Advanced Distributed Algorithms and Data Structures

Chapter 8: Logical Clocks

Christian Scheideler

Institut für Informatik

Universität Paderborn

Overview

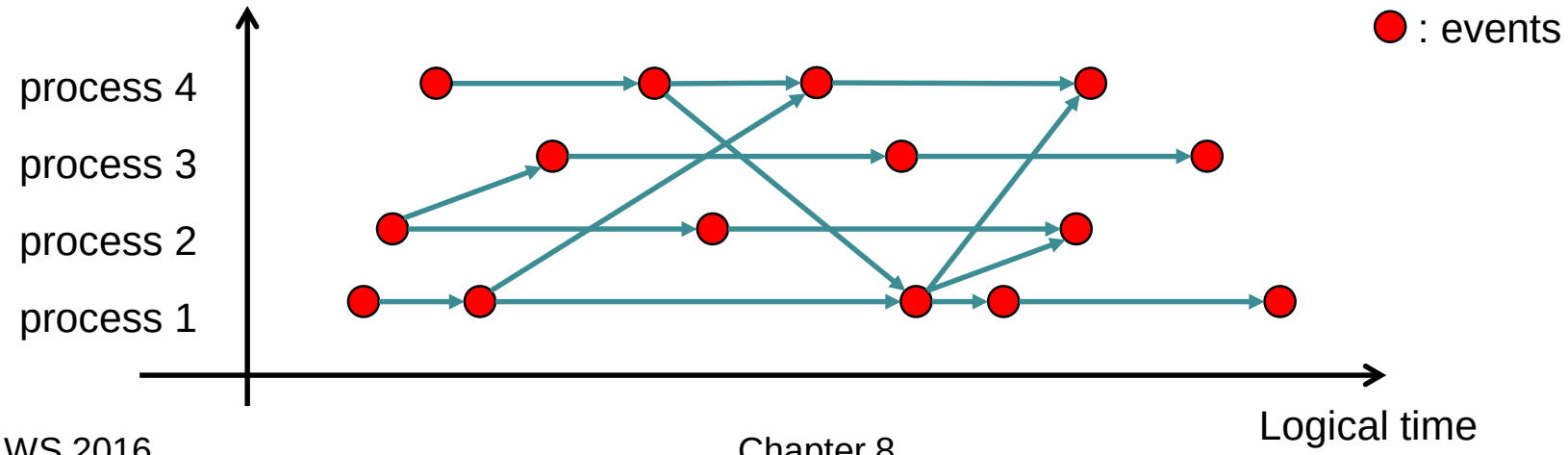
- Basics of Logical Clocks
- Hybrid Logical Clocks
- Applications

Logical Clocks

A logical clock is a mechanism for capturing chronological and causal relationships of events in a distributed system.

In our case:

- events cause executions of actions (recall that actions are of the form $\langle \text{event} \rangle \rightarrow \langle \text{commands} \rangle$)

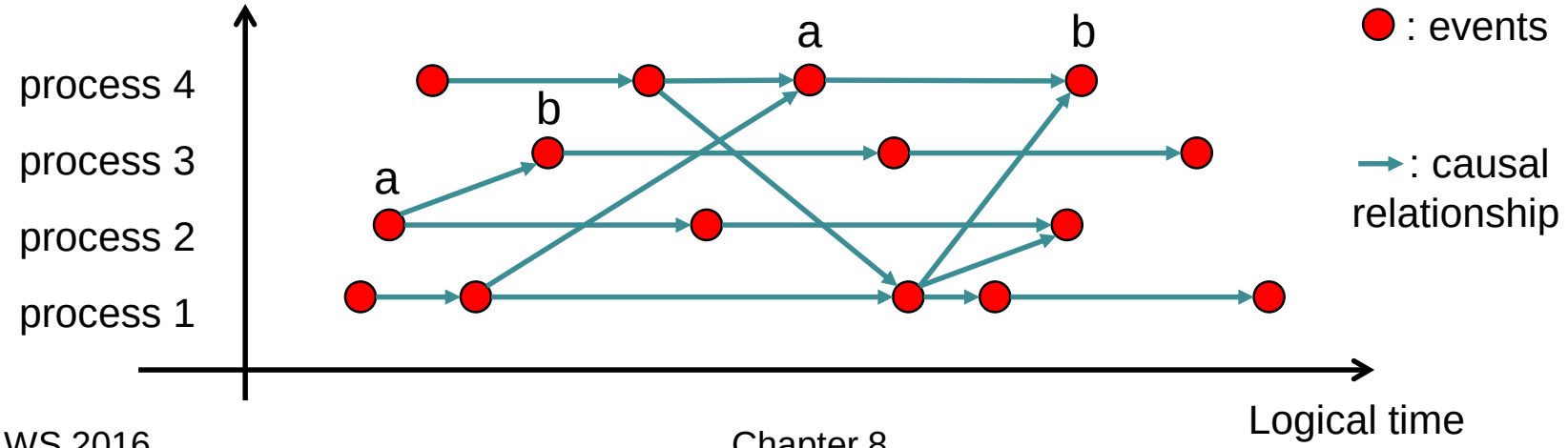


Logical Clocks

A logical clock is a mechanism for capturing chronological and causal relationships of events in a distributed system.

In our case:

- two events a and b are directly causally related () if a happened directly before b in the same process or a triggered b by a message

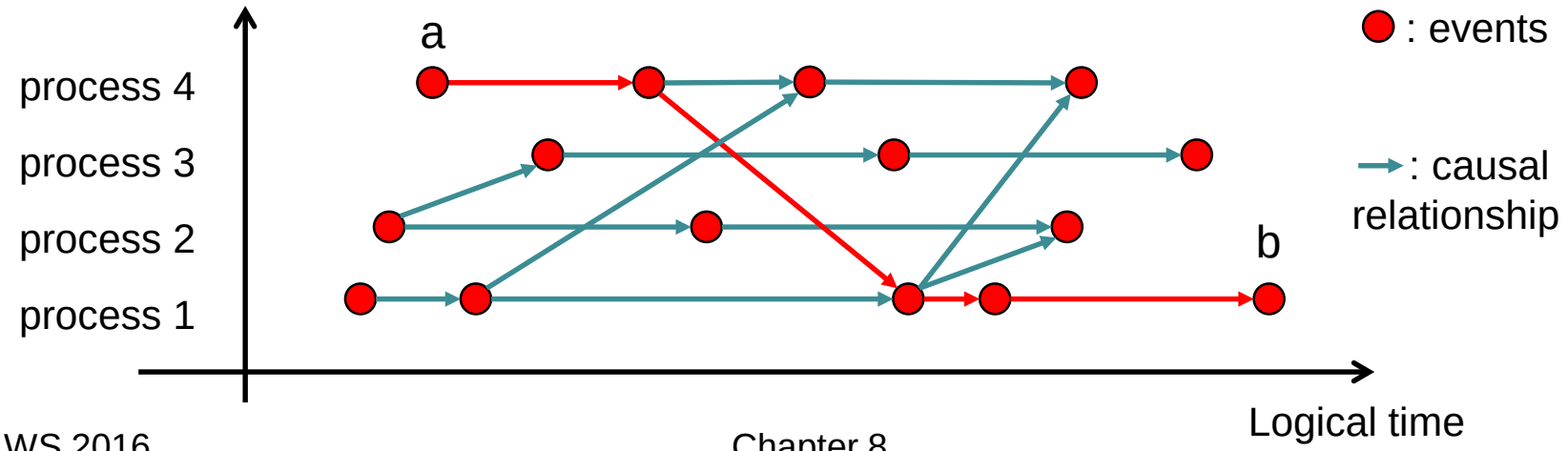


Logical Clocks

A logical clock is a mechanism for capturing chronological and causal relationships of events in a distributed system.

In our case:

- two events a and b are causally related if there is a directed path of direct causal relationships from a to b



Logical Clocks

Definition 8.1 (Lamport): The causal relation „$\rightarrow$“ on the set of events is the smallest relation satisfying the following three conditions:

1. If a is happening directly before b in the same process, then $a \rightarrow b$.
2. If b is triggered by a message from a, then $a \rightarrow b$.
3. If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.

Two events a and b are said to be concurrent if neither $a \rightarrow b$ nor $b \rightarrow a$.

Clock condition for any logical clock implementation C:

If $a \rightarrow b$ then $C(a)<C(b)$.

Logical Clocks

Clock condition for any logical clock implementation C:

If $a \rightarrow b$ then $C(a) < C(b)$.

From Definition 8.1 it follows that a logical clock C satisfying the clock condition must satisfy the following two requirements:

1. If a is happening directly before b in the same process, then $C(a) < C(b)$.
2. If b is triggered by a message from a, then $C(a) < C(b)$.

Simplest solution: Lamport clock L

Logical Clocks

Lamport clock L:

- Each process i maintains a clock L_i starting with 0.
- For an event a happening in process i, the Lamport clock $L(a)$ of a is defined as the clock value L_i of process i at the beginning of a.

Rules in order to satisfy conditions 1 and 2 in Def. 8.1:

- Each process i increments L_i between any two successive events.
- Every message m triggered by event a contains a timestamp $T_m=L(a)$. Upon processing a message m, process i sets $L_i:=\max\{L_i,T_m+1\}$.

Logical Clocks

In our case: an event triggers an action execution.

Recall that we consider the following types of actions:

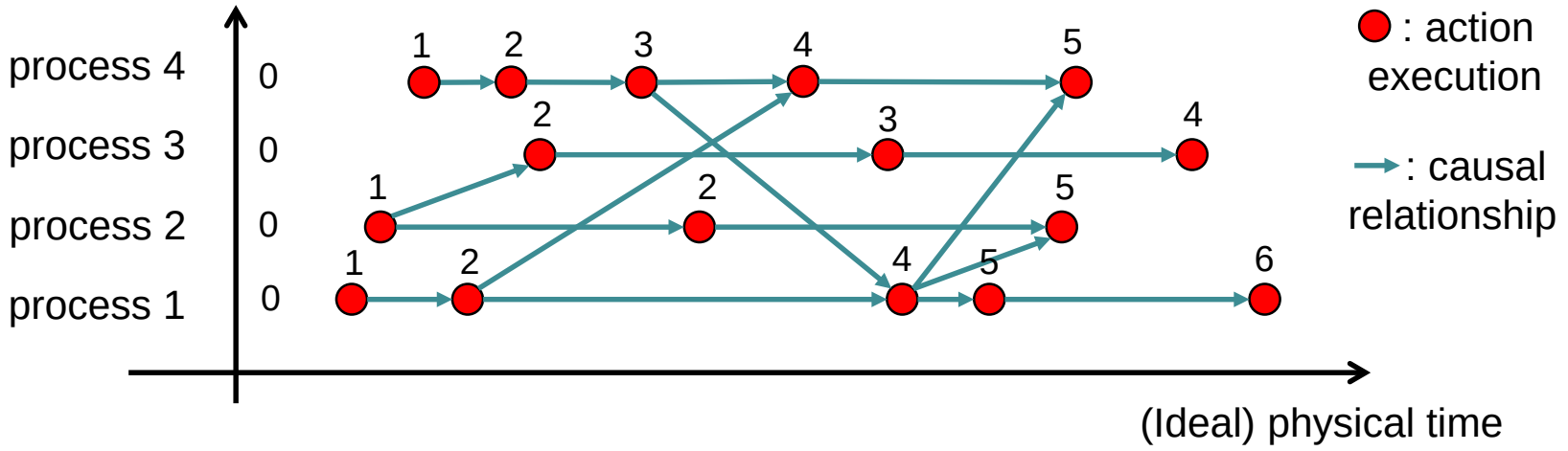
1. Triggered by a local state:
   ⟨name⟩: ⟨predicate⟩ → ⟨commands⟩
2. Triggered by a local/remote call:
   ⟨name⟩(⟨parameters⟩) → ⟨commands⟩

Clock update rules for each process i:

- Whenever an action of type 1 is to be executed, set $L_i := L_i + 1$.
- Whenever an action ot type 2 is to be executed due to a message m, set $L_i := \max\{L_i, T_m\} + 1$.

Logical Clocks

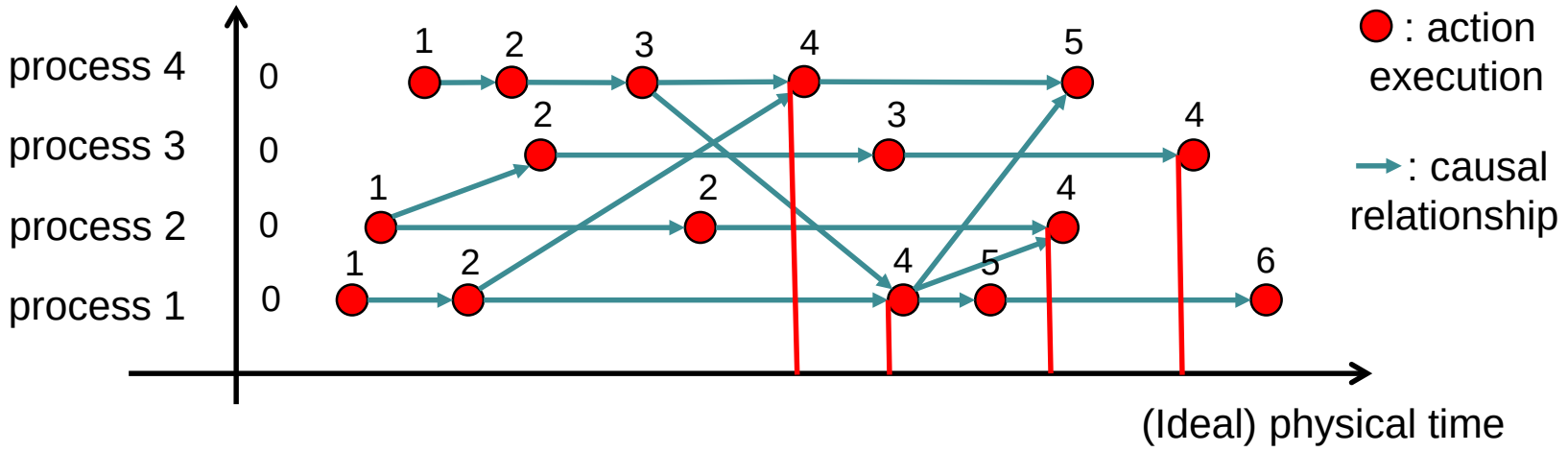
Example:



Problem: Answering „What was the state of the system at logical time x?“ may result in a system state consisting of process states that differ quite significantly w.r.t. their corresponding physical times.

Logical Clocks

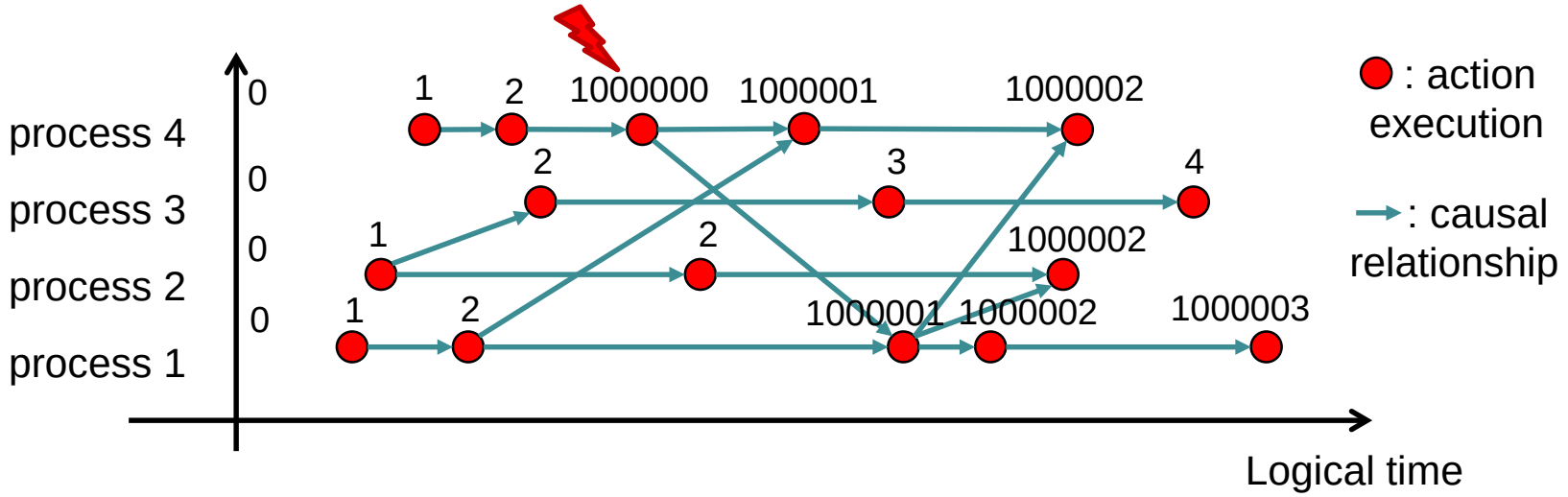
Example:



For example: „What was the state of the system at (logical) time 4?“

Logical Clocks

Example:



Problem: Lamport clocks are not robust against adversarial attacks if they have a finite domain since the adversary can cause an overflow.

Overview

- Basics of Logical Clocks
- Hybrid Logical Clocks
- Applications

Hybrid Logical Clocks

Basic idea: combine physical clocks with Lamport clocks in order to obtain so-called Hybrid Logical Clocks (HLC).

Definition 8.2:

- PC(a): physical clock value of the process at the beginning of event a
- C(a): physical clock value of event a
- L(a): logical clock value of event a
- pred(a): event preceding event a in its process
- HLC(a):=(C(a),L(a)): hybrid clock value of event a
- We define: HLC(a)<HLC(b) if and only if either C(a)<C(b) or C(a)=C(b) and L(a)<L(b).

Clock condition in this case: if $a \rightarrow b$ then HLC(a)<HLC(b).

Hybrid Logical Clocks

Rules for computing HLC(a):

- Action a triggered by a local state:

  $C(a) := \max\{C(pred(a)), PC(a)\}$
  if $C(a) = C(pred(a))$ then $L(a) := L(pred(a)) + 1$
  else $L(a) := 0$

- Action a triggered by a message m:

  $C(a) := \max\{C(pred(a)), PC(a)\}$
  if $C(a) = C(pred(a))$ then $L(a) := L(pred(a)) + 1$
  else $L(a) := 0$
  if $C(a) < C(T_m)$ or $(C(a) = C(T_m)$ and $L(a) \leq L(T_m))$ then
  $C(a) := C(T_m)$
  $L(a) := L(T_m) + 1$

Certainly, these rules satisfy conditions 1 and 2 in Definition 8.1.

Hybrid Logical Clocks

Extreme cases:

Case 1:

- physical clocks are perfectly synchronized
- messages have a delay >0

Then HLC(a)=PC(a) for all events a.

Case 2:

- physical clocks ignored / unavailable (PC(a):=0 in this case)
- messages have arbitrary non-negative delays

Then HLC(a)=L(a) (which is identical with the Lamport clock value) for all events a.

All other cases: HLC(a) somehow between being a physical clock and a logical clock value.

Hybrid Logical Clocks

Consider some ideal clock IC unknown to the processes that never halts or runs backwards but may proceed at different speeds.

Theorem 8.3: If there is an ideal clock so that the physical clocks of the processes are within an additive ε of the ideal clock value, then for any event a, C(a) is within an additive ε of the ideal clock value.

Proof:

- IC(a): ideal clock value at the beginning of event a.
- PC(a): physical clock value at the beginning of event a.
- Note that the events can be topologically ordered based on their causal relationships. We prove by induction on this ordering that Theorem 8.3 holds for every event.
- Induction start:
  For the first event a in each process, C(a) is set to PC(a) and since $|PC(a)-IC(a)|\leq\varepsilon$, also $|C(a)-IC(a)|\leq\varepsilon$.

Hybrid Logical Clocks

Theorem 8.3: If there is an ideal clock so that the physical clocks of the processes are within an additive ε of the ideal clock value, then for any event a, C(a) is within an additive ε of the ideal clock value.

Proof:

Induction step:

Case 1: event a triggered by a local state.

- By induction, we know that $|C(pred(a))-IC(pred(a))|\leq\varepsilon$.
- We also know that $|PC(a)-IC(a)|\leq\varepsilon$, $IC(pred(a))\leq IC(a)$, and C(a) is set to $max\{C(pred(a)),PC(a)\}$.
- Case 1a: $PC(a)<C(pred(a))$.
  Then $C(a) = C(pred(a)) > PC(a)$. Hence,
  (*) $IC(a)-C(a) < IC(a)-PC(a) \leq \varepsilon$.
  On the other hand, $IC(a)\geq IC(pred(a))$. Hence,
  (**) $C(a)-IC(a) = C(pred(a))-IC(a) \leq C(pred(a))-IC(pred(a)) \leq \varepsilon$.
  From (*) and (**) it follows that $|C(a)-IC(a)|\leq\varepsilon$.
- Case 1b: $PC(a)\geq C(pred(a))$.
  Then $C(a)=PC(a)$. Thus, $IC(a)-C(a) = IC(a)-PC(a) \leq \varepsilon$ and $C(a)-IC(a) = PC(a)-IC(a) \leq \varepsilon$, which implies that $|C(a)-IC(a)|\leq\varepsilon$.

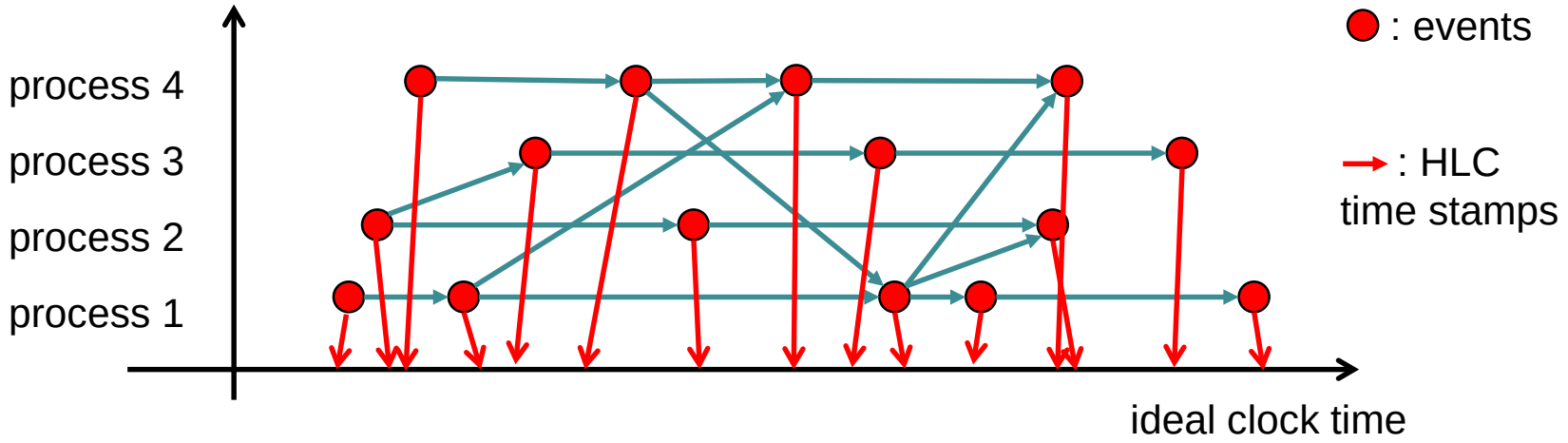
Case 2: event triggered by a message.

Exercise: also show $|C(a)-IC(a)|\leq\varepsilon$ for the case that event a is triggered by a message m.

Hybrid Logical Clocks

Remarks:

- The assumption for the ideal clock in Theorem 8.3 can be achieved under reasonable assumptions on message delays (see the conjecture on slide 44 of Chapter 7).
- If the assumption of Theorem 8.3 holds, the state of the system for some HLC-time is much closer correlated to the physical time.



Robust Hybrid Logical Clocks

Problem: HLC rules still not robust against adversarial attacks that produce overflows or cause $C(a)$ to significantly deviate from $PC(a)$ for some events a.

Idea: put the processing of action a triggered by message m on hold until $PC(a) > C(T_m)$ so that very large $C(T_m)$ or $L(T_m)$ values do not cause overflows or significant jumps of the C or L values of a compared to $pred(a)$.

$\rightarrow$ Message queue becomes priority queue based on $C(T_m)$-values.

A prerequisite of the idea to work is a robust physical clock synchronization mechanism so that clocks of honest processes are not corrupted.

Remark: Recall that the median rule from Chapter 7 can be made very robust against massive DoS attacks and even manipulations of clock values. Yet, large message delays might cause a large ε in Theorem 8.3 over time, but in the worst case the adversary can cause the physical clocks to deviate at most by their natural drift when making message delays very large (see Chapter 7, slide 44), which can be tolerated for quite some time due to the slow drift.

Overview

- Basics of Logical Clocks
- Hybrid Logical Clocks
- Applications

Applications of HLCs

Assumptions:

- We use robust hybrid logical clocks.
- For all processes i, $|PC(i)-IC| \leq \varepsilon$.
- All message delays are at most ε.

Applications:

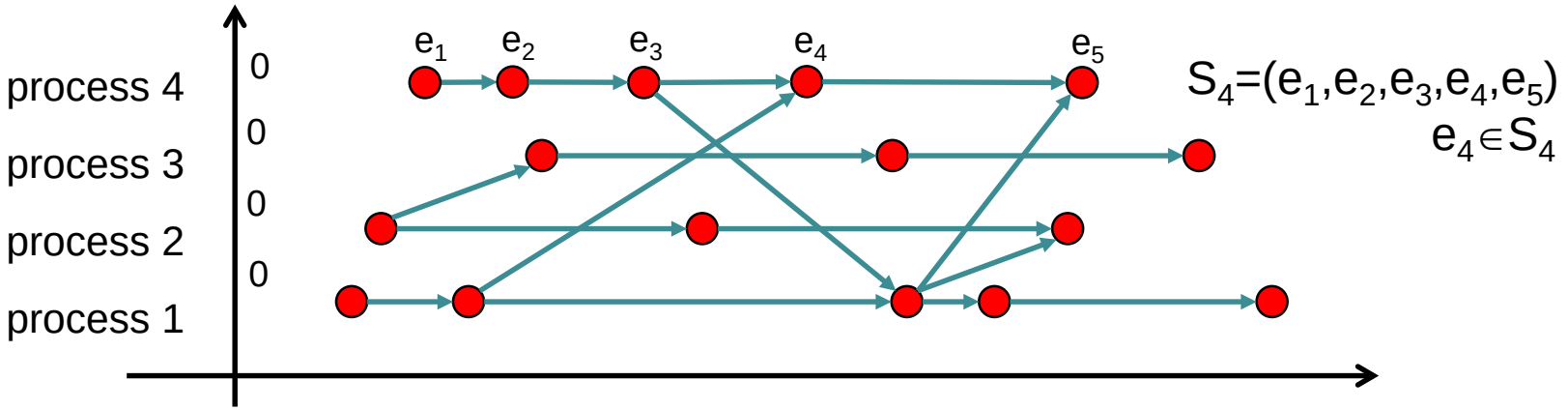
- Snapshots
- Transactions

Snapshots

Definition 8.4:

- State of a process: all data contained in it
- State of network: all messages currently in transit
- State of system: combination of all process states and the state of the network

The state of a process is uniquely characterized by the sequence of local events causing it. Given an event e and a process (resp. system) state S, we say that S depends on e (or short, $e \in S$) if and only if e is part of that state.

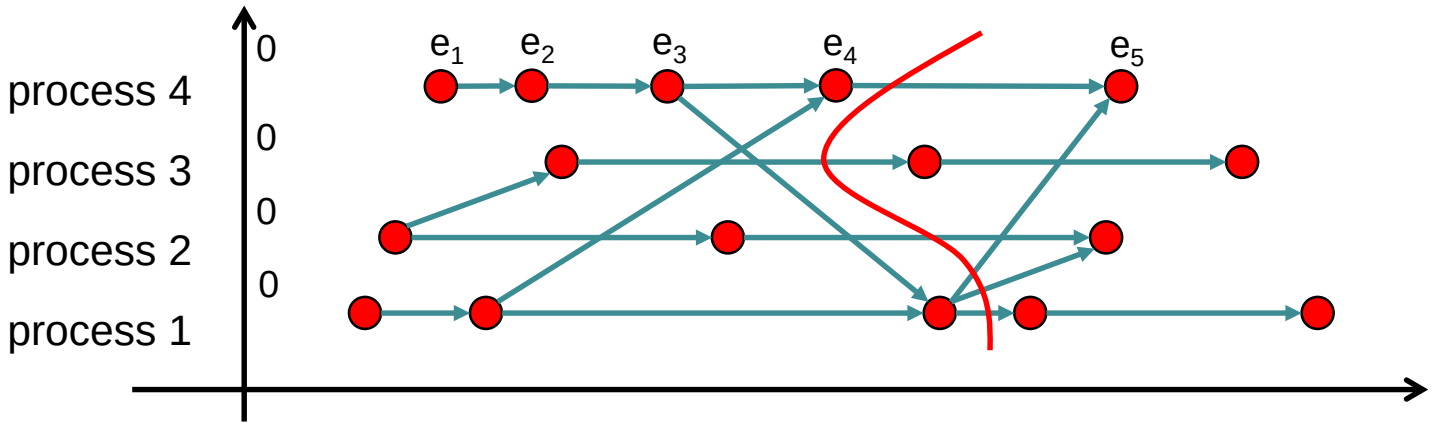


Snapshots

Definition 8.5: A system state S is consistent if for any two events a and b with $a \rightarrow b$: if $b \in S$ then also $a \in S$.

Example:

events left to red curve form consistent system state

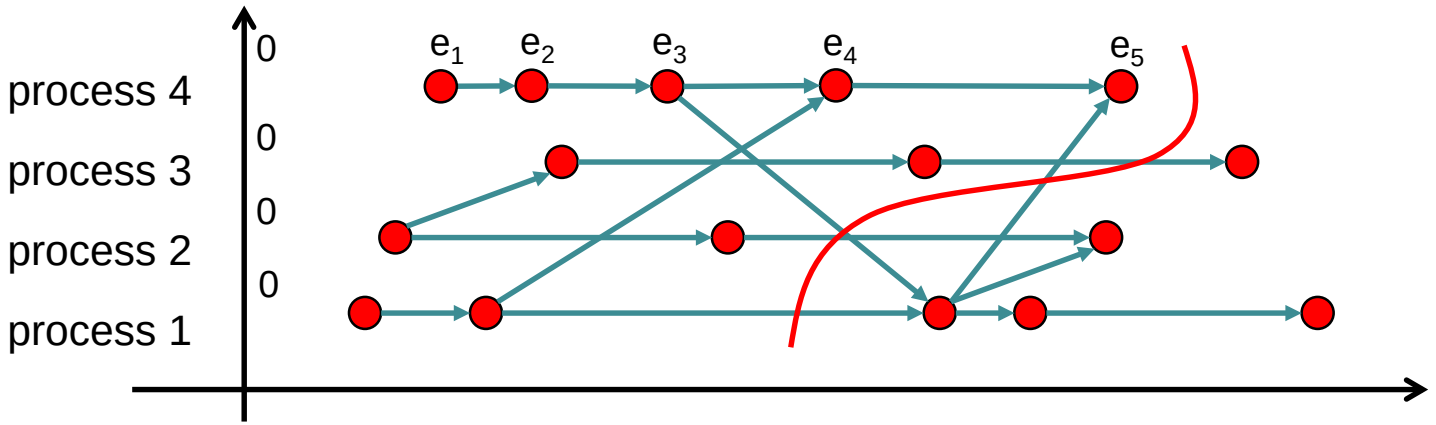


Snapshots

Definition 8.5: A system state S is consistent if for any two events a and b with $a \rightarrow b$: if $b \in S$ then also $a \in S$.

Example:

events left to red curve do not form consistent system state



Snapshots

Definition 8.5: A system state S is consistent if for any two events a and b with $a \rightarrow b$: if $b \in S$ then also $a \in S$.

This means, for example, that whenever an event b is triggered by a message from event a, then for a system state to be consistent it should not happen that it depends on b but not on a.

Snapshot problem: Record a consistent system state.

The recording of a system state is usually triggered by a dedicated process called the observer.

Snapshots

Assumptions:

- We have an HLC implementation.
- All processes keep track of the sequence of events (or more precisely, the changes of their state caused by them) and their HLC-values.

Snapshot algorithm:

1. Process P_0 (the observer) selects some snapshot time T and sends it to all other processes.
2. Every other process P_i sends the unique state back to P_0 representing the sequence of all local events e with HLC-value $<(T,0)$.

The state in step 2 may either be constructed from the log of P_i´s events or returned once P_i´ has had an event with HLC-value $\geq(T,0)$. Note that just waiting for the physical clock to be at least T does not suffice since it might be adjusted backwards.

Snapshots

Recall our assumptions:

1. We use robust hybrid logical clocks.
2. For all processes i, $|PC(i)-IC|\leq\varepsilon$.
3. All message delays are at most ε.

Then it follows from Theorem 8.3:

- If items 1 and 2 hold, then for any snapshot, the time difference of the states of the processes is just $O(\varepsilon)$.
- If items 1-3 hold and the observer wants to have a snapshot of the current time step, then only a rollback of $O(\varepsilon)$ steps of the other processors is needed for the snapshot.

Snapshots

Problem: high contention at the observer since the number of processes resp. their states might be quite large.

Solution:

- Instead of the observer sending a message directly to all processes, it can make use of more scalable broadcasting techniques that are discussed in Chapter 10.
- In many applications (like a fire detection system) just some aggregate value of the process states is needed. We refer to Chapter 10 for techniques to compute such a value in a scalable way.
- In other applications like a system rollback or establishing a checkpoint for rollback, the states of the processes actually do not have to be sent back to the observer but just acknowledgements are needed.

Snapshots

Problem: the current way of taking a snapshot is not sufficient to perform a system rollback in order to continue its execution from that snapshot. The reason is that messages in transit are not recorded by the snapshot.

Possible solutions:

- Keep a log of messages that have not reached their final destination yet. This, however, can be difficult in our TCM model as a message may travel through various relays.
- Alternatively, one may send a snapshot message along each outgoing relay of every process, and if the relays are guaranteed to send their messages in FIFO order, a process knows that it has received all messages sent before the snapshot once it receives the snapshot message from each incoming connection. But this requires all processes to be aware of the number of incoming connectings into some relay. Maybe, some TCM-layer support is justified here?

Applications of HLCs

Assumptions:

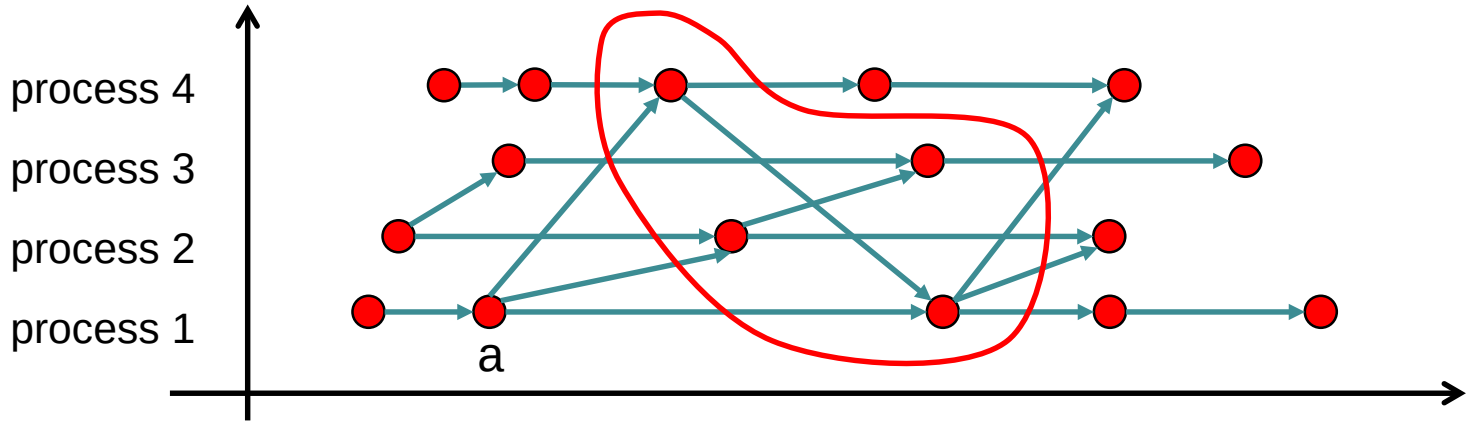
- We use robust hybrid logical clocks.
- For all processes i, $|PC(i)-IC| \leq \varepsilon$.
- All message delays are at most ε.

Applications:

- Snapshots
- Transactions

Transactions

Transaction processing links multiple individual events in a single, indivisible event called transaction.



A transaction is usually rooted at a single event (that triggered it) which might be part of another transaction (like event a in the example).

Transactions

Transaction processing links multiple individual events in a single, indivisible event called transaction.

Definition 8.6:

- Given a transaction T, let P_T be the set of all processes that contain at least one of the events of T.
- Given two transactions T and T', we say that T happens before T' (or short, $T \rightarrow T'$) if there are events $a \in T$ and $b \in T'$ with $a \rightarrow b$.
- Given a system state S and transaction T, we say that S depends on T if there is an event $a \in T$ with $a \in S$.

Definition 8.7: Suppose that all events in a system are associated with transactions. (In the simplest case, a transaction just consists of a single event.) A system state S is called valid if

- Atomicity: For all transactions T that S depends on, all events in T must be in S.
- Isolation: There are no two transactions T and T' with $T \rightarrow T'$ and $T' \rightarrow T$.

A valid state is called consistent if if for any two transactions T and T' with $T \rightarrow T'$: if $T' \in S$ then also $T \in S$.

Transactions

First idea: reduce the problem of processing transactions to the mutual exclusion problem (see Chapter 6).

Basic protocol for transaction T:

1. Send LOCK(T) requests to all processes in P_T (with a probability controlled by a fair MIMD protocol like in Chapter 6).
2. If ACK(T) replies are obtained from all processes in P_T (before the next attempt to send out LOCK(T) requests), then continue with step 3. Otherwise, send RELEASE(T) messages to all processes in P_T to release the locks (instead of sending LOCK(T) requests).
3. Process the transaction. Once all of its events have been processed, send RELEASE(T) messages to all processes in P_T.

A process only replies with an ACK(T) if it is currently not locked and received only one LOCK request.

Problem: the success probability might be very small (if $|P_T|$ is large), and if a process fails, locks will not be released.

Transactions

Problem: the success probability might be very small

Solution:

- Suppose that all information in a process is organized in independent units called objects, i.e., the set of operations (read, write, insert, delete,…) specified for each of these objects does not require access to any of the other objects.
- Also, suppose that we know in advance which objects need to be accessed by a transaction (which is often the case).
- Then a transaction T only requires locks for the objects accessed by it, and if there is no concurrent locking request for one of these objects resp. none of the objects is currently locked, the process can return an ACK(T).

Example: transactions transfer money between bank accounts

Transactions

Problem: if a process fails, the locks will not be released.

Solution:

- Suppose that we know how much time a transaction needs (see Solution 2 on slide 8 of Chapter 7 to obtain a good estimate over time).
- Then it suffices if the process that wants to execute transaction T just asks all processes in P_T to set a lock for a certain HLC-time frame.

Problem: How do processes in P_T learn that transaction T was successful?

Transactions

Problem: How do processes in P_T learn that transaction T was successful?

Basic protocol:

1. Send LOCK(T,I) requests with HLC-interval I to all processes in P_T (with a probability controlled by a fair MIMD protocol like in Chapter 6).
2. If ACK(T) replies are obtained from all processes in P_T by the beginning of I, continue with step 3. Otherwise, send RELEASE(T) messages to all processes in P_T to release the locks.
3. Process the transaction. If all of its events have been processed within interval I, send COMMIT(T) messages to all processes in P_T. Otherwise, send UNDO(T) to all processes in P_T.

We will talk in more detail about transactions in Chapter 11.

Transactions

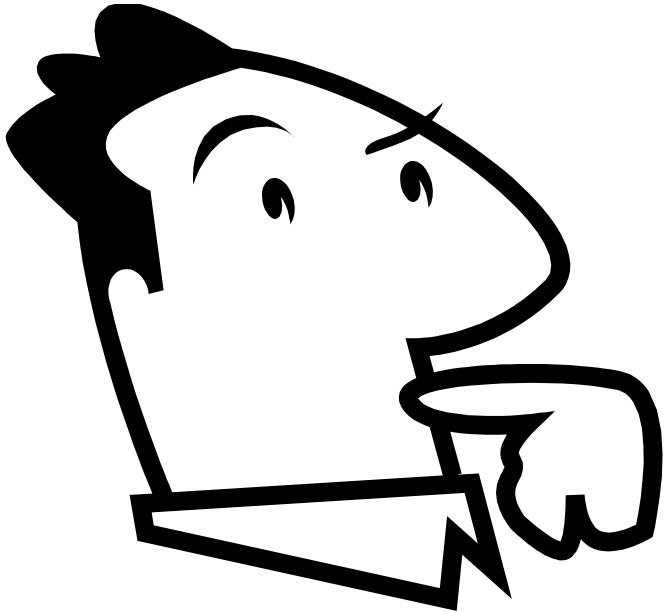
Recall our assumptions:

1. We use robust hybrid logical clocks.
2. For all processes i, $|PC(i)-IC| \leq \varepsilon$.
3. All message delays are at most ε.

Then it follows from Theorem 8.3:

- If items 1-3 hold, then the starting point of the proposed transaction interval just has to be $O(\varepsilon)$ steps in the future to receive the acknowledgements in time.
- Also, if items 1-3 hold, and any transaction requires much more than $O(\varepsilon)$ time, then a rollback of events due to UNDO(T) requests will affect the events of at most one transaction succeeding T in any of the processes in P_T.

  Why?



Questions?