The UNIVERSITY *of* EDINBURGH
SCHOOL *of* INFORMATICS

CS4/MSc

Distributed Systems

Björn Franke

bfranke@inf.ed.ac.uk

Room 2414

(Lecture 3: Remote Invocation and Distributed Objects,
29th September 2005)

Programming models for distributed application

Programming models for distributed applications are based on well-known models for single process applications:

Remote procedure call Client programs call procedures running in server programs running in separate processes, typically on different hosts.

Remote method invocation The state and functionality of the system is partitioned between *objects*. A client object may *invoke* a method on a remote object, residing in another process, on another host.

Event-based processing and event notification The behaviour of the system is driven by events, which represent local state changes within objects. Objects receive notifications of events at other objects in which they have registered an interest.

Interfaces and IDLs

To make interaction possible between remote components an *interface*, detailing the capabilities of the component, must be published. This may be in the programming language concerned (Java RMI) or in a special language designed for the purpose (IDL for CORBA or XDR for Sun RPC).

- Interfaces only describe the methods or procedures which are available — not variables (cf. *attributes* in CORBA IDL).
- There are no constructors for interfaces.
- The specification of a procedure or method describes parameters as *input*, *output* or both.
- Pointers cannot be passed as arguments or returned as results.

Delivery Semantics

When methods or procedures are invoked remotely it is not necessarily certain that they will execute as expected. This uncertainty can lead to different *invocation semantics*:

***Maybe* semantics** If requests are sent in unacknowledged messages there is no certainty that the request ever reached the server. In this case the invocation may have taken place or not.

***At-least-once* semantics** If requests may be retransmitted due to communication failures (eg. no reply) but duplicates are not filtered by the server, the retransmission of a request may result in the re-execution of a the method or procedure. Sun RPC uses *at-least-once* call semantics.

***At-most-once* semantics** If the system supports retransmission of requests and duplicate filtering at the server, we can be sure that re-execution does not happen. Duplicate requests trigger re-transmission of the original result. (Assumes the server maintains some form of *history*.) Java RMI uses *at-most-once* semantics.

Remote Procedure Call

- The *service interface* of the server defines the procedures that are available for calling remotely.
- There is a *stub procedure* in the client for each procedure in the service interface it wishes to access. To the client it looks like a local procedure but instead of executing the call it marshalls the procedure identifier and its arguments into a request message. When the reply message is received it unmarshalls the result and returns it to the calling process.
- A communication module passes the marshalled request to a communication module in the server process.
- The server process contains a *stub procedure* and a *service procedure* for each procedure in the service interface. A *dispatcher* in the server process selects the correct stub procedure, which unmarshalls the arguments and invokes the service procedure. When the procedure returns, the stub procedure marshalls the result and passes it to the communication module.

Files interface in 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;
```

RPC Example

- XDR is the interface language for Sun RPC and has an associated interface compiler `rpcgen` for use with `C`.
- The interface specifies a set of procedures together with supporting type definitions.
- Interfaces do not have names, but a program number (`9999`) and version number (`2`).
- Similarly each procedure has a number as well as a signature. The number is used in the message generated by the stub to identify which procedure is required.
- The results must be returned as a single value.
- Running `rpcgen` over the example will produce the client stub procedures, the server stub procedures, the server dispatcher and the server `main` procedure. In addition XDR marshalling and unmarshalling procedures for use by the dispatcher and the stubs will be produced.

Distributed objects

Distributed object systems have become the predominant paradigm for distributed systems.

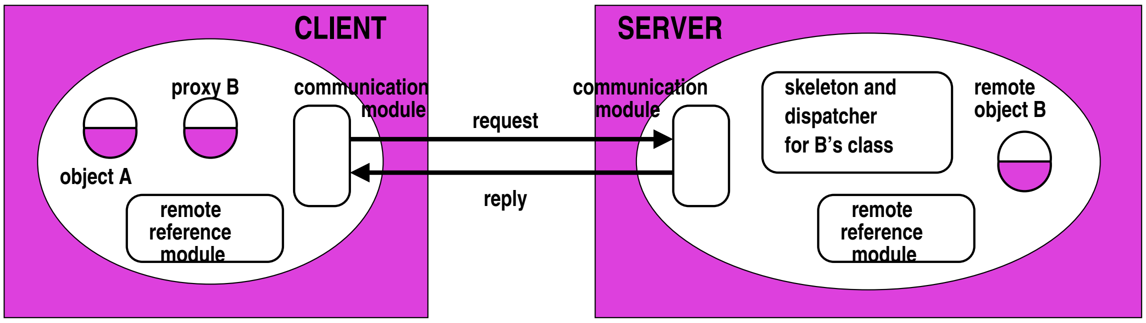
- The encapsulation of state variables within objects provides a logical partition of the state of the complete program. This facilitates distribution of state over several processes.
- Encapsulation within objects can be used to mask heterogeneity.
- Existing, legacy applications can be encapsulated within objects, offering companies a mechanism to migrate to distributed systems without complete re-implementation.
- Replication of objects can increase the reliability and the performance of systems.

Remote Object References

- When a distributed object system is used, a client needs to be able to uniquely identify the remote object on which it wishes to invoke a method. This is achieved using a *remote object reference*.
- A common approach is to incorporate the IP address name of the host and port number of the hosting process into the reference of the object.
- The time of creation may also be included, since the time and port together will uniquely identify a process.
- An object number may be used to differentiate different instantiations of transient objects.

32 bits	32 bits	32 bits	32 bits	
Internet address	port number	time	object number	interface

Anatomy of remote method invocation



- The *proxy* (or *stub*) in the client process makes the remote invocation transparent to the client. When object A wants to invoke a method on object B, it simply invokes the corresponding method on the B proxy object locally.
- The *remote reference module* is responsible for translating between local and remote object references.
- The communication modules implement the *request-reply* protocol.

Participants in remote method invocation (1)

The Communication Module is responsible for transmitting the requests and replies between the client and the server. It will provide the specified invocation semantics. When a request arrives at the server, the communication module will use the remote reference module to obtain the appropriate *dispatcher*.

The Remote Reference Module maintains a *remote object table* which has an entry for each remote object held locally, and each local proxy for a remote object. The module is also responsible for creating remote object references for remote objects the first time they are passed as argument or result.

The proxy makes the remote invocation appear as if it were transparent. It offers a method corresponding to each method of the interface of the remote object. But these methods marshall a reference to the target object, its own *methodId* and its arguments into a request message. It waits for the reply and unmarshalls the results returning them to the invoker.

Participants in remote method invocation (2)

The dispatcher receives incoming requests from the server communication module. When it receives a request message it uses the *methodId* to select the appropriate method in the *skeleton*, passing on the request message which still contains the marshalled arguments. The proxy and the dispatcher use the same allocation of *methodIds* to the methods of the remote interface.

The skeleton implements the methods of the remote interface, but similarly to the proxy, the implementation deals with unmarshalling rather than the functionality of the methods. A skeleton method unmarshalls the arguments in the request message and invokes (locally) the corresponding method in the remote object. When the invocation is complete it marshalls the result, together with any exceptions, in a reply message which is sent to the proxy.

The classes for the proxy, dispatcher and skeleton are generated automatically by an interface compiler.

Java RMI

- Java's *remote method invocation* (RMI) allows objects to be created and deployed from different instances of a JVM. These virtual machines might run on the same host, or they might run on different ones.
- The RMI package is called `java.rmi`. This contains two important sub-packages, `java.rmi.server` for use in implementations of RMI servers and the package `java.rmi.activation` for creating remotely accessible objects which are activated when needed.
- An RMI application declares *remote interfaces*. These are interfaces which define methods which can be invoked remotely. Remote interfaces must be declared public. If not, any client from another package will fail when trying to load a remote object which implements that interface.

Serializable objects and remote interfaces

In order to mark an object as serializable we need only declare that it implements the `java.io.Serializable` interface.

In order to mark a public interface as remote we need only declare that it extends the `java.rmi.Remote` interface. However the methods implementing a remote interface must declare that they throw the exception `java.rmi.RemoteException`.

When a local method invocation fails, its type and degree of failure is available for inspection and enquiry. When a remote invocation fails (even simply on another virtual machine on the same host) we do not have the same level of access to be able to uncover the cause. Thus even the simplest methods in a remote interface must declare that they may throw `java.rmi.RemoteException`. A remote interface will preferably cater for *partial failures*, perhaps by providing *idempotent methods*.

The RMI Registry class `java.rmi.Naming`

A *binder* in a distributed system is a separate service which maintains a table mapping textual names to remote object references. The RMIregistry is the binder for Java RMI. An instance of RMIregistry must run on every server computer that hosts remote objects. It is accessed by methods of the `Naming` class.

- `rebind` and `bind` take a string and a remote object reference and register the remote object by the given name.
- `unbind` removes a binding for a given (name, remote object reference) pair.
- `lookup` takes a name and returns a remote object reference; and
- `list` returns an array of strings containing all the names bound in the registry.

The RMI Security Manager

Code downloaded over the network is always assumed to be untrusted and so it is appropriate that a conservative security manager be installed. The class `java.rmi.RMISecurityManager` defines such a security manager and the first act of an object server which implements a remote interface would be to install such a security manager:

```
System.setSecurityManager(new RMISecurityManager());
```

Classes are passed between client and server only as they are needed. A class such as `java.lang.String` will never need to be downloaded over the network even if `String` values are being exchanged. Classes such as these are already available to both the client and the server.

Java RMI Example

The example shown in the following slides is discussed in more detail in Section 5.5 of CDK. It is intended to represent part of the implementation of a *shared whiteboard*.

- A groups of users share a common view of a drawing surface containing graphical objects.
- Each object has been drawn by one of the users.
- The server maintains the current state of a drawing by providing an operation for clients to inform it about the latest shape their users have drawn and keeping a record of all the shapes received.
- The server also provides operations allowing clients to retrieve the latest shapes drawn by other users by polling the server.
- The server has a version number (integer) that it increments each time a new shape arrives and attaches to the new shape. The server provides operations allowing clients to enquire about its version number and the version number of each shape.

Java Remote Interfaces *Shape* and *Shapelist*

```
import java.rmi.*;
import java.util.Vector;

public interface Shape extends Remote {
   int getVersion() throws RemoteException;
   GraphicalObject getAllState() throws RemoteException;
}

public interface ShapeList extends Remote {
   Shape newShape(GraphicalObject g) throws RemoteException;
   Vector allShapes() throws RemoteException;
   int getVersion() throws RemoteException;
}
```

Java class *ShapeListServer*

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

Java class *ShapeListServant*

```
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
  private int version;
  public ShapeListServant()throws RemoteException{...}
  public Shape newShape(GraphicalObject g) throws RemoteException{
    version++;
    Shape s = new ShapeServant( g, version);
    theList.addElement(s);
    return s;
  }
  public Vector allShapes()throws RemoteException{...}
  public int getVersion() throws RemoteException { ... }
}
```

Java 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");
      Vector sList = aShapeList.allShapes();
    } catch(RemoteException e) {
      System.out.println(e.getMessage());
    } catch(Exception e) {
      System.out.println("Client: " + e.getMessage());
    }
  }
}
```

Support for RMI in the JDK

- SUN's Java Developer's Kit provides an implementation of the *RMI registry*; the *RMI compiler* for generating *client stubs* and *server skeletons* for a given class definition; and the *RMI daemon*, a process which allows objects to be registered and activated in a Java virtual machine.
- The command `rmiregistry` creates and starts a remote object registry on the current host. An optional parameter to the command can be given in order to specify a port. Port 1099 is used if no port number is specified.
- The command `rmic` invokes the RMI compiler to generate stub and skeleton classes for remote objects from compiled class files which implement remote interfaces.
- The command `rmid` invokes the RMI daemon.