Distributed Systems (COMP9243)

Lecture 7 (C): Synchronisation and Coordination
Part 2

Transaction:
- Comes from database world
- Defines a sequence of operations
- Atomic in presence of multiple clients and failures

Mutual Exclusion ++:
- Protect shared data against simultaneous access
- Allow multiple data items to be modified in single atomic action

Transaction Model:
- Operations:
  - BeginTransaction
  - EndTransaction
  - Read
  - Write
- End of Transaction:
  - Commit
  - Abort

Transaction Examples

Inventory:

Banking:

BeginTransaction
b = A.Balance();
A.Withdraw(b);
B.Deposit(b);
EndTransaction
ACID Properties

atomic: all-or-nothing. once committed the full transaction is performed, if aborted, there is no trace left;
consistent: the transaction does not violate system invariants (i.e. it does not produce inconsistent results);
isolated: transactions do not interfere with each other i.e. no intermediate state of a transaction is visible outside (also called serialisable);
durable: after a commit, results are permanent (even if server or hardware fails)

Classification of Transactions

Flat: sequence of operations that satisfies ACID
Nested: hierarchy of transactions
Distributed: (flat) transaction that is executed on distributed data

Flat Transactions:
✓ Simple
✗ Failure → all changes undone

BeginTransaction
accountA -= 100;
accountB += 50;
accountC += 25;
accountD += 25;
EndTransaction

Nested Transaction

Example:
Booking a flight
✓ Sydney → Manila
✓ Manila → Amsterdam
✗ Amsterdam → Toronto

What to do?
→ Abort whole transaction
→ Commit nonaborted parts of transaction only
→ Partially commit transaction and try alternative for aborted part

Subtransactions and parent transactions
→ Parent transaction aborts → all subtransactions abort
Slide 9

**Subtransactions:**
- Subtransaction can abort any time
- Subtransaction cannot commit until parent ready to commit
  - Subtransaction either aborts or commits provisionally
  - Provisionally committed subtransaction reports provisional commit list, containing all its provisionally committed subtransactions, to parent
- On commit, all subtransactions in that list are committed
- On abort, all subtransactions in that list are aborted.

Slide 10

**Transaction Atomicity Implementation**

Private Workspace:
- Perform all tentative operations on a shadow copy
- Atomically swap with main copy on commit
- Discard shadow on abort.

Slide 11

**Writeahead Log:**
- In-place update with writeahead logging
- Roll back on abort

```plaintext
x = 0;  
y = 0;  
BEGIN_TRANSACTION;
  x = x + 1;  
  y = y + 2;  
  x = y * y;  
END_TRANSACTION;
```

Slide 12

**Concurrency Control (Isolation)**

**Simultaneous Transactions:**
- Clients accessing bank accounts
- Travel agents booking flights
- Inventory system updated by cash registers

**Problems:**
- Simultaneous transactions may interfere
  - Lost update
  - Inconsistent retrieval
- Consistency and isolation require that there is no interference
  - Why?

**Concurrency Control Algorithms:**
- Guarantee that multiple transactions can be executed simultaneously while still being isolated.
- As though transactions executed one after another
CONFLICTS AND SERIALISABILITY

Read/Write Conflicts Revisited:

**conflict**: operations (from the same, or different transactions) that operate on same data

**read-write conflict**: one of the operations is a write

**write-write conflict**: more than one operation is a write

Schedule:

→ Total ordering (interleaving) of operations
→ Legal schedules provide results as though transactions serialised (serial equivalence)

Example Schedules:

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Time</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedule 1</td>
<td>x = 0, x = x + 1, x = 0, x = x + 2, x = 0, x = x + 3</td>
<td>Legal</td>
</tr>
<tr>
<td>Schedule 2</td>
<td>x = 0, x = x + 1, x = x + 2, x = 0, x = x + 3</td>
<td>Legal</td>
</tr>
<tr>
<td>Schedule 3</td>
<td>x = 0, x = x + 1, x = 0, x = x + 2, x = x + 3</td>
<td>Legal</td>
</tr>
</tbody>
</table>

SERIALISABLE EXECUTION

Serial Equivalence:

→ conflicting operations performed in same order on all data items
  - operation in $T_1$ before $T_2$, or
  - operation in $T_2$ before $T_1$

Are the following serially equivalent?

→ $R_1(x)W_1(x)R_2(y)W_2(y)R_3(x)W_1(y)$
→ $R_1(x)R_2(y)W_2(y)R_3(x)R_1(y)W_1(y)$
→ $R_1(x)R_2(x)W_1(x)W_2(y)R_3(y)W_1(y)$
→ $R_1(x)W_1(x)R_2(x)W_2(y)R_3(y)W_1(y)$

MANAGING CONCURRENCY

Dealing with Concurrency:

→ Locking
→ Timestamp Ordering
→ Optimistic Control
Transaction Managers:

**Slide 17**

**Transaction Managers:**
- **Transaction manager**
  - BEGIN TRANSACTION
  - END_TRANSACTION
- **Scheduler**
  - LOCK/RELEASE
  - Timestamp operations
- **Data manager**
  - Execute read/write

**Slide 18**

**LOCKING**
- Pessimistic approach: prevent illegal schedules
  - Lock must be obtained from scheduler before a read or write.
  - Scheduler grants and releases locks
  - Ensures that only valid schedules result

**Slide 19**

**TWO PHASE LOCKING (2PL)**
- Lock granted if no conflicting locks on that data item. Otherwise operation delayed until lock released.
- Lock is not released until operation executed by data manager.
- No more locks granted after a release has taken place.
- All schedules formed using 2PL are serialisable.

**Slide 20**

**PROBLEMS WITH LOCKING**
- **Deadlock:**
  - Detect and break deadlocks (in scheduler)
  - Timeout on locks
- **Cascaded Aborts:**
  - Release($T_i, x$) → Lock($T_j, x$) → Abort($T_i$)
  - $T_j$ will have to be aborted too
  - Problem: dirty read: seen value from non-committed transaction
  - Solution: **Strict Two-Phase Locking:**
    - Release all locks at Commit/Abort
**Timestamp Ordering**

- Each transaction has unique timestamp ($ts(T_i)$)
- Each operation ($TS(W), TS(R)$) receives its transaction’s timestamp
- Each data item has two timestamps:
  - read timestamp: $ts_{RD}(x)$ - transaction that most recently read $x$
  - write timestamp: $ts_{WR}(x)$ - committed transaction that most recently wrote $x$
- Also tentative write timestamps (noncommitted writes) $ts_{wr}(x)$
- Timestamp ordering rule:
  - write request only valid if $TS(W) > ts_{WR}$ and $TS(W) ≥ ts_{RD}$
  - read request only valid if $TS(R) > ts_{WR}$
- Conflict resolution:
  - Operation with lower timestamp executed first

**Optimistic Control**

Assume that no conflicts will occur.

- Detect conflicts at commit time
- Three phases:
  - Working (using shadow copies)
  - Validation
  - Update

**Validation:**

- Keep track of read set and write set during working phase
- During validation make sure conflicting operations with overlapping transactions are serialisable
  - Make sure $T_v$ doesn’t read items written by other $T_i$’s
  - Why?
  - Make sure $T_v$ doesn’t write items read by other $T_i$’s
  - Why?
  - Make sure $T_v$ doesn’t write items written by other $T_i$’s
  - Why?
- Prevent overlapping of validation phases (mutual exclusion)
Backward validation:
- Check committed overlapping transactions
- Only have to check if $T_v$ read something another $T_i$ has written
- Abort $T_v$ if conflict
  - $x$ Have to keep old write sets

Forward validation:
- Check not yet committed overlapping transactions
- Only have to check if $T_v$ wrote something another $T_i$ has read
- Options on conflict: abort $T_v$, abort $T_i$, wait
  - $x$ Read sets of not yet committed transactions may change during validation!

Distributed Transactions
- In distributed system, a single transaction will, in general, involve several servers:
  - transaction may require several services,
  - transaction involves files stored on different servers
- All servers must agree to Commit or Abort, and do this atomically.

Transaction Management:
- Centralised
- Distributed
Distributed Concurrency Control

Centralised 2PL:
- Single server handles all locks
- Scheduler only grants locks, transaction manager contacts data manager for operation.

Primary 2PL:
- Each data item is assigned a primary copy
- Scheduler on that server responsible for locks

Distributed 2PL:
- Data can be replicated
- Scheduler on each machine responsible for locking own data
- Read lock: contact any replica
- Write lock: contact all replicas

Distributed Locking

Distributed Timestamps:
Assigning unique timestamps:
- Timestamp assigned by first scheduler accessed
- Clocks have to be roughly synchronized

Distributed Optimistic Control:
- Validation operations distributed over servers
- Commitment deadlock (because of mutual exclusion of validation)
- Parallel validation protocol
- Make sure that transaction serialised correctly

Atomicity and Distributed Transactions

Distributed Transaction Organisation:
- Each distributed transaction has a coordinator, the server handling the initial BeginTransaction call
- Coordinator maintains a list of workers, i.e. other servers involved in the transaction
- Each worker needs to know coordinator
- Coordinator is responsible for ensuring that whole transaction is atomically committed or aborted
  - Require a distributed commit protocol.
**Distributed Atomic Commit**

- Transaction may only be able to commit when all workers are ready to commit (e.g., validation in optimistic concurrency).
- Hence, distributed commit requires at least two phases:
  1. **Voting phase**: all workers vote on commit. Coordinator then decides whether to commit or abort.
  2. **Completion phase**: all workers commit or abort according to decision.

Basic protocol is called **two-phase commit (2PC)**.

---

**Two-phase commit: Coordinator**:

1. Sends `CanCommit`, receives `yes`, `abort`;
2. Sends `DoCommit`, `DoAbort`

---

**Two-phase commit: Worker**:

1. Receives `CanCommit`, sends `yes`, `abort`;
2. Receives `DoCommit`, `DoAbort`

**What are the assumptions?**

**Limitations**:

- Once node voted "yes", cannot change its mind, even if crashes.
- Atomic state update to ensure "yes" vote is stable.
- If coordinator crashes, all workers may be blocked.
- Can use different protocols (e.g., three-phase commit).
- In some circumstances workers can obtain result from other workers.
Two-phase commit of nested transactions:

- Two-phase commit is required, as a worker might crash after provisional commit.
- On CanCommit request, worker:
  - votes “no”: if it has no recollection of subtransactions of committing transaction (i.e. must have crashed recently).
  - otherwise
    - aborts subtransactions of aborted transactions,
    - saves provisionally committed transactions in stable store,
    - votes “yes”.

Two Approaches:
- Hierarchic 2PC
- Flat 2PC

Coordinator:
- Some algorithms rely on a distinguished coordinator process
- Coordinator needs to be determined
- May also need to change coordinator at runtime

Election:
- Goal: when algorithm finished all processes agree who new coordinator is.
Determining a coordinator:

- Assume all nodes have unique id
- possible assumption: processes know all other process’s ids but don’t know if they are up or down
- Election: agree on which non-crashed process has largest id number

Requirements:

1. **Safety**: A process either doesn’t know the coordinator or it knows the id of the process with largest id number
2. **Liveness**: Eventually, a process crashes or knows the coordinator

### Bully Algorithm

- Three types of messages:
  - **Election**: announce election
  - **Answer**: response to election
  - **Coordinator**: announce elected coordinator
- A process begins an election when it notices through a timeout that the coordinator has failed or receives an Election message
- When starting an election, send Election to all higher-numbered processes
- If no Answer is received, the election starting process is the coordinator and sends a Coordinator message to all other processes
- If an Answer arrives, it waits a predetermined period of time for a Coordinator message
- If a process knows it is the highest numbered one, it can immediately answer with Coordinator

### Ring Algorithm

- Two types of messages:
  - **Election**: forward election data
  - **Coordinator**: announce elected coordinator
- Processes ordered in ring
- A process begins an election when it notices through a timeout that the coordinator has failed.
- Sends message to first neighbour that is up
- Every node adds own id to Election message and forwards along the ring
- Election finished when originator receives Election message again
- Forwards message on as Coordinator message

What are the assumptions?
What are the assumptions?

Sender performs a single `send()`
Group of receivers
Membership of group is transparent

**Examples**

Fault Tolerance:
- Replicated (redundant) servers
- Strong consistency: multicast operations

Service Discovery:
- Multicast request for service
- Reply from service provider

Performance:
- Replicated servers or data
- Weaker consistency: multicast operations or data

Event or Notification propagation:
- Group members are those interested in particular events
- Example: sensor data, stock updates, network status
Properties

Group membership:
- Static: membership does not change
- Dynamic: membership changes

Open vs Closed group:
- Closed group: only members can send
- Open group: anyone can send

Reliability:
- Communication failure vs process failure
- Guarantee of delivery:
  - all members (or none) – Atomic
  - all non-failed members

Ordering:
- Guarantee of ordered delivery
- FIFO, Causal, Total Order

Examples Revisited

Fault Tolerance:
- Reliability: Atomic
- Ordering: Total
- Membership: Static
- Group: Closed

Service Discovery:
- Reliability: No guarantee
- Ordering: None
- Membership: Static
- Group: Open

Performance:
- Reliability: Non-failed
- Ordering: FIFO, Causal
- Membership: Dynamic
- Group: Closed

Event or Notification propagation:
- Reliability: Non-failed
- Ordering: Causal
- Membership: Dynamic
- Group: Open

Other Issues

Performance:
- Bandwidth
- Delay

Efficiency:
- Avoid sending a message over a link multiple times (stress)
- Distribution tree
- Hardware support (e.g., Ethernet broadcast)

Network-level vs Application-level:
- Network routers understand multicast
- Applications (or middleware) send unicasts to group members
- Overlay distribution tree

Network-level Multicast

“You put packets in at one end, and the network conspires to deliver them to anyone who asks.” Dave Clark

Ethernet Broadcast:
- all hosts on local network
- MAC address: FF:FF:FF:FF:FF

IP Multicast:
- multicast group: class D Internet address:
  - first 4 bits: 1110 (224.0.0.0 to 239.255.255.255)
  - permanent groups: 224.0.0.1 - 224.0.0.255
- multicast routers
  - join group: Internet Group Management Protocol (IGMP)
  - set distribution trees: Protocol Independent Multicast (PIM)
**Application-level Multicast System Model**

Assumptions:
- reliable one-to-one channels
- no failures
- single closed group

**Basic Multicast**

- no reliability guarantees
- no ordering guarantees

**FIFO Multicast**

- order maintained per sender

```c
B-send(g, m) {
  foreach p in g {
    send(p, m);
  }
}

deliver(m) {
  B-deliver(m);
}
```
FO-init() {
    $S = 0$; // local sequence 
    for ($i = 1$ to $N$) $V[i] = 0$; // vector of last seen seq 
}

FO-send(g, m) {
    $S++$;
    B-send(g, <m,$S$>); // multicast to everyone
}

B-deliver(<m,$S$>) {
    if ($S == V[sender(m)] + 1$) {
        // expecting this msg, so deliver
        FO-deliver(m);
        $V[sender(m)] = S$;
    } else if ($S > V[sender(m)] + 1$) {
        // not expecting this msg, so put in queue for later
        enqueue(<m,$S$>);
    }

    // check if msgs in queue have become deliverable
    foreach <m,$S$> in queue {
        if ($S == V[sender(m)] + 1$) {
            FO-deliver(m);
            dequeue(<m,$S$>);
            $V[sender(m)] = S$;
        }
    }
}

CO-init() {
    // vector of what we've delivered already
    for ($i = 1$ to $N$) $V[i] = 0$;
}

CO-send(g, m) {
    $V[i]++$;
    B-send(g, <m,$V$>);
}

B-deliver(<m,$Vj$>) {
    // j = sender(m)
    enqueue(<m,$Vj$>);
    // make sure we've delivered everything the message
    // could depend on
    wait until $Vj[j] = V[j] + 1$ and $Vj[k] <= V[k]$ ($k != j$)
    CO-deliver(m);
    dequeue(<m,$Vj$>); $V[j]++$;
}
**Sequencer Based:**

- **Slide 61:**
  - **TOTALLY ORDERED MULTICAST**
  - Diagram showing sequence numbers and messages flow.

- **Slide 62:**
  - Sequencer-based ordering diagram with messages labeled 1 - message, 2 - sequence number.

**Agreement-based:**

- **Slide 63:**
  - Agreement-based ordering diagram with messages labeled 1 - message, 2 - proposed sequence, 3 - agreed sequence.

**Other possibilities:**

- Moving sequencer
- Logical clock based
  - Each receiver determines order independently
  - Delivery based on sender timestamp ordering
  - How do you know you have the most recent timestamp?
- Token based
- Physical clock ordering

**Hybrid Ordering:**

- FIFO + Total
- Causal + Total

**Dealing with Failure:**

- Communication
- Process

---

**Slide 64:**

- **HOMEWORK:**
  - Dealing with failure scenarios.
HOMEWORK

- We only discussed distributed transactions, but not replicated transactions. What changes if we introduce replication? Do the techniques we've discussed still work?
- How well does 2PC deal with failure? Can you improve it to deal with more types of failure?

Hacker's edition:
- Do the Multicast (Erlang) exercise

READING LIST

Optional

Slide 66
- Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey everything you always wanted to know...
- Elections in a distributed computing system Bully algorithm