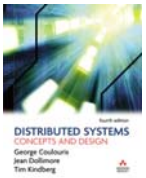
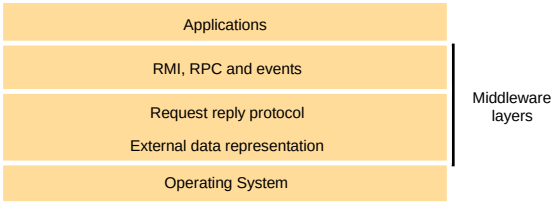
Distributed objects and remote invocation



From Coulouris, Dollimore and Kindberg
Distributed Systems:
Concepts and Design
Edition 4, © Addison-Wesley 2005

Middleware layers

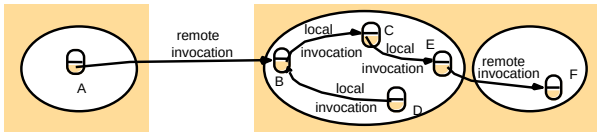


Object Communication (1):

- Distributed objects
  - state: values of its instances variables
  - actions: accessed only by its own methods

Object communication (2):

- Local: within the same process
- Remote: difference processes (could be on different machines)

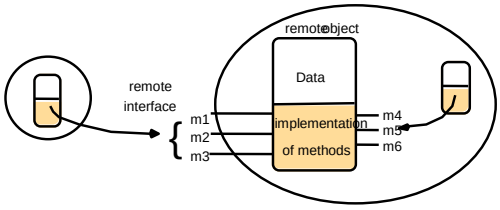


Object communication (3):

- Remote object reference
  - accessing the remote object
  - identifier throughout a distributed system
  - can be passed as arguments

Object communication (4):

- Remote interface
  - specifying which methods can be invoked remotely
  - name, arguments, return type
  - IDL

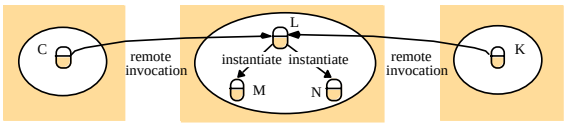


Object communication (5): CORBA IDL example

```
// In file Person.idl
struct Person {
        string name;
        string place;
        long year;
} ;
interface PersonList {
        readonly attribute string listname;
        void addPerson(in Person p) ;
        void getPerson(in string name, out Person p);
        long number();
};
```


Object communication (6):

- Actions
  - Needs remote object reference
  - Calling of methods of objects in another process/host
  - Remote objects might have methods for instantiation (hence remote instantiation)



Object communication (7):

- Garbage collection
  - local garbage collector
  - additional module to coordinate
- Exceptions
  - unexpected events or errors
  - more and different exceptions from local methods

RMI Design Issues (1): Invocation Semantics

- Handling errors
  - retry request?
  - duplicate filtering?
  - retransmission of results?
- Semantics
  - Maybe
  - At least once
  - At most once

RMI Design Issues (2): Invocation semantics

Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

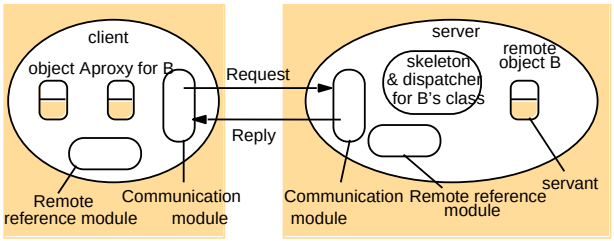
RMI Design Issues (3): Transparency

- like a local call
  - marshalling/unmarshalling
  - locating remote objects
  - accessing/syntax
- latency
- more likely to fail
- errors/exceptions: failure of the network? server? hard to tell
- consistency on the remote machine:
  - Argus: incomplete transactions, abort, restore states [as if the call was never made]

RMI Design Issues (4): Transparency

- syntax might need to be different to handle different local vs remote errors/exceptions (e.g. Argus)
- affects IDL design
- current consensus
  - syntax is transparent
  - different interfaces (e.g., Java: implement Remote interface, RemoteExceptions)

RMI Implementation (1):



RMI Implementation (2):

- Communication module
  - request-reply: message type, requestID, remote object reference
  - implements specific invocation semantics
  - selects the dispatcher, passes on local reference from remote reference module, return request

RMI Implementation (3):

- Remote reference module
  - translating between local and remote object references
  - remote object table
    - remote objects held by the process (B on server)
    - local proxy (B on client)
  - remote object (first time): add to the table, create proxy

RMI Implementation (4):

- RMI software
  - Proxy: behaves like a local object, but represents the remote object
  - Dispatcher: look at the methodID and call the corresponding method in the skeleton
  - Skeleton: implements the method
- Generation of proxies, dispatchers and skeletons
  - IDL (RMI) compiler
- Dynamic invocation
  - Proxies are static—interface complied into client code
  - Dynamic—interface available during run time
    - Generic invocation; more info in "Interface Repository" (COBRA)
    - Dynamic loading of classes (Java RMI)

RMI Implementation (5):

- Server initialization
  - Server creates the first object for remote access
  - Usually clients are not allowed to create servers
- Binder: locating service/object by name
  - Table mapping for names and remote object references
- Server threads
  - concurrency

RMI Implementation (6):

- activation of remote objects
  - many server objects, all running?
  - active and passive status
    - active: available for invocation in a running process
    - passive: not running, state is stored on disk
  - Activation
    - create an active object from a passive object
    - register the new active object
  - Java RMI—objects can be *activatable*
  - similar to inetd

RMI Implementation (7):

- persistent object stores
  - stored in marshaled from on disk for retrieval
  - saved those that were modified
  - persistent or not:
    - persistent root: any descendent objects (reachable from the root) are persistent (eg. persistent Java, PerDiS)
    - certain classes are declared persistent (eg. Arjuna system)

RMI Implementation (8):

- Object location
  - ip address, port #, …
  - location service for migratable objects
    - map remote object references to their probable current locations (Clouds and Emerald systems)
    - Cache/broadcast scheme (similar to ARP)
      - Cache locations
      - If not in cache, broadcast to find it
    - Improvement: forwarding (similar to mobile IP)

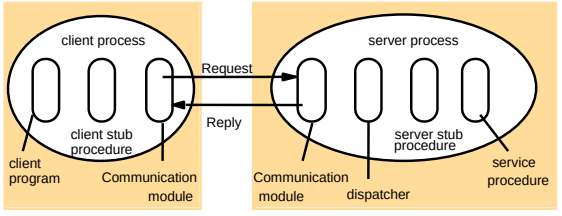
RMI Implementation (9): Distributed garbage collection

- Reference count
- Java's approach
  - the server of an object (B) keeps track of proxies
  - addRef(B) is called when a proxy is created for a remote object
  - addRef(B) tells the server to add an entry
  - when the local host's garbage collector removes the proxy
  - removeRef(B) tells the server to remove the entry
  - when no entries for object B, the object on server is deallocated
- Race condition
  - removeRef(B) from client X
  - addRef(B) from client Y
- Communication failures
  - addRef() didn't return successfully
  - removeRef() will be issued
- Client process failures
  - leases from server
  - renew before expiration
  - entry removed if not renewed before expiration

Remote Procedure Call (1):

- at-least-once or at-most-once semantics
- client: "stub" instead of "proxy" (same function, different names)
  - local call, marshal arguments, communicate the request
- server:
  - dispatcher
  - "stub": unmarshall arguments, communicate the results back

Remote Procedure Call (2)



Sun RPC (1):

- Client-server comm in the SUN NFS (network file system)
- Also called ONC (Open Network Computing) RPC
- In other unix OS as well
- UDP or TCP
- Interface Definition Language (IDL)
  - initially XDR is for data representation, extended to be IDL
  - less modern than CORBA IDL and Java
    - program numbers instead of interface names
    - procedure numbers instead of procedure names
    - single input parameter (structs)
- rpcgen: compiler for XDR
  - client stub
  - server main procedure, dispatcher, and server stub
  - XDR marshalling, unmarshaling

Sun RPC (2): Sun XDR

```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {
    int length;
    char buffer[MAX];
};
struct writeargs {
    FileIdentifier f;
    FilePointer position;
    Data data;
};
```

```
struct readargs {
    FileIdentifier f;
    FilePointer position;
    Length length;
};

program FILEREADWRITE {
  version VERSION {
    void WRITE(writeargs)=1;    1
    Data READ(readargs)=2;      2
  }=2;
} = 9999;
```


Sun RPC (3):

- binding (registry)
  - local binder--portmapper
  - server registers its program/version/port numbers with portmapper
  - client contacts the portmapper at a fixed port with program/version numbers to get the server port
  - different instances of the same service can be run on different computers--different ports
- authentication
  - request and reply have additional fields
  - unix style (uid, gid), shared key for signing, Kerberos

Java RMI (1): Remote interface `ShapeList`

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
   int getVersion() throws RemoteException;
   GraphicalObject  getAllState() throws RemoteException;    1
}
public interface ShapeList extends Remote {
   Shape newShape(GraphicalObject g) throws RemoteException; 2
   Vector allShapes() throws RemoteException;
   int getVersion() throws RemoteException;
}
```


Java RMI (2): `ShapeListServant` implements interface `ShapeList`

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
     private Vector theList;               // contains the list of Shapes    1
     private int version;
     public ShapeListServant()throws RemoteException{...}
     public Shape newShape(GraphicalObject g) throws RemoteException {        2
        version++;
        Shape s = new ShapeServant(g, version);                             3
        theList.addElement(s);
        return s;
     }
     public Vector allShapes()throws RemoteException{...}
     public int getVersion() throws RemoteException { ... }
}
```


Java RMI (3): Server main method

```
import java.rmi.*;
public class ShapeListServer{
   public static void main(String args[]){
      System.setSecurityManager(new RMISecurityManager());
       try{
          ShapeList aShapeList = new ShapeListServant();          1
          Naming.rebind("Shape List", aShapeList );               2
          System.out.println("ShapeList server ready");
       }catch(Exception e) {
          System.out.println("ShapeList server main " + e.getMessage());}
   }
}
```

Java RMI (4): client of *ShapeList*

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
   public static void main(String args[]){
   System.setSecurityManager(new RMISecurityManager());
   ShapeList aShapeList = null;
   try{
      aShapeList  = (ShapeList) Naming.lookup("//bruno.ShapeList") ;       1
      Vector sList = aShapeList.allShapes();                               2
   } catch(RemoteException e) {System.out.println(e.getMessage());
   }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
    }
}
```


Java RMI (5): Java RMIregistry (port 1099)

void rebind (String name, Remote obj)
This method is used by a server to register the identifier of a remote object by name, as shown in Figure 15.13, line 3.

void bind (String name, Remote obj)
This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

void unbind (String name, Remote obj)
This method removes a binding.

Remote lookup(String name)
This method is used by clients to look up a remote object by name, as shown in Figure 15.15 line 1. A remote object reference is returned.

String [] list()
This method returns an array of Strings containing the names bound in the registry.

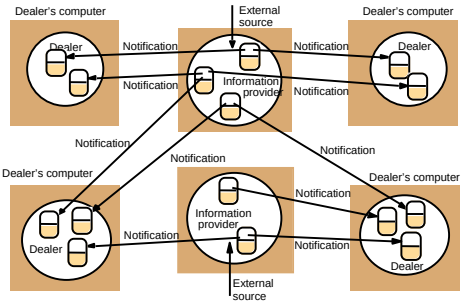
Java RMI (6):

- Callbacks
  - server notifying the clients of events
    - Initiated by server [opposite to initiated by client]
  - why?
    - polling from clients increases overhead on server
    - not up-to-date for clients to inform users
  - how
    - remote object (callback object) on client for server to call
    - client tells the server about the callback object, server put the client on a list
    - server call methods on the callback object when events occur
  - client might forget to remove itself from the list
    - lease--client expire

Events and notifications (1):

- events of changes/updates...
- notifications of events to parties interested in the events
- publish events to send
- subscribe events to receive
- main characteristics in distributed event-based systems:
  - a way to standardize communication in heterogeneous systems (not designed to communicate directly)
  - asynchronous communication (no need for a publisher to wait for each subscriber--subscribers come and go)
- event types
  - each type has attributes (information in it)
  - subscription filtering: focus on certain values in the attributes (e.g. "buy" events, but only "buy car" events)

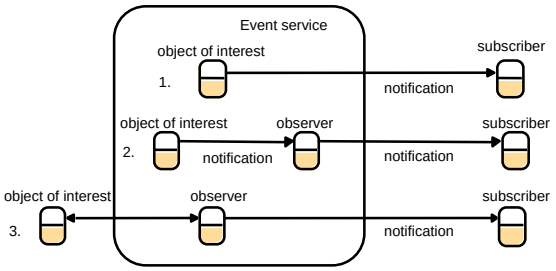
Events and Notifications (2): [subscribers: dealers in stock exchange]



Events and notifications (3):

- Distributed event notification
  - decouple publishers from subscribers via an event service (manager)
- Architecture:
  - object of interest (usually changes in states, e.g. temperature, price)
  - event
  - notification
  - subscriber
  - observer object (proxy) [reduce work on the object of interest]
    - forwarding
    - filtering of events types and content/attributes
    - patterns of events (occurrence of multiple events, not just one) [e.g. drop in temperature for more than five degrees three times in a row]
    - mailboxes (notifications in batches, subscriber might not be ready)
  - publisher (object of interest or observer object)
    - generates event notifications

Events and Notifications (4):



[3: observer checks for changes; object of interest is not part of the event service]

Events and Notifications (5)

- Jini
  - event generators (publishers)
  - remote event listeners (subscribers)
  - remote events (events)
  - third-party agents (observers)