5 Distributed Objects: The Java Approach

Main Points

- Why distributed objects
- Distributed Object design points
- Java RMI
- Dynamic Code Loading

5.1 What's an Object?

An Object is an autonomous entity having state manipulable only by a set of methods

```
public interface BankAccount extends Remote {
   public void deposit(float amount)
      throws RemoteException;

   public void withdraw(float amount)
      throws RemoteException;

   public float balance()
      throws RemoteException;
}
```

5.2 Why Distributed Objects?

Distributed Systems multiplies complexity

- multiple machines
- multiple people
- multiple organisations

Large amount of communication between system designers in producing distributed systems.

Problem is how to manage complexity at *design time*

5.2.1 Software Engineering

Software design should produce well-engineered software which satisfies requirements:

- Comprehensible, so that its easy to maintain and modify. Easier to test
- Reusable, cheaper than rebuilding and fewer bugs

Objects as a basis for distributed system give you techniques to manage complexity:

Abstraction hide unnecessary details, so keep system comprehensible

Encapsulation allows elements to be extracted ⇒ comprehensibility and reusability

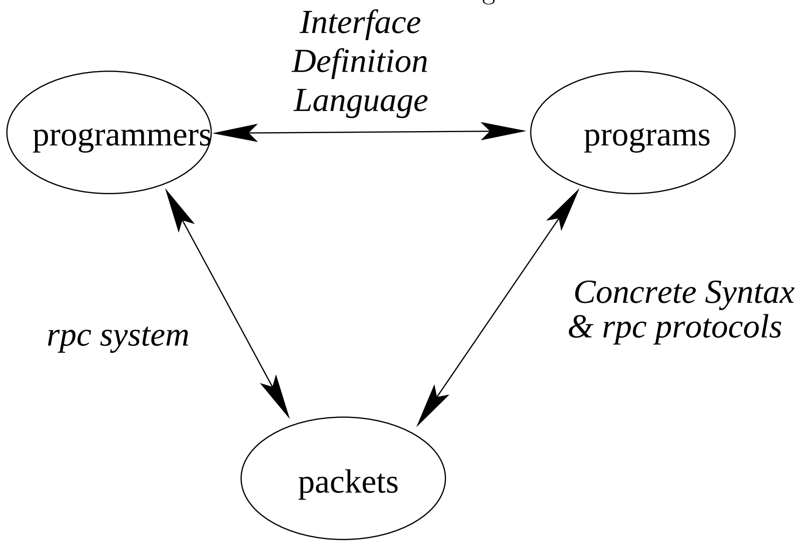
Concurrency control allows easy management of concurrent activities

5.3 How to build Distributed Object Systems

What are the various entities?

- Programmers using existing services
- Programs running on various machines offering services
- Packets using RPC protocol to invoke methods in programs

How do we communicate between these things?


Interface Definition Language
programmers
programs
rpc system
Concrete Syntax & rpc protocols
packets

5.4 Objects and RPC systems

No real distinction between distributed method invocation and rpc systems.
Pure object systems

- Provides dynamic binding through name service, possibly with migration and other features
- Protocol processing can be part of OS, allows asynchronous processing when appropriate

client server

- Examples include Java Remote Method Invocation (RMI), Corba

Static rpc systems such as Sun rpc

- Binding of services to machine by programmer
- Synchronous processing since protocol processing in user thread

5.5 Java RMI

- Java has RPC built in as Remote Method Invocation
- No separate IDL - uses Java for interface definition

5.5.1 Remote Interfaces

- An *interface* in Java specifies a set of methods that the object *implementing* that interface will provide
- Java RMI uses interfaces which extend *java.rmi.remote* as a way of specifying which methods can be invoked remotely.

```
public interface Foo extends Remote {
  public void myRemoteMethod() throws RemoteException;
}

public Bar implements Foo extends RemoteObject {
   ...
   public void myRemoteMethod() throws RemoteException {
      ...
   }
   ...
}
```

5.5.2 Remote Objects and Remote References

- To use a remote object, an object must acquire a *remote reference.*
- In Java, a remote reference looks just like a normal object reference.
- To provide the necessary communications code, a remote object must extend *java.rmi.RemoteObject* or one of its subclasses.
- Java will then provide remote references to the object when a reference as passed out of the local JVM, typically as a method result, or as a field in another result object.

5.5.3 Stub files and generic method dispatch

- To invoke a remote method, code acting as a proxy or *stub* for the remote object must run on the local machine.
- This code implements the appropriate interfaces, and marshalls the required method and arguments before sending them as a byte stream on a TCP connection to the remote machine.
- At the remote end, a generic dispatcher uses *reflection* to determine which method, and calls the *invoke* method from the reflection package to call the method.
- Results or exceptions are then returned to the caller.
- `rmic` is the tool that generates the stub file from the implementation of the remote object.

5.5.4 Java Distributed Garbage Collection

- Garbage collection (GC) is the removal of objects when they are no longer needed.
- Single address space GC basically checks for references to objects. If no references are found, the object is removed.
- Distributed GC is complicated because the traffic to check all possible references is infeasible - references can be passed arbitrarily from machine to machine.
- Instead, a remote reference corresponds to a proxy in the local machine. The proxy informs the remote object it holds a reference.
- When the proxy is GCed, it tells the remote object that it no longer holds a reference.
- When a remote object knows of zero proxies, it is a candidate for GC.

5.5.5 Parameter and Result Passing

- In local method invocation, object references are passed as arguments and results - *call-by-reference.*
- In remote method invocation, only objects which are accessible remotely can be passed by reference.
- Other objects must be passed by value - *call-by-value* - and instantiated as copies on the remote machine
- Objects passed by value must be capable of being passed by value - ie they must support the *java.io.Serializable* interface.

- *Serializable* objects and their associated object graph can be flattened into a byte array.
- If an object implements *Serializable* and all of its references are *Serializable*, it can be passed by value

5.5.6 Remote Exceptions

- The number and probability of failure modes are far higher in distributed systems.
- The designers of rmi decided to make this explicit by forcing programmers to deal with a possible RemoteException in all remote invocations.
- Therefore all methods in a *Remote* interface must throw *RemoteException*.

5.5.7 RMIRegistry

- How do classes get the initial remote reference (bootstrap)?
- Remote objects *bind* themselves against a given textual name (eg *myRemoteName*) with the *rmiregistry*
- Objects can then resolve the name remotely by querying the rmiregistry.
- The rmiregistry will return a remote reference, and hidden from the programmer, the location of the relevant server class files - the interface and the stub files.

5.5.8 Downloading of Classes

- The layout of classes and the bytecodes for implementing class methods are detailed in class files.
- Class files are loaded on demand as objects are created or static methods are invoked.
- Normal class loading comes through the default classloader, which searches the *CLASSPATH*.
- Additonal classloaders can be used by programmers to load class files from more exotic places.

5.5.9 ClassLoaders

- Rmi must allow the interface and stub files for remote objects to be downloaded over the network - uses the *rmiClassLoader*.
- Code loaded from arbitrary places is a security risk.

- Java provides for a security policy to be defined for a classloader so that all classes from that classloader can have their actions *sandboxed*.
- Typically, these actions are network access, file access, screen access etc, and are specified in java policy files.

5.5.10 Activation

- Using an active thread continuously for an object which is accessed infrequently may be a poor use of resources - consider machines with millions of objects.
- Instead, allow objects to change state from active to passive and vice versa.
- When active, they are normal remote objects.
- When passive, the object's state is stored in persistent storage eg a file, and responsibility for accepting calls to that object is handed over to an *activator*.
- When the activator receives a call, it creates a new instance of the object and instantiates its state from its stored state.
- Compare to the use of inetd to control typical Internet services such as ftp, telnet etc.

5.6 Summary

- Described the key elements of Java RMI.
- Refer to these in using rmi to help in udnerstanding some of the problems that occur.
- Other possible choices for distributed objects in the next lecture