Software Fault Tolerance

Competitive Concurrency

Jörg Kienzle
Software Engineering Laboratory
School of Computer Science
McGill University
Overview

(Kienzle Chapter 2)

- Transactions
  - ACID properties
  - Concurrency Control
  - Recovery Support
- Flat Transactions
  - With Savepoints
- Chained Transactions
- Nested Transactions
- Split / Joint Transactions & other models
- Transactions and Exceptions
Competitive Concurrent Systems

• Processes (or threads) running in the system have been designed separately
  • Are not aware of each other
  • Do not synchronize explicitly with other processes
  • Do not communicate directly with other processes

• Each individual process
  • Does not want to be annoyed by other processes
  • Does not want to care about data consistency issues
  • Does not want to be affected by faults in other processes
ACID Properties

• A transaction groups together a set of operations on data objects, guaranteeing the ACID properties
  • Atomicity
  • Consistency
  • Isolation
  • Durability
A Transfer Operation

Bank 1

withdraw(20)

Account A

Bank 2

deposit(20)

Account B
When Things go Wrong (2)

Bank 1

Account A

withdraw(20)

To prevent loss of money

Undo Withdraw or compensate

deposit(20)

Bank 2

Account B

deposit(20)

“All Or Nothing” Atomicity
When Things go Wrong (3)

Durability of Data Updates

Bank 1

Account A

withdraw(20)

Bank 2

Account B

deposit(20)
When People Work Concurrently

Isolation
“No Interference”
Consistency

- A transaction produces consistent results only
- Consistency criteria for the *Transfer* operation:
  Sum of balance of the accounts remains unchanged
- Existing transactional systems provide A, I and D
- Consistency is indirectly ensured if the transactions perform consistent state changes
- The transaction support does not erroneously corrupt the state of the system
Flat Transactions

3 Operations: begin, commit and abort
Flat Transactions

Abort $\Rightarrow$ apply backward error recovery
Serializability

Atomicity + Isolation = Serializability

The results produced by a concurrent execution of a set of transactions must be equivalent to the results produced by executing the same set of transactions sequentially, in some arbitrary order.
Providing Isolation

• Concurrency control must be applied at the level of every transactional object, e.g. the accesses to a transactional object from different transactions must be monitored, making sure they do not conflict
• Prevent transactions from seeing intermediate, possibly inconsistent state
• Prevent the domino effect, also called cascading aborts
Conflicting Operations

• Strict concurrency control is used when no semantic knowledge of the operation is available
• Read / write locking

<table>
<thead>
<tr>
<th></th>
<th>Read(y)</th>
<th>Write(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(x)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Write(x)</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

• If we can’t tell the difference, we must assume that all operations conflict (e.g. are writers)
Semantic-based Concurrency Control

- Some operations do not conflict - they commute
- Example
  - Account with Deposit, Withdraw, Balance operations

<table>
<thead>
<tr>
<th></th>
<th>Deposit</th>
<th>Withdraw</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deposit</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Withdraw</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Balance</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Commutativity and Update Strategy

- Commutativity actually depends on the update strategy used when modifying transactional objects.
- Backward commutativity goes with in-place update.
  - $\text{Op}_1$ commutes with $\text{Op}_2$, iff executing $\text{Op}_1$ has the same effect (final object state and return value) as executing $\text{Op}_2$, then $\text{Op}_1$, and then undoing $\text{Op}_2$.
  - Not symmetric!
- Forward commutativity goes with deferred update.
  - $\text{Op}_1$ commutes with $\text{Op}_2$, iff the return values of executing $\text{Op}_1$ are the same as the return values of executing $\text{Op}_2$ followed by $\text{Op}_1$. 
Pessimistic / Conservative CC

• Before allowing a transaction to perform an operation on a transactional object, it has to get the permission to do so

• If there is a potential conflict with any other ongoing (uncommitted) transaction, access is denied

• Block / abort or notify the calling transaction
Pessimistic Example: Lock-based CC

• Before accessing a transactional object, the calling transaction must acquire the associated lock
  • If the (to be acquired) lock conflicts with any other lock held by other transactions in progress, the calling thread is blocked
  • Otherwise, the lock is granted, and the thread can execute the operation

• 2-phase locking [EGLT76] ensures serializability
  • A transaction is not allowed to acquire new locks, once a lock has been released
  ⇒ locks are gradually acquired during execution and released only upon transaction abort or commit
Deadlocks

• Pessimistic blocking concurrency control might lead to deadlocks because of circular dependencies
  • T1 has acquired A and waits for B
  • T2 has acquired B and waits for A
• Deadlock prevention: Transactional objects must always be acquired in the same order to avoid deadlocks
  • Not realistic
• Detected deadlocks can be broken by aborting one of the transactions
  • Maintaining wait-for graphs and (periodically) performing cycle detection
  • Time-out
void transfer(Account source, Account dest, int amount) {
    source.withdraw(amount);
    dest.deposit(amount);
}

Bank 1
Account A
1000

stop

Bank 2
Account B
1800

transfer(A,B,10)  transfer(B,A,20)
Breakable Deadlock

\[ T_1: \text{transfer}(A,B,10) \quad T_2: \text{transfer}(B,A,20) \]
Deadlock-free Timestamp Ordering [BG81]

- Associate a (logical) time-stamp with each transaction
  - This implicitly specifies the serialization order in case of conflict beforehand
- Process conflicting operations based on the time-stamps of the invoking transactions at the object level
  - If a transaction A attempts to execute an operation on a transactional object that conflicts with an operation executed by transaction B
    - If timestamp(A) > timestamp(B), block A
    - If timestamp(A) < timestamp(B), abort A
  - We can’t abort B because B has acquired the rights already
- Deadlock free (transactions never wait for “newer” ones)
Multi-version Locking

• Idea
  • We don’t want to block late read-only transactions
  • Keep a history of committed states together with the “time range” in which they were valid
• Update (write, or read/write) transactions work on the main copy and have to acquire locks as usual
  • Upon commit, a (logical) timestamp is assigned to the update transaction, and a new committed state is created, annotated with the timestamp
• Read-only transactions get a timestamp at creation time
  • Whenever they read values, they are directed to the committed state corresponding to their timestamp
Optimistic / Aggressive CC

- Transactions are allowed to perform conflicting operations on transactional objects
- Upon commit, validation ensures serializability
- Backward validation
  - Check that previously committed transactions have not invalidated the results of the current transaction
- Forward validation
  - Ensure that a committing transaction does not conflict with transactions still in progress
Backward Validation

• Remember for every operation of a transactional object
  • Last(op), the time-stamp indicating when the most recently committed transaction has called the operation
  • First(t,op), the first time transaction t has invoked the operation

• Validate transaction t iff
  \[ \forall \text{transactionalObject}(\forall \text{calledOp}(\forall \{\text{op'} \mid \text{conflict(calledOp,op')}\}: \text{Last(op')} < \text{First(t, calledOp)})) \]

• Assumption: deferred update
Forward Validation

• Different schemes
  • If there are conflicts with other active transactions, abort the committing one
    ⇒ might lead to wasted aborts

• Broadcast commit [MN82]
  • Abort transactions in progress that have executed conflicting operations

• Ideal scheme depends on the application
  • Semantics of operations, frequency of use of transactional objects, etc…
Providing Atomicity and Durability

- Atomicity and durability of transactions must be ensured at all times.
- Transaction abort
  - Undo the changes made on behalf of the transaction.
- At any time, a machine involved in the transaction might crash
  - Make sure that the changes of committed transactions have been stored in a safe place.
  - Transactions that were not committed at the time of crash are successfully aborted.
Recovery Support

• Store a transaction trace (or transaction history) in a log on stable storage
  • Transaction status
  • Accessed objects
  • Performed operations
  • Checkpoints

• Upon restart of the crashed node
  • Consult the log and perform necessary cleanup operations
Providing Consistency

• Consistency of data is application dependent
  • There’s no way the transaction support can provide consistency

• Idea:
  • Assume a consistent initial state
  • The programmer must write a transaction in such a way that it preserves consistency, i.e. moves the system from one consistent state to some other consistent state
  • Atomicity and isolation prevent inconsistencies from being visible from the outside
Other Transaction Models

• Problems with flat transactions
  • If an error is detected, flat transactions offer only two options
    • Perform manual forward error recovery (e.g. applying compensating actions, etc.)
    • Abort the transaction as a whole
  • For “long-running” transactions, giving up all results is undesirable
Flat Transactions with Savepoints (1)

Additional operations: save work and rollback work
Rollback to any savepoint is possible at any time
Flat Transactions with Savepoints (2)

- Additional operation Save_Work
  - Saves the state of all modified transactional objects
  - Hands back a handle to the application program
- Additional operation Rollback_Work(Handle)
  - Reestablish the state of the designated savepoint
- Begin_Transaction also establishes the first savepoint
- Aborting and rolling back to savepoint 1 are not the same operation!
Chained Transactions (1)

- In flat transactions, you still lose all work in case of a crash failure
- Chained transactions make a compromise between the flexibility of rollback and the amount of work lost after a crash
- Additional operation `Chain_Transaction`
  - Commit what has been done so far
  - Immediately start a new transaction with the same objects
    - Isolation property continues to hold
- Atomic “commit and begin”
Chained Transactions (2)

T₂ inherits access rights of T₁
Nested Transactions (1)

• Nested transactions [Mos81] provide functionality of transactions with savepoints, and allow recursive dynamic structuring of execution.

• Transactions form a tree hierarchy:
  • Top-level transactions
  • Child- or subtransactions / nested transactions
  • Leaf transactions are flat transactions
Nested Transactions (2)

• Starting nested transactions
  • Starting a new transaction from within a transaction creates a subtransaction

• Concurrency control
  • Accesses to transactional objects from inside a nested transaction are isolated with respect to the parent transaction, the sibling transactions, and other transactions
  • Access rights of the parent transaction can be “claimed” by the child
Nested Transactions (3)

- Ending nested transactions
  - A parent can only commit if all of its subtransactions have ended
  - The commit of a subtransaction makes its results visible only to the parent transaction (e.g. the parent transaction “inherits” the access rights of all transactional objects acquired by the child)
  - Aborting a nested transaction results in aborting all containing subtransactions

- Properties of non top-level transactions
  - Atomic with respect to the parent, consistency preserving with respect to the local function they implement, isolated from siblings and other external transactions, not durable
Nested Transactions (4)

Is call to getBalance here possible?
Concurrent Nested Transactions

• Up to this point, all models used a single thread to execute operations on transactional objects
• Sibling transactions are isolated from each other
  • If there is no semantic dependency, they can execute in parallel to enhance performance
• Additional threads are needed
• Transactions themselves are still sequential!
Nested Concurrent Transactions

Account A
withdraw

Account B

Thread

Begin T₁

Cobegin T₁.₁ and T₁.₂ (forks a thread)

Commit T₁.₁

Commit T₁.₂

Commit T₁
Split / Joint Transactions [PKH88]

• Split transactions
  • At any time, a transaction can split, forking a new thread and a new split-off transaction
  • At split-time, responsibility and access rights for already executed operations can be passed to the split-off transaction
  • Depending on conflicts, the split may be serial or independent

• Joint transactions
  • Instead of committing or aborting, a transaction can join another transaction, handing over all its operations and access rights
Splitting a Transaction (1)

1. Begin T1
2. op1
   - Object A
3. op3
4. Object C
5. op4
6. Split Transaction (T1, T2, B.op2, independent)
7. Thread forks and T2 starts
8. Object B
9. op2
10. Commit T1 (op1, op3, op4)
11. Commit T2 (op2, op5)
Splitting a Transaction (2)

Thread

Object A

Object B

Object C

Split Transaction
(T₁, T₂, B.op2, serial)

Abort T₁
(op₁, op₃, op₄)

Commit T₂ not possible!
T₂ must be aborted

Begin T₁

op₁

op₂

op₃

op₄

op₅
Joining a Transaction

Thread 1

Begin T₁

Thread 2

Begin T₂

Object A

op₁

Object B

op₂

Object C

op₃

Commit T₁
(op₁, op₂, op₃, op₄, op₅)

Join Transaction (T₁)
(hand over access rights)

Join Transaction (T₁)
(hand over access rights)
Other Models

• Recoverable Communicating Actions [VRS86]
  • Transactions can communicate results to other transactions
  • Sender aborts \(\Rightarrow\) Receiver aborts
  • Receiver can commit iff Sender commits

• SAGAS [GMS87]
  • A saga is a set of related transactions with a specified execution order
  • Every component transaction has an associated compensating transaction
  • ACID properties guaranteed at the transaction level
  • SAGAS execute entirely or not at all
Transactions and Software Fault Tolerance

- Transactions introduced 35 years ago
- Origin in databases [GR93]
  - Handle concurrent updates of data
  - Provide tolerance of hardware failures
- Software fault tolerance
  - Provide support for backward error recovery
  - Isolation property prevents the domino effect
  - Transaction boundaries coincide with consistent state
  - Powerful if combined with structured exception handling
Transactions and Exceptions

• A & I properties allow transactions to confine erroneous state
• Exceptions signal abnormal situations or potential erroneous state

Idea
• Make transactions exception handling contexts
• Add an acceptance test before transaction commit
• Write self-checking transactional objects [KRS01]
  • Treat unhandled exceptions crossing the transaction border as abort votes
References (1)

References (2)

References (3)