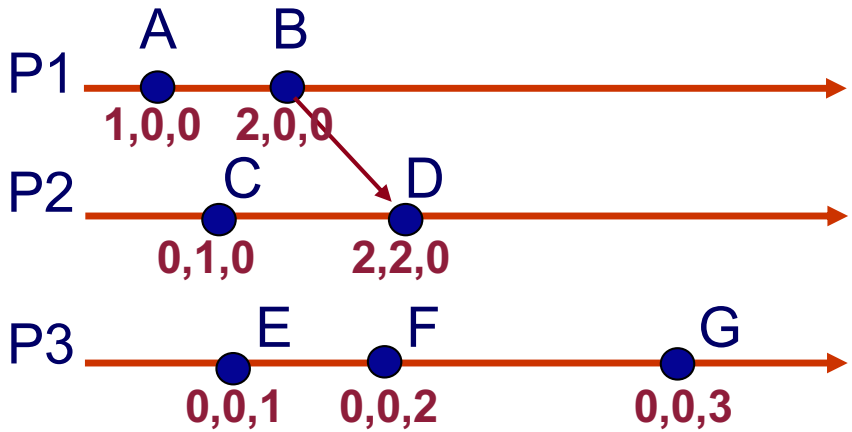
Determining causality

$$e_i \rightarrow e_j \Leftrightarrow VC(e_i) < VC(e_j)$$

$$e_i || e_j \Leftrightarrow VC(e_i) \,||\, VC(e_j)$$

Examples

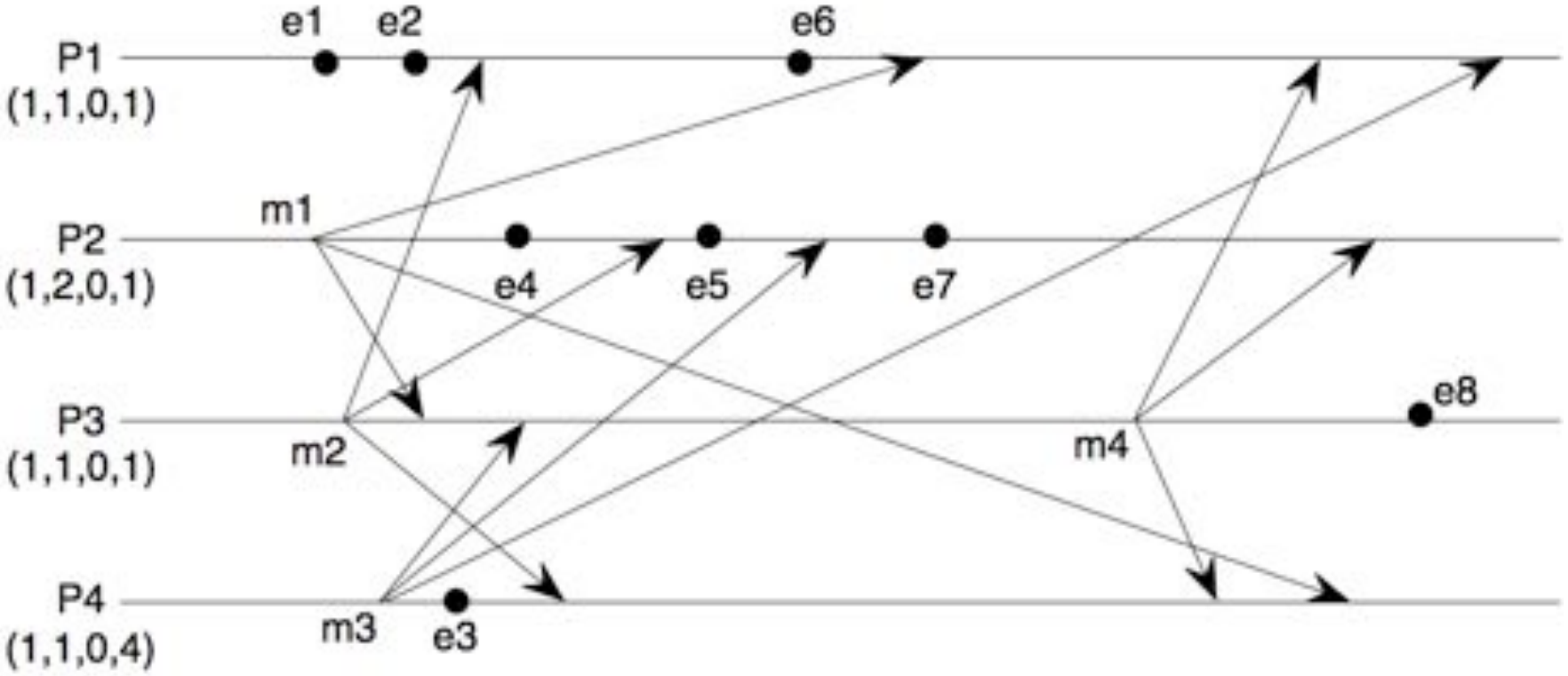




- By looking ***only*** at the timestamps we are able to determine whether two events are causally related or concurrent
- Can be exploited to implement causal message delivery

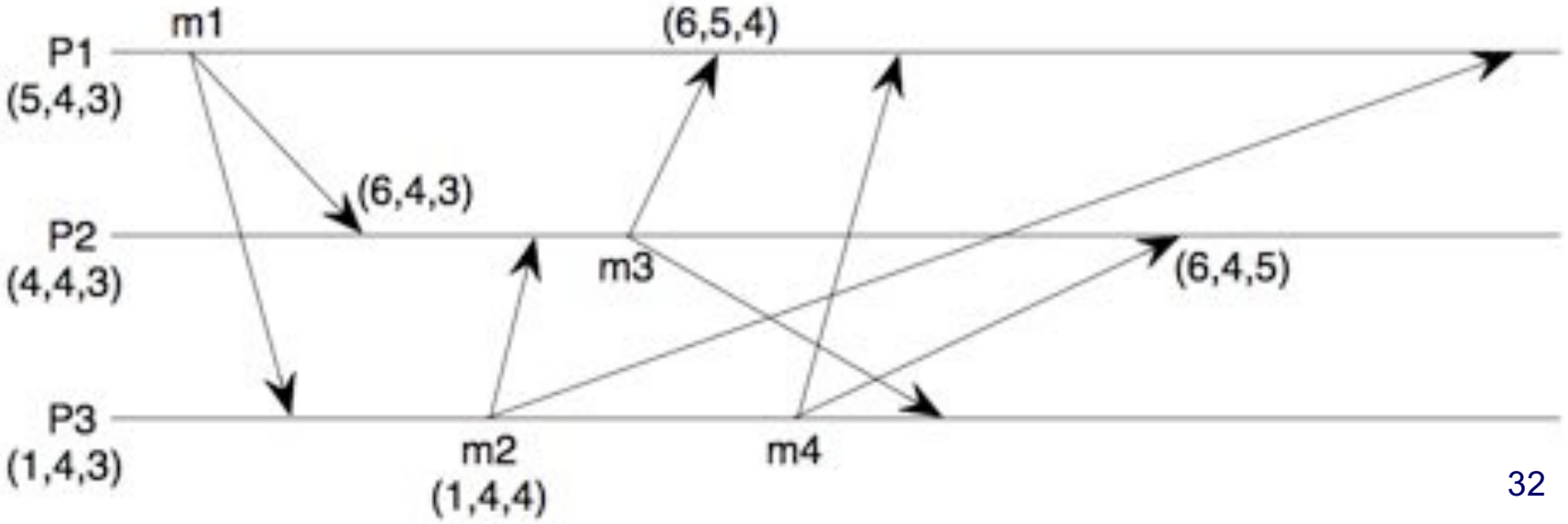
Exercise

- The following diagram represents the exchange of messages m_i among processes P_i, and the generation of local events e_i
- Show the values of the vector clocks for each process at the end of the interaction, assuming the initial clock values shown. Assume that the local clock can be advanced (by a time unit) by an event e_i local to a process, as well as the sending or receiving of a message.

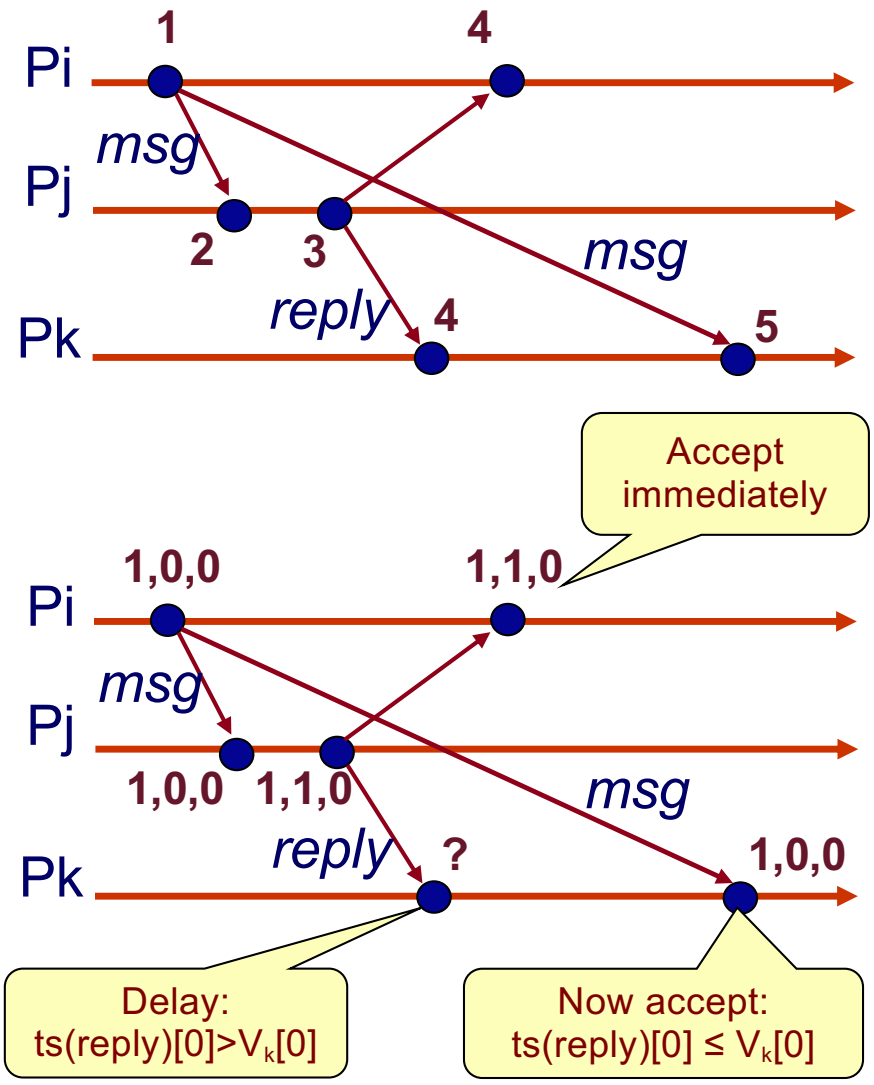


Exercise

- The message diagram below represents the communication taking place between three processes, using vector clocks for maintaining shared logical time. Time advances at a process only due to a local event or message sending, i.e., message receiving is not considered a relevant event for logical time. Messages are always sent in multicast, and links are FIFO. At the extreme left, the initial vector clock is shown for each process. The current value of the vector clock for some events (message sending or receiving) is shown in the diagram for some of the processes.
- The diagram contains at least three errors with respect to the notion of vector clock and/or the assumptions above. Identify and explain these errors



Example: Bulletin Boards



- Must preserve the ordering ***only*** between messages and replies
  - Totally ordered multicast is too strong
    - If M1 arrives before M2, it does not necessarily mean that the two are related
- A variation of vector clocks enables ***causal delivery*** in a fully distributed way
  - Variation: increment clock only when sending a message; on receive only merge, no increment; no other local events
  - Hold a message until the previous messages in its causality chain are received; if j is the "reply" source:
    - $ts(reply)[j] = V_k[j]+1$
    - $ts(reply)[i] \leq V_k[i]$ for all $i \neq j$
  - FIFO links, reliable nodes/links

Global States

Definition (Local state)

- The local state of process p_i after the execution of event e_i^k is denoted σ_i^k
- The local state contains all data items accessible by that process
- Local state is completely private to the process
- σ_i^0 is the initial state of process p_i

Definition (Global state)

The global state of a distributed computation is an n-tuple of local states $\Sigma = (\sigma_1, \ldots, \sigma_n)$, one for each process.

- Determining the global state is important, e.g., for distributed debugging, property checking, etc.
- It is complicated by the asynchronicity of messages, which determine a (distributed) state change

Cut

Definition (Cut)

A cut of a distributed computation is the union of n partial histories, one for each process:

$$C = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_n^{c_n}$$

- A cut can be described by a tuple $(c_1, c_2, \ldots, c_n)$ containing a set of events from each process (i.e., the ***frontier*** of the cut)
- Each cut has a corresponding global state $(\sigma_1^{c_1}, \sigma_2^{c_2}, \ldots, \sigma_n^{c_n})$
- To be useful, a cut must be ***consistent***, i.e., represent a global state that may have happened in reality

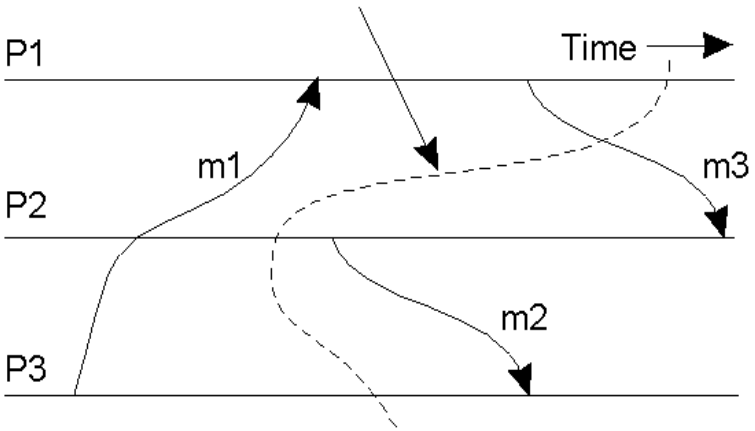
Consistent Cut

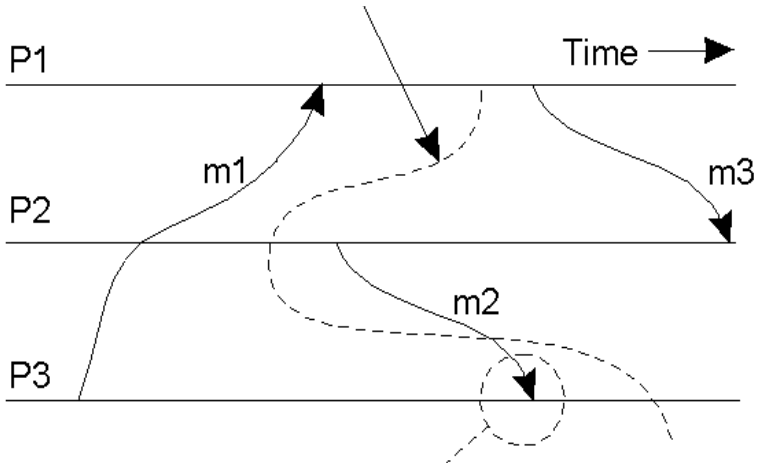
Definition (Consistent cut)

A cut C is consistent, if for all events e and e',

$$(e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in C$$

- The cut defines a (logical) “past” and “future”; a message received “in the past” cannot be sent “from the future”





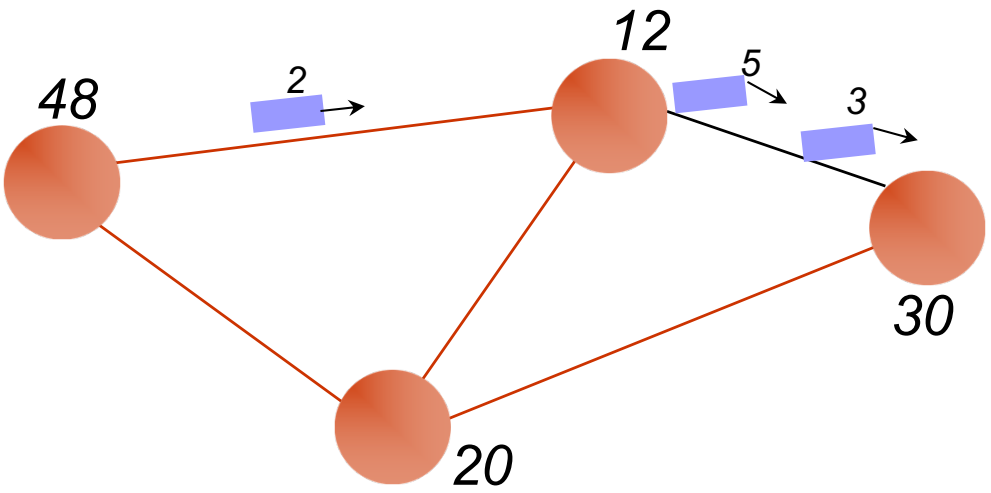
Distributed Snapshots

- A ***distributed snapshot*** reflects a consistent, global state in which the distributed system *might have been*
- Particular care must be taken when reconstructing the global state, to preserve consistency
  - No messages to jump from the future into the past
  - If a message receipt is recorded, the message send must as well
- In other words, we need/want to reconstruct a consistent cut…
- … considering that the global state consists of the local state of each process, together with the messages in transit over the links

A Banking Example

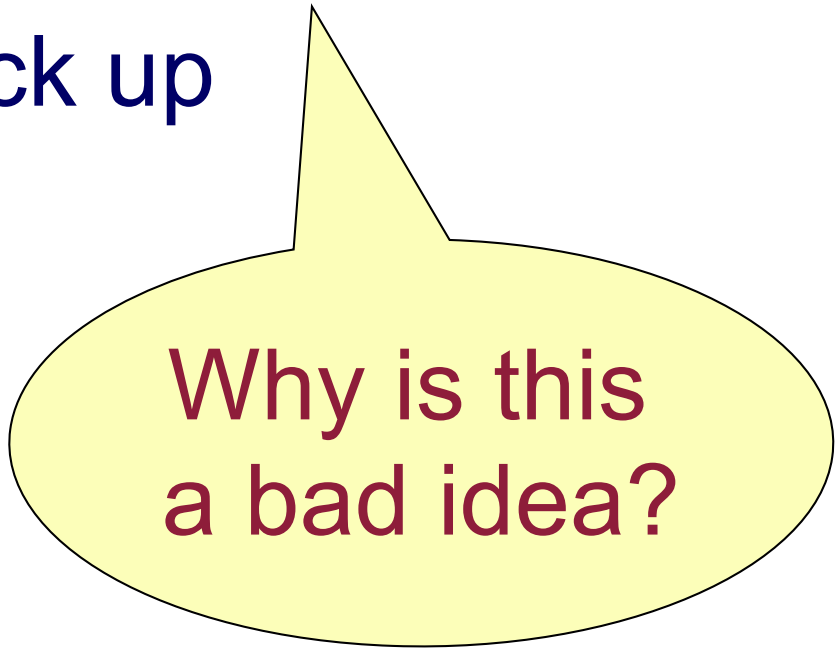
- Constant amount of money in the bank system
- Money is transferred among banks in messages
- Problem: find out of any money accidentally lost
- ***Not*** the problem: determine ***where*** the money is located

The problem is easy to solve with a global clock, but we cannot rely on it:
we must deal with recordings potentially occurring at different times

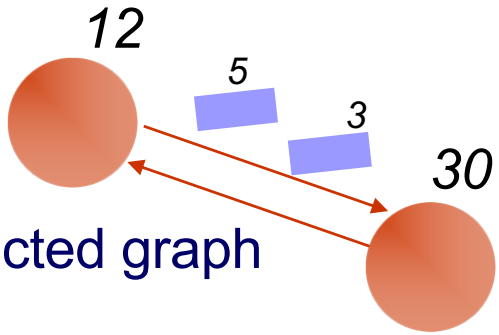


Simplistic Algorithm

- Stop all processes
- Let all channels empty (not trivial)
- Record the state
- Start everything back up



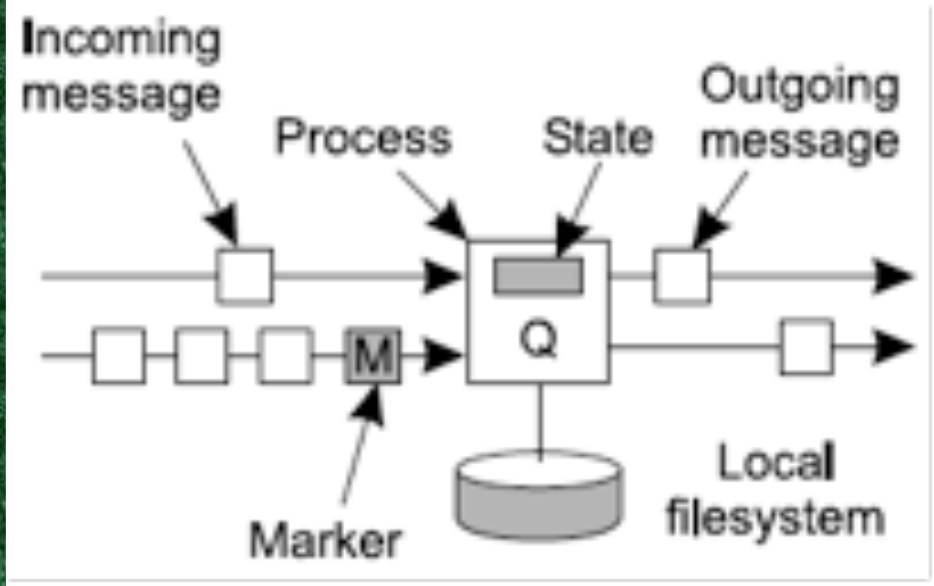
Distributed Snapshot
Chandy-Lamport, 1985



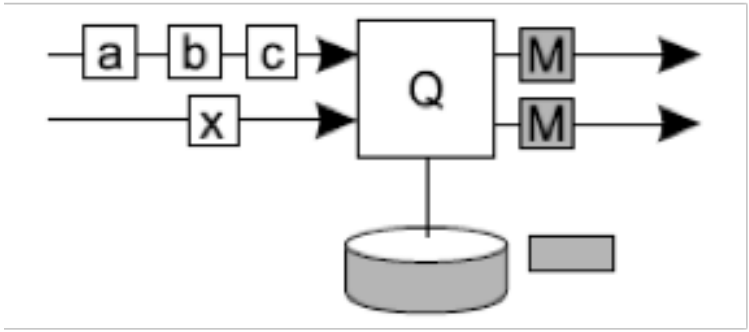
- Assume FIFO, reliable links/nodes, strongly connected graph
  - i.e., a path exists between any two nodes of the graph
- Any process can initiate a snapshot by
  - Recording internal state
  - Sending a token on all outgoing channels
  - Start recording local snapshot
    - Record messages arriving on every incoming channel
- Upon receiving a token:
  - If not already recording local snapshot
    - Record internal state
    - Send token on all outgoing channels
    - Start recording local snapshot
  - In any case stop recording incoming message on channel the token arrived along
- Recording messages
  - If a message arrives on a channel which is recording messages, record the arrival of the message, then process the message as normal
  - Otherwise, just process the message as normal
- Snapshot complete when token has arrived on all incoming channels
  - Irrelevant how the various data are collected and/or transmitted

performed as a single, indivisible action

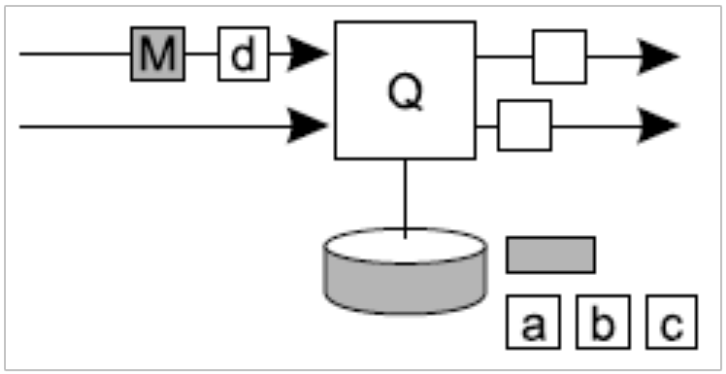
Processing



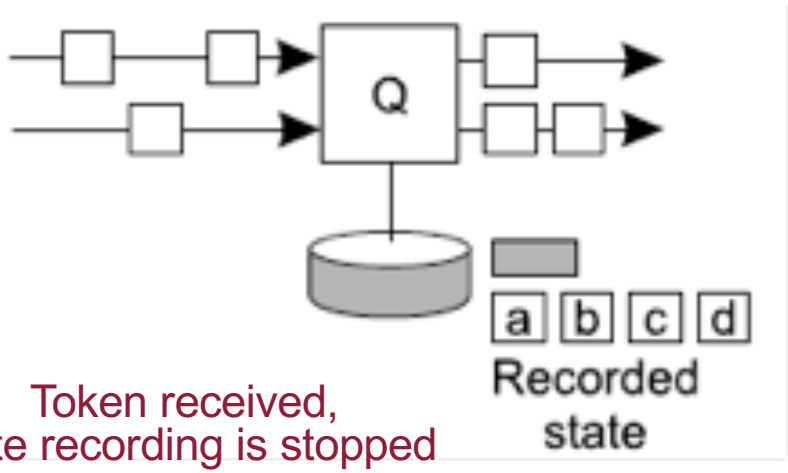
Normal processing, first marker about to be received



Token just received for the first time, state saved, token forwarded to outgoing links, begin recording messages

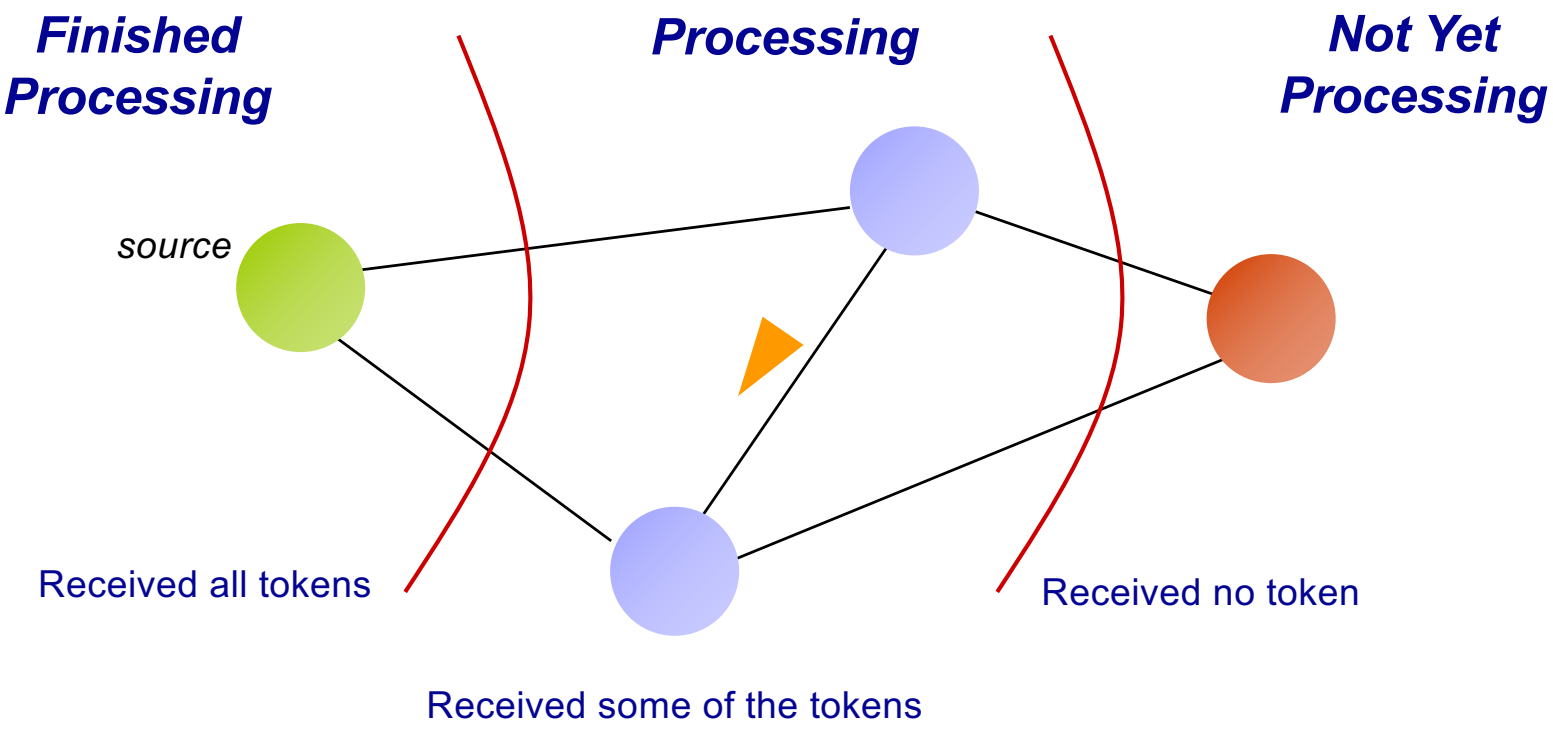


Recording of messages from the incoming links from which a token has been received

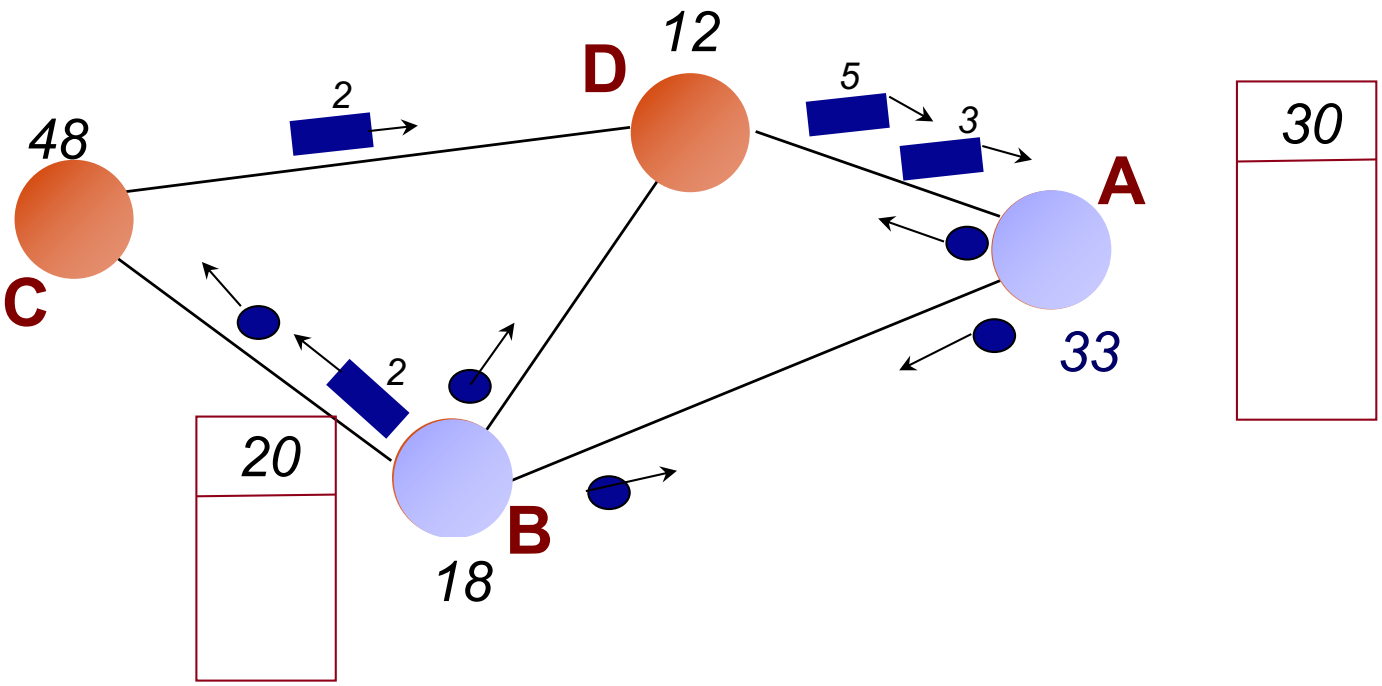


Token received, state recording is stopped

Distributed Snapshot: Intuition

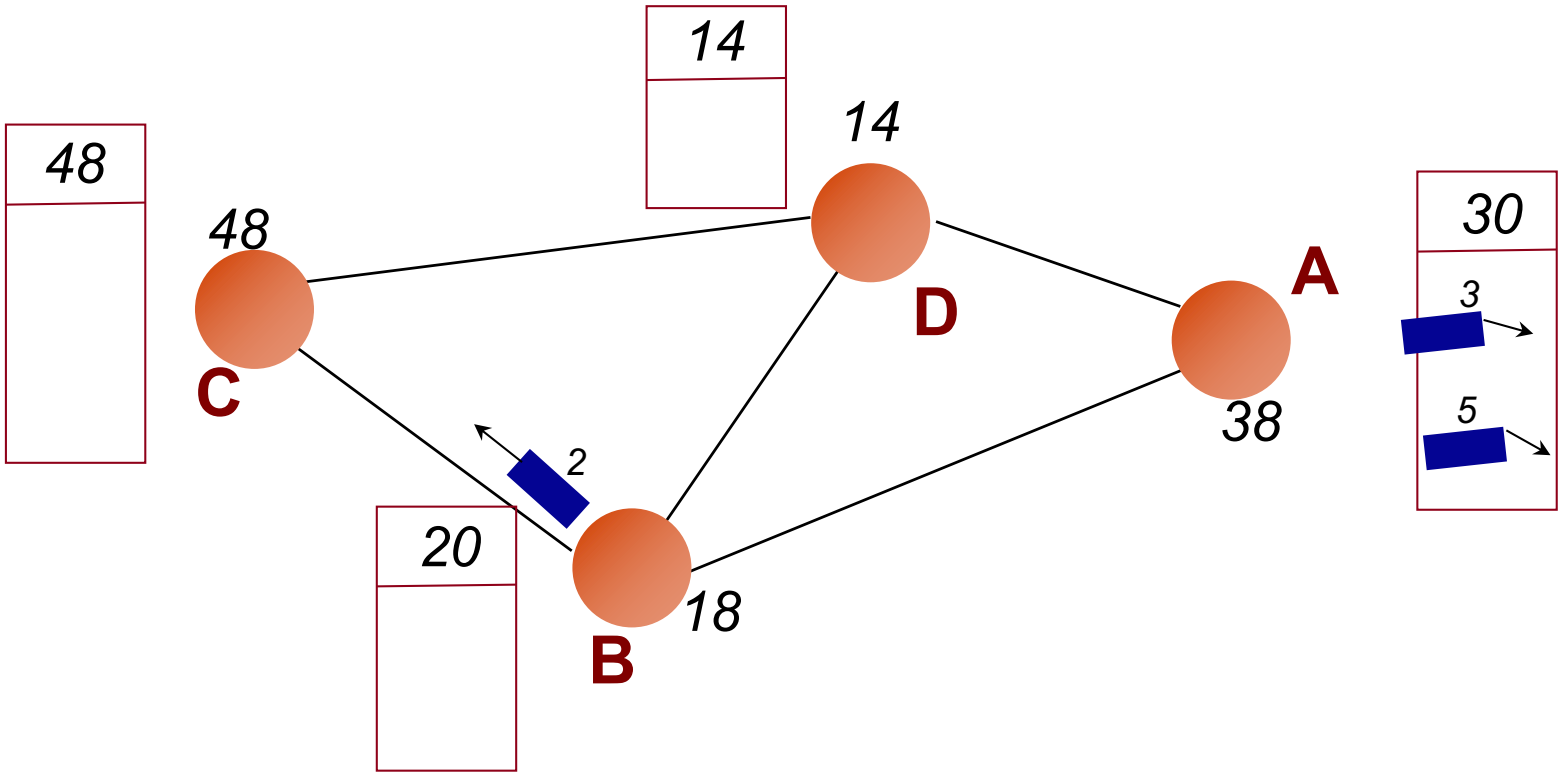


Example

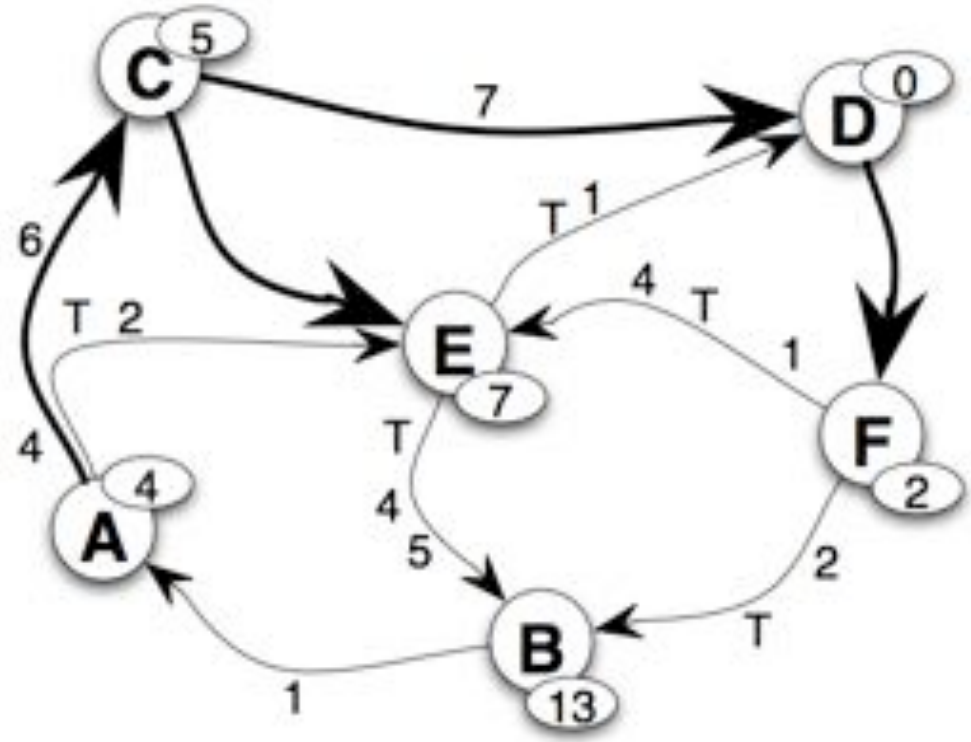


- Assume bi-directional links (although not necessary)
- Let node A start the processing

Example



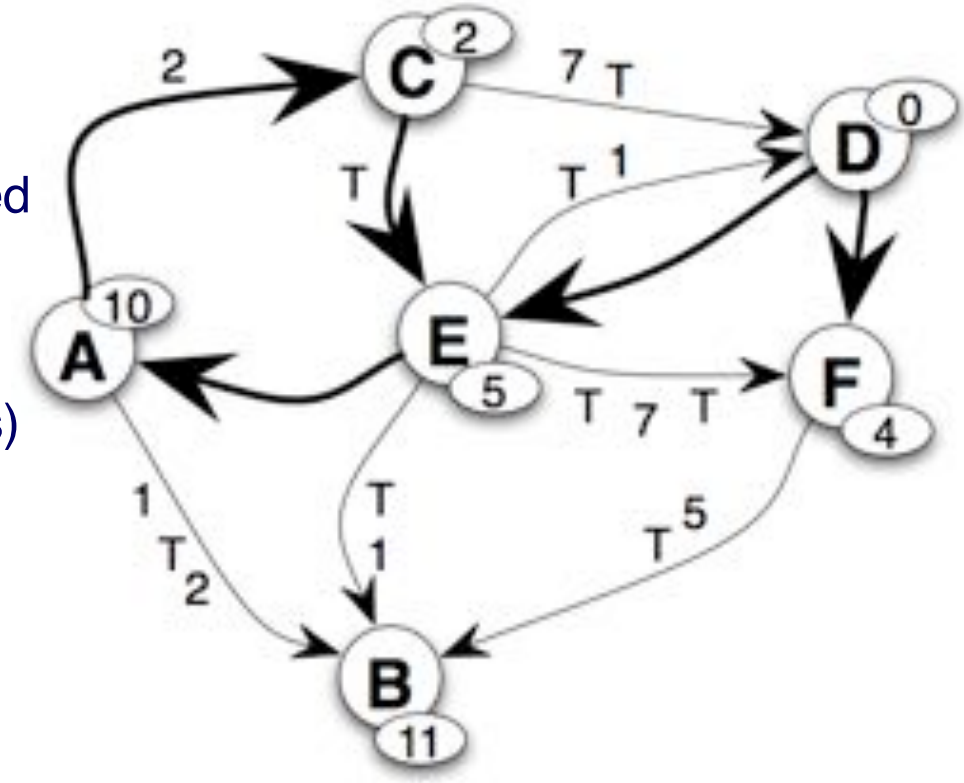
Exercise



- The diagram represents a distributed snapshot. Node A is the initiator of the snapshot; the markers already propagated along the thick lines, and the nodes A, C, D, E, F already recorded their local state. None of these nodes has recorded messages from their incoming links. The values in the oval attached to a node denote its current state. The labels on the arches denote the value of the messages exchanged, with the label T denoting the marker. Links are directional, with the direction shown by the arrows
- Assuming that no other message exchange takes place besides those in the figure, show the local state (node and links) recorded by the distributed snapshot at each node

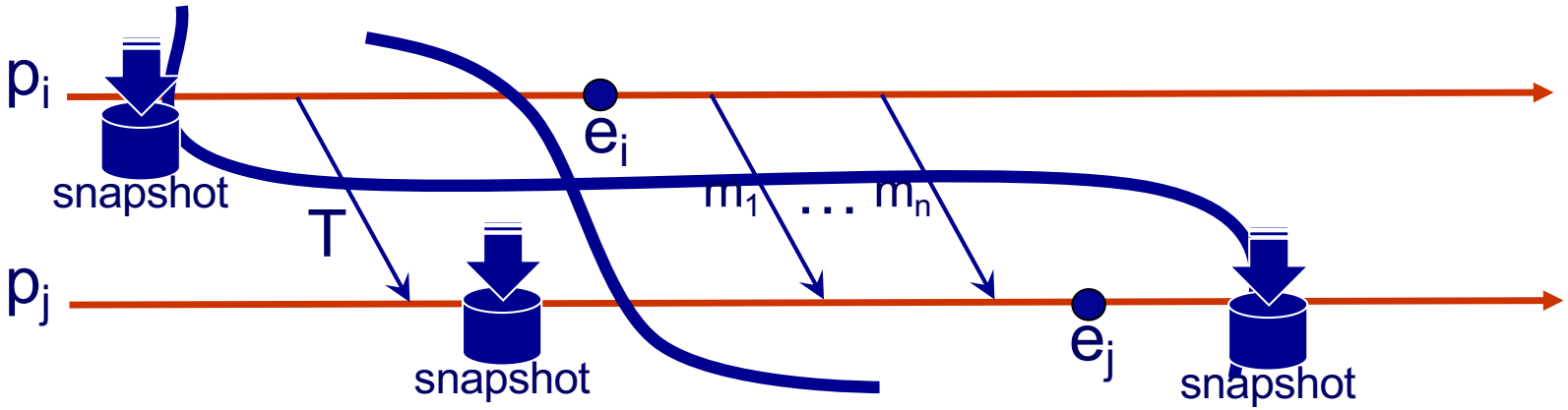
Exercise

- The diagram represents a distributed snapshot. Links are unidirectional. Node A is the snapshot initiator. The marker already propagated through the thick lines. The value in the oval attached to a node represents the current value of the node's state. The arc labels denote the value associated to messages, or T for a marker.
- Assuming there are no other messages exchanges besides those shown in the figure:
  - there are at least three "errors" in the figure, i.e., details incompatible with the distributed snapshot algorithm. Identify and describe them.
  - Show and explain all the possible states (node and links) that can be recorded by B as part of the snapshot



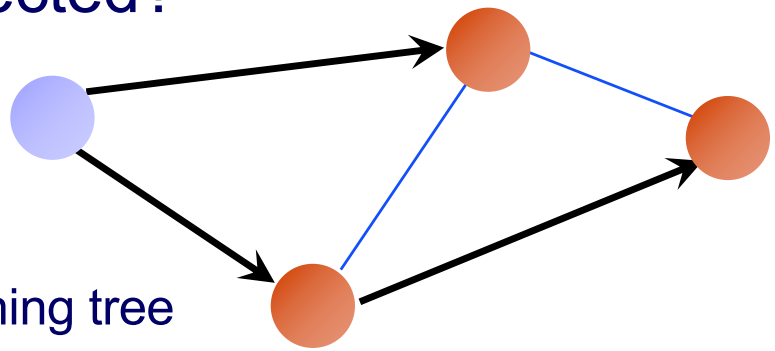
Some Observations—1

- Distributed snapshot effectively selects a consistent cut from the history of execution
- Proof:
  - Consider e_i and e_j events at p_i and p_j, so that $e_i \rightarrow e_j$
  - For the cut to be consistent, if e_j is in the cut, so does e_i
    - In other words, if e_j occurred before p_j recorded its state, the same must have happened for e_i and p_i
    - Obvious if $p_i = p_j$
  - Assume that e_i and e_j are in the cut, although p_i recorded its state before e_i
  - Consider the messages $m_1 \ldots m_n$ that establish $e_i \rightarrow e_j$
  - FIFO and marker rules are such that the marker would have reached p_j before e_j
    - If e_i occurred after p_i saved its state, then p_i sent a marker ahead of $m_1 \ldots m_n$; state recording occurs at p_j before processing of $m_1 \ldots m_n$
  - Therefore, e_j cannot be in the cut



Some Observations—2

- Important: the distributed snapshot algorithm does not require blocking of the computation
  - Collecting the snapshot is interleaved with processing
- What happens if the snapshot it started at more than one location at the same time?
  - Easily dealt with by associating an identifier to each snapshot, set by the initiator
- Several variations have been devised
  - E.g., incremental snapshots take an initial snapshot, each node remembers where it has sent/received messages, and when a new snapshot is requested, only these links are included in the result
- How is the snapshot result collected?
  - Again, several variations
    - Not part of the global snapshot
  - Easiest: send back to the node that sent the marker
    - Marker propagation builds a spanning tree



Distributed Transactions

- Clients sometimes require that a sequence of separate requests to a server be treated as ***atomic,*** i.e.:
  - Either all of the operations complete successfully or they must have no effect ("all or nothing")
  - They are free from interference from operations performed on behalf of other clients (i.e,. they execute as if they were "alone")
- Transactions are a construct that:
  - Allows programmers to clearly identify these sequences
  - Allows the underlying system to provide guarantees on the transaction's execution

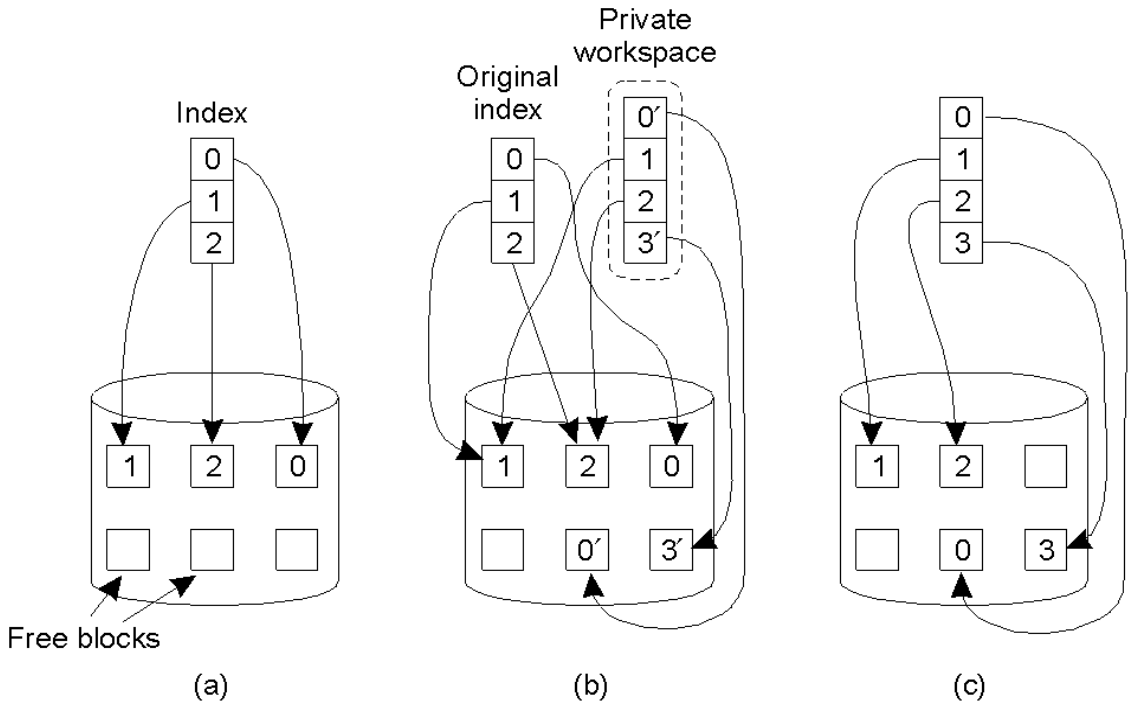
Primitive	**Description**
BEGIN_TRANSACTION	Mark the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

Transactions: Base Concepts

- "All-or-nothing" property (atomicity):
  - If a transaction ends successfully (***commit***), its results are available to the rest of the system
  - Otherwise, a ***rollback*** occurs, and no trace of the aborted transaction remains in the system
    - Originally, the rollback was literally performed by rewinding the magnetic tapes used as mass storage
- More generally, transactions have the following "ACID" properties:
  - ***Atomic***: to the outside world, the transaction happens indivisibly
  - ***Consistent***: the transaction does not violate system invariants, i.e., takes the system from one consistent state to another consistent state
  - ***Isolated*** (or ***serializable***): concurrent transactions do not interfere with each other
  - ***Durable***: once a transaction commits, the changes are permanent

Implementing Transactions—1
Commit & Rollback on a Node

- Private workspace
  - Copy what the transaction modifies into a separate memory space, creating ***shadow blocks*** of the original file
  - If the transaction is aborted, this private workspace is deleted, otherwise it becomes part of the parent's workspace
  - Optimize by replicating the index, and not the whole file
  - Works fine also for the local part of distributed transactions



Implementing Transactions—2
Commit & Rollback on a Node

- Writeahead log
  - Files are modified in place, but a log is kept with
    - Transaction that made the change
    - Which file/block
    - Old and new values
  - After the log is written successfully, the file is actually modified
  - If transaction succeeds, commit written to log; if it aborts, original state restored based on logs, starting at the end (***rollback***)

```
x = 0;                                  Log          Log          Log
y = 0;
BEGIN_TRANSACTION;
  x = x + 1;                            [x = 0/1]    [x = 0/1]    [x = 0/1]
  y = y + 2                                          [y = 0/2]    [y = 0/2]
  x = y * y;                                                      [x = 1/4]
END_TRANSACTION;
          (a)                            (b)          (c)          (d)
```

Concurrency Control

The Lost Update Problem

- Three bank accounts: A=€100, B=€200, C=€300

```
balance=b.getBalance()
b.setBalance(balance*1.1)
a.withdraw(balance/10)
```

T1

```
balance=b.getBalance()
b.setBalance(balance*1.1)
c.withdraw(balance/10)
```

T2

B=242

```
balance=b.getBalance()


b.setBalance(balance*1.1)
a.withdraw(balance/10)
```

T1

```
balance=b.getBalance()
b.setBalance(balance*1.1)


c.withdraw(balance/10)
```

T2

B=220

Concurrency Control

The Inconsistent Retrieval Problem

- Three bank accounts: A=€100, B=€200, C=€300

```
a.withdraw(100)
b.deposit(100)
```

T1

```
total = a.getBalance()
total = total + b.getBalance()
total = total + c.getBalance()
```

T2

total=600

```
a.withdraw(100)



b.deposit(100)
```

T1

```
total = a.getBalance()
total = total + b.getBalance()
total = total + c.getBalance()
```

T2

total=500

Conflicts & Serializability

Operations		Conflict?	Reason
read	read	no	The effect of two read does not depend on the order in which they are executed
read	write	yes	The effect of read and write depends on the order in which they are executed
write	write	yes	The effect of write and write depends on the order in which they are executed

- ***Serially equivalent interleaving*** (of transaction operations): one in which the the operations have the same combined effect as if the transactions had been performed one at a time, in some order
- Two transactions T_1 and T_2 are serially equivalent iff all pairs of conflicting operations of T_1 and T_2 are executed in the same order at all of the objects they both access

Example

```
x = read(i)
write(i,10)


write(j,20)
```

T1

```
y = read(j)
write(j,30)

z = read(i)
```

T2

- T_1 makes all of its accesses to `i` before T_2 does, and similarly T_2 for `j`
  - however this is not sufficient, intermediate values are visible and yield a final value impossible in a sequential order
- To be serially equivalent:
  - T_1 must access `i` and `j` before T_2, or
  - T_2 must access `i` and `j` before T_1
- Three commonly used methods to ensure serializability
  - locking, timestamp ordering, optimistic control

Dirty Reads & Cascading Rollbacks

```
balance = a.getBalance()
a.setBalance(balance+10)



abort
```

T1

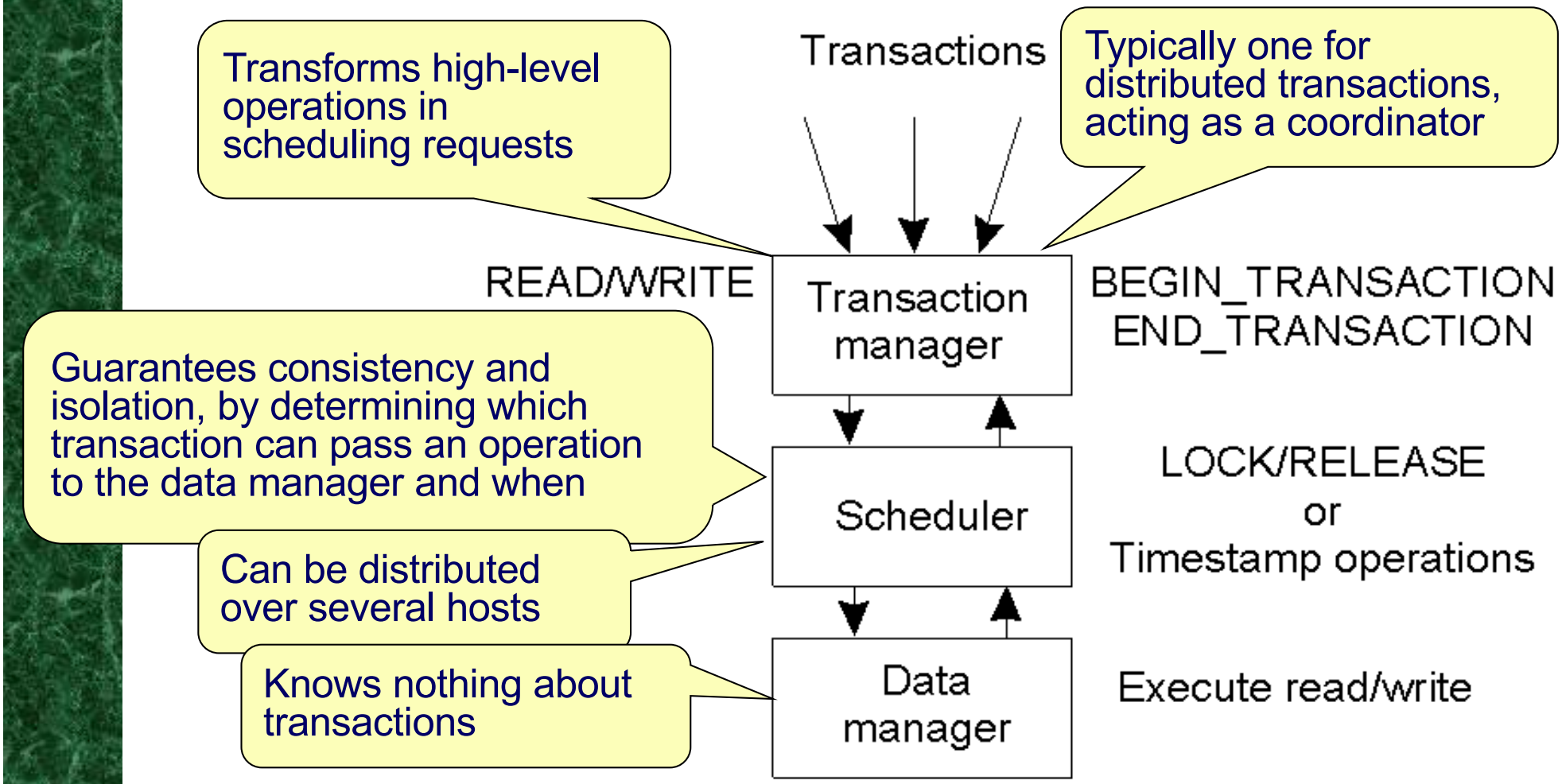
```
balance = a.getBalance()
a.setBalance(balance+20)
commit
```

T2

- If T_2 is allowed to "see" an uncommitted ("dirty") object modified by T_1, it may commit state meant to be rolled back
  - one strategy: delay the commit of T_2 until T_1 commits; if T_1 aborts, T_2 must abort as well
- However, if a T_3 reads **a** from T_2 before the latter completes, and then T_1 aborts, we have a ***cascading rollback***

Concurrency Control

- Allow several transactions to be executed simultaneously, but safely (i.e., consistently)



Using Locks

- A server (scheduler) can be used to serialize access to objects, e.g., granting ***exclusive*** locks

```
begin transaction
balance=b.getBalance()
b.setBalance(balance*1.1)
a.withdraw(balance/10)
end transaction
```

T1

b

a

```
begin transaction
balance=b.getBalance()


b.setBalance(balance*1.1)
c.withdraw(balance/10)
end transaction
```

T2

b

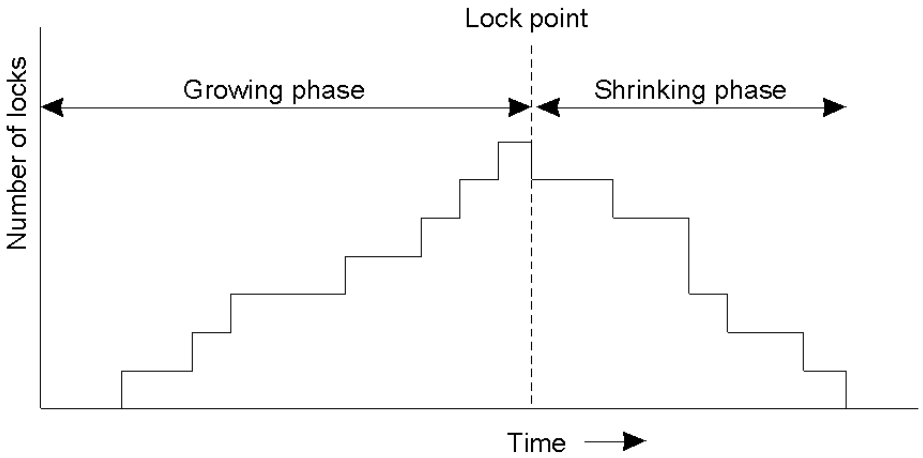
c

- Exclusive locks limit concurrency; an alternative is to use two types of locks (read and write)

		Lock requested	
		read	write
Lock already set	none	OK	OK
	read	OK	wait
	write	wait	wait

Two-Phase Locking (2PL)

- When a transaction *T* needs to access data *x* it must request the scheduler to grant a lock for *op(T,x)*
- Growing phase:
  - when the scheduler receives a request for op(T,x), it checks whether *x* is already "owned" by a conflicting lock
  - if so, op(T,x) is delayed, and therefore transaction T is blocked waiting …
  - … otherwise, the lock is granted
- Shrinking phase:
  - locks are released either as soon as they are no longer needed, or ...
  - ... (***strict 2PL***) only after making the commit/abort decision, and only after updates are persistent
  - once a lock for a transaction *T* has been released, *T* can no longer acquire it
- Strict 2PL prevents cascaded rollbacks
- Widely used: centralized and decentralized implementations



2PL and Serializability

- It can be formally proven that 2PL guarantees serializability
- Assume that *T'* performs an operation that conflicts with one that *T* has done
  - e.g., *T'* wants to update a data item *x* that was read or updated by *T*
  - e.g., *T* updated item *y*, and *T'* wants to read it
- *T* must have had a lock (e.g., *x* or *y*) that conflicts with the lock *T'* wants
  - *T* will not release the lock until commit or abort
  - *T'* will wait until *T* commits or aborts
- However, 2PL may cause ***deadlocks***

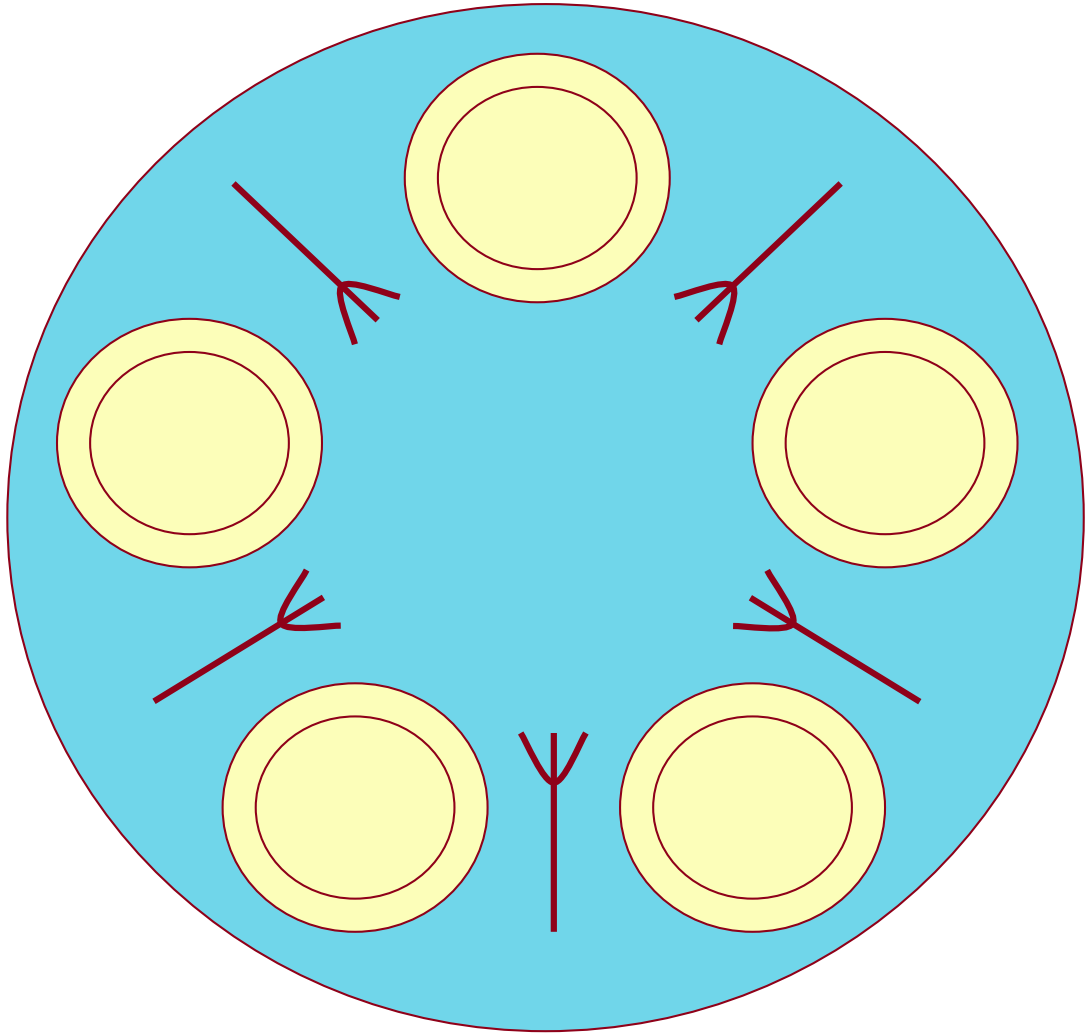
Detour: Safety and Liveness

- In concurrent systems (and therefore distributed as well) it is important to guarantee the following properties
- ***Safety property***: "nothing bad happens"
  - Some property of the system must not be violated because of concurrency
    - e.g., a valve should be opened only when a sensor is above threshold
- ***Liveness property***: "something good eventually happens"
  - The system eventually makes progress, i.e., correct (safe) actions are taking place

Detour: Deadlock and starvation

- Deadlock and starvation are two ways in which a concurrent system may lose its liveness property
- ***Deadlock:*** the entire system is stalled, each activity is waiting for someone else's resources, necessary to computation
- ***Starvation:*** one of the activities, although ready to make progress, cannot do so because only the other activities succeed in obtaining the necessary resources

The Dining Philosophers



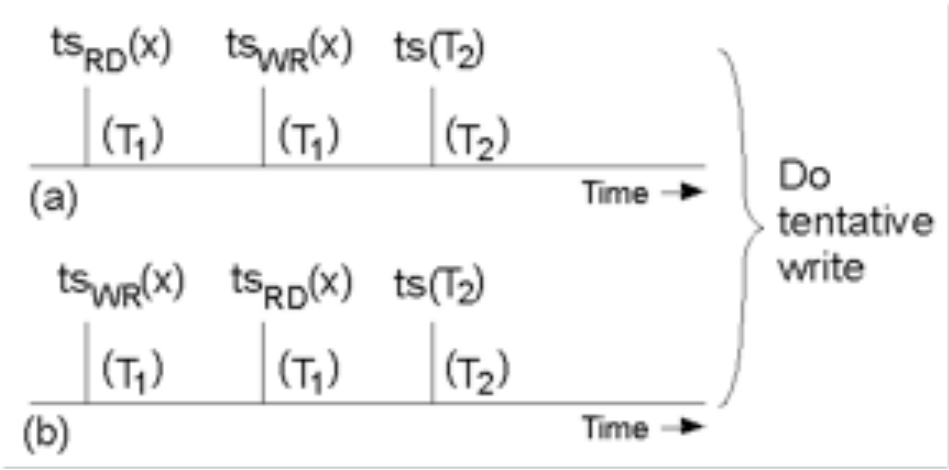
- Five philosophers meet for dinner
- Each needs two forks to eat (spaghetti)
- Each philosopher alternately eats or thinks
- Deadlock occurs for instance when each philosopher takes always the fork to the left, and waits for the right one to become available
- Starvation occurs when the neighbors of a philosopher F are always faster than F in picking up both of the forks close to F

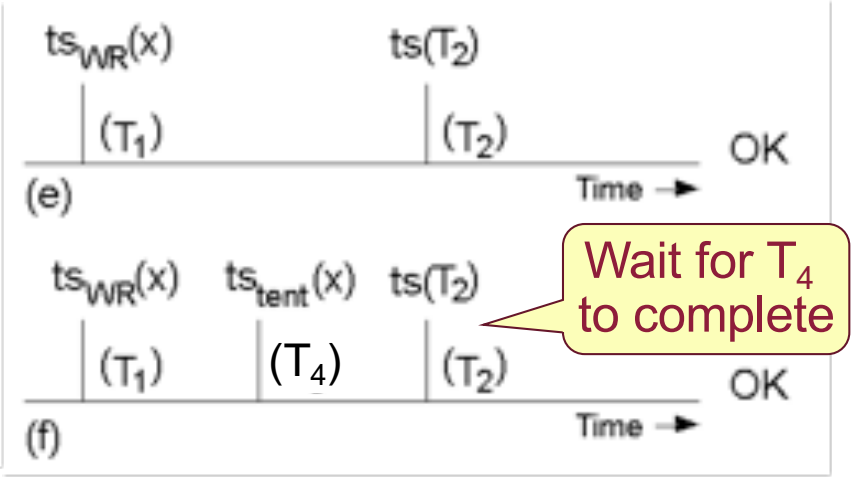
Pessimistic Timestamp Ordering

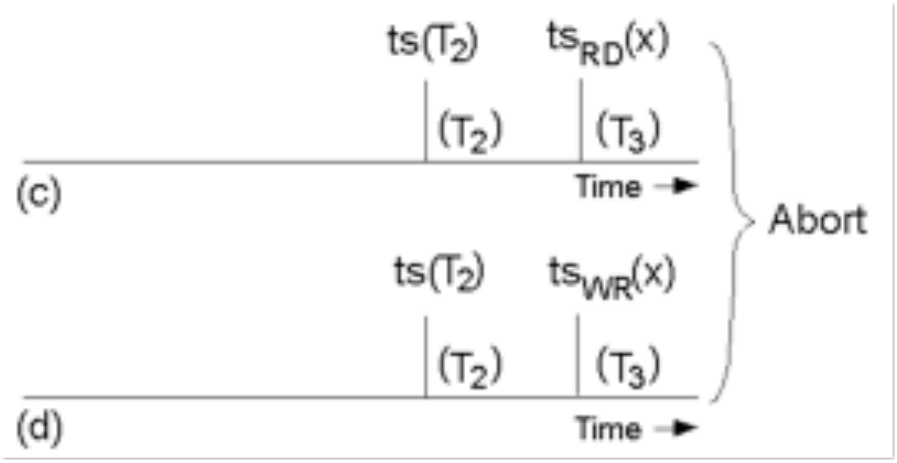
- Basic idea:
  - Assign a timestamp to each transaction: this defines the ordering among transactions
  - Assign to each data a (last) read and write timestamp
    - $ts_{RD}(x)$ and $ts_{WR}(x)$
  - Rule: A transaction can access an object only if it was accessed by "earlier" transactions
- More in detail:
  - If two operations conflict, the scheduler processes the one with the lowest timestamp first
  - Scheduler receives *read(T,x)*; *T* has timestamp *ts*
    - If $ts<ts_{WR}(x)$ abort T, a more recent write exists
    - If $ts>ts_{WR}(x)$ allow T and set $ts_{RD}(x)$ to $max(ts,ts_{RD}(x))$
  - Scheduler receives *write(T,x)*; *T* has timestamp *ts*
    - If $ts<ts_{RD}(x)$ **or** $ts<ts_{WR}(x)$, abort T, it is too late
    - If $ts>ts_{RD}(x)$ **and** $ts>ts_{WR}(x)$, allow T and set $ts_{WR}(x)$ to $max(ts,ts_{WR}(x))$
- Aborted transactions will reapply for a new timestamp and simply retry
- Deadlock-free

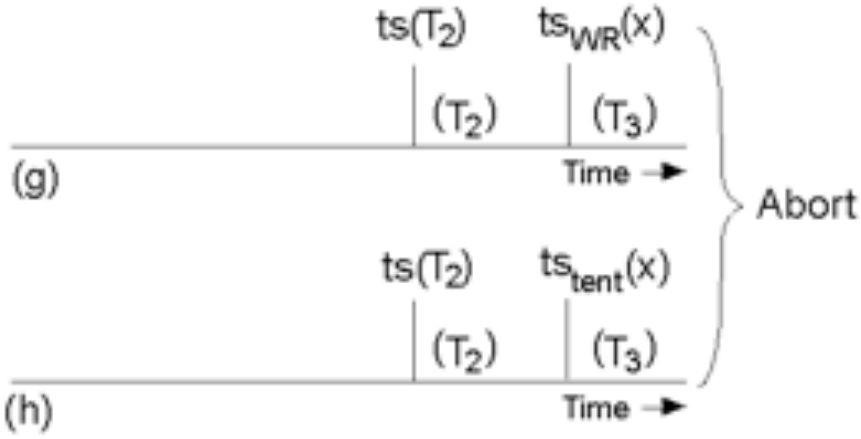
Example

- T_1 ran a long ago; T_2 and T_3 start concurrently, with $ts(T_2) < ts(T_3)$









T_2 tries to do a write

T_2 tries to do a read

Optimistic Timestamp Ordering

- Based on the assumption that conflicts are rare
- Therefore: do what you want without caring about others, fix conflicts later
  - Stamp data items with start time of transaction T
  - At commit of T, if any items “seen” by T have been changed by other transactions “overlapping” in time with T, T is aborted, otherwise committed
  - Best implemented with private workspaces
- Deadlock-free, allows maximum parallelism
- Under heavy load, there may be too many rollbacks