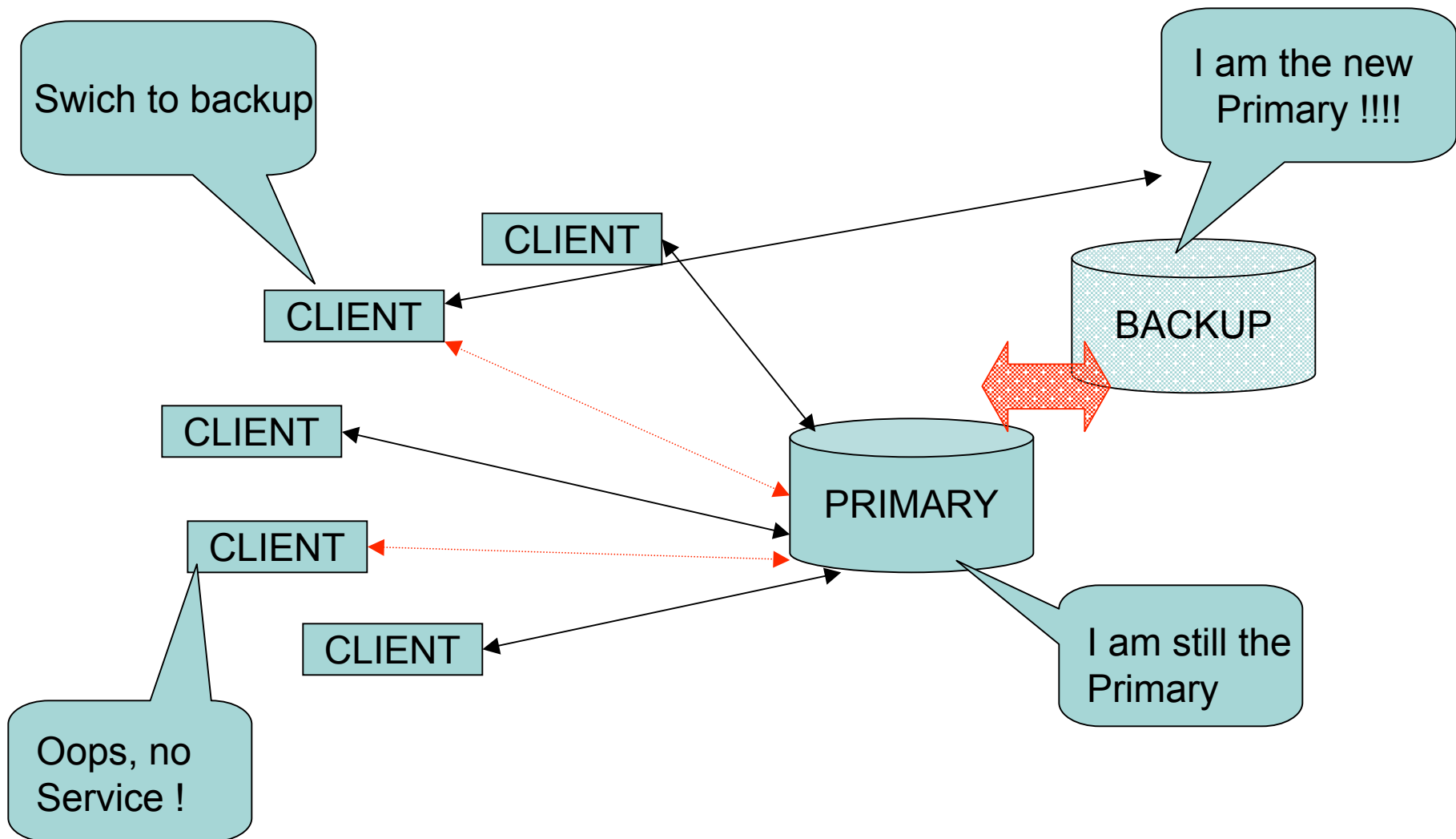


CS603: Distributed Systems

Lecture 4: Overcoming failures in distributed systems

Things go **very** wrong...



Outline

Processes do not have the same ‘view’ of the system, some perceived ‘primary down’, some perceived ‘primary up’

- Order of events in distributed systems
- Failure detection
- Membership



THE BAD NEWS

- We can not detect failures in a trustworthy, consistent manner
- We can not reach a state of “common knowledge” concerning something not agreed upon in the first place
- We can not guarantee agreement on things (election of a leader, update to a replicated variable) in a way certain to tolerate failures

CAN WE DO ANYTHING?

System Model Dimensions

- Non-deterministic processes
- Communication is through messages
- Network can be a clique or a graph, not every machine can connect to every other machine
- Network packets can be lost, duplicated, delivered very late or out of order, spied upon, replayed, corrupted, source or destination address can lie
- Communication can be authenticated or not
- Execution model can be
 - Asynchronous: no synchronized clocks or time-bounds on message delays.
 - Synchronous: execution is partitioned in rounds, all messages send in a round are delivered in that round

Execution, Configuration, Events

- Set of processes p_i , each process with a state s_i
- **Configuration C_t** : set of state of each process at some moment
- **Events**: send and deliver, events can change the state at a process
- **Execution**: sequence of configuration and events

Safety and Liveness

- **Safety**: a condition that must hold in every finite prefix of a sequence (from an execution)

“nothing bad happens”

- **Liveness**: a condition that must hold a certain number of times

“something good happens”

Ordering of Events

- Order of events, particularly causality helps in reasoning or analyzing a system
- Single process: follow the sequence of events, each event has a timestamp and the causality relation between events is given by time
- Distributed processes: many events generated at different processes, how to order events?
- Time is essential for ordering events in a distributed system
 - Physical time: local clock; global clock
 - Logical time: partial ordering, total ordering

Using Real Clocks

- Global clock: processes have access to such a central global clock, each event will carry a timestamp
- Local clock: each process has its own clock
 - What if the clocks are not synchronized
 - What if events happened at the same time?

Synchronizing Physical Clocks

- **External synchronization:** Consider the source S and the synchronization bound $B > 0$, then none of the clocks drift with more than B from S , at any time
- **Internal synchronization:** Consider the synchronization bound $B > 0$, then at any time, the difference between any two clocks is within B

Algorithms for Clock Synchronization

Cristian's Algorithm

- Uses a time server to synchronize clocks
- Clients ask the time server for time and adjust their clock, based on the server's response
- RTT estimated by the client: $T_{\text{send}} - T_{\text{receive}}$

$$T_{\text{client}} = T_{\text{server}} + (\text{RTT} / 2)$$

Algorithms for Clock Synchronization

Berkeley Algorithm

- Uses a elected master to synchronize
- The master obtains the local time from all machines, adjusts times received for RTT & latency, averages times, and tells each machine how to adjust.
- In some systems multiple time servers are used.
- Time is more accurate, but still drifts

From Theory to Practice

- What does it take to synchronize many computers across several networks?
- NTP
- How does NTP protocols relate to the protocols described before?
- A good source is:
 - www.eecis.udel.edu/~mills/database/brief/overview/overview.ppt

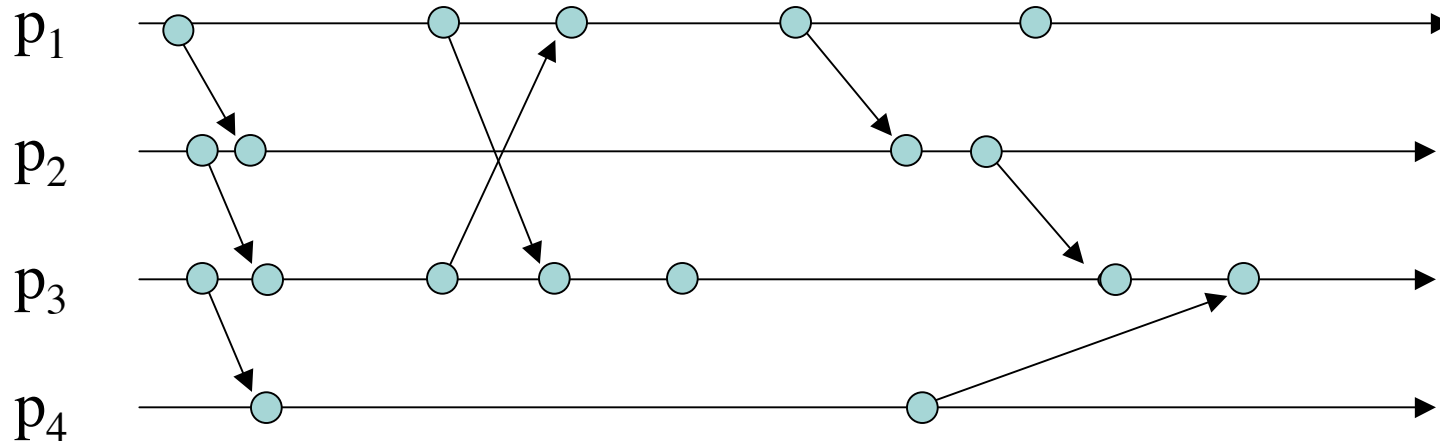
From Theory to Practice

- Consider a sensor network
- Communication is expensive (even if a node does not have any data to receive, just listening consumes power)
- Power is limited
- Synchronization is important because
 - Nodes can sleep and save battery
 - Communication may be avoided

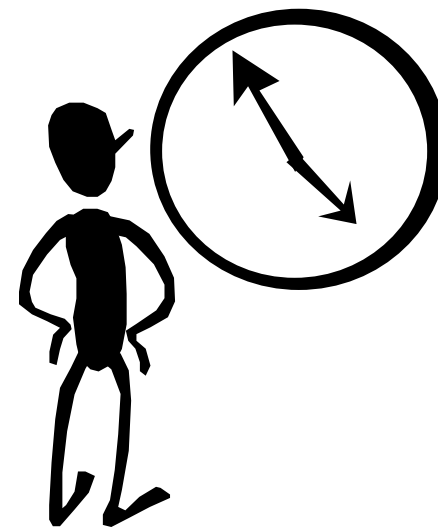
From Physical Clocks to Logical Clocks

- Synchronized clocks are great if we have them, but
- Why do we need the time anyway?
- In distributed systems we care about ‘what happened before what’

“HAPPENED BEFORE”



- If events a and b take place at the same process and a **occurs before** b
 $a \sqsubset b$
- If a is send event at p_1 and b is deliver event at p_2 , $p_1 \neq p_2$
 $a \sqsubset b$
- If $a \sqsubset b$ and $b \sqsubset c$ then $a \sqsubset c$



Logical Clocks: Lamport Clocks

- Each process maintains his own clock C_i (a counter)
- Clock Condition: for any events a and b in process p_i
if $a \square b$ then $C_i(a) < C_i(b)$
- Implementation:
 - each process p_i increments C_i between any successive events
 - on send event a , attach to the message m local clock
 $T_m = C_i(a)$
 - on receive of message m process P_k sets C_k to
 $C_k = \max(C_k, T_m) + 1$

Lamport Clocks: Total Order

- Logical Clocks only provide partial order
- Create Total Order by breaking the ties
- Example to break ties, use process identifiers, have an order on process identifiers:

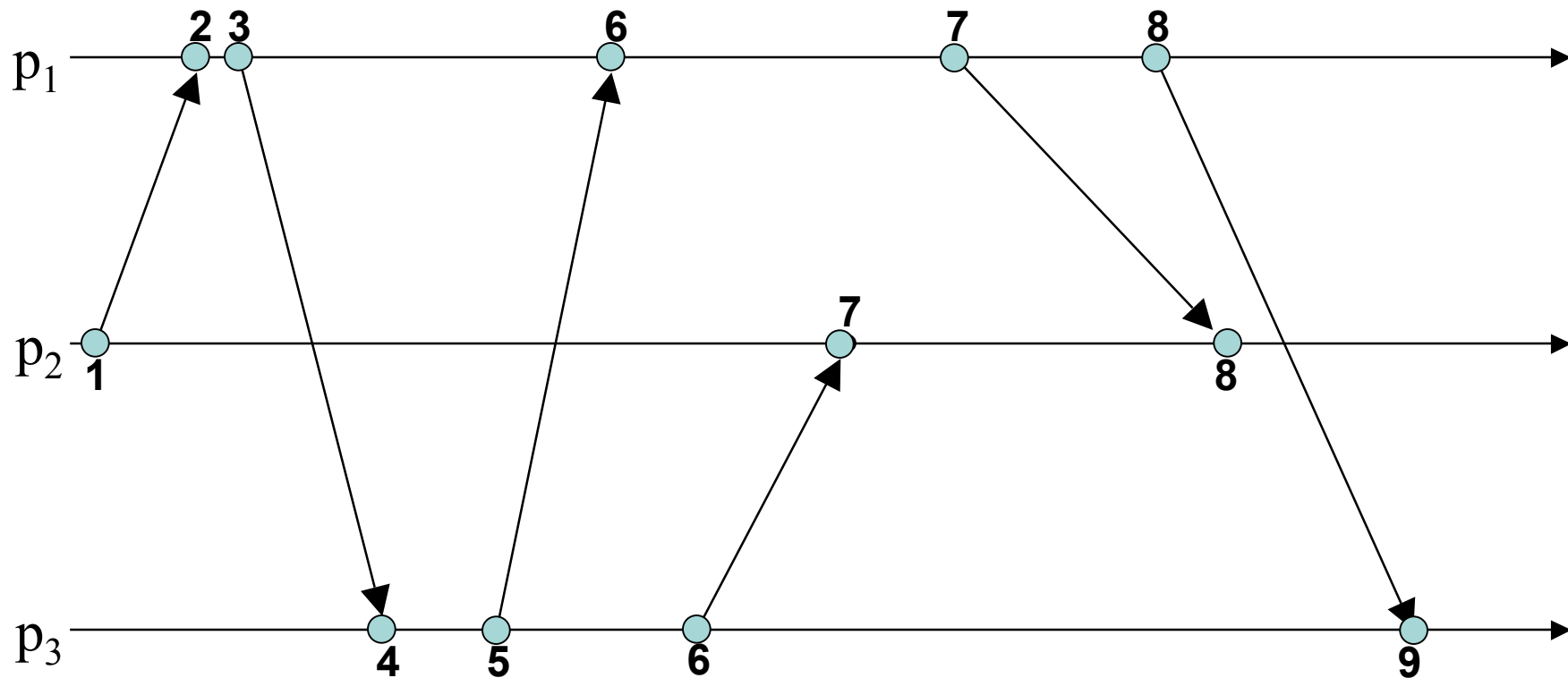
If a is event in p_i and b is event in p_j then

$a \leq b$ iff

$C_i(a) < C_j(b)$ or

$C_i(a) = C_j(b)$ and $p_i < p_j$

Lamport Clocks: Example



Reminder: Partial and Total Order

- **Definition:** A relation R over a set S is a **partial order** iff for each $a, b,$ and c in S :
 - aRa (reflexive).
 - $aRb \wedge bRa \wedge a = b$ (antisymmetric).
 - $aRb \wedge bRc \wedge aRc$ (transitive).
- **Definition:** A relation R over a set S is **total order** if for each distinct a and b in S , R is antisymmetric, transitive and either aRb or bRa .

Concurrent Events

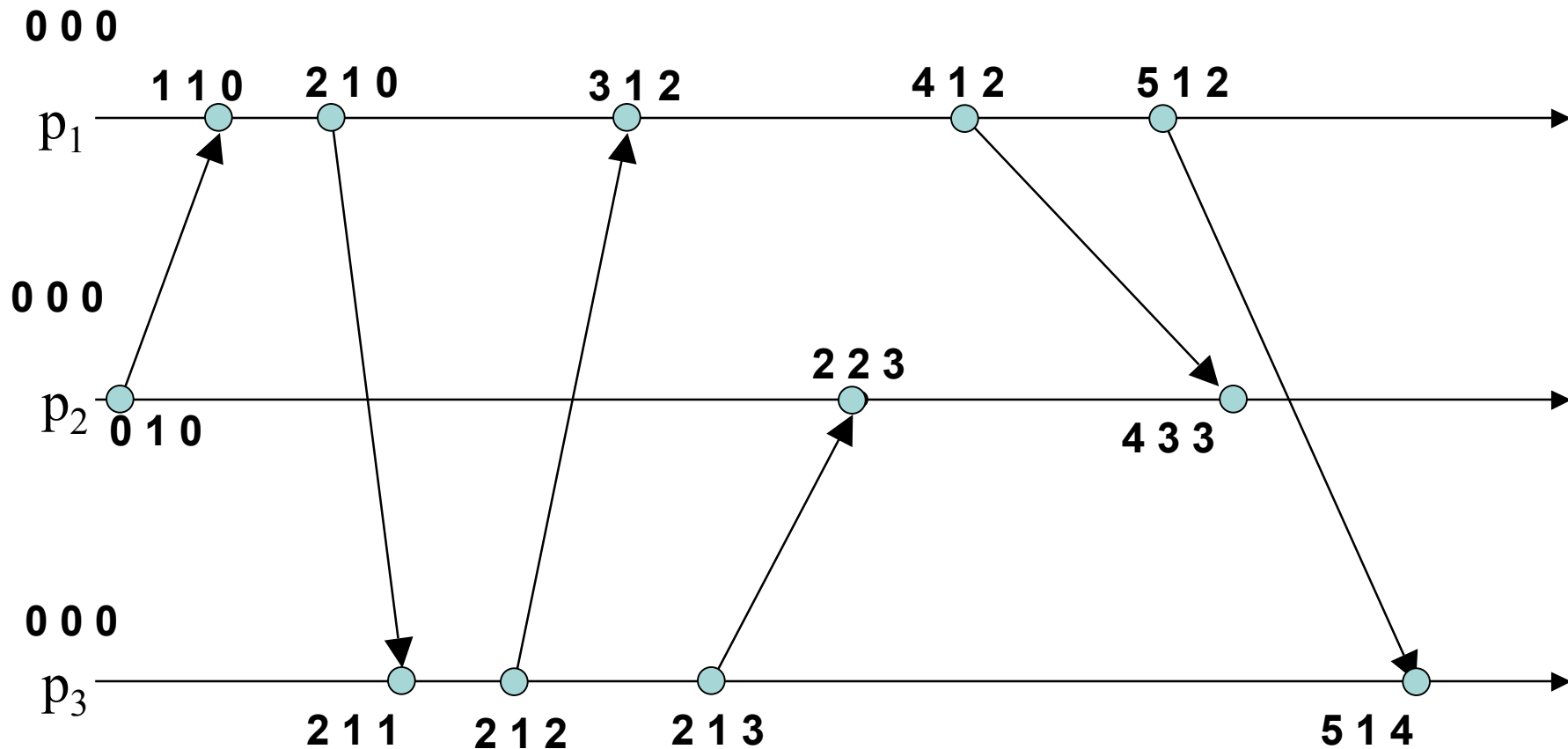
- Concurrent events:
If $a \not\prec b$ and $b \not\prec a$ then
a and b are concurrent
- Logical clocks assigns order to events that are causally independent, in other words events that are causally independent appear as if they happened in a certain order
- We need a 'vector time'

Vector Clocks

- Each process maintains a vector C_i initially $[0, 0, \dots, 0]$.
- When p_i executes an event, it increments $C_i[i]$
- When p_i sends a message m to p_j , it piggybacks C_i on m .
- When p_i receives a message m ,
 $\forall j: 1 \leq j \leq n, j \neq i: C_i[j] = \max(C_i[j], m.C[j])$
 $C_i[i] = C_i[i] + 1.$



Vector Clocks: Example



How to Order with Vector Clocks

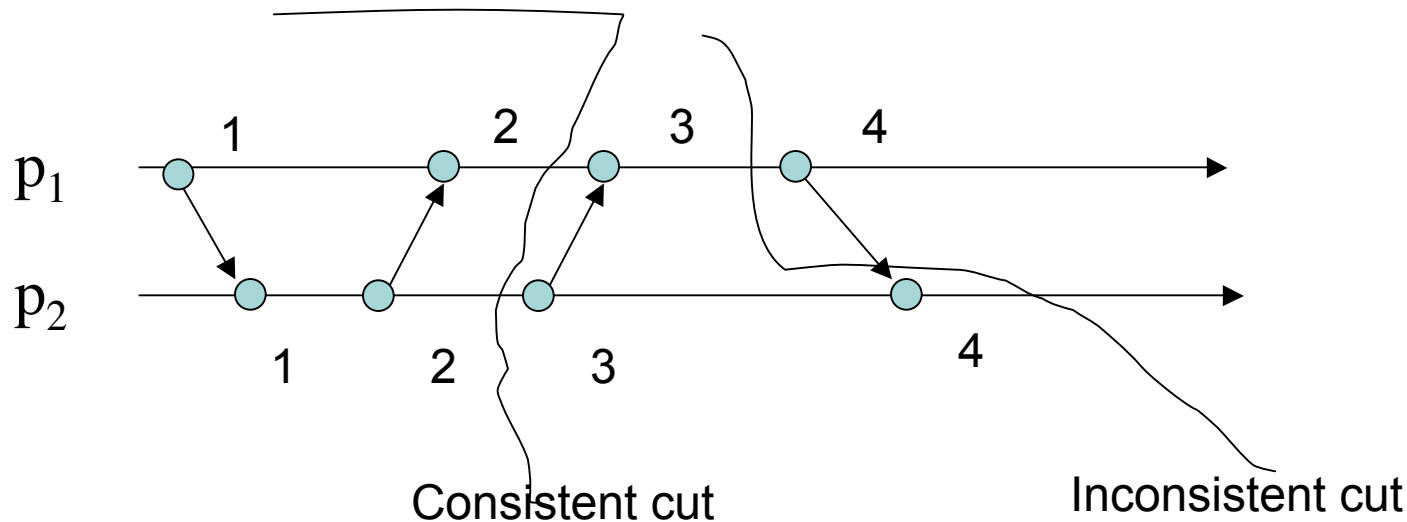
- Given two events a and b , $a \sqsubseteq b$ if and only if
- b has a counter value for the process in which a occurred greater than or equal to the value of that process at event a inclusive, and
- a has a counter value for the process in which b occurred strictly less than the value of that process at event b inclusive.

$$b \sqsubseteq a \equiv \forall i: 1 \leq i \leq n: V(b)[i] \geq V(a)[i]$$
$$\quad \exists i: 1 \leq i \leq n: V(b)[i] < V(a)[i]$$

$$b \parallel a \equiv \forall i: 1 \leq i \leq n: V(b)[i] < V(a)[i]$$
$$\quad \exists i: 1 \leq i \leq n: V(a)[i] < V(b)[i]$$

Using Ordering...: Consistent Cuts

- There is no outside observer that can look at the system and detect problems, for example a deadlock
- Cut: n -vector (k_0, \dots, k_{n-1}) of positive integers
- Consistent cut: if for all i, j , $(k_i + 1)$ event at process p_i did not 'happened before' k_j event at p_j



Detecting failures

- **Impossibility result:** it is impossible to design an asynchronous fault-tolerant consensus algorithm, even when only one process can crash. (FLP85)
- **Proof Idea:** It is shown how an infinite sequence of events can be constructed such that the algorithm never terminates (stays indecisive forever).
- **The impossibility comes from the fact that in an asynchronous system, it is impossible to distinguish between a faulty-process and a slow process.**

Failure Detectors as an Abstraction

- **Failure detector**: distributed oracle that makes guesses about process failures
- **Accuracy**: the failure detector makes no mistakes when labeling processes as faulty.
- **Completeness**: the failure detector “eventually” (after some time) suspects every process that actually crashes.
- Classified based on their properties
- Used to solve different distributed systems problems

Completeness

- **Strong Completeness:** There is a time after which every process that crashes is suspected by **EVERY** correct process.
- **Weak Completeness:** There is a time after which every process that crashes is permanently suspected by **SOME** correct process.

Accuracy

- **Strong Accuracy**: No process is suspected before it crashes.
- **Weak Accuracy**: Some correct process is never suspected. (at least one correct process is never suspected)
- **Eventual Strong Accuracy**: *There is a time* after which correct processes are not suspected by any correct process.
- **Eventual Weak Accuracy**: *There is a time* after which **some** correct process is never suspected by any correct process.

Perfect Failure Detector

- A perfect failure detector has strong accuracy and strong completeness
- THIS IS AN ABSTRACTION
- IT IS IMPOSSIBLE TO HAVE A PERFECT FAILURE DETECTOR
- We have to live with ... unreliable failures detectors...

Unreliable Failure Detectors

- Unreliable failure detectors can make mistakes
- A process is suspected that it was faulty, that can be true or false, if false the list of alive processes is modified.
- Failure detectors can add/remove processes from the list of suspects; **different processes have different lists.**
- The assumptions are that:
 - After a while the network becomes stable so the failure detector does not make mistakes anymore.
 - In the unstable period, the failure detector can make mistakes.

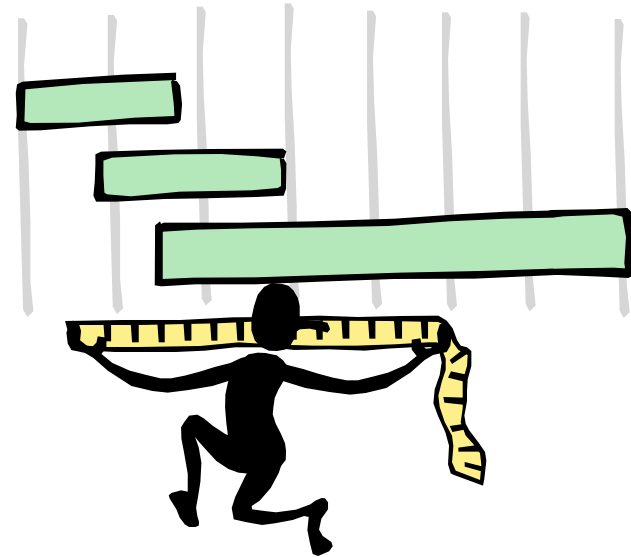
Failure Detection Implementation

- **Push:** processes keep sending heartbeats “I am alive” to the monitor. If no message is received for awhile from some process, that process is suspected as being dead.
- **Pull:** monitor asks the processes “Are you alive?”, and process will respond “Yes, I am alive”. If no answer is received from some process, the process is suspected as being dead.
- What are advantages and disadvantages of these two models?



Metrics for Failure Detectors

- Detection time
- Mistake recurrence time
- Mistake duration
- Average mistake rate
- Query accuracy probability
- Good period duration
- Network load



Failure Detectors Implementation

- Every process must know about who failed
- How to disseminate the information
- How about if not every node can communicate directly with another node?

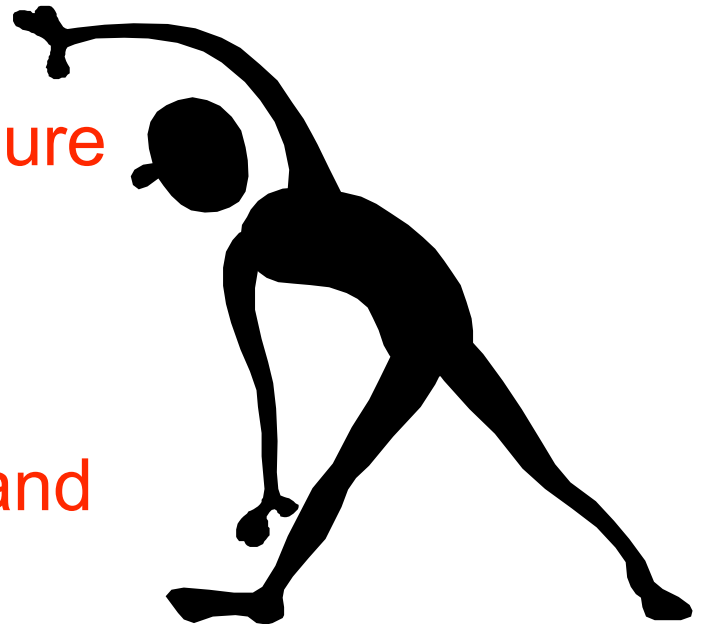
Exercise

For next lecture, everybody looks on the web for a paper about

- implementation/design of failure detectors
- synchronization issues in wireless sensor networks

reads (at least) the abstract and tells us about what he learnt

**DO NOT ASK YOUR FRIEND
WHAT PAPER DID HE FOUND**



REQUIRED READING

- Leslie Lamport for "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, July 1978, 21(7):558-565.
- Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson for "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, April 1985, 32(2):374-382.
- Unreliable Failure Detectors for Reliable Distributed Systems, T. Chandra and S. Toueg. 1996.

