Distributed Algorithms

Distributed Transactions

PD Dr.-Ing. Gero Mühl

Kommunikations- und Betriebssysteme
Fakultät für Elektrotechnik u. Informatik
Technische Universität Berlin
Overview

> Motivation for Transactions

> Background
  > ACID-Properties
  > Locking-based Concurrency Control

> Distributed Transactions
  > Two-Phase Commit
Motivation for Transactions

> Critical sections (mutual exclusion)
  > used to achieve consistency in distributed systems
  > manually applied by the developer
  > complicated and error-prone (e.g., risk of deadlocks)

> Rather needed: high-level concept automatically ensuring consistency even in the face of failures
  → transactions
Transactions

> Important concept in databases and distributed systems
> Atomic execution of a set of instructions
  > e.g., bank transfer: debit source account and deposit destination account
> Completing a transaction
  > **Commit**: Transaction is successfully completed
  > Final state is stored *persistently* and then becomes visible outside of the transaction
  > **Abort**: Transaction is aborted
  > **Rollback** to initial state, i.e., the effects of the transaction are undone
## ACID-Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Atomicity</strong></td>
<td>All-or-Nothing</td>
</tr>
<tr>
<td><strong>Consistency</strong></td>
<td>Transition from one <em>consistent</em> state to another <em>consistent</em> state</td>
</tr>
<tr>
<td><strong>Isolation</strong></td>
<td>Intermediate states are not visible outside the transaction’s boundary</td>
</tr>
<tr>
<td><strong>Durability</strong></td>
<td>The final state is stored persistently and is not lost even in the case of a failure</td>
</tr>
</tbody>
</table>
Concurrency Control and Recovery

> The ACID-properties are endangered by *interfering concurrent transactions* and by *faulty environments*
Concurrency Control

> Supervision of simultaneously executing transactions
> Each TX is modeled as a sequence of read and write operations (called schedule) on individual data items followed by either commit or abort

> A schedule is
  > **serializable** iff it is equivalent to a serial schedule
  > **recoverable** iff any TX is only committed after all TXs from which TX has read data have been committed
  > **avoiding cascading aborts** iff no TX reads uncommitted data
  > **strict** iff no TX reads or overwrites uncommitted data
Concurrency Control

correct = RC \cap SR
Concurrency Control

> In practice, at least serializability and recoverability are used to ensure the ACID properties \( \rightarrow \) correct schedules

> Two main variants of concurrency approaches
  > pessimistic (locking)
  > optimistic (non-locking)

> Our focus: Locking
Scheduler

> How can correct schedules be enforced automatically?
> **Scheduler** reorders the operations issued by the TXs
> For each operation there are three possibilities
  > immediate execution,
  > delaying the execution, and
  > rejecting the execution
  (> respective TX is aborted)
> But how must a scheduler reorder the operations to enforce correct schedules?
Locking

> Similar to critical sections (mutual exclusion)
> However, locks are not granted by the programmer but automatically by the scheduler

> Decreases concurrency
  (TXs may have to wait until required locks are granted)
> Usually two types of locks are used to increase concurrency
  > **Read lock**: TX can read data item after lock was granted
  > **Write lock**: TX can read and write data item after lock was granted
Lock compatibility

<table>
<thead>
<tr>
<th></th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>+</td>
<td>–</td>
</tr>
<tr>
<td>Write</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

> After a read lock was granted, only further read locks can be granted but no write locks → shared lock
> After a write lock was granted, no further locks can be granted → exclusive lock
> A read lock can be upgraded to a write lock provided that no further read locks have been granted
> A write lock can be downgraded to a read lock if the TX has not yet written the data item. After a downgrade, further read locks can be granted
2PL (Two Phase Locking)

> After a TX has released a lock, it cannot request a new lock
  → a TX must hold all locks until it is sure it needs no further lock
  → reduces concurrency because locks may be hold longer
> At the end of the TX, all remaining locks are released
> Ensures serializability, but not recoverability, deadlock freeness, and avoiding cascading aborts
More stringent variants of 2PL

> **Strict 2PL**
  (all write-locks are hold until TX end)
  > refers only to the release of locks
  > ensures strict schedules (→ RC & ACA) in addition to serializability
  > can still deadlock (risk even higher!)
> **Conservative 2PL**
  (all locks are acquired at TX start)
  > refers only to the acquisition of locks
  > ensures deadlock freeness in addition to serializability
  > Combination of strict and conservative 2PL is not often used due to degraded concurrency!
  > Strict 2PL most used in practice!
Granularity of Data Items / Lock

> Granularity of data items and locks
  > depends on the application scenario and
  > varies from individual values to sets of files
> Determines concurrency and locking overhead (→ tradeoff)
  > *Fine-grained locking*
    > high concurrency but also high locking overhead
  > *Coarse-grained locking*
    > low locking overhead but also low concurrency
> *Lock escalation*
  > TXs starts locking items of fine granularity
  > If a TX acquires too many locks, the granularity of locks is increased
Distributed Transactions
Distributed Transactions

> Often local transactions are not sufficient
  > E.g., booking a journey requires *atomic* booking of a flight, a hotel, and a rental car at the airline, the hotel, and the car rental service

> Distributed transactions allow transactions to span multiple independent participants on different nodes
  > Commit and abort of distributed TXs have to be coordinated among the participants
Distributed Transactions

> More complicated than centralized transactions (e.g., in a database) due to the nature of distributed systems

> Arbitrary communication delays
> Distinct execution speeds
> Link failures (→ network partitions)
> Node failures (→ process crashes)
> Partial failures instead of total failures
Distributed 2PL

> Each data item is stored at exactly one node
> Each node has a scheduler managing its local data items
> The schedulers at all nodes, taken together, constitute a distributed scheduler
> Granting a lock on data item $x$ only depends on the locks currently active on $x$ $\rightarrow$ decision can be taken locally
> The schedulers must agree on the beginning of the shrinking phase, i.e., on the first release of a lock
> Commit or abort operation is sent to all nodes where the TX has accessed data items $\rightarrow$ atomic commit protocol (ACP) needed!
> If strict 2PL is used, there is not need for the schedulers to agree on the beginning of the shrinking phase; they simply release all locks of a TX when they receive the commit or the abort command
Two-Phase Commit (2PC)

- Prevalent protocol for distributed transactions
- Achieves only **atomicity** (other ACID properties neglected!)
- Participants go through **two phases** which are needed to allow **unilateral aborts** of participants

1. **Prepare (to commit)** (aka. Voting Phase)
   - Each participant votes to commit or to abort the TX
   - Once a participant has voted to commit, it can no longer abort the TX unilaterally

2. **Commit** (aka. Completion Phase)
   - Participants actually commit, after **consensus** has been reached that all participants are prepared. Otherwise all participants abort
Two-Phase Commit (contd.)

> Any *participant* can initiate to commit (*commit command*) or to abort the transaction (*abort command*)
  > In client/server systems, usually the client initiates commit
  > In messaging system, any participant can initiate commit
> **Coordinator** is used to achieve consensus
  > All participants must have registered at the coordinator
  > Coordinator requests all participants to vote
  > Decides to commit if *all* participants have voted to commit
  > Decides to abort if *any* participant has voted to abort
> Also cooperative decentralized implementations possible
2PC State Transitions

Cooperator

Initial

Wait

Commit

Abort

Commit Command/Abort

Commit Command/Prepare

∀P. Vote Commit/Commit

∃P. Vote Abort/Abort

Participant

Initial

Prepared

Commit

Abort

Commit Command/Abort

Prepare/Vote Commit

Commit/Vote Abort

Abort/Commit

Abort/Acknowledge

Abort/Acknowledge

Abort/Message send in turn
Centralized 2PC (successful completion)

Phase I

- Initial
- Initial
- Initial

prepare
vote commit

Phase II

- Prepared
- Prepared
- Prepared

commit
acknowledge
Centralized 2PC (unsuccessful completion)

Phase I

- Initial
- Initial
- Initial

Phase II

- Prepared
- Abort
- Prepared

Actions:
- vote commit
- vote abort
- commit command
- prepare
- abort
- acknowledge
Centralized 2PC (unsuccessful completion)

Phase I

Initial

Abort

abort command

abort

acknowledge
Linear 2PC (successful completion)

- Requires knowledge of next node
  - can be transmitted along with messages
- Fewer messages but no parallelism
Linear 2PC (unsuccessful completion)
Coping with Failures

> Timeouts are used to cope with failures such as lost messages; however, timeouts are difficult to choose

> Examples of timeout actions not requiring consultation of other parties
  > Coordinator aborts TX when it timeouts in *initial* or *wait* state
  > Coordinator retransmits commit/abort message to participants when it timeouts in *commit/abort* state
  > A participant aborts TX when it timeouts in *initial* state

> However, when a participant timeouts in the *prepared* state, consultation is required
Coping with Failures

> The period from the moment a participant has voted to commit and to the moment it knows the global decision is called **uncertainty period**

> An uncertain participant is blocked until it becomes certain
  > It cannot unilaterally abort because it cannot revoke its vote
  > It cannot unilaterally commit because the global decision may be to abort
  > It can try to contact other participants to find one which is certain (that either voted abort or that already knows the global decision)
  > If it can only contact uncertain participants, it is blocked (reason may be communication failure or failure of all other sites)
The Downsides of Distributed Transactions

> Not all resources may support distributed transactions
> Long-running transactions may block resources due to locking resulting in degraded throughput
> Distributed transactions introduce a large overhead due to necessary coordination
Bibliography


