



UPPSALA
UNIVERSITET

Synchronization and global states





The need for synchronization...

- Time is relative!
- Can depend upon:
 - Speed
 - Computer hardware
 - Distance
 - Day of week
 - Time of year
 - ...



"I'M AFRAID I DON'T UNDERSTAND ALL THE REPORTS OF OUR UPGRADE HAVING A DELAYED RELEASE DATE, UNLESS, ... WAIT A MINUTE - HOW MANY PEOPLE HERE DIDN'T KNOW I WAS SPEAKING IN DOG-MONTHS?"

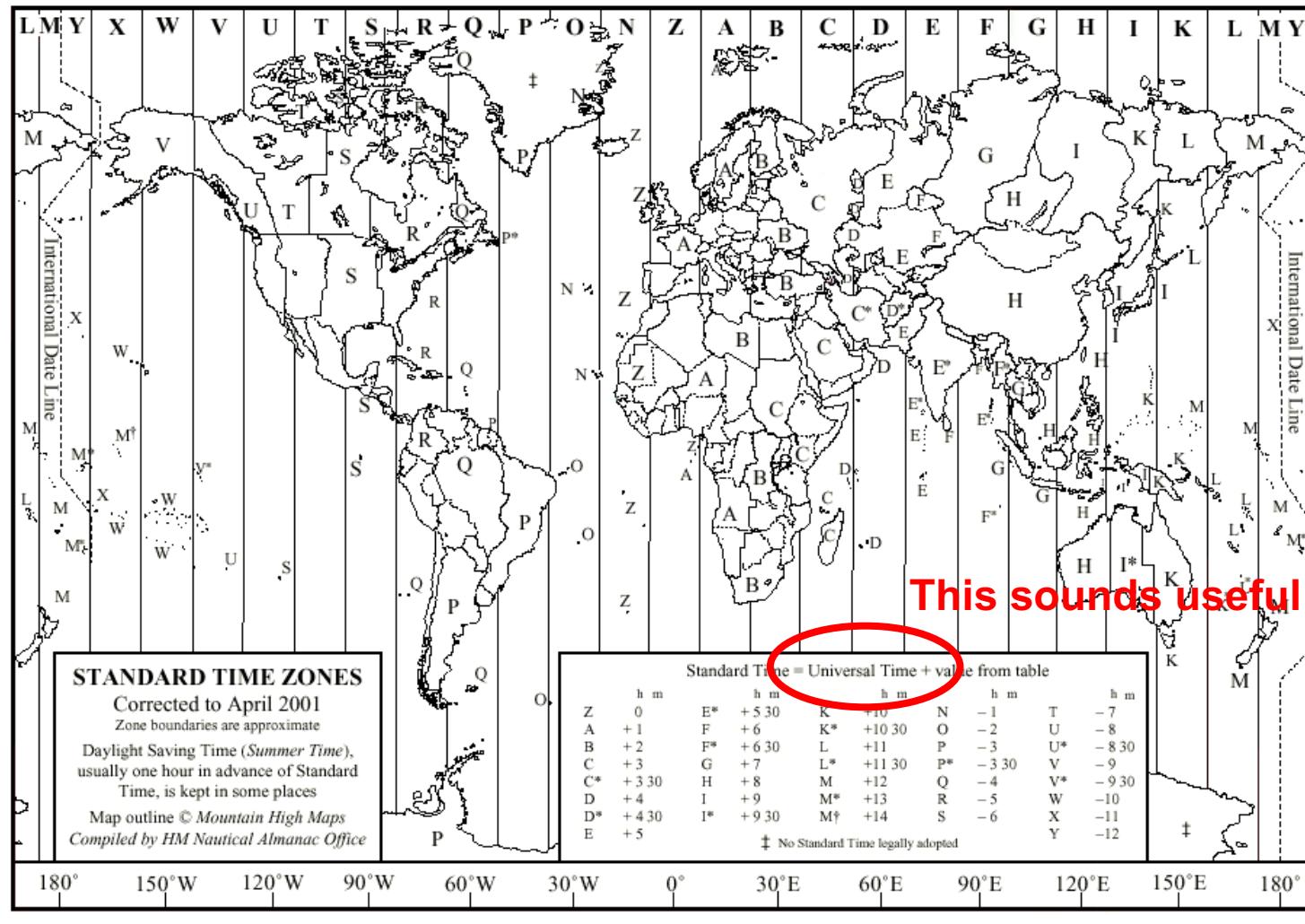


...in distributed systems

- Need to define an *order* of events...
- ... that can occur at different nodes...
- ... that most likely use have different clocks...
- ... that show different times
- Clock drifting
 - Due to physical variations in oscillators



So, what time is it?



This sounds useful!



Coordinated Universal Time (UTC)

- One clock to rule them all...
 - Replicated
 - Broadcasted via radio
- So it's easy!
 - Tune in to UTC
 - Update your clock often

...well, it's not *that* easy...





Clock Synchronization

- External synchronization
 - Synchronize against UTC clock
- Internal synchronization
 - Synchronize among set of nodes
 - Exact UTC-based time not *that* important
- Hard in an asynchronous system
 - Delay to UTC server not well known
 - Delays may be asymmetric



Different time standards

- International atomic time (TAI)
 - No leap seconds
- Coordinated universal time (UTC)
 - Adjusts over time
- GPS time (GPST)
 - Offset to TAI, 19 seconds behind



Network Time Protocol (NTP)

- A time service for the Internet
 - synchronizes clients to UTC
 - Servers divided into different *strata*

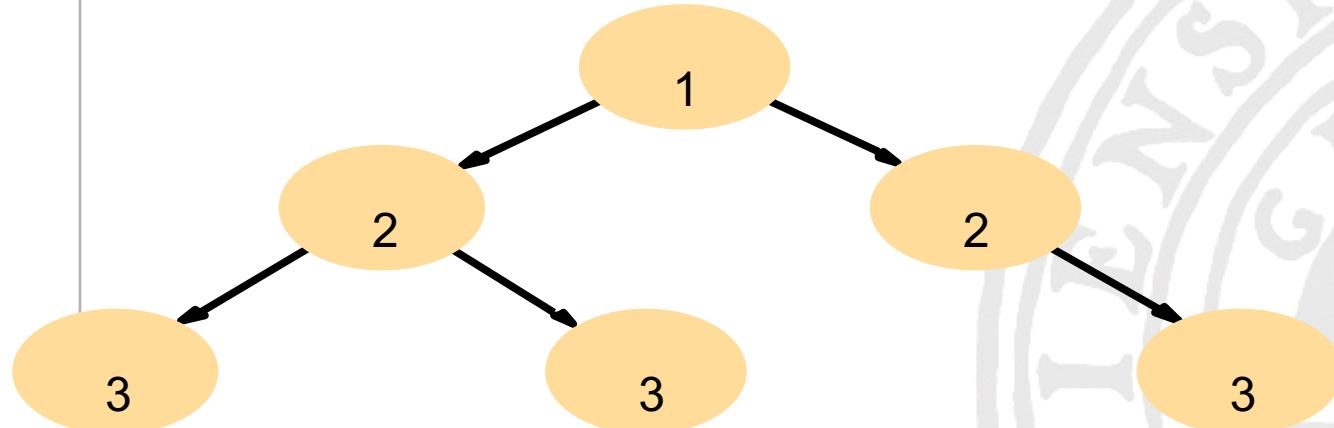


Figure 14.3



UPPSALA
UNIVERSITET

Coordination

© Original Artist
Reproduction rights obtainable from
www.CartoonStock.com



"Whose turn to talk to Aunt Gertie?"

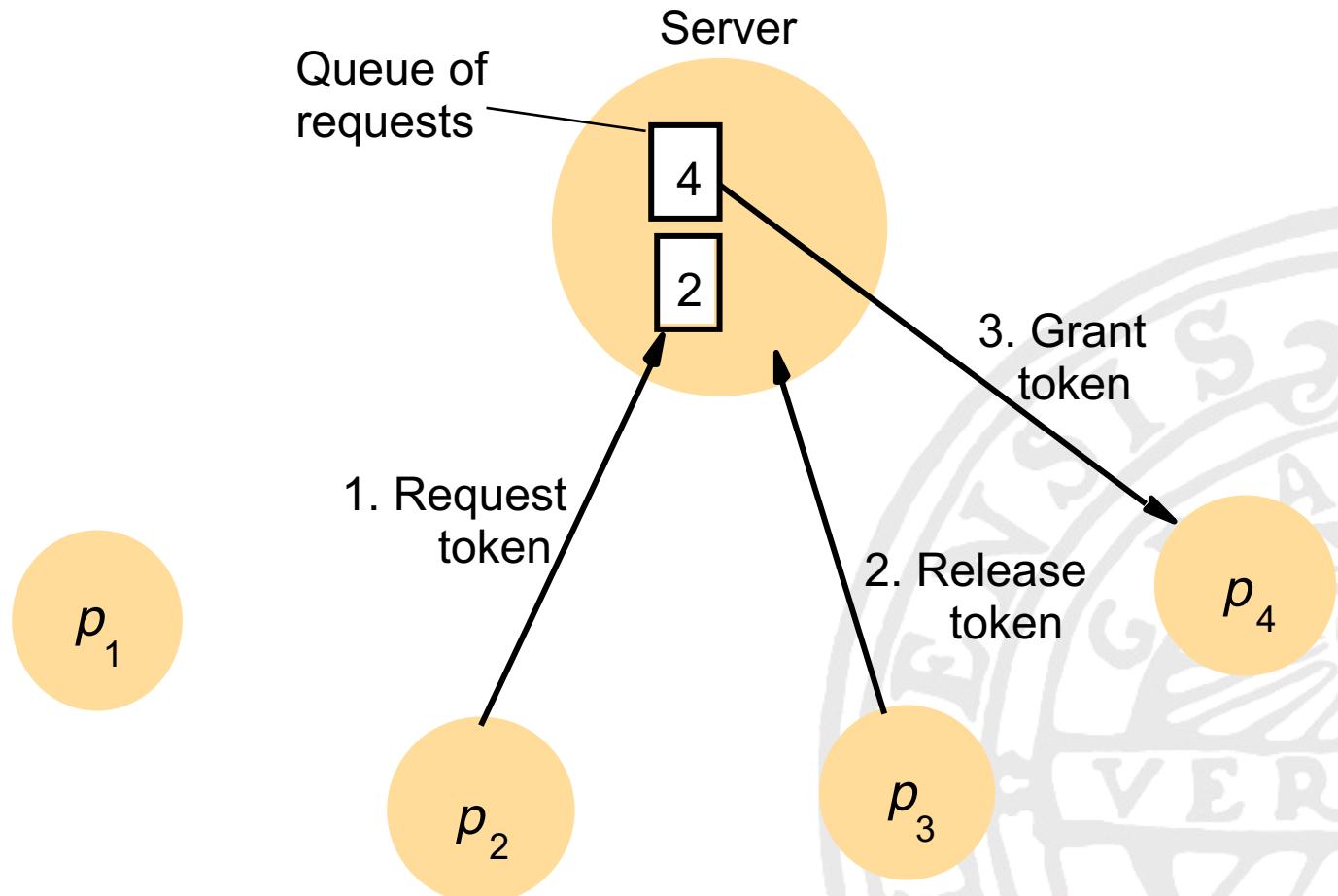


Mutual exclusion

- Critical section
 - Piece of code, shared resource...
 - Only one can use the critical section
- Need for coordination:
 - Who gets access to the critical section?
 - In what order is access granted?
- Evaluation criteria for different solutions
 - Bandwidth overhead (how much work?)
 - Client delay (how long must I wait?)
 - Throughput (clients/time)

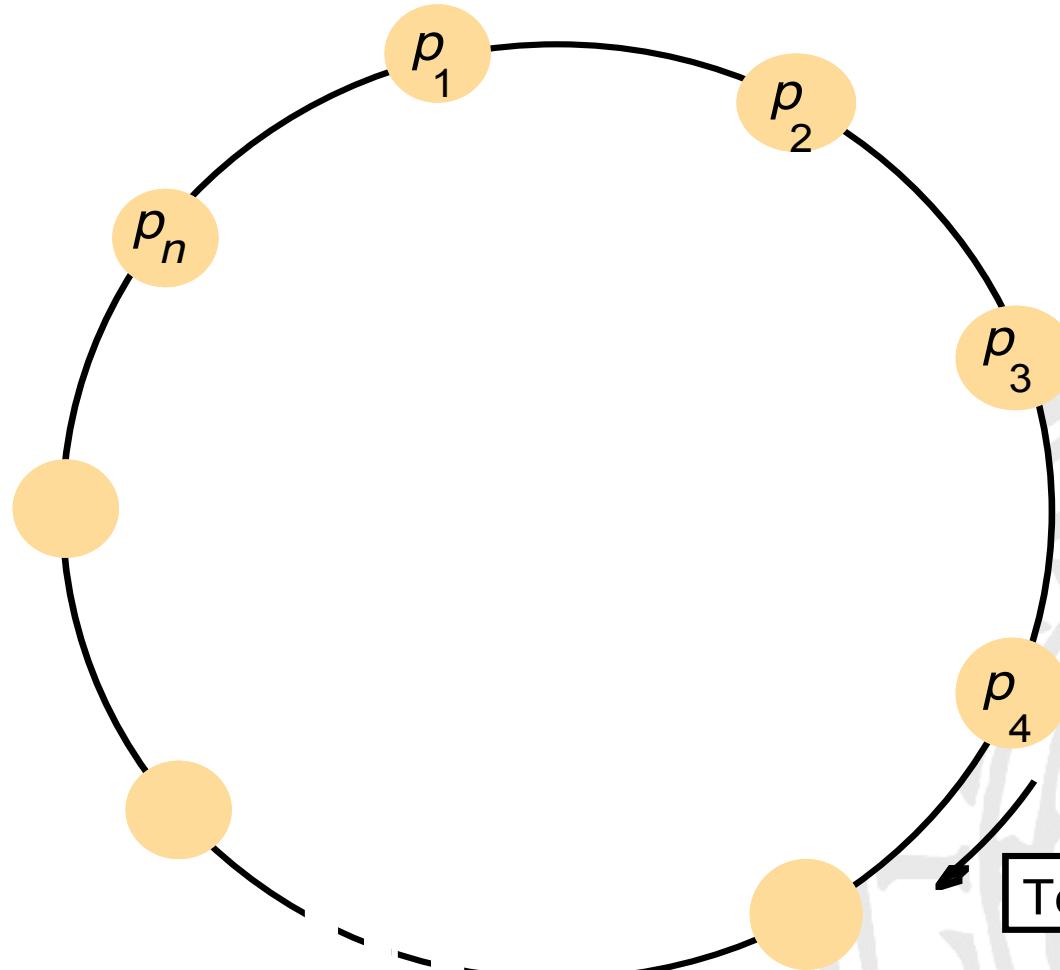


Central server solution



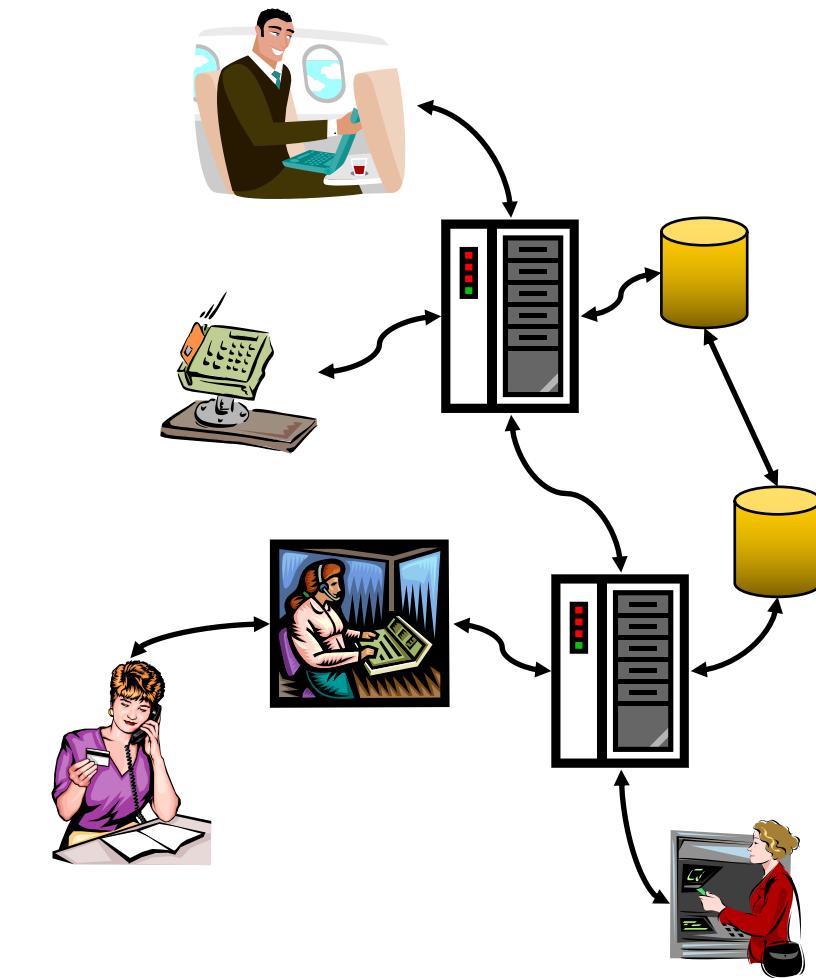


Ring-based solution





Transactions and concurrency



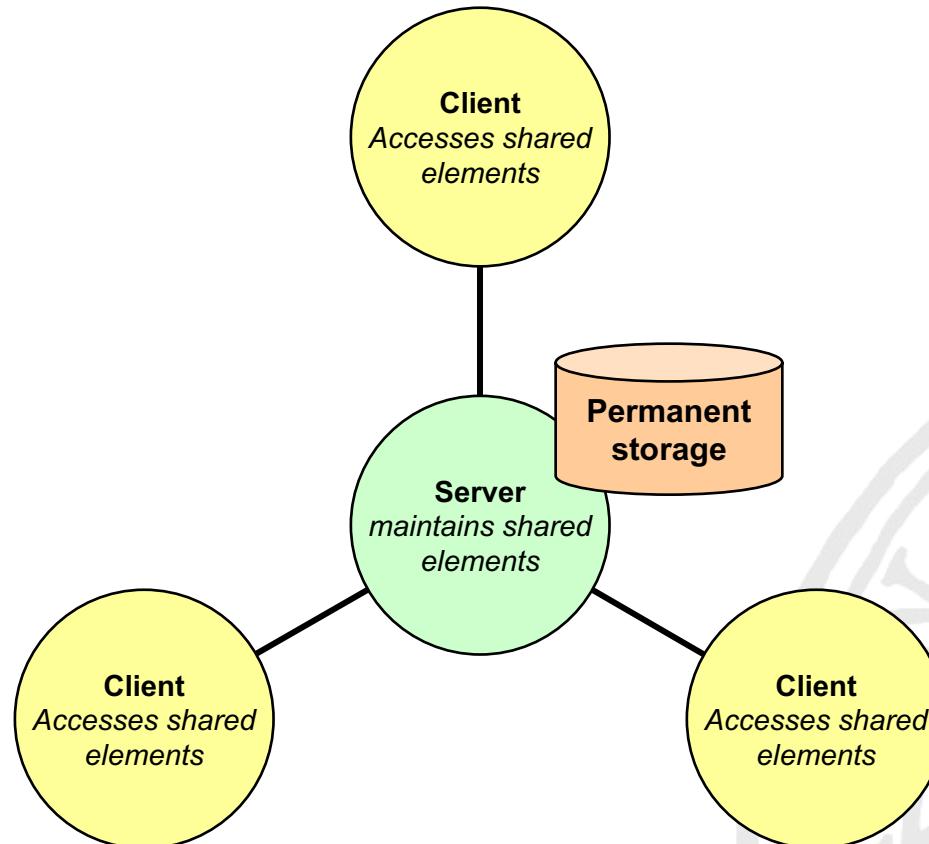


The problem

- Systems with potential many participants
 - Some access services "simultaneously"
- Need to keep consistency
 - In presence of byzantine failures
 - In presence of server crashes
- Ordering of "simultaneous" events
 - The order of events must be clearly defined



Structural assumption





Defining *Transactions*

- An event that occur free of interference
- The event must either:
 - complete successfully, or
 - have no effect on the state of the system
- Assumption:
 - Presence of permanent storage
- Ends by an *abort* or *commit* operation



Desired properties of Transactions

- **Atomicity**
 - "all or nothing"
 - A "complete" transaction requires permanent storage of its effects
- **Consistency**
 - A transaction takes the system from one consistent state to another
- **Isolation**
 - Transactions must not effect eachother
- **Durability**
 - The effect of a transaction must be durable



Serial Equivalence

- Servers often support concurrent transactions
 - To reduce service time for clients
- Concurrent transactions must be *serially equivalent*
 - The outcome must be the same as if they occurred serially



Are these serially equivalent?

T	U
<pre>read(x,a); write(a, x*5); withdraw(b, x/2); <i>commit transaction</i></pre>	<pre>read(x,a); write(a, x * 5); withdraw(c,x/2); <i>commit transaction</i></pre>

NO!

When T writes to a , the value of x is no longer the same as a !



Are these serially equivalent?

T	U
<pre>read(x,a); write(a, x*5); withdraw(b, x/2); <i>commit transaction</i></pre>	<pre>read(x,a); write(a, x * 5); withdraw(c,x/2); <i>commit transaction</i></pre>

YES!

The interleaved *write* operations will not affect the *read* operation of the concurrent transaction



Are these serially equivalent?

T	U
<pre>read(x,a); write(a, x*5); <i>abort transaction;</i></pre>	<pre>read(x,a); write(a, x * 5); <i>commit transaction;</i></pre>

NO!

Since T aborts, the value that U read into x was invalid. However, as U has already committed, this must be dealt with in some way...



What is important to understand

- How to achieve serial equivalence
- How to recover from aborted transactions
- How to use *locks* to protect shared data
- How to extend the commit operation to cope with aborted transactions
- How to protect against *deadlocks*





The problem (revisited)

- Server maintains shared resources
- Clients access resources through *transactions*
 - Atomic operations on shared resources
 - **ACID**: All-or-Nothing, Consistency, Isolation, Durability
- Transactions can occur concurrently
 - To reduce client delay
 - To process more clients / time unit
- Concurrent transactions may access same object
 - Can cause inconsistency
 - Transactions may abort
 - ...



Concurrency control

- Management of shared resources
- Different strategies:
 - Locks (most common approach)
 - Optimistic Concurrency Control
 - Timestamp ordering



Aborted transactions

- Can cause *dirty reads*
 - Someone aborts a transaction where a value has been updated
 - The value was read by someone else
 - This someone else has already committed
- Solution: delay *commit* operation
 - Keep track of changes
 - Delay transactions that work with updated values
 - If a transaction reads a value written by a concurrent transaction that aborts, force transaction to abort
 - New potential problem: *cascading aborts*

T	U
write(i,5); <i>abort</i> ;	j = read(i); write(k,j); <i>commit</i> ;



Locks



- A technique to enable serial equivalence
 - Access to a resource requires a *lock*
 - Locks can be of different types
 - Represent different operations (reads, writes...)
 - Write locks must wait for all read locks to disappear
 - New read lock is not allowed if write lock is pending
 - Shared or private
 - Read locks are normally shared (many can read)
 - Write locks are normally private (only one can write)
 - Special locks for nested transactions:
 - Children and parents inherit locks from each other
 - Children and parents can not use the same lock concurrently
 - Locks are shared among siblings
 - ...which means that siblings must be serially equivalent



Where are locks used?

- Everywhere!
 - File systems
 - Databases
 - Booking systems
 - Financial applications
 - ...
- Sometimes locks are called something else
 - Semaphores, flags, mutex protectors ...



Deadlocks

- Several transactions are waiting for someone else to release a lock
- Example :

T	U
a.deposit(100) <i>write lock A</i>	b.deposit(200) <i>write lock B</i>
b.withdraw(100) <i>waits for U's lock on B</i>	a.withdraw(200) <i>waits for T's lock on A</i> ;

- How do we resolve this?



Optimistic Concurrency Control

- Alternative to locks
 - Assumes that concurrency-related problems are somewhat rare
- Transactions have *tentative* versions of resources
- A *write* operation updates the tentative version
- When a transaction commits:
 - Validation phase that resolve concurrency conflicts
 - Abort transactions retroactively if needed



Timestamping

- Yet another alternative to locks and optimistic concurrency control
- Each transaction gets an unique timestamp
- Operations are validated as they are carried out
- Rules:
 - Writing is granted only if the resource was last read and written by earlier transactions
 - Reading is granted only if the resource was last written by earlier transactions



Summary

- Synchronization is about establishing a shared view, e.g. what time it is
- Coordination is about establishing what must be done (and by whom)
- Transactions are interference-free events
 - Must complete entirely or not happen at all
 - Desired properties: ACID
- Serial equivalence is an interleaving of parallel transactions with the same result as if they were executed after each other in a specific order
- Concurrency control can be solved with locks, OCC, timestamps...