Synchronization

Fall 2010

Jussi Kangasharju
Chapter Outline

- Clocks and time
- Global state
- Mutual exclusion
- Election algorithms
- Distributed transactions

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

<table>
<thead>
<tr>
<th>What we need?</th>
<th>How to solve?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real time</td>
<td>Universal time (Network time)</td>
</tr>
<tr>
<td>Interval length</td>
<td>Computer clock</td>
</tr>
<tr>
<td>Order of events</td>
<td>Network time (Universal time)</td>
</tr>
</tbody>
</table>

**NOTE:** *Time is monotonous*
Measuring Time

- Traditionally time measured astronomically
  - Transit of the sun (highest point in the sky)
  - Solar day and solar second
- Problem: Earth’s rotation is slowing down
  - Days get longer and longer
  - 300 million years ago there were 400 days in the year ;-
- Modern way to measure time is atomic clock
  - Based on transitions in Cesium-133 atom
  - Still need to correct for Earth’s rotation
- Result: **Universal Coordinated Time (UTC)**
  - UTC available via radio signal, telephone line, satellite (GPS)
Hardware/Software Clocks

- Physical clocks in computers are realized as crystal oscillation counters at the hardware level
  - Correspond to counter register \( H(t) \)
  - Used to generate interrupts
- Usually scaled to approximate physical time \( t \), yielding software clock \( C(t) \), \( C(t) = \alpha H(t) + \beta \)
  - \( C(t) \) measures time relative to some reference event, e.g., 64 bit counter for # of nanoseconds since last boot
  - Simplification: \( C(t) \) carries an approximation of real time
  - Ideally, \( C(t) = t \) (never 100% achieved)
  - Note: Values given by two consecutive clock queries will differ only if clock resolution is sufficiently smaller than processor cycle time
Problems with Hardware/Software Clocks

- **Skew**: Disagreement in the reading of two clocks
- **Drift**: Difference in the rate at which two clocks count the time
  - Due to physical differences in crystals, plus heat, humidity, voltage, etc.
  - Accumulated drift can lead to significant skew
- **Clock drift rate**: Difference in precision between a perfect reference clock and a physical clock,
  - Usually, $10^{-6}$ sec/sec, $10^{-7}$ to $10^{-8}$ for high precision clocks
Skew between computer clocks in a distributed system

Figure 10.1
Clock Synchronization

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

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

- Fast clock: $\frac{dC}{dt} > 1$
- Perfect clock: $\frac{dC}{dt} = 1$
- Slow clock: $\frac{dC}{dt} < 1$

- Drift rate: $10^{-6}$
  - $1\text{ ms} \sim 17\text{ min}$
  - $1\text{ s} \sim 11.6\text{ days}$

UTC: coordinated universal time
- Accuracy:
  - Radio: 0.1 – 10 ms,
  - GPS: 1 us
Synchronizing Clocks

- **External synchronization**
  - Synchronize process’s clock with an authoritative external reference clock $S(t)$ by limiting skew to a delay bound $D > 0$
    - $|S(t) - C_i(t)| < D$ for all $t$
  - For example, synchronization with a UTC source

- **Internal synchronization**
  - Synchronize the local clocks within a distributed system to disagree by not more than a delay bound $D > 0$, without necessarily achieving external synchronization
    - $|C_i(t) - C_j(t)| < D$ for all $i, j, t$

- Obviously:
  - For a system with external synchronization bound of $D$, the internal synchronization is bounded by $2D$
Clock Correctness

- When is a clock correct?
  1. If drift rate falls within a bound $r > 0$, then for any $t$ and $t'$ with $t' > t$ the following error bound in measuring $t$ and $t'$ holds:

\[
(1-r)(t'-t) \leq H(t') - H(t) \leq (1+r)(t'-t)
\]

- Consequence: No jumps in hardware clocks allowed

  2. Sometimes monotonically increasing clock is enough:

\[
t' > t \Rightarrow C(t') > C(t)
\]

  3. Frequently used condition:

- Monotonically increasing
- Drift rate bounded between synchronization points
- Clock may jump ahead at synchronization points
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)
Christian’s Algorithm

- Observations
  - Round trip times between processes are often reasonably short in practice, yet theoretically unbounded.
  - Practical estimate possible if round-trip times are sufficiently short in comparison to required accuracy.

- Principle
  - Use UTC-synchronized time server S.
  - Process P sends requests to S.
  - Measures round-trip time $T_{\text{round}}$.
    - In LAN, $T_{\text{round}}$ should be around 1-10 ms.
    - During this time, a clock with a $10^{-6}$ sec/sec drift rate varies by at most $10^{-8}$ sec.
    - Hence the estimate of $T_{\text{round}}$ is reasonably accurate.
  - Naive estimate: Set clock to $t + \frac{1}{2}T_{\text{round}}$. 
Both $T_0$ and $T_1$ are measured with the same clock.

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

Problems:
- time must never run backward
- variable delays in message passing / delivery
Christian’s Algorithm: Analysis

**Accuracy of estimate?**

**Assumptions:**
- requests and replies via same net
- $min$ delay is either known or can be estimated conservatively

**Calculation:**
- Earliest time that S can have sent reply: $t_0 + min$
- Latest time that S can have sent reply: $t_0 + T_{round} - min$
- Total time range for answer: $T_{round} - 2 * min$
- Accuracy is $\pm \left(\frac{1}{2}T_{round} - min\right)$

**Discussion**
- Really only suitable for LAN environment or Intranet
- Problem of failure of S
Alternative Algorithm

- **Berkeley algorithm** (Gusella&Zatti ‘89)
  - No external synchronization, but one master server
  - Master polls slaves periodically about their clock readings
  - Estimate of local clock times using round trip estimation
  - Averages the values obtained from a group of processes
    - Cancels out individual clock’s tendencies to run fast
  - Tells slave processes by which amount of time to adjust local clock
  - Master failure: Master election algorithm (see later)

- **Experiment**
  - 15 computers, local drift rate < 2x10⁻⁵, max round-trip 10 ms
  - Clocks were synchronized to within 20-25 ms

- **Note**: Neither algorithm is really suitable for Internet
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
Clock Synchronization: NTP

Goals

- ability to externally synchronize clients via Internet to UTC
- provide reliable service tolerating lengthy losses of connectivity
- enable clients to resynchronize sufficiently frequently to offset typical HW drift rates
- provide protection against interference

Synchronization subnets

- strata 1
- strata 2
- strata 3 (user workstations)
NTP Basic Idea

- Layered client-server architecture, based on UDP message passing
- Synchronization at clients with higher strata number less accurate due to increased latency to strata 1 time server
- Failure robustness: if a strata 1 server fails, it may become a strata 2 server that is being synchronized though another strata 1 server
NTP Modes

- **Multicast:**
  - One computer periodically multicasts time info to all other computers on network
  - These adjust clock assuming a very small transmission delay
  - Only suitable for high speed LANs; yields low but usually acceptable sync.

- **Procedure-call:** similar to Christian’s protocol
  - Server accepts requests from clients
  - Applicable where higher accuracy is needed, or where multicast is not supported by the network’s hard- and software

- **Symmetric:**
  - Used where high accuracy is needed
Procedure-Call and Symmetric Modes

- All messages carry timing history information
  - local timestamps of send and receive of the previous NTP message
  - local timestamp of send of this message

- For each pair \( i \) of messages \((m, m')\) exchanged between two servers
  the following values are being computed

  \[
  \langle T_{i-3}, T_{i-2}, T_{i-1}, m' \rangle
  \]

  (based on 3 values carried w/ msg and 4\textsuperscript{th} value obtained via local timestamp):
  - offset \( o_i \): estimate for the actual offset between two clocks
  - delay \( d_i \): true total transmission time for the pair of messages
Let $o$ the true offset of B’s clock relative to A’s clock, and let $t$ and $t’$ the true transmission times of $m$ and $m’$ ($T_i, T_{i-1} \ldots$ are not true time).

**Delay**

\[ T_{i-2} = T_{i-3} + t + o \]  \hspace{1em} \text{(1)}

\[ T_i = T_{i-1} + t’ - o \]  \hspace{1em} \text{(2)}

which leads to

\[ d_i = t + t’ = T_{i-2} - T_{i-3} + T_i - T_{i-1} \]  \hspace{1em} \text{(clock errors zeroed out \( \Rightarrow \) true d)}

**Offset**

\[ o_i = \frac{1}{2} \left( T_{i-2} - T_{i-3} + T_{i-1} - T_i \right) \]  \hspace{1em} \text{(only an estimate)}
NTP Implementation

- Statistical algorithms based on 8 most recent \(<o_i, d_i>\) pairs: \(\rightarrow\) determine quality of estimates
- The value of \(o_i\) that corresponds to the minimum \(d_i\) is chosen as an estimate for \(o\)
- Time server communicates with multiple peers, eliminates peers with unreliable data, favors peers with higher strata number (e.g., for primary synchronization partner selection)
- NTP phase lock loop model: modify local clock in accordance with observed drift rate
- Experiments achieve synchronization accuracies of 10 msecs over Internet, and 1 msec on LAN using NTP
Clocks and Synchronization

Requirements:

- "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 perfect physical clock is sufficient!
A perfect physical clock is impossible to implement!
Above requirements met with much lighter solutions!
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 || b if neither a -> b nor b -> a (a and b are concurrent)

Note: 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. \( p_i: (L_i = L_i + 1); t = L_i; \text{message} = (m, t) \);
    2. \( p_j: L_j = \max(L_j, t); L_j = L_j + 1; \)
    3. \( L_j(\text{receive event}) = L_j \);
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 imply 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 (1)

Updating a replicated database and leaving it in an inconsistent state.
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
Example: Totally-Ordered Multicasting (4)

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

Original timestamps
P₁ 19
P₂ 29
P₃ 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
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
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

$L(e) < L(e') \Rightarrow \text{ "e happened before e' "}$

$\Rightarrow$

Vector clock

each process $P_i$ maintains a vector $V_i$:

1. $V_i[i]$ is the number of events that have occurred at $P_i$

   \text{(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 $P_j$

   \text{(the local time at $P_j$ was $k$, as $P_j$ sent the last message that $P_i$ has received from it)}
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)
- \( e \rightarrow e' \) \( \Rightarrow \) \( V(e) < V(e') \)
- \( V(e) < V(e') \) \( \Rightarrow \) \( e \rightarrow e' \)
- concurrency:
  - \( e \parallel e' \) if not \( V(e) \leq V(e') \) and not \( V(e') \leq V(e) \)
Causal Ordering of Multicasts (1)

**Event:** message sent

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

- R: m1 [100] m4 [211] m2 [110] m5 [221] m3 [101]
- m5 [211] vs. 111
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 $k$ ($k \neq i$): $V_j[k] \geq vt[k]$
     ($P_j$ 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
   (note: the delivery queue preserves causal ordering)
5) At delivery: $V_j[i] = V_j[i] + 1$
Causal Ordering of a Bulletin Board (1)

User ⇔ BB ("local events")
- read: \( bb \leq BB_i \) (any BB)
- write: to a \( BB_j \) that contains all causal predecessors of all \( bb \) messages

\( BB_i \Rightarrow BB_j \) ("messages")
- \( BB_j \) must contain all nonlocal predecessors of all \( BB_i \) messages

Assumption:
reliable, order-preserving BB-to-BB transport
Causal Ordering of a Bulletin Board (2)

Lazy propagation of messages between bulletin boards:
1) user => $P_i$
2) $P_i \Leftrightarrow P_j$

Vector clocks: counters
- Messages from users to node $i$
- Messages originally received by node $j$

Timestamps:

- $P_1$
  - $2 \ 1 \ 2$
  - $1 \ 2 \ 3$

- $P_2$
  - $1 \ 3 \ 0$
  - $1 \ 2 \ 3$

- $P_3$
  - $2 \ 1 \ 2$
  - $1 \ 2 \ 3$
nodes

clocks (value: visible user messages)

bulletin boards (timestamps shown)

user: read and reply

- read stamp: 023

- reply can be delivered to: 1, 2, 3
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 $v_t[*]$
  
  - Condition for delivery (to $P_i$ from $P_j$):
    
    wait until for all $k: k \neq j$: $V_i[k] \geq v_t[k]$
  
  - Update at the delivery: $V_i[j] = v_t[j]$
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
Detecting Global Properties

Figure 10.8

a. Garbage collection

b. Deadlock

c. Termination
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\})

\[ \Rightarrow \text{system state } \{ S_i \} \]

- A cut: the system state \( \{ S_i \} \) ”at time T”

- Requirement:
  - \( \{S_i\} \) might have existed \( \Leftrightarrow \) consistent with respect to some criterion
  - one possibility: consistent wrt ”happened-before relation”
Ad-hoc State Snapshots

state changes: money transfers $A \leftrightarrow B$

invariant: $A + B = 700$

(account A) 500e $\rightarrow$ channel $\rightarrow$ 200e (account B)

450e $\leftarrow$ 50 $\Rightarrow$ B $\Rightarrow$ 200e

450e $\leftarrow$ 250e

(inconsistent or)

strongly consistent

cut 1

cut 2

cut 3
Consistent and Inconsistent Cuts

- P1
- m1
- P2
- m2
- P3
- m3
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".
The snapshot algorithm of Chandy and Lamport

a) Organization of a process and channels for a distributed snapshot
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
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 11.10
Implementation of Snapshot

point-to-point, order-preserving connections

Chandy, Lamport
Coordination and Agreement

Coordination of functionality

- reservation of resources \((\text{distributed mutual exclusion})\)
- elections (coordinator, initiator)
- multicasting
- distributed transactions
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”
Mutual Exclusion: A Centralized Algorithm (1)

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

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

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)

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 $P_j$”)
A Distributed Algorithm (2)

- **On initialization**
  
  \( \text{state} := \text{RELEASED}; \)

- **To enter the section**
  
  \( \text{state} := \text{WANTED}; \)
  
  \( T := \text{request’s timestamp}; \)
  
  Multicast request to all processes;
  
  Wait until (number of replies received = \( (N-1) \));
  
  \( \text{state} := \text{HELD}; \)

- **On receipt of a request** \(<T_i, p_i>\) at \( p_j \) \((i \neq j)\)
  
  if \((\text{state} = \text{HELD} \text{ or } (\text{state} = \text{WANTED} \text{ and } (T, p_j) < (T_i, p_i)))\)
  
  then
  
  queue request from \( p_i \) without replying;

  else
  
  reply immediately to \( p_i \);

  end if;

- **To exit the critical section**
  
  \( \text{state} := \text{RELEASED}; \)
  
  reply to all queued requests;

Fig. 11.4  **Ricart - Agrawala**
Multicast Synchronization

Decision base: Lamport timestamp

Fig. 11.5 Ricart - Agrawala
A Token Ring Algorithm

An unordered group of processes on a network.

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

A logical ring constructed in software.
## Comparison

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Messages per entry/exit</th>
<th>Delay before entry (in message times)</th>
<th>Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>3</td>
<td>2</td>
<td>Coordinator crash</td>
</tr>
<tr>
<td>Distributed</td>
<td>2 ( (n - 1) )</td>
<td>2 ( (n - 1) )</td>
<td>Crash of any process</td>
</tr>
<tr>
<td>Token ring</td>
<td>1 to ( \infty )</td>
<td>0 to ( n - 1 )</td>
<td>Lost token, process crash</td>
</tr>
</tbody>
</table>

A comparison of three mutual exclusion algorithms.
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. $P_i$ to {all $P_j$ st $P_j > P_i$}: ELECTION!
  2. if no one responds => $P_i$ is the coordinator
  3. some $P_j$ responds => $P_j$ takes over, $P_i$'s job is done

- $P_i$ gets an ELECTION! message:
  1. reply OK to the sender
  2. if $P_i$ 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
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
d) Process 6 tells 5 to stop

e) Process 6 wins and tells everyone
A Ring Algorithm (1)

- Group \{P_i\} "fully connected"; election: ring
- Pi notices: coordinator lost
  - send $ELECTION(P_i)$ to the next $P$
- Pj receives $ELECTION(P_i)$
  - send $ELECTION(P_i, P_j)$ to successor
- ...
- Pi receives $ELECTION(\ldots, P_i, \ldots)$
  - active_list = \{collect from the message\}
  - NC = max \{active_list\}
  - send $COORDINATOR(NC; \text{active_list})$ to the next $P$
- ...

Kangasharju: Distributed Systems
A Ring Algorithm (2)

Election algorithm using a ring.

Previous coordinator has crashed

No response

Election message
Distributed Transactions

Database

Atomic
Consistent
Isolated
Durable

client

server

server

client

Database

server

server

client

server

client
The Transaction Model (1)

Updating a master tape is fault tolerant.
The Transaction Model (2)

Examples of primitives for transactions.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN_TRANSACTION</td>
<td>Make the start of a transaction</td>
</tr>
<tr>
<td>END_TRANSACTION</td>
<td>Terminate the transaction and try to commit</td>
</tr>
<tr>
<td>ABORT_TRANSACTION</td>
<td>Kill the transaction and restore the old values</td>
</tr>
<tr>
<td>READ</td>
<td>Read data from a file, a table, or otherwise</td>
</tr>
<tr>
<td>WRITE</td>
<td>Write data to a file, a table, or otherwise</td>
</tr>
</tbody>
</table>
The Transaction Model (3)

BEGIN_TRANSACTION
reserve WP -> JFK;
reserve JFK -> Nairobi;
reserve Nairobi -> Malindi;
END_TRANSACTION

BEGIN_TRANSACTION
reserve WP -> JFK;
reserve JFK -> Nairobi;
reserve Nairobi -> Malindi full =>
ABORT_TRANSACTION

(a)
(b)

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

Note:
• a transaction must have a name
• the name must be attached to each operation, which belongs to the transaction
Distributed Transactions

a) A nested transaction
b) A distributed transaction
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.
### The lost update problem

**Transaction** \( T \):

\[
\begin{align*}
balance & = b\text{.getBalance}(); \\
b & .setBalance(balance \times 1.1); \\
a & .withdraw(balance / 10)
\end{align*}
\]

**Transaction** \( U \):

\[
\begin{align*}
balance & = b\text{.getBalance}(); \\
b & .setBalance(balance \times 1.1); \\
c & .withdraw(balance / 10)
\end{align*}
\]

---

<table>
<thead>
<tr>
<th>Step</th>
<th>( T )</th>
<th>( U )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( balance )</td>
<td>$200 | $200</td>
<td></td>
</tr>
<tr>
<td>2. ( b ) setBalance</td>
<td>$220 | $220</td>
<td></td>
</tr>
<tr>
<td>3. ( a ) withdraw</td>
<td>$80 |</td>
<td></td>
</tr>
<tr>
<td>4. ( c ) withdraw</td>
<td>| $280</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 12.5**  
Initial values  
\( a \): \$100,  \( b \): \$200  \( c \): \$300
## The inconsistent retrievals problem

<table>
<thead>
<tr>
<th>Transaction V</th>
<th>Transaction W</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a.withdraw(100)</code></td>
<td><code>aBranch.branchTotal()</code></td>
</tr>
<tr>
<td><code>b.deposit(100)</code></td>
<td></td>
</tr>
<tr>
<td><code>a.withdraw(100); $100</code></td>
<td><code>total = a.getBalance(); $100</code></td>
</tr>
<tr>
<td></td>
<td><code>total = total+b.getBalance(); $300</code></td>
</tr>
<tr>
<td><code>b.deposit(100) $300</code></td>
<td><code>total = total+c.getBalance()</code></td>
</tr>
</tbody>
</table>

Figure 12.6  Initial values  **a:** $200,  **b:** $200
A serially equivalent interleaving of $T$ and $U$

<table>
<thead>
<tr>
<th>Transaction $T$ :</th>
<th>Transaction $U$ :</th>
</tr>
</thead>
<tbody>
<tr>
<td>$balance = b.getBalance()$</td>
<td>$balance = b.getBalance()$</td>
</tr>
<tr>
<td>$b.setBalance(balance*1.1)$</td>
<td>$b.setBalance(balance*1.1)$</td>
</tr>
<tr>
<td>$a.withdraw(balance/10)$</td>
<td>$c.withdraw(balance/10)$</td>
</tr>
</tbody>
</table>

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

Figure 12.7 The result corresponds the sequential execution $T, U$
### A dirty read when transaction $T$ aborts

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th>Transaction $U$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a.getBalance()$</td>
<td>$a.getBalance()$</td>
</tr>
<tr>
<td>$a.setBalance(balance + 10)$</td>
<td>$a.setBalance(balance + 20)$</td>
</tr>
<tr>
<td>$balance = a.getBalance()$ $100$</td>
<td>$balance = a.getBalance()$ $110$</td>
</tr>
<tr>
<td>$a.setBalance(balance + 10)$ $110$</td>
<td>$a.setBalance(balance + 20)$ $130$</td>
</tr>
<tr>
<td><strong>abort transaction</strong></td>
<td><strong>commit transaction</strong></td>
</tr>
</tbody>
</table>

**Figure 12.11**

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)
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
Writeahead Log

x = 0;

y = 0;

BEGIN_TRANSACTION;

x = x + 1;
y = y + 2
x = y * y;

END_TRANSACTION;

(a) A transaction
(b) – d) The log before each statement is executed
Concurrency Control (1)

Transactions

BEGIN_TRANSACTION
END_TRANSACTION

LOCK/RELEASE
or
Timestamp operations

Execute read/write

READ/WRITE

responsible for atomicity!

General organization of managers for handling transactions.
Concurrency Control (2)

- General organization of managers for handling distributed transactions.
Serializability

(a)

BEGIN_TRANSACTION
x = 0;
END_TRANSACTION

(b)

BEGIN_TRANSACTION
x = 0;
END_TRANSACTION

x = x + 1;

(c)

BEGIN_TRANSACTION
x = 0;
END_TRANSACTION

x = x + 2;

(d)

Possible schedules:

| 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 |

a) – c) Three transactions T₁, T₂, and T₃; d) Possible schedules

**Legal**: there exists a serial execution leading to the same result.
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
## Transactions $T$ and $U$ with Exclusive Locks

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th>Transaction $U$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$balance = b \cdot \text{getBalance}()$</td>
<td>$balance = b \cdot \text{getBalance}()$</td>
</tr>
<tr>
<td>$b \cdot \text{setBalance}(bal*1.1)$</td>
<td>$b \cdot \text{setBalance}(bal*1.1)$</td>
</tr>
<tr>
<td>$a \cdot \text{withdraw}(bal/10)$</td>
<td>$c \cdot \text{withdraw}(bal/10)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{openTransaction}$</td>
<td>$\text{lock}\ B$</td>
</tr>
<tr>
<td>$bal = b \cdot \text{getBalance}()$</td>
<td>$\text{lock}\ A$</td>
</tr>
<tr>
<td>$b \cdot \text{setBalance}(bal*1.1)$</td>
<td></td>
</tr>
<tr>
<td>$a \cdot \text{withdraw}(bal/10)$</td>
<td></td>
</tr>
<tr>
<td>$\text{closeTransaction}$</td>
<td>$\text{unlock}\ A, B$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{openTransaction}$</td>
<td>$\text{waits for } T$’s $\text{lock on}\ B$</td>
</tr>
<tr>
<td>$bal = b \cdot \text{getBalance}()$</td>
<td>$\text{lock}\ B$</td>
</tr>
<tr>
<td>$\text{closeTransaction}$</td>
<td>$\text{unlock}\ B, C$</td>
</tr>
</tbody>
</table>

**Figure 12.14**
Two-Phase Locking (1)

Problem: dirty reads?

Two-phase locking (2PL).
Two-Phase Locking (2)

Strict two-phase locking.

Centralized or distributed.
Pessimistic Timestamp Ordering

- Transaction timestamp $\text{ts}(T)$
  - given at \text{BEGIN_TRANSACTION} (must be unique!)
  - attached to each operation
- Data object timestamps $\text{ts}_{\text{RD}}(x), \text{ts}_{\text{WR}}(x)$
  - $\text{ts}_{\text{RD}}(x) = \text{ts}(T)$ of the last $T$ which read $x$
  - $\text{ts}_{\text{WR}}(x) = \text{ts}(T)$ of the last $T$ which changed $x$
- Required serial equivalence: $\text{ts}(T)$ order of $T$’s
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., if tentative versions are used, the final decision is made at END_TRANSACTION
Write Operations and Timestamps

(a) $T_3$ write

Before

After

Time

(b) $T_3$ write

Before

After

Time

(c) $T_3$ write

Before

After

Time

(d) $T_3$ write

Before

After

Time

Figure 13.30

Transaction aborts

Key:

Committed

Tentative

object produced by transaction $T_i$
(with write timestamp $T_i$)

$T_1 < T_2 < T_3 < T_4$
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
Validation of Transactions

Figure 12.28
Validation of Transactions

**Backward validation** of transaction $T_v$

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

**Forward validation** of transaction $T_v$

```java
boolean valid = true;
for (int $T_id = active1; T_id <= activeN; T_id++){
    if (write set of $T_v$ intersects read set of $T_id$) valid = false;
}
```

CoDoKi: Page 547-548
Chapter Summary

- Synchronization
- Clocks
- Logical and vector clocks
- Coordination, elections
- Transactions