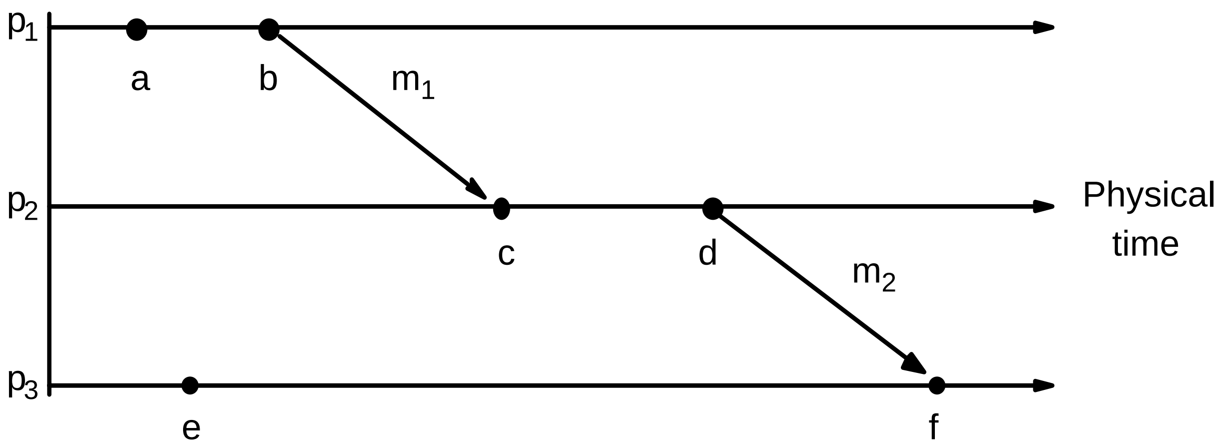
Logical time and logical clocks

- Knowing the ordering of events is important
  - not enough with physical time
- Two simple points [Lamport 1978]
  - the order of two events in the same process
  - the event of sending message always happens before the event of receiving the message.
- happened-before relations: partial order, →
  - HB1, HB2
  - HB3 means happened-before relation is transitive



$a \rightarrow b$ (at $p1$) $c \rightarrow d$ (at $p2$)

$b \rightarrow c$ ($m1$)

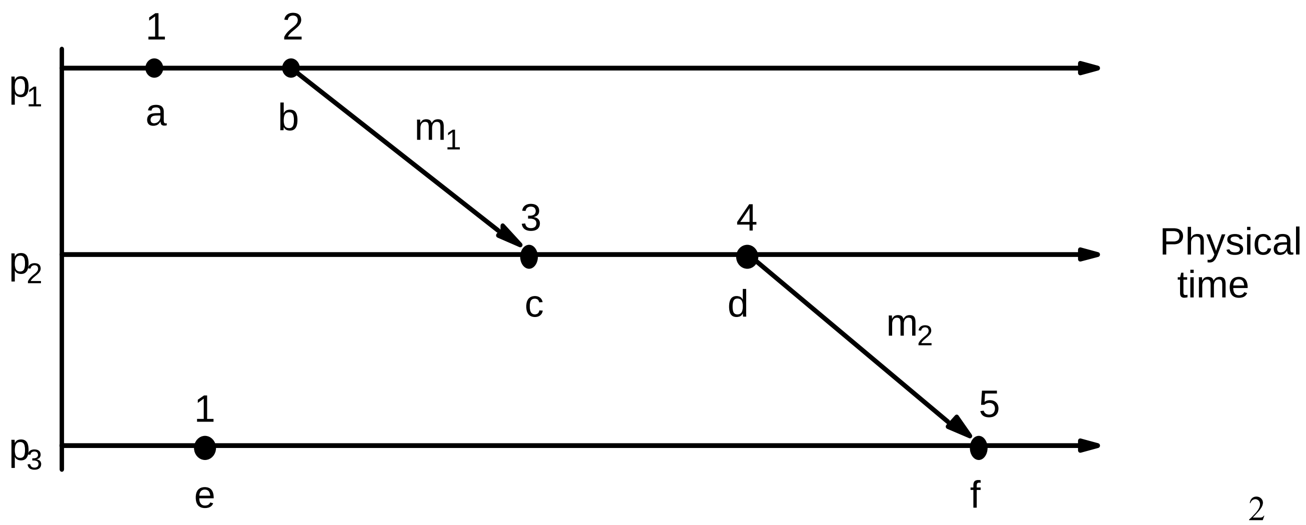
also $d \rightarrow f$ ($m2$)

Not all events are related by →, e.g., a $\not\rightarrow$ e and e $\not\rightarrow$ a they are said to be concurrent; write as $a \parallel e$

Lamport's logical clocks

- It is a monotonically increasing software counter. It need not relate to a physical clock
- Each process p_i has a logical clock L_i
  - LC1: L_i is incremented by 1 before each event at process p_i
  - LC2: (a) when process p_i sends message m, it piggybacks $t = L_i$

    (b) when p_j receives (m,t), it sets $L_j := max(L_j, t)$ and applies LC1 before timestamping the event *receive* (m)
- $e \rightarrow e' \Rightarrow L(e) < L(e')$ but not vice versa
  - Example: event b and event e
  - shortcoming of Lamport's clock



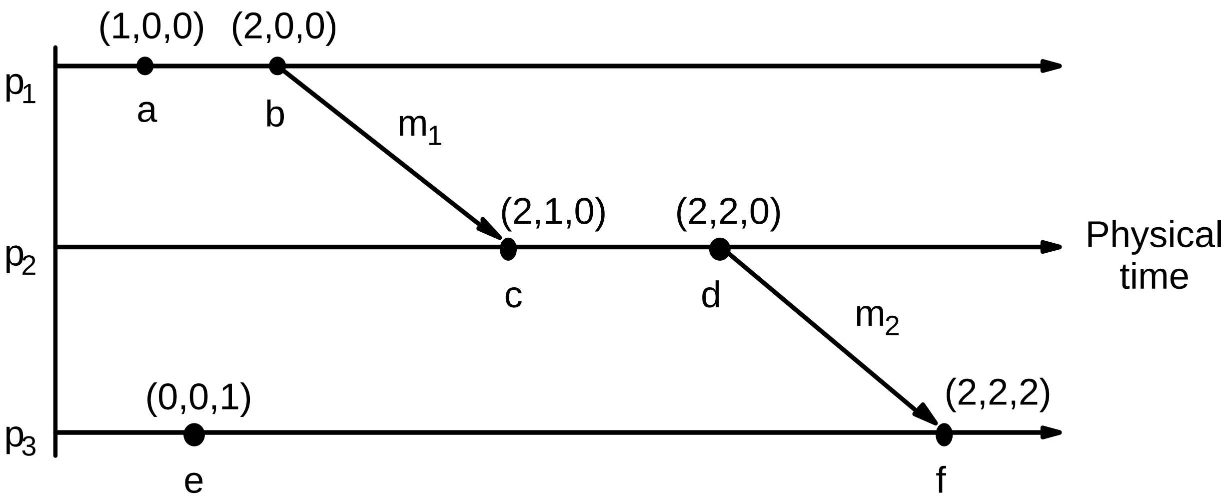
2

Vector clocks (Mattern [1989] and Fidge [1991])

- Fix the problem in Lamport's clock
- Vector clock: an array of N integers for a system with N processes. Each process Pi has its own local vector clock Vi.
- Rules for updating clocks:
  - VC1:initially $V_i[j] = 0$ for $i, j = 1, 2, \ldots N$
  - VC2:before p_i timestamps an event it sets $V_i[i] := V_i[i] + 1$
  - VC3: p_i piggybacks $t = V_i$ on every message it sends
  - VC4: when p_i receives (m,t) it sets $V_i[j] := \max(V_i[j], t[j])$ $j = 1, 2, \ldots N$ (then adds I to its own element using VC2)
    - Merge operation
- E.g. at p_2, (0, 0, 0) -> (0, 1, 0) -> (0, 2, 0) -> (0, 3, 0) … -> (1, 4, 3)
  - Now, received a mes. from p_3 that piggybacks t = (1,0,3).
- $V_i[i]$ is precise information; $V_i[j]$ ($j \neq i$) is updated from received messages.
  - In RIP, periodic updates and triggered updates
  - only triggered updates by received messages

Compare vector timestamps

- Meaning of =, <=, < for vector timestamps
  - (1) V = V' iff V[j] = V'[j] for j = 1, 2, …, N
  - (2) $V \leq V'$ iff $V[j] \leq V'[j]$ for j = 1, 2, …, N
  - (3) $V < V'$ iff $V \leq V'$ and $V \neq V'$
- Examples: (1, 3, 2)<(1, 3, 3); (1, 3, 2)| |(2, 3, 1)
- Note that $e \rightarrow e'$ implies $V(e) < V(e')$. The converse is also true.



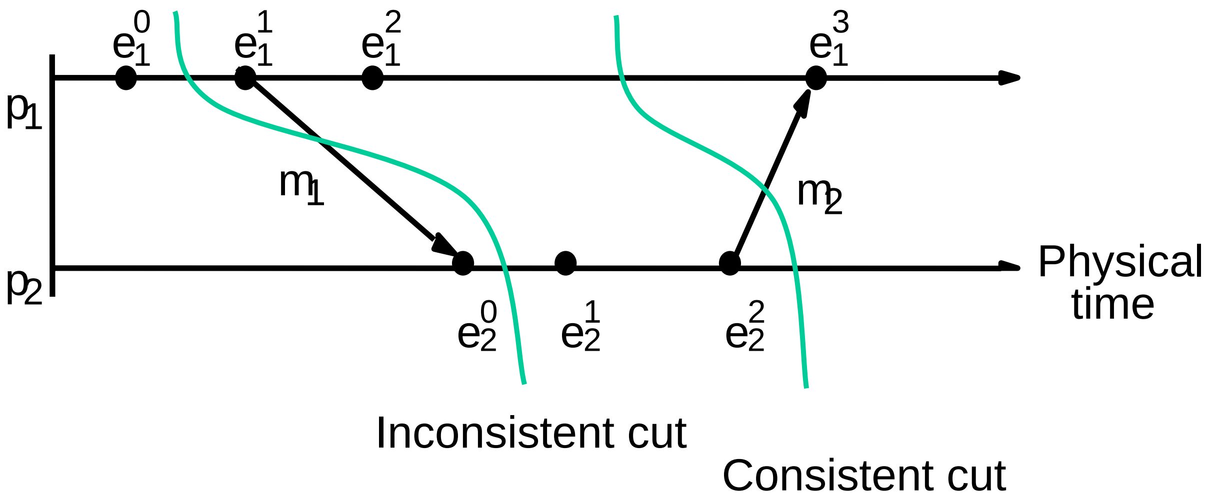
Global states

- Hard to obtain a global state of distributed system
  - consists of states of multiple processes and channel states
  - concurrency, independent failure, **no global clock**
  - only by message passing → the state of each process (data and variables), is private information.
- If all processes do agree on the time, the state recorded at processes is a global state of the system.
  - But, no perfect clock synchronization
- How to obtain a **meaningful** global state from local states recorded at different real times?
- Some definitions
  - A **history** h_i of process p_i is a series of events happened at process p_i.
  - The **state** of process p_i just before the k-th event is denoted by s_i^k.
  - A **global history** H is the union of the N process histories.
  - A **cut** is a subset of its global history that is a union of prefixes of process histories.
  - The global state of a cut is the set of states $S=(s_1,\ldots,s_N)$, where s_i is the state of p_i just after the last event of p_i in the cut.

5

Cut

- A cut C divides all events to P_C (those happened before C) and F_C (future events)
- A Cut C is **consistent** if there is no message whose sending event is in F_C and whose receiving event is in P_C
  - Inconsistent cut: an 'effect' without a 'cause'
  - it's enough to check message sending and receiving events in the cut
  - Consistent/inconsistent states.



Global states

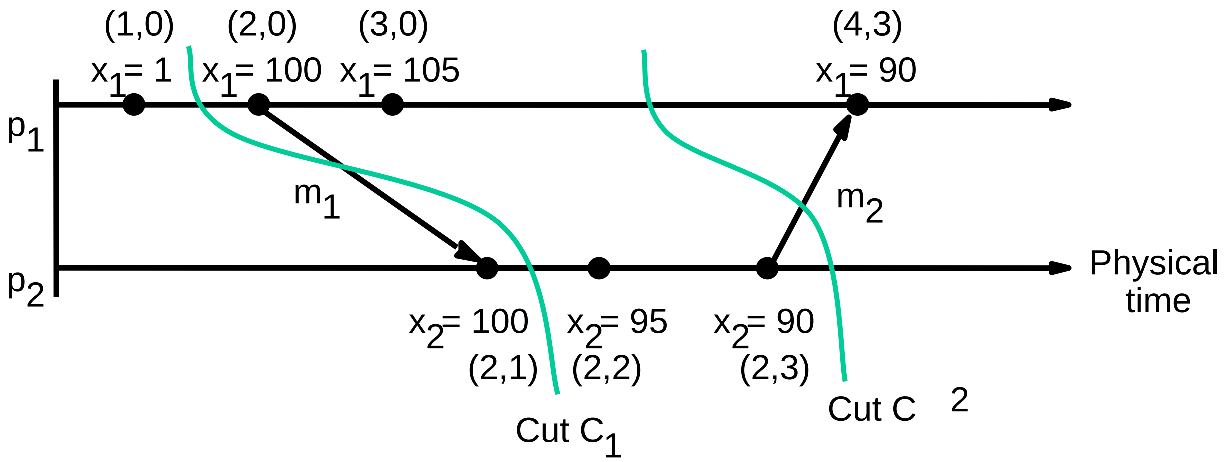
- Consider the execution of a distributed system as a sequence of transitions between global states of the system.
- In each transition, exact one event happens at some single process in the system.
  - sending message event, receiving message event, or an internal event
- A **run** is an ordering of the events that satisfies the happened-before relation in one process.
- A **consistent run** is an ordering of the events that satisfies all the happened-before relations.
- Clearly, not all runs pass through consistent global states, but all consistent runs do pass through consistent global states.
- We say that a state S' is **reachable** from a state S if there exists a consistent run from S to S'.
  - May exist more than one consistent run, since the ordering from happened-before relation is a partial order.

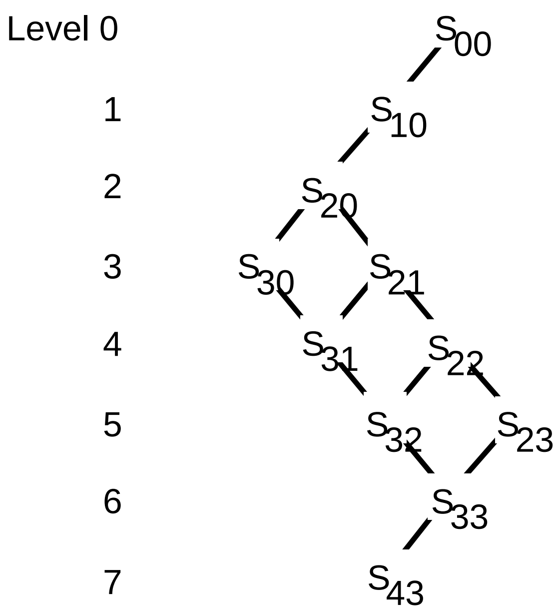
Global states of distributed systems

- ‘Snapshot’ algorithm, [Chandy & Lamport 1985]: to determine global states of distributed systems.
  - It’s a distributed algorithm to collect local states.
- Another approach is to collect local states in a centralized fashion.
  - processes → Monitor process.
- Example: distributed debugging
  - Evaluating possibly predicate X, evaluating definitely predicate X’.
- Collecting the state
  - state messages
  - two simple ways to reduce the state-message traffic to the monitor.
    - predicate may depend on only partial part of the processes’ states
    - send their state when the predicate may be changed
- Obtaining consistent global states
  - The ordering of states, from the vector timestamps of the state messages.
    - Since different message latencies, not depend on the ordering of received state messages.

Check if one global state is consistent

- Let $S=(s_1,\ldots,s_N)$ be a global state received from the state messages.
- Let $V(s_i)$ be the vector timestamp of state s_i, received from p_i.
- S is a consistent global state if and only if: $V(s_i)[i] >= V(s_j)[i]$ for $i,j=1,\ldots,N$.





Sij = global state after *i* events at process 1 and *j* events at process 2

9

Algorithms to evaluate *possibly X* and *definitely* X'

- To evaluate "possibly": evaluate the value at each reachable node from initial state. Stops when it evaluates to True.
- To evaluate "definitely": find a set of states such that all consistent runs must pass (a separator in graph theory), then the evaluation value of each state in this set is true.

1. *Evaluating possibly ϕ for global history H of N processes*

```
L := 0;
States := { (s_1^0, s_2^0, ..., s_N^0) };
while (φ(S) = False for all S ∈ States)
    L := L + 1;
    Reachable := { S′: S′ reachable in H from some S ∈ States ∧ level(S′) = L};
    States := Reachable
end while
output "possibly φ";
```

2. *Evaluating definitely ϕ for global history H of N processes*

```
L := 0;
if (φ(s_1^0, s_2^0, ..., s_N^0)) then States := {} else States := { (s_1^0, s_2^0, ..., s_N^0) };
while (States ≠ {})
    L := L + 1;
    Reachable := { S′: S′ reachable in H from some S ∈ States ∧ level(S′) = L};
    States := { S ∈ Reachable: φ(S) = False}
end while
output "definitely φ";
```

1
0

Transactions and concurrency control

- The goal of transactions
  - the objects managed by a server must remain in a consistent state
    - when they are accessed by multiple transactions and
    - in the presence of server crashes
- Recoverable objects
  - can be recovered after their server crashes
  - objects are stored in permanent storage
- A **transaction** is a set of operations on objects, specified by a client, to be performed as a unit operation at the server side.
  - a unit operation for other clients
- Chapter 13 focuses on the issues for a transaction at a single server. Chapter 14 discusses issues for transactions that involve several servers.

Bank example

- Operations of the *Account* interface

deposit(amount)
deposit amount in the account
withdraw(amount)
withdraw amount from the account
getBalance() -> amount
return the balance of the account
setBalance(amount)
set the balance of the account to amount

- Simple synchronization (without transactions)
  - multiple threads → several client operations concurrently → inconsistent states
  - objects should be designed for safe concurrent access
  - Synchronized method in Java: each time, only one thread can be used to access an object.
  - *E.g. public synchronized void deposit(int amount) throws RemoteException*
  - ***atomic operations*** are free from interference from concurrent operations in other threads.
  - use any available mutual exclusion mechanism (e.g. mutex)
- Failure model: disks, servers, communication
  - Stable storage: atomic write operation, by replicating
  - Stable processor: using stable storage to recover objects
  - Reliable RPC

Transactions

- Transactions originally come from database management systems.
- Transactional file servers were built in the 1980s
- Transactions on distributed objects late 1980s and 1990s.
- From client's viewpoint, a transaction=single step.
- A client's banking transaction

Transaction T:
a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);

- Atomicity of transactions
  - they are not affected by operations being performed for other concurrent clients (called "isolation");
  - either all of the operations are completed successfully or they have no effect at all in the presence of server crashes (called "all or nothing" effect)

Transactions

- Isolation
  - Synchronize operations at server side
  - One way: perform the transaction serially
    - not suitable for servers whose resources are shared by multiple users
    - The aim for any server that supports transactions is to maximize concurrency.
  - concurrency control
- "All or nothing"
  - the objects must be recoverable
  - When a server acknowledges the completion of a client's transaction, record the objects in permanent storage
- How to add transaction capabilities to servers?

  openTransaction() -> trans;
  closeTransaction(trans) -> (commit, abort);
  abortTransaction(trans);

- Each transaction is created and managed by a coordinator
- A transaction: cooperation between a client program, some recoverable objects, and a coordinator.
- invokes "openTransaction" to introduce a new transaction (TID: transaction identifier), e.g. deposit(trans, amount)
- invokes "closeTransaction" to indicate its end.

Concurrency control

- Two well-known problems of concurrent transactions
- Assume that the operations *deposit*, *withdraw*, *getBalance* and *setBalance* are *synchronized* operations (atomic).
- 'lost update' problem
  - two transactions both read the old value of a variable and use it to calculate a new value
- 'Inconsistent retrieval' problem
  - a retrieval transaction runs concurrently with an update transaction.
- There is no such problem if transactions are done one at a time
- Serially equivalent interleaving
  - An interleaving of the operations of transaction such that its effect is the same as if the transactions are performed one at a time
  - avoid these problems
- the same effect means
  - the read operations return the same values
  - the instance variables of the objects have the same values at the end

Recoverability from aborts

- **Dirty reads**
  - caused by the interaction between a read operation in one transaction U and an earlier write operation in another transaction T on the same object, and after U is committed, T is aborted.
  - a transaction that committed with a 'dirty read' is not recoverable
  - Fix: delays the commit operation
  - **Cascading aborts**: the aborting of the transactions may cause other transactions to be aborted.
  - To avoid it, transactions are only allowed to read objects that were written by committed transactions.
  - Avoidance of cascading aborts is a stronger condition than recoverability
- **Premature writes**
  - caused by the interaction between 'write' operations on the same object, in different transactions.
- **Strict executions of transactions**
  - to avoid both 'dirty reads' and 'premature writes'.
    - delay both read and write operations
  - executions of transactions are called **strict** if both *read* and *write* operations on an object are delayed until all transactions that previously wrote that object have either committed or aborted.

Concurrency control approaches

- serialize transactions in their access to objects, to achieve 'isolation'
- Locking
  - Used by most practical systems
  - set a lock on each object just before it is accessed, and remove these locks when the transaction has completed.
  - The lock is labeled with the transaction ID.
  - Only the corresponding transaction can access that locked object. Other transaction may wait or in some cases, share the lock (such as sharing read locks).
  - Problem: deadlock
- optimistic concurrency control
  - a transaction proceeds until it asks to commit
  - before it's allowed to commit, the server will check if this transaction has some performed operations on objects that conflict with the operations of other concurrent transactions.
- timestamp ordering
  - For each object, the server records the most recent time of reading and writing operation on it;
  - For each operation, the timestamp of the transaction is compared with the timestamp of the object to determine whether the operation can be done, delayed or rejected.