DCOM, CORBA, and EJB

1. **DCOM, CORBA, and EJB**
   Generic distributed object systems with object RPC, unique object references, generated stubs, and persistent objects.

2. **DCOM and CORBA**
   Designed to work across languages; generate stubs from an IDL.

3. **CORBA and EJB**
   Designed to work across “containers” or “ORBs” from different vendors, so standardize wire protocol, object referencing, object-container contracts, etc.

4. **DCOM and EJB**
   Designed to support distributed transactional applications, while isolating application developers from the resulting mess.
   Include handling of distributed refcounts and garbage collection.
COM and DCOM

Microsoft’s DCOM extends COM (the basis for OLE) to distributed object applications.

- COM is a binary standard for assembling software components into applications.
  - combine components from different languages and vendors
- Components support multiple interfaces referenced by interface identifiers (IPIDs), which are UUIDs.
  - IPIDs expose the object interface, but not implementation
  - exactly similar to C++ virtual function tables (method vectors)
  - cheap intra-process invocation (like Emerald)
- Extends to network through client-side proxies.
  - Based on OSF-DCE-RPC wire protocol and implementation.
DCOM Issues/Features

1. referencing and type matching
   Object references, interface identifiers, and class identifiers are UUIDs; versions by reassigning UUIDs.
   (Also uses UUIDs as causality IDs and xids on RPC call chains.)

2. automatic reference counting and object state management
   DCOM components inherit from IUnknown, which provides reference-counting primitives called externally.

3. supports scalable/persistent/transactional objects managed by Microsoft Transaction Server (MTS)
   persistent references by monikers
   TP monitor is “viper”, cluster support is “wolfpack”, etc.
   (for this stuff we’ll look at the EJB framework)
Enterprise Java Beans

*Enterprise Java Beans* (EJB) is an evolving distributed object server standard supported by IBM, Sun, and others.

- An EJB is a component class; a *bean instance* is an object.
  (But it is popular today to blur class-instance distinctions.)
- EJBs are implemented in Java and packaged in jars.
  “Write once, run anywhere.”
- Clients create/destroy instances using EJB *home interfaces*.
  Access beans through name services (e.g., JNDI).
- EJB instances are invoked by Java clients using RMI.
  References can be passed freely and garbage collected, but instances are not mobile.
Session Beans vs. Entity Beans

Session beans are transient.

- created and used by a single client
  maintained on that client’s behalf
- client creates instances through the bean’s home interface
  create/destroy interface exported through JNDI

Entity beans are persistent and sharable.

nobody supports them (yet)

- home interface adds a lookup primitive by primary key
- supports load/store from entity state in an external database
- object handles can be serialized and stored persistently
CORBA References

CORBA specs define variable-length object references (IOR).

- **type name** also known as **repository ID**
  
  unique code for the narrowest type of the object

  used to retrieve interface descriptors from an **interface repository**

  service...allows dynamic stub imports, parent classes, etc.

- **transport-specific binding info**
  
  e.g., DNS hostname + port for object server or binding server

  could be more than one record if object is replicated

- **opaque object key**
  
  unique vendor tag + **object adapter** key + object key

  exact format is ORB-specific (vendor-specific)
CORBA: Transient and Persistent Refs

1. *Transient IORs* must become invalid forever after the server fails, to prevent dangling references.
   
   e.g., use a UUID for the object ID

2. *Persistent IORs* name objects whose lifetimes and locations are independent of any particular server.
   
   • The transport binding info names an *implementation repository* service rather than an actual object server.
     
     Logically part of ORB, but need not be on same node as server.

   • Client binds to object by sending a *locate* request for IOR to repository, which returns binding info.

   • Repository auto-starts server if necessary.
Issues for CORBA Object Binding

1. How to make the implementation repository scalable?
   • Repository may switch on adapter ID and ignore object ID. Objects with the same adapter can’t migrate independently.
   • Partition objects (or adapters) among more repository servers.

2. Vendor-specific shortcuts are possible.
   If client uses same ORB, peek in IOR to reuse cached bindings or put a location hint in the object key.

3. Can the binding change after the initial bind phase?
   CORBA objects have unique identity and encapsulate all state.
   Clients using the same “entity” bind to the same CORBA object.
Reliable Distributed Computing with Objects

EJB and DCOM are intended as frameworks for building reliable distributed systems easily.

- Calls on objects are synchronized by implicit monitors, so object authors may ignore concurrency.

- Objects can be transactional.
  
  Client may make a sequence of calls on multiple objects, grouped together into a single transaction with ACID properties.

  Key goal: isolate programmers from details of transactions and synchronization.

- Design for ease-of-use and scalability by viewing objects as temporary vessels for long-term database state.

  Objects load/store their state from databases using JDBC or SQL.
Transactions: ACID Properties

“Full-blown” transactions guarantee four intertwined properties:

- **Atomicity.** Transactions can never “partly commit”; their updates are applied “all or nothing”.
  The system guarantees this using logging, shadowing, distributed commit.

- **Consistency.** Each transaction $T$ transitions the dataset from one semantically consistent state to another.
  The application guarantees this by correctly marking transaction boundaries.

- **Independence/Isolation.** All updates by $T1$ are either entirely visible to $T2$, or are not visible at all.
  Guaranteed through locking or timestamp-based concurrency control.

- **Durability.** Updates made by $T$ are “never” lost once $T$ commits.
  The system guarantees this by writing updates to stable storage.
Transactional Objects: the EJB Model

EJBs are based on a transactional object model, also shared by DCOM and CORBA (with different terminology).

Containers and their Beans

Container supports:
- serializing non-reentrant beans
- object persistence
- passivation and activation
- transaction coordination
- reference management

DB Managers (optional)
Object/Container Interactions for Tx

If objects store/retrieve their essential state in external data managers, the container may cache/recycle/pool them at times when no state is held in the object.

Object->container controls:
- Method $M$ requires a transaction.
- What transaction am I part of?
- Abort this transaction.
- I relinquished my state; I may be pooled.

Container->object calls:
- Activate()/Passivate()
- LoadState()/StoreState()
- AfterBegin($T$)
- AfterCompletion($T$, boolean)
Anatomy of an “Entity Bean”

The “bean” is an object view of an “entity” in a database; its “identity” is defined by a unique primary key.

Client requests indirect through EJB object so container may interpose.

Activate()/Passivate()
Load()/Store()
Remove()
set/unset EntityContext
findByPrimaryKey()
find*( ) finder methods
Key Design Properties for EJBs

1. Client requests to a “bean instance” indirect through a (small) container-affiliated “EJB object”.
   Container invisibly controls the binding of object to instance.

2. At certain times the container may change the entity associated with a given instance.
   Container can reassign the instance’s primary key.

3. “Bean instances” must be capable of storing and (re)loading their entity state at the request of the container.
   Allows container to manage a cache (or “pool”) of instances.
   Container issues store request for two-phase commit protocol.
   Container can synchronize instances bound to the same entity.
Bean Pooling

Client requests to a “bean instance” indirect through a container-affiliated “EJB object”, so the container may:

- Move the instance object, e.g., to another node in a cluster. useful for load balancing
- Select the instance bound to an EJB object on the fly.
  The container must keep idle EJB objects around, but it can passivate and/or reuse the instance for a different entity.
- Recycle instance husks and their database connections. reduce the cost of instantiation
- Create multiple instances for the same entity (primary key).
  Container keeps instance consistent with entity by forcing load/store of the instance’s cached entity state as needed.
“Bean” Instance States

Container can change primary key while in the “pooled” state. Container promises to call `load()` after `activate()`, before calling an app method.

A container may cache loaded instances in the ready state. Could I write a bean to tell if the container does this?

To execute a `find()` method, the container just selects a random pooled instance of the correct class, with no identity.

Container serializes all calls.
Achieving Atomic Durability

Atomic durability dictates that the system schedule its stable writes in a way that guarantees two key properties:

1. Each transaction’s updates are **tentative until commit**.
   
   Database state must not be corrupted with uncommitted updates.
   
   If uncommitted updates can be written to the database, it must be possible to **undo** them if the transaction fails to commit.

2. **Buffered updates are written to stable storage synchronously with commit**.

   **Option 1**: *force* dirty data out to the permanent (*home*) database image at commit time.
   
   **Option 2**: commit by recording updates in a *log* on stable storage, and defer writes of modified data to home (**no-force**).
Logging (Journaling) Updates

Logging appends updates to a sequential file in temporal order.

- Log typically supplements a home image optimized for reads.
- The log localizes updates that must be done synchronously.
  well-suited to rotational devices with high seek times
  drawback: some updates are written twice (once to log, once to home)
- A home checkpoint scheme prevents unbounded log growth.
  But updates to home must be deferred until the log is forced.

Key idea: group each set of related updates into a single log record that can be written to disk atomically.

- Each log record is trailed by a marker (e.g., checksum) that says “this log record is complete”.
- To recover, scan the log and reapply logged updates to home.
Committing Distributed Transactions

Transactions may touch data stored at more than one site.

Each site commits (i.e., logs) its updates independently.

*Problem:* any site may fail while a commit is in progress, but after updates have been logged at another site.

An action could “partly commit”, violating atomicity.

Basic problem: individual sites cannot unilaterally choose to abort without notifying other sites.

“Log locally, commit globally.”
Two-Phase Commit (2PC)

**Solution**: all participating sites must agree on whether or not each action has committed.

- **Phase 1.** The sites vote on whether or not to commit.
  
  *precommit*: Each site prepares to commit by logging its updates before voting “yes”.

- **Phase 2.** Commit iff all sites voted to commit.
  
  A central transaction *coordinator* gathers the votes.

  If any site votes “no”, the transaction is aborted.

  Else, coordinator writes the *commit* record to its log.

  Coordinator notifies participants of the outcome.

*Note*: one server ==> no 2PC is needed, even with multiple clients.
The 2PC Protocol

1. $Tx$ requests commit, by notifying coordinator ($C$)
   $C$ must know the list of participating sites.

2. Coordinator $C$ requests each participant ($P$) to prepare.

3. Participants validate, prepare, and vote.
   Each $P$ validates the request, logs validates updates locally, and
   responds to $C$ with its vote to commit or abort.

   If $P$ votes to commit, $Tx$ is said to be “prepared” at $P$.

4. Coordinator commits.
   Iff $P$ votes are unanimous to commit, $C$ writes a commit record to its
   log, and reports “success” for commit request. Else abort.

5. Coordinator notifies participants.
   $C$ asynchronously notifies each $P$ of the outcome for $Tx$.

   Each $P$ logs the outcome locally and releases any resources held for $Tx$. 
Handling Failures in 2PC

How to ensure consensus if a site fails during the 2PC protocol?

1. A participant $P$ fails before preparing.
   
   Either $P$ recovers and votes to abort, or $C$ times out and aborts.

2. Each $P$ votes to commit, but $C$ fails before committing.
   
   Participants wait until $C$ recovers and notifies them of the decision to abort. The outcome is uncertain until $C$ recovers.

3. $P$ or $C$ fails during phase 2, after the outcome is determined.
   
   Carry out the decision by reinitiating the protocol on recovery.
   
   Again, if $C$ fails, the outcome is uncertain until $C$ recovers.