1. The Basics - What’s a Transaction?

- The execution of a program that performs an administrative function by accessing a shared database, usually on behalf of an on-line user.

Examples

- Reserve an airline seat. Buy an airline ticket
- Withdraw money from an ATM.
- Verify a credit card sale.
- Order an item from an Internet retailer
- Place a bid at an on-line auction
- Submit a corporate purchase order
The “ities” are What Makes Transaction Processing (TP) Hard

- Reliability - system should rarely fail
- Availability - system must be up all the time
- Response time - within 1-2 seconds
- Throughput - thousands of transactions/second
- Scalability - start small, ramp up to Internet-scale
- Security – for confidentiality and high finance
- Configurability - for above requirements + low cost
- Atomicity - no partial results
- Durability - a transaction is a legal contract
- Distribution - of users and data
What Makes TP Important?

• It’s at the core of electronic commerce

• Most medium-to-large businesses use TP for their production systems. The business can’t operate without it.

• It’s a huge slice of the computer system market. One of the largest applications of computers.
TP System Architecture

- **Web Browser**
  - http
  - Queues
  - Request Controller
  - Transaction Server
  - other TP systems

- **Web Server**
  - http
  - Message Inputs
  - Requests
  - intranet
System Characteristics

• Typically < 100 transaction types per application

• Transaction size has high variance. Typically,
  – 0-30 disk accesses
  – 10K - 1M instructions executed
  – 2-20 messages

• A large-scale example: airline reservations
  – hundreds of thousands of active display devices
  – plus indirect access via Internet
  – tens of thousands of transactions per second, peak
Availability

- Fraction of time system is able to do useful work
- Some systems are very sensitive to downtime
  - airline reservation, stock exchange, telephone switching
  - downtime is front page news

<table>
<thead>
<tr>
<th>Downtime</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 hour/day</td>
<td>95.8%</td>
</tr>
<tr>
<td>1 hour/week</td>
<td>99.41%</td>
</tr>
<tr>
<td>1 hour/month</td>
<td>99.86%</td>
</tr>
<tr>
<td>1 hour/year</td>
<td>99.9886%</td>
</tr>
<tr>
<td>1 hour/20years</td>
<td>99.99942%</td>
</tr>
</tbody>
</table>

- Contributing factors
  - failures due to environment, system mgmt, h/w, s/w
  - recovery time
2. The ACID Properties

- Transactions have 4 main properties
  - Atomicity - all or nothing
  - Consistency - preserve database integrity
  - Isolation - execute as if they were run alone
  - Durability - results aren’t lost by a failure
Atomicity

• All-or-nothing, no partial results.
  – E.g. in a money transfer, debit one account, credit the other. Either debit and credit both run, or neither runs.
  – Successful completion is called *Commit*.
  – Transaction failure is called *Abort*.

• Commit and abort are irrevocable actions.

• An Abort *undoes* operations that already executed
  – For database operations, restore the data’s previous value from before the transaction
  – But some real world operations are not undoable. They require special treatment
  Examples - transfer money, print ticket, fire missile
Consistency

Every transaction should maintain DB consistency

- Referential integrity - E.g. each order references an existing customer number and existing part numbers
- The books balance (debits = credits, assets = liabilities)

➤ Consistency preservation is a property of a transaction, not of the TP system
( unlike the A, I, and D of ACID)

• If each transaction maintains consistency, then serial executions of transactions do too.
Isolation

• Intuitively, the effect of a set of transactions should be the same as if they ran independently.

• Formally, an interleaved execution of transactions is *serializable* if its effect is equivalent to a serial one.

• Implies a user view where the system runs each user’s transaction stand-alone.

• Of course, transactions in fact run with lots of concurrency, to use device parallelism.
Durability

• When a transaction commits, its results will survive failures (e.g. of the application, OS, DB system … even of the disk).

• Makes it possible for a transaction to be a legal contract.

• Implementation is usually via a log
  – DB system writes all transaction updates to its log
  – To commit, it adds a record “commit(T_i)” to the log
  – When commit(T_i) is on disk, T_i is committed.
  – System waits for disk ack before acking to user
3. Atomicity and Two-Phase Commit

• Distributed systems make atomicity harder
• Suppose a transaction updates data managed by two DB systems.
• One DB system could commit the transaction, but a failure could prevent the other system from committing.
• The solution is the two-phase commit protocol.
  – Abstract “DB system” by resource manager
  – Could be a SQL DBMS, message mgr, queue mgr, file system, OO DBMS, etc.
Two-Phase Commit

• Main idea - all resource managers (RMs) save a durable copy of the transaction’s updates before any of them commit.

• If one RM fails after another commits, the failed RM can still commit after it recovers.

• The protocol to commit transaction T
  – Phase 1 - T’s coordinator asks all participant RMs to “prepare T”. Each participant RM replies “prepared” after T’s updates are durable.
  – Phase 2 - After receiving “prepared” from all participant RMs, the coordinator tells all participant RMs to commit.
Two-Phase Commit System Architecture

1. Start transaction returns a unique transaction identifier.
2. Resource accesses include the transaction identifier. For each transaction, RM registers with TM.
3. When application asks TM to commit, the TM runs two-phase commit.

Application Program

Resource Manager

Transaction Manager (TM)

Other Transaction Managers

Read, Write

Start Commit, Abort
4. Replication Basics

- Replication - using multiple copies of a server or resource for better availability and performance.
  - Replica and Copy are synonyms

- If you’re not careful, replication can lead to
  - worse performance - updates must be applied to all replicas and synchronized
  - worse availability - some algorithms require multiple replicas to be operational for any of them to be used
Synchronous Replication

• Replicas function just like a non-replicated resource
  – Txn writes data item x. System writes all replicas of x.
  – Synchronous – replicas are written within the update txn
  – Asynchronous – One replica is updated immediately. Other replicas are updated later

• Problems with synchronous replication
  – Expensive due to 2-phase commit
  – Can’t control when updates are applied to replicas
Asynchronous Replication

- Asynchronous replication
  - Each transaction updates one replica.
  - Updates are propagated later to other replicas.

- Primary copy: Each data item has a primary copy
  - All transactions update the primary copy
  - Other copies are for queries and failure handling

- Multi-master: Transactions update different copies
  - Useful for disconnected operation, partitioned network, mobile
  - Useful when weak consistency is good enough

- Both approaches ensure that
  - Updates propagate to all replicas
  - If new updates stop, replicas converge to the same state

- We focus here on primary copy
5. Primary-Copy Replication

- Designate one replica as the primary copy (publisher)
- Transactions may update only the primary copy
- Updates to the primary are sent later to secondary replicas (subscribers) in the order they were applied to the primary

T1: Start
... Write(x1) ...
Commit

T2
...
Tn

Primary Copy

x1

x2

xm

Secondaries
Update Propagation

- Collect updates at the primary using triggers or by post-processing the log
  - Triggers: on every update at the primary, a trigger fires to store the update in the update propagation table.
  - Log post-processing: “sniff” the log to generate update propagations

- Log post-processing (vs. triggers)
  - Saves triggered update overhead during on-line txn.
  - But R/W log synchronization has a (small) cost
  - Requires admin (what if the log sniffer fails?)

- Optionally identify updated fields to compress log
- Most DB systems support this today.
Failure & Recovery Handling

• Secondary failure - nothing to do till it recovers
  – At recovery, apply the updates it missed while down
  – Needs to determine which updates it missed, just like non-replicated log-based recovery
  – If down for too long, may be faster to get a whole copy

• Primary failure
  – Normally, secondaries wait till the primary recovers
  – Can get higher availability by electing a new primary
    • Hold that thought ....
Communications Failures

- Secondaries can’t distinguish a primary failure from a communication failure that partitions the network.
- If the secondaries elect a new primary and the old primary is still running, there will be a reconciliation problem when they’re reunited. This is multi-master.
- To avoid this, one partition must know it’s the only one that can operate. It can’t communicate with other partitions to figure this out.
- Could make a static decision. E.g., the partition that has the primary wins.
- Dynamic solutions are based on Majority Consensus
Majority Consensus

• Whenever a set of communicating replicas detects a replica failure or recovery, they test if they have a majority (more than half) of the replicas.
• If so, they can elect a primary
• Only one set of replicas can have a majority.
• Doesn’t work with an even number of copies.
  – Useless with 2 copies
• Quorum consensus
  – Give a weight to each replica
  – The replica set that has a majority of the weight wins
  – E.g. 2 replicas, one has weight 1, the other weight 2
ELECTING A NEW PRIMARY

• A secondary S that detects primary’s failure starts a new election by sending invitations to all secondaries
  – Other secondaries reply with their replica identifier
  – If S gets replies from a majority, it selects the largest replica identifier as the winner
  – If not, it tells everyone to wait for more recoveries

• What if replicas fail and recover during the election?
  – Use Paxos. S includes a unique epoch number in its invitation.
  – A recipient accepts the invitation only if it hasn’t accepted another invitation with higher epoch
  – S backs off on retries, to avoid an arms race
After Electing a New Primary

- All replicas must now check that they have the same updates from the failed primary.
- During the election, each replica reports the id of the last log record it received from the primary.
- The most up-to-date replica sends its latest updates to (at least) the new primary.
- Could still lose an update that committed at the primary and wasn’t forwarded before the primary failed … but solving it requires synchronous replication (2-phase commit to propagate updates to replicas).
Conclusion

• Primary copy is just one replication
• Other models
  – Multi-master replication
  – Shared storage
• All 3 models are used commercially