Distributed Databases

Dr. Julian Bunn
Center for Advanced Computing Research
Caltech

Based on material provided by:
Jim Gray (Microsoft), Heinz Stockinger (CERN), Raghu Ramakrishnan (Wisconsin)

Outline

- ✍ Introduction to Database Systems
- ✍ Distributed Databases
- ✍ Distributed Systems
- ✍ Distributed Databases for Physics

Part I

Introduction to Database Systems

Julian Bunn

California Institute of Technology

What is a Database?

- A large, integrated collection of data
- Entities (things) and Relationships (connections)
- Objects and Associations/References
- A Database Management System (DBMS) is a software package designed to store and manage Databases
- “Traditional” (ER) Databases and “Object” Databases

Why Use a DBMS?

- Data Independence
- Efficient Access
- Reduced Application Development Time
- Data Integrity
- Data Security
- Data Analysis Tools
- Uniform Data Administration
- Concurrent Access
- Automatic Parallelism
- Recovery from crashes

Cutting Edge Databases

- Scientific Applications
- Digital Libraries, Interactive Video, Human Genome project, Particle Physics Experiments, National Digital Observatories, Earth Images
- Commercial Web Systems
- Data Mining / Data Warehouse
- Simple data but very high transaction rate and enormous volume (e.g. click through)

Data Models

- Data Model: A Collection of Concepts for Describing Data
- Schema: A Set of Descriptions of a Particular Collection of Data, in the context of the Data Model
- Relational Model:
  - E.g. A Lecture ***is attended by*** zero or more Students
- Object Model:
  - E.g. A Database Lecture ***inherits attributes*** from a general Lecture

Data Independence

- Applications insulated from how data in the Database is structured and stored
  - Logical Data Independence: Protection from changes in the logical structure of the data
  - Physical Data Independence: Protection from changes in the physical structure of the data

Concurrency Control

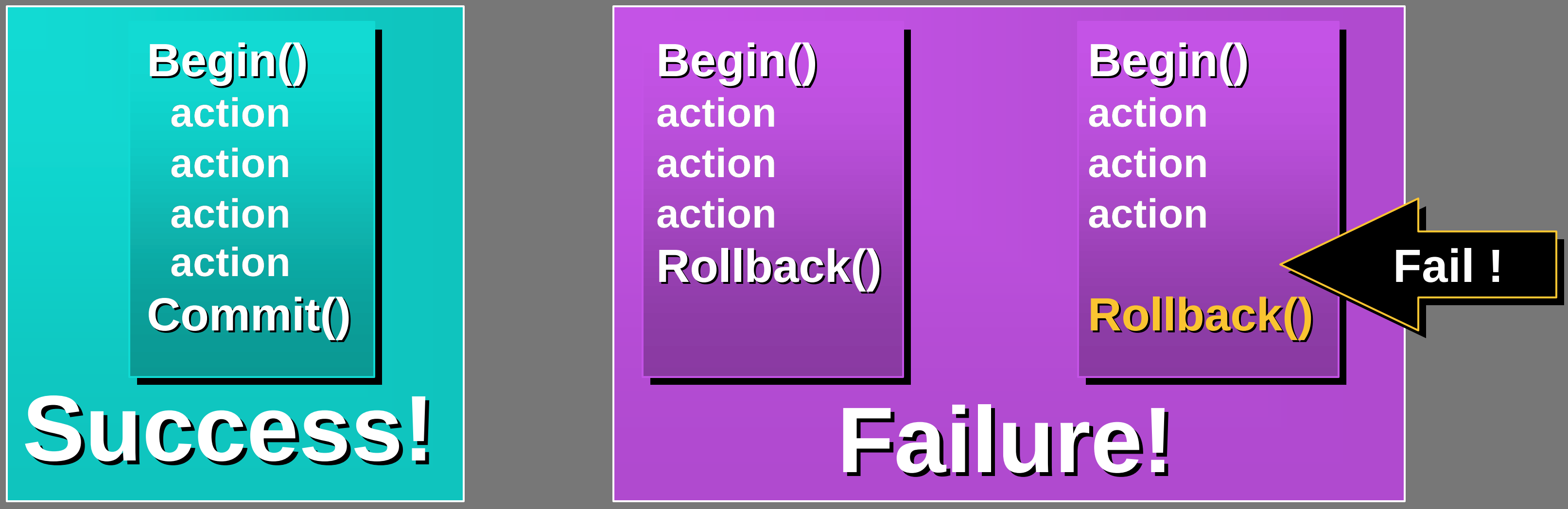
- Good DBMS performance relies on allowing concurrent access to the data by more than one client
- DBMS ensures that interleaved actions coming from different clients do not cause inconsistency in the data
  - E.g. two simultaneous bookings for the same airplane seat
- Each client is unaware of how many other clients are using the DBMS

Transactions

- A Transaction is an atomic sequence of actions in the Database (reads and writes)
- Each Transaction has to be executed *completely*, and must leave the Database in a consistent state
  - The definition of "consistent" is ultimately the client's responsibility!
- If the Transaction fails or aborts midway, then the Database is "rolled back" to its initial consistent state (when the Transaction began).

What Is A Transaction?

- Programmer's view:
  - Bracket a collection of actions
- A *simple* failure model
  - Only two outcomes:



Begin()
action
action
action
action
Commit()

Success!

Begin()
action
action
action
Rollback()

Begin()
action
action
action
Rollback()

Fail !

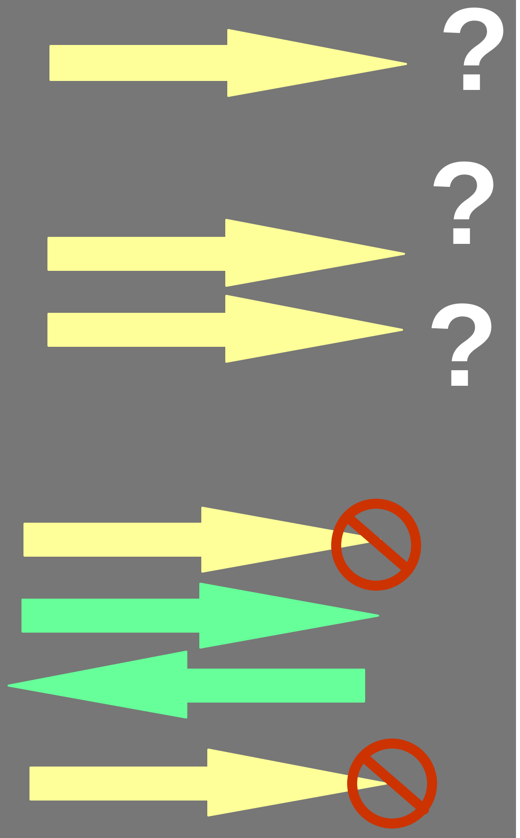
Failure!

ACID

- **Atomic**: all or nothing
- **Consistent**: state transformation
- **Isolated**: no concurrency anomalies
- **Durable**: committed transaction effects persist

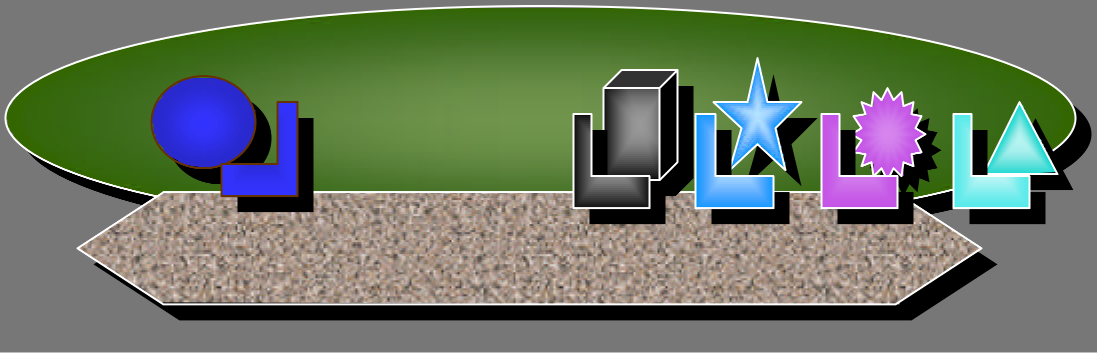
Why Bother: Atomicity?

- RPC semantics:
  - At most once: try one time
  - At least once: keep trying ’till acknowledged
  - Exactly once: keep trying ’till acknowledged and server discards duplicate requests



Why Bother: Atomicity?

- Example: insert record in file
  - At most once: time-out means “maybe”
  - At least once: retry may get “duplicate” error or retry may do second insert
  - Exactly once: you do not have to worry
- What if operation involves
  - Insert several records?
  - Send several messages?
- Want ALL or NOTHING for group of actions



Why Bother: Consistency

- Begin-Commit brackets a set of operations
- You can violate consistency inside brackets
  - Debit but not credit (destroys money)
  - Delete old file before create new file in a copy
  - Print document before delete from spool queue
- Begin and commit are points of consistency

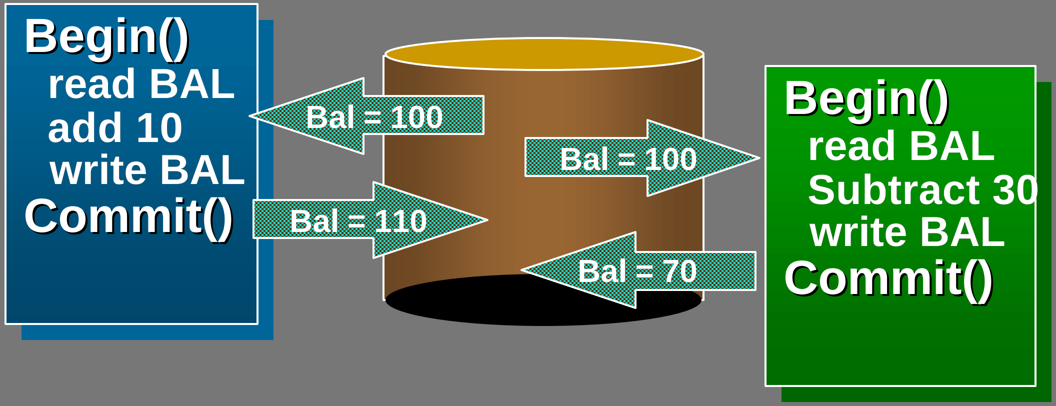
Begin

State transformations
new state under construction

Commit

Why Bother: Isolation

- Running programs concurrently on same data can create concurrency anomalies
  - The shared checking account example



Begin()
read BAL
add 10
write BAL
Commit()

Bal = 100

Bal = 110

Bal = 100

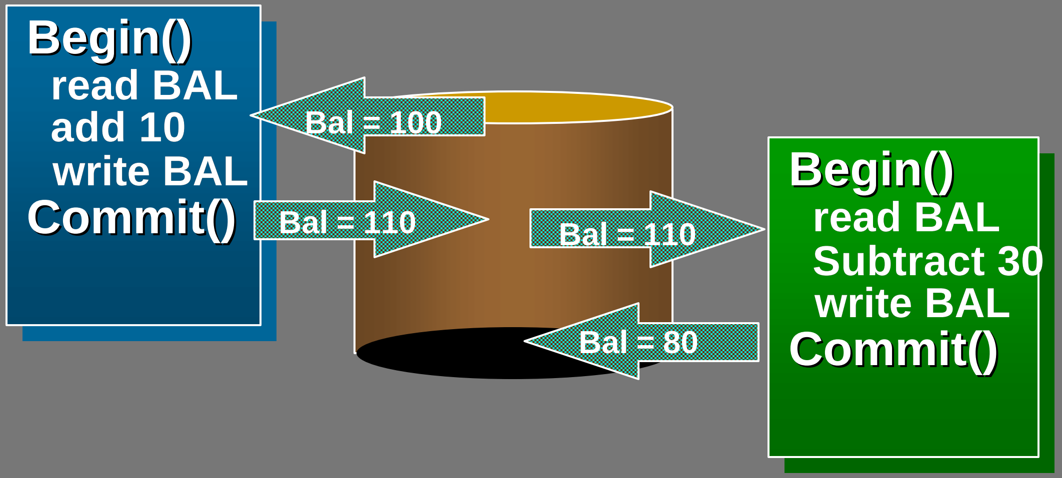
Bal = 70

Begin()
read BAL
Subtract 30
write BAL
Commit()

- Programming is hard enough without having to worry about concurrency

Isolation

- It is as though programs run one at a time
  - No concurrency anomalies
- System automatically protects applications
  - Locking (DB2, Informix, Microsoft® SQL Server™, Sybase…)
  - Versioned databases (Oracle, Interbase…)



Why Bother: Durability

- Once a transaction commits,
  want effects to survive failures
- Fault tolerance:
  old master-new master won’t work:
  - Can’t do daily dumps:
    would lose recent work
  - Want “continuous” dumps
- Redo “lost” transactions
  in case of failure
- Resend unacknowledged messages

Why ACID For Client/Server And Distributed

- ACID is important for centralized systems
- Failures in centralized systems are simpler
- In distributed systems:
  - More and more-independent failures
  - ACID is harder to implement
- That makes it even MORE IMPORTANT
  - Simple failure model
  - Simple repair model

ACID Generalizations

- Taxonomy of actions
  - Unprotected: not undone or redone
    - Temp files
  - Transactional: can be undone before commit
    - Database and message operations
  - Real: cannot be undone
    - Drill a hole in a piece of metal, print a check
- Nested transactions: subtransactions
- Work flow: long-lived transactions

Scheduling Transactions

- The DBMS has to take care of a set of Transactions that arrive concurrently
- It converts the concurrent Transaction set into a new set that can be executed sequentially
- It ensures that, before reading or writing an Object, each Transaction waits for a *Lock* on the Object
- Each Transaction releases all its Locks when finished
  - (Strict Two-Phase-Locking Protocol)

Concurrency Control

Locking

- How to automatically prevent concurrency bugs?
- Serialization theorem:
  - If you lock all you touch and hold to commit: no bugs
  - If you do not follow these rules, you may see bugs
- Automatic Locking:
  - Set automatically (well-formed)
  - Released at commit/rollback (two-phase locking)
- Greater concurrency for locks:
  - Granularity: objects or containers or server
  - Mode: shared or exclusive or…

Reduced Isolation Levels

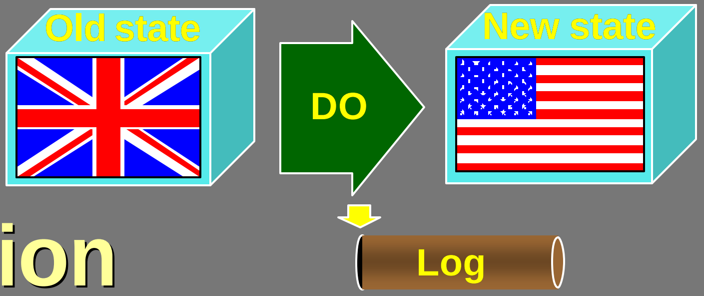
- It is possible to lock less and risk fuzzy data
- Example: want statistical summary of DB
  - But do not want to lock whole database
- Reduced levels:
  - Repeatable Read: may see fuzzy inserts/delete
    - But will serialize all updates
  - Read Committed: see only committed data
  - Read Uncommitted: may see uncommitted updates

Ensuring Atomicity

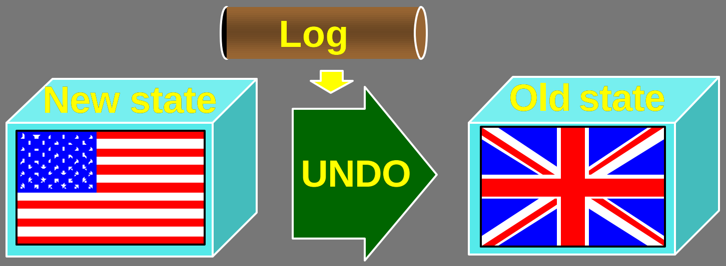
- The DBMS ensures the ***atomicity*** of a Transaction, even if the system crashes in the middle of it
- In other words ***all*** of the Transaction is applied to the Database, or ***none*** of it is
- How?
    - Keep a ***log/history*** of all actions carried out on the Database
    - Before making a change, put the log for the change somewhere "safe"
    - After a crash, effects of partially executed transactions are undone using the log

DO/UNDO/REDO

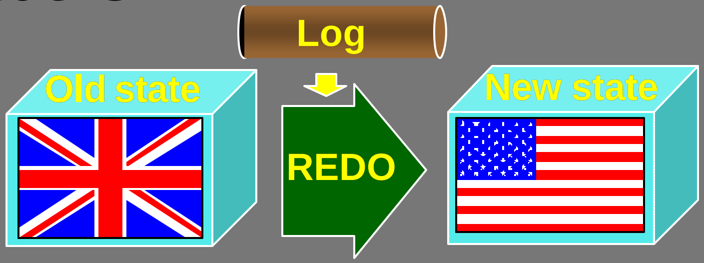
- Each action generates a log record



- Has an UNDO action



- Has a REDO action



What Does A Log Record Look Like?

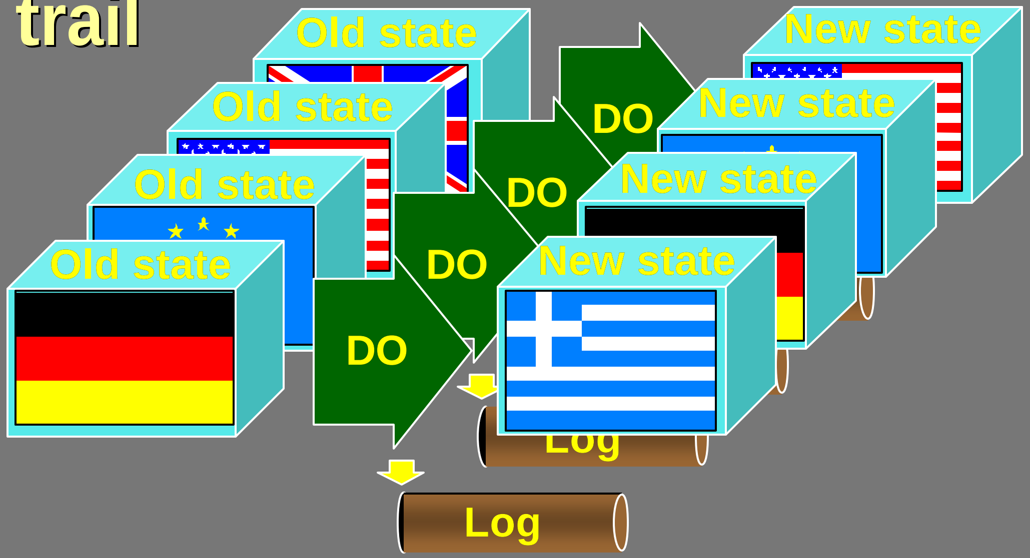
- Log record has
  - Header (transaction ID, timestamp…)
  - Item ID
  - Old value
  - New value

? Log ?

- For messages: just message text and sequence #
- For records: old and new value on update
- Keep records small

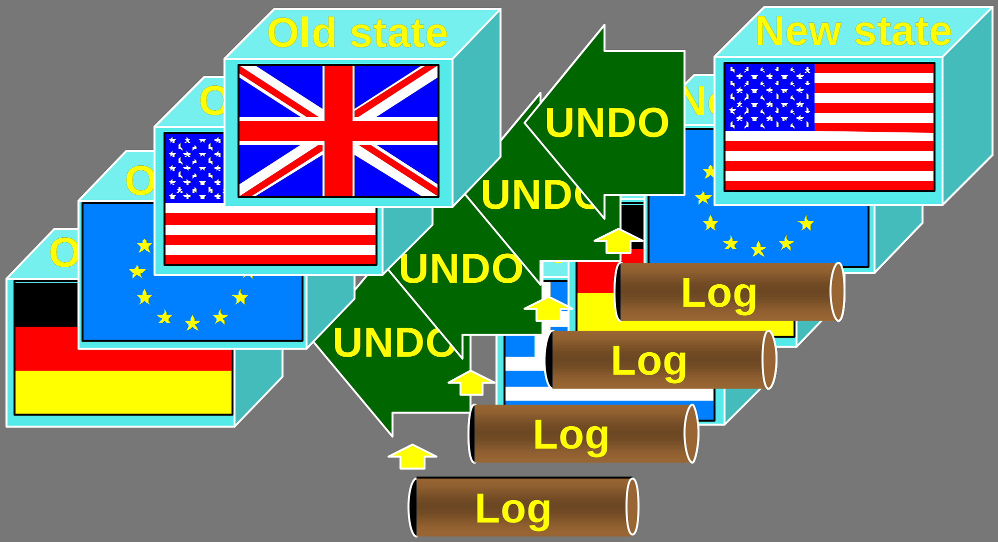
Transaction Is A Sequence Of Actions

- Each action changes state
  - Changes database
  - Sends messages
  - Operates a display/printer/drill press
- Leaves a log trail



Transaction UNDO Is Easy

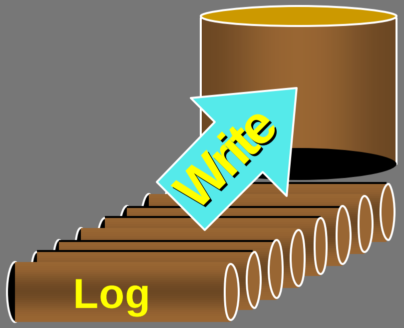
- ✍ Read log backwards
- ✍ UNDO one step at a time
- ✍ Can go half-way back to get nested transactions



J.J.Bunn, Distributed Databases, 2001

Durability: Protecting The Log

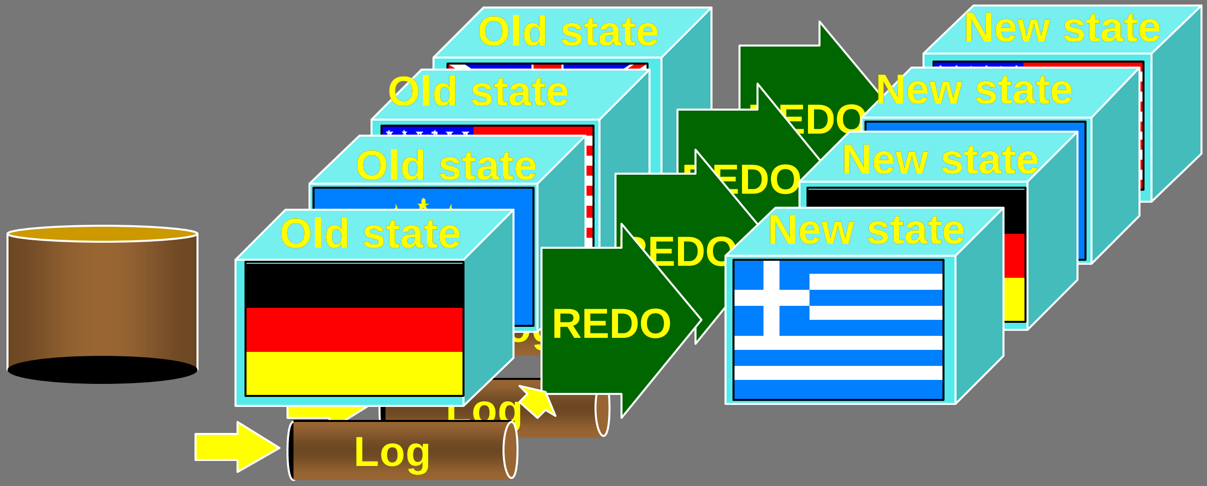
- When transaction commits
  - Put its log in a durable place (duplexed disk)
  - Need log to redo transaction in case of failure
    - System failure: lost in-memory updates
    - Media failure (lost disk)



- This makes transaction durable
- Log is sequential file
  - Converts random IO to single sequential IO
  - See NTFS or newer UNIX file systems

Recovery After System Failure

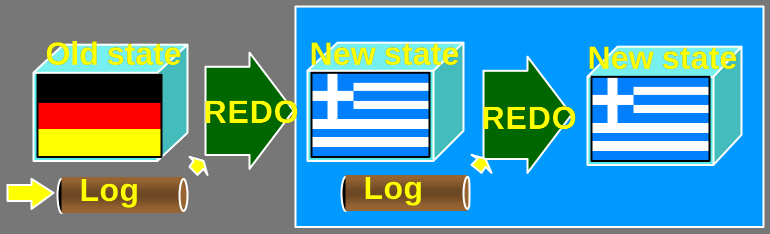
- ✍ During normal processing,
  write checkpoints on non-volatile storage
- ✍ When recovering from a system failure…
  - ✍ return to the checkpoint state
  - ✍ Reapply log of all committed transactions
  - ✍ Force-at-commit insures log will survive restart
- ✍ Then UNDO all uncommitted transactions

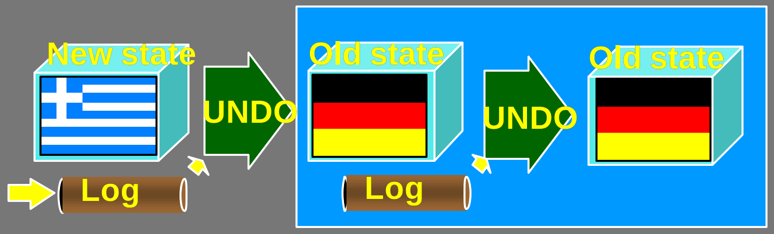


Idempotence

Dealing with failure

- What if fail during restart?
  - REDO many times
- What if new state not around at restart?
  - UNDO something not done

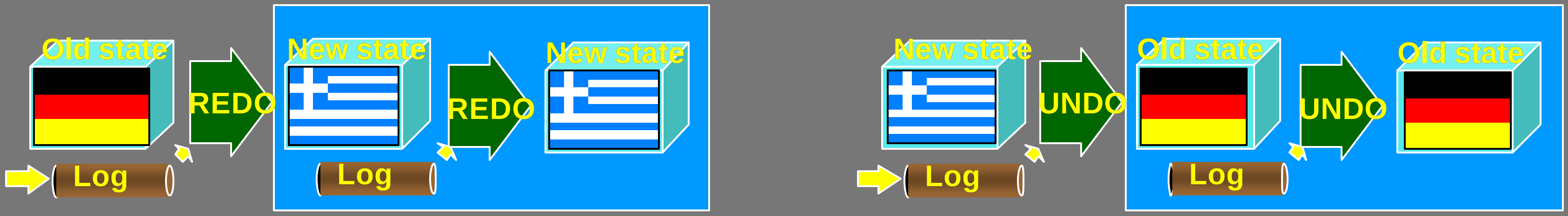




Idempotence

Dealing with failure

- Solution: make F(F(x))=F(x) (idempotence)
  - Discard duplicates
    - Message sequence numbers
      to discard duplicates
    - Use sequence numbers on pages to detect state
  - (Or) make operations idempotent
    - Move to position x, write value V to byte B…

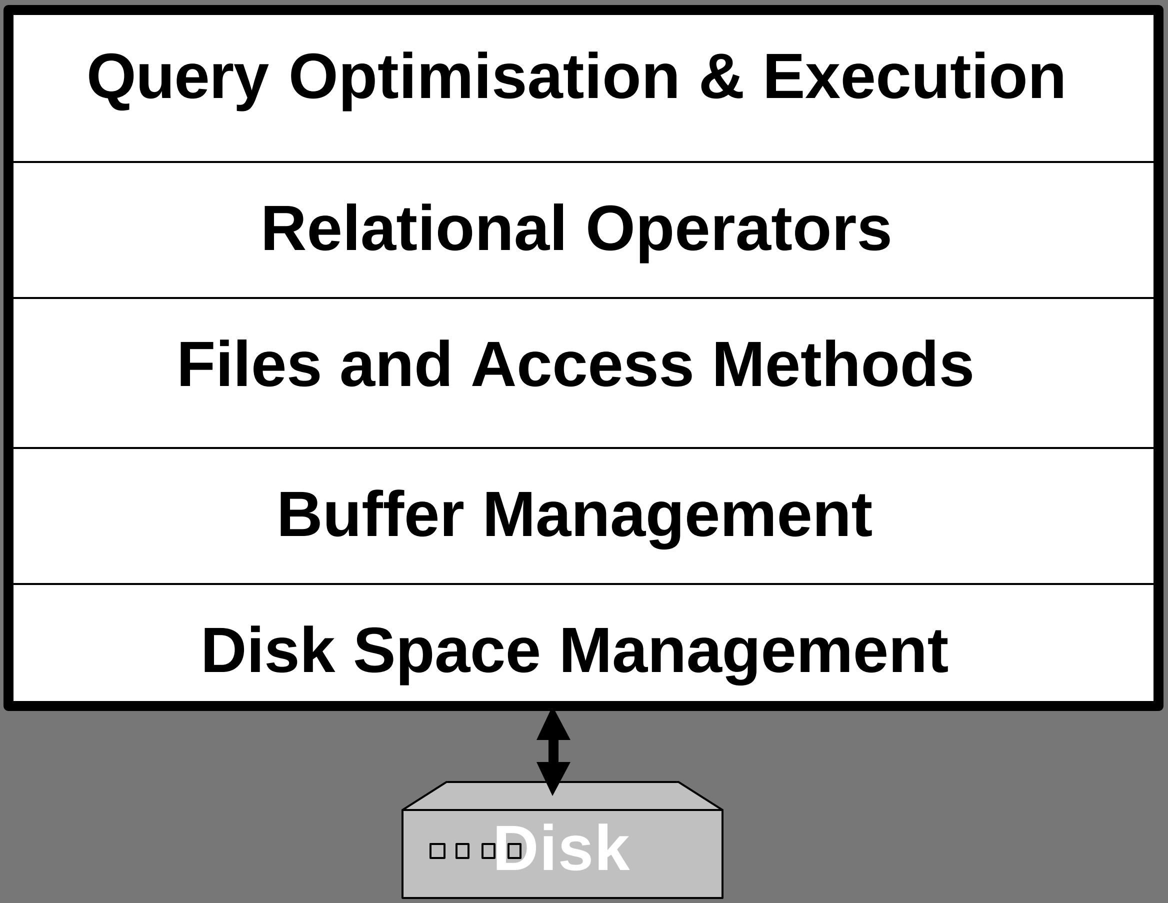


The Log: More Detail

- Actions recorded in the Log
  - Transaction writes an Object
    - Store in the Log: Transaction Identifier, Object Identifier, new value and old value
    - This must happen before actually writing the Object!
  - Transaction commits or aborts
- Duplicate Log on “stable” storage
- Log records chained by Transaction Identifier: easy to undo a Transaction

Structure of a Database

- Typical DBMS has a layered architecture



Database Administration

- Design Logical/Physical Schema
- Handle Security and Authentication
- Ensure Data Availability, Crash Recovery
- Tune Database as needs and workload evolves

Summary

- Databases are used to maintain and query large datasets
- DBMS benefits include recovery from crashes, concurrent access, data integrity and security, quick application development
- Abstraction ensures independence
- ACID
- Increasingly Important (and Big) in Scientific and Commercial Enterprises

Part 2

Distributed Databases

Julian Bunn

California Institute of Technology

Distributed Databases

- Data are stored at several locations
  - Each managed by a DBMS that can run autonomously
- Ideally, location of data is unknown to client
  - *Distributed Data Independence*
- Distributed Transactions are supported
  - Clients can write Transactions regardless of where the affected data are located
  - *Distributed Transaction Atomicity*
  - Hard, and in some cases undesirable
    - E.g. need to avoid overhead of ensuring location transparency

Types of Distributed Database

- Homogeneous: Every site runs the same type of DBMS
- Heterogeneous: Different sites run different DBMS (maybe even RDBMS and ODBMS)

Distributed DBMS Architectures

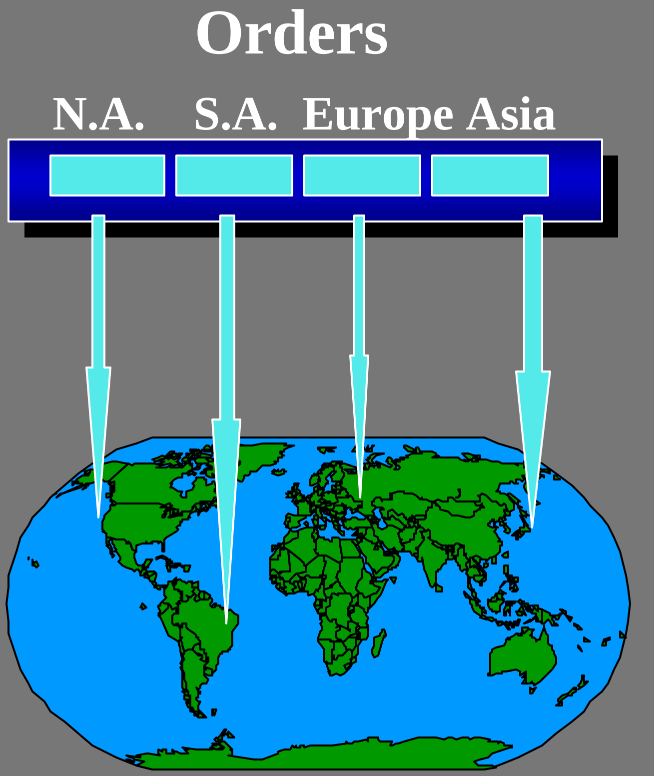
- Client-Servers
  - Client sends query to each database server in the distributed system
  - Client caches and accumulates responses
- Collaborating Server
  - Client sends query to “nearest” Server
  - Server executes query locally
  - Server sends query to other Servers, as required
  - Server sends response to Client

Storing the Distributed Data

- In fragments at each site
  - Split the data up
  - Each site stores one or more fragments
- In complete replicas at each site
  - Each site stores a replica of the complete data
- A mixture of fragments and replicas
  - Each site stores some replicas and/or fragments or the data

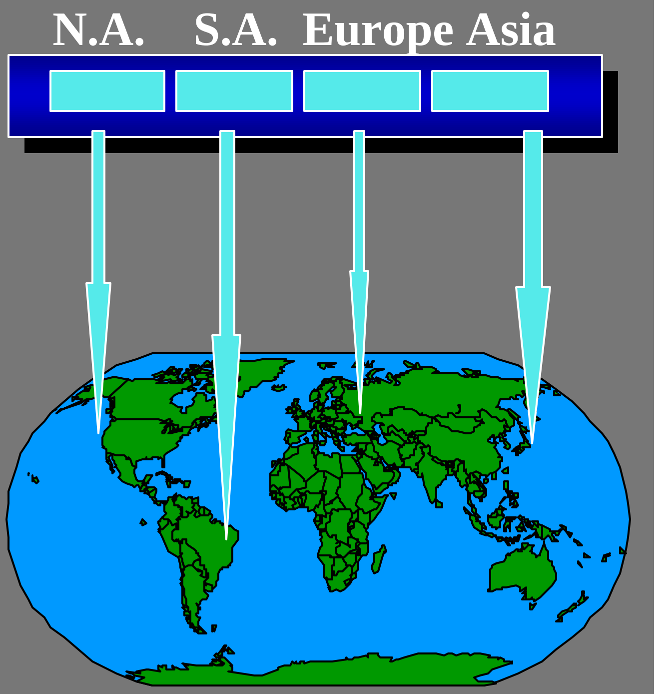
Partitioned Data
Break file into disjoint groups

- Exploit data access locality
  - Put data near consumer
  - Less network traffic
  - Better response time
  - Better availability
  - Owner controls data autonomy
- Spread Load
  - data or traffic may exceed single store



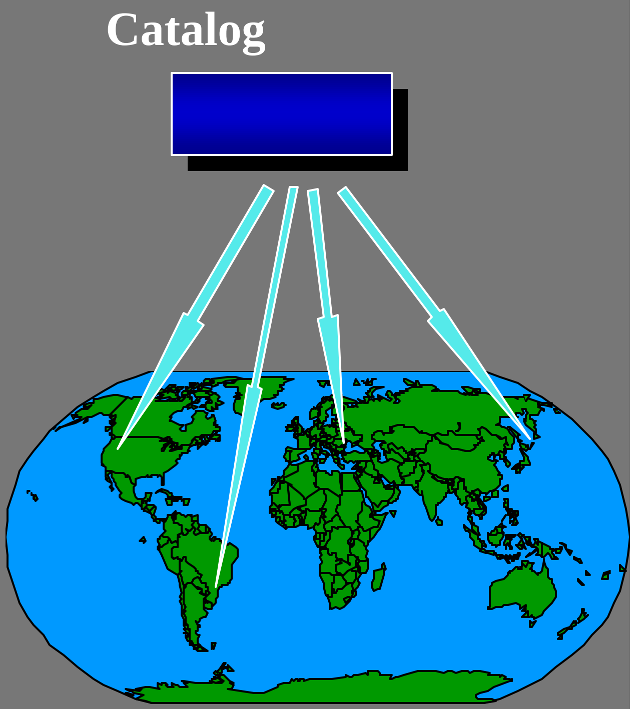
How to Partition Data?

- How to Partition
  - by attribute or
  - random or
  - by source or
  - by use
- Problem: to find it must have
  - Directory (replicated) or
  - Algorithm
- Encourages attribute-based partitioning



Replicated Data
Place fragment at many sites

- Pros:
  - + Improves availability
  - + Disconnected (mobile) operation
  - + Distributes load
  - + Reads are cheaper
- Cons:
  - N times more updates
  - N times more storage
- Placement strategies:
  - Dynamic: cache on demand
  - Static: place specific



Fragmentation

- Horizontal – “Row-wise”
  - E.g. rows of the table make up one fragment
- Vertical – “Column-Wise”
  - E.g. columns of the table make up one fragment

ID	#Particles	Energy	Event#	Run#	Date	Time
…	…	…	…	…	…	…
10001	3	121.5	111	13120	3/1406	13:30:55.0001
10002	3	202.2	112	13120	3/1406	13:30:55.0001
10003	4	99.3	113	13120	3/1406	13:30:55.0001
10004	5	231.9	120	13120	3/1406	13:30:55.0001
10005	6	287.1	125	13120	3/1406	13:30:55.0001
10006	6	107.7	126	13120	3/1406	13:30:55.0001
10007	6	98.9	127	13120	3/1406	13:30:55.0001
10008	9	100.1	128	13120	3/1406	13:30:55.0001
…	…	…	…	…	…	…

Replication

- Make synchronised or unsynchronised copies of data at servers
  - Synchronised: data are always current, updates are constantly shipped between replicas
  - Unsynchronised: good for read-only data
- Increases availability of data
- Makes query execution faster

Distributed Catalogue Management

- Need to know where data are distributed in the system
- At each site, need to name each replica of each data fragment
  - "Local name", "Birth Place"
- Site Catalogue:
  - Describes all fragments and replicas at the site
  - Keeps track of replicas of relations at the site
  - To find a relation, look up Birth site's catalogue: "Birth Place" site never changes, even if relation is moved

Replication Catalogue

- Which objects are being replicated
- Where objects are being replicated to
- How updates are propagated

- Catalogue is a set of tables that can be backed up, and recovered (as any other table)
- These tables are themselves replicated to each replication site
  - No single point of failure in the Distributed Database

Configurations

- Single Master with multiple read-only snapshot sites
- Multiple Masters
- Single Master with multiple updatable snapshot sites
- Master at record-level granularity
- Hybrids of the above

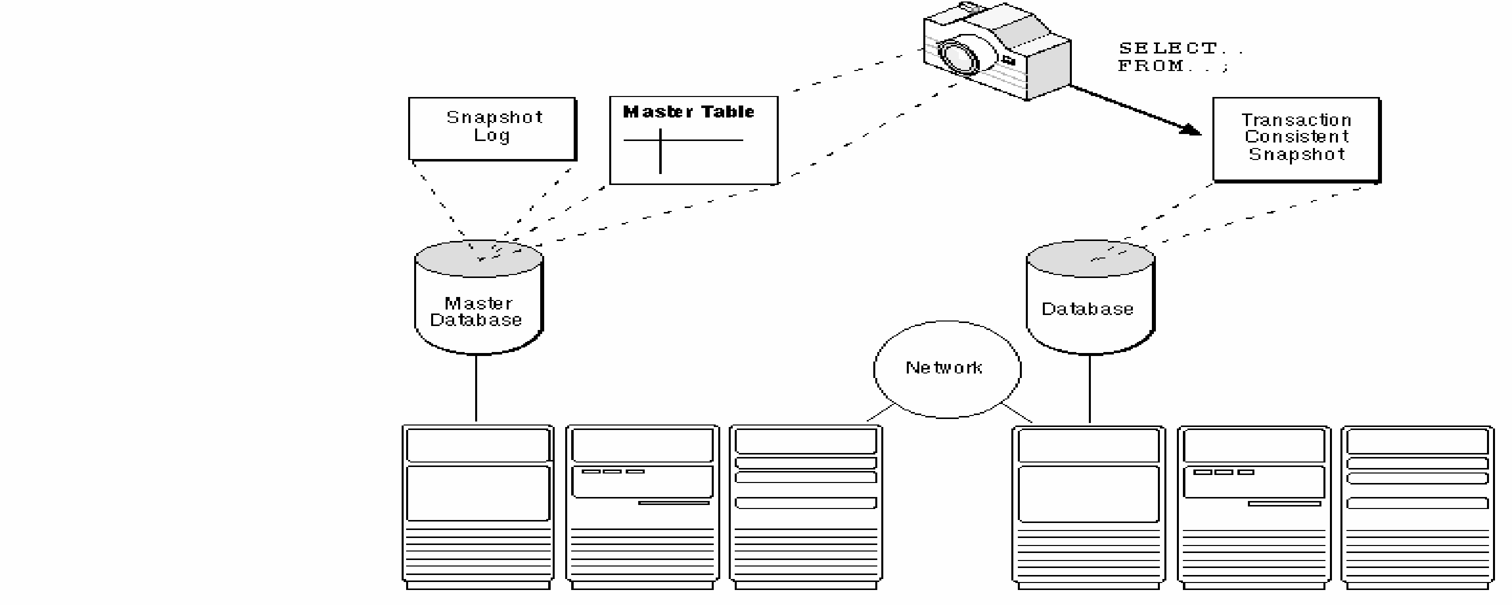


Table Replication Using Snapshots

Distributed Queries

Islamabad

ID	#Particles	Energy	Event#	Run#	Date	Time
...	...	...	...	...	...	...
10001	3	121.5	111	13120	3/1406	13:30:55.0001
10002	3	202.2	112	13120	3/1406	13:30:55.0001
10003	4	99.3	113	13120	3/1406	13:30:55.0001
10004	5	231.9	120	13120	3/1406	13:30:55.0001
10005	6	287.1	125	13120	3/1406	13:30:55.0001
10006	6	107.7	126	13120	3/1406	13:30:55.0001
10007	6	98.9	127	13120	3/1406	13:30:55.0001
10008	9	100.1	128	13120	3/1406	13:30:55.0001
...	...	...	...	...	...	...

Geneva

ID	#Particles	Energy	Event#	Run#	Date	Time
...	...	...	...	...	...	...
10001	3	121.5	111	13120	3/1406	13:30:55.0001
10002	3	202.2	112	13120	3/1406	13:30:55.0001
10003	4	99.3	113	13120	3/1406	13:30:55.0001
10004	5	231.9	120	13120	3/1406	13:30:55.0001
10005	6	287.1	125	13120	3/1406	13:30:55.0001
10006	6	107.7	126	13120	3/1406	13:30:55.0001
10007	6	98.9	127	13120	3/1406	13:30:55.0001
10008	9	100.1	128	13120	3/1406	13:30:55.0001
...	...	...	...	...	...	...

- SELECT AVG(E.Energy) FROM Events E WHERE E.particles > 3 AND E.particles < 7
- Replicated: Copies of the complete Event table at Geneva and at Islamabad
- Choice of where to execute query
  - Based on local costs, network costs, remote capacity, etc.

Distributed Queries (contd.)

- SELECT AVG(E.Energy) FROM Events E WHERE E.particles > 3 AND
  E.particles < 7

ID	#Particles	Energy	Event#	Run#	Date	Time
…	…	…	…	…	…	…
10001	3	121.5	111	13120	3/1406	13:30:55.0001
10002	3	202.2	112	13120	3/1406	13:30:55.0001
10003	4	99.3	113	13120	3/1406	13:30:55.0001
10004	5	231.9	120	13120	3/1406	13:30:55.0001
10005	6	287.1	125	13120	3/1406	13:30:55.0001
10006	6	107.7	126	13120	3/1406	13:30:55.0001
10007	6	98.9	127	13120	3/1406	13:30:55.0001
10008	9	100.1	128	13120	3/1406	13:30:55.0001
…	…	…	…	…	…	…

- Row-wise fragmented:

  Particles < 5 at Geneva, Particles > 4 at Islamabad

  - Need to compute SUM(E.Energy) and COUNT(E.Energy) at *both* sites
  - If WHERE clause had E.particles > 4 then only need to compute at Islamabad

Distributed Queries (contd.)

- **SELECT AVG(E.Energy) FROM Events E WHERE E.particles > 3 AND E.particles < 7**

- **Column-wise Fragmented:**

ID	#Particles	Energy	Event#	Run#	Date	Time
…	…	…	…	…	…	…
10001	3	121.5	111	13120	3/1406	13:30:55.0001
10002	3	202.2	112	13120	3/1406	13:30:55.0001
10003	4	99.3	113	13120	3/1406	13:30:55.0001
10004	5	231.9	120	13120	3/1406	13:30:55.0001
10005	6	287.1	125	13120	3/1406	13:30:55.0001
10006	6	107.7	126	13120	3/1406	13:30:55.0001
10007	6	98.9	127	13120	3/1406	13:30:55.0001
10008	9	100.1	128	13120	3/1406	13:30:55.0001
…	…	…	…	…	…	…

ID, Energy and Event# Columns at Geneva, ID and remaining Columns at Islamabad:

  - **Need to join on ID**
  - **Select IDs satisfying Particles constraint at Islamabad**
  - **SUM(Energy) and Count(Energy) for those IDs at Geneva**

Joins

- Joins are used to compare or combine relations (rows) from two or more tables, when the relations share a common attribute value
- Simple approach: for every relation in the first table “S”, loop over all relations in the other table “R”, and see if the attributes match
- N-way joins are evaluated as a series of 2-way joins
- Join Algorithms are a continuing topic of intense research in Computer Science

Join Algorithms

- Need to run in memory for best performance
- Nested-Loops: efficient only if “R” very small (can be stored in memory)
- Hash-Join: Build an in-memory hash table of “R”, then loop over “S” hashing to check for match
- Hybrid Hash-Join: When “R” hash is too big to fit in memory, split join into partitions
- Merge-Join: Used when “R” and “S” are already sorted on the join attribute, simply merging them in parallel
- Special versions of Join Algorithms needed for Distributed Database query execution!

Distributed Query Optimisation

- Cost-based:
  - Consider all "plans"
  - Pick cheapest: include communication costs
- Need to use distributed join methods
- Site that receives query constructs Global Plan, hints for local plans
  - Local plans may be changed at each site

Replication

- Synchronous: All data that have been changed must be propagated before the Transaction commits
- Asynchronous: Changed data are periodically sent
  - Replicas may go out of sync.
  - Clients must be aware of this

Synchronous Replication Costs

- Before an update Transaction can commit, it obtains locks on all modified copies
  - Sends lock requests to remote sites, holds locks
  - If links or remote sites fail, Transaction cannot commit until links/sites restored
  - Even without failure, *commit protocol* is complex, and involves many messages

Asynchronous Replication

- Allows Transaction to commit before all copies have been modified
- Two methods:
  - Primary Site
  - Peer-to-Peer

Primary Site Replication

- One copy designated as “Master”
- Published to other sites who subscribe to “Secondary” copies
- Changes propagated to “Secondary” copies
- Done in two steps:
  - Capture changes made by committed Transactions
  - Apply these changes

The Capture Step

- Procedural: A procedure, automatically invoked, does the capture (takes a snapshot)

- Log-based: the log is used to generate a Change Data Table
  - Better (cheaper and faster) but relies on proprietary log details

The Apply Step

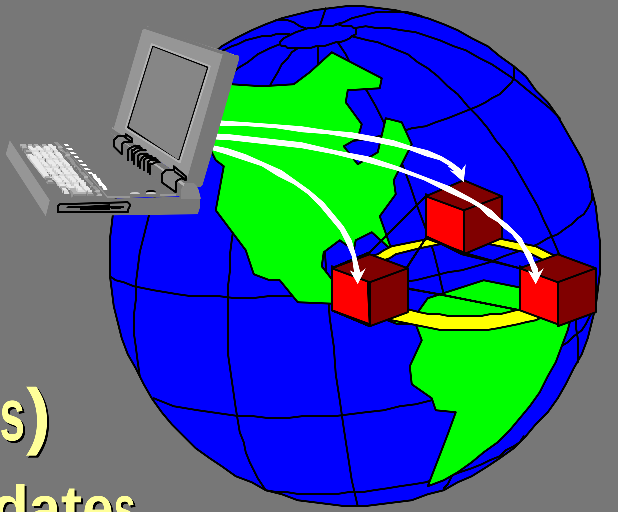
- The Secondary site periodically obtains from the Primary site a snapshot or changes to the Change Data Table
  - Updates its copy
  - Period can be timer-based or defined by the user/application
- Log-based capture with continuous Apply minimises delays in propagating changes

Peer-to-Peer Replication

- More than one copy can be “Master”
- Changes are somehow propagated to other copies
- Conflicting changes must be resolved
- So best when conflicts do not or cannot arise:
    - Each “Master” owns a disjoint fragment or copy
    - Update permission only granted to one “Master” at a time

Replication Examples

- Master copy, many slave copies (SQL Server)
  - always know the correct value (master)
  - change propagation can be
    - transactional
    - as soon as possible
    - periodic
    - on demand



- Symmetric, and anytime (Access)
  - allows mobile (disconnected) updates
  - updates propagated ASAP, periodic, on demand
  - non-serializable
  - colliding updates must be reconciled.
  - hard to know “real” value

Data Warehousing and Replication

- Build giant “warehouses” of data from many sites
  - Enable complex decision support queries over data from across an organisation
- Warehouses can be seen as an instance of asynchronous replication
  - Source data is typically controlled by different DBMS: emphasis on “cleaning” data by removing mismatches while creating replicas
- Procedural Capture and application Apply work best for this environment

Distributed Locking

- How to manage Locks across many sites?
  - Centrally: one site does all locking
    - Vulnerable to single site failure
  - Primary Copy: all locking for an object done at the primary copy site for the object
    - Reading requires access to locking site as well as site which stores object
  - Fully Distributed: locking for a copy done at site where the copy is stored
    - Locks at all sites while writing an object

Distributed Deadlock Detection

- Each site maintains a local “waits-for” graph
- Global deadlock might occur even if local graphs contain no cycles
  - E.g. Site A holds lock on X, waits for lock on Y
  - Site B holds lock on Y, waits for lock on X
- Three solutions:
  - Centralised (send all local graphs to one site)
  - Hierarchical (organise sites into hierarchy and send local graphs to parent)
  - Timeout (abort Transaction if it waits too long)

Distributed Recovery

- Links and Remote Sites may crash/fail
- If sub-transactions of a Transaction execute at different sites, all or none must commit
- Need a commit protocol to achieve this
- Solution: Maintain a Log at each site of commit protocol actions
  - Two-Phase Commit

Two-Phase Commit

- Site which originates Transaction is coordinator, other sites involved in Transaction are subordinates
- When the Transaction needs to Commit:
  - Coordinator sends “prepare” message to subordinates
  - Subordinates each force-writes an *abort* or *prepare* Log record, and sends “yes” or “no” message to Coordinator
  - If Coordinator gets unanimous “yes” messages, force-writes a *commit* Log record, and sends “commit” message to all subordinates
  - Otherwise, force-writes an *abort* Log record, and sends “abort” message to all subordinates
  - Subordinates force-write abort/commit Log record accordingly, then send an “ack” message to Coordinator
  - Coordinator writes *end* Log record after receiving all acks

Notes on Two-Phase Commit (2PC)

- First: voting, Second: termination – both initiated by Coordinator
- Any site can decide to abort the Transaction
- Every message is recorded in the local Log by the sender to ensure it survives failures
- All Commit Protocol log records for a Transaction contain the Transaction ID and Coordinator ID. The Coordinator’s abort/commit record also includes the Site IDs of all subordinates

Restart after Site Failure

- If there is a commit or abort Log record for Transaction T, but no end record, then must undo/redo T
  - If the site is Coordinator for T, then keep sending ***commit/abort*** messages to Subordinates until ***acks*** received
- If there is a prepare Log record, but no commit or abort:
  - This site is a Subordinate for T
  - Contact Coordinator to find status of T, then
    - write ***commit/abort*** Log record
    - ***Redo/undo*** T
    - Write ***end*** Log record

Blocking

- If Coordinator for Transaction T fails, then Subordinates who have voted “yes” cannot decide whether to *commit* or *abort* until Coordinator recovers!
- T is *blocked*
- Even if all Subordinates are aware of one another (e.g. via extra information in “prepare” message) they are blocked
  - Unless one of them voted “no”

Link and Remote Site Failures

- ***If*** a Remote Site does not respond during the Commit Protocol for T
  - E.g. it crashed or the link is down
- ***Then***
  - If current Site is Coordinator for T: abort
  - If Subordinate and not yet voted “yes”: abort
  - If Subordinate and has voted “yes”, it is blocked until Coordinator back online

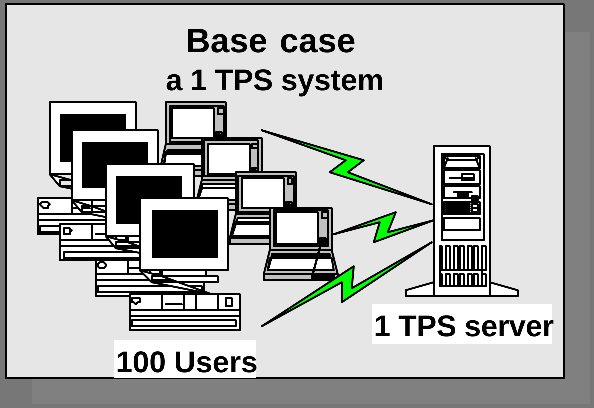
Observations on 2PC

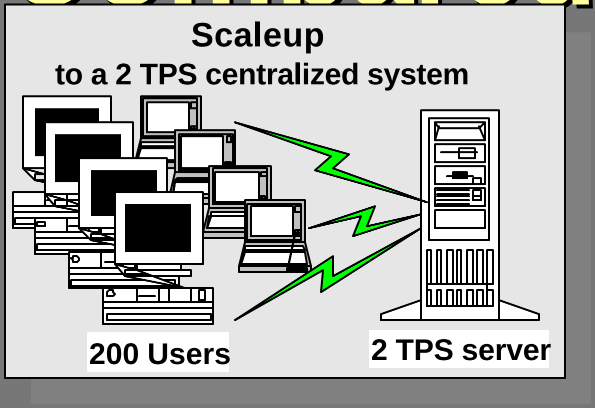
- Ack messages used to let Coordinator know when it can “forget” a Transaction
  - Until it receives all acks, it must keep T in the Transaction Table
- If Coordinator fails after sending “prepare” messages, but before writing commit/abort Log record, when it comes back up, it aborts T
- If a subtransaction does no updates, its commit or abort status is irrelevant

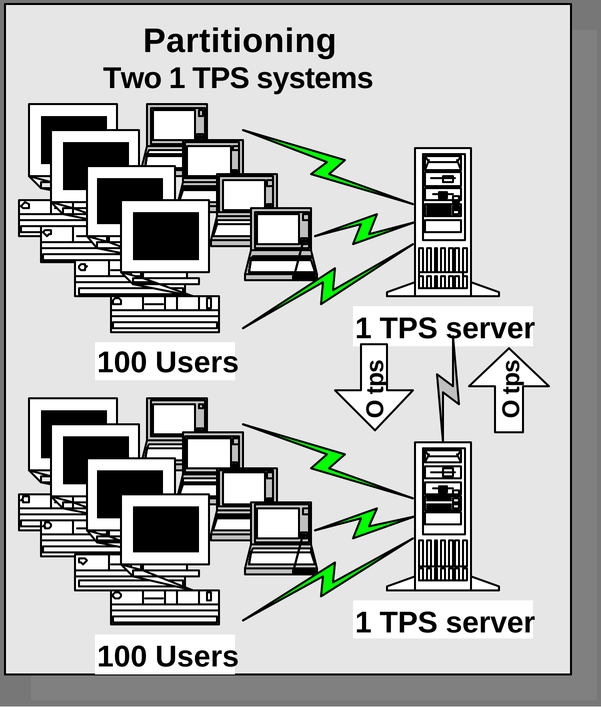
2PC with Presumed Abort

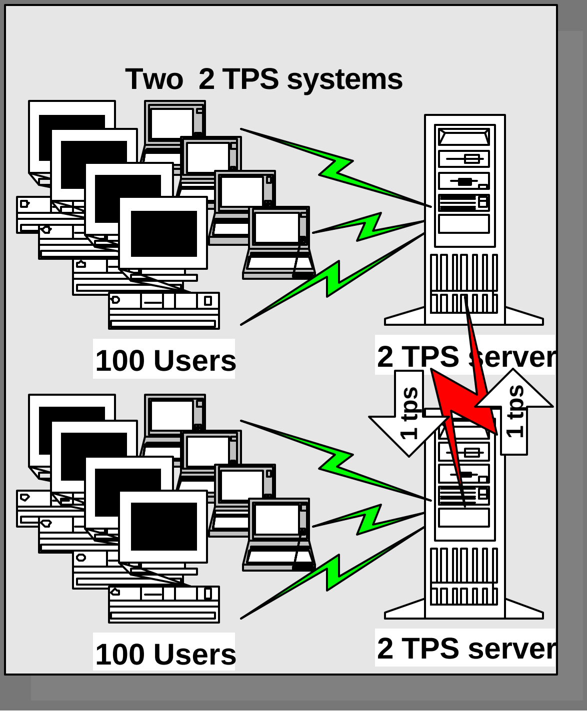
- When Coordinator aborts T, it undoes T and removes it from the Transaction Table immediately
  - Doesn't wait for "acks"
  - "Presumes Abort" if T not in Transaction Table
  - Names of Subordinates not recorded in ***abort*** Log record
- Subordinates do not send "ack" on abort
- If subtransaction does no updates, it responds to "prepare" message with "reader" (instead of "yes"/"no")
- Coordinator subsequently ignores "reader"s
- If all Subordinates are "reader"s, then 2nd. Phase not required

Replication and Partitioning Compared





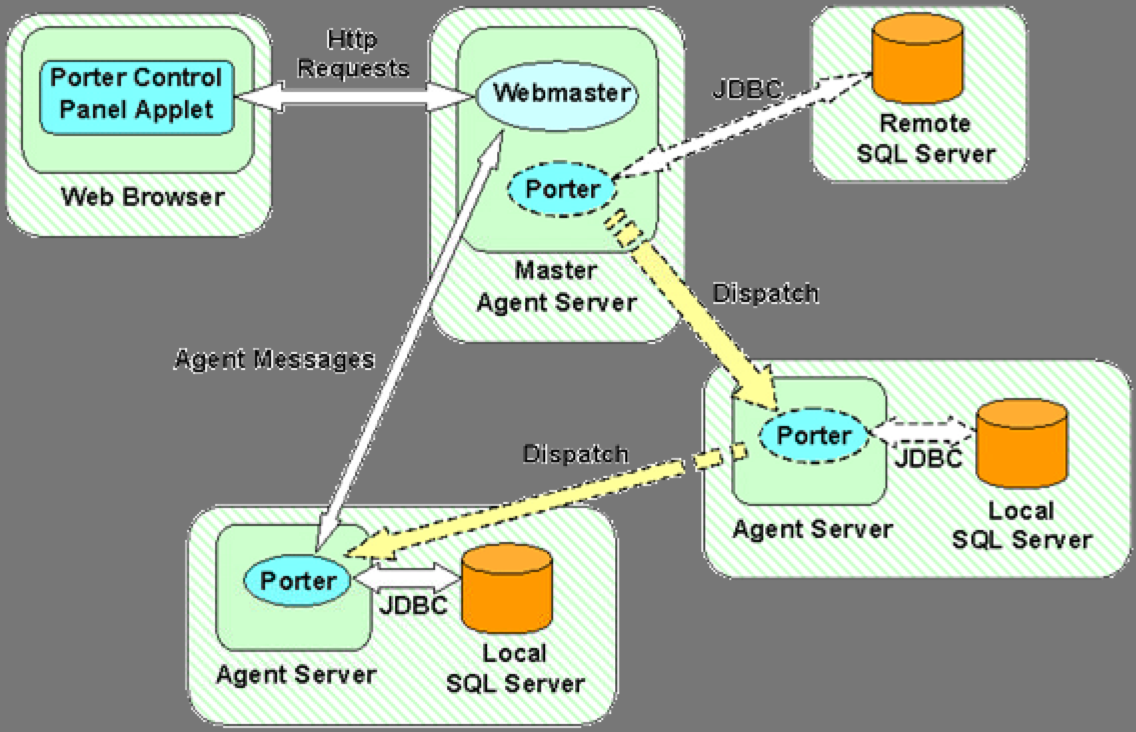




- *Central Scaleup 2x* more work
- *Partition Scaleup 2x* more work
- *Replication Scaleup 4x* more work

“Porter” Agent-based Distributed Database

- Charles Univ, Prague
- Based on “Aglets” SDK from IBM



Part 3

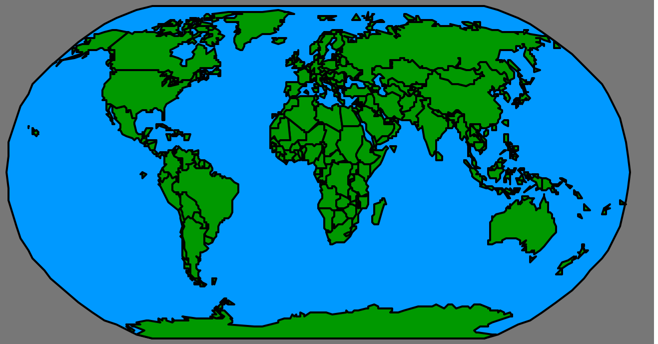
Distributed Systems

Julian Bunn

California Institute of Technology

What’s a Distributed System?

- Centralized:
  - everything in one place
  - stand-alone PC or Mainframe
- Distributed:
  - some parts remote
    - distributed users
    - distributed execution
    - distributed data

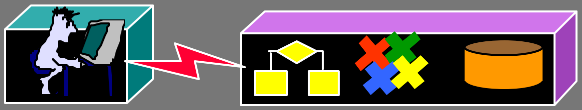


Why Distribute?

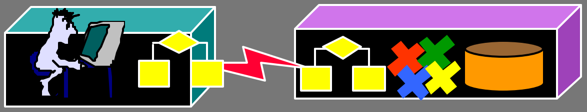
- No best organization
- Organisations constantly swing between
  - Centralized: focus, control, economy
  - Decentralized: adaptive, responsive, competitive
- Why distribute?
  - reflect organisation or application structure
  - empower users / producers
  - improve service (response / availability)
  - distribute load
  - use PC technology (economics)

What
Should Be Distributed?

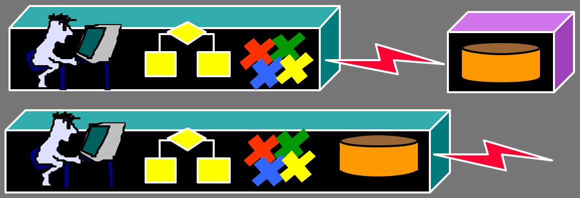
- Users and User Interface
  - Thin client



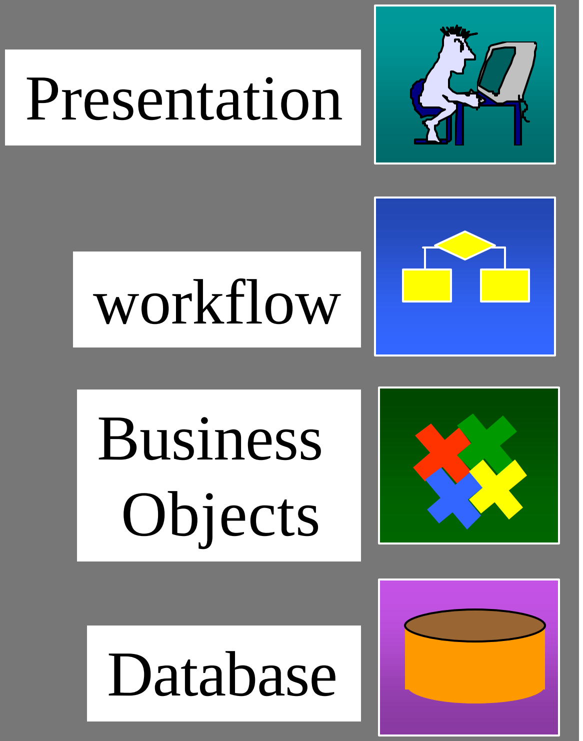
- Processing



  - Trim client
- Data
  - Fat client



- Will discuss tradeoffs later



Presentation

workflow

Business Objects

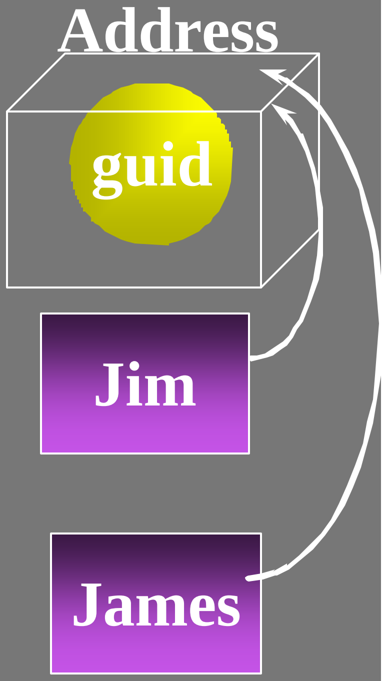
Database

Transparency in Distributed Systems

- Make distributed system as easy to use and manage as a centralized system
- Give a Single-System Image
- <u>Location transparency:</u>
  - hide fact that object is remote
  - hide fact that object has moved
  - hide fact that object is partitioned or replicated
- Name doesn’t change if object is replicated, partitioned or moved.

Naming - The basics

- Objects have
  - Globally Unique Identifier (GUIDs)
  - location(s) = address(es)
  - name(s)
  - addresses can change
  - objects can have many names
- Names are context dependent:
  - (Jim @ KGB ? ? ? ?? ? ?? ? ? ?? ? ?Jim @ CIA)
- Many naming systems
  - UNC: \\node\device\dir\dir\dir\object
  - Internet: http://node.domain.root/dir/dir/dir/object
  - LDAP: ldap://ldap.domain.root/o=org,c=US,cn=dir

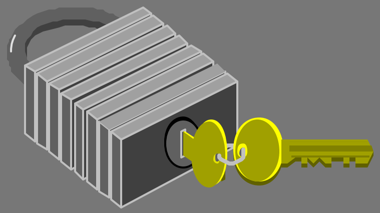


Name Servers in Distributed Systems

- Name servers translate
  names + context
  to address (+ GUID)
- Name servers are partitioned
  (subtrees of name space)
- Name servers replicate root
  of name tree
- Name servers form a hierarchy
- Distributed data from hell:
  - high read traffic
  - high reliability & availability
  - autonomy

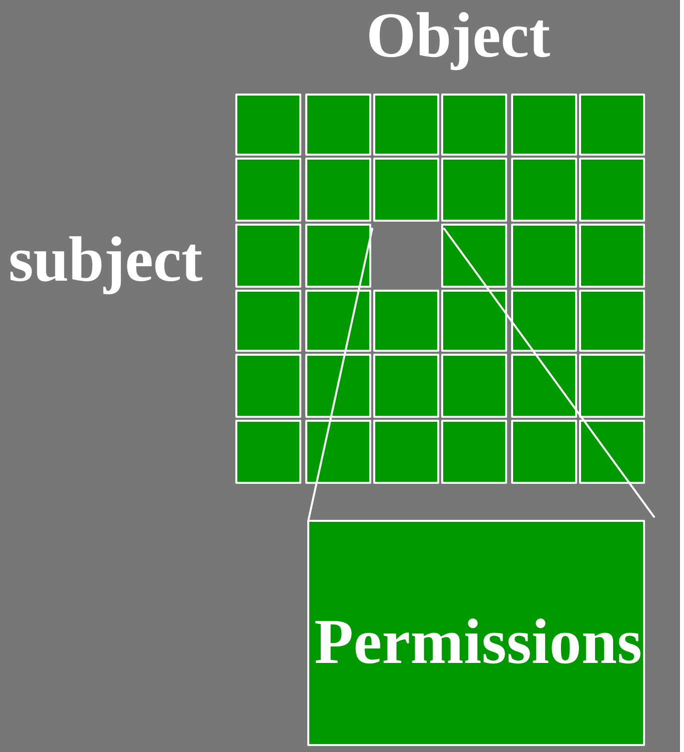
Autonomy in Distributed Systems

- Owner of site (or node, or application, or database) Wants to control it
- If my part is working, must be able to access & manage it (reorganize, upgrade, add user,…)
- Autonomy is
  - Essential
  - Difficult to implement.
  - Conflicts with global consistency
- examples: naming, authentication, admin…



Security
The Basics

- Authentication server
  subject + Authenticator = >
  (Yes + token) | No
- Security matrix:
  - who can do what to whom
  - Access control list is column of matrix
  - "who" is authenticated ID
- In a distributed system, "who" and "what" and "whom" are distributed objects



Security in Distributed Systems

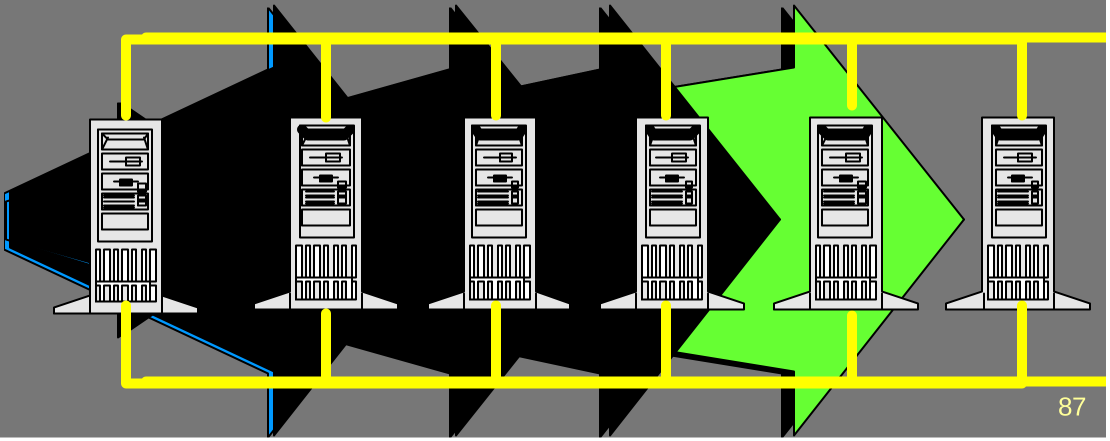
- <u>Security domain</u>:
  nodes with a shared security server.
- Security domains can have trust relationships:
  - A trusts B: A “believes” B when it says this is Jim@B
- Security domains form a hierarchy.
- **<u>Delegation:</u>** passing authority to a server
  when A asks B to do something (e.g. print a file, read a database)
  B may need A’s authority
- Autonomy requires:
  - each node is an authenticator
  - each node does own security checks
- Internet Today:
  - no trust among domains (fire walls, many passwords)
  - trust based on digital signatures

Clusters

The Ideal Distributed System.

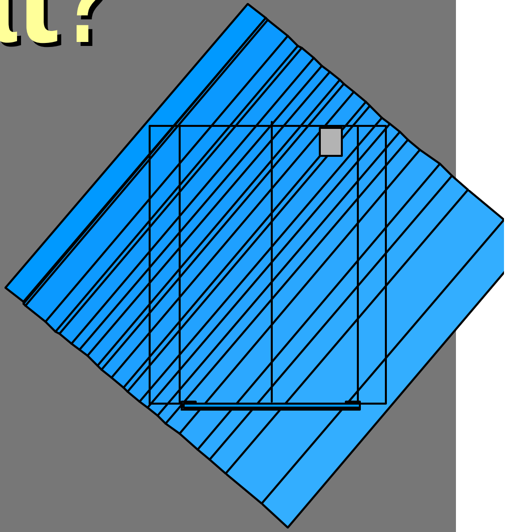
- Cluster is distributed system BUT single
  - location
  - manager
  - security policy
- relatively homogeneous
- communications is
  - high bandwidth
  - low latency
  - low error rate

- Clusters use distributed system techniques for
  - load distribution
    - storage
    - execution
  - growth
  - fault tolerance

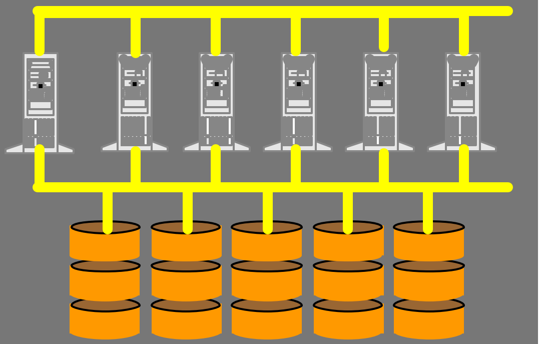


Cluster: Shared What?

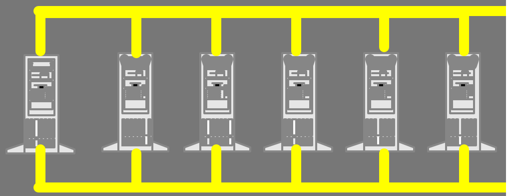
- Shared Memory Multiprocessor
  - Multiple processors, one memory
  - all devices are local
  - HP V-class



- Shared Disk Cluster
  - an array of nodes
  - all shared common disks
  - VAXcluster + Oracle



- Shared Nothing Cluster
  - each device local to a node
  - ownership may change
  - Beowulf,Tandem, SP2, Wolfpack

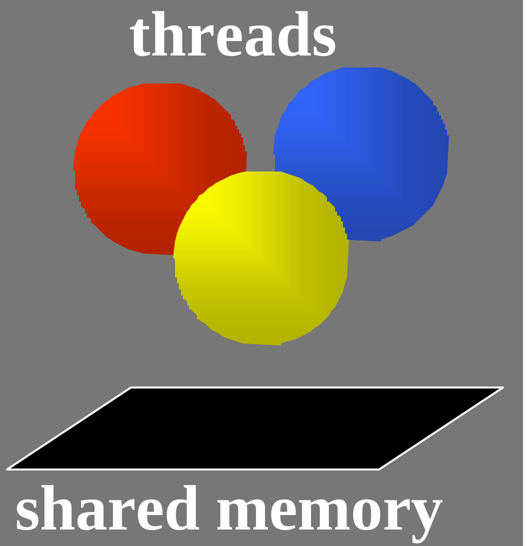


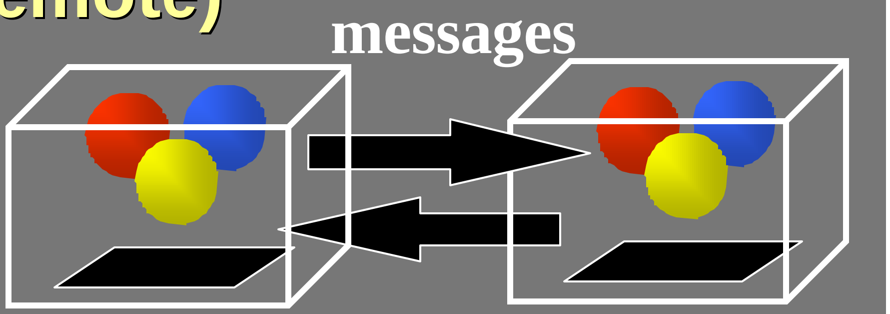
Distributed Execution

Threads and Messages

- Thread is Execution unit

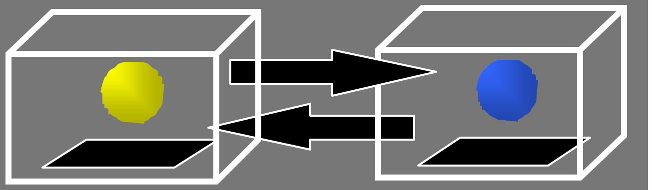
  (software analog of cpu+memory)
- Threads execute at a node
- Threads communicate via
  - Shared memory (local)
  - Messages (local and remote)



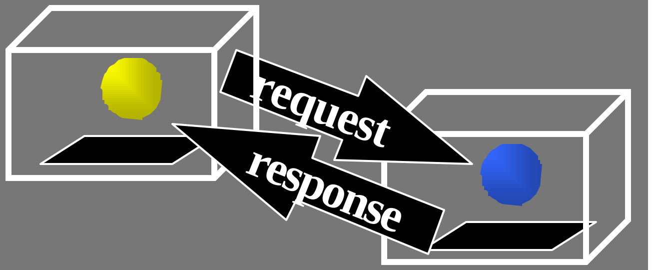


Peer-to-Peer or Client-Server

- Peer-to-Peer is symmetric:
  - Either side can send

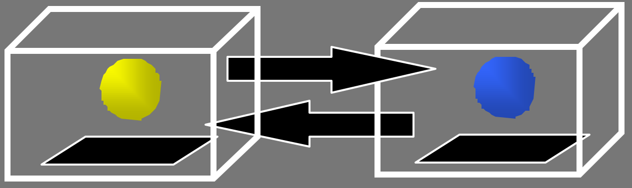


- Client-server
  - client sends requests
  - server sends responses
  - simple subset of peer-to-peer

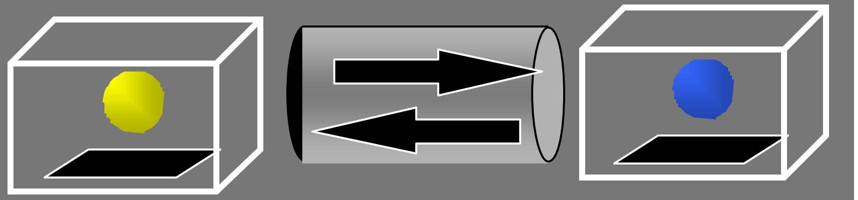


Connection-less or Connected

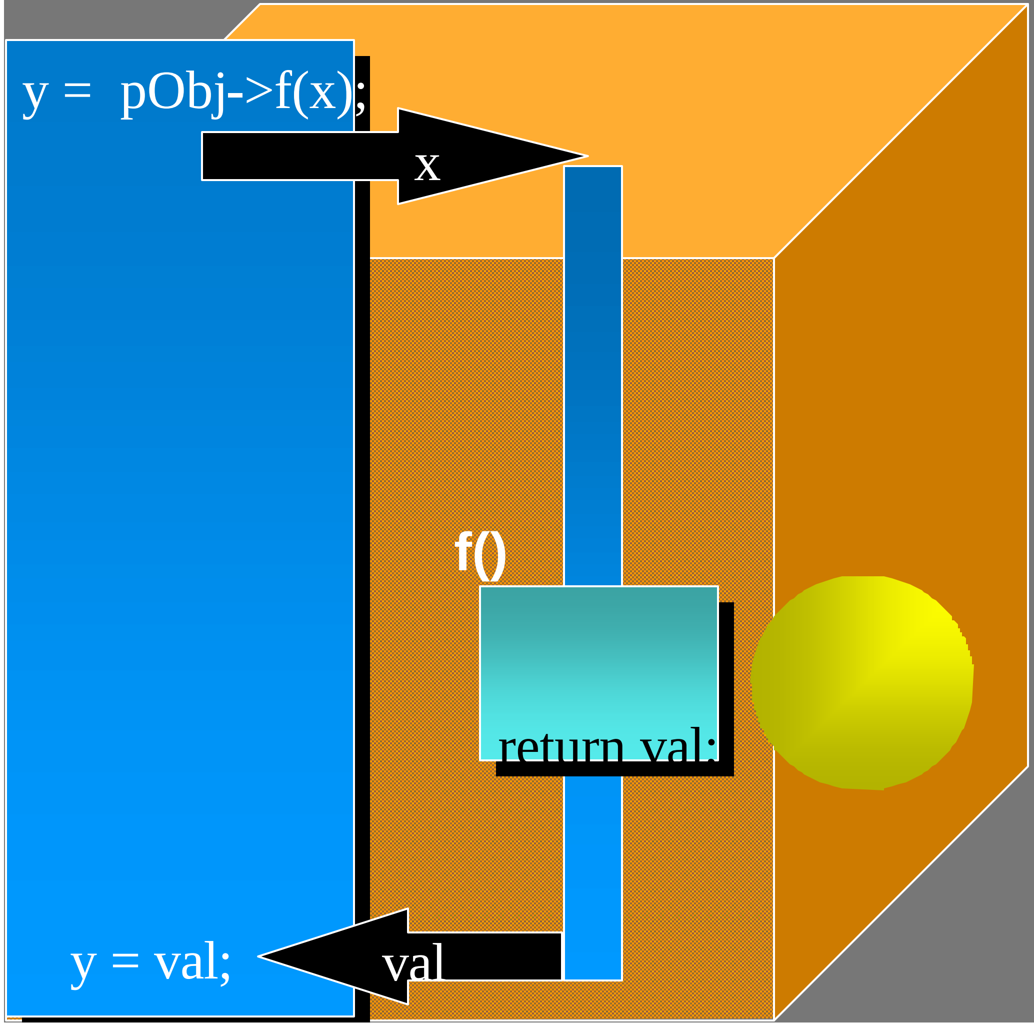
- Connection-less
  - request contains
    - client id
    - client context
    - work request
  - client authenticated on each message
  - only a single response message
  - e.g. HTTP, NFS v1



- Connected (sessions)
  - open - request/reply - close
  - client authenticated once
  - Messages arrive in order
  - Can send many replies (e.g. FTP)
  - Server has client context (context sensitive)
  - e.g. Winsock and ODBC
  - HTTP adding connections



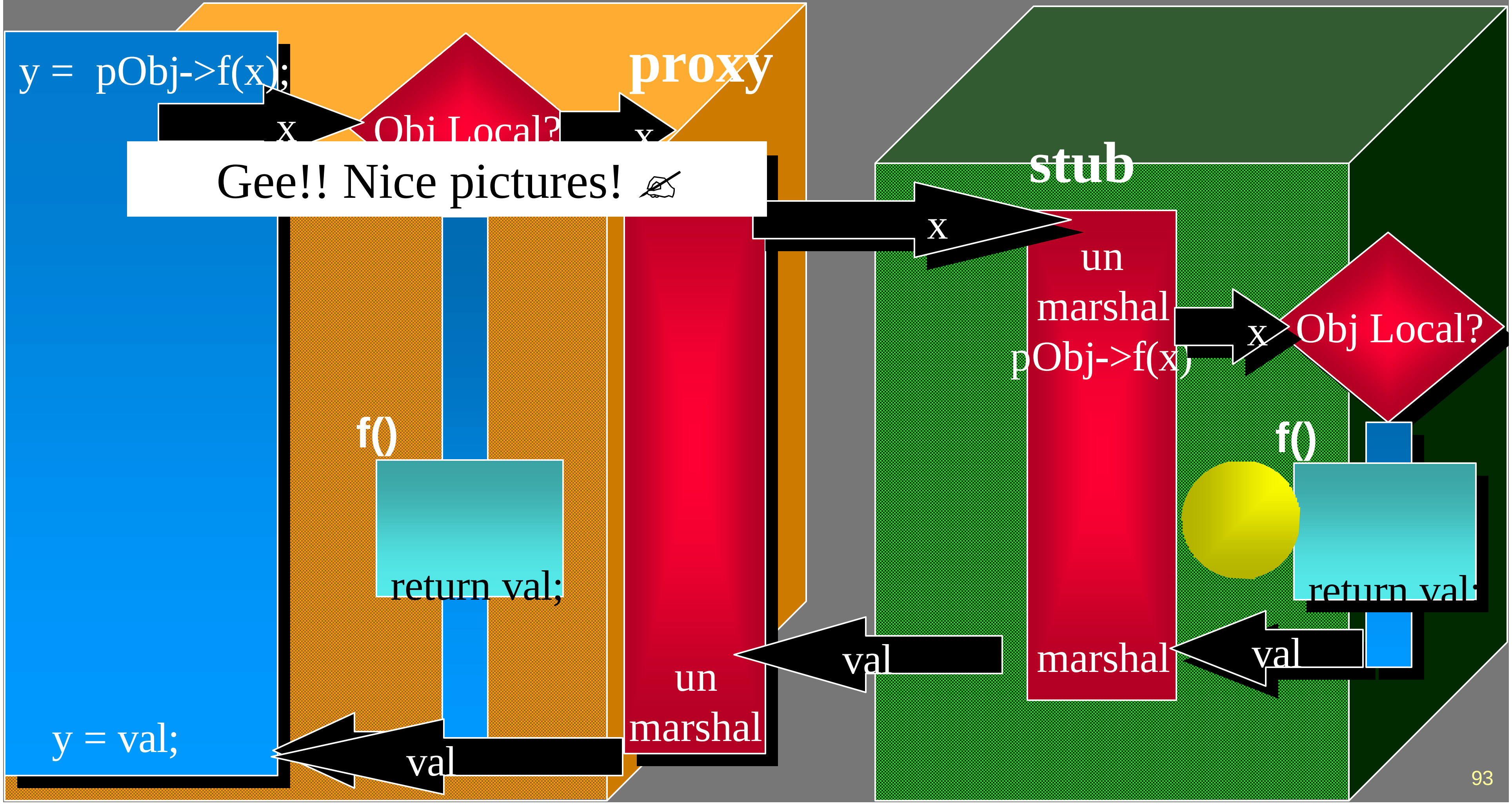
Remote Procedure Call: The key to transparency



- Object may be local or remote
- Methods on object work wherever it is.
- Local invocation

Remote Procedure Call: The key to transparency

- ✍ Remote invocation



y = pObj->f(x);

x

Obj Local?

x

proxy

Gee!! Nice pictures! ✍

stub

x

un marshal pObj->f(x)

x

Obj Local?

f()

return val;

f()

return val;

val

marshal

val

un marshal

y = val;

val

Object Request Broker (ORB)

Orchestrates RPC

- Registers Servers
- Manages pools of servers
- Connects clients to servers
- Does Naming, request-level authorization,
- Provides transaction coordination (new feature)
- Old names:
  - Transaction Processing Monitor,
  - Web server,
  - NetWare

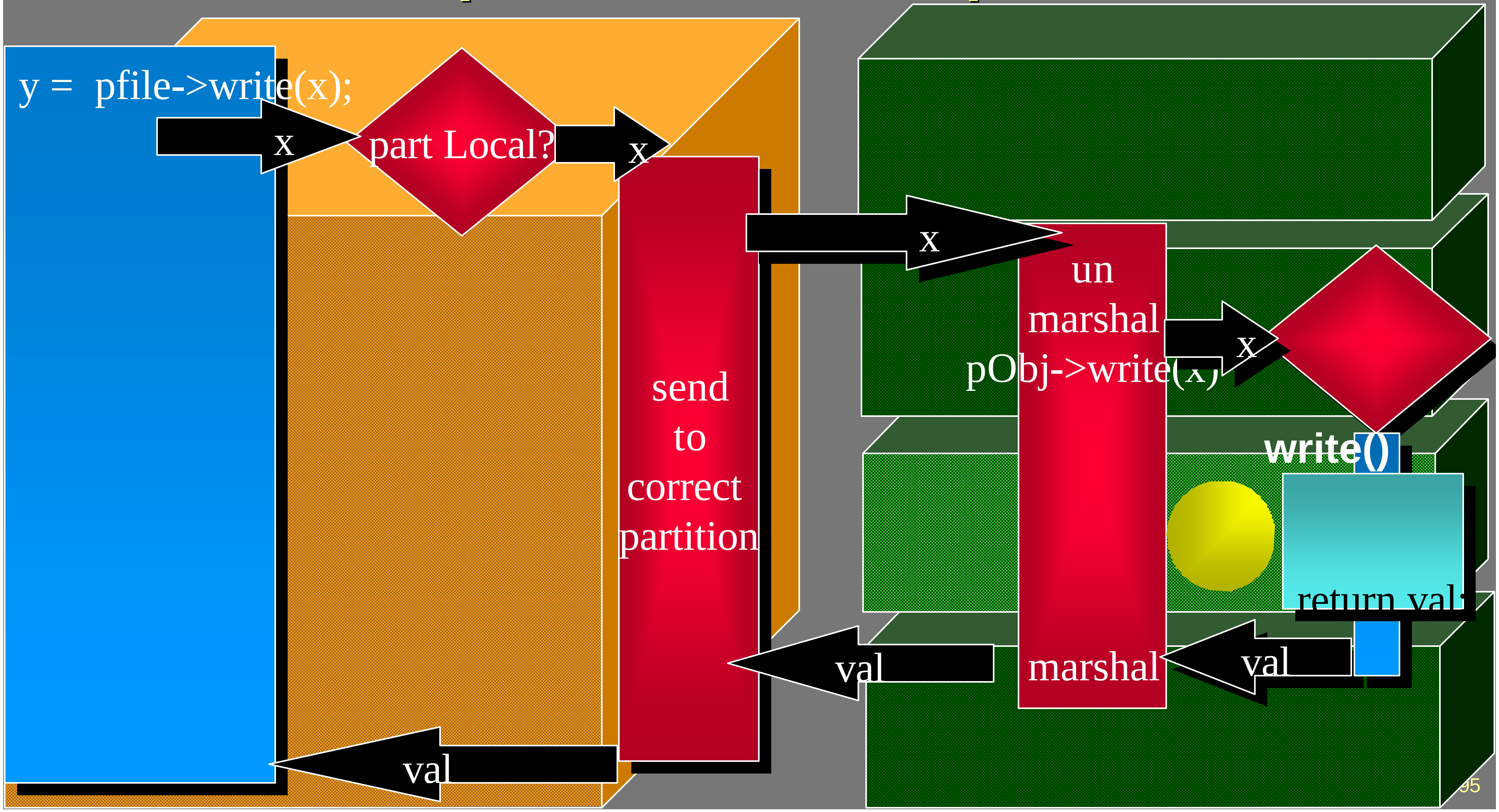
Transaction

Object-Request Broker

Using RPC for Transparency

Partition Transparency

- ✍ Send updates to correct partition



y = pfile->write(x);

x

part Local?

x

send
to
correct
partition

x

un
marshal
pObj->write(x)

x

write()

return val;

val

marshal

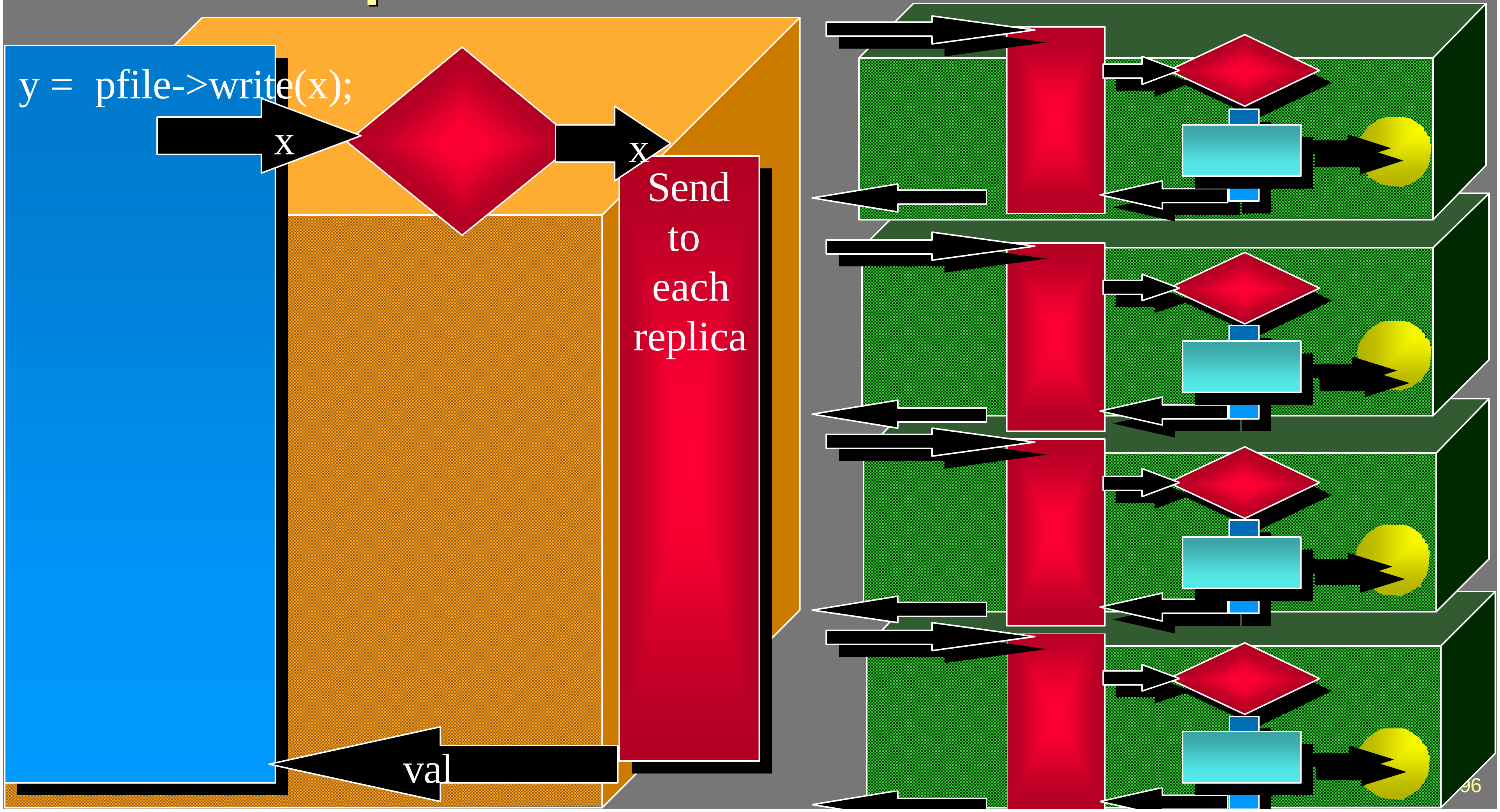
val

val

Using RPC for Transparency

Replication Transparency

- ✍ Send updates to EACH node



Client/Server Interactions

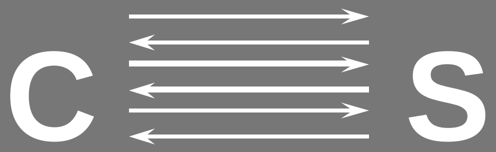
All can be done with RPC

- **Request-Response**

  response may be many messages

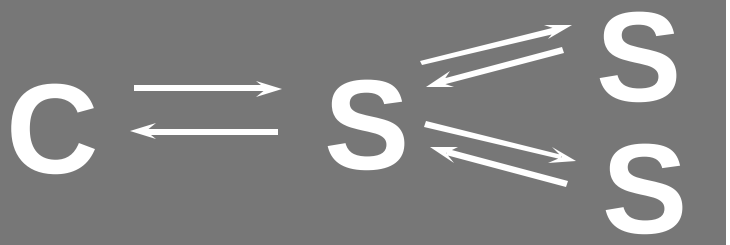
- **Conversational**

  server keeps client context



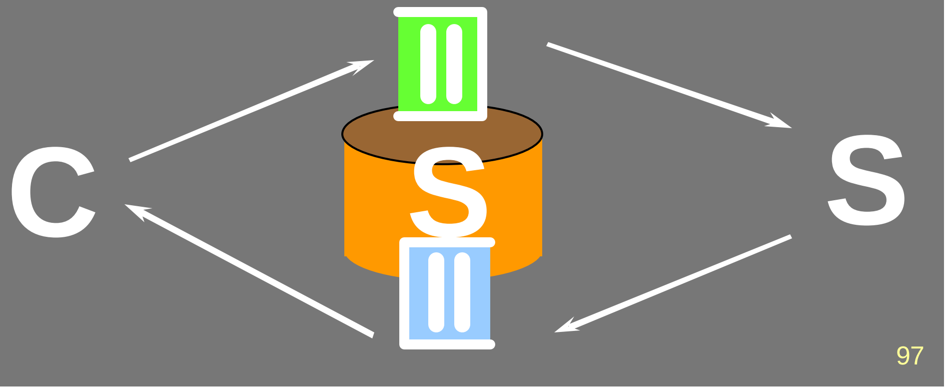
- **Dispatcher**

  three-tier: complex operation at server



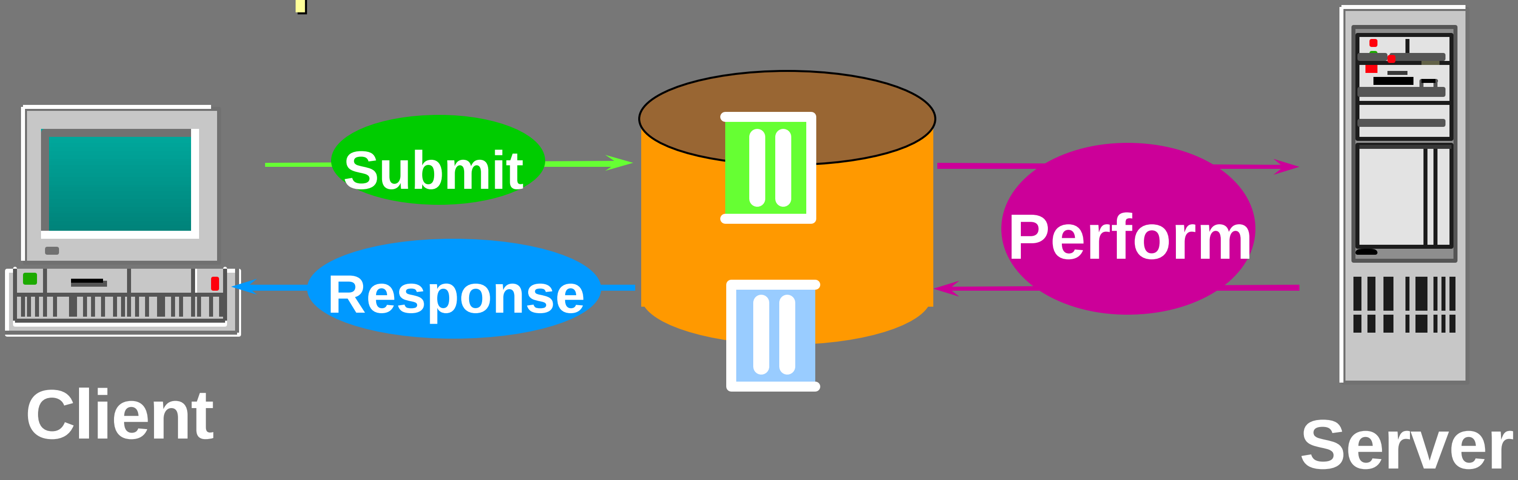
- **Queued**

  de-couples client from server
  allows disconnected operation



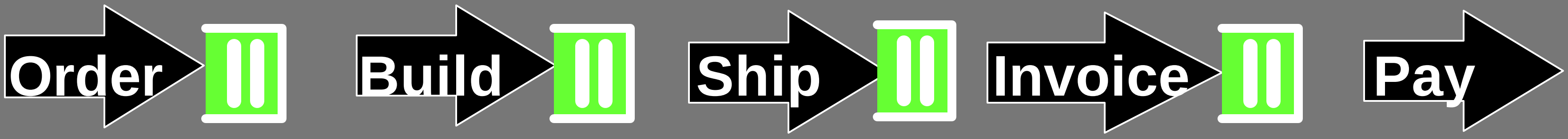
Queued Request/Response

- Time-decouples client and server
  - Three Transactions
- Almost real time, ASAP processing
- Communicate at each other's convenience
  Allows mobile (disconnected) operation
- Disk queues survive client & server failures

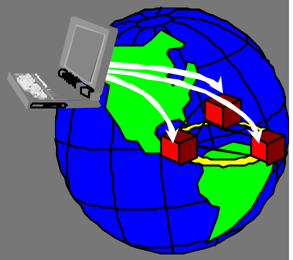


Why Queued Processing?

- Prioritize requests
  - ambulance dispatcher favors high-priority calls
- Manage Workflows



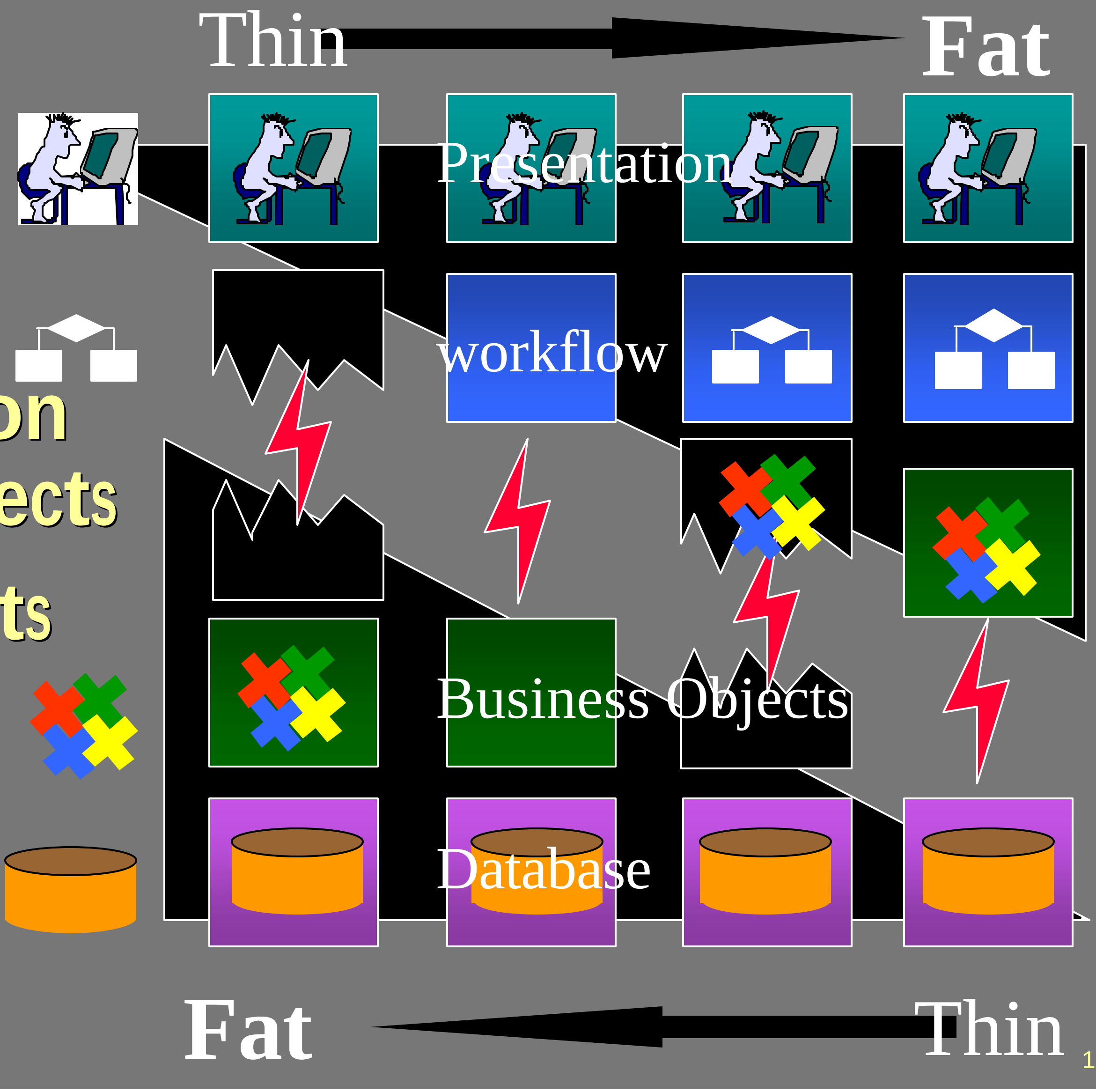
- Deferred processing in mobile apps



- Interface heterogeneous systems
  - EDI,
  - MOM: Message-Oriented-Middleware
  - DAD: Direct Access to Data

Work Distribution Spectrum

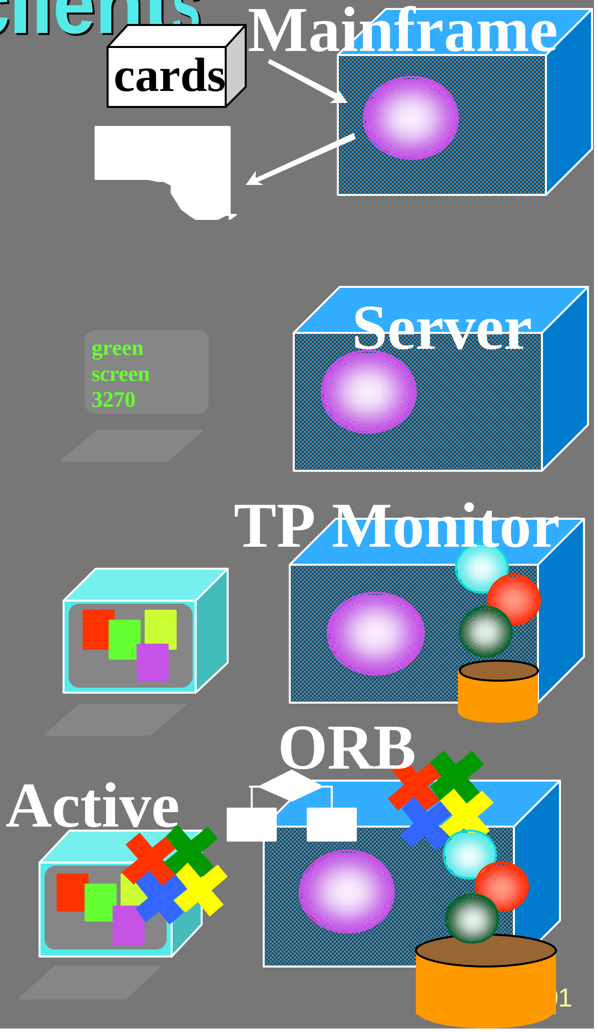
- Presentation and plug-ins
- Workflow manages session & invokes objects
- Business objects
- Database



Transaction Processing Evolution to Three Tier

Intelligence migrated to clients

- Mainframe Batch processing (centralized)
- Dumb terminals & Remote Job Entry
- Intelligent terminals database backends
- Workflow Systems
  Object Request Brokers
  Application Generators



Web Evolution to Three Tier

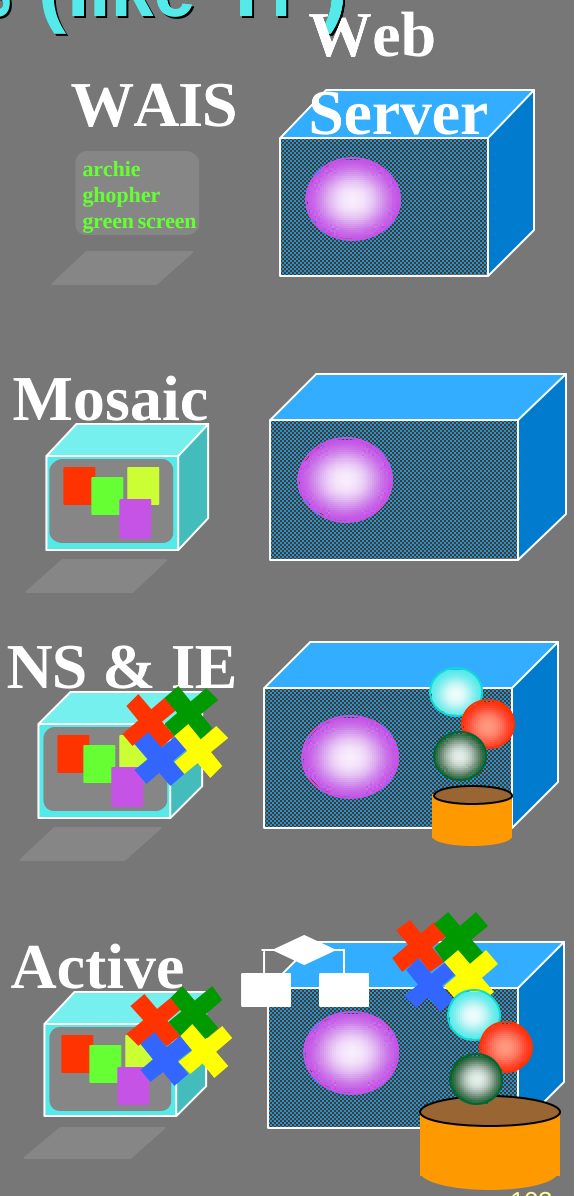
Intelligence migrated to clients (like TP)

- Character-mode clients,
  smart servers

- GUI Browsers - Web file servers

- GUI Plugins - Web dispatchers - CGI

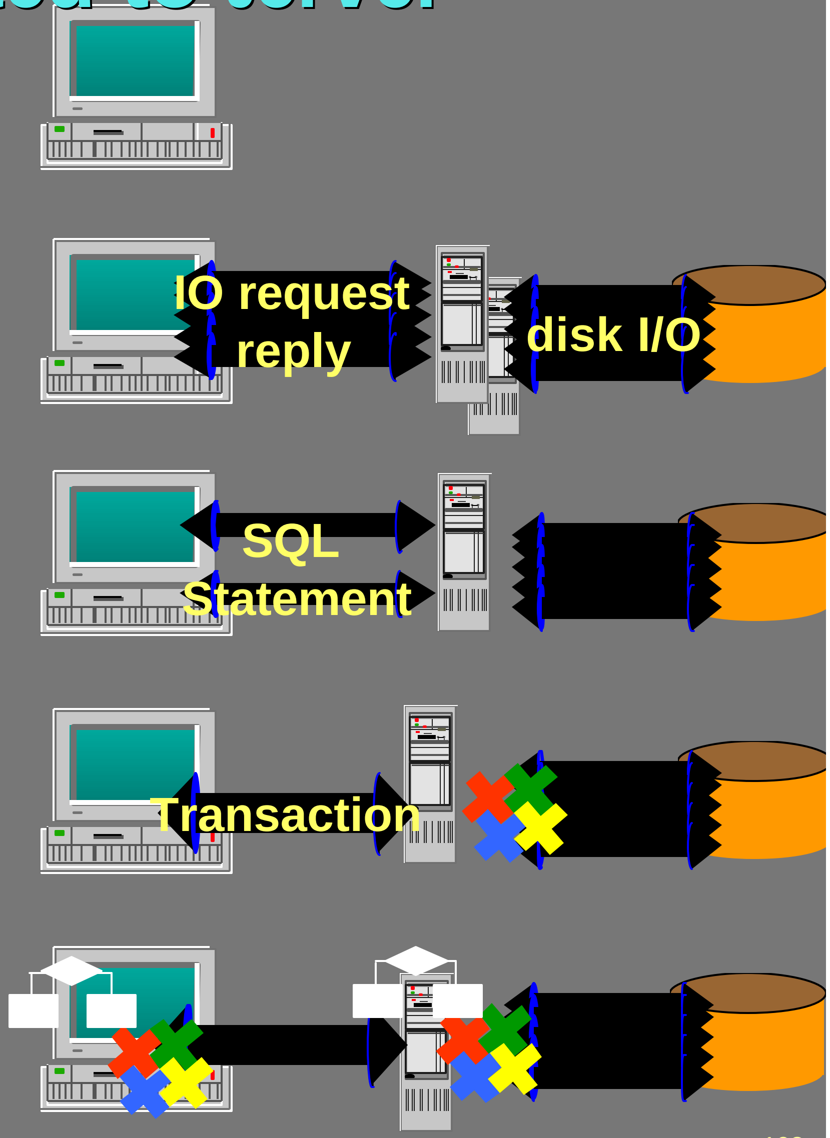
- Smart clients - Web dispatcher (ORB)
  pools of app servers (ISAPI, Viper)
  workflow scripts at client & server



PC Evolution to Three Tier

Intelligence migrated to server

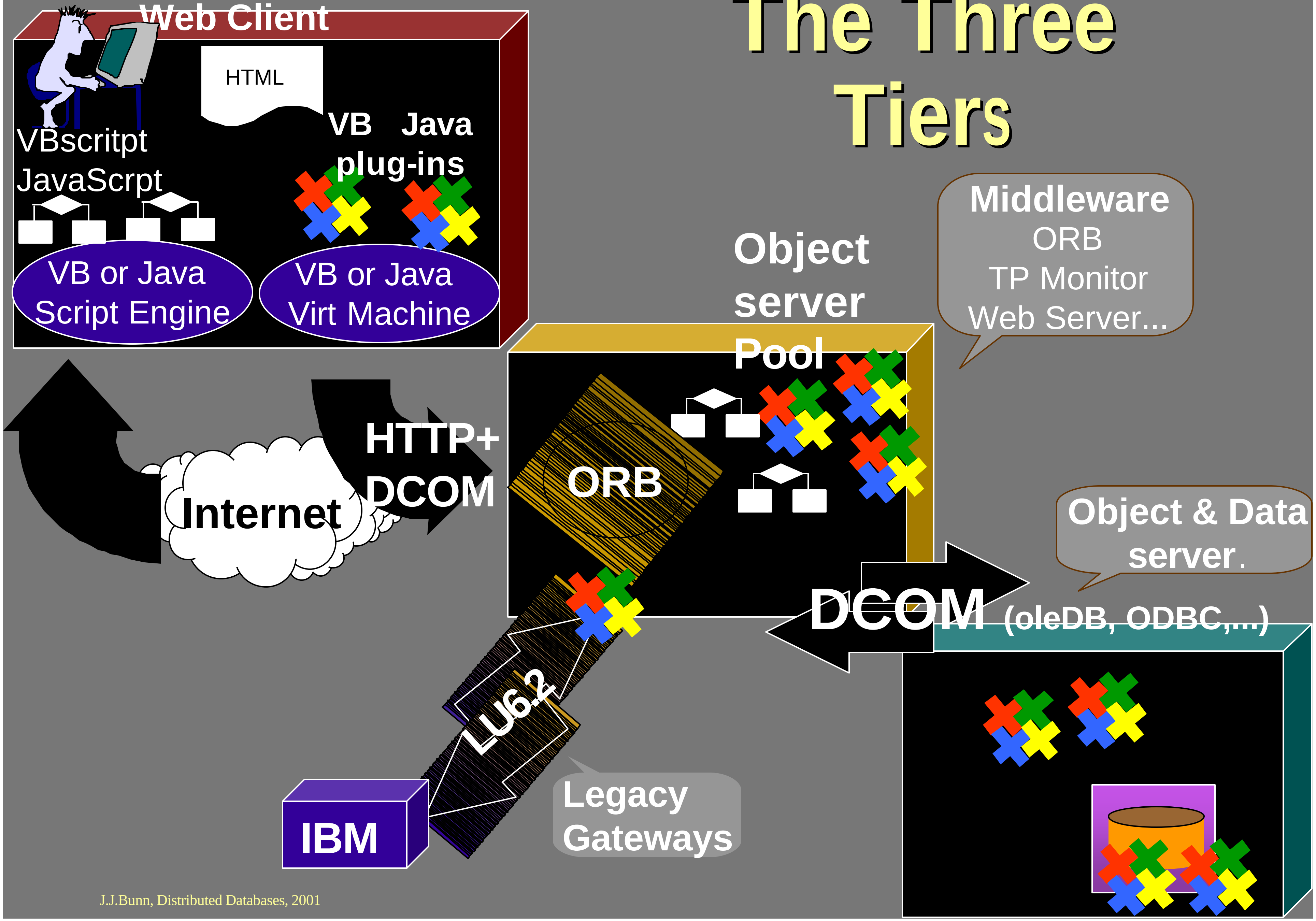
- Stand-alone PC
  (centralized)
- PC + File & print server
  message per I/O
- PC + Database server
  message per SQL statement
- PC + App server
  message per transaction
- ActiveX Client, ORB
  ActiveX server, Xscript



The Pattern:
Three Tier Computing

- Clients do presentation, gather input
- Clients do some workflow (Xscript)
- Clients send high-level requests to ORB (Object Request Broker)
- ORB dispatches workflows and business objects -- proxies for client, orchestrate flows & queues
- Server-side workflow scripts call on distributed business objects to execute task

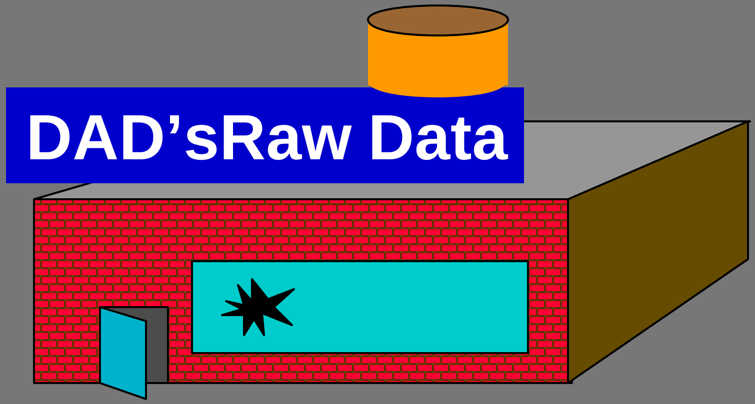
The Three Tiers



Why Did Everyone Go To Three-Tier?

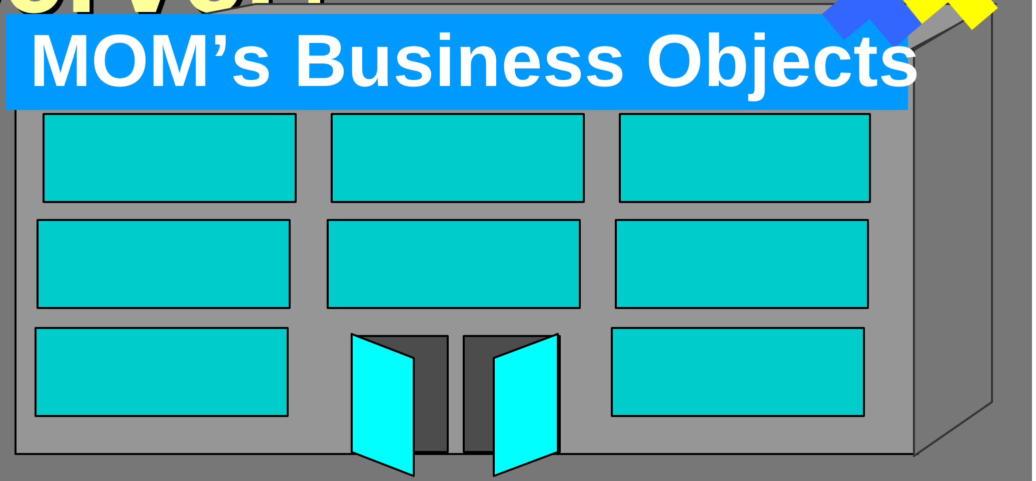
- Manageability
  - Business rules must be with data
  - Middleware operations tools
- Performance (scaleability)
  - Server resources are precious
  - ORB dispatches requests to server pools
- Technology & Physics
  - Put UI processing near user
  - Put shared data processing near shared data

Why Put Business Objects at Server?



Customer comes to store
Takes what he wants
Fills out invoice
Leaves money for goods

Easy to build
No clerks



Customer comes to store with list
- Gives list to clerk
- Clerk gets goods, makes invoice

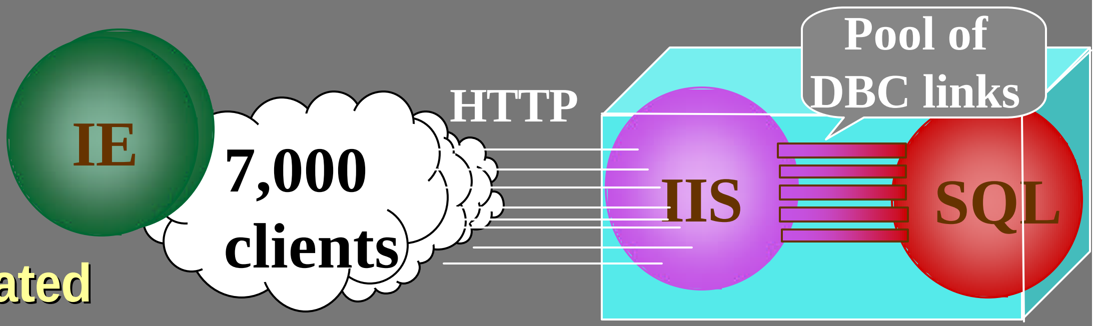
Customer pays clerk, gets goods

Easy to manage
Clerks controls access
Encapsulation

Why Server Pools?

- Server resources are precious.
  Clients have 100x more power than server.
- Pre-allocate everything on server
  - preallocate memory
  - pre-open files
  - pre-allocate threads
  - pre-open and authenticate clients
- Keep high duty-cycle on objects
  (re-use them)
  - Pool threads, not one per client
- Classic example:
  TPC-C benchmark
  - 2 processes
  - everything pre-allocated

N clients x N Servers x F files =
N x N x F file opens!!!



Classic Mistakes

- Thread per terminal
  fix: DB server thread pools
  fix: server pools
- Process per request (CGI)
  fix: ISAPI & NSAPI DLLs
  fix: connection pools
- Many messages per operation
  fix: stored procedures
  fix: server-side objects
- File open per request
  fix: cache hot files

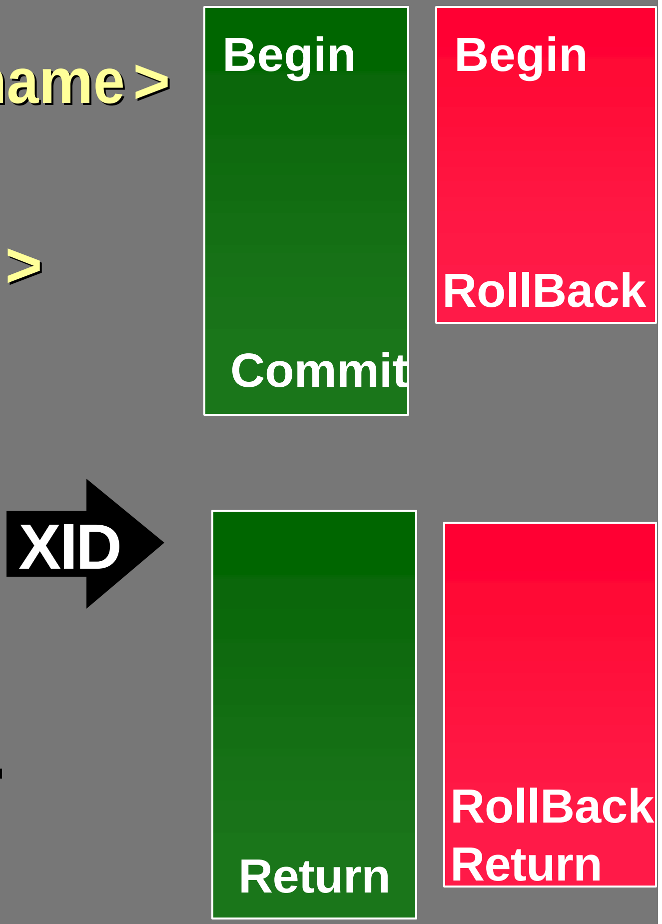
Distributed Applications need Transactions!

- Transactions are key to structuring distributed applications
- ACID properties ease exception handling
  - Atomic: all or nothing
  - Consistent: state transformation
  - Isolated: no concurrency anomalies
  - Durable: committed transaction effects persist

Programming & Transactions

The Application View

- **You Start (e.g. in TransactSQL):**
  - Begin [Distributed] Transaction <name>
  - Perform actions
  - Optional Save Transaction <name>
  - Commit or Rollback
- **You Inherit a XID**
  - Caller passes you a transaction
  - You return or Rollback.
  - You can Begin / Commit sub-trans.
  - You can use save points



Begin

Commit

Begin

RollBack

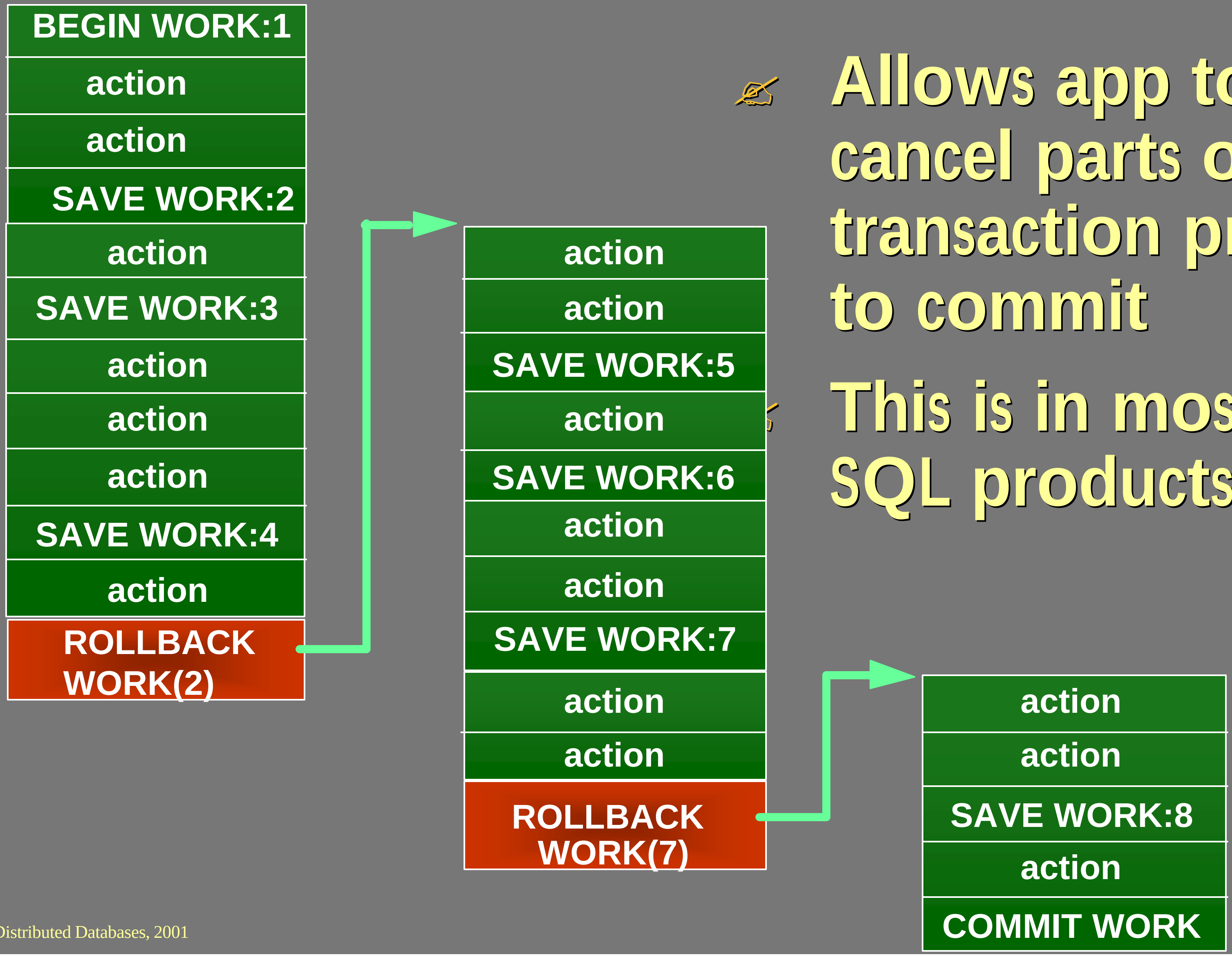
XID

Return

RollBack
Return

Transaction Save Points

Backtracking within a transaction



BEGIN WORK:1
action
action
SAVE WORK:2
action
SAVE WORK:3
action
action
action
SAVE WORK:4
action
ROLLBACK WORK(2)

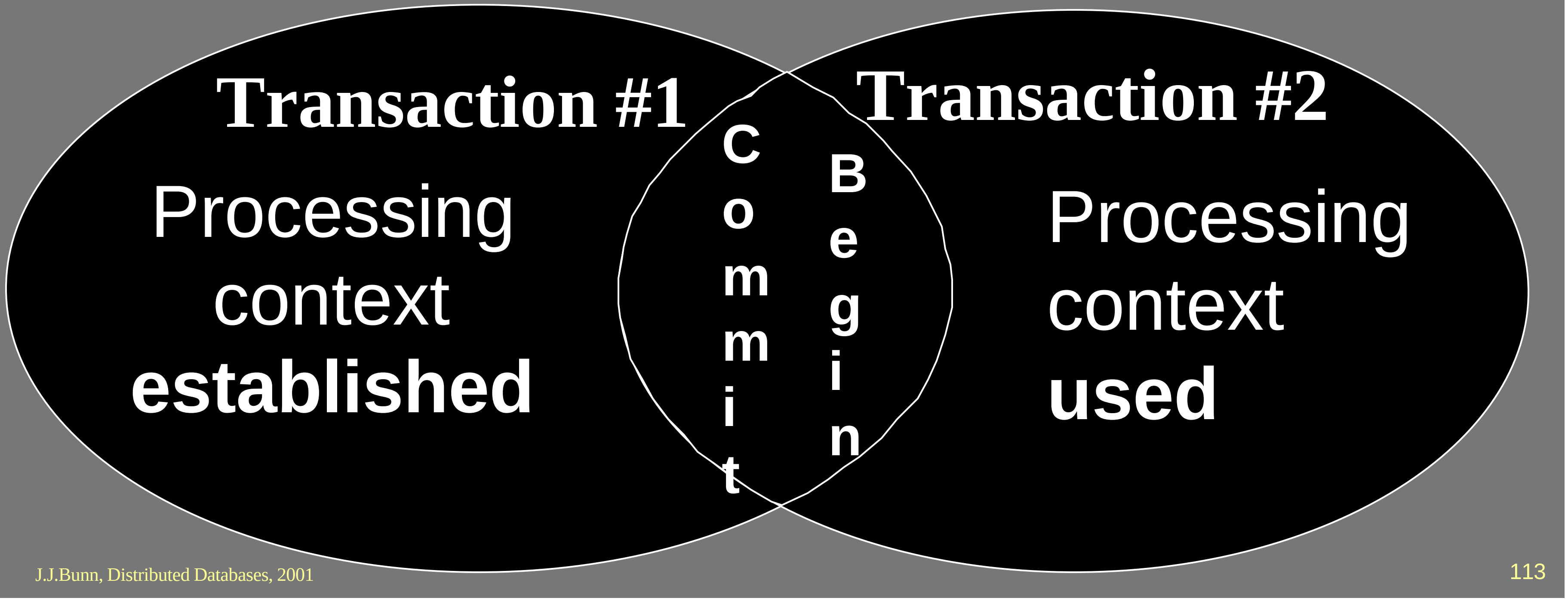
action
action
SAVE WORK:5
action
SAVE WORK:6
action
action
SAVE WORK:7
action
action
ROLLBACK WORK(7)

action
action
SAVE WORK:8
action
COMMIT WORK

- Allows app to cancel parts of a transaction prior to commit
- This is in most SQL products

Chained Transactions

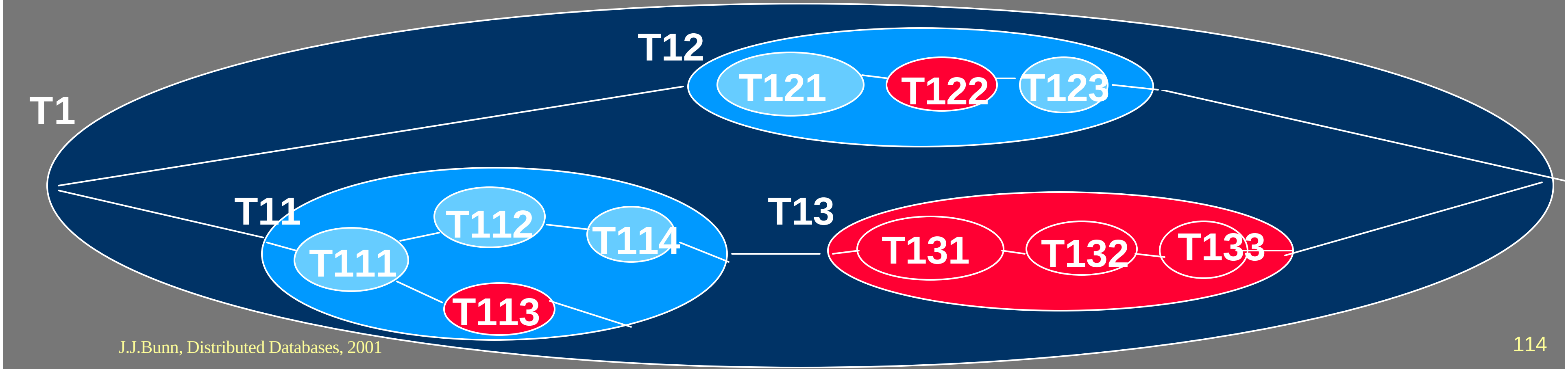
- Commit of T1 implicitly begins T2.
- Carries context forward to next transaction
  - cursors
  - locks
  - other state



Nested Transactions

Going Beyond Flat Transactions

- Need transactions within transactions
- Sub-transactions commit only if root does
- Only root commit is durable.
- Subtransactions may rollback
  if so, all its subtransactions rollback
- Parallel version of nested transactions



Workflow:

A Sequence of Transactions

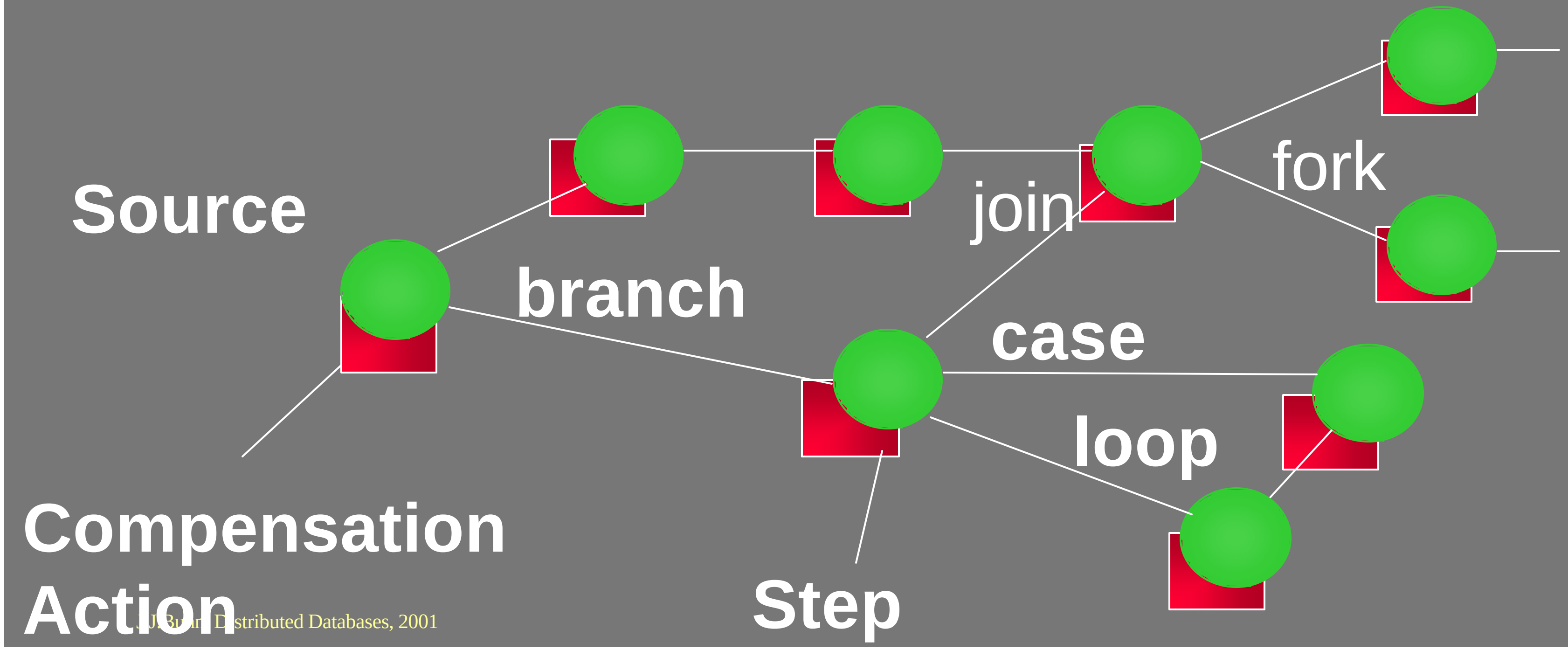
- Application transactions are multi-step
  - order, build, ship & invoice, reconcile
- Each step is an ACID unit
- Workflow is a script describing steps
- Workflow systems
  - Instantiate the scripts
  - Drive the scripts
  - Allow query against scripts
- Examples

  Manufacturing Work In Process (WIP)
  Queued processing
  Loan application & approval,
  Hospital admissions…

Workflow Scripts

- Workflow scripts are programs

  (could use VBScript or JavaScript)
- If step fails, compensation action handles error
- Events, messages, time, other steps cause step.
- Workflow controller drives flows



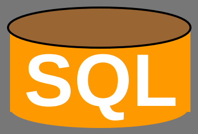
Workflow and ACID

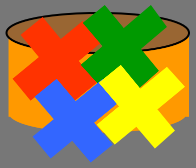
- Workflow is not Atomic or Isolated
- Results of a step visible to all
- Workflow is Consistent and Durable
- Each flow may take hours, weeks, months
- Workflow controller
  - keeps flows moving
  - maintains context (state) for each flow
  - provides a query and operator interface
    e.g.: “what is the status of Job # 72149?”

ACID Objects Using ACID DBs

The easy way to build transactional objects

- **Application uses transactional objects** (objects have ACID properties)
- **If object built on top of ACID objects, then object is ACID.**
  - **Example: New, EnQueue, DeQueue on top of SQL**
- **SQL provides ACID**





Business Object: Customer

Business Object Mgr: CustomerMgr

```
dim c as Customer
dim CM as CustomerMgr
...
set C = CM.get(CustID)
...
C.credit_limit = 1000
...
CM.update(C, CustID)
..
```

Persistent Programming languages automate this.

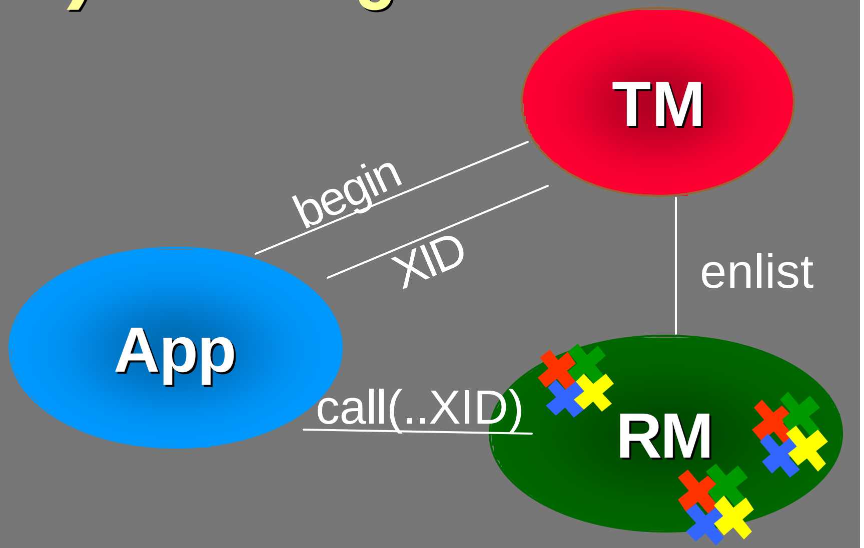
ACID Objects From Bare Metal

The Hard Way to Build Transactional Objects

- Object Class is a <u>Resource Manager (RM)</u>
  - Provides ACID objects from persistent storage
  - Provides Undo (on rollback)
  - Provides Redo (on restart or media failure)
  - Provides Isolation for concurrent ops
- Microsoft SQL Server, IBM DB2, Oracle,… are Resource managers.
- Many more coming.
- RM implementation techniques described later

Transaction Manager

- Transaction Manager (TM): manages transaction objects.
  - XID factory
  - tracks them
  - coordinates them
- App gets XID from TM
- Transactional RPC
  - passes XID on all calls
  - manages XID inheritance
- TM manages commit & rollback



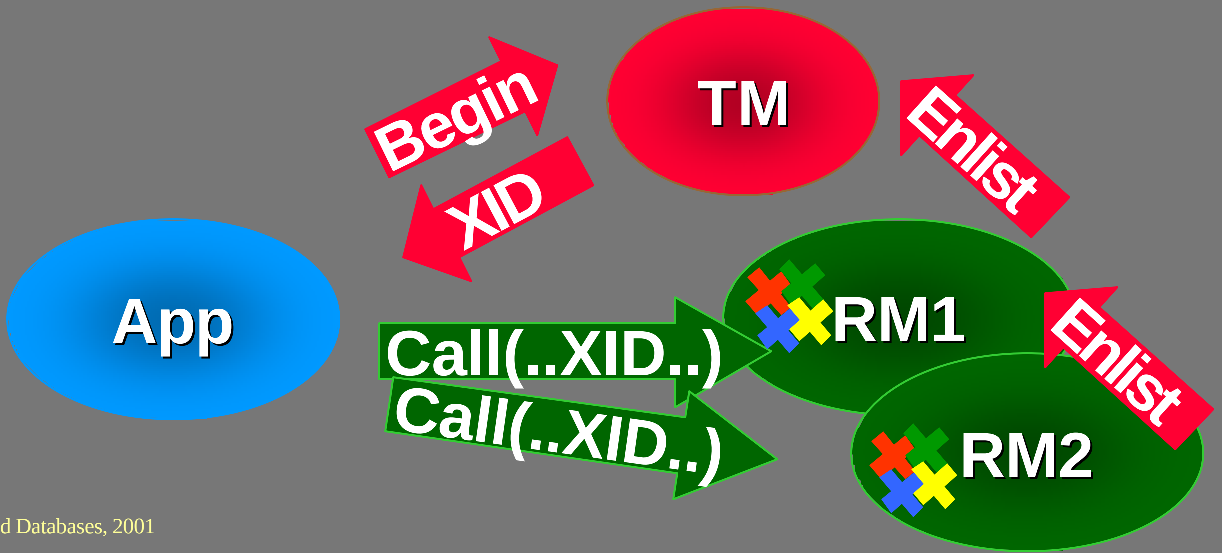
TM Two-Phase Commit

Dealing with multiple RMs

- **If all use one RM, then all or none commit**
- **If multiple RMs, then need coordination**
- **Standard technique:**
  - **Marriage: Do you? I do. I pronounce…Kiss**
  - **Theater: Ready on the set? Ready! Action! Act**
  - **Sailing: Ready about? Ready! Helm's a-lee! Tack**
  - **Contract law: Escrow agent**
- **Two-phase commit:**
  - **1. Voting phase: can you do it?**
  - **2. If all vote yes, then commit phase: do it!**

Two-Phase Commit In Pictures

- Transactions managed by TM
- App gets unique ID (XID) from TM at Begin()
- XID passed on Transactional RPC
- RMs Enlist when first do work on XID


TM

Begin

XID

Enlist

App

RM1

Call(..XID..)

Call(..XID..)

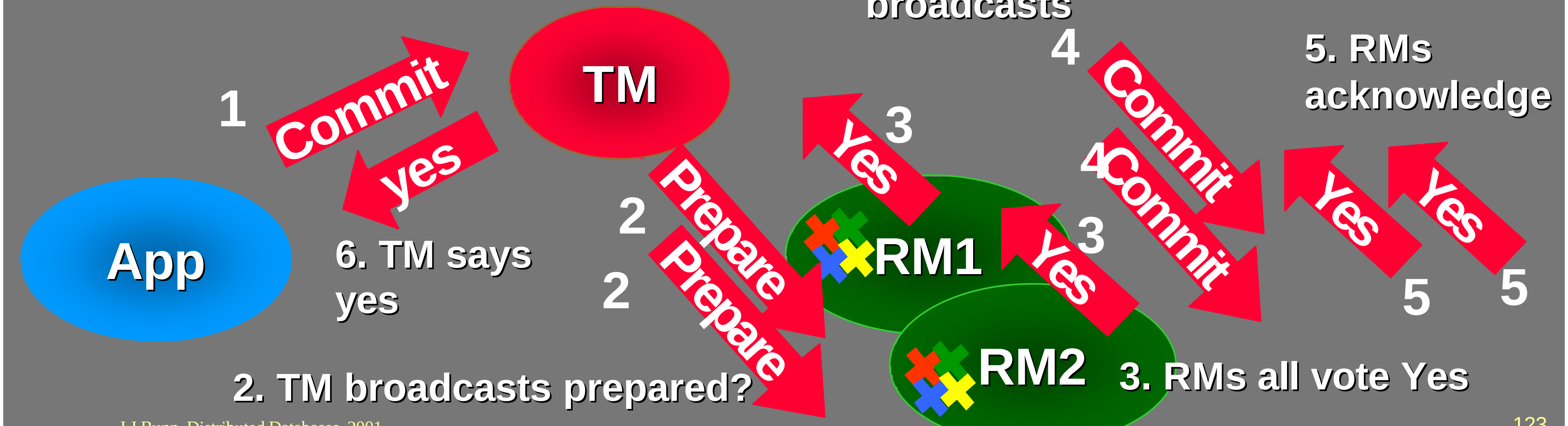
Enlist

RM2

When App Requests Commit

Two Phase Commit in Pictures

- TM tracks all RMs enlisted on an XID
- TM calls enlisted RM’s Prepared() callback
- If all vote yes, TM calls RM’s Commit()
- If any vote no, TM calls RM’s Rollback()



1. Application requests Commit

2. TM broadcasts prepared?

3. RMs all vote Yes

4. TM decides Yes, broadcasts

5. RMs acknowledge

6. TM says yes

Implementing Transactions

- Atomicity
  - The DO/UNDO/REDO protocol
  - Idempotence
  - Two-phase commit
- Durability
  - Durable logs
  - Force at commit
- Isolation
  - Locking or versioning

Part 4

Distributed Databases for Physics

Julian Bunn

California Institute of Technology

Distributed Databases in Physics

- Virtual Observatories (e.g. NVO)
- Gravity Wave Data (e.g. LIGO)
- Particle Physics (e.g. LHC Experiments)

Distributed Particle Physics Data

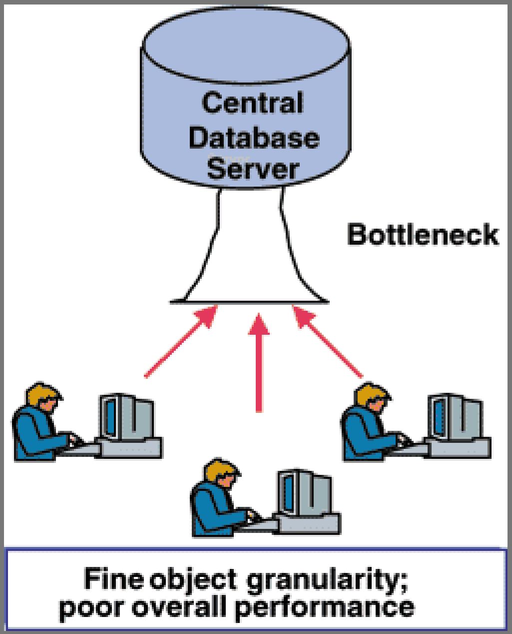
- Next Generation of particle physics experiments are data intensive
  - Acquisition rates of 100 MBytes/second
  - At least One PetaByte (10^{15} Bytes) of raw data per year, per experiment
  - Another PetaByte of reconstructed data
  - More PetaBytes of simulated data
  - Many TeraBytes of MetaData
- To be accessed by ~2000 physicists sitting around the globe

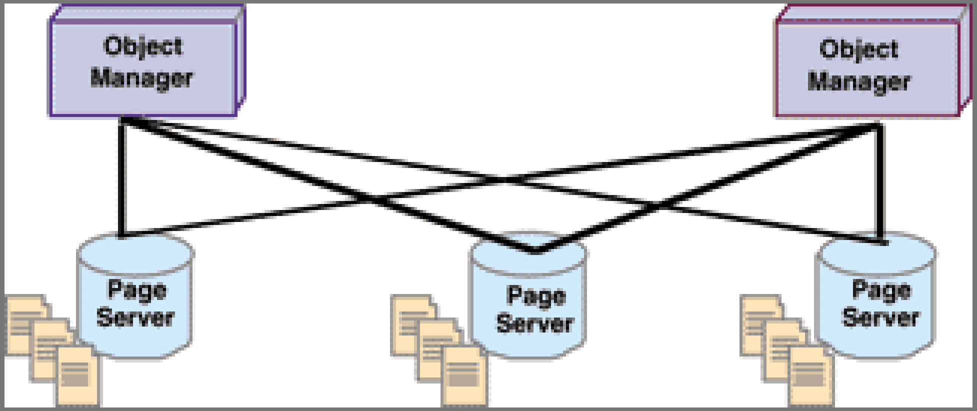
An Ocean of Objects

- Access from anywhere to any object in an Ocean of many PetaBytes of objects
- Approach:
  - Distribute collections of useful objects to where they will be most used
  - Move applications *to* the collection locations
  - Maintain an up-to-date catalogue of collection locations
  - Try to balance the global compute resources with the task load from the global clients

RDBMS vs. Object Database

- Users send requests into the server queue
- all requests must first be serialized through this queue.
- to achieve serialization and avoid conflicts, all requests must go through the server queue.
- Once through the queue, the server may be able to spawn off multiple threads





- DBMS functionality split between the client and server
  - allowing computing resources to be used
  - allowing scalability.
- clients added without slowing down others,
- ODBMS automatically establishes direct, independent, parallel communication paths between clients and servers
- servers added to incrementally increase performance without limit.

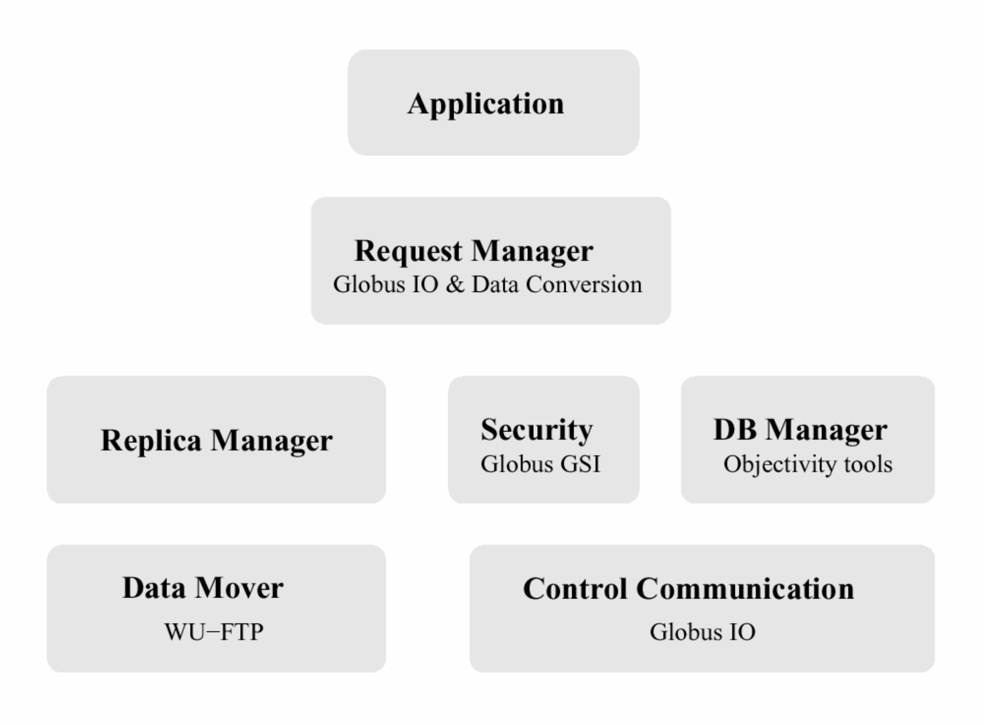
Designing the Distributed Database

- Problem is: how to handle distributed clients and distributed data whilst maximising client task throughput and use of resources
- Distributed Databases for:
  - The physics data
  - The metadata
- Use middleware that is conscious of the global state of the system:
  - Where are the clients?
  - What data are they asking for?
  - Where are the CPU resources?
  - Where are the Storage resources?
  - How does the global system measure up to it workload, in the past, now and in the future?

Distributed Databases for HEP

- Replica synchronisation usually based on small transactions
  - But HEP transactions are large (and long-lived)
- Replication at the Object level desired
  - Objectivity DRO requires dynamic quorum
    - bad for unstable WAN links
  - So too difficult – use file replication
    - E.g. GDMP Subscription method
- Which Replica to Select?
  - Complex decision tree, involving
    - Prevailing WAN and Systems conditions
    - Objects that the Query "touches" and "needs"
    - Where the compute power is
    - Where the replicas are
    - Existence of previously cached datasets

Distributed LHC Databases Today



Application

Request Manager
Globus IO & Data Conversion

Replica Manager

Security
Globus GSI

DB Manager
Objectivity tools

Data Mover
WU–FTP

Control Communication
Globus IO

- ✍ Architecture is loosely coupled, autonomous, Object Databases
- ✍ File-based replication with
- ✍ Globus middleware
- ✍ Efficient WAN transport