Events e and f are *independent* if $\neg(e \xrightarrow{p} f)$ and $\neg(f \xrightarrow{p} e)$.

A potential causality diagram is defined as a partially ordered set $(E, \xrightarrow{p})$, where E is a set of events and $\xrightarrow{p}$ is the potential causality relation.

Given $(E, \xrightarrow{p})$ a happened-before diagram is *consistent* with it if $\xrightarrow{p} \subseteq \rightarrow$.

In summary we can set up the following table:

Model	Basis	Type
Interleaving	Physical time	total order on all events
Happened-before	Logical order	total order on each process
Potential causality	Causality	partial order on each process.

Table 1: Properties of the fundamental models

5 Logical Clocks (week 5)

Logical clocks[1] in general use the happened-before relation. Normally, we order events, but state-based models are possible too and they are very popular.

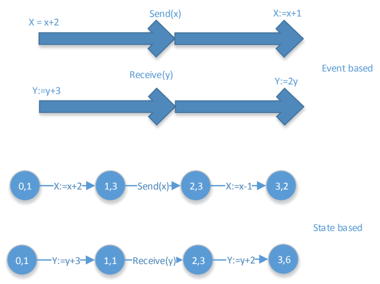

Figure 2: Equivalent models based on events and states

- events cause state transaction
- state includes program counter (PC)

Properties:

- any partially ordered set(POSET) of events in which all events on one process are totally ordered is a distributed computation in the happened before model.
- not every poset of states is a valid distributed computation

  we can construct partially ordered states, that have no corresponding partially ordered events as there are cycles on events 1) (e, f) and 2) (e, f, g) c.f. Figure 3. Please note that events are the arcs between states. Following the happened-before relation, we can either formulate a relation for states, or for events.

[1]The following material has been written down by Arndt Lasarzik.

- partial order of states that avoids cycles of events in deposed(decomposed partially ordered set)

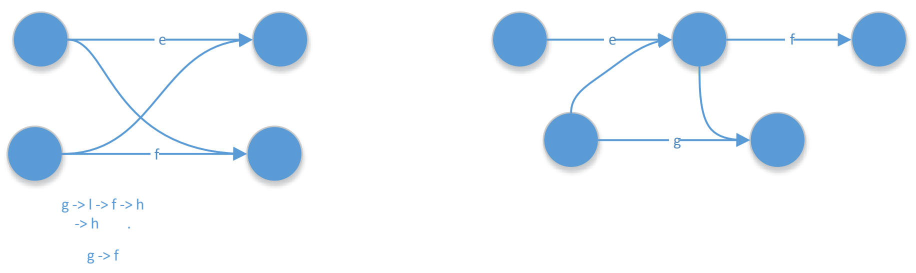

Figure 3: Cycles of events

Generally speaking, clocks are used for tracking those relations[2]

1. global total order is tracked by logical clock
2. happened before can be tracked by a vector clock
3. tracking potential causality needs mechanisms to detect independence and is omitted here for complexity reasons.

Logical clocks create order as it <u>could have</u> happened in a distributed system. (In the interleaving model this is a global total order because we map all events onto one time line). Normally, we have no global wall-clock time, that would allow us to reason about duration of events. We can merely state the order of events

- within one process
- send/receive of messages

Even in the interleaving model, if we have no absolute time, we have a global total order without duration of events. This total order can be represented using logical clocks.

Definition logical clock A logical clock C is a mapping from $S \in \mathcal{S} \to \mathbb{N}$ (i.e. the set of all events, or states) with

$$\forall s, t \in \mathcal{S} : s \prec_{lm} t \vee s \rightsquigarrow t \Rightarrow C(s) < C(t)$$

equivalently

$$(\mathbf{CC}) \qquad \forall s, t \in \mathcal{S} : s \to t \Rightarrow \forall c \in \mathcal{C}\ C(s) < C(t) \qquad \text{Lamport's logical clock condition (CC)}$$

legend: $\to$ = happened before, $\rightsquigarrow$ = remotely precedes, $\prec$ = immediately precedes

Each state in each process has a clock value that is the assigned natural number. We may wonder which states are parallel? We can show that for parallel states there exists a clock such that the

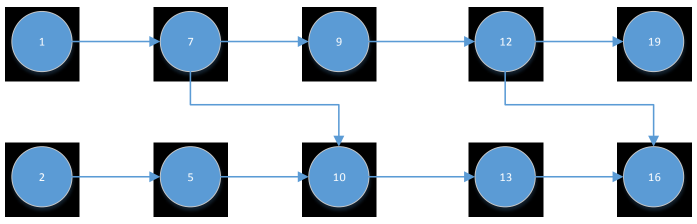

Figure 4: Events in two processes

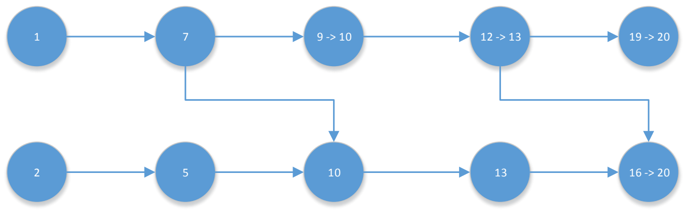

Figure 5: Possible adjustment of logical clock algorithm in two processes

clock values are identical. A clock assignment could for instance create u, v and assign to both $\max(u, v)$ and then increase all later clock values by $[u - v]$:

Clock algorithms in the happened-before (distributed) model must create a partial order. A logical clock satisfies

$$s \to t \Rightarrow s.c < t.c$$

This means, that if s happened before t then the clock value of s will be truely less than the clock value of t.

The converse is may not be true.

The objective of a vector clock is to create a situation where we can conclude from clock values to the happened before relation of events. This happens through a vector of state identifiers:

Definition 9 *Vector clock. A vector clock is a mapping from S to $\mathbb{N}^N$ with*

$$\forall s, t \qquad s \to t \Leftrightarrow s.v < t.v$$

We note that a vector clock has N elements if there are N processes. Then $s.p$ denotes the process in which event s happens and $s.v$ is the vector of clock values in process s. For a vector clock we know that if two clock values are ordered, then also the coresponding events are in happened-before relation. But also vector clocks only create a partial order, i.e. there may be events with vector clock values that cannot be ordered.

To be able to reason about ordering of vector clock values we must be able to compare vector clock value x with y. This is done in the following way

$$\begin{aligned} x < y \quad &\Leftrightarrow \quad (\forall k : 1 \leq k \leq N : x[k] \leq y[k]) \wedge \\ & \quad (\exists j : 1 \leq j \leq N : x[j] < y[j]) \\ x \leq y \quad &\Leftrightarrow \quad (x < y) \vee (x = y) \end{aligned}$$

[2]1) global order of events, 2) happend before order, 3) potential causality order

For more than one process the order will only be partial, as there can be states/events for which no order can be established.

A logical clock algorithm could be the following: Upon receiving a message the algorithm takes

```
$P_i$ ::
var
    c: integer initially 0;

send event (s, send, t);
    // s.c is sent as part of the message
    $t.c := s.c + 1$;

receive event(s, receive(u), t);
    $t.c := \max(s.c, u.c) + 1$;

internal event(s, internal, t);
    $t.c := s.c + 1$;
```

Figure 6: A logical clock algorithm

the maximum of the two timestamps and adds one to it. Every algorithm that satisfies Lamport's logical clock algorithm (**CC**) would be a valid algorithm. Conformity to this property can be proven.

Vector clock algorithms do essentially the same, but they require more notation since every process holds a vector of clock values for all processes.

Obviously, for several processes only a partial order can be established.

```
$P_j$ ::
var
    v: array[1...N] of integer
        initially $(\forall i, i \neq j : v[i] = 0) \wedge (v[j] = 1)$;

send event (s, send, t);
    $t.v := s.v$;
    $t.v[j] := t.v[j] + 1$;

receive event(s, receive(u), t);
    for $i := 1$to $N$ do
        $t.v[i] := \max(s.v[i], u.v[i])$;
    $t.v[j] := t.v[j] + 1$;

internal event(s, internal, t);
    $t.v := s.v$;
    $t.v[j] := t.v[j] + 1$;
```

Figure 7: A vector clock algorithm