Fundamentals of Distributed Systems

Clock Synchronization

Mutual Exclusion

Deadlocks

Agreement

Concurrency Control

Clock Synchronization

Problem: In a distributed system the notion of global time does not exist.

Assume: A central clock broadcasts periodically global time to all processes.

Problem: Due to different delays of messages in the network, two processes receive message with the same timestamp at different times.

Assume: Each computer in the system has a local physical clock and computers try to synchronize them.

Problem: Physical clocks tend to drift and the clock drift is different at each computer.

Solution to clock synchronization:
• Abandon the idea of physical time.
• Introduce logical (or virtual) time.
• Synchronize logical clocks at each computer.
Lamport’s Logical Clocks

• Lamport presents a scheme for ordering events in a distributed system.

• Approach:
  • Define a partial ordering of events.
  • Extend the definition to a total ordering.

Here:
• Execution of processes is viewed as a sequence of events.
• All events in a single process are totally ordered.
• Receiving and transmitting a message are events.

Some Background on Orderings

Partial Ordering, Total Ordering:

• Given a set \( A \) of elements and a relation \( R \).

• \( R \) defines a partial ordering in \( A \) if:
  1. \( R \) is reflexive,
     \( (aRa \text{ for all } a \in A) \).
  2. \( R \) is antisymmetric,
     \( (\text{if } aRb \text{ and } bRa, \text{ then } a = b) \).
  3. \( R \) is transitive.
     \( (\text{if } aRb \text{ and } bRc, \text{ then } aRc) \).

• A partial ordering \( R \) is a total ordering in \( A \) if for all \( a \in A \) and \( b \in A \), either \( aRb \) or \( bRa \).

• An ordering is said to be irreflexive, if it is not reflexive.
Happened Before Relation

- The Happened Before Relation ("\(\rightarrow\)) describes causal ordering of events:

1. If \(a\) and \(b\) are events in the same process, and \(a\) comes before \(b\), then \(a \rightarrow b\).

2. If \(a\) is the sending of a message \(m\) by some process, and \(b\) is the receipt of the same message \(m\) by some other process, then \(a \rightarrow b\).

3. If \(a \rightarrow b\) and \(b \rightarrow c\), then \(a \rightarrow c\).

Also: For all events \(a \rightarrow a\) does not hold.

- Concurrent Events:
  Two events \(a\) and \(b\) are concurrent (\(a \parallel b\)), if neither \(a \rightarrow b\) nor \(b \rightarrow a\).

Example: Order the events in the following space-time diagram according to "\(\rightarrow\)."
Lamport’s Logical Clocks

- **Goal:** Implement relation “⇒” in a distributed system.

- There is one clock $C_i$ for each process $P_i$. These clocks do not correspond to physical clocks, and, thus, are called *logical clocks*.

- A clock $C_i$ assigns a value $C_i(a)$ to any event $a$ in a process $P_i$. $C_i(a)$ is the *timestamp* of event $a$.

- The values assigned by $C_i$ are *monotonically increasing* values.

**System of logical clocks**

- A *system of logical clocks* is represented by a function $C$ which assigns a number $C(a)$ to any event $a$ in the distributed system.

Logical Clocks

- **Clock Condition:** If $a \Rightarrow b$, then $C(a) < C(b)$.

  *Basically:* If event $a$ occurs before event $b$, then the system of clocks should show $C(a) < C(b)$.

  *Note:* The other direction does not work. That is, if the $C(a) < C(b)$, it is not determined if a really occurred before $b$.

- **Correctness Conditions:** The following conditions must be satisfied by the (local) clocks $C_i$ to satisfy the above clock condition:

  - (C1) For any two events $a$ and $b$ on the same process $P_i$, $a$ occurs before $b$ if $C_i(a) < C_i(b)$.

  - (C2) If $a$ is the event of sending a message $m$ in process $P_i$, and $b$ is the event of receiving the same message $m$ at $P_j$ then $C_i(a) < C_j(b)$. 
Implementation of Logical Clocks

The following implementation rules (IR) enforce that conditions (C1) and (C2) (and therefore the clock condition) hold:

(IR1) Clock $C_i$ is incremented between any two successive events in process $P_i$:

\[ C_i := C_i + d \quad (d > 0) \]

That is: If $a$ and $b$ are two successive events and $a \rightarrow b$, then $C_i(b) = C_i(a) + d$.

(IR2) (a) If $a$ is the sending of a message $m$ by process $P_i$ then the message is timestamped with $t_m = C_i(a)$.

(b) If $b$ is the event of receiving message $m$ by process $P_j$ then $C_j$ is set to a value greater than or equal to its present value and greater than $t_m$:

\[ C_j := \max (C_j, t_m + d) \]

Obtaining a total ordering with logical clocks

- A total ordering of all events can be obtained with a total ordering “<<” of the processes. (“<<” breaks ties between processes).

- Total ordering of events ($a \Rightarrow b$):
  - If $a$ is an event in process $P_i$, and $b$ is an event in process $P_j$, then $a \Rightarrow b$ if and only if
    - (a) $C_i(a) < C_j(b)$, or
    - (b) $C_i(a) = C_j(b)$ and $P_i << P_j$.

- Note that the total ordering “⇒” depends on the system of clocks. In contrast, the partial ordering “⇒” is uniquely determined by the system of events.
Vector Clocks

Problem with Lamport’s Logical Clocks:
• If $C(a) < C(b)$ one cannot follow that $a \rightarrow b$.
• More general: Timestamps do not necessarily say something about causal order.

• Improvement through Vector Clocks (1988).
  (Here: for a system of $n$ processes)

The clock at a process $P_i$ is a vector
$C_i = (C_i[1], C_i[2], ..., C_i[n])$
• $C_i[i]$ is $P_i$’s own logical time.
• $C_i[j]$ ($i \neq j$) is $P_i$’s best guess of $P_j$’s logical time (based on the reception of messages).

Implementation of Vector Clocks

(IR1) Clock $C_i$ is incremented between any two successive events in process $P_i$:
$C_i[i] := C_i[i] + d$ \hspace{1cm} (d>0)

(IR2) (a) If $a$ is the sending of a message $m$ by process $P_i$ then the message is timestamped with (a vector timestamp) $t_m=C_i(a)$.
(b) If $b$ is the event of receiving message $m$ by process $P_j$ then $C_j$ is updated as follows:
$\forall k: C_j[k] := \max(C_j[k], t_m[k])$

Then the following holds:
$\forall i: \forall j \hspace{0.5cm} C_i[i] \preceq C_j[i]$
Properties of Vector Clocks

Vector timestamps $t^a$ and $t^b$ for two event $a$ and $b$ are defined as follows:

- **Equal:** $t^a = t^b$ iff. $\forall i: t^a[i] = t^b[i]$
- **Not equal:** $t^a \neq t^b$ iff. $\exists i: t^a[i] \neq t^b[i]$
- **Less than or equal:** $t^a \leq t^b$ iff. $\forall i: t^a[i] \leq t^b[i]$
- **Less than:** $t^a < t^b$ iff. $t^a \leq t^b$ and $t^a[i] \neq t^b[i]$
- **Concurrent:** $t^a \parallel t^b$ iff. (not $t^a < t^b$) and (not $t^b < t^a$)

**Causal ordering:**

If $t^a < t^b$ or $t^a < t^b$ then events are causally ordered. Otherwise, the events are concurrent.

We obtain:

$a \rightarrow b$ iff. $t^a < t^b$

... and this solves the problem with Lamport’s clocks.

---

**Vector Clocks**

**Example:** Advancing of vector clocks in a system of three processes.
Distributed Mutual Exclusion

More difficult than centralized mutual exclusion:
- **No shared memory**
  → semaphores cannot be used.
- **No common physical clock.**
- **Unpredictable network delays.**

- Several solutions to mutual exclusion in distributed systems. They can be grouped into

  **Centralized Algorithms:**
  - Simple, but susceptible to failures.

  **Non-Token-based Algorithm:**
  - Distributed
  - Apply some sort of logical clocks.

  **Token-Based Algorithms:**
  - Distributed
  - A process can enter the CS if it owns the token.

A Centralized Algorithm

- One process is the *coordinator*.
- Whenever a process wants to enter the critical section (CS), it sends a request to coordinator.
- The coordinator grants permission to enter the CS. If a process is in a CS, incoming requests will be queued.
Non-Token Based Algorithm

- There are several other non-token based algorithms for mutual exclusion (consult the textbook).
- We discuss the mutual exclusion algorithm by Lamport.

**Lamport’s Algorithm:**

- **Requirements:**
  1. A process in the critical section (CS) must release it before it can be granted to another process.
  2. Requests must be granted in the order in which they are made.
  3. If every process releases CS, then each request will eventually be met.

- **Assumptions on the Network:**
  1. Messages between two processes are received in the order in which they are sent.
  2. Every message is eventually received.
  3. Each process can send a message to every other process.

Lamport’s Mutex Algorithm

- N processes: $P_1, P_2, \ldots, P_N$
- Each process keeps a queue of requests for entering the CS, $request\_queue_i$.
- Entries in $request\_queue_i$ are ordered by a timestamp $(ts_i, i)$.
- Assume total ordering $\Rightarrow$ of events:

1. **Requesting the critical section:**
   1. If $P_i$ wants to receive the resource, it sends a message “Request$(ts_j, i)$” to all processes in its request set $R_i$, and puts the message in its own $request\_queue_i$.
   2. When $P_j$ receives a message “Request$(ts_i, i)$” from $P_i$, it puts the message in its $request\_queue_j$, and sends a timestamped REPLY message to $P_i$. 
Lamport’s Mutex Algorithm

(2) Entering the CS:
P_i can enter the CS if the following conditions are met:

(L1) P_i has received a message with times-tamp > (ts_i, i) from all other processes.

(L2) The "Request(ts_i, i)" is at the top of request_queue_i.

(3) Leaving the CS:

• P_i removes the "Request(ts_i, i)" from the top of the request_queue_i and sends a times-tamped RELEASE message to all processes in R_i.

• When P_j receives a RELEASE message from P_i, it deletes the "Request(ts_i, i)" from its request_queue_j.

Example

• P_1 and P_2 want to enter the CS:

• P_2 enters the CS:
Example

- P₂ leaves the CS and P₁ enters the CS:

```
P₁: 2, 1 1, 2
P₂: 2, 1 1, 2
P₃: 2, 1 1, 2
```

here P₁ enters the CS

Distributed Mutual Exclusion with a Token Based Algorithm

- In token-based algorithms for distributed mutual exclusion all processes are positioned in a logical ring.

  Note: It is not required that the network is a ring network.

**Basic Idea:**
- Processes exchange a (unique) token.
- Process which holds the token can enter CS.

**Features:**
- Token-based mutual exclusion algorithms typically use sequence numbers (rather than timestamps.) Each time a process wants the token, it increments a sequence number.
- Proof of correctness of token-based algorithm is trivial. However, starvation and deadlock freedom must be addressed.
Suzuki-Kasami Algorithm

- There are several algorithms for token-based mutual exclusion.
- We discuss the Suzuki-Kasami algorithm (1985).

**Principle idea:**
- A process which requests the CS broadcasts a REQUEST message to all other processes.
- The process which has the token will send the token to the requesting process after it has left the CS.

**Data Structures:**

- **REQUEST** \((j,n)\) Request message from process \(P_j\) for its \(n\)-th request to access the CS.
- \(RN_i[1..N]\) \(RN_i[j]\) is largest sequence number in a request message from \(P_j\) that was received by \(P_i\).

- If \(P_i\) receives a message \(REQUEST (j,n)\) it sets \(RN_i[j] = max (RN_i[j], \ n)\).
- If \(RN_i[j] > n\), then the message is outdated.
Suzuki-Kasami Algorithm

**Token:**

- $Q$ Queue of requesting processes.
- $LN[1..N]$ Array of integers
  - $LN[j]$ is the sequence number of the request that $P_i$ executed most recently.

- When leaving the CS, process $P_i$ sets $LN[i] := RN_i[j]$.
- At $P_j$: If $RN_i[j] = LN[i] + 1$, then $P_j$ is currently requesting the token.
- When leaving the CS, a process updates the $Q$ based on the comparison of $RN_i[j]$ and $LN[i]$.

(1) **Requesting the CS:**

- If $P_i$ wants to receive the resource and does not have the token, it sets $RN_i[i] := RN_i[i] + 1 (= sn)$ and sends $REQUEST(i, sn)$ to all processes.
- When $P_j$ receives a message $REQUEST(i, sn)$, it sets $RN_j[i] := max (RN_j[i], sn)$.
  - If $P_j$ has the token and does not hold the resource and $RN_j[i] = LN[i] + 1$, it sends the token to $P_i$.

(2) **Using the resource:**

- If $P_i$ has the token, it can use the resource.

(3) **Releasing the resource (by $P_i$)**

- Set $LN[i] := RN_i[i]$.
- For all processes $P_j$ with $RN_j[i] = LN[i] + 1$ and who are not in queue $Q$, add $P_j$ to $Q$.
- If $Q$ is non-empty, delete the first entry, say $P_k$, and send the token to $P_k$.
Example

- P₁ and P₂ wish to enter the CS:

- Why is the Suzuki-Kasami algorithm free of starvation and deadlocks?
- How many messages are sent for each assignment of a resource (assuming there are N processes)?

Deadlocks in Distributed Systems

Types of Deadlocks in Distributed Systems
- Communication Deadlock
  Assume: A process can block on a receive or send operation.
  Then a deadlock can occur for a set of processes, if each process in the set is blocked at some other process.
- Resource Deadlock
  Assume: A process can wait for a set of resources while holding other resources.
  Then a deadlock can occur for a set of processes, if each process in the set waits for a resource held by another process.

Wait-For-Graphs (WFG)
- Processes are nodes. If Pᵢ is blocked at Pⱼ, then there is an edge Pᵢ → Pⱼ.
- A system of processes is deadlocked if and only if the WFG has a cycle or a knot.
Deadlocks

What to do about Deadlocks?

1. Deadlock Detection
2. Deadlock Prevention
3. Deadlock Avoidance

✗ Deadlock prevention reduces concurrency.
✗ Deadlock avoidance has considerable overhead.

Deadlock Detection

- Principles of a deadlock detection algorithm:
  - Maintain Wait-For-Graph (WFG)
  - Search for cycles (or knots) in WFG.
- Two basic approaches:
  - centralized
  - distributed
- Problem: **Phantom deadlocks.**
  Occurs if a detection algorithm reports a deadlock that does not exist.
  Phantom deadlocks are mostly due to absence of a common clock, lack of shared memory, or unpredictable network delays.
Centralized Deadlock Detection

- **General approach**: Extend the notion of the *resource allocation graph* to a distributed environment.

  (1) Each site keeps its own version of the local resource allocation graph.
  (2) A *central coordinator* contains the global allocation graph. If the coordinator detects a cycle in the graph, it terminates a process.

- **Problem**: *Phantom deadlocks* possible.
  (a) M0 releases R and sends a message to C.
  (b) M0 requests access to T, and sends message to C.
  Phantom deadlock if second message arrives earlier than first message.

✘ All sites must send *request resource* and release resource messages to the coordinator even if the request is local.
✘ Coordinator is single point of failure.
Distributed Deadlock Detection

• Several distributed deadlock detection algorithms have been proposed.
• Here: Chandy-Misra-Haas Algorithm.

• Principle idea:
  • Special messages (probes) are sent along the edges of the WFG.
  • If a blocked process receives a probe, it forwards the probe on the outgoing edge of the WFG.
  • If a probe returns to its originator, a deadlock exists.

• Assume: Processes can request multiple resources at one time; and are blocked until all resources all available.

Chandy-Misra-Haas Algorithm

• Probe is a triple \((i, j, k)\)
  • \(i\) : Deadlock detection started by \(P_i\).
  • \(j\) : Sender of this probe is \(P_j\).
  • \(k\) : Receiver of this probe is \(P_j\).

• Algorithm:
  
  (1) Whenever a process, say \(P_i\), becomes blocked due to another process, say \(P_j\), it sends a probe packet \((i, i, j)\) to process \(P_j\).
  
  (2) If \(P_j\) receives the packet \((i, i, j)\), and \(P_j\) itself is blocked due to a process, say \(P_k\), it sends a message \((i, j, k)\) to process \(P_k\).
    If \(P_j\) is not blocked no message is sent.
  
  (3) If the probe packet returns to process \(P_i\), then a deadlock exists.
Chandy-Misra-Haas Algorithm

- Example:

```
P1  P2  P3
Site 1

(1, 3, 4)

P9  P8
P6  P4
P5  P7
P10
Site 2

Site 3
```

Distributed Agreement

- Processors in a distributed system must often reach some form of agreement on the state of the distributed system.

- Typical way of reaching agreement between processes:
  - Take majority.
  - Exchange numbers and take maximum.

- Problem: Reach agreement even if some processors do not cooperate.

- Agreement protocols: Reach a distributed agreement between processors, even if some of the processors
  - do not respond.
  - crash
  - send wrong messages.

Agreement is found between nonfaulty processors.
System Assumptions

Processor failures:
• The number of failing processors is limited.
• Three types of processor fault:
  • Crash fault: Processor stops functioning.
  • Omission fault: Processor does not send some messages that it is expected to send.
  • Malicious: Processor may send falsified messages.

Communication network
• Processors can directly communicate with each other via messages.
• All messages have the same network delay.
• No communication errors.

Authentication of messages
• Receiving processor knows identity of the sender.
• Messages are not authenticated (“oral”).

Byzantine Generals Problem

• There is one General and \( n-1 \) Lieutenants.
• The General gives an order “attack” or “retreat” to the lieutenants.
• General and Lieutenants are either “loyal” or “traitors”.

Byzantine Generals Problem: The general sends an order to his \( n-1 \) lieutenants such that
  (IC1) All loyal lieutenants obey the same order.
  (IC2) If the general is loyal, then every loyal lieutenants obeys the order he sends.

• Situation: A processor broadcasts a value to all other processors.
• Agreement: All nonfaulty processes agree on the same value.
• Validity: If the source processor is nonfaulty, then all nonfaulty processors agree on the value sent by the source processor.
Why is the problem hard?

- **Assume:** 1 general and 2 lieutenants; one of the three is a traitor.
- **Claim:** Agreement cannot be reached!

**Proof:**
- On the left: To satisfy (IC2), *Lieut. I* must attack.
- On the right: Commands look the same --> *Lieut. I* must attack.
- Hence, whenever *General* sends attack to loyal *Lieut. I* he must attack.
- Similarly: Whenever *General* sends retreat to loyal *Lieut. II*, he must retreat.
- Thus: *Lieut. I* and *II* violate (IC1) in the right figure.

Byzantine Generals Problem is hard

**Generalization:** Generals = Lieutenants+General.
- In other words: everyone can give an order.

*Then we have shown:*
There exists no solution to the Byzantine Generals problem with 3 generals that can cope with 1 traitor.

*In general:*
There exists no solution to the Byzantine Generals problem with less than $3m+1$ generals that can cope with $m$ traitors.
Solution to Byzantine Generals Problem

Solution by Lamport/Shostak/Pease (1982):

- $3m+1$ generals with at most $m$ traitors
  (1 commander and $3m$ lieutenants).

- Generals can send “oral” messages to each other with the following properties:
  
  $(A0)$ Every general can sent a message to every other general.
  $(A1)$ Every message that is sent is delivered correctly.
  $(A2)$ The receiver of a message knows who sent it.
  $(A3)$ The absence of a message can be detected.

More assumptions:
- A traitorous commander may not send any order.
- For a lieutenant, the default order is “retreat”.

The algorithm works with a set of values $v_i$ for each processor $i$.
So, for the attack/retreat scenario we would have:
$$v_i \in \{\text{retreat, attack}\}.$$

For reaching an agreement each general uses a function $\text{majority}$.
- $\text{majority} \ (v_1, v_2, \ldots, v_{n-1})$ is the majority value in $v_1, v_2, \ldots, v_{n-1}$.
- If no majority value exist, then $\text{majority} \ (v_1, v_2, \ldots, v_{n-1}) = \text{retreat}.$
Oral Message Algorithm

- The Oral Message Algorithm $OM(m)$ solves the Byzantine Generals problem for $3m+1$ or more generals under the assumption that there are at most $m$ traitors.

- **Idea:** Successively divide the set of generals into smaller groups, and achieve the agreement recursively.

- **Note:** Step 3 is applied when the recursion unfolds. It selects the majority of the values received in a round of message exchanges.

Algorithm $OM(0)$:

1. The commander sends his value to every lieutenant.
2. Each lieutenant uses the value he receives from the commander, or uses the value $\text{retreat}$, if he receives no value.

Algorithm $OM(m)$, $m>0$:

1. The commander sends value to every lieutenant.
2. For each $i$, let $v_i$ be the value that Lieutenant $i$ receives from the commander, or $v_i = \text{retreat}$ if Lieutenant $i$ does not receive a value from the commander. Lieutenant $i$ acts as the commander in Algorithm $OM(m-1)$ where he sends the value $v_i$ to each of the $n-2$ lieutenants.
3. For each $i$, and each $j \neq i$, let $v_j$ be the value Lieutenant $i$ receives from Lieutenant $j$ (in Step 2 in Algorithm $OM(m-1)$), or else $\text{retreat}$, if he received no such value. Lieutenant $i$ uses the value $\text{majority} (v_1, v_2, \ldots, v_{n-1})$. 
Oral Message Algorithm

Example

Example 1: $n=4$ generals, and
1 traitor (=Lieutenant 3).
correct value = v
traitors send = x

$OM(1).Step 1$

$OM(1).Step 2$: for $i=$Lieut. 1
now execute: $OM(0)$

$OM(1).Step 2$: for $i=$Lieut. 2
now execute: $OM(0)$

$OM(1).Step 2$: for $i=$Lieut. 3
now execute: $OM(0)$

• Next: $OM(1).Step 3$:
  • for $i=$Lieut. 1 $\rightarrow v_1 =$
  • for $i=$Lieut. 2 $\rightarrow v_2 =$
  • for $i=$Lieut. 3 $\rightarrow v_3 =$
**Oral Message Algorithm**

**Example**

**Example 2:** Same Example but Commander is the traitor.
- **correct value** = \(v\)
- **traitors send** = \(x, y, z\)

**MO(1).Step1**

- In **MO(1).Step3:**
  - for \(i=Lieut. 1\) --> \(v_1 = \text{majority}(x, y, z)\)
  - for \(i=Lieut. 2\) --> \(v_2 = \text{majority}(x, y, z)\)
  - for \(i=Lieut. 3\) --> \(v_3 = \text{majority}(x, y, z)\)

---

**Comments on \(OM\) Algorithm**

**Complexity of executing \(OM(m)\) with \(n\) generals**

- \(OM(m)\) makes \(n-1\) calls to \(OM(m-1)\) \(n\).
- Each call to \(OM(m-1)\) results in \(m-1\) calls to \(OM(m-2)\).
- ..... and so forth.
- Thus: There are \((n-1)(n-2)(n-3)\ldots(n-m+1)\) executions of algorithm \(OM(k)\) for \(k=n-1,n-2,\ldots,n-m+1\).
- Message complexity of \(OM(m)\) with \(n\) generals: \(O(n^m)\).

**Results for \(OM(m)\)** (Proofs are in the paper/book):

- **Lemma 1:**
  
  *For any \(m\) and \(k\), Algorithm \(OM(m)\) satisfies \((IC2)\) if there are more than \(2k+m\) generals and at most \(k\) traitors.*

- **Theorem 1:**
  
  *For any \(m\), Algorithm \(OM(m)\) satisfies conditions \((IC1)\) and \((IC2)\) if there are more than \(3m\) generals and at most \(m\) traitors.*
Concurrency Control in Distributed Systems

- We will use the notion of a transaction to describe a high level operation in a distributed system. A transaction performs `READ()` and `WRITE()` operations on global data (=database).

- Properties of a Transaction:
  - **Serializability** -- If a number of concurrently executed transactions complete execution, the result must be the same as some serial execution of the same transaction.
  - **Atomicity** -- transactions appear as a single indivisible operation. ("all or nothing property").
  - **Consistency** -- When a transaction completes execution it has changed the database from one consistent state to another consistent state.
  - **Permanence** -- After a transaction terminates, the changes to the database are permanent.

Transactions

- Example:

  **T1:**
  
  ```
  T1: T2:
  READ(A);   READ(A);
  A:= A-50;  tmp:= A*0.1;
  WRITE(A);  A:=A-tmp;
  READ(B);   WRITE(A);
  B:=B+50;   READ(B);
  WRITE(B);  B:=B+tmp;
  WRITE(B);
  ```

Schedules

- **Schedule**: Sequence of instructions from a transaction.

- **Serial Schedule**: Schedule where all instructions of a transaction appear as a group.

- **Serializable Schedule**: Schedule where the outcome is equivalent to a serial schedule.

(Non-) Serializable Schedules

- Example:

  ```
  \begin{align*}
  T1 & \\
  \text{READ}(A); & \\
  A := A - 50; & \\
  \text{READ}(A); & \\
  \text{tmp} := A \times 0.1; & \\
  A := A - \text{tmp}; & \\
  \text{WRITE}(A); & \\
  \text{WRITE}(A); & \\
  \text{READ}(B); & \\
  B := B + 50; & \\
  \text{WRITE}(B); & \\
  \text{READ}(B); & \\
  B := B + \text{tmp}; & \\
  \text{WRITE}(B); & \\
  \end{align*}
  
  T2
  ```

- The schedule makes the database inconsistent. Also, the schedule is not serializable.

- **Note**: Non-serializable schedule may lead to an inconsistent database (but not necessarily).
Precedence Graph for Transactions

Vertices: the set of transactions

Edges: Add an edge $T(i) \rightarrow T(j)$ ...

... if $T(i)$ does a READ($X$) and $T(j)$ does a WRITE($X$) after,
... if $T(i)$ does a WRITE($X$) and $T(j)$ does a READ($X$) after,
... if $T(i)$ does a WRITE($X$) and $T(j)$ does a WRITE($X$) after.

Theorem:
If the precedence graph of a schedule has no cycle then the schedule is serializable.

Goal of Concurrency Control

- Goal: Maintain consistency of the database by only allowing serializable schedules.

- Two general approaches:
  - Locking
  - Timestamping
Locking

• Each object must be “Lock”-ed before it can be accessed.
  • Shared Lock (LS): read object.
  • Exclusive Lock (LX): write/read object.

<table>
<thead>
<tr>
<th></th>
<th>LS</th>
<th>LX</th>
</tr>
</thead>
<tbody>
<tr>
<td>LS</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>LX</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

• Compatibility matrix:

• Locking alone does not guarantee serializability!

Example:

T1:

LX (B)
READ (B)
B := B - 50
WRITE (B)
UNLOCK (B)

T2:

LS (A)
READ (A)
UNLOCK (A)
LS (B)
READ (B)
UNLOCK (A)

LX (A)
READ (A)
A := A + 50
WRITE (A)

• Non-serializable:
  T2 sees the old value of A, but the new value of B.
Two-Phase Locking (2-PL)

- Each transaction has two phases.
  - Phase 1 = growing phase: Locks can only be requested in the growing phase.
  - Phase 2 = shrinking phase: In the shrinking phase, locks can only be released and no new locks can be requested.

- 2-PL yields serializable schedules.

Timestamp Ordering

- Each transaction is assigned a unique timestamp $\tau_s(T_i)$ (e.g., with Lamport’s algorithm).
- $\tau_s(T_i) < \tau_s(T_j)$ is interpreted as “$T_i$ is earlier than $T_j$.”

- Each data item $Q$ has two timestamps:
  - $w_{\tau_s}(Q)$: largest timestamp of any transaction that did a write on this item.
  - $r_{\tau_s}(Q)$: largest timestamp of any transaction that did a read on this item.
**Timestamp Ordering**

**T**\(_i\) does a \texttt{READ}(Q):

\[
\text{if } ts(T_i) < w_{ts}(Q) \\
\quad \text{then} \\
\quad \quad \text{Abort}(T_i) \\
\text{else} \\
\quad r_{ts}(Q) := \max(r_{ts}(Q), ts(T_i))
\]

\[T_i\] does a \texttt{WRITE}(Q):

\[
\text{if } ts(T_i) < \max(r_{ts}(Q), w_{ts}(Q)) \\
\quad \text{then} \\
\quad \quad \text{Abort}(T_i) \\
\text{else} \\
\quad w_{ts}(Q) := ts(T_i)
\]

**Example:**

\begin{align*}
T1: & \quad \text{READ}(B) \quad \text{READ}(B) \\
& \quad B := B - 50 \quad \text{WRITE}(B) \\
& \quad \text{READ}(A) \quad \text{READ}(A) \\
& \quad \text{DISPLAY}(A+B) \quad \text{DISPLAY}(A+B) \\
T2: & \quad \text{WRITE}(A) \quad \text{WRITE}(A) \\
& \quad A := A + 50 \quad \text{DISPLAY}(A+B)
\end{align*}

- Assume: \(ts(T_i) = 10\) and \(ts(T_j) = 20\).
- \(w_{ts}(Q) = r_{ts}(Q) = 0\).

- Is the given schedule successful?
- Is the same schedule successful in 2-PL?
Problems with Concurrency Control

Problems with Timestamping:

Infinite restarts:
- Assume $t_s(T1) < t_s(T2)$.

<table>
<thead>
<tr>
<th>T1:</th>
<th>T2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ (A)</td>
<td>READ (A)</td>
</tr>
<tr>
<td>WRITE (A)</td>
<td>WRITE (A)</td>
</tr>
</tbody>
</table>

- Now WRITE (A) by T1 fails, and T1 is restarted with a (new) timestamp $t_s(T1) > t_s(T2)$.
- If T1 performs READ (A) before T2 does WRITE (A), then T2 is restarted.
- ... and so on.

Unnecessary restarts:
- Occurs if the actual sequence of execution is in the reverse order of timestamps.
- The schedule may be serializable, but the older transaction may be restarted.

Problems with Locking:

- (2-PL) Locking can result in deadlocks (Note: no deadlocks with timestamp ordering). Thus, a deadlock detection or prevention algorithm is required.

<table>
<thead>
<tr>
<th>T1:</th>
<th>T2:</th>
<th>T2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>LX (A)</td>
<td>LX (B)</td>
<td>LX (C)</td>
</tr>
<tr>
<td>LX (C)</td>
<td>LX (A)</td>
<td></td>
</tr>
<tr>
<td>LX (B)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Deadlock detection ✓
- Deadlock Prevention:
  - Obtain all locks before transactions starts execution.
    - ✗ starvation
    - ✗ low concurrency
  - Preemption and Rollback.
Preemption and Rollback
• Methods to prevent deadlocks with 2-PL use timestamps to decide which transaction gets aborted:
  • *Wait-Die Protocol*
  • *Wound-Wait-Protocol*

*Wait-Die Protocol*: Old transaction waits for completion of younger transaction.
• Assume \( T_i \) wants an item that \( T_j \) has locked:
  \[
  \text{if } ts(T_i) < ts(T_j) \quad \text{then} \]
  \( T_i \) waits
  \[
  \text{else} \quad T_j \text{ aborts}
  \]

• Assume \( T_i \) wants an item that \( T_j \) has locked:
  \[
  \text{if } ts(T_i) > ts(T_j) \quad \text{then} \]
  \( T_i \) waits
  \[
  \text{else} \quad T_j \text{ aborts}
  \]

Fault Tolerance
• **Goal**: Maintain consistency in the presence of errors.

**Atomic Actions**:
• Central concept for fault-tolerance.
  • An atomic action is indivisible and instantaneous.
  • Processes performing an *atomic action*...
    (1) ... are not aware of the existence of processes not involved in the atomic action.
    (2) .... do not communicate with other processes while action is performed.
    (3) ... cannot detect any state changes other than their own.
Recoverability

- If a failure prevents an atomic action from completing successfully, all objects that were modified by the action must be restored to the state at the beginning of the action (=Recoverability).

- Recoverability implies:
  - An action does not release updated objects until it is completed.
  - Any modified objects can be reconstructed to its original state (must keep a log which allows undoing and re-doing the action).

Transactions as Atomic Actions

- The atomicity property of transactions requires that transactions are atomic actions.
- Each transaction performs a commit operation at the end to make any changes to the database final.
- If a transaction terminates before the commit is issued all changes to the database must be undone ("rollback").

- Feasible outcomes:

  successful    “suicidal”    “murdered”
  BEGIN         BEGIN         BEGIN
  action        action        action
  action        action        action
  ....          ....          .....  
  action        action        action
  Commit        ABORT        <---ABORT


Cascading Rollback

- Problem with rollbacks: The rollback of a transaction may require the rollback of other transactions ("cascading rollback")

- Cascading Rollback can be avoided by introducing periodic checkpoints (checkpoints are not covered here).

Committing an atomic action in a distributed environment

- Assume a distributed system where multiple processes (=cohorts) are involved in a single (atomic) transaction.

- When committing the transaction, all cohorts must agree to commit or abort the transaction.

- Commit protocol:
  - Protocol that ensures agreement on the outcome of an atomic action even in the presence of failures.

- Problem of commit protocols: General’s Paradox
General’s Paradox

- There are 2 armies on different sides of a hill.
- The armies want to capture the hill.
- The armies can succeed only if both attack simultaneously.
- If only one army attacks, it will be defeated.
- The two armies can communicate with each other by sending messengers.
- The messenger can get captured.

- It can be shown that there is no finite solution to the General’s Paradox.

However, by relaxing the assumption on the finite length, a protocol can be given.

One such protocol is the **Two-Phase commit protocol**.

Two Phase Commit Protocol

- For the protocol, one process acts as the *Coordinator*.

**Phase I:**
- At coordinator:
  1. Send a `PREPARE` message to each cohort.
  2. Wait for reply from each cohort.
     - (a) If any cohort replies with `ABORTED` or after a timeout, send `ABORT` to all cohorts (and write `ABORT` into the log).
     - (b) If a `PREPARED` message is received from all cohorts, start with Phase 2.

- At cohorts:
  1. Complete execution.
  2. Wait for `PREPARE` from coordinator.
  3. Enter all changes into the local `log`.
  4. If successful send `PREPARED` message to coordinator, otherwise send `ABORTED`. 
Two Phase Commit Protocol

Phase II:

• At coordinator:
  (1) Write COMMIT into log, and send a COMMIT message to each cohort.
  (2) Wait for ACK_COMMIT reply from each cohort (retransmit after a timeout).
  (3) Write a COMPLETE entry into log.

• At cohorts:
  (1) Wait for decision of coordinator.
  (2) If decision is:
    (a) ABORT, then undo the transaction, release all locks, and send ACK_ABORT to coordinator.
    (b) COMMIT, then release all locks and send ACK_COMMIT to coordinator.

Coordinator-Cohort Interactions

• Successful commit:
  
  ![Diagram of successful commit](image)

• Cohort aborts commit:
  
  ![Diagram of cohort abort](image)

• Coordinator aborts commit:
  
  ![Diagram of coordinator abort](image)