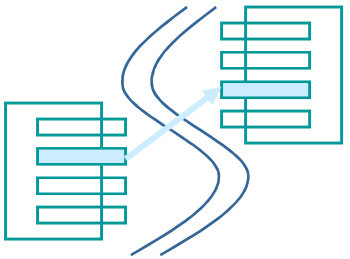
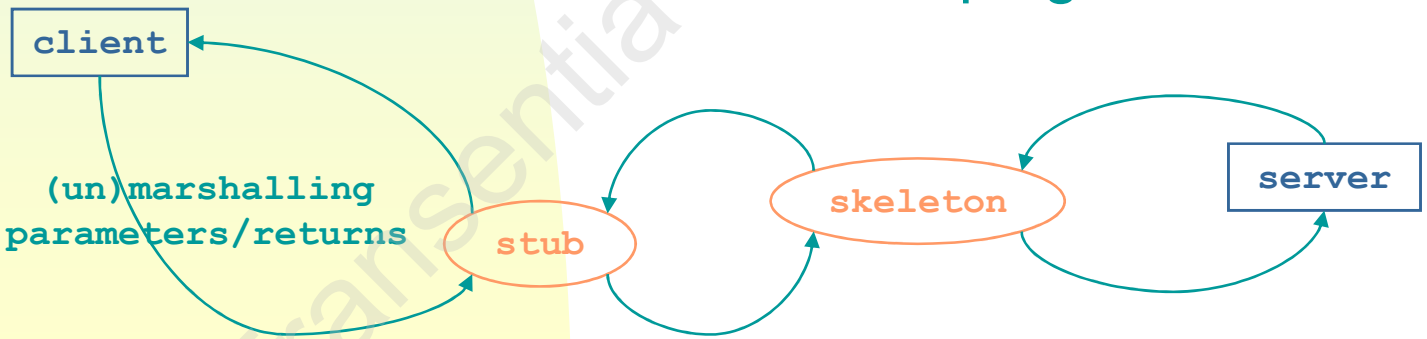
RMI

- Remote Method Invocation
  - *"…the action of invoking a method of a remote interface on a remote object."*
    - maintains type safety, security, etc.
  - greater degree of abstraction than sockets
    - lots of the work done for you
      - parameter/return marshalling
      - name lookups
    - still lots of housekeeping to do





RMI

- goals:
  - support seamless remote invocation on objects in different virtual machines
  - support callbacks from servers to applets
  - integrate the distributed object model into Java while retaining most of the language's semantics
  - make differences between the distributed object model and local Java object model apparent, while preserving the safety features of the Java language and runtime environment
  - make distributed systems as simple as possible to program

RMI

- RMI obviates the need to develop fiddly Application Layer protocols that don't have any direct relevance to a project
- main aspects to examine:
  - client side
  - server side
- also:
  - registry/naming
  - stubs and skeletons
  - security

RMI

- Client side
  - *"…whole point of RMI is to make the use of remote objects very simple…only extra thing needed is to obtain a reference to the remote interface"*
  - java.rmi.RemoteException Class
    - makes it possible to distinguish local exceptions, & exceptions specific to the method, from exceptions thrown by the underlying mechanisms
    - can be constructed with a nested exception (a Throwable): the underlying exception that occurred during an RMI call

RMI

- The Naming Class
  - allows remote objects to be retrieved and defined via URLs
    - rmi://java.sun.com:2001/root
  - methods (from java.rmi.registry.Registry interface)
    - bind, rebind
    - unbind
    - lookup
    - list
  - registry supports bind, unbind, and rebind only on the same host as the originating server; a lookup can be done from any host
  - JDK supplies rmiregistry
    - simple registry server

RMI

```
import java.rmi.*;
import java.rmi.registry.*;

public class DisplayTime
  {
  public static void main (String [] args)
    {
    System.setSecurityManager (new RMISecurityManager ());
    try
      {
      TimeSvcI t = (TimeSvcI) Naming.lookup ("//machine.name:2005/TimeSvc");
      System.out.println ("The time = " + t.getTime());
      }
    catch(Exception e)
      {
      e.printStackTrace();
      }
    }
  }
```

RMI

- Server side
  - java.rmi.Remote Interface

```
public interface Remote { }
```

    - marker interface: all remote objects must directly or indirectly implement this
  - java.rmi.server.UnicastRemoteObject
    - provides support for point-to-point active object references using TCP-based streams
    - server must extend this and implement the java.rmi.Remote interface
  - java.rmi.server.Unreferenced interface
    - lets server know when no clients reference it

RMI

```
import java.rmi.*;

interface TimeSvcI extends Remote
  {
  long getTime() throws RemoteException;
  }
```

RMI

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class TimeSvc extends UnicastRemoteObject implements TimeSvcI
  {
  public long getTime() throws RemoteException
    { return (System.currentTimeMillis ()); }

  // must implement constructor to throw RemoteException
  public TimeSvc () throws RemoteException
    { /* super () Called automatically */ }

  public static void main (String [] args)
    {
    System.setSecurityManager (new RMISecurityManager ());
    try
      {
      TimeSvc t = new TimeSvc ();
      Naming.bind ("//machine.name:2005/TimeSvc", t);
      System.out.println ("TimeSvc running...");
      }
    catch (Exception e)
      { e.printStackTrace (); }
    }
  }
```

RMI

- Stub
  - stub == client-side proxy
  - implements all the interfaces that are supported by the remote object. Responsible for:
    - initiating a call to the remote object
    - marshalling arguments to a stream (obtained from the remote reference layer)
    - informing the remote reference layer that the call should be invoked
    - unmarshalling the return value or exception
    - informing the remote reference layer that the call is complete

RMI

- Skeleton
  - server-side entity which dispatches calls to the actual remote object implementation. Responsible for:
    - unmarshalling arguments from the marshal stream
    - making the up-call to the actual remote object implementation
    - marshalling the return value of the call or an exception (if one occurred) onto the marshal stream
- args/return types must implement the Serializable interface

RMI

- Stubs & skeletons
  - loaded dynamically “behind the scenes”
    - must be accessible to ‘real’ code (ie on CLASSPATH or downloaded from server)
  - generated using the rmic compiler
    - *“The compiler is invoked with the package qualified class name of the remote object class. The class must previously have been compiled successfully.”*
      - rmic TimeSvc *(since no package in this case…)*
    - creates:
      - TimeSvc_Stub.class
      - TimeSvc_Skel.class

RMI

- RMISecurityManager
  - simple security manager disables all functions except class definition and access…a downloaded class is allowed to make a connection if the connection was initiated via the RMI transport mechanism
  - if no security manager set, stub loading is disabled...ensures that some security manager must be supplied
  - applets: *"…does not apply to applets, which run under the protection of their browser's security manager."*
  - callbacks: *"If an applet creates and passes a remote object to the server, the server can use RMI to make a callback to the remote object."*

RMI

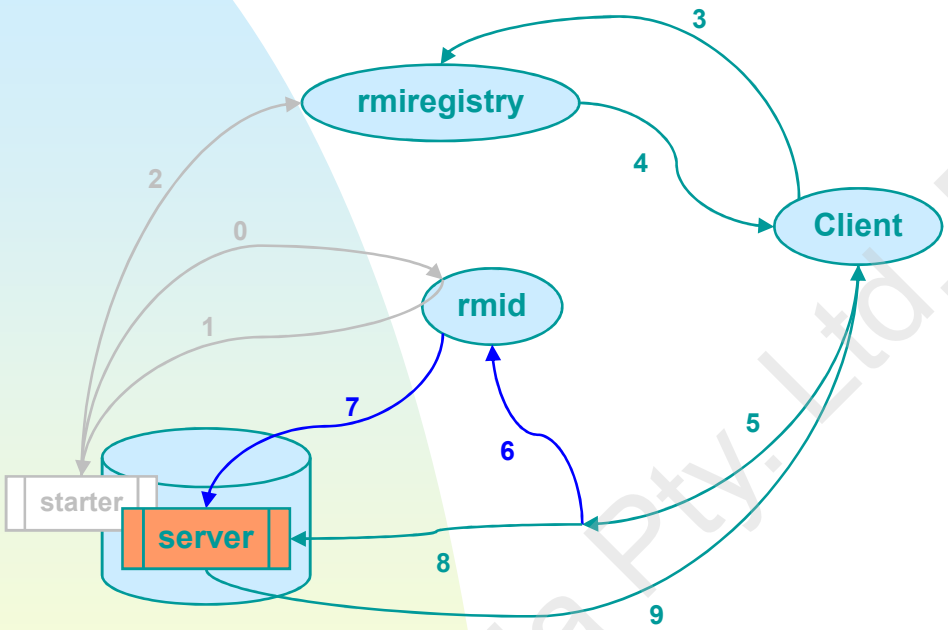
- Remote Activation
  - obviates the need to have a server running continuously, even when unused
  - RMI catches up with CORBA
  - new *rmid* activation daemon
  - java.rmi.activate package
    - Activable
      - analogous to UnicastRemoteObject for activable servers
    - ActivationDesc
      - provides all the info. that rmid needs to create a new instance of the implementation class
    - MarshalledObject
      - *"provides a flexible mechanism for passing persistence or initialization data…into the implementation's class file"*

RMI

- need to create additional starter application:
  - tells rmid that the server is activable
  - may initialise parameters/persistent data stores
  - establishes runtime security policy regime for the server (what files, etc. can be accessed…)
  - advertises server ‘proper’ with rmiregistry

RMI

Activation Mechanism



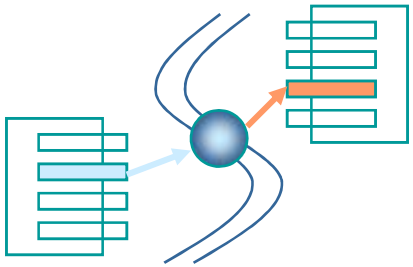
0. starter application registers server for later activation
1. starter receives reference to server's interface
2. starter registers server's interface
3. client looks up interface
4. gets reference back
5. calls method
6. call causes stub to 'fault'
7. rmid activates server
8. method call proceeds
9. method returns results (if any)

RMI

- Development process:
  - start *rmiregistry* (only needed once)
  - start *rmid* if servers are activable (" " ")
  - compile source files
  - apply *rmic* to any class that implements the Remote interface
  - run the server, or run the starter application, as appropriate
  - run the client

RMI

- Object Request Brokers (ORBs)
  - RMI is direct object-object, Java-Java
  - ORBs interpose between the two objects
    - similar, but greater flexibility
    - allow mixed languages
      - C++, Smalltalk, Ada, Python, etc
    - lots more work
      - only really worth it for heterogeneous enterprise solutions
    - much (standardisation) activity
      - IIOP
      - Java ORBs



RMI

- The RMI/CORBA political situation & future
  - RMI and IIOP statements:
    - *"Sun will continue to support and evolve the Java API"*
    - *"You will be able to access CORBA-based objects through IIOP"*
    - *"… it remains the technology of choice for Java-based distributed computing."*
    - *"Support [for the CORBA IDL]…will be included in the next version of the JDK"*
  - *"…an application in which clients access data from a large number of sources is particularly well-suited to CORBA."*