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.

[Source: SOS system, Marc Shapiro, The Proxy Principle (1986)]

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

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

Per-process object tables hash stubs and skeletons by external OID (passed on the wire).

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].

RMI registry

The registry provides a bootstrap naming service using URLs.

*rmi://slowww.server.edu/object1*

1: Naming.bind(URL, obj1)

2: stub1 = Naming.lookup(URL)

3: stub2 = stub1->method()
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

Examples
- replica
- reconnectable
- cacheable

RemoteServer
UnicastRemoteObject
YourSubcontract
YourClassHere

called by stub when corresponding event occurs

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

It is clear that RMI intends to support the subcontract model, but it is not clear (to me) to what degree it succeeds.
RMI Parameters and Serialization

Arguments to RMI calls are passed using *object serialization*. Argument classes must implement *Serializable*.

- Local objects are passed by copy/value (*marshalling*).
  - 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.
  currently being standardized by 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
   - (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.

\[
\text{RMIClassLoader}
\]