1. Introduction

CSE 593 Transaction Processing

Philip A. Bernstein

Outline

1. The Basics
2. ACID Properties
3. Atomicity and Two-Phase Commit
4. Availability
5. Performance
6. Styles of System

1.1 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.
- Place an order using an on-line catalog on the Internet
- Fire a missile
- Download a video clip

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
- Configurability - for above requirements + low cost
- Atomicity - no partial results
- Durability - a transaction is a legal contract
- Distribution - of users and data

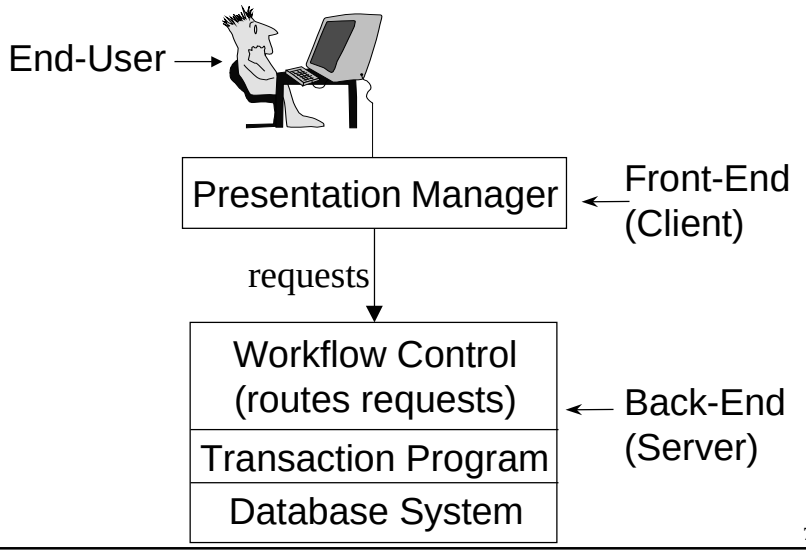
What Makes TP Important?

- 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 — over $50B/year. Probably the single largest application of computers.
- It's the glue that enables Internet-based commerce

TP System Infrastructure

- From the user's viewpoint, it
  - Gets a request from a display or other device
  - Performs some application-specific work, which includes database accesses
  - Usually, returns a reply
- It ensures each transaction is an independent unit of work that executes exactly once and produces permanent results.
- Makes it easy to write transactions

TP System Infrastructure …
Defines System and Application Structure



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
  - 150,000 active display devices
  - thousands of disk drives
  - 3000 transactions per second, peak

TP Monitors

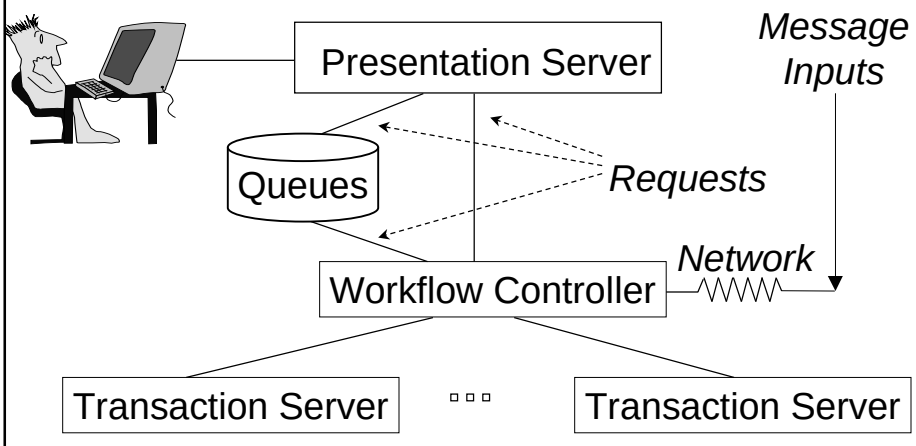
- A software product to create, execute and manage TP applications
- Takes an application written to process a single request and scales it up to a large, distributed system
  - E.g. application developer writes programs to debit a checking account and verify a credit card purchase.
  - TP monitor helps system engineer deploy it to 10s/100s of servers and 10Ks of displays

TP Monitors (cont'd)

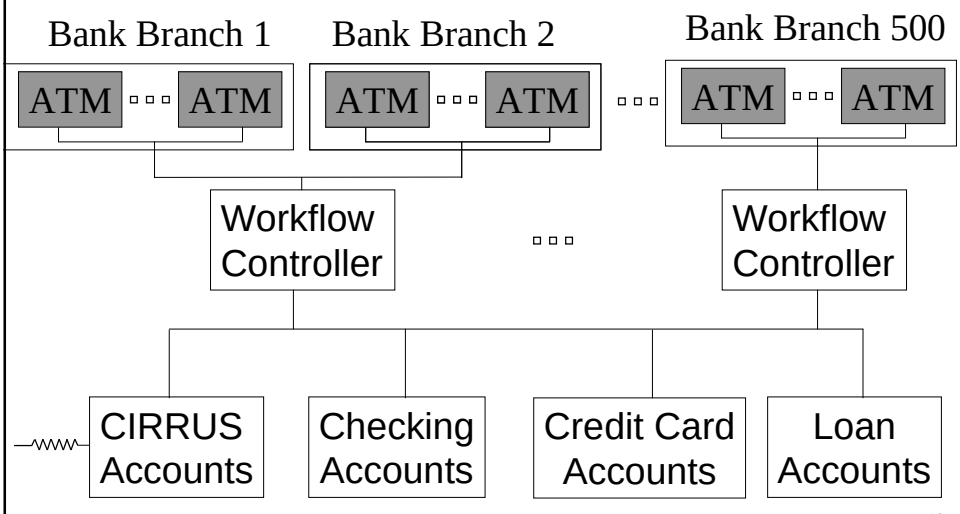
- Includes an application programming interface (API), and tools for program development and system management

TP Monitor Architecture

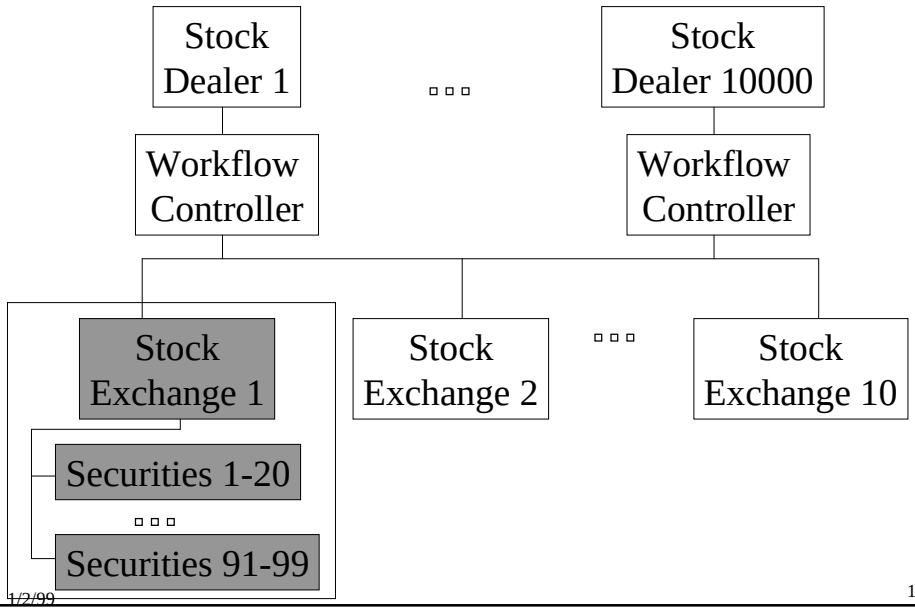
- Boxes below can be distributed around a network



Automated Teller Machine (ATM) Application Example



Automated Stock Exchange



1/2/99 13

Outline

✓1. The Basics
2. ACID Properties
3. Atomicity and Two-Phase Commit
4. Availability
5. Performance
6. Styles of System

1/2/99 14

1.2 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. Examples - transfer money, print ticket, fire missile

Example - ATM Dispenses Money a non-undoable operation

```
T1: Start
    . . .
      Dispense Money
    Commit
```

System crashes
Transaction aborts
Money is dispensed

```
T1: Start
    . . .
    Commit
Dispense Money
```

System crashes

Deferred operation never gets executed

Reading Uncommitted Output Isn't Undoable

```
T1: Start
    . . .
    Display output
    . . .
    If error, Abort
```

User reads output
…
User enters input

Brain transport

```
T2: Start
    Get input from display
    . . .
    Commit
```

Compensating Transactions

- A transaction that reverses the effect of another transaction (that committed). For example,
  - "Adjustment" in a financial system
  - Annul a marriage
- Not all transactions have complete compensations
  - E.g. Certain money transfers (cf. "The Firm")
  - E.g. Fire missile, cancel contract
  - Contract law has a lot to say about appropriate compensations
- A well-designed TP application should have a compensation for every transaction type

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.

Some Notation

- $r_i[x]$ = Read(x) by transaction T_i
- $w_i[x]$ = Write(x) by transaction T_i
- c_i = Commit by transaction T_i
- a_i = Abort by transaction T_i
- A *history* is a sequence of such operations, in the order that the database system processed them.

Consistency Preservation Example

T_1: Start;	T_2: Start;
A = Read(x);	B = Read(x);
A = A - 1;	C = Read(y);
Write(y, A);	If (B > C+1) then B = B - 1;
Commit;	Write(x, B);
	Commit;

- Consistency predicate is x > y.
- Serial executions preserve consistency. Interleaved executions may not.
- H = $r_1[x]$ $r_2[x]$ $r_2[y]$ $w_2[x]$ $w_1[y]$
  - e.g. try it with x=4 and y=2 initially

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.

A Serializability Example

T_1: Start;	T_2: Start;
A = Read(x);	B = Read(x);
A = A + 1;	B = B + 1;
Write(x, A);	Write(y, B);
Commit;	Commit;

- H = $r_1[x]$ $r_2[x]$ $w_1[x]$ c_1 $w_2[y]$ c_2
- H is equivalent to executing T_2 followed by T_1
- Note, H is *not* equivalent to T_1 followed by T_2
- Also, note that T_1 started and finished before T_2, yet the effect is that T_2 ran first.

Serializability Examples (cont'd)

- Client must control the relative order of transactions, using handshakes
  (wait for $T_1$to commit before submitting T_2).
- Some more serializable executions:
  $r_1[x]\ r_2[y]\ w_2[y]\ w_1[x] \equiv T_1\ T_2 \equiv T_2\ T_1$

  $r_1[y]\ r_2[y]\ w_2[y]\ w_1[x] \equiv T_1\ T_2 \not\equiv T_2\ T_1$

  $r_1[x]\ r_2[y]\ w_2[y]\ w_1[y] \equiv T_2\ T_1 \not\equiv T_1\ T_2$
- Serializability says the execution is equivalent to *some* serial order, not necessarily to *all* serial orders

Non-Serializable Examples

- $r_1[x]\ r_2[x]\ w_2[x]\ w_1[x]$ (*race condition*)
  - e.g. T_1 and T_2 are each adding 100 to x
- $r_1[x]\ r_2[y]\ w_2[x]\ w_1[y]$
  - e.g. each transaction is trying to make x = y, but the interleaved effect is a swap
- $r_1[x]\ r_1[y]\ w_1[x]\ r_2[x]\ r_2[y]\ c_2\ w_1[y]\ c_1$ (*inconsistent retrieval*)
  - e.g. T_1 is moving $100 from x to y.
  - T_2 sees only half of the result of T_1
- Compare to OS view of synchronization

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 the commit record is on disk, the transaction is committed.
  - system waits for disk ack before acking to user

Outline

✓1. The Basics
✓2. ACID Properties
3. Atomicity and Two-Phase Commit
4. Availability
5. Performance
6. Styles of System

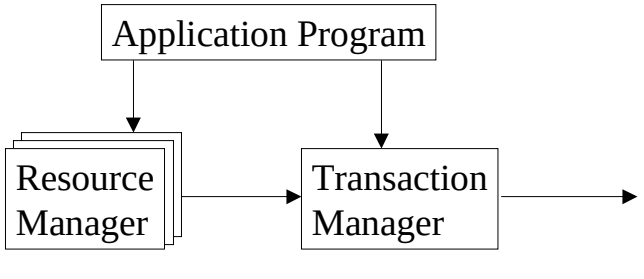
1.3 Atomicity and Two-Phase Commit

- 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 message mgr, queue mgr, 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 resource manager fails after another commits, the failed system can still commit after it recovers.
- The protocol to commit transaction T
  - Phase 1 - T’s coordinator asks all participant RMs to “prepare the transaction”. Participant RMs replies “prepared” after the 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 (txn) returns a unique *transaction identifier*
2. Resource accesses include the transaction identifier. For each txn, resource manager registers with txn manager
3. When application asks transaction manager to commit, the transaction manager runs two-phase commit.

Outline

✓1. The Basics
✓2. ACID Properties
✓3. Atomicity and Two-Phase Commit
4. Availability
5. Performance
6. Styles of System

1.4 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	Availability
1 hour/day	95.8%
1 hour/week	99.41%
1 hour/month	99.86%
1 hour/year	99.9886%
1 hour/20years	99.99942%

- Contributing factors
  - failures due to environment, system mgmt, h/w, s/w
  - recovery time

1.5 Performance Requirements

- Measured in max transaction per second (tps) or per minute (tpm), and dollars per tps or tpm.
- TP Performance Council (*TPC*) sets standards
  - http://www.tpc.org.
- Standards TPC A & B (‘89-’95), now TPC-C
- TPC A & B models a bank teller system.
  - Obsolete (a retired standard), but interesting
  - TPC-A includes terminals and comm. TPC-B is DB only
  - input is 100 byte message requesting deposit/withdrawal
  - Database tables = {Accounts, Tellers, Branches, History}

TPC-A/B Transaction Program

```
Start
  Read message from terminal (100 bytes)
  Read+write account record (random access)
  Write history record (sequential access)
  Read+write teller record (random access)
  Read+write branch record (random access)
  Write message to terminal (200 bytes)
Commit
```

- Database (DB) has 100K accounts / tps
  - ➔ Accounts table must be indexed
- End of history and branch records are bottlenecks

TPC-A/B numbers

- TPC-A/B had a *huge* effect on products.
- Drove down transaction resource consumption to
  - less than 100K instructions
  - 2.1 forced disk I/Os per transaction
  - 2 display I/Os per transaction
- Dollars measured by list purchase price plus 5 year vendor maintenance (“cost of ownership”)
- Workload has this profile:
  - 10% TP monitor plus application
  - 30% communications system (not counting presentation)
  - 50% DB system

The TPC-C Order-Entry Benchmark

Table	Rows/Whse	Bytes/row
Warehouse	1	89
District	10	95
Customer	30K	655
History	30K	46
Order	30K	24
New-Order	9K	8
OrderLine	300K	54
Stock	100K	306
Item	100K	82

- TPC-C uses heavier weight transactions

TPC-C Transactions

- New-Order
  - Get records describing given warehse, customer, district
  - Update the district
  - Increment next available order number
  - Insert record in Order and New-Order tables
  - For 5-15 items, get Item record, get/update Stock record
  - Insert Order-Line Record
- Payment, Order-Status, Delivery, Stock-Level have similar complexity, with different frequencies
- tpmC = number of New-Order transaction per min.

Comments on TPC-C

- Enables apples-to-apples comparison of TP systems
- Does not predict how *your* application will run, or how much hardware you will need, or even which system will work best on your workload
- Not all vendors optimize for TPC-C. E.g., IBM has claimed DB2 is optimized for a different workload, so they have only recently published TPC numbers
- UI is typically HTML, for Microsoft at least.

Typical TPC-C Numbers

- $33 - $200 / tpmC. Uniform spread across the range.
  - All $33-50 results on MS SQL Server.
  - Some $50-60 on Sybase. No one else is under $85.
- System cost $153K (Intergraph) - $7M (Sun)
- Throughput 1 - 50K tpmC
  - Sun 52K tpmC, $7M, $135/tpmC (Oracle 8, Tuxedo 4.2)
- Typical low-end - Compaq ProLiant
  - 3.9K tpmC, $206K, $53/tpmC
  - 10.5K tpmC, $351K, $34/tpmc
  - SQL Server 6.5 and BEA Systems Tuxedo
- Results are very sensitive to date published.

Outline

✓1. The Basics
✓2. ACID Properties
✓3. Atomicity and Two-Phase Commit
✓4. Availability
✓5. Performance
6. Styles of System

1.6 TP is a Style of System

- Batch processing - You *submit* a job, and receive output as a file.
- Time sharing - You *invoke programs* in a process, which may interact with the process's display
- Real time - You *submit requests* (very short jobs) which must be processed by a deadline
- Client/server - PC client *calls* a server over a network to do work (access files, run applications)
- Decision support - You *submit queries* to a shared database, and process the result with desktop tools
- TP - You *submit a request* to run a transaction

TP vs. Batch Processing (BP)

- A BP application is usually uniprogrammed (one transaction at a time), so serializability is trivial. TP is multiprogrammed with much concurrency.
- BP performance is measured by throughput. TP is also measured by response time.
- BP can optimize by sorting transactions by the file key, so it can merge the transactions with the file. TP must handle random transaction arrivals.
- BP produces new output file. To recover from a failure, just re-run the application.
- BP has fixed and predictable load, unlike TP.

TP vs. Batch Processing (cont'd)

- But, where there is TP, there is almost always BP too.
  - TP gathers the input
  - BP post-processes work that has weak response time requirements
- So, TP systems must also do BP well.

TP vs. Timesharing (TS)

- TS is a utility with highly unpredictable load. Different programs run each day, exercising features in new combinations.
- By comparison, TP is highly regular.
- TS has less stringent availability and atomicity requirements. Downtime isn't as expensive.

TP vs. Real Time (RT)

- RT has more stringent response time requirements. It may control a physical process.
- RT deals with more specialized devices.
- RT doesn't need or use a transaction abstraction
  - usually loose about atomicity and serializability
- RT has less stringent correctness guarantees. It's usually tolerable to lose some requests, to help meet response time goals.

TP and Client/Server (C/S)

- Is commonly used for TP, where client prepares requests and server runs transactions
- In a sense, TP systems were the first C/S systems, where the client was a terminal

TP and Decision Support Systems (DSSs)

- A.k.a. data warehouse (DSS is the more generic term, i.e. DW is a kind of DSS.)
- TP systems provide the raw data for DSSs
- TP runs short updates with high data integrity requirements.
- DSSs run long queries, usually with lower data integrity requirements

Outline

- ✓1. The Basics
- ✓2. ACID Properties
- ✓3. Atomicity and Two-Phase Commit
- ✓4. Availability
- ✓5. Performance
- ✓ 6. Styles of System

What's Next?

- This section covered TP system structure and properties of transactions and TP systems
- The rest of the course drills deeply into each of these areas, one by one.