Distributed Objects: A Lightning Tour
What is an “object”? 

*Objects* are units of data with the following properties:

- **typed and self-contained**
  
  Each object is an instance of a *type* that defines a set of *methods* (signatures) that can be invoked to operate on the object.

- **encapsulated**
  
  The only way to operate on an object is through its methods; the internal representation/implementation is hidden from view.

- **dynamically allocated/destroyed**
  
  Objects are created as needed and destroyed when no longer needed, i.e., they exist outside of any program scope.

- **uniquely referenced**
  
  Each object is uniquely identified during its existence by a name/OID/reference/pointer that can be held/passed/stored/shared.
Why are objects useful for systems?

The properties of objects make them useful as a basis for defining persistence, protection, and distribution.

- **Objects are self-contained and independent.**
  
  Objects are a useful granularity for persistence, caching, location, replication, and/or access control.

- **Objects are self-describing.**
  
  Object methods are dynamically bound, so programs can import and operate on objects found in shared or persistent storage.

- **Objects are abstract and encapsulated.**
  
  It is easy to control object access by verifying that all clients invoke the object’s methods through a legal reference.

  Invocation is syntactically and semantically independent of an object’s location or implementation.
1. Extend the object name space outside of a process and across a distributed system.
   - Linked data structures can be partitioned across the nodes and traversed with *location-independent invocation*.
     Emerald, Guide

2. Extend the object name space across secondary storage.
   - Objects (and their references) may live longer than processes; fault objects into memory as they are referenced.
     POMS and other persistent object stores and OODBs
   - Eliminate "*impedance mismatch*” between memory/disk.
     type-checked secondary storage with type evolution
3. Define RPC services as objects.
   - Allows persistent, location-independent name space with
dynamic binding and/or dynamic activation.
     Argus, Eden, Clouds, Arjuna
   - Encapsulate with a clean object wrapper for external access.

4. Make object references unforgeable and reject invocation
   attempts with invalid references.
   - An unforgeable object reference is called a capability.
     Cambridge CAP, IBM System/38 and AS/400, Intel 432
     CMU Hydra and Mach, Stanford V, Amoeba, Eden
   - Use as a basis for protected sharing/interaction/extension.
Emerald

Emerald is a classic and influential distributed object system.

- Distribution is fully integrated into the language, its implementation, and even its type model.
  This is a strength and a weakness: combines language issues and system issues that should be separated.

- Objects can be freely moved around the network
  Programmers see a uniform view of local and remote objects.
  Moving objects “take their code and threads with them”.

- Local invocation is fast; remote invocation is transparent.
  supports pass-by-reference for RPC
Understanding Emerald

1. Emerald was marketed to OS researchers as a lightweight alternative to *process migration* (a hot topic at the time).

   Process migration was accepted as a means to balance load, handle failures, or initiate a remote activity.

2. Emerald eliminated key problems with process migration.

   OS-dependent state associated with migrating processes
   high cost of interaction among colocated processes

3. Emerald was seen as a sort of lightweight “operating system” as well as a language.

   The “kernel” is a runtime library in a Unix process (one per node) within which all Emerald programs run.
   The Emerald “kernel” had its own support for “processes”, which we would now call “threads”, and execution...protection...persistence.
**Issues for Emerald**

1. How to implement object references so that they are location-independent?
   - How to ensure uniqueness of object IDs?
   - How to locate remote objects, e.g., if they have moved?

2. What is the “hook” for transparent location-independent invocation?
   - How to make it fast if the invoked object is local?

3. How to migrate and dynamically import code and threads?

4. What are the semantics of argument passing?

5. Who’s going to implement distributed garbage collection?
Uniform Mobility: an Example

**Step 1:** a thread invokes a purple object on node A, which recursively invokes a blue object on the same node.

**Step 2:** the blue object moves to node B concurrently with the invocation.

How to preserve inter-object pointers across migration?
How to keep threads “sticky” with migrating objects?
How to maintain references in stack activation records?
How to maintain linkages among activation records?
What about virtual addresses in CPU registers?
Object References in Emerald

Emerald represents inter-object references as pointers into an object descriptor in an object table hashed by a unique object identifier (OID).

The object table has a descriptor for every resident object, and for every remote object referenced by a resident object, and then some.

When an object moves, its containing references must be found (using its template) and updated to point to descriptors on the destination node.

References to the moving object need not be updated because they indirect through the object table.
Uniform Mobility Example, Continued

Node A

Step 3: the purple object moves to node C before the invocation returns.

Node B

What to do with the thread’s activation record for the purple object?
- cost of context switch

How to find the purple object to return into its activation record?

How to keep forwarding pointers up to date? (eager vs. lazy)
- iterative lookup
- piggyback on passed references and remote returns

Node C
The Relevance of Emerald

Emerald defines a conceptual basis for understanding today’s distributed object systems.

CORBA, RMI, EJB, DCOM

Emerald showed what is possible from a distributed object environment in its purest form.

1. Uniform view of local/remote objects: orthogonality of location.
   referencing, invocation/return
   garbage collection

2. Uniform object model is compatible with (local) performance.
   extended features impose a cost only when used

3. Location of mobile objects by reference hints and forwarding.
Distributed Objects in the Real World (I)

The purity of Emerald flows from a common language, architecture, and security domain.

1. Can we use distributed objects as a basis for interoperability among software modules written in different languages?

   IDL converts distributed objects into a packaging/integration technology.
   What about type checking? Garbage collection?

2. Can objects interact across systems with different data formats?

   *IOP and C/XDR define standard wire formats for transmitted data.

3. Can objects interact securely across mutually distrusting nodes and/or object infrastructures by different vendors?

   How are object references stored, transmitted, and validated?
Distributed Objects in the Real World (II)

Emerald has no provision for handling failures of any kind.

How can we find objects in the presence of node failures?

What should we do about activities that were pending in failed nodes/objects?

How can we recover object state after failures? How can we ensure that the recovered state is consistent?

Can we safely execute object invocations from nodes with intermittent connectivity?

What about long-term storage of objects, and invocation of stored objects that are not currently active?

persistence/uniqueness/stability of object IDs
Distributed Objects in the Marketplace

1. Remote Method Invocation (RMI)
   API and architecture for distributed Java objects

2. Microsoft *Component Object Model* (COM/DCOM)
   binary standard for distributed objects for Windows platforms
   e.g., clients generated with Visual Basic, servers in C++
   extends OSF DCE standard for RPC

3. CORBA (Common Object Request Broker Architecture)
   OMG consortium formed in 1989
   multi-vendor, multi-language, multi-platform standard

4. Enterprise Java Beans (EJB) [1998]
   CORBA-compliant distributed objects for Java, built using RMI

5. Web services and SOAP
RMI and Network Objects

Our goal now is to look at some current distributed object systems.

We start with systems that preserve the single-language model of Emerald, with uniform garbage collection:

- RMI for Java
- Network Objects for Modula-3

We then move on to more general and full-featured cross-language and cross-platform schemes.

- CORBA, DCOM, EJB
Remote objects are referenced through "proxy" or "surrogate" objects, which "masquerade" as the actual remote object.

[SOs system, Marc Shapiro, The Proxy Principle (1986)]

Proxy objects are type-equivalent with their remote objects, but their methods are marshaling stubs.

Skeletons/guards may perform access checks as well as marshaling and method dispatch.

Proxy/stub objects can encapsulate caching, replication, or other aspects of distribution that are best kept hidden from the client (also cf. subcontracts [Hamilton et. al., SOSP 93]).
Remote Method Invocation (RMI)

RMI is “RPC in Java”, supporting Emerald-like distributed object references, invocation, and garbage collection, derived from SRC Modula-3 network objects [SOSP 93].

```
1: Naming.bind(URL, obj1)
2: stub1 = Naming.lookup(URL)
3: stub2 = stub1->method()
```

The registry provides a bootstrap naming service using URLs.

```
rm://slowww.server.edu/object1
```
Some RMI Classes

In Modula-3 network objects, the stub type and implementation type are both subtypes of an abstract interface type T.

Java achieves type compatibility using interfaces.

java.rmi.server.*

A stub class implements the same set of Remote interfaces as its corresponding server class.
Subcontracts

Subcontracts allow complex distribution behaviors hidden behind the proxy/stub.

[Hamilton et al, Sun Spring project, SOSP 93]

Subcontract Hooks
marshal
unmarshal
invoke
marshal-copy

called by stub when corresponding event occurs

Examples
replica
reconnectable
cacheable

UnicastRemoteObject
unicast to a single server instance
references are valid only while server process is alive

RemoteServer
YourSubcontract
UnicastRemoteObject
YourClassHere

It is clear that RMI intends to support the subcontract model, but it is not clear (to me) to what degree it succeeds.
Arguments to RMI calls are passed using *object serialization*. Argument classes must implement *Serializable*.

- Local objects are passed by copy/value (*marshaling*).
  - no coherency
  - no static members
  - no handles to state in the VM (e.g., open files)

  What about threads? AWT components?
  - Classes must be loadable by client in the usual way.

- *RemoteObjects* are passed by reference.
  - Stub/skeleton classes loaded (e.g., from server) by *RMIClassLoader*. 
Distributed Garbage Collection

RMI uses a distributed garbage collection scheme based on the SRC network objects collector.

Garbage Collection Protocol, version 1.0

client
1. When creating a new stub, send `object->dirty()` invocation to server.
2. When destroying a stub, send `object->clean()` invocation to server.

server
1. On `object->dirty()`, increment object’s external reference count.
2. On `object->clean()`, decrement object’s external reference count.
3. Reclaim object when:
   - no local references remain
   - AND
   - external reference count is zero.
Garbage Collection: Complications

0. Cycles
1. What if a client fails without releasing object references?
   We can detect a broken connection and decrement counts, but we must associate counts with unique clientIDs.

2. What if an object is reclaimed prematurely due to a transient network failure that heals?
   must guarantee that the server detects the dangling reference
   requires unique objectIDs

3. What if dirty and clean messages from a given client are delivered out of order?
   tag messages with increasing sequence-numbers

4. What about races if a last reference passes from one client to another?
   for RPC, only a problem for returns
Reliable Garbage Collection: *Client*

Garbage Collection Protocol, version 2.0

1. When creating a stub, send `object->dirty()`. Always await acknowledgement for `dirty` message before acknowledging receipt of the reference.

2. When destroying a stub, send `object->clean()`. Never destroy a stub until all transmitted references have been acknowledged by their recipients.

3. Resend `object->dirty()` for each referenced stub every *lease interval*.

4. Tag each garbage collection message with:
   (i) a strictly increasing `sequence-number`
   (ii) a `clientID` guaranteed unique across all clients.
Reliable Garbage Collection: Server

Garbage Collection Protocol, version 2.0

1. On `object->dirty()`, add `clientID` to object’s `referenced-set`.
   - `referenced-set` record shows `(clientID, dirty-time, sequence#)
   - `dirty-time` is the server’s time when it received the `dirty` message
   - `sequence#` is the client’s `sequence-number` recorded in the `dirty` message

2. On `object->clean()`, remove `clientID` from object’s `referenced-set`.
   - discard `clean` messages bearing `sequence-number < sequence#` in record

3. Periodically scan all `(object, clientID)` pairs in referenced sets.
   - if `dirty-time` is older than `lease interval`
     - remove `clientID` from `referenced-set`

4. Reclaim object when `referenced-set == {}` and no local references exist

Would this protocol work for Emerald?
Some GC Points for Java/RMI

• Local garbage collector has a hook to upcall RMI layer when a RemoteObject is reclaimed.

• The server RMI layer holds “weak” references to exported remote objects.
  
  In 1.1, weak refs collect iff the JVM “really needs the memory”.
  
  ...thus a client cannot force a server to fail by acquiring references.

• The registry is included in the referenced-set for registered objects.
  
  Unreferenced objects exist as long as they are named.

• So many messages....

• What about unique identifiers?
  
  RMI depends on unique client ID, unique object ID
**Digression: Unique Identifiers (UUIDs)**

DCE, CORBA and DCOM use common approaches to generating unique identifiers.

UUID/GUID scheme has origins in OSF DCE interface IDs.
standardized through IETF [Paul Leach]

**Goals:**
- unique in space and time, with extremely high probability
- UUID assignments without centralized authority
  (but relies on uniquely assigned node numbers)
- support very high assignment rates
- easily manageable 128-bit quantities
  (with 7 bits of type/variant)
Time-Based UUIDs

The standard *time-based UUID* has the following fields:

- **48-bit unique *node identifier***
  IEEE 802 node number, or randomly generate (w/ high bit)

- **60-bit UTC *time value* with 100-nanosecond precision**
  allows 10M UUID creations per-node per-second
  stall if UUIDs requested at too high a rate
  note the “Year 3400 Problem”

- **13 bit *clock sequence number***
  randomize to start
  increment or randomize if clock may have been set back
  e.g., if system changes node number (e.g., due to NIC switch)
RMI Unique IDs

1. **ObjIDs** assigned as unique within a server VM.
   - unique object number (64-bit)
   - UID for address space
   - \((\text{InetAddress, ObjID})\) pair is equivalent to a UUID.

2. **UIDs** uniquely identify an address space (VM) on a host.
   - process ID (32-bit)
   - timestamp (64-bit): one second resolution
   - clock sequence (16-bit)

3. **VMIDs** are globally unique virtual machine identifiers.
   - InetAddress
   - UID
DCOM Reference Counting

DCOM uses a similar “pinging protocol” for reference-counting and garbage-collecting distributed objects

- ping per \((client, server)\) pair instead of per \((client, object)\) pair
  - client runtime aggregates objects from the same server
  - client sends server a list of objects held in each ping interval

- \textit{delta pinging} reduces the size of ping messages
  - client sends just a list of references cleaned or dirtied
  - server remembers client’s reference list: don’t resend it

- ping periods are dynamically negotiable
  - performance and intermittent connectivity

- server objects ultimately control their own lifetimes
Type Matching

How can we guarantee type matching for remote interfaces and serialized objects?

- **Modula-3**: types must be linked into program in advance. Stubs installed independently on client and server use unique type *fingerprints* to find/check matching local types using *narrowest surrogate rule* (for references) each type and each supertype carries a separate fingerprint

- **Java**: stubs and classes may be dynamically imported. Classes have string names, with location specified by: URL encoded in marshal stream server *codebase* for stubs etc.

\[ \text{RMIClassLoader} \]
Some Other Aspects of Object Models

1. Objects may be *active* or *passive*.

   An *active* object contains its own thread(s); typically incoming invocations are queued and serviced by these threads.

   *Passive* objects sit there and wait to be invoked; the invoking thread enters the object for the duration of the call.

2. An object’s mapping to the underlying OS or machine features is often expressed in terms of *granularity*.

   A *coarse-grained* object is equivalent to a process or address space invoked with messages or cross-domain calls.

   A *medium-grained* object lives with others within a process and is protected by its addressing wrapper.

   A *fine-grained* object is a heap-allocated block of memory.
The Trouble with Objects

Why were these OO systems seen to have failed by the U.S. systems research community?

• Many sacrificed performance for elegance.
  “Performance is paramount” is (was?) an accepted axiom.

• Many depended on (slow and/or obscure) OO languages at a time when C was dominant in systems.
  OO concepts had not yet penetrated the culture.

• Those that were not integrated with OO languages could not benefit fully from the elegance of the model.
  nonuniform view of “system objects” and “language objects”

• Few adherents were able to communicate the relevance of OO systems to real application needs.