Transactions

Concept of transactions is strongly related to Mutual Exclusion:

- **Mutual exclusion**
  - Shared resources (data, servers, ...) are controlled in a way, that no more than one process is allowed to access it at a time

- **Transactions**
  - Protect shared data against simultaneous access by concurrent processes to ensure a consistent state
  - Avoid 'faulty accesses' caused e.g. by process failures
    - Atomic operations, i.e. the steps of a sequence in a transaction are finished all or none. After starting a transaction, all involved processes can perform operations (create, delete, modify). Sometime, the initiator asks them all to commit their work. If all do so, the changes became permanent. Otherwise (only one process refuses or crashes), the state before beginning with the operations is reloaded.

That means: A transaction is a sequence of operations that is guaranteed to be **atomic**, even in presence of multiple concurrent clients and process crashes.

Distributed Transactions

Simple World

- 1960s: all data were held on magnetic tape; simple transaction management
- Example *supermarket with automatic inventory system*:
  - One tape for yesterday's inventory, one for today's changes; the output is the new inventory
  - In case of a failure: rewind tapes and restart job

- Modern transaction concept originates from *database management systems* (1980)
- Transactions for distributed objects: late 1980s and 1990s

Problem today

Example banking application using a PC (an account is a remote object)

1. Withdraw amount \( x \) from account \( a \)
2. Deposit amount \( x \) to account \( b \)

But... When there is a connection loss after the first step, the money is lost. Thus: group the operations belonging together in a transaction.

**Key issue**: rolling back to the initial state

**Key operations**:

- deposit(amount)
- withdraw(amount)
- getBalance() → amount
- setBalance(amount)
- create(name) → account
- lookUp(name) → account
- branchTotal() → amount
**Transaction Primitives**

- Programming with transactions: special primitives are required
- Typical example:

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN_TRANSACTION</td>
<td>Start of a new 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>

**Transaction Example**

Simple example: booking a stop-over flight

Transaction to reserve three flights commits:

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

Transaction aborts when third flight is unavailable:

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

Now the bookings for the first two flights have to be undone.

**Characteristics of Transactions**

Each transaction has to fulfill the **ACID** properties:

- **A** – Atomic. From 'outside', all operations of a transaction seem to be a single unit. Complete transaction or none of it.
- **C** – Consistent. The transaction does not violate system invariants. When starting in a consistent state, the transaction has to end in a consistent state, even in the presence of concurrent transactions. The transaction leaves the system in a consistent state, even if other concurrent transactions also change the system.
- **I** – Isolated. Concurrent transactions do not interfere with each other. Intermediate effects of a transactions must not be visible to other transactions. The effects of a transaction are visible only after its completion.
- **D** – Durable. When a transaction is permitted, all changes are saved in a permanent storage. Changes of a transaction must be permanent. Changes in transaction must be stored to durable storage.

That means:

1. Either all of the operations must be completed successfully or they must have no effect at all even in the presence of server crashes (A, D)
2. Transactions are free from interference by operations performed on behalf of other concurrent clients (I, C)

**Classification of Transactions**

- **Flat transactions**
  - Simply a series of operations (in a single machine) that satisfy the ACID properties.
  - Simplest type of transactions, most often used.
  - Limitations:
    - partial results cannot be committed or aborted
    - what to do with transactions distributed across several machines?
  - Better approaches
    - **Nested transactions**
      - Transactions can fork into sub-transactions
    - **Distributed transactions**
      - Basing on flat or nested transactions, but dealing with transaction operations distributed over several machines
Nested Transactions

Nested transactions allow more concurrency:
- a transaction is divided into a number of sub-transactions
- Top-level transaction forks off children which run (independently) in parallel
- Hierarchies of sub-transactions are allowed
- Considerable amount of administration!

Sub-Transactions

- Logical division of the work of the top-level transaction
- When a (sub-)transaction starts, it gets a private copy of all data. This makes rollbacks simple.
- To a parent, a sub-transaction is atomic with respect to failures and concurrent access
- Transactions at the same level (e.g. \( T_1 \) and \( T_2 \)) can run concurrently but access to common objects is controlled
- A sub-transaction can fail independently of its parent and other sub-transactions, but its parent decides, what to do (give up, start another sub-transaction, ...)
- Only the top-level transaction has permanence; when it commits, all (successful) sub-transactions become permanent

Distributed Transactions

- Problem with flat and nested transactions: transactions cannot be distributed across multiple machines
- Solution: distributed transactions: flat or nested transactions that operate on distributed data
- Problem now: need for distributed algorithms to guarantee the ACID properties

Distributed Transactions

Distributed transactions normally have a coordinator which coordinates all transaction operations at the involved machines:
Coordination in Distributed Transactions

Transaction Management

Distributed Transaction Management

Most work: Concurrency Control
Concurreny Control

Problem for the scheduler: guarantee consistency and isolation

- Controlling the execution of concurrent transactions working on shared data at the same time
- Achieved by giving concurrent transactions access to data in a specific order. The final result must be the same as if all transactions would have been executed sequentially

If there would be no concurrency control, two problems with concurrent transactions would arise:

1. **Lost update** problem: two transactions both read the same old value of a variable and use it to calculate a new value
2. **Inconsistent retrievals** problem: a retrieval transaction observes values that are involved in an ongoing updating transaction

The Lost Update Problem

<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>
<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>$200</td>
</tr>
<tr>
<td></td>
<td>$220</td>
</tr>
<tr>
<td></td>
<td>$80</td>
</tr>
</tbody>
</table>

- The initial balances of accounts A, B, C are $100, $200, $300
- Both transaction increase B's balance by 10% - the final result should be $242.
- Finally, B only gets to $220 - T had worked with a wrong value, thus the update made by U gets lost

The Inconsistent Retrievals Problem

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

- A and B both initially have $200.
- V transfers $100 from A to B while W calculates branch total (which should be $400 for A and B together)
- It occurs an inconsistent retrieval because V has only done the withdraw part when W sums the balances of A and B

Serialisation

The sketched problems can be avoided:

- Two operations are in conflict, if their combined effect depends on the order they are executed
- The transactions have to be scheduled to avoid overlapping access to the accounts accessed by both of them
- Solution: serially equivalence. Two transactions are serially equivalent, if all pairs of conflicting operations are executed in the same order for all objects they access.
- We have to find a serially equivalent interleaving to achieve that the combined effect is the same as if the transactions had been done one at a time in some order
- The same effect means
  - The read operations return the same values
  - The instance variables of the objects have the same values at the end
- Serial equivalence is the basis for concurrency control protocols for transactions
Chapter 6: Transaction Processing

Serially Equivalent Interleaving of T and U (Lost Updates)

<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>

<table>
<thead>
<tr>
<th>balance = b.getBalance()</th>
<th>balance = b.getBalance()</th>
</tr>
</thead>
<tbody>
<tr>
<td>$200</td>
<td>$220</td>
</tr>
<tr>
<td>b.setBalance(balance*1.1)</td>
<td>b.setBalance(balance*1.1)</td>
</tr>
<tr>
<td>$220</td>
<td>$242</td>
</tr>
<tr>
<td>a.withdraw(balance/10)</td>
<td>c.withdraw(balance/10)</td>
</tr>
<tr>
<td>$80</td>
<td>$278</td>
</tr>
</tbody>
</table>

- If one of T and U runs before the other, they can’t get a lost update
- The same is true if they are run in a serially equivalent ordering

Serially Equivalent Interleaving of V and W (Inconsistent Retrievals)

<table>
<thead>
<tr>
<th>Transaction V:</th>
<th>Transaction W:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.withdraw(100); b.deposit(100)</td>
<td>aBranch.branchTotal()</td>
</tr>
<tr>
<td>a.withdraw(100); balance = b.getBalance()</td>
<td>b.deposit(100); total = a.getBalance()</td>
</tr>
<tr>
<td>balance = 100</td>
<td>total = 300</td>
</tr>
<tr>
<td>total = total+b.getBalance()</td>
<td>total = total+c.getBalance()</td>
</tr>
<tr>
<td>$220</td>
<td>$100</td>
</tr>
</tbody>
</table>

- If W is run before or after V, the problem will not occur
- Therefore it will not occur in a serially equivalent ordering of V and W
- The illustration is serial, but it need not to be

Serialisation

How to achieve a serially equivalent interleaving?

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

Three transactions working on the same data item.

Possible schedules would be:

<table>
<thead>
<tr>
<th>Schedule 1</th>
<th>Schedule 2</th>
<th>Schedule 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3</td>
<td>x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3</td>
<td>x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3</td>
</tr>
</tbody>
</table>

- Schedule 1 is legal
- Schedule 2 is not legal, but produces no illegal results
- Schedule 3 is illegal: conflicting write operations!

Serialisation

- Representation of transactions is a series of read and write operations
- Example: T: x = 0; x = x + 1
  \[ \Rightarrow \text{write}(T, x); \text{read}(T, x); \text{write}(T, x) \]
- Concurrency control schedules conflicting operations to serialise them
- Conflict: two operations operate on the same data item, at least one is a write operation
- For two transactions to be serially equivalent, it is necessary and sufficient that all pairs of conflicting operations of the two transactions are executed in the same order at all of the objects they both access
Non-serially Equivalent Interleaving of Operations

Consider:
- \( T: x = \text{read}(i); \text{write}(i, 10); \text{write}(j, 20); \)
- \( U: y = \text{read}(j); \text{write}(j, 30); z = \text{read}(i); \)

<table>
<thead>
<tr>
<th>Transaction ( T ):</th>
<th>Transaction ( U ):</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \text{read}(i) )</td>
<td>( y = \text{read}(j) )</td>
</tr>
<tr>
<td>( \text{write}(i, 10) )</td>
<td>( \text{write}(j, 30) )</td>
</tr>
<tr>
<td>( \text{write}(j, 20) )</td>
<td>( z = \text{read}(i) )</td>
</tr>
</tbody>
</table>

- Each transaction’s access to \( i \) and \( j \) is serialised w.r.t one another, but
  - \( T \) makes all accesses to \( i \) before \( U \) does
  - \( U \) makes all accesses to \( j \) before \( T \) does
- Therefore this interleaving is not serially equivalent

read and write Operation Conflict Rules

<table>
<thead>
<tr>
<th>Operations of different conflict transactions</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>read read</td>
<td>No</td>
</tr>
<tr>
<td>Because the effect of a pair of read operations does not depend on the order in which they are executed</td>
<td></td>
</tr>
<tr>
<td>read write</td>
<td>Yes</td>
</tr>
<tr>
<td>Because the effect of a read and a write operation depends on the order of their execution</td>
<td></td>
</tr>
<tr>
<td>write write</td>
<td>Yes</td>
</tr>
<tr>
<td>Because the effect of a pair of write operations depends on the order of their execution</td>
<td></td>
</tr>
</tbody>
</table>

- Conflicting operations
- A pair of operations conflicts if their combined effect depends on the order in which they were performed
  - e.g. read and write (whose effects are the result returned by read and the value set by write)

Concurrency Control Algorithms

Three categories of algorithms:
- Locking:
  - A transaction is granted the exclusive access by setting locks
  - Most used practice for concurrency control
- Timestamp:
  - Operations are ordered by using timestamps before they are carried out
  - No conflicts can happen
- Optimistic:
  - No control while executing operations
  - Synchronisation takes place at the end of a transaction
  - If conflicts had occurred, some transactions are forced to abort

Locking

Oldest and most widely used algorithm is locking:
- When a process needs to read/write on a data item, it requests the scheduler to grant it a lock for this data item
- After performing read/write, the scheduler is asked to release the lock
- The scheduler is responsible to make valid schedules from all requests; it must be provided with an algorithm that produces only serialised schedules
Transactions with Exclusive Locks

when T is about to use b, it is locked for T

<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(bal*1.1)</td>
<td>b.setBalance(bal*1.1)</td>
</tr>
<tr>
<td>a.withdraw(bal/10)</td>
<td>c.withdraw(bal/10)</td>
</tr>
</tbody>
</table>

Operations | Locks | Operations | Locks
---|---|---|---|
BEGIN_TRANSACTION | bal = b.getBalance() | lock B | BEGIN_TRANSACTION | bal = b.getBalance() | waits for T’s lock on B |
| | b.setBalance(bal*1.1) | lock A | | | |
| | a.withdraw(bal/10) | lock A | | | |
| END_TRANSACTION | unlock A,B | | | | |

when T commits, it unlocks B

Thus: the use of the lock on b effectively serialises access to b

Algorithm: Two-phase locking

Steps in two-phase locking (2PL):

1. When the scheduler receives a request op(T, x) from the transaction manager, it tests, if the operation conflicts with any other operation already granted a lock.
   - Yes: the operation is delayed
   - No: a lock is granted for data item x, the operation is passed on to the data manager
2. When the data manager acknowledges the execution of the operation, the belonging lock is released by the scheduler
3. When a lock for T is released, no more locks are granted for T, no matter for which data item T is requesting a lock

The algorithm is widely used, because by rule 3, serialisability is given

Locks hold by a Transaction

Growing phase: locks are requested by the transaction when needed

Shrinking phase: when the first lock is released, no more locks can be requested

Strict Two-Phase Locking

In many systems, locks are released only if the transaction commits or aborts. (delaying read and write operations)

Main advantage:
- If T₂ reads a data item formerly modified by T₁, it can be sure that the value is already committed and no rollback is made after its reading operation
- This avoids cascaded aborts
Types of Locks

- Concurrency control protocols are designed to deal with conflicts between operations in different transactions on the same object.
- Describe protocols in terms of read and write operations, which are assumed to be atomic.
- Read operations of different transactions do not conflict, therefore exclusive locks reduce concurrency more than necessary.
- The ‘many reader/single writer’ scheme allows several transactions to read an object or a single transaction to write it (but not both).
- It uses read locks (sometimes called shared locks) and write locks.

<table>
<thead>
<tr>
<th>For one object</th>
<th>Lock requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock already set</td>
<td>read  write</td>
</tr>
<tr>
<td>none</td>
<td>OK          OK</td>
</tr>
<tr>
<td>read</td>
<td>OK          wait</td>
</tr>
<tr>
<td>write</td>
<td>wait        wait</td>
</tr>
</tbody>
</table>

Implementation of Lock Manager

- The granting of locks will be implemented by a separate object in a server that is called lock manager.
- The lock manager holds a set of locks, for example in a hash table.
- Each lock is an instance of the class Lock and is associated with a particular object.
  - its variables refer to the object, the holder(s) of the lock and its type.
- The lock manager code uses wait (when an object is locked) and notify when the lock is released.
- The lock manager provides setLock and unLock operations for use by the server.

Lock Class

```java
public class Lock {
    private Object object; // the object being protected by the lock
    private Vector holders; // the TIDs of current holders
    private LockType lockType; // the current type

    public synchronized void acquire(TransID trans, LockType aLockType) {
        try {
            while(/*another transaction holds the lock in conflicting mode*/) {
                wait();
                catch (InterruptedException e) {/*...*/
            }
            if(holders.isEmpty()) { // no TIDs hold lock
                holders.addElement(trans);
                lockType = aLockType;
            } else if(/*another transaction holds the lock, share it*/) {
                if(/* this transaction not a holder*/ holders.addElement(trans);
                lockType = aLockType;
            } else if(/*this transaction is a holder but needs a more exclusive lock*/
                lockType.promote();
            }
        }
    }

    public synchronized void release(TransID trans) {
        holders.removeElement(trans); // remove this holder
        // set locktype to none
        notifyAll();
    }
}
```

Release lock; all waiting transactions are notified.
LockManager Class

```java
public class LockManager {
    private Hashtable theLocks;

    public void setLock(Object object, TransID trans, LockType lockType) {
        Lock foundLock;
        synchronized(this) {
            // find the lock associated with object
            // if there isn't one, create it and add to the hashtable
            if (foundLock == null) {
                foundLock = new Lock();
                foundLock.acquire(trans, lockType);
            } else {
                foundLock.acquire(trans, lockType);
            }
        }
    }

    public synchronized void unLock(TransID trans) {
        Enumeration e = theLocks.elements();
        while (e.hasMoreElements()) {
            Lock aLock = (Lock)(e.nextElement());
            if (aLock.getTrans().equals(trans)) {
                aLock.release(trans);
            }
        }
    }
}
```

Locking of Nested Transactions

- Each set of nested transactions with the same parent is a single entity that must not access partial effects of other subtransaction sets
  - Any lock obtained or inherited by a subtransaction is inherited by its parent upon provisional commit of the subtransaction
- Each transaction in a set of subtransactions must not access partial effects of others in the set
  - Parent transactions cannot run concurrently with their children.
  - If a parent transaction has a lock on an object, it retains the lock during the time that its child transaction is executing. That is, children transactions can obtain temporary access of their parent’s locks.
  - Subtransactions at the same level can run concurrently, but when they access the same objects, the locking scheme must serialise the access.

Two-phase Locking in Distributed Systems

Several ways for implementation in distributed systems:

- **Centralised 2PL**
  A single site (lock manager) is responsible for granting and releasing locks. Each transaction manager communicates with it to get its locks. If a lock is granted, the transaction manager communicates directly with the responsible data managers (notice: data can be replicated over several machines). After finishing all operations, the lock is given back to the lock manager.

- **Primary 2PL**
  Each data item is assigned a primary copy. The lock manager on that copy’s machine is responsible for granting and releasing locks. By this, centralised 2PL is enhanced by distributing the locking across multiple machines.

- **Distributed 2PL**
  Data may be replicated across multiple machines. The schedulers on each machine now are not only granting and releasing local locks, but also forward operations to the (local) data manager.
Deadlocks when using Locks

Notice: locking can lead to deadlocks: if two processes acquire the same two locks in opposite order, a deadlock may result.

<table>
<thead>
<tr>
<th>Operations</th>
<th>Locks</th>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.deposit(100);</td>
<td>write lock A</td>
<td>b.deposit(200);</td>
<td>write lock B</td>
</tr>
<tr>
<td>b.withdraw(100);</td>
<td>waits for U's lock on B</td>
<td>a.withdraw(200);</td>
<td>waits for T's lock on A</td>
</tr>
<tr>
<td>***</td>
<td></td>
<td>***</td>
<td></td>
</tr>
</tbody>
</table>

- When locks are used, each of T and U acquires a lock on one account and then gets blocked when it tries to access the account the other one has locked.
- We have a deadlock.
- The lock manager must be designed to deal with deadlocks.

Deadlocks

Definition of deadlock:
- deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock.
- a wait-for graph can be used to represent the waiting relationships between current transactions (nodes = transactions, edges = wait-for relationships between transactions)

More complex wait-for Graph

- T, U and V share a read lock on C and
- W holds write lock on B (which V is waiting for)
- T and W then request write locks on C and a deadlock occurs e.g. V is in two cycles - look on the left

Deadlock prevention

... is unrealistic:
- e.g. lock all of the objects used by a transaction when it starts
  - unnecessarily restricts access to shared resources.
  - it is sometimes impossible to predict at the start of a transaction which objects will be used.
- Deadlocks can also be prevented by requesting locks on objects in a predefined order
  - but this can result in premature locking and a reduction in concurrency
Deadlock detection

...by finding cycles in the wait-for graph
  − After detecting a deadlock, a transaction must be selected to be aborted to break the cycle
  − The software for deadlock detection can be part of the lock manager
  − The lock manager holds a representation of the wait-for graph so that it can check it for cycles from time to time
  − Edges are added to the graph and removed from the graph by the lock manager’s `setLock` and `unlock` operations
  − When a cycle is detected, choose a transaction to be aborted and then remove from the graph all the edges belonging to it
  − it is hard to choose a victim - e.g. choose the oldest transaction or the one in the most cycles

Timeouts on locks

Alternative for deadlock detection by analysing the wait-for graph:

- **Lock timeouts** can be used (are commonly used) to resolve deadlocks
  - Each lock is given a limited period in which it is invulnerable
  - After this time, a lock becomes vulnerable
  - Provided that no other transaction is competing for the locked object, the vulnerable lock is allowed to remain
  - But if any other transaction is waiting to access the object protected by a vulnerable lock, the lock is broken (that is, the object is unlocked) and the waiting transaction resumes
  - The transaction whose lock has been broken is normally aborted

- Problems with lock timeouts
  - Locks may be broken when there is no deadlock
  - If the system is overloaded, lock timeouts will happen more often and long transactions will be penalised
  - It is hard to select a suitable length for a timeout

Distributed Deadlocks

Distributed transactions lead to distributed deadlocks
  − In theory a global wait-for graph can be constructed from local ones
  − One server is the global deadlock detector and collects local wait-for graphs in certain intervals
  − A cycle in a global wait-for graph that is not in local ones is a distributed deadlock

But a centralised solution has poor availability, no fault tolerance, bad scalability, and causes high communication costs. Furthermore: Phantom Deadlocks

- A ‘deadlock’ that is detected, but is not really one
- Happens when there appears to be a cycle, but one of the transactions has released a lock, due to time lags in distributing graphs
- In the figure suppose U releases the object at X then waits for V at Y
  - and the global detector gets Y’s graph before X’s \( T \rightarrow U \rightarrow V \rightarrow T \)
No global graph is constructed, but each server knows about some of the edges
- Servers try to find cycles by sending probes which follow the edges of the graph through the distributed system
- Send a probe for an edge \( T_1 \rightarrow T_2 \) when \( T_2 \) is waiting for some time
- By forwarding probes along a waiting chain, cycles can be found
- Each coordinator records whether its transactions are active or waiting
  - the local lock manager tells coordinators if transactions start/stop waiting
  - when a transaction is aborted to break a deadlock, the coordinator tells the participants, locks are removed and edges taken from wait-for graphs

Edge-Chasing Algorithm

Three steps:
- **Initiation:**
  - When a server notes that \( T \) starts waiting for \( U \), where \( U \) is waiting at another server, it initiates detection by sending a probe containing the edge \( < T \rightarrow U > \) to the server where \( U \) is blocked.
  - If \( U \) is sharing a lock, probes are sent to all the holders of the lock.
- **Detection:**
  - Detection consists of receiving probes and deciding whether a deadlock has occurred and whether to forward the probes.
  - E.g. when a server receives probe \( < T \rightarrow U > \) it checks if \( U \) is waiting, e.g. \( U \rightarrow V \), if so it forwards \( < T \rightarrow U \rightarrow V > \) to server where \( V \) waits
  - When a server adds a new edge, it checks whether a cycle is created
- **Resolution:**
  - When a cycle is detected, a transaction in the cycle is aborted to break the deadlock.

Example of edge chasing starts with \( X \) sending \( < W \rightarrow U > \), then \( Y \) sends \( < W \rightarrow U \rightarrow V > \), then \( Z \) sends \( < W \rightarrow U \rightarrow V \rightarrow W > \)

Probe to detect a cycle with \( N \) transactions will require \( 2(N-1) \) messages
- Studies of databases show that the average deadlock involves 2 transactions
- the above algorithm detects deadlock provided that waiting transactions do not abort
- no process crashes, no lost messages
- to be realistic it would need to allow for the above failures
- refinements of the algorithm to avoid more than one transaction causing a detection to start and then more than one being aborted
(Pessimistic) Timestamp Ordering

Different approach to concurrency control:

- Assign a timestamp $ts(T)$ to transaction $T$ at the moment it starts.
- Using Lamport's timestamps: total order is given. In distributed systems, a timestamp consists of a pair $<\text{local timestamp}, \text{server-id}>$.
- $ts(T)$ is assigned with each operation done by $T$. This allows a total order on all read and write events.
- Each data item $x$ is assigned with two timestamps $ts_{RD}(x)$ and $ts_{WR}(x)$ for read/write operations. The read timestamp is set to $ts(T)$ of the transaction $T$ which most recently read $x$ (analogously: write timestamp).
- If two operations conflict, the data manager processes the one with the lower timestamp first.
- A transaction's request to write an object is valid only if that object was last read and written by earlier transactions. A transaction's request to read an object is valid only if that object was last written by an earlier transaction.

(Tentative operations: work on own copy, make them permanent when committing.)

Rules for read and write Operations

<table>
<thead>
<tr>
<th>Rule</th>
<th>$T_c$</th>
<th>$T_s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>write</td>
<td>read</td>
</tr>
<tr>
<td>2.</td>
<td>write</td>
<td>read</td>
</tr>
<tr>
<td>3.</td>
<td>read</td>
<td>write</td>
</tr>
</tbody>
</table>

Example:

- $T$ sends a read($T, x$) with $ts(T) < ts_{WR}(x)$: $T$ is aborted.
- $T$ sends a read($T, x$) with $ts(T) > ts_{WR}(x)$: read operation is performed, $ts_{RD}(x) = \max\{ts(T), ts_{RD}(x)\}$.
- $T$ sends a write($T, x$) with $ts(T) < ts_{RD}(x)$: $T$ is aborted.
- $T$ sends a write($T, x$) with $ts(T) > ts_{RD}(x)$: no younger operation has read the value; change it, set $ts_{WR}(x) = \max\{ts(T), ts_{WR}(x)\}$.

Pessimistic Timestamp Ordering

- Let $ts_{RD}(x) = ts_{WR}(x) = ts(T_1)$, where $T_1$ has ended some time before.
- $T_2$ and $T_3$ start concurrently with $ts(T_2) < ts(T_3)$.
- Look at $T_2$'s write (a - d) and read (e - h):

Write Operations and Timestamps

if ($T_c \geq$ maximum read timestamp on object $D$ \&\& $T_c >$ write timestamp on committed version of $D$) perform write operation on tentative version of $D$ with write timestamp $T_c$
else /* write is too late */ Abort transaction $T_c$

Key:
- **Committed**
- **Tentative**
- object produced by transaction $T_j$ (with write timestamp $T_j$)
Read Operations and Timestamps

```plaintext
if (Tc > write timestamp on committed version of D) {
    let D_{selected} be the version of D with the maximum write timestamp ≤ Tc
    if (D_{selected} is committed)
        perform read operation on the version D_{selected}
    else
        Wait until the transaction that made version D_{selected} commits or aborts
        then reapply the read rule
} else
    Abort transaction Tc
```

(a) T1 read
(b) T2 read
(c) T3 read
(d) T4 read

Key:
- Committed
- Tentative
- Object produced by transaction Ti (with write timestamp Ti)

- T1 < T2 < T3 < T4

Transaction Commits with Timestamp Ordering

- When a coordinator receives a commit request, it will always be able to carry it out because all operations have been checked for consistency with earlier transactions
  - Committed versions of an object must be created in timestamp order
  - The server may sometimes need to wait, but the client need not wait
  - To ensure recoverability, the server will save the ‘waiting to be committed versions’ in permanent storage

- The timestamp ordering algorithm is strict because
  - The read rule delays each read operation until previous transactions that had written the object have committed or aborted
  - Writing the committed versions in order ensures that the write operation is delayed until previous transactions that have written the object have committed or aborted

Remarks on Timestamp Ordering

Concurrency Control

The method avoids deadlocks, but e.g. a transaction seeing a larger timestamp aborts, whereas in locking it could wait or even proceed immediately (Pessimistic ordering)

- Modification known as ‘ignore obsolete write’ rule is an improvement
  - If a write is too late it can be ignored instead of aborting the transaction, because if it had arrived in time its effects would have been overwritten anyway.
  - However, if another transaction has read the object, the transaction with the late write fails due to the read timestamp on the item

- Multiversion timestamp ordering
  - Allows more concurrency by keeping multiple committed versions
  - Late read operations need not be aborted

Remarks on Timestamp Ordering

Optimistic Concurrency Control

Third approach to concurrency control:

- Just do your operations without taking care to any operations of other processes. Problems can be thought about later. (Optimistic control)
- This should work, because in practise, conflicts are seldom
- Control algorithm:
  - keep track on all data items used by a transaction
  - When committing, all other transactions are checked, if they had modified one of the used data items since the transaction had started.
  - If yes, the transaction is aborted.
- Deadlock free, no process has to wait for locks
- Disadvantage: a transaction can fail unnecessarily and forced to run again. In cases of heavy load, this decreases performance heavily
- No real comparison with other approaches are possible: has primarily focused on non-distributed systems, has seldom been implemented in commercial systems
Optimistic Concurrency Control

Each transaction has three phases:

1. **Working phase**
   - the transaction uses a tentative version of the objects it accesses (dirty reads can’t occur as we read from a committed version or a copy of it)
   - the coordinator records the readset and writerset of each transaction

2. **Validation phase**
   - at **END_TRANSACTION** the coordinator validates the transaction (looks for conflicts)
   - if the validation is successful the transaction can commit
   - if it fails, either the current transaction, or one it conflicts with is aborted

3. **Update phase**
   - If validated, the changes in its tentative versions are made permanent
   - read-only transactions can commit immediately after passing validation

Validation of Transactions

- Use the read-write conflict rules to ensure that a particular transaction is serially equivalent with respect to all other overlapping transactions
- Each transaction is given a transaction number (defines its position in time) when it starts validation (the number is kept if it commits)
- The rules ensure serialisability of transaction \( T_v \) (transaction being validated) with respect to transaction \( T_i \)

\[
\begin{array}{c|c|c}
\text{Transaction number} & \text{Read} & \text{Write} \\
\hline
\text{Rule} & \text{read} & \text{write} \\
\hline
1. & \text{must not read objects written by } T_v & \\
2. & T_v & \text{must not read objects written by } T_i \\
3. & T_v & \text{must not write objects written by } T_i \\
& & \text{and } T_v \text{ must not write objects written by } T_i \\
\end{array}
\]

Rule 3 is valid if only one process at once can be in the update phase

Rules 1 and 2 are more complicated: test for overlaps between \( T_v \) and \( T_i \)

Backward Validation of Transactions

- check \( T_v \) with preceding overlapping transactions

\[
\text{boolean valid = true;}
\text{for (int } T_i = \text{start}Tn + 1; T_i <= \text{finish}Tn; T_i++){
\text{if (read set of } T_v \text{ intersects write set of } T_i \text{ valid = false;}
\}
\]

- \( \text{start}Tn \) is the biggest transaction number assigned to some other committed transaction when \( T_v \) started its working phase
- \( \text{finish}Tn \) is biggest transaction number assigned to some other committed transaction when \( T_v \) started its validation phase
- In figure, \( \text{Start}Tn + 1 = T_2 \) and \( \text{finish}Tn = T_3 \). In backward validation, the read set of \( T_v \) must be compared with the write sets of \( T_2 \) and \( T_3 \).
- The only way to resolve a conflict is to abort \( T_v \)
Forward Validation

- Rule 1: the write set of $T_v$ is compared with the read sets of all overlapping active transactions ($active1$ and $active2$).
- Rule 2: $(read\ T_v\ vs\ write\ T_i)$ is automatically fulfilled because the active transactions do not write until $T_v$ has completed.

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

Comparison of Forward and Backward Validation

- Which transaction should be aborted?
  - Forward validation allows flexibility, whereas backward validation allows only one choice (the one being validated)
- In general, read sets are larger than write sets.
  - Backward validation
    - compares a possibly large read set against the old write sets
    - overhead of storing old write sets
  - Forward validation
    - checks a small write set against the read sets of active transactions
    - need to allow for new transactions starting during validation

Optimistic Control in Distributed Systems

Problem: different servers can order transactions in different ways

Parallel validation of transactions

Solution:
- Global validation: the orderings of all servers are checked to be serialisable
- Globally unique transaction numbers for validation

Comparison of Methods for Concurrency Control

- Pessimistic approaches (detect conflicts as they arise)
  - Timestamp ordering: serialisation order decided statically
  - Locking: serialisation order decided dynamically
  - Timestamp ordering is better for transactions where reads $>>$ writes,
  - Locking is better for transactions where writes $>>$ reads
  - Strategy for aborts
    - Timestamp ordering – immediate
    - Locking – waits but can get deadlock
- Optimistic method
  - All transactions proceed, but may need to abort at the end
  - Efficient operations when there are few conflicts, but aborts lead to repeating work