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?"

UPPSALA UNIVERSITET

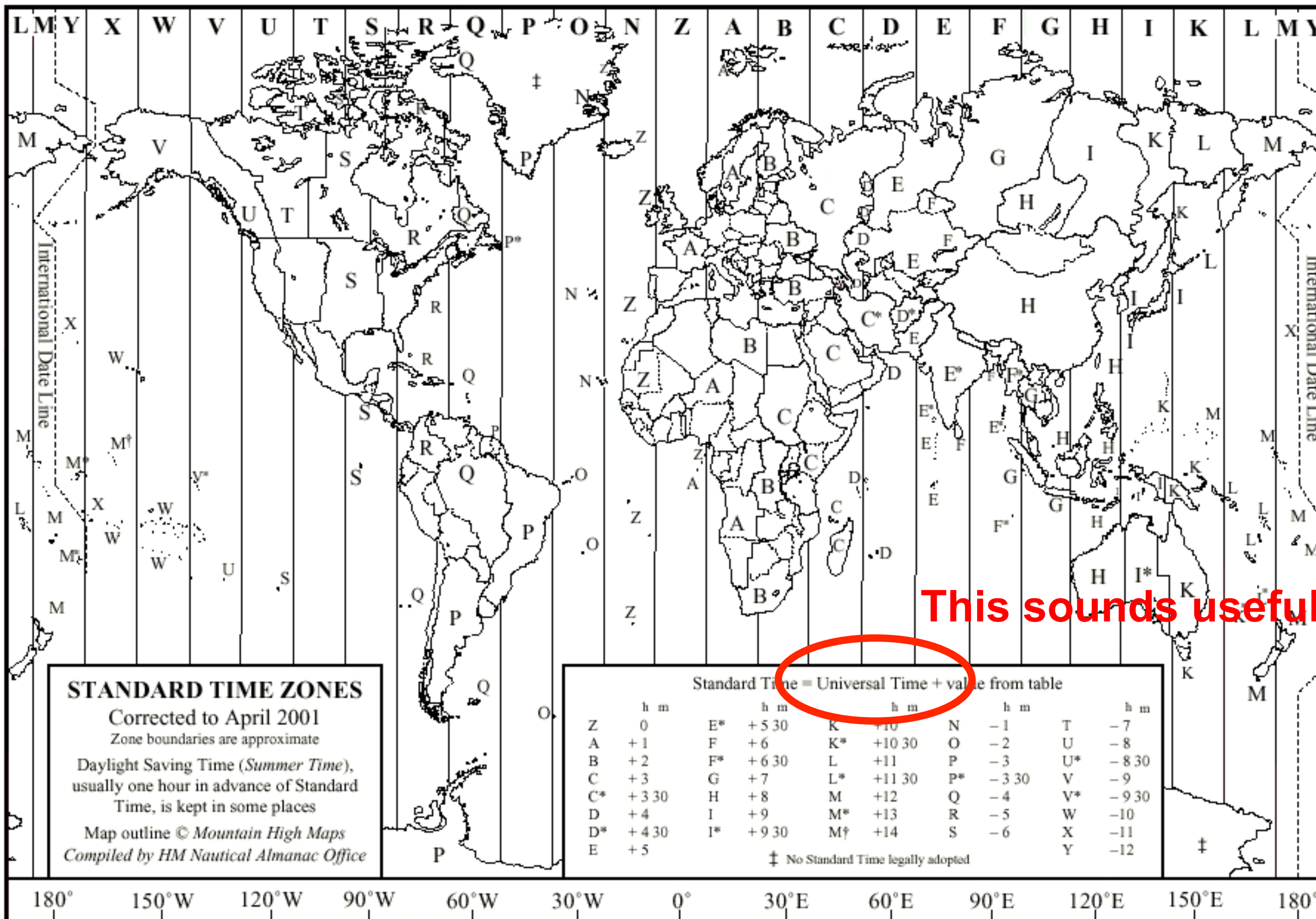
…in distributed systems

- 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

UPPSALA UNIVERSITET

So, what time is it?

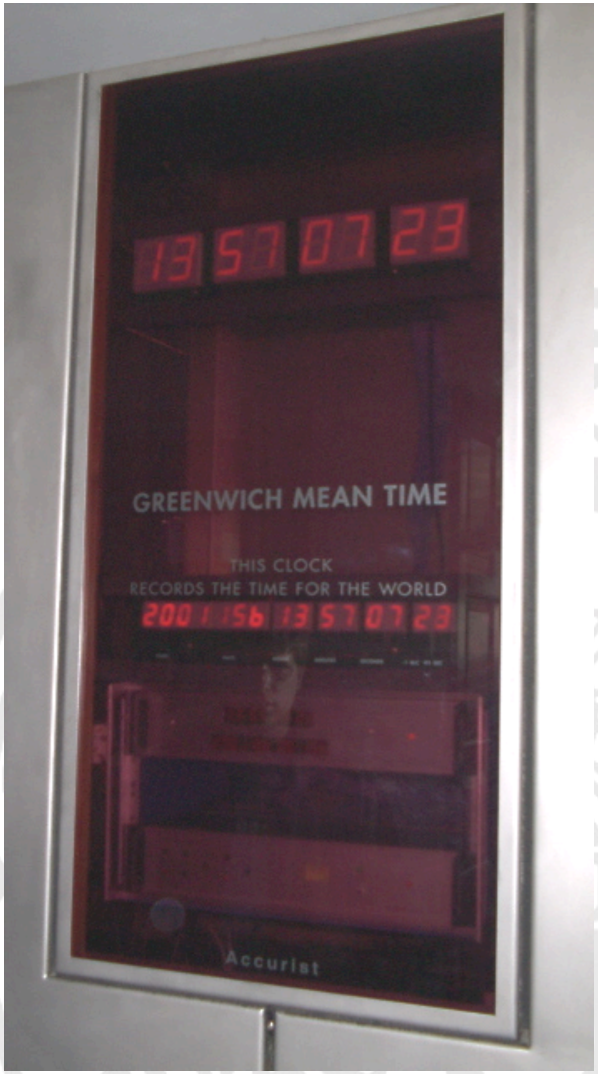


This sounds useful!

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…



UPPSALA UNIVERSITET

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

UPPSALA UNIVERSITET

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

UPPSALA UNIVERSITET

Network Time Protocol (NTP)

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

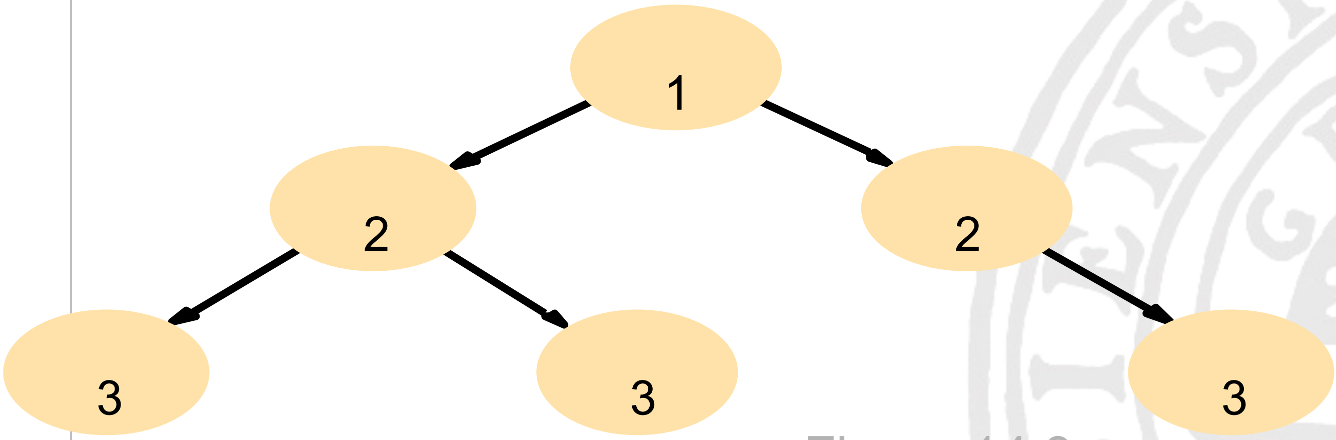


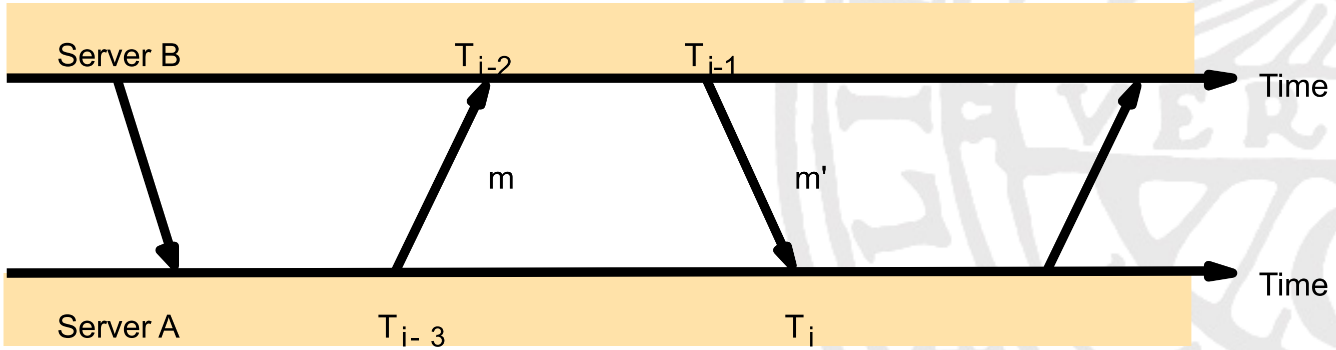
Figure 14.3

NTP – synchronisation of servers

- The synchronization subnet can reconfigure if failures occur, e.g.
  - a primary that loses its UTC source can become a secondary
  - a secondary that loses its primary can use another primary
- Modes of synchronization:
  - Multicast
    - A server within a high speed LAN multicasts time to others which set clocks assuming some delay (not very accurate)
  - Procedure call
    - A server accepts requests from other computers (like Cristiain's algorithm). Higher accuracy. Useful if no hardware multicast.
  - Symmetric
    - Pairs of servers exchange messages containing time information
    - Used where very high accuracies are needed (e.g. for higher levels)

NTP messages

- All modes use UDP
- Each message bears timestamps of recent events:
  - Local times of *Send* and *Receive* of previous message
  - Local times of *Send* of current message
- Recipient notes the time of receipt T_i (we have T_{i-3}, T_{i-2}, T_{i-1}, T_i)
- In symmetric mode there can be a non-negligible delay between messages



UPPSALA UNIVERSITET

Accuracy of NTP

- For each pair of messages between two servers, NTP estimates an offset *o*, between the two clocks and a delay d_i (total time for the two messages, which take *t* and *t'*)

  $T_{i-2} = T_{i-3} + t + o$ and $T_i = T_{i-1} + t' - o$
- This gives us (by adding the equations) :

  $d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$
- Also (by subtracting the equations)

  $o = o_i + (t' - t)/2$ where $o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$
- Using the fact that *t*, *t'*>0 it can be shown that

  $o_i - d_i/2 \leq o \leq o_i + d_i/2$.
  - Thus o_i is an estimate of the offset and d_i is a measure of the accuracy
- NTP servers filter pairs <o_i, d_i>, estimating reliability from variation, allowing them to select peers
- Accuracy of 10s of millisecs over Internet paths (1 on LANs)

UPPSALA
UNIVERSITET

Ignoring time, considering order

- We don’t care of the time
  – Not even internal synchronization
- What matters is the *order*
  – Determening the order of events
    (even if they occur at different machines)

UPPSALA UNIVERSITET

Logical time and logical clocks

- If two events occured at the same process p_i , they occur in the order observed by p_i
- If a message *m* is sent, *send(m)* occur before *receive(m).*
- The ”happened before” relation is transitive

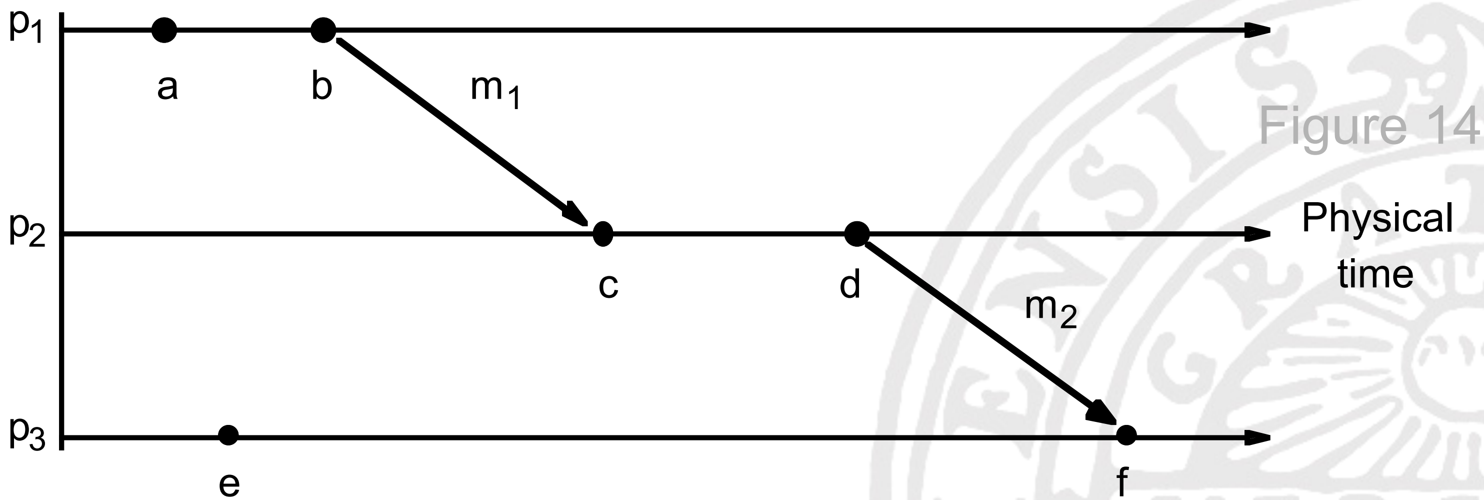


Figure 14.5

- $a \rightarrow b$, $c \rightarrow d$ (at p_1 and p_2 respectively)
- $b \rightarrow c$ because of m_1
- $d \rightarrow f$ because of m_2

UPPSALA UNIVERSITET

Lamport's logical clocks

- A logical clock is a monotonically increasing software counter. It need not relate to a physical clock.
- Each process p_i has a logical clock, L_i which can be used to apply logical timestamps to events
  - 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 *pj* receives *(m,t)* it sets $L_j := max(L_j, t)$ and applies LC1 before timestamping the event *receive (m)*

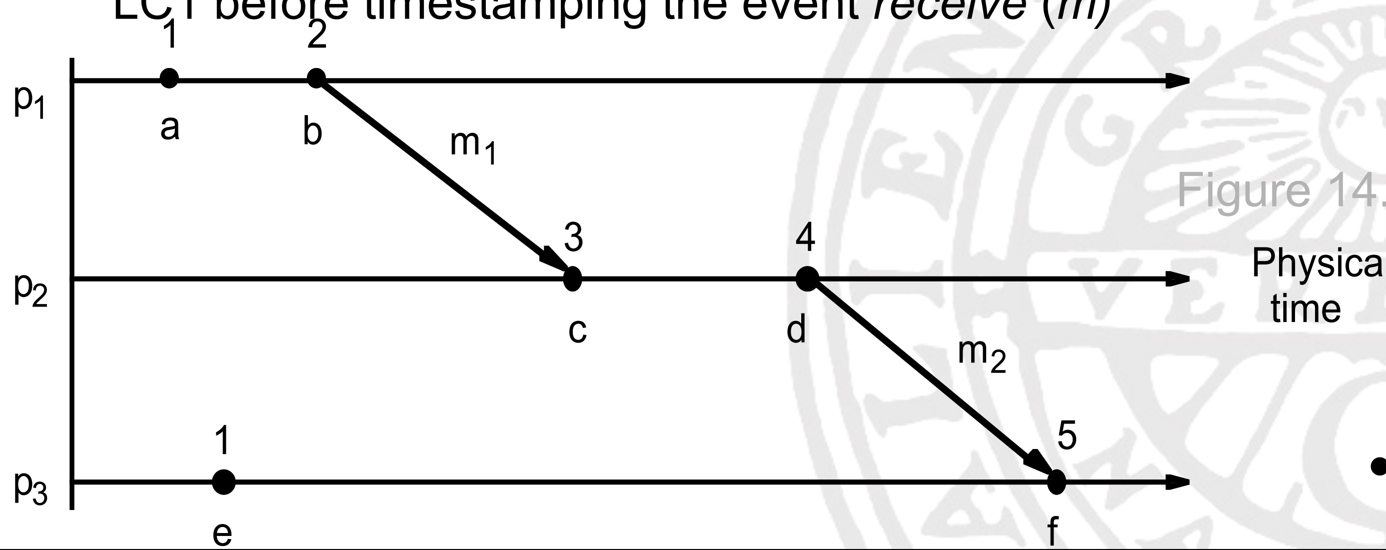
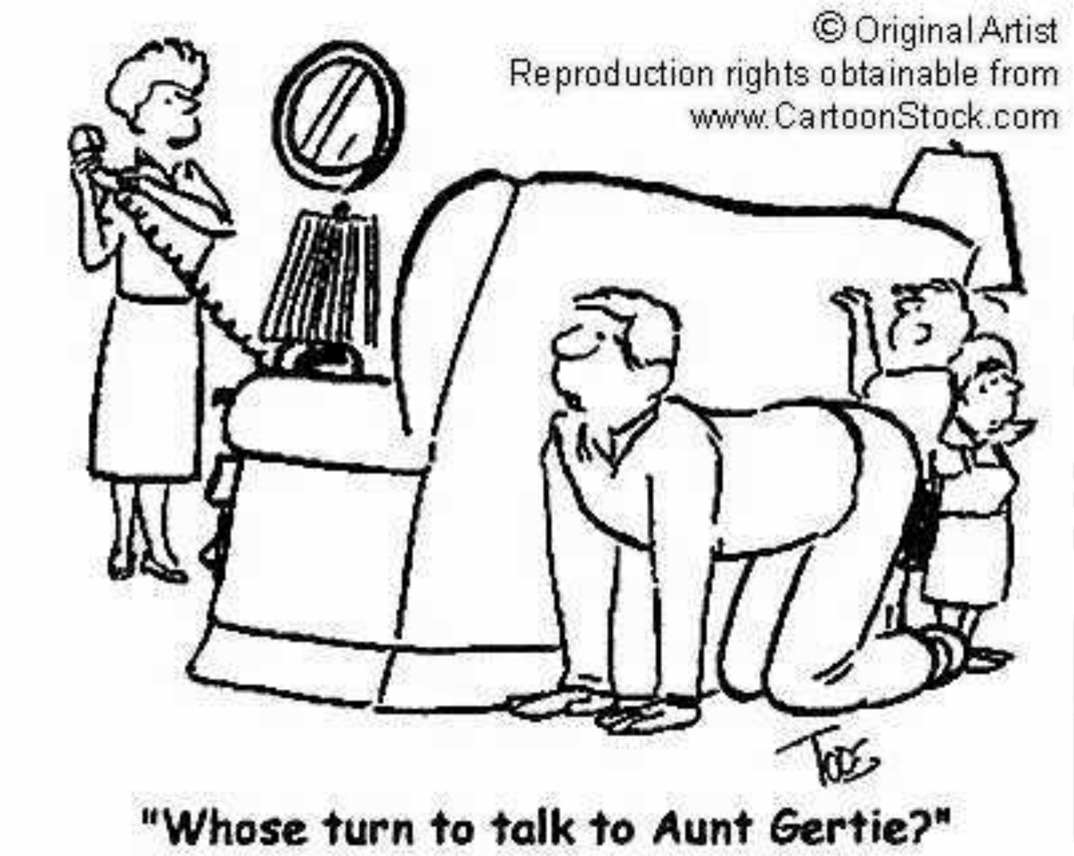


Figure 14.6

UPPSALA UNIVERSITET

Coordination

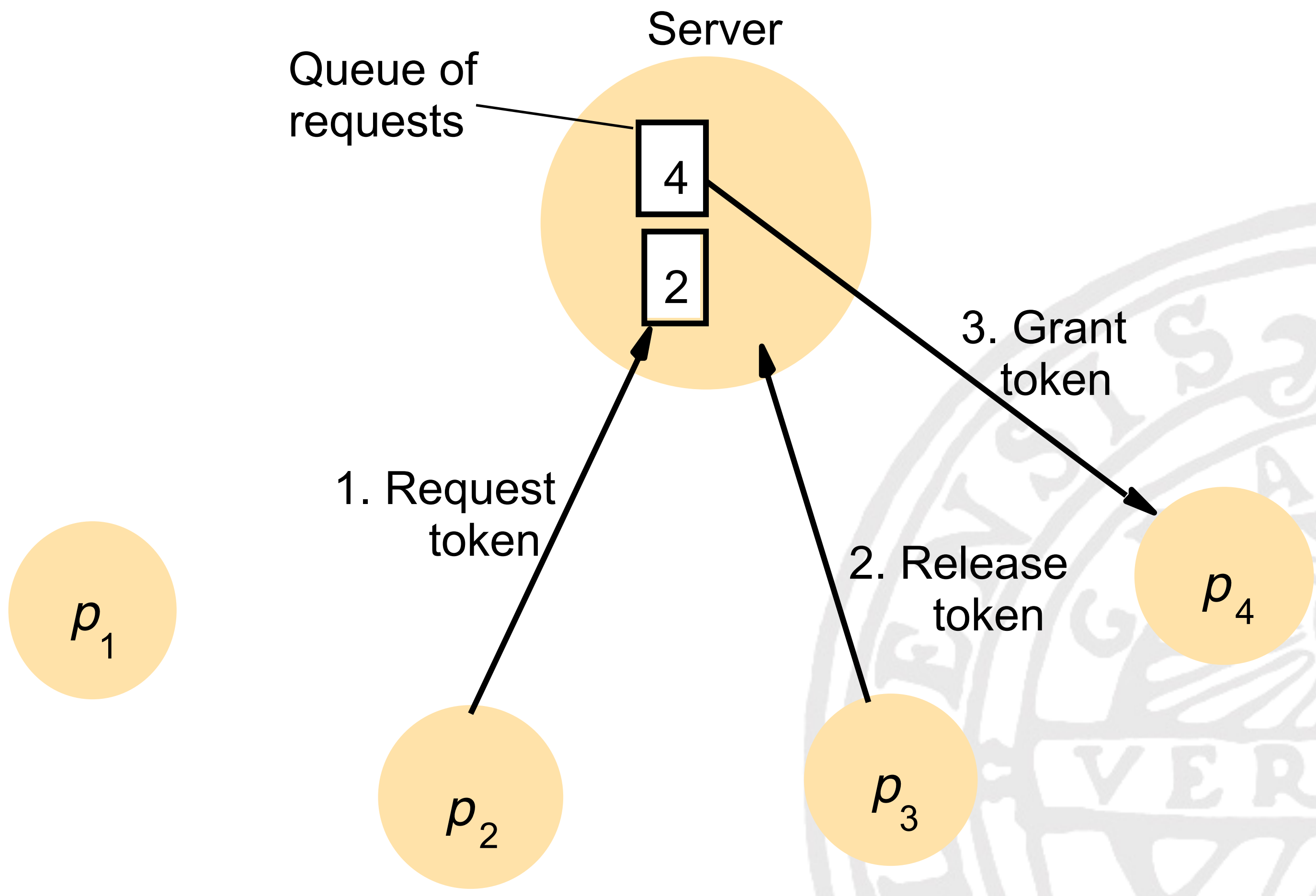


UPPSALA
UNIVERSITET

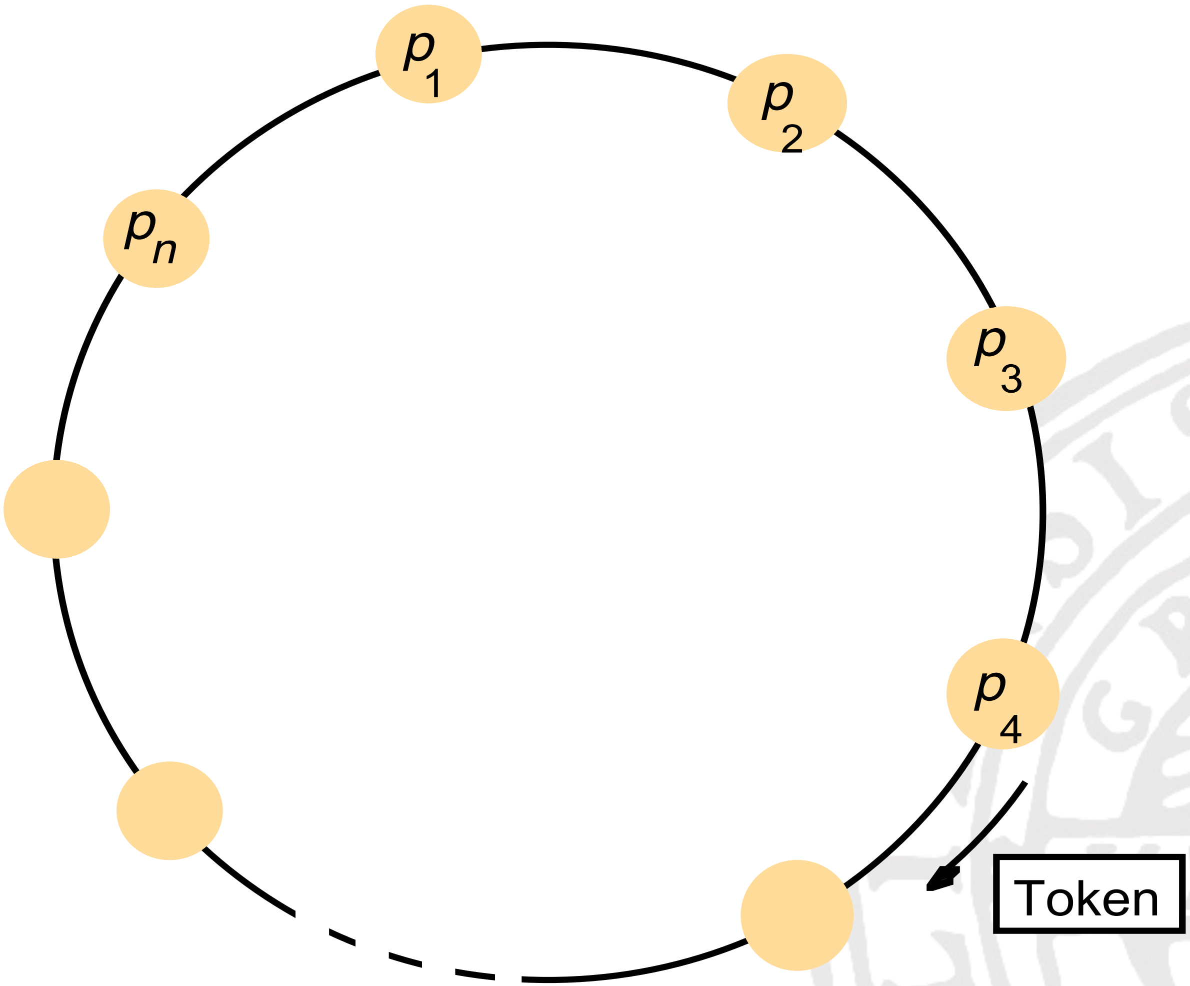
Mutual exclusion

- 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)

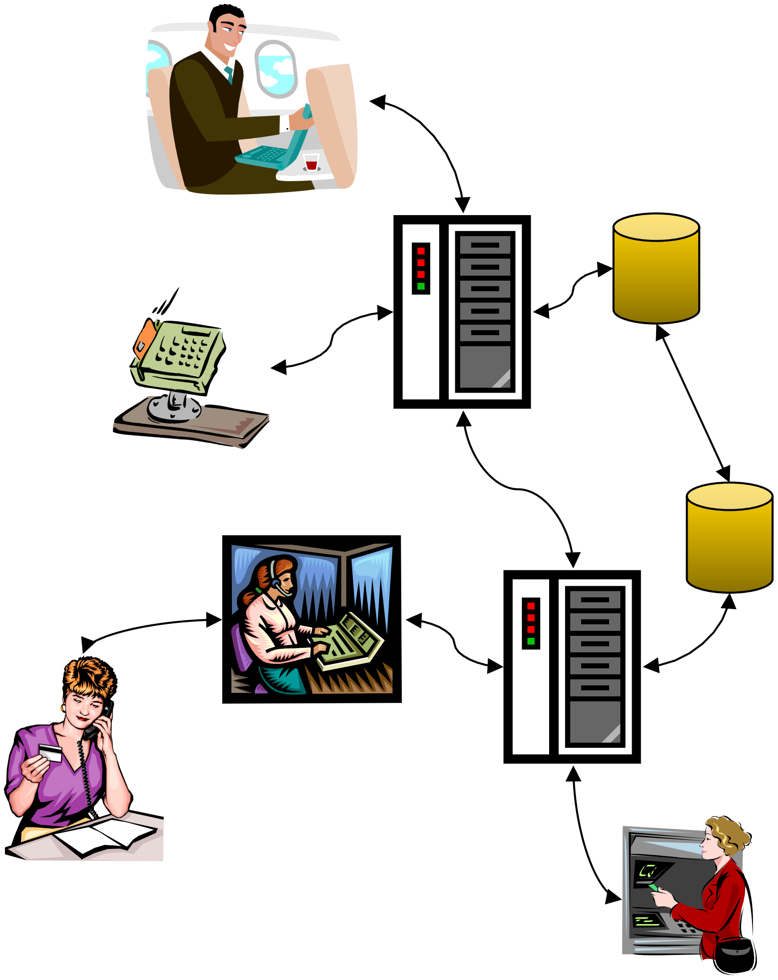
Central server solution



Ring-based solution



Transactions and concurrency



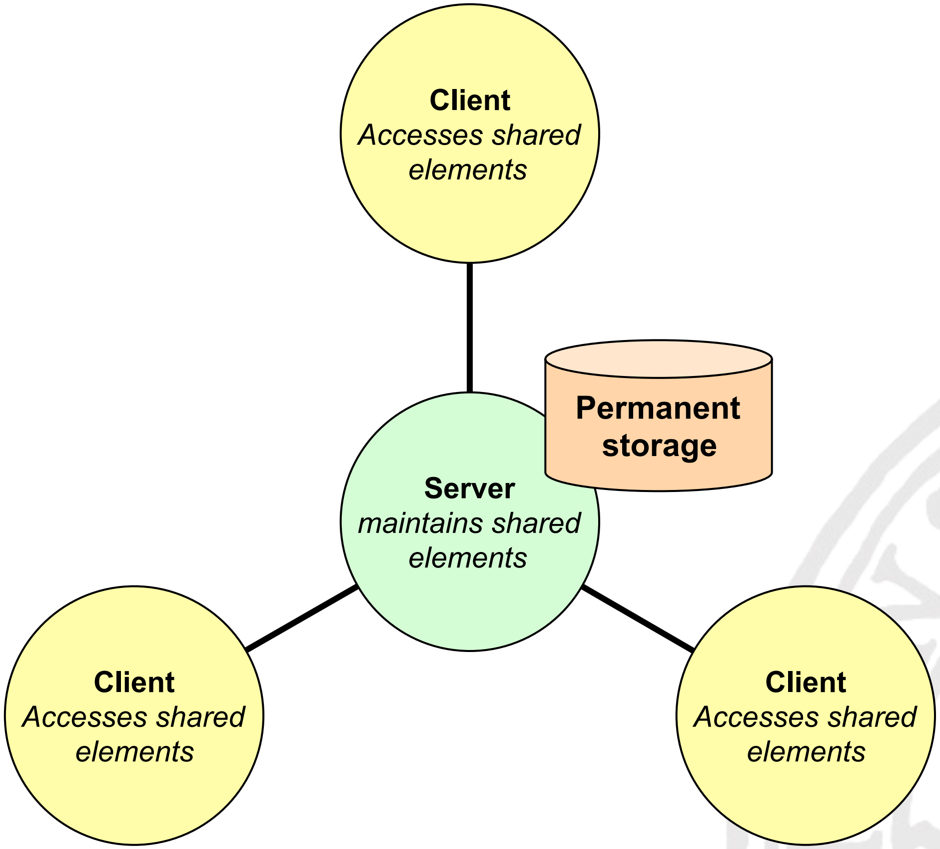
UPPSALA UNIVERSITET

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

UPPSALA UNIVERSITET

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

- **A**tomicity
  - ”all or nothing”
  - A ”complete” transaction requires permanent storage of its effects
- **C**onsistency
  - A transaction takes the system from one consistent state to another
- **I**solation
  - Transactions must not effect eachother
- **D**urability
  - 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 occured serially

UPPSALA UNIVERSITET

Are these serially equivalent?

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

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

UPPSALA UNIVERSITET

Are these serially equivalent?

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

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

UPPSALA UNIVERSITET

Are these serially equivalent?

T	U
read(x,a);	
write(a, x*5);	
	read(x,a);
	write(a, x * 5);
	commit transaction;
abort transaction;	

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
  - …

UPPSALA UNIVERSITET

Concurrency control

- Management of shared resources

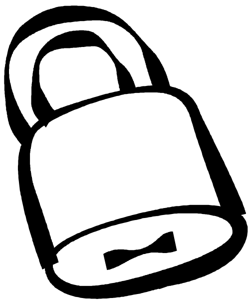
- Different strategies:
  - Locks (most common approach)
  - Optimistic Concurrency Control
  - Timestamp ordering

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 commit:ed
- 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);	
	j = read(i);
	write(k,j);
	commit;
abort;	

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 allowd 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 nestled 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 ...

UPPSALA UNIVERSITET

Deadlocks

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

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

- 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