Chapter 10: Distributed Object-Based Systems

Version: December 2, 2009
Remote distributed objects

- Data and operations **encapsulated** in an object
- Operations implemented as **methods** grouped into **interfaces**
- Object offers only its **interface** to clients
- **Object server** is responsible for a collection of objects
- **Client stub (proxy)** implements interface
- **Server skeleton** handles (un)marshaling and object invocation
Remote distributed objects

Types of objects I

- **Compile-time objects**: Language-level objects, from which proxy and skeletons are automatically generated.
- **Runtime objects**: Can be implemented in any language, but require use of an object adapter that makes the implementation appear as an object.

Types of objects II

- **Transient objects**: live only by virtue of a server: if the server exits, so will the object.
- **Persistent objects**: live independently from a server: if a server exits, the object’s state and code remain (passively) on disk.
Example: Enterprise Java Beans (EJB)

What is it
Java object hosted by special server that allows for different means of calling the object by remote clients.
Types of EJBs

Four different types

- **Stateless session bean**: Transient object, called once, does its work and is done. **Example**: execute an SQL query and return result to caller.
- **Stateful session bean**: Transient object, but maintains client-related state until the end of a session. **Example**: shopping cart.
- **Entity bean**: Persistent, stateful object, can be invoked during different sessions. **Example**: object maintaining client info on last number of sessions.
- **Message-driven bean**: Reactive objects, often triggered by message types. Used to implement publish/subscribe forms of communication.
Globe distributed objects

Observation
Most distributed objects are not distributed at all: state is kept at a single node. **Alternative:** Globe objects, which are physically distributed across multiple machines.
Globe distributed objects

**Note**

To make DSOs generic, we need to separate function from distribution support

Same interface as implemented by semantics subobject
Globe distributed objects

Note
Replication subobject essentially decides how and when the local semantics subobject will be invoked.
Processes: Object servers

**Servant**
The actual implementation of an object, sometimes containing only method implementations:
- Collection of C or COBOL functions, that act on structs, records, database tables, etc.
- Java or C++ classes

**Skeleton**
Server-side stub for handling network I/O:
- Unmarshalls incoming requests, and calls the appropriate servant code
- Marshalls results and sends reply message
- Generated from interface specifications
Processes: Object servers

Object adapter

The “manager” of a set of objects:

- Inspects (as first) incoming requests
- Ensures referenced object is activated (requires identification of servant)
- Passes request to appropriate skeleton, following specific activation policy
- Responsible for generating object references
Observation
Object servers determine how their objects are constructed
Example: Ice

```cpp
main(int argc, char* argv[]) {
    Ice::Communicator ic;
    Ice::ObjectAdapter adapter;
    Ice::Object object;
    ic = Ice::initialize(argc, argv);

    adapter = ic->createObjectAdapterWithEndpoints
        ( "MyAdapter","tcp -p 10000" );
    object = new MyObject;
    adapter->add(object, objectID);
    adapter->activate();

    ic->waitForShutdown();
}
```

**Note**

Activation policies can be changed by modifying the *properties* attribute of an adapter. Ice aims at *simplicity*, and achieves this partly by putting policies into the middleware.
Client-to-object binding

Object reference
Having an object reference allows a client to **bind** to an object:
- Reference denotes server, object, and communication protocol
- Client loads associated stub code
- Stub is instantiated and initialized for specific object

Two ways of binding
- **Implicit**: Invoke methods directly on the referenced object
- **Explicit**: Client must first explicitly bind to object before invoking it
Client-to-object binding: implicit/explicit

Distr_object* obj_ref;
obj_ref = ...;
obj_ref->do_something();

Distr_object* obj_ref;
Local_object* obj_ptr;
obj_ref = ...;
obj_ptr = bind(obj_ref);
obj_ptr->do_something();

Some remarks
- Reference may contain a URL pointing to an implementation file
- (Server,object) pair is enough to locate target object
- We need only a standard protocol for loading and instantiating code

Observation
Remote-object references allow us to pass references as parameters. This was difficult with ordinary RPCs.
Remote Method Invocation (RMI)

Basics
(Assume client stub and server skeleton are in place)

- Client invokes method at stub
- Stub marshals request and sends it to server
- Server ensures referenced object is active:
  - Create separate process to hold object
  - Load the object into server process
  - ...

- Request is unmarshaled by object’s skeleton, and referenced method is invoked
- If request contained an object reference, invocation is applied recursively (i.e., server acts as client)
- Result is marshaled and passed back to client
- Client stub unmarshals reply and passes result to client application
RMI: Parameter passing

Object reference
Much easier than in the case of RPC:
- Server can simply bind to referenced object, and invoke methods
- Unbind when referenced object is no longer needed
RMI: Parameter passing

Object-by-value
A client may also pass a complete object as parameter value:

- An object has to be marshaled:
  - Marshall its state
  - Marshall its methods, or give a reference to where an implementation can be found

- Server unmarshals object. Note that we have now created a copy of the original object.
- Object-by-value passing tends to introduce nasty problems
RMI: Parameter passing

Note

Systemwide object reference generally contains server address, port to which adapter listens, and local object ID. Extra: Information on protocol between client and server (TCP, UDP, SOAP, etc.)
RMI: Parameter passing

Question

What’s an alternative implementation for a remote-object reference?
Object-based messaging

1. Call by the application

Client application

Client proxy

Client RTS

2. Request to server

3. Response from server

4. Call by the RTS

Callback interface

Polling interface

2. Request to server

3. Response from server

4. Call by the application
Observation

In order to invoke remote objects, we need a means to uniquely refer to them. **Example:** CORBA object references.
Object references

Observation
It is not important how object references are implemented per object-based system, as long as there is a standard to exchange them between systems.

Solution
Object references passed from one RTS to another are transformed by the bridge through which they pass (different transformation schemes can be implemented)
Object references

Object system A  Object system B
Object server Interoperable references (Half) gateway

Observation
Passing an object reference $refA$ from RTS A to RTS B circumventing the A-to-B bridge may be useless if RTS B doesn’t understand $refA$
Globe object references: location independent

Stacked address
Stack of addresses representing the protocol to speak:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protocol ID</td>
<td>Constant representing a (known) protocol</td>
</tr>
<tr>
<td>Protocol addr.</td>
<td>Protocol-specific address</td>
</tr>
<tr>
<td>Impl. handle</td>
<td>Reference to a file in a repository</td>
</tr>
</tbody>
</table>

Instance address
Contains all that is needed to talk in a proprietary way to an object:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Impl. handle</td>
<td>Reference to a file in a repository</td>
</tr>
<tr>
<td>Initialization string</td>
<td>Used to initialize an implementation</td>
</tr>
</tbody>
</table>
Synchronization

Issue

Client stubs hide many of the interactions with objects, which may not always be a good idea.
Other problem

Should we have server-side or client-side synchronization:

- **Server**: Problems when client crashes
- **Clients**: Problems when clients at different machines need to synchronize

Java

Clients at same machine are automatically synchronized; those from different machines are not.
Observation

Objects form a natural means for realizing entry consistency:

- Data are grouped into units, and protected by a synchronization variable (i.e., lock)
- Synchronization variables adhere to sequential consistency (i.e., values are set atomically)
- Operations of grouped data can be nicely grouped: object

Problem

What happens when objects are replicated? One way or the other we need to ensure that operations on replicated objects are properly ordered.
Replicated objects

Problem
We need to make sure that requests are ordered correctly at the servers and that threads are deterministically scheduled.
Replicated objects

**Observation**

We are dealing with nasty issues here. Simplicity may dictate completely serialized (i.e., single-threaded) executions at the server.
Active replication

Updates are forwarded to multiple replicas, where they are carried out. There are some problems to deal with in the face of replicated invocations.
Replicated invocations

**Solution**

Assign a coordinator on each side (client and server), which ensures that only one invocation, and one reply is sent.
Fault tolerance

Observation
Nothing really new to report – object systems handle fault tolerance as you would expect them to do.

Example
CORBA uses special interoperable object group references to allow a client to access replicas:
CORBA FT: Architecture

- Replication manager
- Object group manager
- Property manager
- Middleware/RTS
  - Interceptor
  - Replication
  - Reliable multicasting

Logging & Recovery

- Log

To other replicas

Client or object server
Security

When objects can (also) be moved/copied around

We need to consider three different security aspects:

- **Secure binding**: is client ↔ object binding authorized?
- **Secure invocation**: is client authorized to invoke a method?
- **Platform security**: is object protected against underlying platform and vice versa?

Example

For Globe we have the following public/private key pairs:

- **Object key**: Owner of the private key can set access rights for users and replica servers.
- **Replica key**: Used to make sure that a replica server is hosting an object.
- **User key**: Associated with every user.
### Globe security

<table>
<thead>
<tr>
<th>User certificate</th>
<th>Replica certificate</th>
<th>Administrative certificate</th>
</tr>
</thead>
<tbody>
<tr>
<td>( K^+_\text{Alice} )</td>
<td>( K^+_\text{Repl} )</td>
<td>( K^+_\text{Adm} )</td>
</tr>
<tr>
<td>( U: 0010011100 )</td>
<td>( R: 1100011100 )</td>
<td>( R: 1101111100 )</td>
</tr>
<tr>
<td>( \text{sig}(O, {U, K^+_\text{Alice}}) )</td>
<td>( \text{sig}(O, {R, K^+_\text{Repl}}) )</td>
<td>( \text{sig}(O, {R, U, D, K^+_\text{Adm}}) )</td>
</tr>
</tbody>
</table>

- **User certificate**: \( U[i] = 1 \) iff user is allowed to invoke method \( M_i \).
- **Replica certificate**: \( R[i] = 1 \) iff replica is allowed to execute \( M_i \).
- **Administrative certificate**: \( U[i] = 1 \) iff a user certificate for \( M_i \) may be issued. Same for \( R[i] \).
Secure method invocation

1: Application issues an invocation request.
2: Control subobject checks user permissions; security object should have valid user certificate.
3: Request is marshaled and passed on.
4: Replication subobject requests middleware to set up secure channel to suitable replica.
5: Security object initiates a replica lookup.
6: Security subobject sets up secure channel. The replica must prove it is allowed to carry out the requested invocation.
Secure method invocation

7: Request is passed on to comm. subobject.
8: Comm. subobject encrypts and signs the request.
9: Request is decrypted and authenticated.
10: Request is passed to server-side repl. subobject.
11: Authorization takes place; user certificate from client-side stub is passed to replica to verify that request can be carried out.

12: Request is unmarshaled.

13: Operation can be executed.