Distributed Systems (5DV147)

Transactions

Fall 2016
Motivation

Objects $a$, $b$, $c$

Transfer 100 from $a$ to $b$
Transfer 200 from $c$ to $b$

```plaintext
a.withdraw(100);

b.deposit(100);

c.withdraw(200);

b.deposit(200);
```

Something can go wrong in the middle

Transactions are indivisible units that either ...

- ... complete successfully (changes recorded on permanent storage)
- ... or have no effect at all
- These under crash-failures and when multiple transactions operate on same objects (require concurrency control)
Transactions
ACID Properties

Atomicity: “all or nothing”
Consistency: transactions take system from one consistent state to another consistent state
Isolation: transactions do not interfere with each other
Durability: committed results of transactions are permanent

- recoverable objects
Operations

\texttt{openTransaction()} \rightarrow \textit{trans};

starts a new transaction and delivers a unique TID \textit{trans}. This identifier will be used in the other operations in the transaction.

\texttt{closeTransaction(\textit{trans})} \rightarrow (\textit{commit, abort});

ends a transaction: a \textit{commit} return value indicates that the transaction has committed; an \textit{abort} return value indicates that it has aborted.

\texttt{abortTransaction(\textit{trans})};

aborts the transaction.
Types of transactions
Flat transactions

- We have seen those already: 
  \texttt{open-transaction()} \ldots \texttt{commit()}/\texttt{abort()}

- The entire transaction must commit or abort
Nested transactions

- Tree-structured
- Sub-transactions at one level may execute concurrently
  - Shared objects’ accesses are serialized
- Sub-transactions may provisionally commit or abort independently
  - Parent may decide whether to abort or not
  - Provisional commit is not a proper commit!
Rules for committing nested transactions

1. All children transactions need to complete before deciding on commit/abort the parent transaction
2. Sub-transactions independently provisionally commit or abort – abort is final
3. When parent aborts, all sub-transactions abort
4. When a sub-transaction aborts, parent decide what to do
5. If the top-level transaction commits, all sub-transactions that have provisionally committed may commit as well if none of their ancestors has aborted
Flat and nested distributed transactions

- Distributed transaction:
  - Transactions accessing objects managed by more than one server (processes)
  - All servers need to commit or abort a transaction

- Allows for even better performance
  - At the price of increased complexity

- One coordinator and multiple participants
Flat transactions

- Requests are made to more than one server
- Access to servers is sequential
- A transaction can only wait for one object that is locked at a time
Nested transactions

- Sub-transactions can be opened to any depth
- Sub-transactions at the same level can run concurrently
- If sub-transactions run on different servers, they can run in parallel
Example: Distributed flat transaction

$T = \text{openTransaction}$

- $a.\text{withdraw}(4);$
- $c.\text{deposit}(4);$
- $b.\text{withdraw}(3);$
- $d.\text{deposit}(3);$
- $\text{closeTransaction}$

Note: the coordinator is in one of the servers, e.g. BranchX

Figure adapted from Instructor’s Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – based on Figure 17.3
Concurrent transactions
Problems with concurrent transactions

- Transactions are carried out concurrently for higher performance
  - Otherwise, painfully slow
- Two common problems that appear if performance is not handled correctly
  - Lost update
  - Inconsistent retrieval
- Solution
  - Serial equivalence: manage conflicting operations and create schedules that ensure the consistency requirement
Lost update

\( T_1: A = \text{read}(x), \text{write}(x, A \times 10) \)

\( T_2: B = \text{read}(x), \text{write}(x, B \times 10) \)

If not properly isolated, we could get the following interleaving:

\[
\begin{align*}
&T_1 \text{ A} = \text{read}(x) \\
&T_2 \text{ B} = \text{read}(x) \\
&T_1 \text{ write}(x, A \times 10) \\
&T_2 \text{ write}(x, B \times 10)
\end{align*}
\]

original value of x

Executing \( T_1 \) and \( T_2 \) should have increased \( x \) by ten times twice, but we lost one of the updates

Two transactions read the old value of the variable an then use that value to calculate a new value
Inconsistent retrieval

\( T_1: \) withdraw(x, 10), deposit(y, 10)

\( T_2: \) sum all accounts

Improper interleaving:

\( (T_1) \) withdraw(x, 10)
\( (T_2) \) sum+=read(x)
\( (T_2) \) sum+=read(y)

Read concurrent with update transaction

\( (T_1) \) deposit(y, 10)
\( (T_2) \) sum+=read(x)
\( (T_2) \) sum+=read(y)

... The sum is incorrect, because it doesn’t account for the 10 that are ‘in transit’ – neither in x nor in y – the retrieval is inconsistent

\( (T_1) \) withdraw(x, 10)
\( (T_1) \) deposit(y, 10)
\( (T_2) \) sum+=read(x)
\( (T_2) \) sum+=read(y)

... A retrieval transaction runs concurrent with an update transaction
How to work around these problems?

Serial Equivalence

- Interleaved operations produce same effect as if transactions have been performed one at a time

\[(T_1)\ A=\text{read}(x)\]
\[(T_1)\ \text{write}(x, A\times 10)\]
\[(T_2)\ B=\text{read}(x)\]
\[(T_2)\ \text{write}(x, B\times 10)\]

- Does not mean to actually perform one transaction at a time, as this would lead to horrible performance
A better example

<table>
<thead>
<tr>
<th>Transaction</th>
<th>T:</th>
<th>Transaction</th>
<th>U:</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance = b.getBalance();</td>
<td></td>
<td>balance = b.getBalance();</td>
<td></td>
</tr>
<tr>
<td>b.setBalance(balance * 1.1);</td>
<td></td>
<td>b.setBalance(balance * 1.1);</td>
<td></td>
</tr>
<tr>
<td>a.withdraw(balance/10)</td>
<td></td>
<td>c.withdraw(balance/10)</td>
<td></td>
</tr>
<tr>
<td>$200</td>
<td></td>
<td>$200</td>
<td></td>
</tr>
<tr>
<td>balance = b.getBalance();</td>
<td>$200</td>
<td>b.setBalance(balance * 1.1);</td>
<td>$220</td>
</tr>
<tr>
<td>b.setBalance(balance * 1.1);</td>
<td></td>
<td>c.withdraw(balance/10)</td>
<td></td>
</tr>
<tr>
<td>a.withdraw(balance/10)</td>
<td>$80</td>
<td>c.withdraw(balance/10)</td>
<td>$280</td>
</tr>
</tbody>
</table>

Better interleaving

$balance = 100$
$balance = 200$
$c.balance = 300$

Figure adapted from Instructor’s Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – based on Figures 16.5 and 16.7
Conflicting operations

- Two operations are in conflict if the final result depends on the order of execution
  - Value set by a write
  - Result of a read

Read – Read  ≠ No conflict
Read – Write (or Write – Read)  ≠ Conflict!
Write – Write  ≠ Conflict!
Back to the example

<table>
<thead>
<tr>
<th>Transaction</th>
<th>T:</th>
<th>Transaction</th>
<th>U:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>balance = b.getBalance();</td>
<td>balance = b.getBalance();</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b.setBalance(balance * 1.1);</td>
<td>b.setBalance(balance * 1.1);</td>
<td></td>
</tr>
</tbody>
</table>

T: (1) B.Read, (2) B.Write

U: (3) B.Read, (4) B.Write

Conflicts: (1,4), (2,3)

Interleave: 1, 3, 4, 2

The problem is that the pairs of conflicting operations should be performed in the same order! e.g., [(1,4),(2,3)] or [(4,1), (3,2)]
Serializability

- For two transactions to be *serially equivalent*, it is necessary and sufficient that all pairs of conflicting operations of the two transactions be executed in the same order at all of the objects they both access.
- Produce consistent schedules.
Concurrent transactions

Concurrency control protocols

- Ensure serially equivalent interleavings
- Maximize concurrency
  - Locks (wait for access)
  - Optimistic concurrency control (check for conflicts at the end)
  - Timestamp ordering (check to delay or reject)
Some more things to consider...
Problems when aborting transactions

- Results from transactions that commit must be recorded
- Results from transactions that abort should be forgotten
- Transactions can be aborted for whatever reason
  - Nature of transaction
  - Conflicts with another transaction
  - Crash of a process or computer
- Two common problems associated with aborted transactions
  - Dirty reads
  - Premature writes
Dirty reads

- T1 reads a value that T2 wrote, then commits and later, T2 aborts
  - The value is “dirty”, since the update to it should not have happened
  - T1 has committed, so it cannot be undone

---

**Transaction T:**
- a.getBalance()
- a.setBalance(balance + 10)

**Transaction U:**
- a.getBalance()
- a.setBalance(balance + 20)

---

<table>
<thead>
<tr>
<th>Transaction</th>
<th>T</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.getBalance()</td>
<td></td>
<td>a.getBalance()</td>
</tr>
<tr>
<td>a.setBalance(balance + 10)</td>
<td></td>
<td>a.setBalance(balance + 20)</td>
</tr>
</tbody>
</table>

---

balance = a.getBalance()  
$100  
a.setBalance(balance + 10)  
$110

---

balance = a.getBalance()  
$110  
a.setBalance(balance + 20)  
$130

---

abort transaction  
commit transaction
Handling dirty reads

- **New rule:** let T1 wait until T2 commits/aborts!
  - But if T2 aborts, we must abort T1
    ...and so on: others may depend on T1
    ...cascading aborts

- **Better rule:**
  - Transactions are only allowed to read objects that *committed* transactions have written
  - Delay commits until after all transactions whose uncommitted state has been seen (delay reads for writes)
Premature writes

- Sometimes “Before images” are used when recovering from an aborted transaction

Let $x = 50$ initially

**T1**: $\text{write}(x, 10)$; **T2**: $\text{write}(x, 20)$

Let T1 execute before T2

What happens if either one aborts?

Order of commit/abort matters!

- T2 aborts, T1 commits ($x=10$)
- T2 commits, T1 aborts ($x=50$)
- T2 aborts, T1 aborts ($x=10$)
Handling premature writes

- Delay writes to objects until other, earlier, transactions that write to the same object have committed/aborted.
- Systems that avoid both dirty reads and premature writes are “strict”:
  - Delay read and writes
  - Highly desirable, enforce isolation
  - Tentative versions (local to each transaction)
Two-phase commit
Atomic commit

- Distributed transaction
  - Transactions dealing with objects managed by different servers
- All servers commit or all abort
  - ... at the same time
  - in spite of (crash) failures and asynchronous systems

Problem of ensuring atomicity relies on ensuring that all participants vote and reach the same decision

Two-phase commit
Two-phase commit protocol

Phase 1: Coordinator collects votes

“Abort”, any participant can abort its part of the transaction

“Prepared to commit”, save updates to permanent storage to survive crashes (May not change vote to “abort”)

Phase 2: Participants carry out the joint decision

Protocol can fail due to servers crashing or network partition

- Log actions into permanent storage
Algorithm

Phase 1 (voting)
1. Coordinator sends “canCommit?” to each participant
2. Participants answer “yes” or “no”
   • “Yes”: update saved to permanent storage
   • “No”: abort immediately

Phase 2 (completion)
3. Coordinator collects votes (including own)
   – No failures and all “yes”? Send “doCommit” to each participant, otherwise, send “doAbort”
4. Confirm commit via “haveCommitted”

Note: Participants are in “uncertain” state until they receive “doCommit” or “doAbort”, and may act accordingly (send “getDecision” message to coordinator)
Timeout actions

If coordinator fails:
- Participants are “uncertain”
  - Participants can request status (send “getDecision” message to coordinator)
  - If some have received an answer (or they can figure it out themselves), they can coordinate themselves
- If participant has not received “canCommit?” and waits too long, it may abort

If participant fails:
- No reply to “canCommit?” in time?
  - Coordinator can abort
  - Crash after “canCommit?”
  - Use permanent storage to get up to speed

---

Figure adapted from Instructor’s Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – based on Figure 17.6
Two-phase commit protocol for nested transactions

- Sub-transactions “provisional commit”
  - Nothing written to permanent storage
    - Ancestor could still abort!
  - If they crash, the replacement cannot commit
- Status information is passed upward in tree
  - List of provisionally committed sub-transactions eventually reach top level
- Hierarchical or flat voting phase
Hierarchic voting

- Responsibility to vote passed one level at a time through the tree

Two-phase commit
Flat voting

- The coordinator contacts participants directly
  - Sends: Transaction ID and the list of transactions that are reported as aborted
- Coordinators may manage more than one sub-transaction, and due to crashes, this information may be required
- Coordinators must check if managed sub-transactions have an aborted ancestor (from the aborted transactions list)
Summary

- Transactions - specify sequence of operations that are atomic in presence of server crashes
- ACID properties
- Types of transactions
  - Flat and nested transactions
  - Distributed—flat and nested- transactions
- Problems due to concurrency
  - Lost update
  - Inconsistent retrieval
Serial equivalence (Serializability)
- Conflicting operations – read-read, read-write, write-read

Aborted transactions
- Dirty reads, premature writes

Atomic commit

Two-phase commit
Linearizability (Correction)

- Last lecture I remember saying something stupid that hints that linearizability is a form of strict consistency
- It is a stricter consistency model compared to sequential consistency, but it is not strict consistency
Example

<table>
<thead>
<tr>
<th>Process P1</th>
<th>Process P2</th>
<th>Process P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 1;$</td>
<td>$y = 1;$</td>
<td>$z = 1;$</td>
</tr>
<tr>
<td>print (y, z);</td>
<td>print (x, z);</td>
<td>print (x, y);</td>
</tr>
</tbody>
</table>
Linearizability Example

- Four valid execution sequences for the processes of the previous slide. The vertical axis is time. Original values of x, y, and z are zero.

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
<th>Process 3</th>
<th>Process 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1;</td>
<td>y = 1;</td>
<td>z = 1;</td>
<td>x = 1;</td>
</tr>
<tr>
<td>print (y, z);</td>
<td>print (x, z);</td>
<td>print (x, z);</td>
<td>print (x, z);</td>
</tr>
<tr>
<td>y = 1;</td>
<td>print (x, z);</td>
<td>z = 1;</td>
<td>print (x, y);</td>
</tr>
<tr>
<td>print (x, z);</td>
<td>print (y, z);</td>
<td>z = 1;</td>
<td>print (x, z);</td>
</tr>
<tr>
<td>z = 1;</td>
<td>print (x, y);</td>
<td>z = 1;</td>
<td>print (x, z);</td>
</tr>
<tr>
<td>print (x, y);</td>
<td>print (x, y);</td>
<td>print (y, z);</td>
<td>print (x, z);</td>
</tr>
</tbody>
</table>

Prints: 001011
Signature: 001011
(a)

Prints: 101011
Signature: 101011
(b)

Prints: 010111
Signature: 110101
(c)

Prints: 111111
Signature: 111111
(d)
Client-centric Consistency Models

- Assume read operations by a single process P at two different local copies of the same data store
  - Four different consistency semantics

- Monotonic reads
  - Once read, subsequent reads on that data items return same or more recent values

- Monotonic writes
  - A write must be propagated to all replicas before a successive write by the same process
  - Resembles FIFO consistency (writes from same process are processed in same order)

- Read your writes: read(x) always returns write(x) by that process

- Writes follow reads: write(x) following read(x) will take place on same or more recent version of x
Eventual Consistency

- There are replica situations where updates (writes) are rare and where a fair amount of inconsistency can be tolerated.
  - DNS – names rarely changed, removed, or added and changes/additions/removals done by single authority
  - Web page update – pages typically have a single owner and are updated infrequently.
- If no updates occur for a while, all replicas should gradually become consistent.
- May be a problem with mobile user who access different replicas (which may be inconsistent with each other).
A mobile user may access different replicas of a distributed database at different times. This type of behavior implies the need for a view of consistency that provides guarantees for a single client regarding accesses to the data store.
Session Guarantees

• When client move around and connects to different replicas, strange things can happen
  ▪ Updates you just made are missing
  ▪ Database goes back in time
• Responsibility of “session manager”, not servers
• Two sets:
  ▪ Read-set: set of writes that are relevant to session reads
  ▪ Write-set: set of writes performed in session
• Update dependencies captured in read sets and write sets
• Four different client-central consistency models
  ▪ Monotonic reads
  ▪ Monotonic writes
  ▪ Read your writes
  ▪ Writes follow reads
A data store provides monotonic read consistency if when a process reads the value of a data item $x$, any successive read operations on $x$ by that process will always return the same value or a more recent value.

Example error: successive access to email have ‘disappearing messages’

a) A monotonic-read consistent data store
b) A data store that does not provide monotonic reads.
Monotonic Writes

A write operation by a process on a data item $x$ is completed before any successive write operation on $x$ by the same process. Implies a copy must be up to date before performing a write on it.

Example error: Library updated in wrong order.

a) A monotonic-write consistent data store.

b) A data store that does not provide monotonic-write consistency.
The effect of a write operation by a process on data item \( x \) will always be seen by a successive read operation on \( x \) by the same process.

Example error: deleted email messages re-appear.

- A data store that provides read-your-writes consistency.
- A data store that does not.
A write operation by a process on a data item $x$ following a previous read operation on $x$ by the same process is guaranteed to take place on the same or a more recent value of $x$ that was read.

Example error: Newsgroup displays responses to articles before original article has propagated there

- A writes-follow-reads consistent data store
- A data store that does not provide writes-follow-reads consistency
Readings

- http://courses.cs.vt.edu/~cs5204/fall00/distributedDBMS/srenu/3pc.html
Next Lecture

Concurrency Control