Object Request Brokers

- invoke methods at remote objects (objects that run in another JVM)
- example ORBs: RMI, JavaIDL

Components of an ORB

- Communication Layer: transfers data between hosts
- RPC Layer: defines method invocation semantics and marshalling
- Object Store: lifecycle management of remote objects
- Stub and Skeleton Generator: create code for stubs and skeletons
- Nameserver: find object by name

Stubs and Skeletons

- Stub: Proxy for remote object
- Skeleton: Invokes methods at remote object depending on the method ID

1. Stub
2. Skeleton
3. Remote Object
4. RPC and Communication Layers

Stub

- similar to a proxy for the remote object; implements the remote interface
- pack method call into a request packet:
  - object ID, method ID, method parameters
- uses the RPC layer to send request
- transform returned object into appropriate type and return
- example generated method code (without exception handling)

```java
public int deposit(int param0) {
    Object[] parameters = new Object[1];
    parameters[0] = new Integer(param0);
    Request req = new Request(ctx, oid, 9, parameters);
    Object ret = req.send();
    return ((Integer)ret).intValue();
}
```
40.1 Stub

- example generated stub class

```java
package test;
import orb.*;
public class AccountImpl_Stub implements test.Account {
    int oid;
    ClientContext ctx;
    public AccountImpl_Stub(int oid, ClientContext ctx) {
        this.oid = oid;
        this.ctx = ctx;
    }
    // generated methods
    // ...
}
```

- ClientContext should contain information that is common to all stub objects

40.1.2 Skeleton

- invoke method at "real" object, given
  - object reference
  - method ID
  - parameters

40.1.3 Skeleton example

```java
package test;
import orb.*;
public class AccountImpl_Skel implements Skeleton {
    AccountImpl object;
    public AccountImpl_Skel() {}    
    public void init(int oid, Object obj) {
        this.oid = oid;
        this.object = (AccountImpl) obj;
    }
    public Object invoke(int mid, Object[] parameters) throws Exception {
        switch(mid) {
            ...
            case 9: {
                return new Integer(object.deposit(((Integer)parameters[0]).intValue()));
            }
        }
    }
}
```

40.2 Communication Layer

- data transfer between hosts
- use DatagramSocket (UDP)
### 40.3 RPC Layer
- handles forwarding of method invocations
- is used by the stubs/skeletons
- uses the communication layer to ship bytes

#### 40.3.1 RPC
- simplest RPC is a primitive request/reply protocol

#### 40.3.2 Request
- handles forwarding of method invocations
- is used by the stubs/skeletons
- uses the communication layer to ship bytes
- structure of the RPC layer:

![Diagram of RPC layer]

- simplest RPC is a primitive request/reply protocol
- structure of the RPC layer:

#### 40.3.3 Reply
- handles forwarding of method invocations
- is used by the stubs/skeletons
- uses the communication layer to ship bytes
- structure of the RPC layer:
40.3.4 Marshalling

- RPC layer transfers parameters in request packet and return value in reply packet == marshalling
- copy all parameters to the server (better would be: send reference if parameter implements a remote interface)
- use ObjectStreams/ByteArrayStreams/Datagrams to transfer objects

```java
ByteArrayOutputStream stream = new ByteArrayOutputStream();
ObjectOutputStream out = new ObjectOutputStream(stream);
out.writeObject(...);
byte[] buf = stream.toByteArray();
DatagramPacket packet = new DatagramPacket(buf, ...);
```

40.3.5 Failure Handling

- implements a certain invocation semantics
  - exactly once
  - at least once
  - at most once
  - last of many
  - ...
- depends on the service quality of the communication layer
- cope with communication failures:
  - lost packet
  - non-FIFO packet order
  - duplicated packet
  - modified packet

40.3.6 Failure Handling

- Communication failures may lead to the following problems
  - loss of request message

40.3.7 Failure Handling

- Communication failures may lead to the following problems
  - loss of reply message

Diagram: Request message sent from client to server, but lost during transmission.
**40.3.8 Failure Handling**

- unsuccessful execution of the request

**40.3.9 Failure Handling**

- approximation of exactly once:
  - to cope with lost request messages resend packet after timeout
  - to avoid duplicated invocations when resending use request ID that is the same for the original request and all retransmissions
  - the server uses a reply buffer in case replies get lost (to avoid an additional execution of the request); the reply buffer can be implemented using a Hashtable that maps from request IDs to Reply objects
  - buffered replies can be deleted when the server receives the next request from this client thread
  - coping with server crashes is a bit more complicated ... so we ignore it

**40.4 Object Store**

- Interface:
  - `int registerObject(Object obj)`: returns unique object ID
  - `Object lookupByID(int oid)`: find object given an object ID

- Implementation technique:
  - use `java.util.Hashtable` to remember mapping between ID and object reference (hint: first create an `Integer` wrapper for the id)

**40.5 Nameserver**

- split into an ID finder and a proxy creator
- ID finder: map names to OIDs
  - `void bind(String name, int oid)`
  - `int lookupID(String name)`
- use `Hashtable`
- implement as remote object with OID 1
- a realistic nameserver would return references to stub objects instead of OIDs
- this requires that the class name of the stub is returned together with the OID
- proxy creator: find ID and class name; create proxy object
  - `Object lookup(String name)`
- the Nameserver loads the class and creates an instance of the class
That’s it!

Now have fun with CORBA!