Ch 4 Synchronization

Clocks and time
Global state
Mutual exclusion
Election algorithms
Distributed transactions

Tanenbaum, van Steen: Ch 5
CoDoKi: Ch 10-12 (3rd ed.)

23-Feb-06 1

Skew between computer clocks in a distributed system

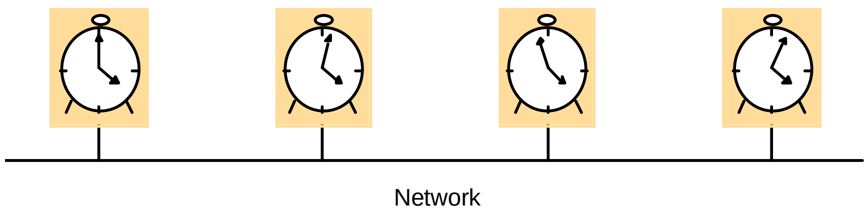
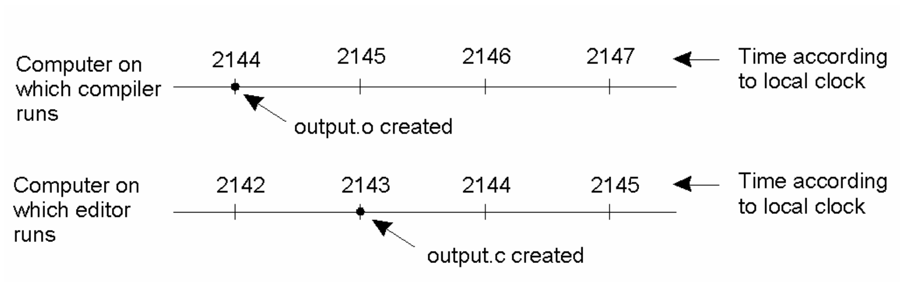


Figure 10.1

23-Feb-06 2

Clock Synchronization



When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

23-Feb-06 3

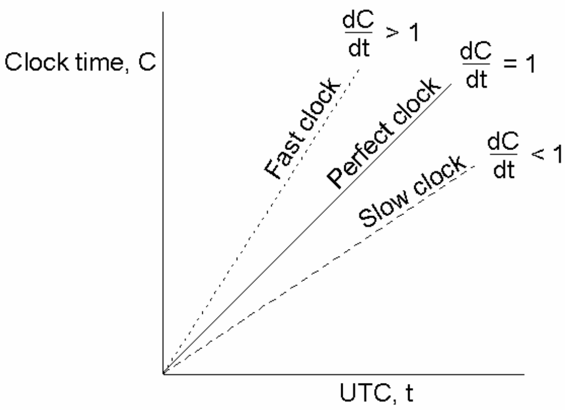
Time and Clocks

Needs	Clocks
real time	universal time (network time)
interval length	computer clock
order of events	network time (universal time)

NOTICE: ***time*** is ***monotonous***

23-Feb-06 4

Clock Synchronization Problem



drift rate: 10^{-6}

1 ms ~ 17 min

1 s ~ 11.6 days

UTC: coordinated universal time

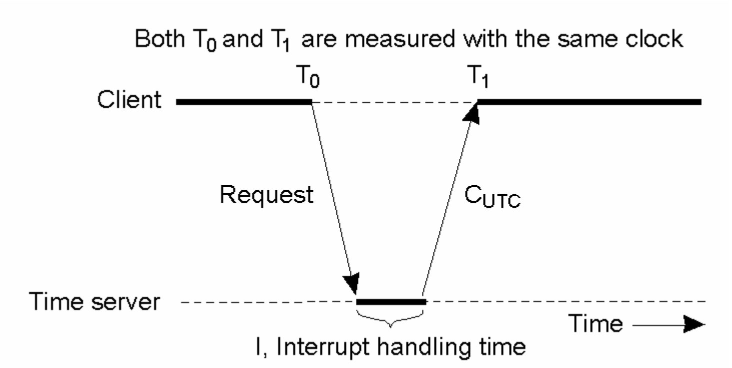
accuracy:
radio 0.1 – 10 ms,
GPS 1 us

The relation between clock time and UTC when clocks tick at different rates.

Synchronization of Clocks: Software-Based Solutions

- Techniques:
  - time stamps of real-time clocks
  - message passing
  - round-trip time (local measurement)
- Cristian's algorithm
- Berkeley algorithm
- Network time protocol (Internet)

Cristian's Algorithm



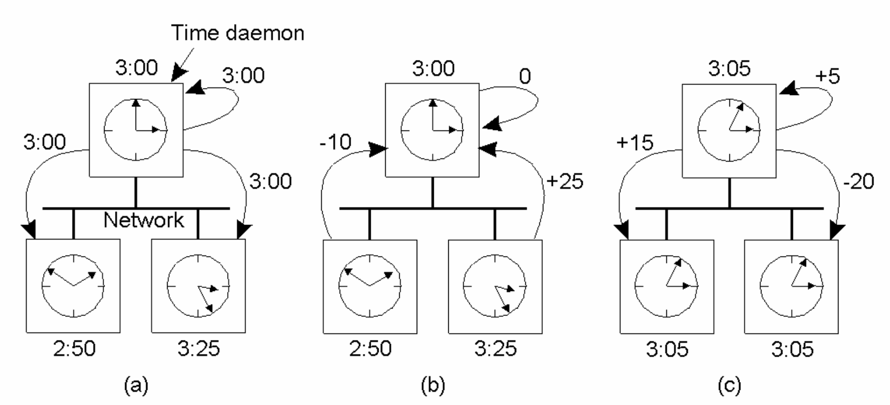
Current time from a time server: UTC from radio/satellite etc

Problems:

- time must never run backward
- variable delays in message passing / delivery

23-Feb-06 7

The Berkeley Algorithm



a) The **time daemon asks** all the other machines for their clock values

b) The machines answer

c) The time daemon tells everyone how to adjust their clock *(be careful with averages!)*

23-Feb-06 8

Clocks and Synchronization

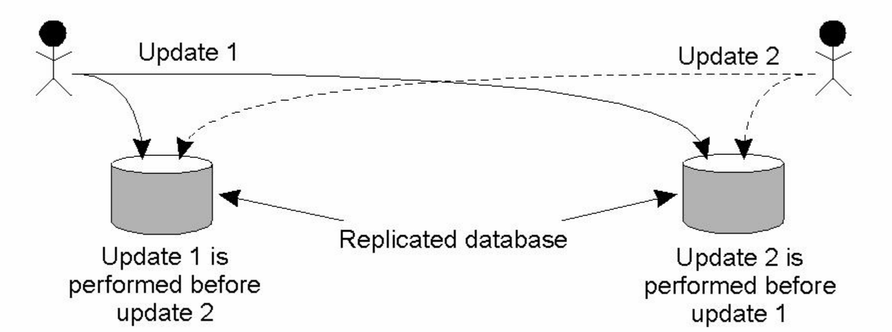
Needs

- ”*causality*”: real-time order ~ timestamp order (”behavioral correctness” – seen by the user)
- *groups* / *replicates*: all members see the events in the same order
- ”*multiple-copy-updates*”: order of updates, consistency conflicts?
- *serializability of transactions*: bases on a common understanding of transaction order

A physical clock is **not always** sufficent!

23-Feb-06 9

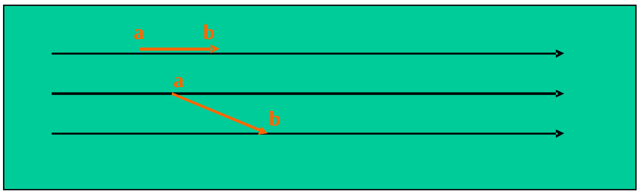
Example: Totally-Ordered Multicasting (1)



Updating a replicated database and leaving it in an inconsistent state.

23-Feb-06 10

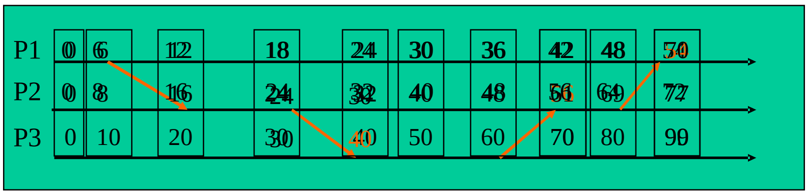
Happened-Before Relation ”a -> b”



- if a, b are *events in the same process*, and a occurs before b, then a -> b
- if a is the event of a *message being sent*, and
  b is the event of the *message being received*,
  then a -> b
- a || c if neither a -> b nor b -> a (a and b are *concurrent*)

Notice: if a -> b and b -> c then a -> c

Logical Clocks: Lamport Timestamps



process p_i , event e , clock L_i , timestamp $L_i(e)$

§ ***at*** p_i : before each event $L_i = L_i + 1$

§ when p_i sends a ***message*** m to p_j

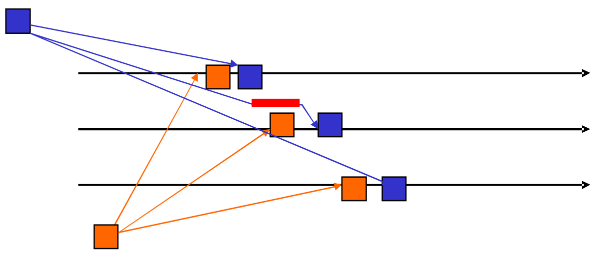
1. p_i: ($L_i = L_i + 1$); $t = L_i$; message = (m, t) ;
2. p_j: $L_j = \max(L_j, t)$; $L_j = L_j + 1$;
3. L_j(receive event) = L_j ;

Lamport Clocks: Problems

1. Timestamps do not specify the order of events
   - e -> e' => L(e) < L(e')

   BUT
   - L(e) < L(e') does not implicate that e -> e'
2. Total ordering
   - problem: define order of e, e' when L(e) = L(e')
   - solution: extended timestamp (T_i, i), where T_i is L_i(e)
   - definition: (T_i, i) < (T_j, j)
     if and only if
     either $T_i < T_j$
     or $T_i = T_j$ and i < j

Example: Totally-Ordered Multicasting (2)



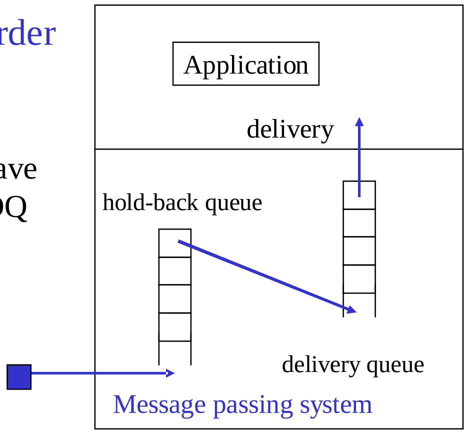
Total ordering:
all receivers (applications) see all messages in the same order (which is not necessarily the original sending order)

Example: multicast operations, group-update operations

Example: Totally-Ordered Multicasting (3)

Guaranteed delivery order

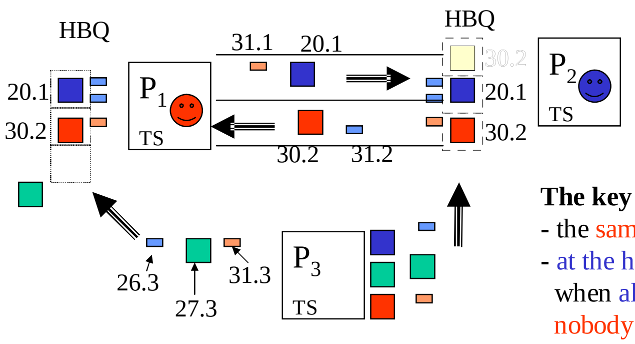
- *new* message => HBQ
- when *all predecessors* have arrived: message => DQ
- when *at the head of DQ*: message => application (application: *receive* …)



Algorithms:
see. Defago et al ACM CS, Dec. 2004

23-Feb-06 15

Example: Totally-Ordered Multicasting (4)



Original timestamps
P_1 19
P_2 29
P_3 25

The key idea
- the same order in all queues
- at the head of HBQ: when all ack's have arrived nobody can pass you

Multicast:
- everybody receives the message (incl. the sender!)
- messages from one sender are received in the sending order
- no messages are lost

23-Feb-06 16

Various Orderings

- Total ordering
- Causal ordering
- FIFO (First In First Out)

(wrt an individual communication channel)

Total and causal ordering are independent: neither induces the other;

Causal ordering induces FIFO

Total, FIFO and Causal Ordering of Multicast Messages

Notice the consistent ordering of totally ordered messages T_1 and T_2, the FIFO-related messages F_1 and F_2 and the causally related messages C_1 and C_3 – and the otherwise arbitrary delivery ordering of messages.

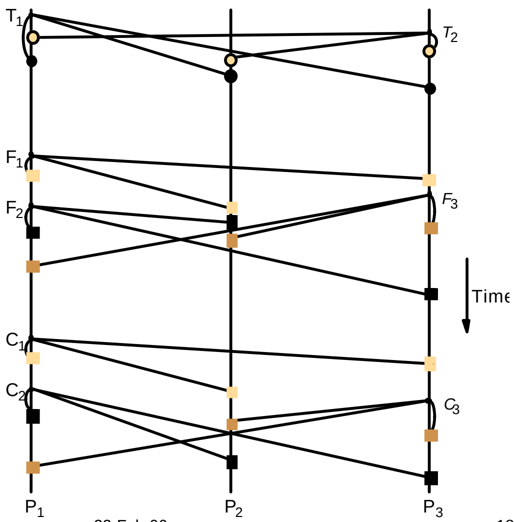


Figure 11.12

Vector Timestamps

Goal:

timestamps should reflect *causal ordering*

L(e) < L(e') => " e happened before e' "

=>

Vector clock

each process P_i maintains a vector V_i :

1. $V_i[i]$ is the number of events that have occurred at P_i
   (the current local time at P_i)
2. if $V_i[j] = k$ then P_i knows about (the first) k events that have occurred at P_j
   (the local time at P_j was k, as P_j sent the last message that P_i has received from it)

23-Feb-06 19

Order of Vector Timestamps

Order of timestamps

- $V = V'$ iff $V[j] = V'[j]$ for all j
- $V \leq V'$ iff $V[j] \leq V'[j]$ for all j
- $V < V'$ iff $V \leq V'$ and $V \neq V'$

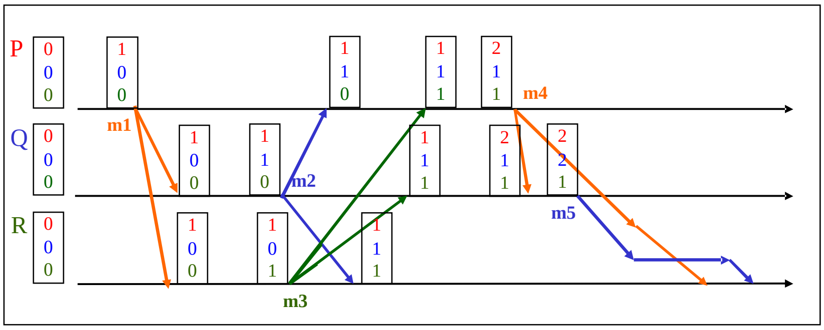
Order of events *(causal order)*

- e -> e' => V(e) < V(e')
- V(e) < V(e') => e -> e'
- concurrency:

  e || e' if **not** $V(e) \leq V(e')$
  and **not** $V(e') \leq V(e)$

23-Feb-06 20

Causal Ordering of Multicasts (1)



Event:
message sent

Timestamp [i,j,k] :
i messages sent from P
j messages sent form Q
k messages sent from R

R: m1 [100] m4 [211]
m2 [110] m5 [221]
m3 [101]

m5 [221] vs. 111

23-Feb-06 21

Causal Ordering of Multicasts (2)

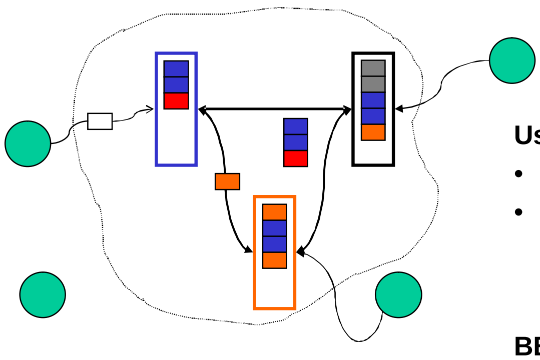
Use of timestamps in causal multicasting

1) P_i multicast: $V_i[i] = V_i[i] + 1$
2) Message: include vt = $V_i[*]$
3) Each receiving P_j : the message **can be delivered when**
   - **$vt[i] = V_j[i] + 1$** *(all previous messages from P_i have arrived)*
   - for each component **k (k≠i): $V_j[k] \geq vt[k]$**

     (P_j has now seen all the messages that P_i had seen when the message was sent)
4) When the message from P_i becomes deliverable at P_j the message is inserted into the delivery queue
   (notice: the delivery queue preserves causal ordering)
5) **At delivery**: $V_j[i] = V_j[i] + 1$

23-Feb-06 22

Causal Ordering of a Bulletin Board (1)



User ó BB *("local events")*

- read: bb <= BB_i (any BB)
- write: to a BB_j that contains all causal predecessors of all bb messages

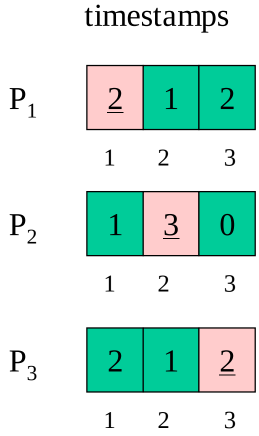
BB_i => BB_j ("messages")

- BB_j must contain all nonlocal predecessors of all BB_i messages

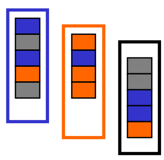
Assumption:
reliable, order-preserving
BB-to-BB transport

23-Feb-06 23

Causal Ordering of a Bulletin Board (2)



Lazy propagation of messages betw. bulletin boards



1) user => P_i
2) P_i ó P_j

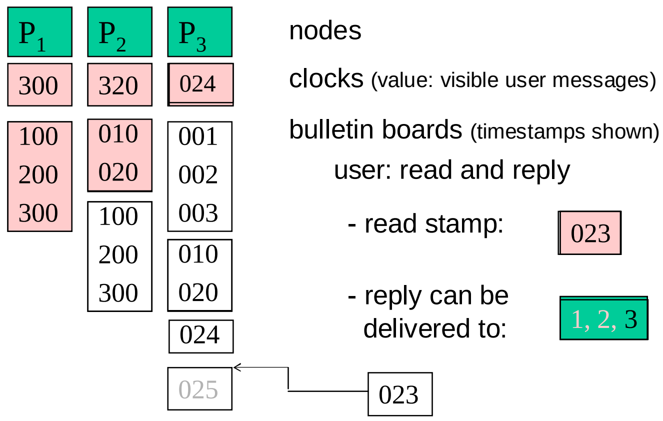
vector clocks: counters

N_i messages from users to the node i

N_j messages originally received by the node j

23-Feb-06 24

Causal Ordering of a Bulletin Board (3)



23-Feb-06 25

Causal Ordering of a Bulletin Board (4)

Updating of vector clocks

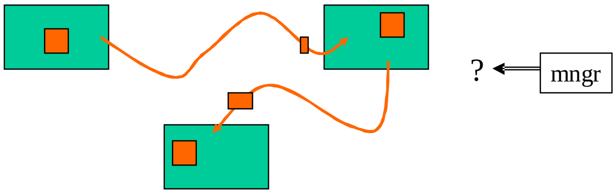
Process P_i

- Local vector clock V_i [∗]
- Update due to a local event: V_i [i] = V_i [i] + 1

- Receiving a message with the timestamp vt [∗]
  - Condition for delivery (to P_i from P_j):
    wait until for all k: k≠j: V_i [k] ≥ vt [k]
  - Update at the delivery: V_i [j] = vt [j]

23-Feb-06 26

Global State (1)



- Needs: checkpointing, garbage collection, deadlock detection, termination, testing
- How to observe the state
  - states of processes
  - messages in transfer

A **state**: application-dependent specification

23-Feb-06 27

Detecting Global Properties

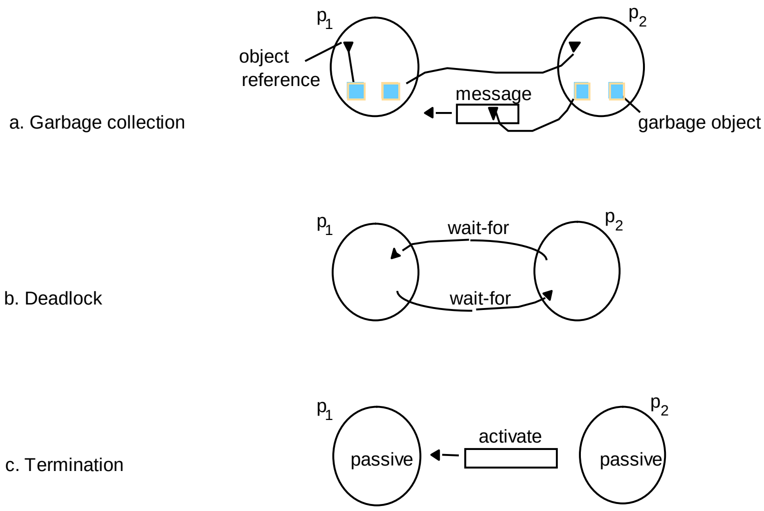


Figure 10.8

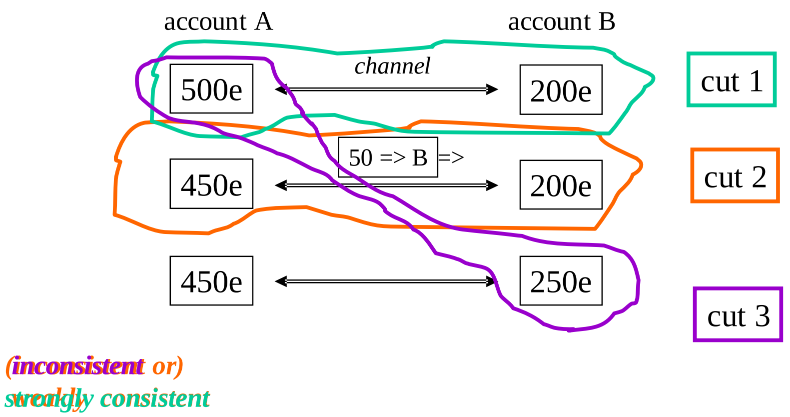
23-Feb-06 28

Distributed Snapshot

- Each node: history of important events
- Observer: at each node i
  - time: the local (logical) clock ” T_i ”
  - state S_i (history: {event, timestamp})
  - => system state { S_i }
- A *cut:* the system state { S_i } ”at time T”
- Requirement:
  - {Si} might have existed ó consistent with respect to some criterion
  - one possibility: consistent wrt ” happened-before relation ”

23-Feb-06 29

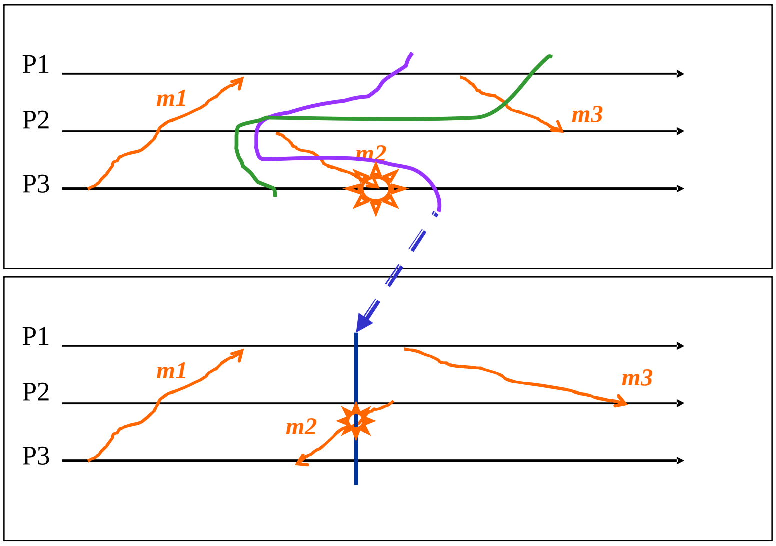
Ad-hoc State Snaphots



state changes: money transfers A ó B
invariant: A+B = 700

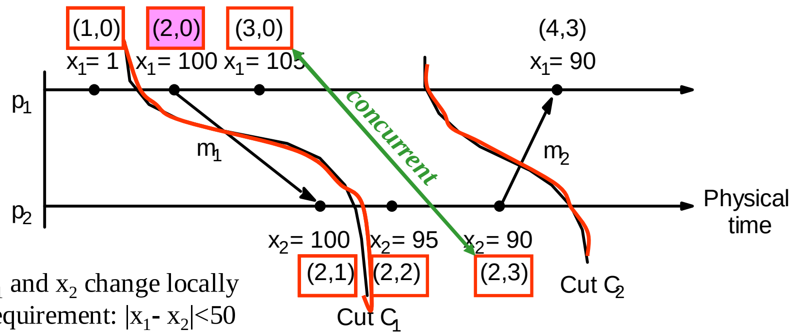
23-Feb-06 30

Consistent and Inconsistent Cuts



23-Feb-06 31

Cuts and Vector Timestamps



x_1 and x_2 change locally
requirement: $|x_1 - x_2| < 50$
a ”large” change (”>9”) =>
send the new value to the other process

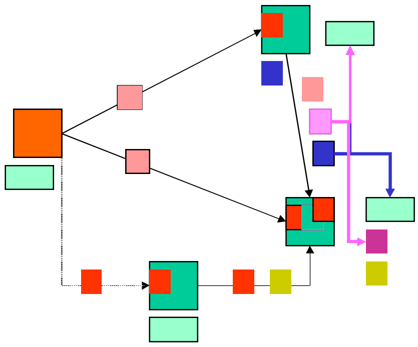
event: a change of the local x
=> increase the vector clock

{S_i} system state history: all events
Cut: all events before the ”cut time”

A **cut is consistent** if, for each event, it also contains all the events that ”happened-before”.

23-Feb-06 32

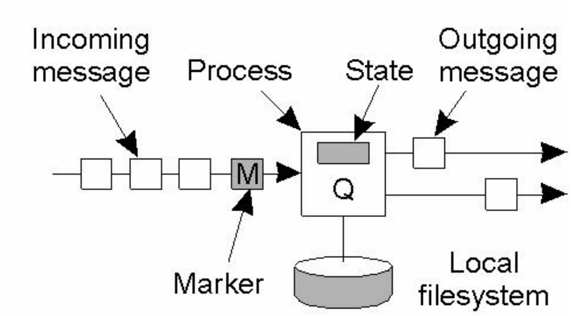
Implementation of Snapshot



Chandy, Lamport

Assumption: point-to-point, order-preserving connections

Chandy Lamport (1)

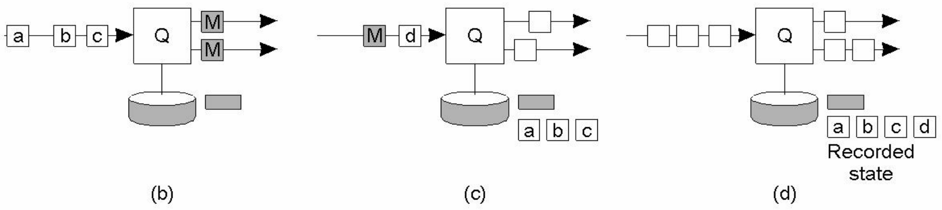


(a)

The snapshot algorithm of Chandy and Lamport

a) Organization of a process and channels for a distributed snapshot

Chandy Lamport (2)



- b) Process Q receives a marker for the first time and records its local state
- c) Q records all incoming messages
- d) Q receives a marker for its incoming channel and finishes recording the state of this incoming channel

23-Feb-06 35

Chandy and Lamport's 'Snapshot' Algorithm

Marker receiving rule for process p_i

On p_i's receipt of a *marker* message over channel *c*:

if (p_i has not yet recorded its state) it

records its process state now;

records the state of *c* as the empty set;

turns on recording of messages arriving over other incoming channels;

else

p_i records the state of *c* as the set of messages it has received over *c* since it saved its state.

end if

Marker sending rule for process p_i

After p_i has recorded its state, for each outgoing channel *c*:

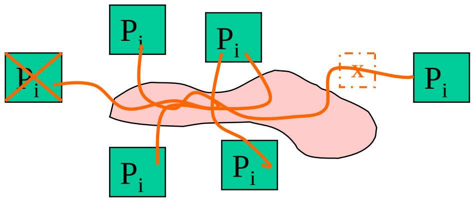
p_i sends one marker message over *c*

(before it sends any other message over *c*).

Figure 10.10

23-Feb-06 36

Coordination and Agreement



Coordination of functionality

- reservation of resources *(distributed mutual exclusion)*
- elections (coordinator, initiator)
- multicasting
- distributed transactions

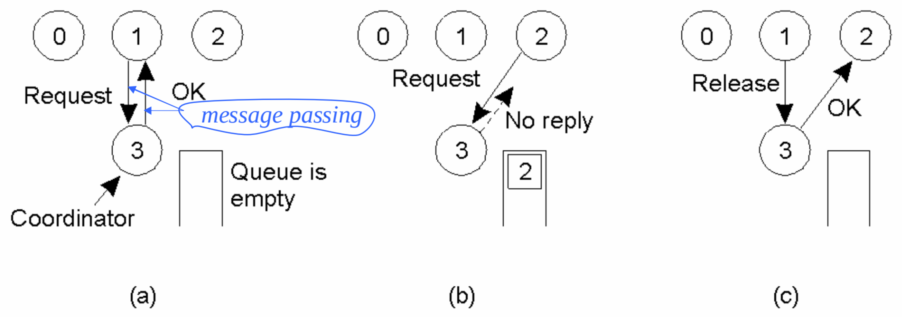
23-Feb-06 37

Decision Making

- Centralized: one coordinator (decision maker)
  - algorithms are simple
  - no fault tolerance *(if the coordinator fails)*
- Distributed decision making
  - algorithms tend to become complex
  - may be extremely fault tolerant
  - behaviour, correctness ?
  - assumptions about failure behaviour of the platform !
- Centralized role, changing “population of the role”
  - easy: one decision maker at a time
  - challenge: management of the “role population”

23-Feb-06 38

Mutual Exclusion: A Centralized Algorithm (1)



a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted

b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply.

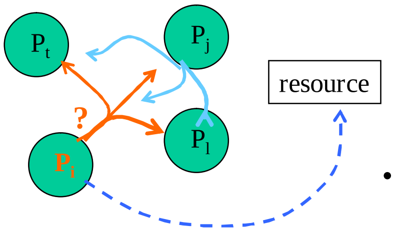
c) When process 1 exits the critical region, it tells the coordinator, which then replies to 2

Mutual Exclusion: A Centralized Algorithm (2)

- **Examples** of usage
  - a stateless server (e.g., Network File Server)
  - a separate lock server
- General **requirements** for mutual exclusion
  1. **safety**: at most one process may execute in the critical section at a time
  2. **liveness**: requests (enter, exit) eventually succeed *(no deadlock, no starvation)*
  3. **fairness** (ordering): if the request A *happens before* the request B then A is honored before B

– **Problems**: fault tolerance, performance

A Distributed Algorithm (1)



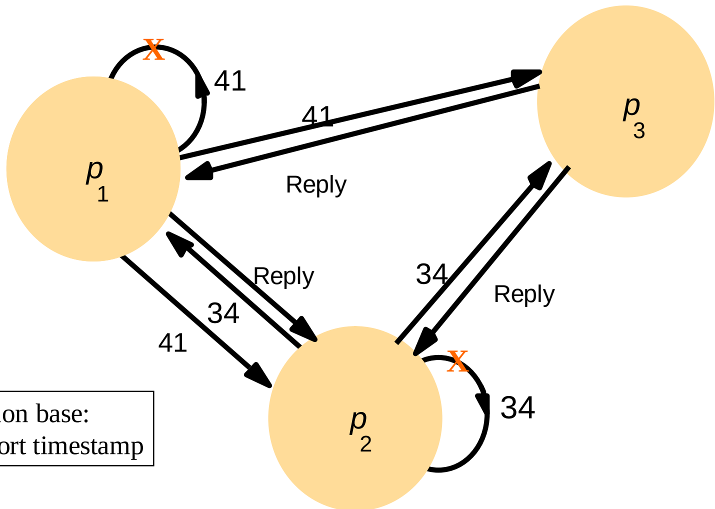
Ricart – Agrawala

- The general idea:
  - ask everybody
  - wait for permission from everybody

The problem:

- several simultaneous requests (e.g., P_i and P_j)
- all members have to agree (*everybody*: “first P_i then P_j”)

Multicast Synchronization



Decision base:
Lamport timestamp

Fig. 11.5 Ricart - Agrawala

A Distributed Algorithm (2)

On initialization
  state := RELEASED;
To enter the section
  state := WANTED;
  T := request's timestamp;
  Multicast *request* to all processes;
  Wait until (number of replies received = *(N-1)*);
  state := HELD;

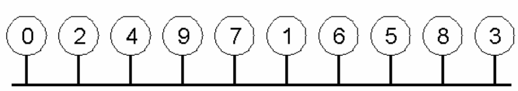
} request processing deferred here

On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
  if (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
  then
    queue *request* from p_i without replying;
  else
    reply immediately to p_i;
  end if;
To exit the critical section
  state := RELEASED;
  reply to all queued requests;

Fig. 11.4 **Ricart - Agrawala**

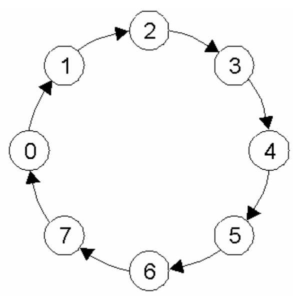
23-Feb-06 43

A Token Ring Algorithm



An unordered group of processes on a network.

(a)



(b)

A logical ring constructed in software.

Algorithm:

- token passing: straightforward
- lost token: 1) detection? 2) recovery?

23-Feb-06 44

Comparison

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

A comparison of three mutual exclusion algorithms.

Notice: the system may contain a remarkable amount of sharable resources!

Election Algorithms

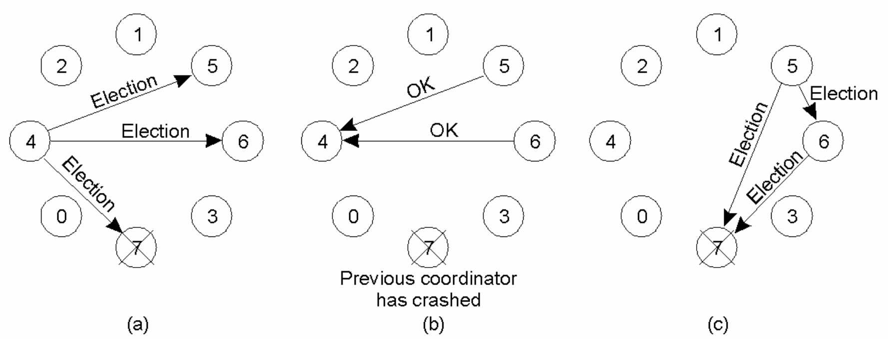
- Need:
  - computation: a group of concurrent actors
  - algorithms based on the activity of a special role (coordinator, initiator)
  - election of a coordinator: initially / after some special event (e.g., the previous coordinator has disappeared)
- Premises:
  - each member of the group {Pi}
    - knows the identities of all other members
    - does not know who is up and who is down
  - all electors use the same algorithm
  - election rule: the member with the highest Pi
- Several algorithms exist

The Bully Algorithm (1)

- P_i notices: coordinator lost
  1. Pi to {all Pj st Pj>Pi}: ELECTION!
  2. if no one responds => Pi is the coordinator
  3. some Pj responds => Pj takes over, Pi's job is done
- P_i gets an ELECTION! message:
  1. reply OK to the sender
  2. if Pi does not yet participate in an ongoing election: hold an election
- The new coordinator P_k to everybody: " P_k COORDINATOR"
- P_i: ongoing election & no "P_k COORDINATOR": hold an election
- P_j recovers: hold an election

23-Feb-06 47

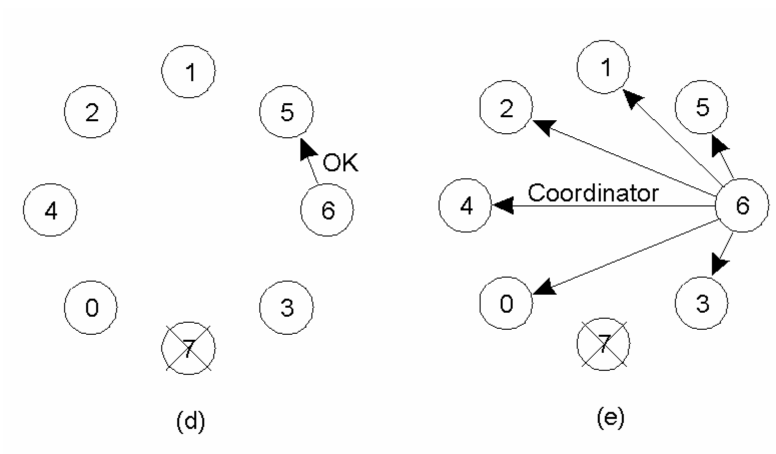
The Bully Algorithm (2)



The bully election algorithm
a) Process 4 holds an election
b) Process 5 and 6 respond, telling 4 to stop
c) Now 5 and 6 each hold an election

23-Feb-06 48

The Bully Algorithm (3)



d) Process 6 tells 5 to stop
e) Process 6 wins and tells everyone

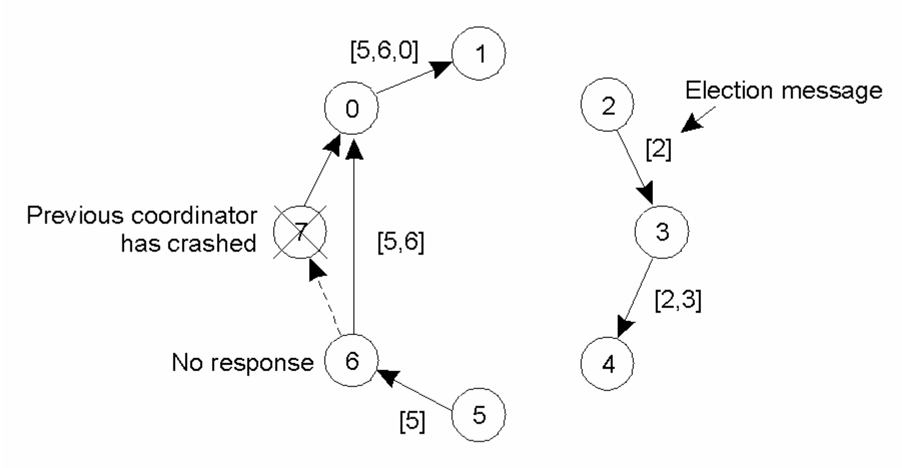
23-Feb-06 49

A Ring Algorithm (1)

- Group {Pi} ”fully connected”; election: ring
- Pi notices: coordinator lost
  - send ELECTION(Pi) to the next P
- Pj receives ELECTION(Pi)
  - send ELECTION(Pi, Pj) to successor
- . . .
- Pi receives ELECTION(..., Pi, ...)
  - active_list = {collect from the message}
  - NC = max {active_list}
  - send COORDINATOR(NC; active_list) to the next P
- …

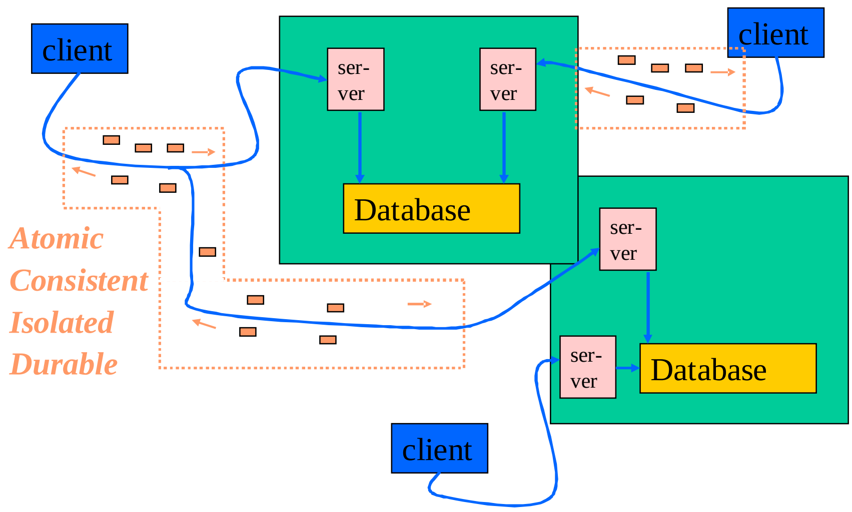
23-Feb-06 50

A Ring Algorithm (2)

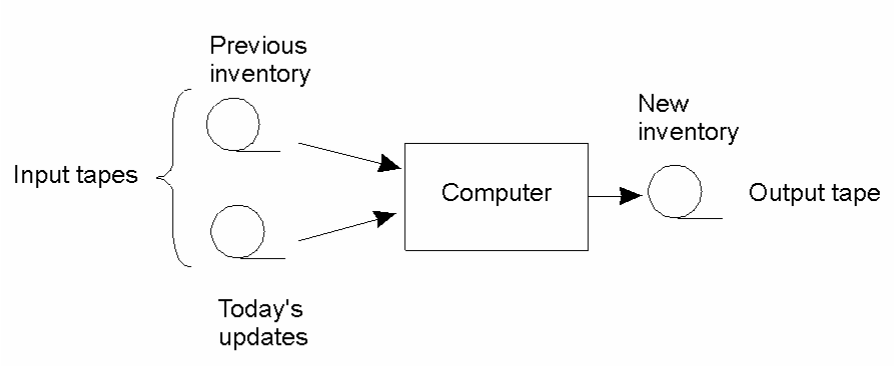


Election algorithm using a ring.

Distributed Transactions



The Transaction Model (1)



Updating a master tape is fault tolerant.

23-Feb-06 53

The Transaction Model (2)

Primitive	Description
BEGIN_TRANSACTION	Make the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

Examples of primitives for transactions.

23-Feb-06 54

The Transaction Model (3)

```
BEGIN_TRANSACTION
  reserve WP -> JFK;
  reserve JFK -> Nairobi;
  reserve Nairobi -> Malindi;
END_TRANSACTION
            (a)
```

```
BEGIN_TRANSACTION
  reserve WP -> JFK;
  reserve JFK -> Nairobi;
  reserve Nairobi -> Malindi full =>
ABORT_TRANSACTION
            (b)
```

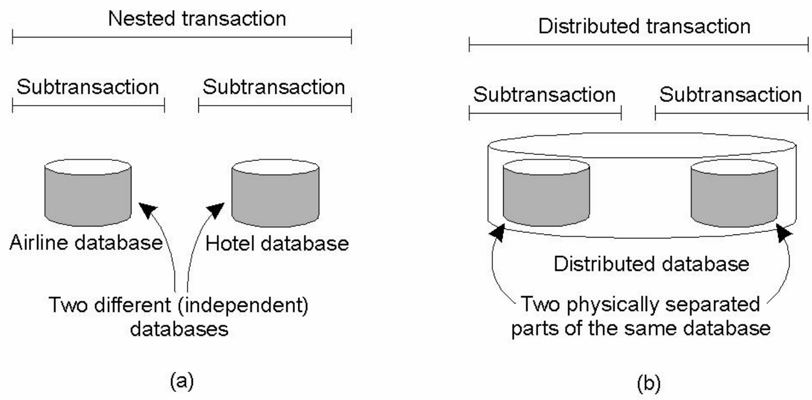
a) Transaction to reserve three flights commits
b) Transaction aborts when third flight is unavailable

Notice:

- a transaction must have a name
- the name must be attached to each operation, which belongs to the transaction

23-Feb-06 55

Distributed Transactions



a) A nested transaction
b) A distributed transaction

23-Feb-06 56

Concurrent Transactions

- Concurrent transactions proceed in parallel
- Shared data (database)

- Concurrency-related problems
  (if no further transaction control):
  - lost updates
  - inconsistent retrievals
  - dirty reads
  - etc.

23-Feb-06 57

The lost update problem

Transaction *T* :		**Transaction *U* :**	
balance = b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*		*balance = b.getBalance();* *b.setBalance(balance*1.1);* *c.withdraw(balance/10)*	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

Figure 12.5 Initial values **a**: $100, **b**: $200 **c**: $300

23-Feb-06 58

The inconsistent retrievals problem

Transaction *V* : *a.withdraw(100)* *b.deposit(100)*		**Transaction *W* :** *aBranch.branchTotal()*	
a.withdraw(100);	$100		
		total = a.getBalance()	$100
		total = total+b.getBalance()	$300
		total = total+c.getBalance()	
b.deposit(100)	$300	:	

Figure 12.6 Initial values **a**: $200, **b**: $200

A serially equivalent interleaving of *T* and *U*

Transaction *T* : *balance = b.getBalance()* *b.setBalance(balance*1.1)* *a.withdraw(balance/10)*		**Transaction *U* :** *balance = b.getBalance()* *b.setBalance(balance*1.1)* *c.withdraw(balance/10)*	
balance = b.getBalance()	$200		
*b.setBalance(balance*1.1)*	$220		
		balance = b.getBalance()	$220
		*b.setBalance(balance*1.1)*	$242
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$278

Figure 12.7 The result corresponds the sequential execution T, U

A dirty read when transaction *T* aborts

Transaction *T*:		**Transaction *U*:**	
a.getBalance() *a.setBalance(balance + 10)*		*a.getBalance()* *a.setBalance(balance + 20)*	
balance = a.getBalance()	**\$100**		
a.setBalance(balance + 10)	\$110		
		balance = a.getBalance()	**\$110**
		a.setBalance(balance + 20)	\$130
		commit transaction	
abort transaction			

Figure 12.11

23-Feb-06 61

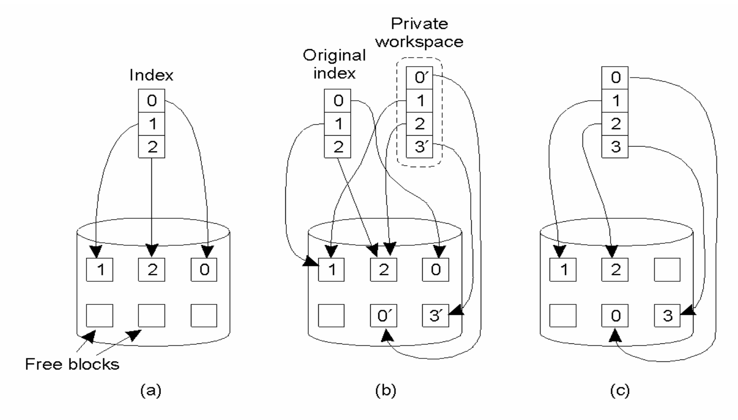
Methods for ACID

- Atomic
  - private workspace,
  - writeahead log
- Consistent

  concurrency control => serialization
  - locks
  - timestamp-based control
  - optimistic concurrency control
- Isolated (see: atomic, consistent)
- Durable (see: Fault tolerance)

23-Feb-06 62

Private Workspace



a) The file index and disk blocks for a three-block file
b) The situation after a transaction has modified block 0 and appended block 3
c) After committing

23-Feb-06 63

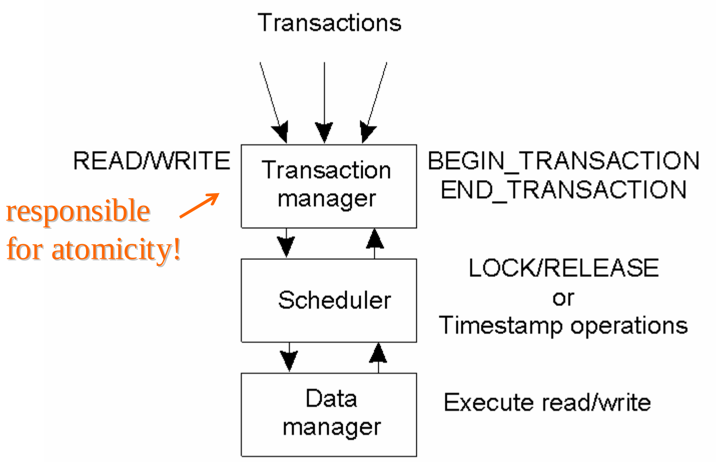
Writeahead Log

x = 0;	Log	Log	Log
y = 0;			
BEGIN_TRANSACTION;			
x = x + 1;	[x = 0 / 1]	[x = 0 / 1]	[x = 0 / 1]
y = y + 2		[y = 0/2]	[y = 0/2]
x = y * y;			[x = 1/4]
END_TRANSACTION;			
(a)	(b)	(c)	(d)

- a) A transaction
- b) – d) The log before each statement is executed

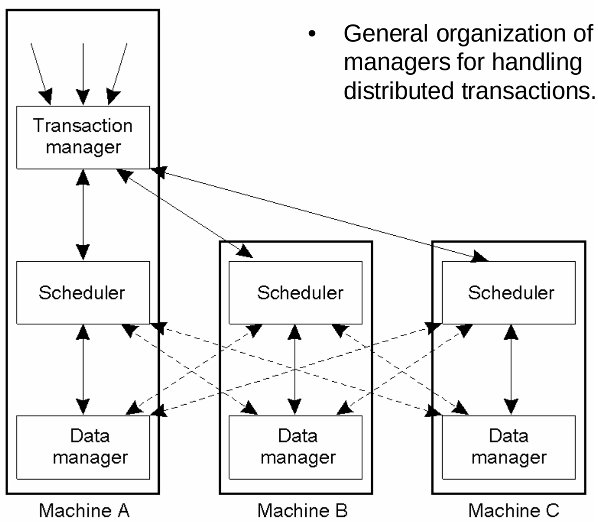
23-Feb-06 64

Concurrency Control (1)



General organization of managers for handling transactions.

Concurrency Control (2)



- General organization of managers for handling distributed transactions.

Serializability

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 1;
END_TRANSACTION
```

(a)

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 2;
END_TRANSACTION
```

(b)

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 3;
END_TRANSACTION
```

(c)

Schedule 1	x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3	Legal
Schedule 2	x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 3;	Legal
Schedule 3	x = 0; x = 0; x = x + 1; x = 0; x = x + 2; x = x + 3;	Illegal

(d)

a) – c) Three transactions T_1, T_2, and T_3; d) Possible schedules
Legal: there exists a serial execution leading to the same result.

23-Feb-06 67

Implementation of Serializability

Decision making: the transaction scheduler

- Locks
  - data item ~ lock
  - request for operation
    - a corresponding lock (read/write) is granted OR
    - the operation is delayed until the lock is released
- Pessimistic timestamp ordering
  - transaction <= timestamp; data item <= R-, W-stamps
  - each request for operation:
    - check serializability
    - continue, wait, abort
- Optimistic timestamp ordering
  - serializability check: at END_OF_TRANSACTION, only

23-Feb-06 68

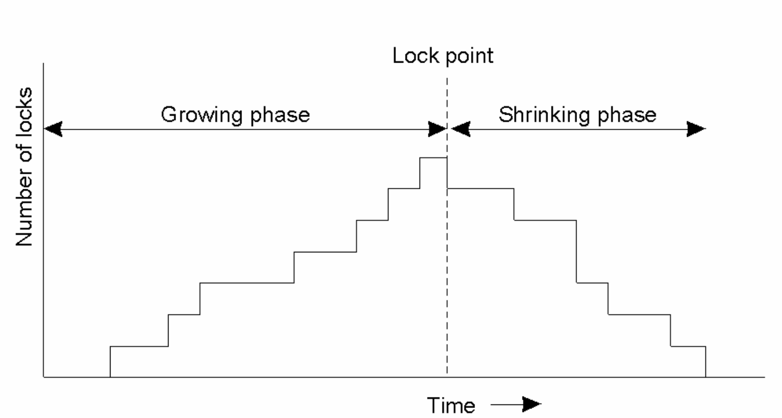
Transactions *T* and *U* with Exclusive Locks

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance()		*balance = b.getBalance()*	
*b.setBalance(bal*1.1)*		*b.setBalance(bal*1.1)*	
a.withdraw(bal/10)		*c.withdraw(bal/10)*	
Operations	Locks	Operations	Locks
openTransaction			
bal = b.getBalance()	lock *B*		
*b.setBalance(bal*1.1)*		*openTransaction*	
a.withdraw(bal/10)	lock *A*	*bal = b.getBalance()*	waits for *T*'s lock on *B*
closeTransaction	unlock *A*, *B*	•••	
			lock *B*
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	lock *C*
		closeTransaction	unlock *B*, *C*

Figure 12.14

23-Feb-06 69

Two-Phase Locking (1)



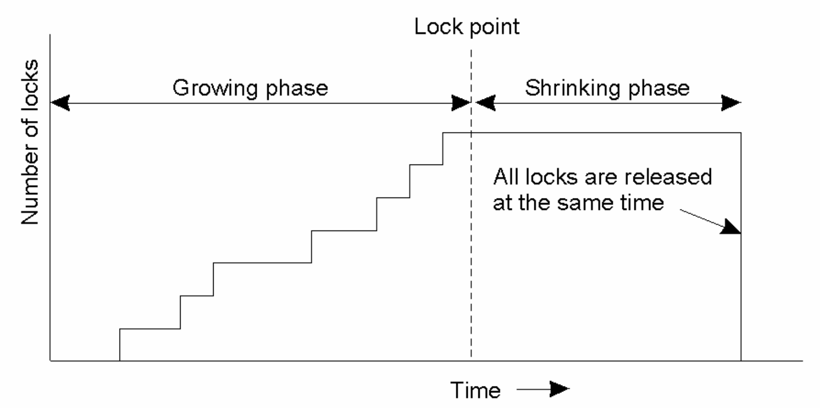
Two-phase locking (2PL).

Releases: application controlled

Problem: dirty reads?

23-Feb-06 70

Two-Phase Locking (2)



Strict two-phase locking.

Centralized or distributed.

23-Feb-06 71

Pessimistic Timestamp Ordering

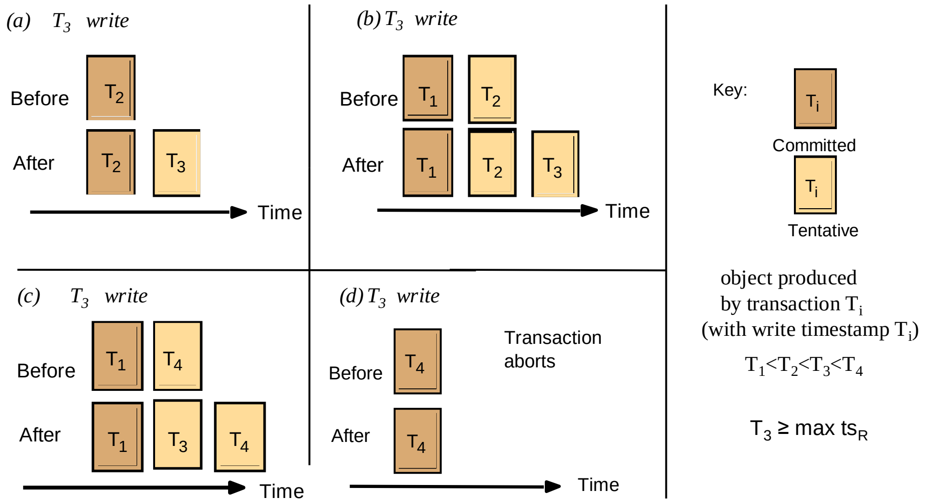
- Transaction timestamp ts(T)
  - given at BEGIN_TRANSACTION (must be unique!)
  - attached to each operation
- Data object timestamps $ts_{RD}(x)$, $ts_{WR}(x)$
  - $ts_{RD}(x)$ = ts(T) of the last T which read x
  - $ts_{wr}(x)$ = ts(T) of the last T which changed x
- Required serial equivalence: ts(T) order of T's

23-Feb-06 72

Pessimistic Timestamp Ordering

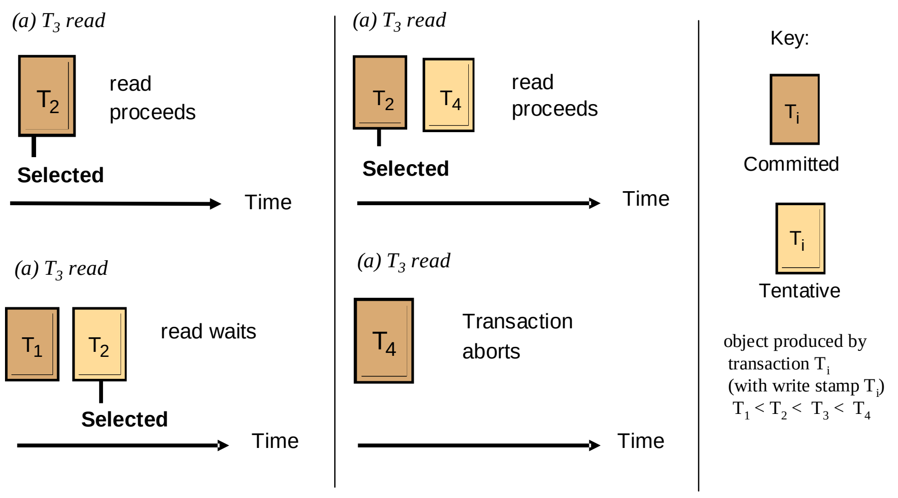
- The rules:
  - you are not allowed to change what later transactions already have seen (or changed!)
  - you are not allowed to read what later transactions already have changed
- Conflicting operations
  - process the older transaction first
  - violation of rules: the transaction is aborted (i.e., the older one: it is too late!)
  - if tentative versions are used, the final decision is made at END_TRANSACTION

Write Operations and Timestamps



CoDoKi: Figure 12.30

Read Operations and Timestamps



CoDoKi: Figure 12.31

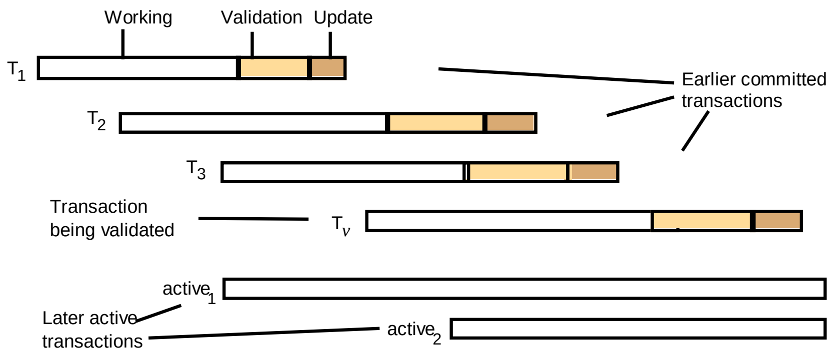
23-Feb-06 75

Optimistic Timestamp Ordering

- Problems with locks
  - general overhead (must be done whether needed or not)
  - possibility of deadlock
  - duration of locking (=> end of the transaction)
- Problems with pessimistic timestamps
  - overhead
- Alternative
  - proceed to the end of the transaction
  - validate
  - applicable if the probability of conflicts is low

23-Feb-06 76

Validation of Transactions



CoDoKi: Figure 12.28

23-Feb-06 77

Validation of Transactions

Backward validation of transaction T_v

```
boolean valid = true;
for (int Ti = startTn+1; Ti <= finishTn; Ti++){
        if (read set of Tv intersects write set of Ti) valid = false;
}
```

Forward validation of transaction T_v

```
boolean valid = true;
for (int Tid = active1; Tid <= activeN; Tid++){
        if (write set of Tv intersects read set of Tid) valid = false;
}
```

CoDoKi: Page 499-500

23-Feb-06 78