Synchronization

- Clock Synchronization
- Logical clocks
- Global state
- Election algorithms
- Mutual exclusion
- Distributed transactions

Clock Synchronization

- Time is counted based on “tick”
- Time judged by query to counter
- Also:
  - Time synchronized from external source

- Fine for centralized system, but:
  - Distributed system?
Example: distributed make

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

Physical clocks

- Solar time
  - 1 second = average time per day / 86400
  - Problem: days are getting longer
- Astronomical time
  - Greenwich mean time
- Atomic time
  - 1 second = 9,192,631,770 transition of Cesium 133 atom
  - TAI: international atomic time
Universal Coordinated Time

UTC:
- TAI seconds are of constant length, unlike solar seconds.
- Leap seconds are introduced when necessary to keep in phase with the sun.

UTC is broadcast via
- Shortwave radio
- Satellite

Clock Synchronization Algorithms

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

• Drift rate:
  \[ 1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho. \]

• To assure no more than \( \delta \) difference
  – synchronize at least every:
  \[ \frac{\delta}{(2\rho)} \]
  seconds.

Cristian's Algorithm

Both \( T_0 \) and \( T_1 \) are measured with the same clock

- \( C_{\text{UTC}} \)
- \( I \), interrupt handling time
- Request
- Time server
- Time server

Getting the current time from a time server

Needs to be adjusted
- UTC response
- Request + response time
- Interrupt handling time
- Only forward adjustments
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

Logical Clock

- Ordering more important than actual time
- **happened-before** relation:
  - If $a$ and $b$ are two events in the same process, and $a$ comes before $b$, then $a \rightarrow b$
  - If $a$ is the sending of a message, and $b$ is the receipt of that message, then $a \rightarrow b$
  - If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$
Logical Clock

• timestamp $C(e)$ for each event $e$

• If $a$ and $b$ are two events in the same process, and $a \rightarrow b$, then we demand that $C(a) < C(b)$

• If $a$ corresponds to sending a message $m$, and $b$ to the receipt of that message, then also $C(a) < C(b)$

Lamport Algorithm

• Each process $P_i$ maintains a **local** counter $C_i$ and adjusts this counter according to the following rules:
  1. For any two successive events that take place within $P_i$, $C_i$ is incremented by 1.
  2. Each time a message $m$ is sent by process $P_i$, the message receives a timestamp $T_m = C_i$.
  3. Whenever a message $m$ is received by a process $P_j$, $P_j$ adjusts its local counter $C_j$:
     \[ C_j = \max\{C_j + 1, T_m + 1\} \]
Lamport Timestamps

Three processes, each with its own clock
The clocks run at different rates

Lamport's algorithm corrects the clocks

Example: Totally-Ordered Multicasting

Updating a replicated database and leaving it in an inconsistent state
Global State

- Consists of local state for each process, plus all messages in transit
- Cut:
  - distributed snapshot of the global state

Global State Cut examples

(a) consistent cut
(b) inconsistent cut

Sender of m2 cannot be identified with this cut
Global State determination 1/2

Organization of a process and channels for a distributed snapshot

Global State determination 2/2

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

- Need to select process
  - Act as coordinator
  - Act as initiator
  - Special purpose
- Selection from equals
- Assumption: process has unique number

Bully Election Algorithm 1/2

- 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
Bully Election Algorithm 2/2

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

A Ring Election Algorithm

Previous coordinator has crashed

No response

Election message

22
Mutual Exclusion

- Gain safe access to shared resource in a distributed system
- **solutions:**
  - Via a centralized server
  - Completely distributed with no topology imposed
  - Completely distributed, making use of a (logical) ring

---

**Mutual Exclusion: A Centralized Algorithm**

(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, when then replies to 2
A Distributed Algorithm

a) Two processes want to enter the same critical region at the same moment
b) Process 0 has the lowest timestamp, so it wins
c) When process 0 is done, it sends an OK also, so 2 can now enter the critical region

A Token Ring Algorithm

a) An unordered group of processes on a network.
b) A logical ring constructed in software.
c) A token is passed around the ring to signify access right
Mutual Exclusion Algorithms 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>

Distributed Transactions

- Transaction model
- Transaction classifications
- Implementations
- Concurrency control
The Transaction Model (1)

Updating a master tape is fault tolerant

The Transaction Model (2)

<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>

Examples of primitives for transactions
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;
END_TRANSACTION

ABORT_TRANSACTION

(b)

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

Transaction properties

• Atomic
  – To the outside world, the transaction happens indivisibly

• Consistent
  – The transaction does not violate system invariants

• Isolated
  – Concurrent transactions do not interfere with each other

• Durable
  – Once a transaction commits, the changes are permanent
Distributed Transactions

Implementations

- Use a **private workspace**
  - Each client gets its own copy of the (part of the) database
  - When things go wrong delete copy, otherwise commit the changes to the original

- Example:
  - File system

- Write-ahead log
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

Write-ahead 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
Concurrent control

- **Problem:**
  - Increase efficiency by allowing several transactions to execute at the same time

- **Constraint:**
  - Effect should be the same as if the transactions were executed in some serial order
Concurrency Control Management 2/2

General organization of managers for handling distributed transactions.

Serializability

BEGIN_TRANSACTION
x = 0;
x = x + 1;
END_TRANSACTION
BEGIN_TRANSACTION
x = 0;
x = x + 2;
END_TRANSACTION
BEGIN_TRANSACTION
x = 0;
x = x + 3;
END_TRANSACTION

• Potential values for x: 1, 2, 3

• Schedules

<table>
<thead>
<tr>
<th>Schedule</th>
<th>x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3</th>
<th>Legal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedule 2</td>
<td>x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 3;</td>
<td>Legal</td>
</tr>
<tr>
<td>Schedule 3</td>
<td>x = 0; x = 0; x = x + 1; x = 0; x = x + 2; x = x + 3;</td>
<td>Illegal</td>
</tr>
</tbody>
</table>
Concurrency control implementation

- Transactions are a sequence of read and write operations
- Goal: properly schedule conflicting operations
  - Read-write conflict
  - Write-write conflict
- Multiple read operations to not conflict
- 2 approaches
  - Pessimistic vs. optimistic

Two-Phase Locking
Distributed Two-Phase Locking

- **Centralized 2PL:**
  - A single site handles all locks

- **Primary 2PL:**
  - Each data item is assigned a primary site to handle its locks. Data is not necessarily replicated

- **Distributed 2PL:**
  - Assumes data can be replicated. Each primary is responsible for handling locks for its data, which may reside at remote data managers.

---

Two-Phase Locking

- **Problem 1:**
  - System can come into a **deadlock**
  - Practical solution: put a timeout on locks and abort transaction on expiration

- **Problem 2:**
  - When should the scheduler actually release a lock:
    1. when it knows that no more locks will be requested
    2. when all operations have been executed
      - **Strict two-phase locking**
**Strict Two-Phase Locking**

![Diagram of Strict Two-Phase Locking]

**Timestamp ordering**

- Transaction manager assigns a unique timestamp to each transaction
- Every data item is time-stamped with the timestamp of the transaction that read and wrote it
- If 2 operations conflict then the data manager processes the one with the lowest timestamp first
Timestamp Ordering

Concurrency control using timestamps