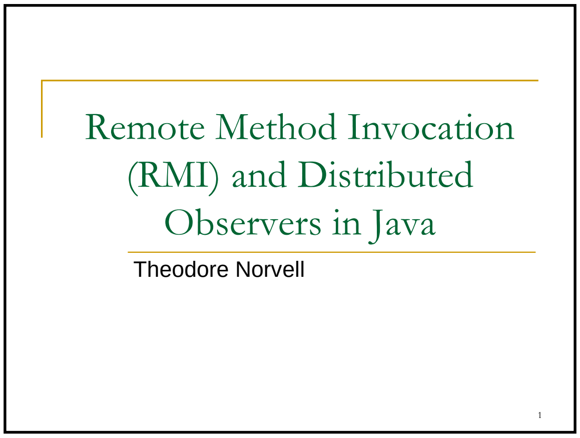
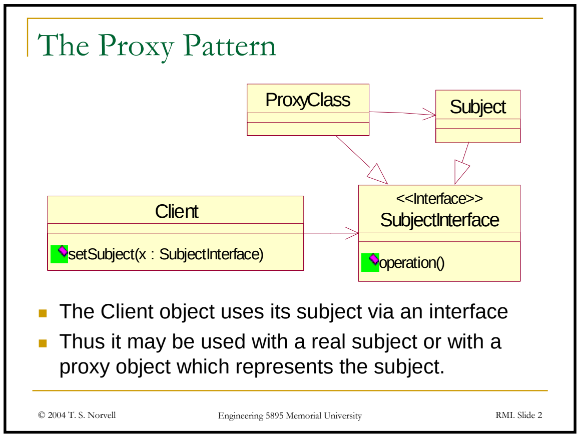


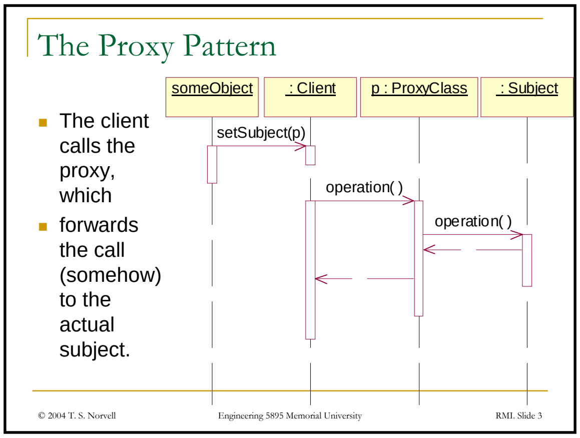
Remote Method Invocation (RMI) and Distributed Observers in Java

Theodore Norvell



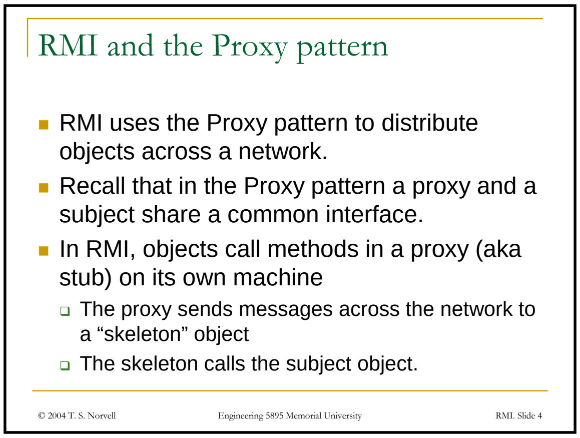
The Proxy Pattern

- The Client object uses its subject via an interface
- Thus it may be used with a real subject or with a proxy object which represents the subject.



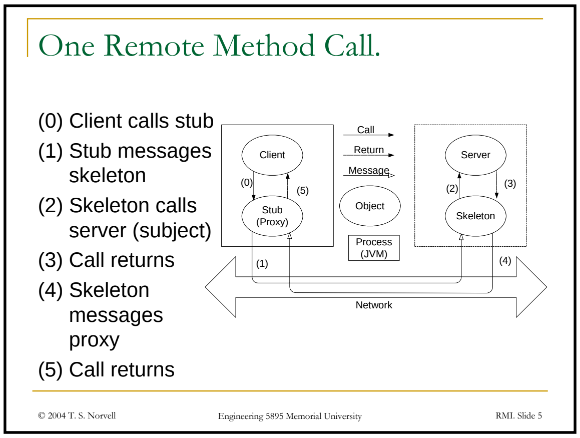
The Proxy Pattern

- The client calls the proxy, which
- forwards the call (somehow) to the actual subject.



RMI and the Proxy pattern

- RMI uses the Proxy pattern to distribute objects across a network.
- Recall that in the Proxy pattern a proxy and a subject share a common interface.
- In RMI, objects call methods in a proxy (aka stub) on its own machine
  - The proxy sends messages across the network to a “skeleton” object
  - The skeleton calls the subject object.



One Remote Method Call.

(0) Client calls stub

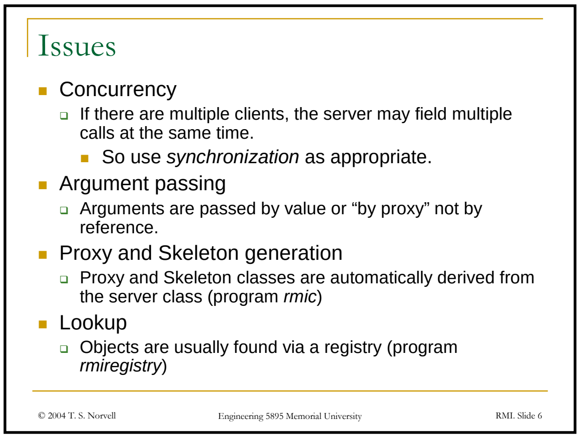
(1) Stub messages skeleton

(2) Skeleton calls server (subject)

(3) Call returns

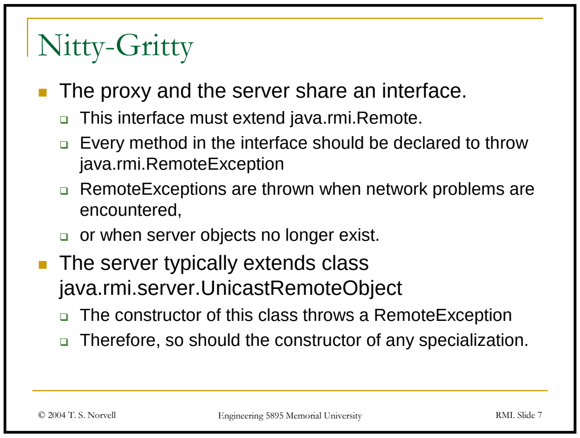
(4) Skeleton messages proxy

(5) Call returns



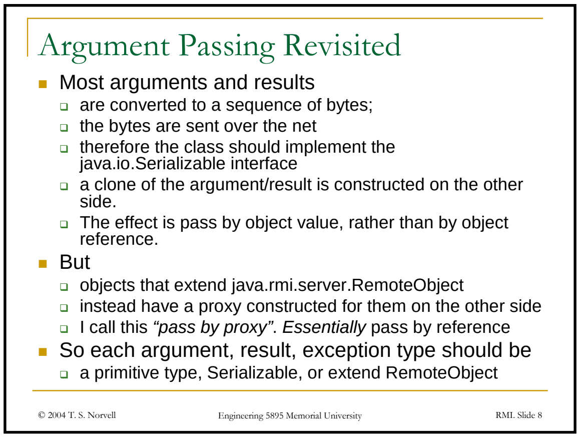
Issues

- Concurrency
  - If there are multiple clients, the server may field multiple calls at the same time.
    - So use *synchronization* as appropriate.
- Argument passing
  - Arguments are passed by value or “by proxy” not by reference.
- Proxy and Skeleton generation
  - Proxy and Skeleton classes are automatically derived from the server class (program *rmic*)
- Lookup
  - Objects are usually found via a registry (program *rmiregistry*)



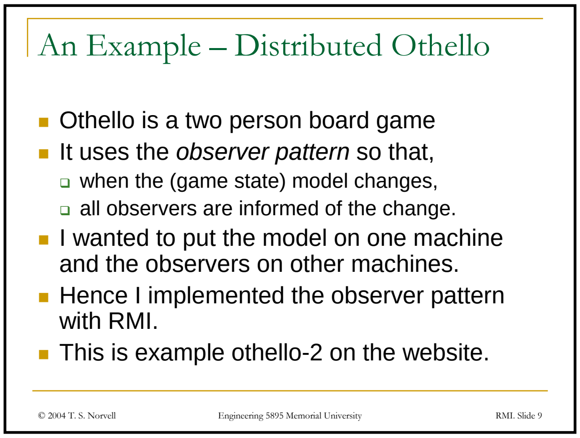
Nitty-Gritty

- The proxy and the server share an interface.
  - This interface must extend java.rmi.Remote.
  - Every method in the interface should be declared to throw java.rmi.RemoteException
  - RemoteExceptions are thrown when network problems are encountered,
  - or when server objects no longer exist.
- The server typically extends class java.rmi.server.UnicastRemoteObject
  - The constructor of this class throws a RemoteException
  - Therefore, so should the constructor of any specialization.



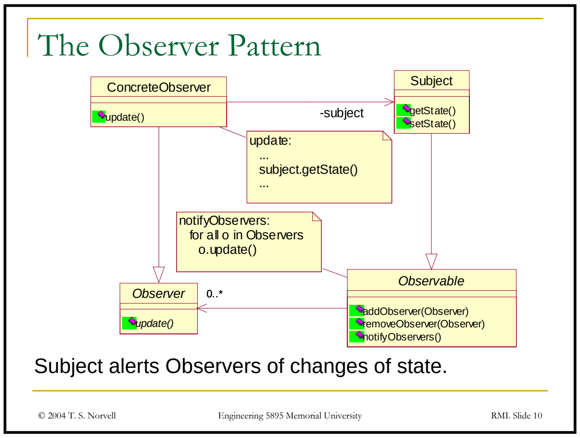
Argument Passing Revisited

- Most arguments and results
  - are converted to a sequence of bytes;
  - the bytes are sent over the net
  - therefore the class should implement the java.io.Serializable interface
  - a clone of the argument/result is constructed on the other side.
  - The effect is pass by object value, rather than by object reference.
- But
  - objects that extend java.rmi.server.RemoteObject
  - instead have a proxy constructed for them on the other side
  - I call this *"pass by proxy"*. *Essentially* pass by reference
- So each argument, result, exception type should be
  - a primitive type, Serializable, or extend RemoteObject



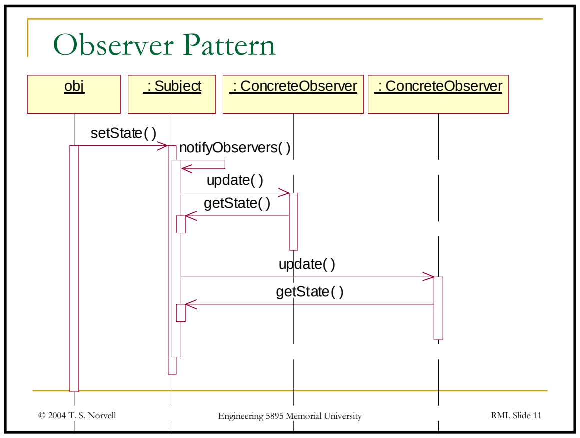
An Example – Distributed Othello

- Othello is a two person board game
- It uses the *observer pattern* so that,
  - when the (game state) model changes,
  - all observers are informed of the change.
- I wanted to put the model on one machine and the observers on other machines.
- Hence I implemented the observer pattern with RMI.
- This is example othello-2 on the website.

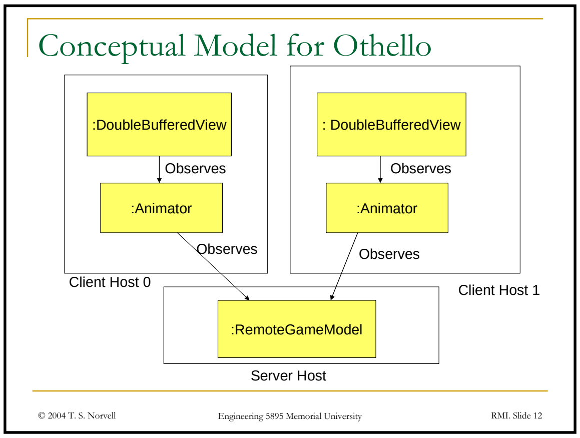


The Observer Pattern

Subject alerts Observers of changes of state.

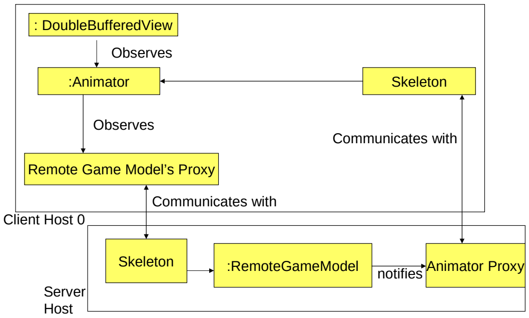


Observer Pattern

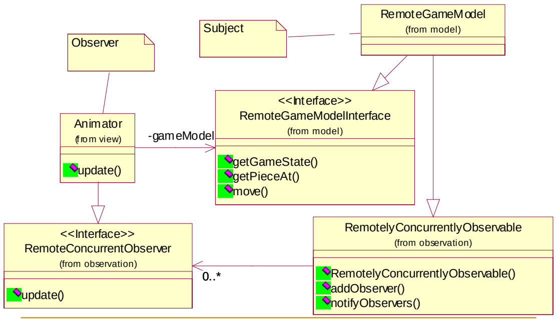


Conceptual Model for Othello

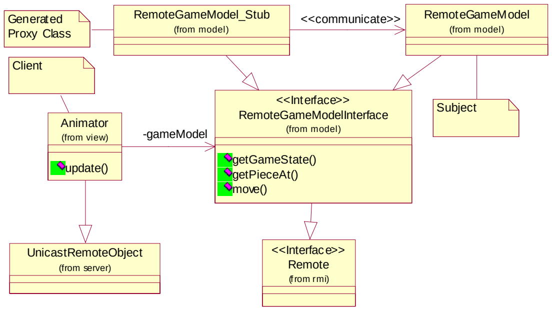
Detailed View



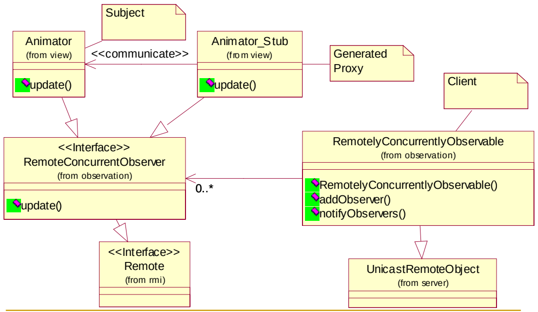
The Observer Pattern in Othello



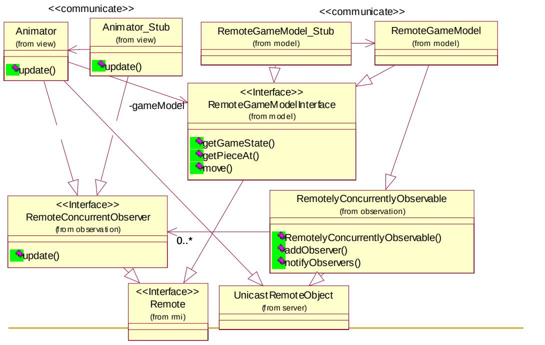
Proxies for the Remote Game Model



Proxies for the Animator

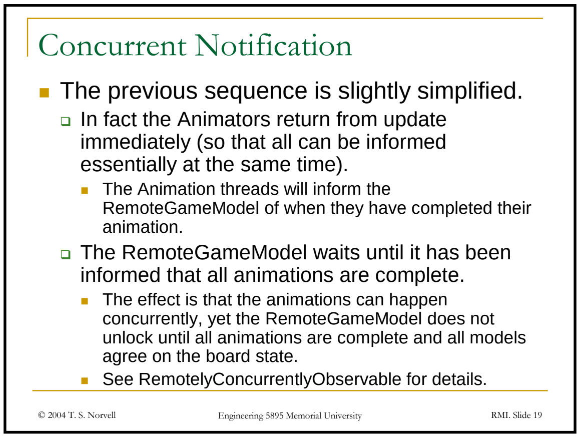


Animator / RemoteGameModel Relationship



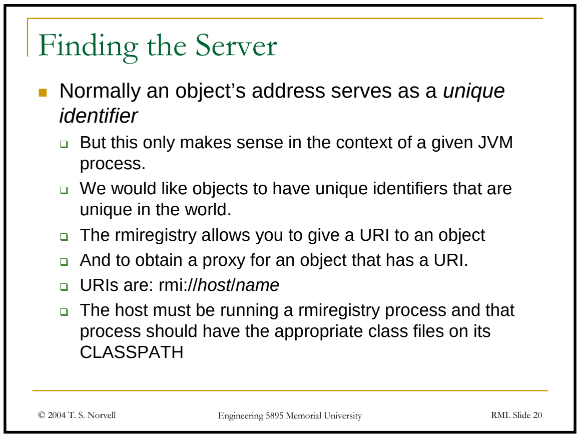
Typical sequence (slightly simplified).

- A remote client calls a synchronized mutator on the RemoteGameModel via its stub & skeleton
  - The RemoteGameModel updates its state and notifies each Animator via its stubs & skeletons.
    - The Observers (Animator objects) call the RemoteGameModel accessor “getPieceAt” via its stubs and skeleton.
    - (Therefore this accessor must *not* be synchronized!)
    - getPieceAt returns
  - The update routines return.
- The original mutator call returns and the RemoteGameModel becomes unlocked.



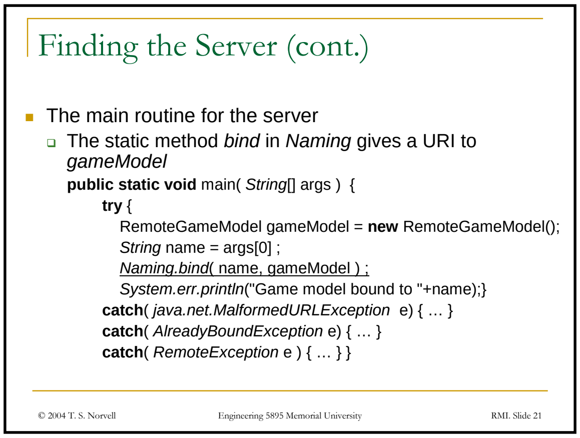
Concurrent Notification

- The previous sequence is slightly simplified.
  - In fact the Animators return from update immediately (so that all can be informed essentially at the same time).
    - The Animation threads will inform the RemoteGameModel of when they have completed their animation.
  - The RemoteGameModel waits until it has been informed that all animations are complete.
    - The effect is that the animations can happen concurrently, yet the RemoteGameModel does not unlock until all animations are complete and all models agree on the board state.
    - See RemotelyConcurrentlyObservable for details.



Finding the Server

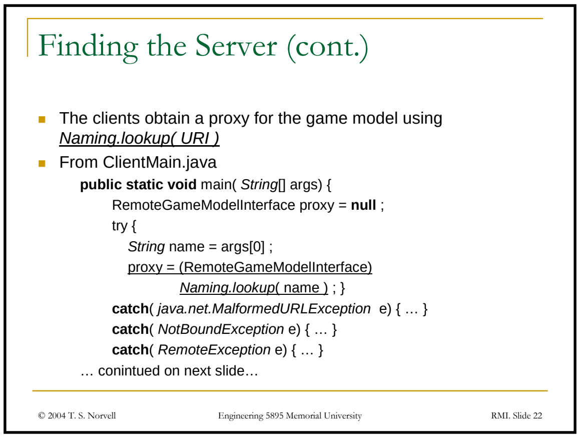
- Normally an object's address serves as a *unique identifier*
  - But this only makes sense in the context of a given JVM process.
  - We would like objects to have unique identifiers that are unique in the world.
  - The rmiregistry allows you to give a URI to an object
  - And to obtain a proxy for an object that has a URI.
  - URIs are: rmi://*host*/*name*
  - The host must be running a rmiregistry process and that process should have the appropriate class files on its CLASSPATH



Finding the Server (cont.)

- The main routine for the server
  - The static method *bind* in *Naming* gives a URI to *gameModel*

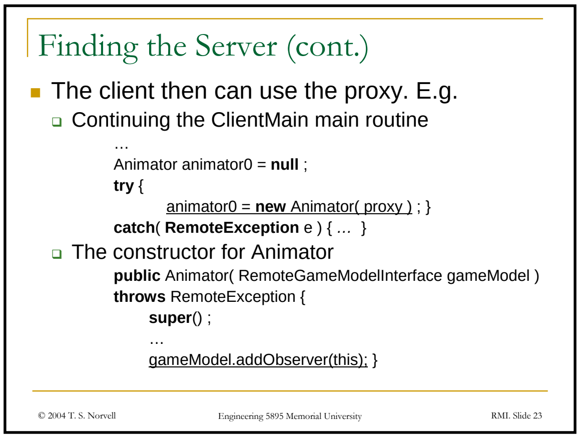
```
public static void main( String[] args )  {
      try {
         RemoteGameModel gameModel = new RemoteGameModel();
         String name = args[0] ;
         Naming.bind( name, gameModel ) ;
         System.err.println("Game model bound to "+name);}
      catch( java.net.MalformedURLException  e) { … }
      catch( AlreadyBoundException e) { … }
      catch( RemoteException e ) { … } }
```



Finding the Server (cont.)

- The clients obtain a proxy for the game model using *Naming.lookup(URI)*
- From ClientMain.java

```
public static void main( String[] args) {
      RemoteGameModelInterface proxy = null ;
      try {
         String name = args[0] ;
         proxy = (RemoteGameModelInterface)
                  Naming.lookup( name ) ; }
      catch( java.net.MalformedURLException  e) { … }
      catch( NotBoundException e) { … }
      catch( RemoteException e) { … }
… conintued on next slide…
```



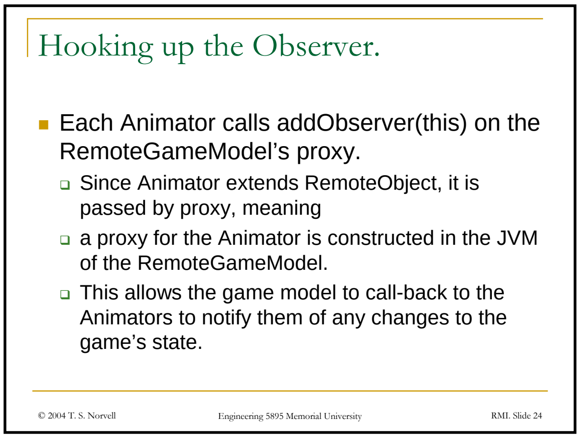
Finding the Server (cont.)

- The client then can use the proxy. E.g.
  - Continuing the ClientMain main routine

```
…
Animator animator0 = null ;
try {
      animator0 = new Animator( proxy ) ; }
catch( RemoteException e ) { …  }
```

  - The constructor for Animator

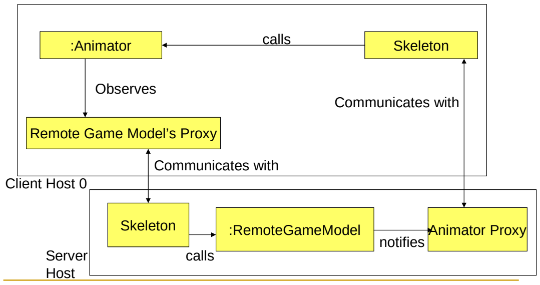
```
public Animator( RemoteGameModelInterface gameModel )
throws RemoteException {
      super() ;
      …
      gameModel.addObserver(this); }
```



Hooking up the Observer.

- Each Animator calls addObserver(this) on the RemoteGameModel's proxy.
  - Since Animator extends RemoteObject, it is passed by proxy, meaning
  - a proxy for the Animator is constructed in the JVM of the RemoteGameModel.
  - This allows the game model to call-back to the Animators to notify them of any changes to the game's state.

The final hookup (again)



:Animator

calls

Skeleton

Observes

Communicates with

Remote Game Model's Proxy

Communicates with

Client Host 0

Skeleton

:RemoteGameModel

notifies

Animator Proxy

Server Host

calls

Creating the stubs and skeletons

- First we write a server class: E.g.

```
public class RemoteGameModel
   extends RemotelyConcurrentlyObservable
   implements RemoteGameModelInterface
{
    public RemoteGameModel() throws RemoteException {
         super() ; } … }
```

  - We compile it with “javac”
  - Then we use the command “rmic”

```
D:\othello-2> cd classes
D:\othello-2\classes> rmic othello.model.RemoteGameModel
```

  - This creates two new “.class” files

```
D:\othello-2\classes> dir othello\model
28-03-2003  01:29           2,672 RemoteGameModel_Skel.class
28-03-2003  01:29           5,033 RemoteGameModel_Stub.class
…
```

A Few Words of Warning

- RMI makes it seductively easy to treat remote objects as if they were local.
- Keep in mind
  - Partial Failure
    - Part of the system of object may fail
    - Partial failures may be intermittent
    - Network delays
    - On a large network, delays are indistinguishable from failures
      - In the Othello example, failure was not considered. The system is not designed to be resilient to partial failures.

A Few Words of Warning (cont.)

- Keep in mind (cont.)
  - Performance
    - Remote calls are several orders of magnitude more expensive than local calls (100,000 or more to 1)
      - E.g. in the Othello example, this motivated splitting the model into local and remote copies.
  - Concurrency
    - Remote calls introduce concurrency that may not be in a nondistributed system.
      - E.g. in Othello, I had to be careful not to use **synchronized** for accessors called by remote Observers and to consider the data integrity consequences of not doing so.

A Few Words of Warning (cont.)

- Keep in mind (cont.)
  - Semantics changes
    - In Java, local calls pass objects by reference.
    - Remote calls pass objects either by copy or by copying a proxy.
      - E.g. in the Othello game as I converted from the nondistributed to a distributed version, the semantics of some calls changed, even though I did not change the source code for those calls.