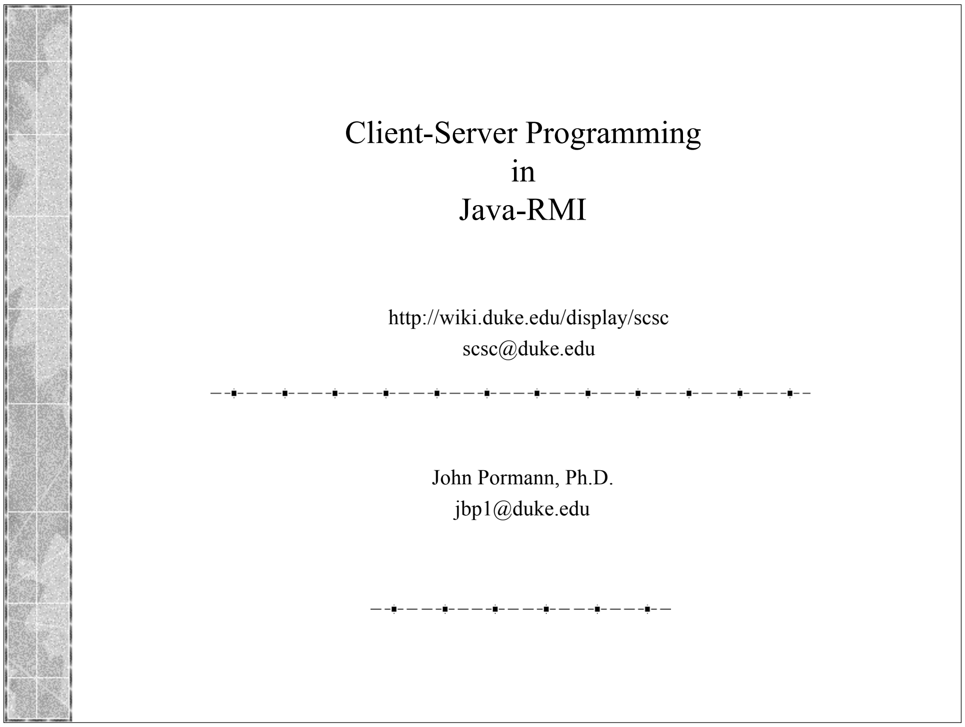
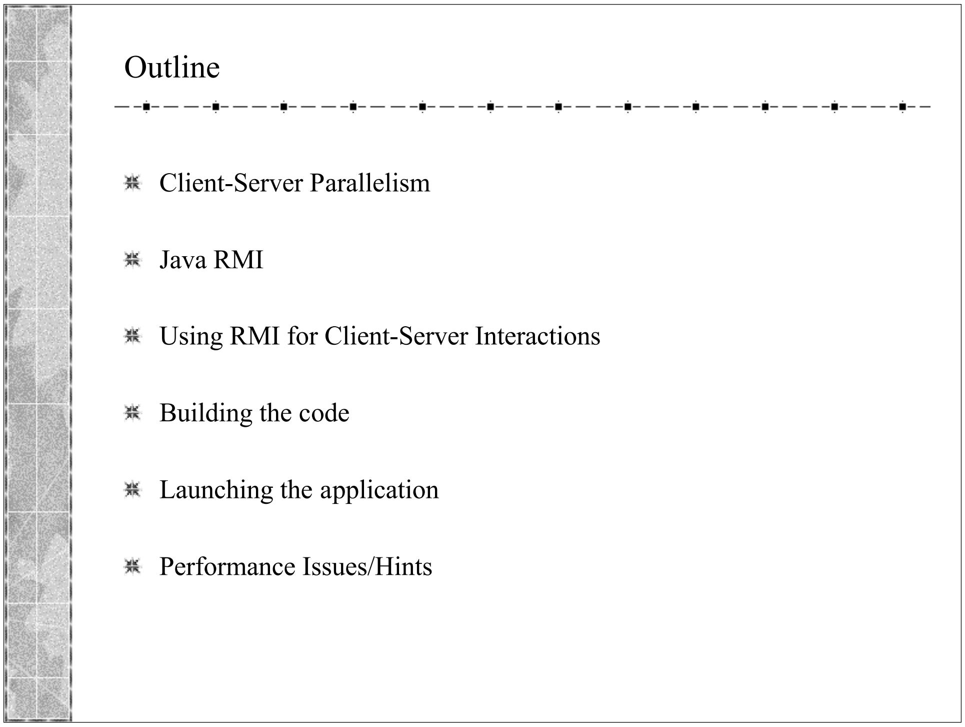


Client-Server Programming in Java-RMI

http://wiki.duke.edu/display/scsc
scsc@duke.edu

John Pormann, Ph.D.
jbp1@duke.edu



Outline

- Client-Server Parallelism
- Java RMI
- Using RMI for Client-Server Interactions
- Building the code
- Launching the application
- Performance Issues/Hints

Client-Server Parallelism

- Multiple independent clients that ONLY interact with the server
    - Client-to-Client data exchange may not be needed at all, or else it is handled through the server as 2 separate exchanges
        - Client #1 talks to Server
        - Server talks to Client #2
    - Server coordinates any data distribution that is needed

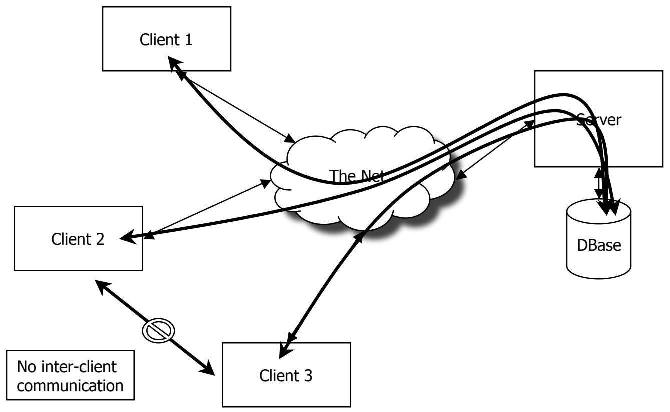
- Sometimes referred to as "Remote Procedure Calls"
    - Clients interact with server as though it was a local subroutine:

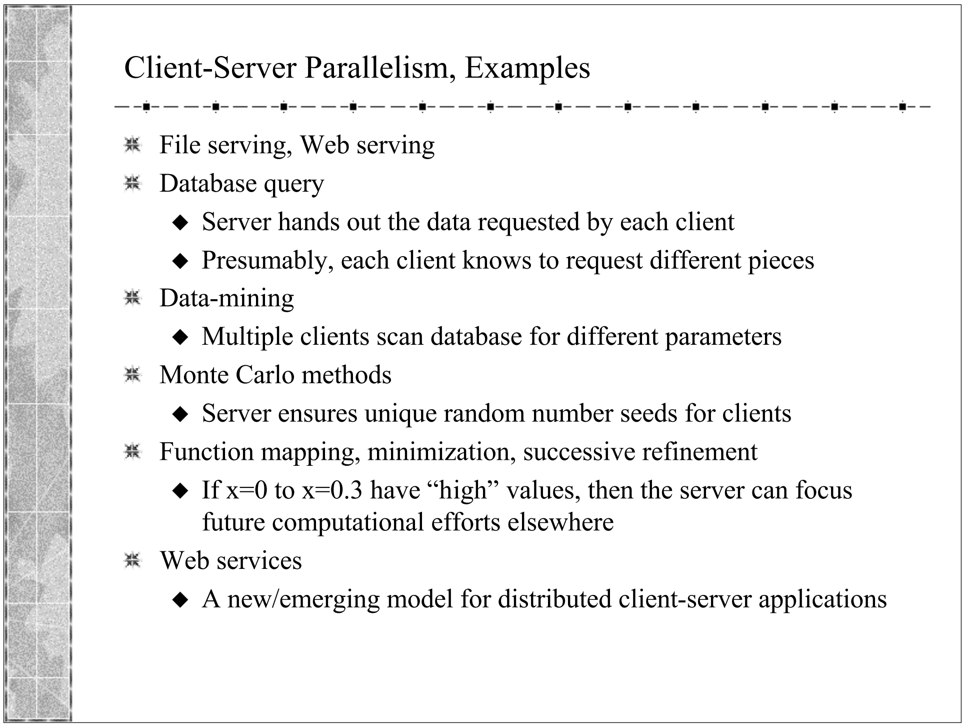
```
my_data = get_job_data( server );
value = compute( my_data );
err = post_job_result( server, my_data, value );
```

-
    -
        - Where the get_job_data and post_job_results functions just happen to execute on another machine
    - What if server has too many clients and slows down?

Java calls them "Remote Method Invocations"

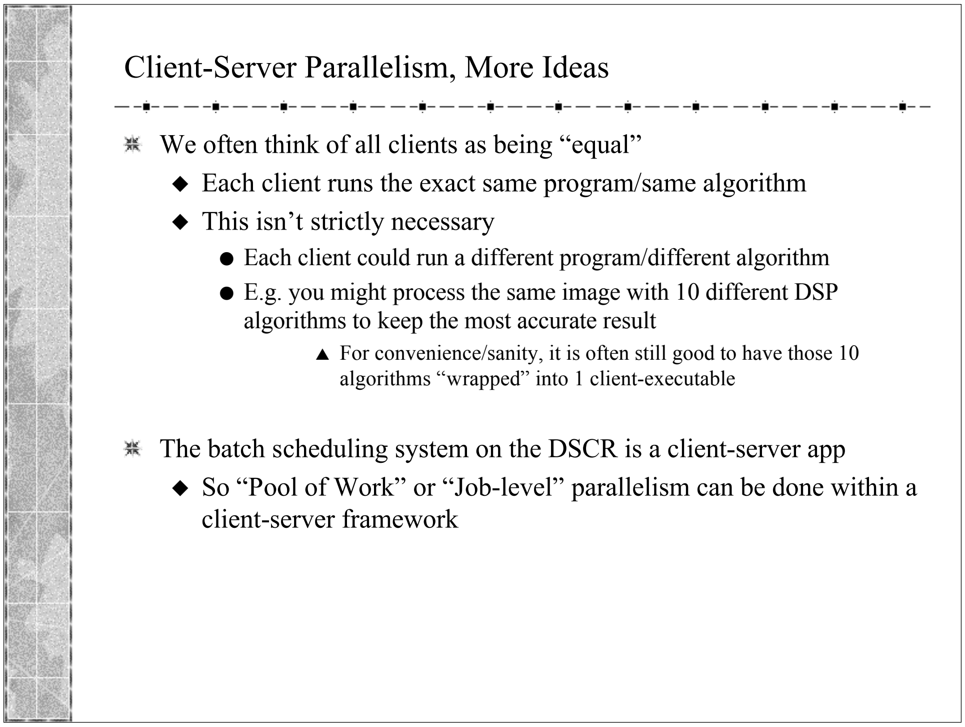
Client-Server Parallelism, cont'd





Client-Server Parallelism, Examples

- File serving, Web serving
- Database query
  - Server hands out the data requested by each client
  - Presumably, each client knows to request different pieces
- Data-mining
  - Multiple clients scan database for different parameters
- Monte Carlo methods
  - Server ensures unique random number seeds for clients
- Function mapping, minimization, successive refinement
  - If x=0 to x=0.3 have “high” values, then the server can focus future computational efforts elsewhere
- Web services
  - A new/emerging model for distributed client-server applications



Client-Server Parallelism, More Ideas

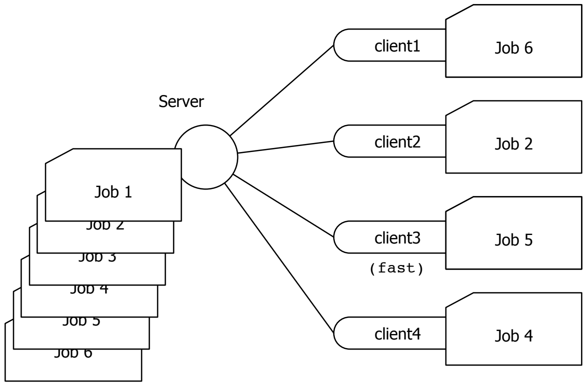
- We often think of all clients as being “equal”
  - Each client runs the exact same program/same algorithm
  - This isn’t strictly necessary
    - Each client could run a different program/different algorithm
    - E.g. you might process the same image with 10 different DSP algorithms to keep the most accurate result
      - For convenience/sanity, it is often still good to have those 10 algorithms “wrapped” into 1 client-executable
- The batch scheduling system on the DSCR is a client-server app
  - So “Pool of Work” or “Job-level” parallelism can be done within a client-server framework

Pros/Cons of Client-Server Parallelism

- Cons:
  - You must be able to fit each client-job onto a single machines
    - Memory could be a limiting factor
  - It will still take X hours for any one client to finish, it’s just that you will get back 10 or 100 results every X hours
    - Sometimes (fast) partial results can be helpful
  - Scalability is a function of the (coarse) problem “size”

- Pros:
  - It doesn’t require “real” parallel programming!
  - This can be one of the most efficient forms of parallelism
  - It provides automatic load-balancing

Automatic Load-Balancing



The Java Programming Language

- Java is a full-featured, object-oriented programming language
  - “Write once, Run anywhere” -- uses a Java Virtual Machine to hide the details of the hardware
    - The Java Byte-Code (assembly language for the JVM) can be shipped to remote machines (clients) and you know it will execute there
  - “try..catch” blocks are particularly well-suited for distributed, networked programming (where things frequently fail)

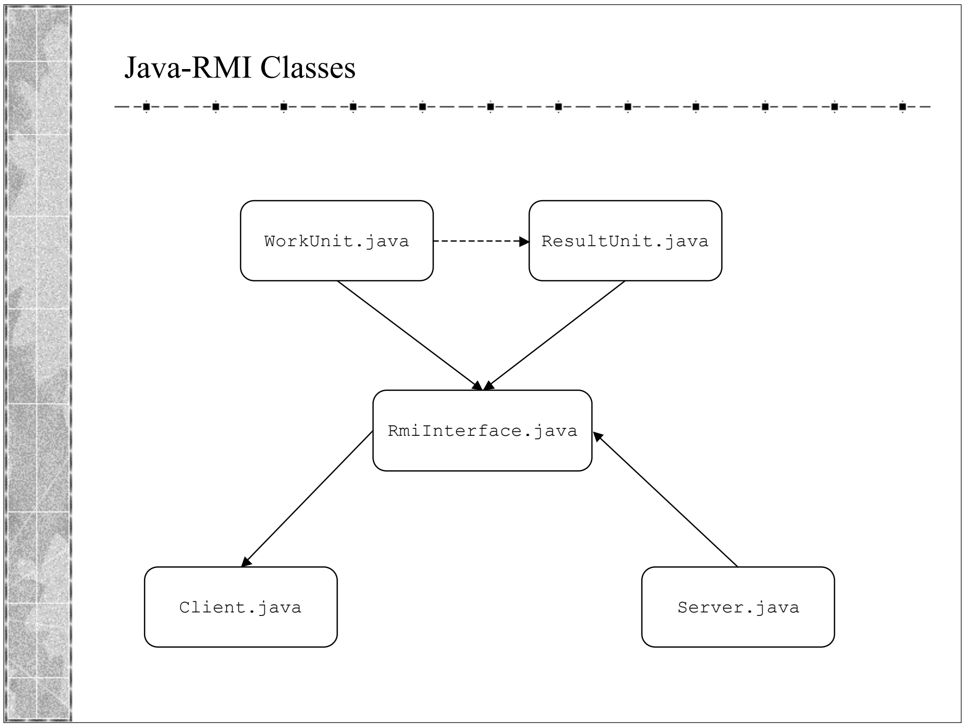
- Sun pioneered the idea of “Remote Procedure Calls” (RPC), so it was natural for them to roll out “Remote Method Invocations” (RMI)
  - Same basic approach is used: automatically generates stubs for the remote methods, so the user only deals with the “real code” (not the network sockets, parallel programming code)

Java-RMI

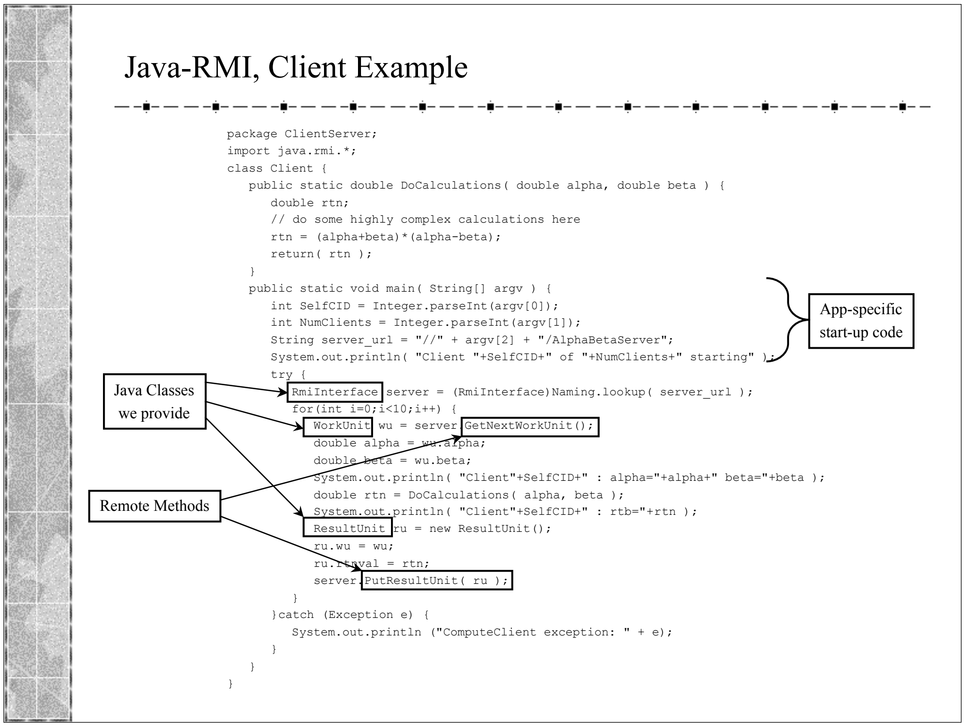
- In Java, “Remote Methods” look like regular method (function) calls

```
RmiInterface server = (RmiInterface)Naming.lookup( some_url );
WorkUnit wu = server.GetNextWorkUnit();
```

  - The only hint that it is a remote method is the “Naming.lookup” call with a URL
    - (Hidden) Java RMI classes do all the “heavy lifting” -- opening network sockets, transfer the function arguments, retrieve the results, close the network sockets
  - On the server side, there is a “Naming.rebind” call in the main method (to bind the server to a network location/URL)
    - Again, the Java RMI classes to the “heavy lifting” -- converting the remote method call into a “regular” method call on “normal” Java code running within the server

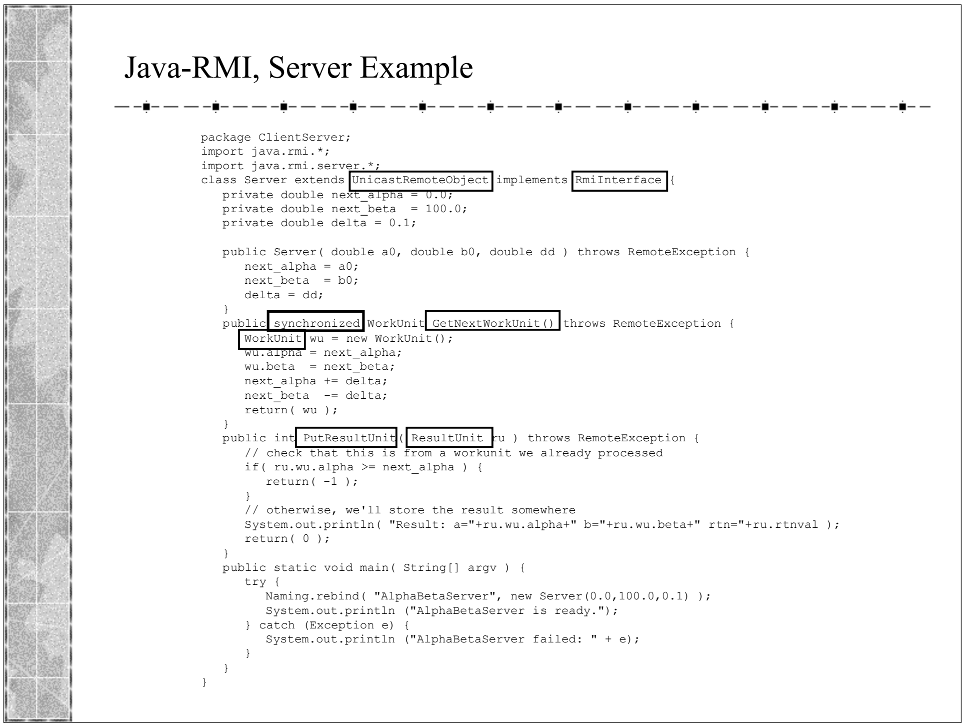


Java-RMI Classes



Java-RMI, Client Example

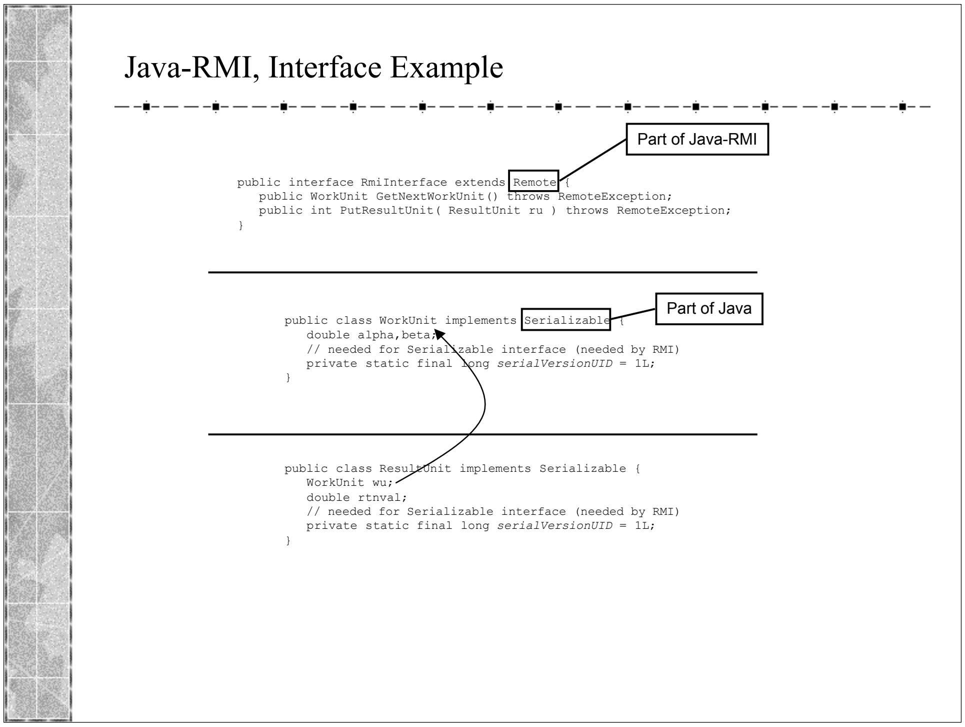
```
package ClientServer;
import java.rmi.*;
class Client {
   public static double DoCalculations( double alpha, double beta ) {
      double rtn;
      // do some highly complex calculations here
      rtn = (alpha+beta)*(alpha-beta);
      return( rtn );
   }
   public static void main( String[] argv ) {
      int SelfCID = Integer.parseInt(argv[0]);
      int NumClients = Integer.parseInt(argv[1]);
      String server_url = "//" + argv[2] + "/AlphaBetaServer";
      System.out.println( "Client "+SelfCID+" of "+NumClients+" starting" );
      try {
         RmiInterface server = (RmiInterface)Naming.lookup( server_url );
         for(int i=0;i<10;i++) {
            WorkUnit wu = server.GetNextWorkUnit();
            double alpha = wu.alpha;
            double beta = wu.beta;
            System.out.println( "Client"+SelfCID+" : alpha="+alpha+" beta="+beta );
            double rtn = DoCalculations( alpha, beta );
            System.out.println( "Client"+SelfCID+" : rtb="+rtn );
            ResultUnit ru = new ResultUnit();
            ru.wu = wu;
            ru.rtnval = rtn;
            server.PutResultUnit( ru );
         }
      }catch (Exception e) {
         System.out.println ("ComputeClient exception: " + e);
      }
   }
}
```



Java-RMI, Server Example

```
package ClientServer;
import java.rmi.*;
import java.rmi.server.*;
class Server extends UnicastRemoteObject implements RmiInterface {
   private double next_alpha = 0.0;
   private double next_beta  = 100.0;
   private double delta = 0.1;

   public Server( double a0, double b0, double dd ) throws RemoteException {
      next_alpha = a0;
      next_beta  = b0;
      delta = dd;
   }
   public synchronized WorkUnit GetNextWorkUnit() throws RemoteException {
      WorkUnit wu = new WorkUnit();
      wu.alpha = next_alpha;
      wu.beta  = next_beta;
      next_alpha += delta;
      next_beta  -= delta;
      return( wu );
   }
   public int PutResultUnit( ResultUnit ru ) throws RemoteException {
      // check that this is from a workunit we already processed
      if( ru.wu.alpha >= next_alpha ) {
         return( -1 );
      }
      // otherwise, we'll store the result somewhere
      System.out.println( "Result: a="+ru.wu.alpha+" b="+ru.wu.beta+" rtn="+ru.rtnval );
      return( 0 );
   }
   public static void main( String[] argv ) {
      try {
         Naming.rebind( "AlphaBetaServer", new Server(0.0,100.0,0.1) );
         System.out.println ("AlphaBetaServer is ready.");
      } catch (Exception e) {
         System.out.println ("AlphaBetaServer failed: " + e);
      }
   }
}
```



Java-RMI, Interface Example

Part of Java-RMI

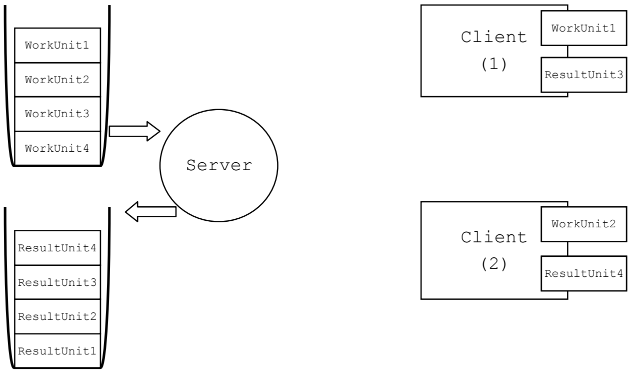
```
public interface RmiInterface extends Remote {
   public WorkUnit GetNextWorkUnit() throws RemoteException;
   public int PutResultUnit( ResultUnit ru ) throws RemoteException;
}
```

Part of Java

```
public class WorkUnit implements Serializable {
   double alpha,beta;
   // needed for Serializable interface (needed by RMI)
   private static final long serialVersionUID = 1L;
}
```

```
public class ResultUnit implements Serializable {
   WorkUnit wu;
   double rtnval;
   // needed for Serializable interface (needed by RMI)
   private static final long serialVersionUID = 1L;
}
```

What happens in the Example Code?



WorkUnit1
WorkUnit2
WorkUnit3
WorkUnit4
Server
ResultUnit4
ResultUnit3
ResultUnit2
ResultUnit1
Client (1)
WorkUnit1
ResultUnit3
Client (2)
WorkUnit2
ResultUnit4

Some Caveats

- The example shown here:
  - Does NOT provide verification or redundancy in calculations
  - Does NOT provide redundancy for hardware failures
  - You could LOSE some results if the network is flaky
    - or if one of the machines is powered off
- It is suitable for cluster computing environments
  - Hardware is reasonably stable
  - Unlikely that there are any “hackers” trying to crack your app
  - Not good for wide-area or grid computing environments

Building the Client

- The Client build is fairly straightforward
  - We need the WorkUnit, ResultUnit, and RmiInterface classes built first (and in that order)

```
jbp@hostname [ 12 ] % make Client.class
javac -classpath ../ WorkUnit.java
javac -classpath ../ ResultUnit.java
javac -classpath ../ RmiInterface.java
javac -classpath ../ Client.java
```

```
Directory structure:
./ClientServer/:
  WorkUnit.java
  ResultUnit.java
  RmiInterface.java
  Client.java
  Server.java
  makefile
```

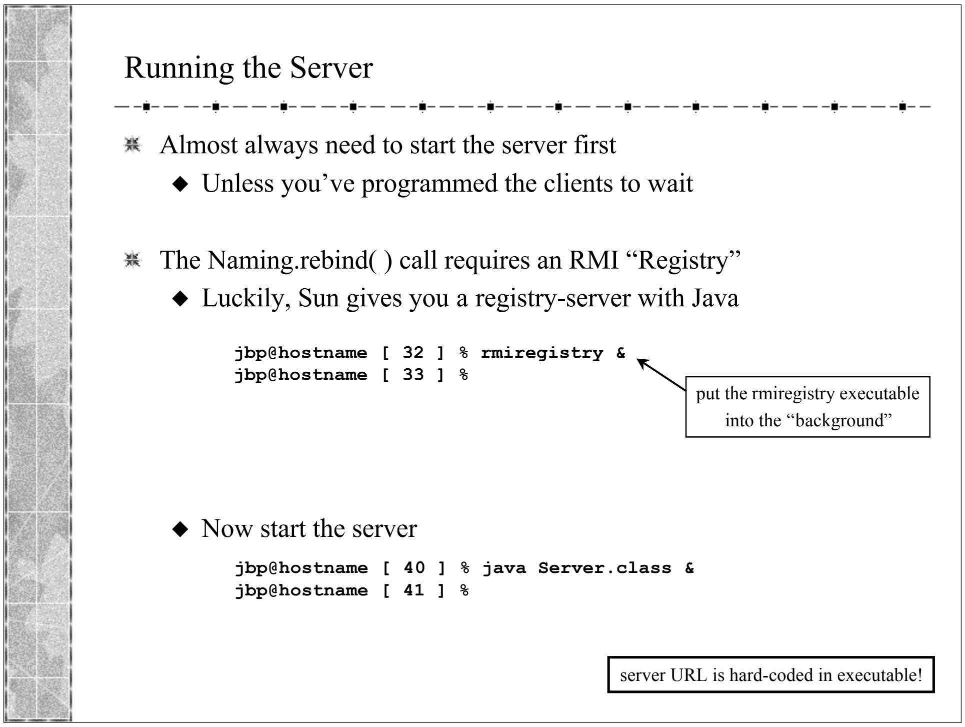
Building the Server

- The Server build is also fairly straightforward ... with one extra step
  - We need the WorkUnit, ResultUnit, and RmiInterface classes built first (and in that order)

```
jbp@hostname [ 22 ] % make Server.class
javac -classpath ../ Server.java
( cd .. ; rmic ClientServer.Server )
```

rmic is the RMI compiler that creates a server “stub” class.
The stub class holds all the networking bits.

```
jbp@hostname [ 36 ] % ls
Client.class         RmiInterface.class    Server_Stub.class
Client.java          RmiInterface.java     WorkUnit.class
ResultUnit.class     Server.class          WorkUnit.java
ResultUnit.java      Server.java           makefile
```



Running the Server

- Almost always need to start the server first
  - Unless you’ve programmed the clients to wait
- The Naming.rebind( ) call requires an RMI “Registry”
  - Luckily, Sun gives you a registry-server with Java

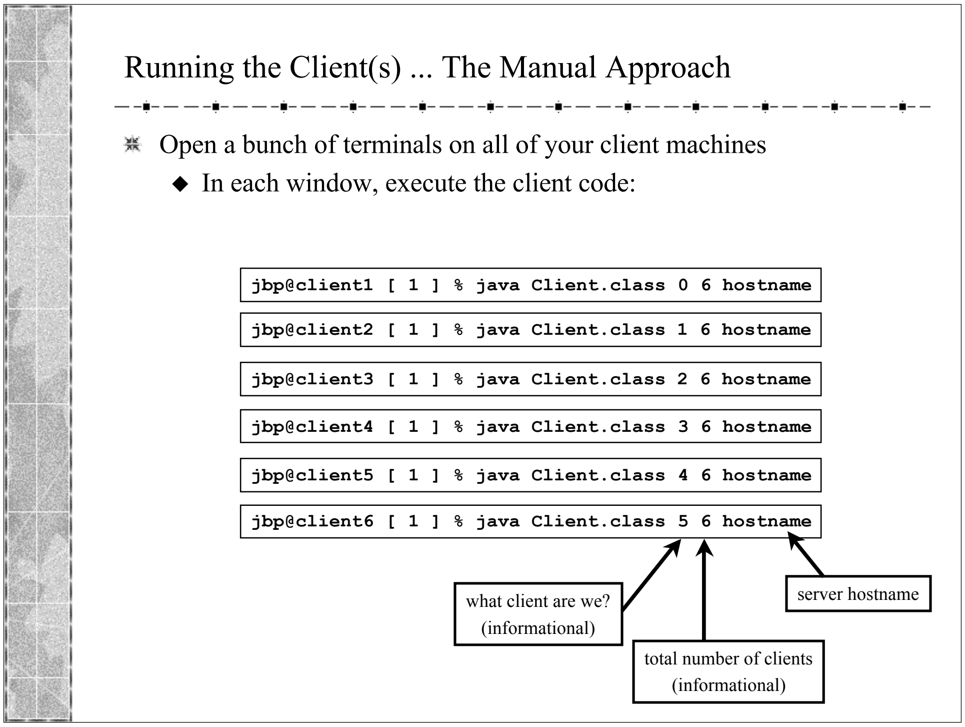
```
jbp@hostname [ 32 ] % rmiregistry &
jbp@hostname [ 33 ] %
```

put the rmiregistry executable into the “background”

  - Now start the server

```
jbp@hostname [ 40 ] % java Server.class &
jbp@hostname [ 41 ] %
```

server URL is hard-coded in executable!



Running the Client(s) ... The Manual Approach

- Open a bunch of terminals on all of your client machines
  - In each window, execute the client code:

```
jbp@client1 [ 1 ] % java Client.class 0 6 hostname
jbp@client2 [ 1 ] % java Client.class 1 6 hostname
jbp@client3 [ 1 ] % java Client.class 2 6 hostname
jbp@client4 [ 1 ] % java Client.class 3 6 hostname
jbp@client5 [ 1 ] % java Client.class 4 6 hostname
jbp@client6 [ 1 ] % java Client.class 5 6 hostname
```

what client are we? (informational)

total number of clients (informational)

server hostname

Running the Client and Server Together

- ‘rmirun’ is a home-grown Perl script that will launch the registry, one server, and (N-1) clients

```
jbp@hostname [ 1 ] % rmirun -n 8 Server.class Client.class
```

  - It uses ‘ssh’ to connect to the remote clients
    - Assumes that you can ssh to them without a password (ssh-keygen)
  - In a batch/queuing environment, like SGE, you can use ‘qrsh’ instead of ‘ssh’
    - Then rmirun will interact with the batch scheduler and automatically connect to less-loaded machines

```
#$ -cwd -o output.txt
#$ -pe low* 4-20
rmirun -S -n $NSLOTS -m $TMPDIR/machines Server.class Client.class
```

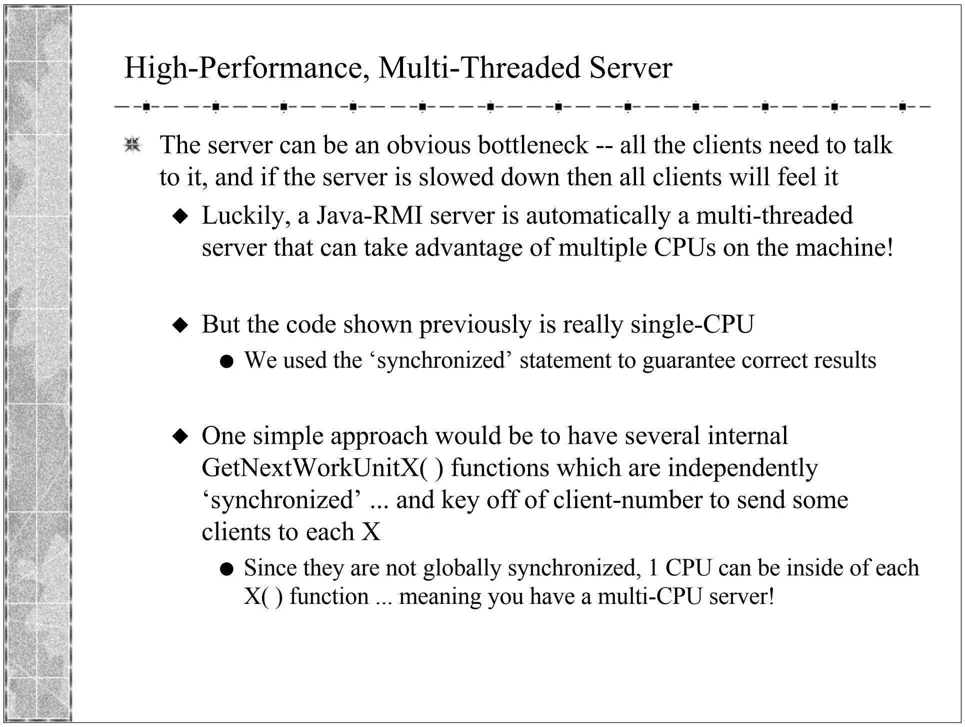
This is home-grown code, **NOT** part of Java-RMI

‘rmirun’ and Application Start-Up

- Recall that we added code to the clients to read some command-line arguments

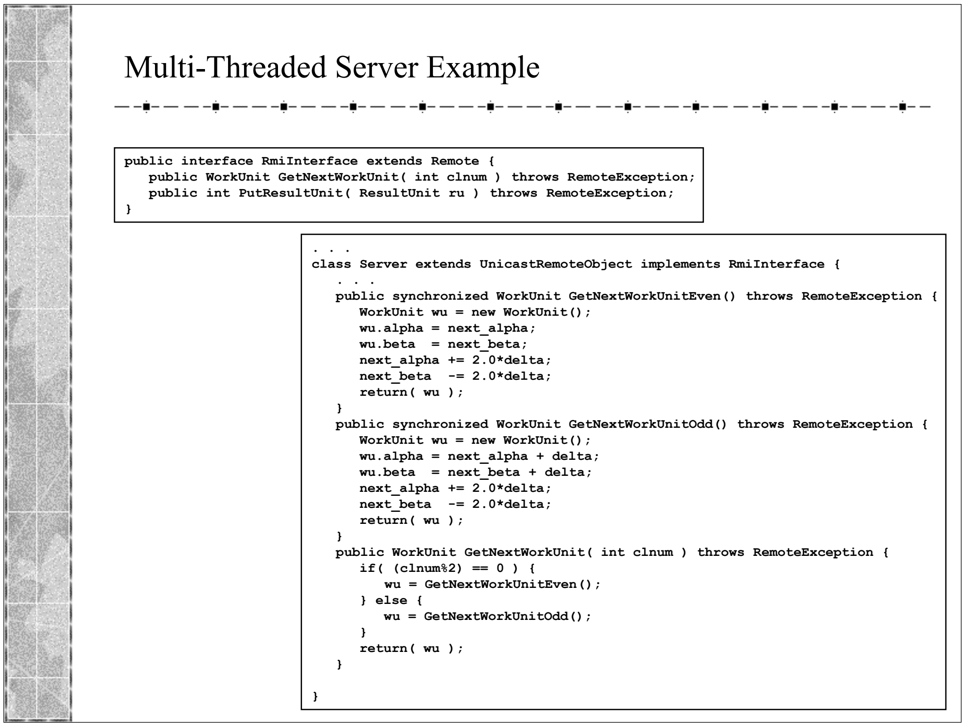
```
int SelfCID = Integer.parseInt(argv[0]);
int NumClients = Integer.parseInt(argv[1]);
String server_url = "//" + argv[2] + "/AlphaBetaServer";
```

  - None of this is really necessary
    - In a stand-alone config (in your own lab), you could hard-code the server URL directly into the client
      - But you’d have to recompile the clients just to move the server
    - In a cluster, the server could be scheduled onto any machine
  - The “MPI Programmer” in me wants to know how many clients we started and which one we are ... not really necessary
    - But it is useful during debugging, since otherwise all client error messages look the same
      - “client-0 error”, “client-1 error”, etc., is much more helpful



High-Performance, Multi-Threaded Server

- The server can be an obvious bottleneck -- all the clients need to talk to it, and if the server is slowed down then all clients will feel it
  - Luckily, a Java-RMI server is automatically a multi-threaded server that can take advantage of multiple CPUs on the machine!
  - But the code shown previously is really single-CPU
    - We used the ‘synchronized’ statement to guarantee correct results
  - One simple approach would be to have several internal GetNextWorkUnitX( ) functions which are independently ‘synchronized’ ... and key off of client-number to send some clients to each X
    - Since they are not globally synchronized, 1 CPU can be inside of each X( ) function ... meaning you have a multi-CPU server!

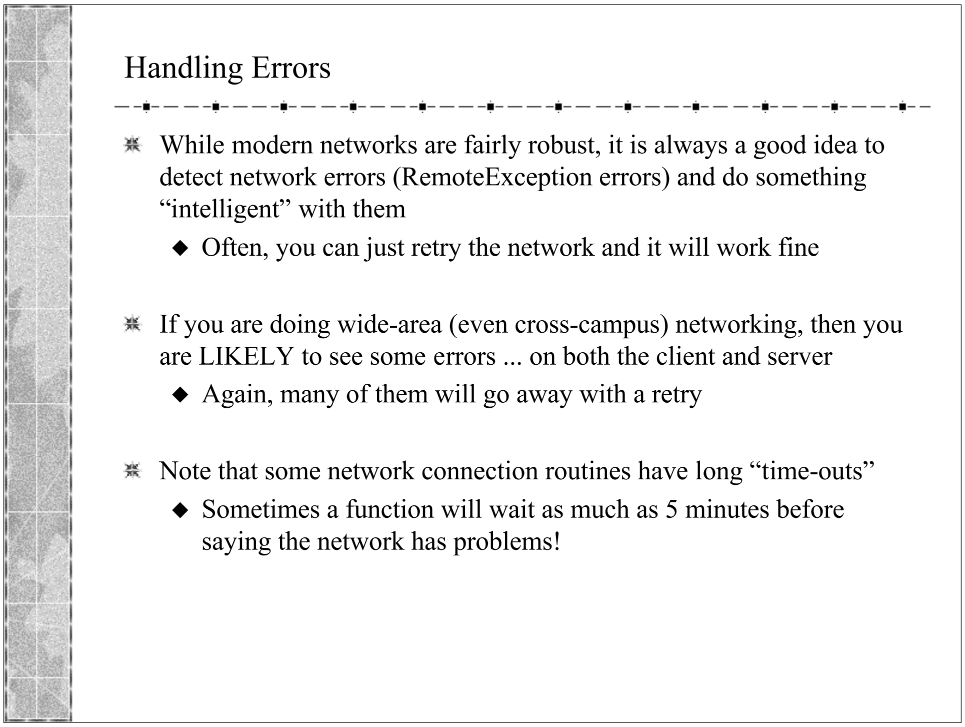


Multi-Threaded Server Example

```
public interface RmiInterface extends Remote {
   public WorkUnit GetNextWorkUnit( int clnum ) throws RemoteException;
   public int PutResultUnit( ResultUnit ru ) throws RemoteException;
}
```

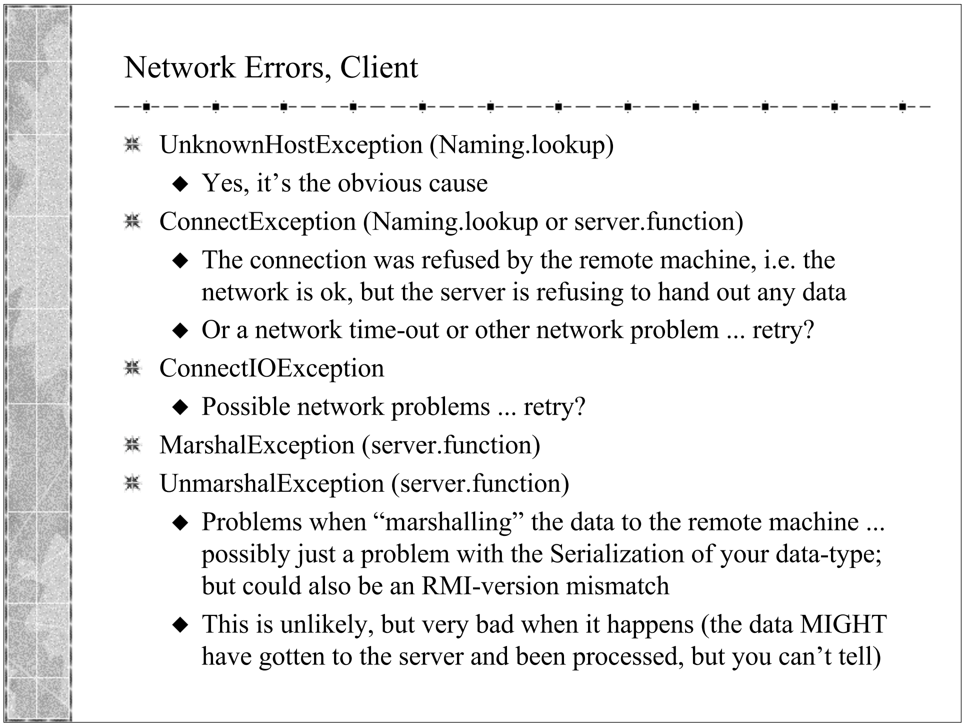
```
. . .
class Server extends UnicastRemoteObject implements RmiInterface {
   . . .
   public synchronized WorkUnit GetNextWorkUnitEven() throws RemoteException {
      WorkUnit wu = new WorkUnit();
      wu.alpha = next_alpha;
      wu.beta  = next_beta;
      next_alpha += 2.0*delta;
      next_beta  -= 2.0*delta;
      return( wu );
   }
   public synchronized WorkUnit GetNextWorkUnitOdd() throws RemoteException {
      WorkUnit wu = new WorkUnit();
      wu.alpha = next_alpha + delta;
      wu.beta  = next_beta + delta;
      next_alpha += 2.0*delta;
      next_beta  -= 2.0*delta;
      return( wu );
   }
   public WorkUnit GetNextWorkUnit( int clnum ) throws RemoteException {
      if( (clnum%2) == 0 ) {
         wu = GetNextWorkUnitEven();
      } else {
         wu = GetNextWorkUnitOdd();
      }
      return( wu );
   }

}
```



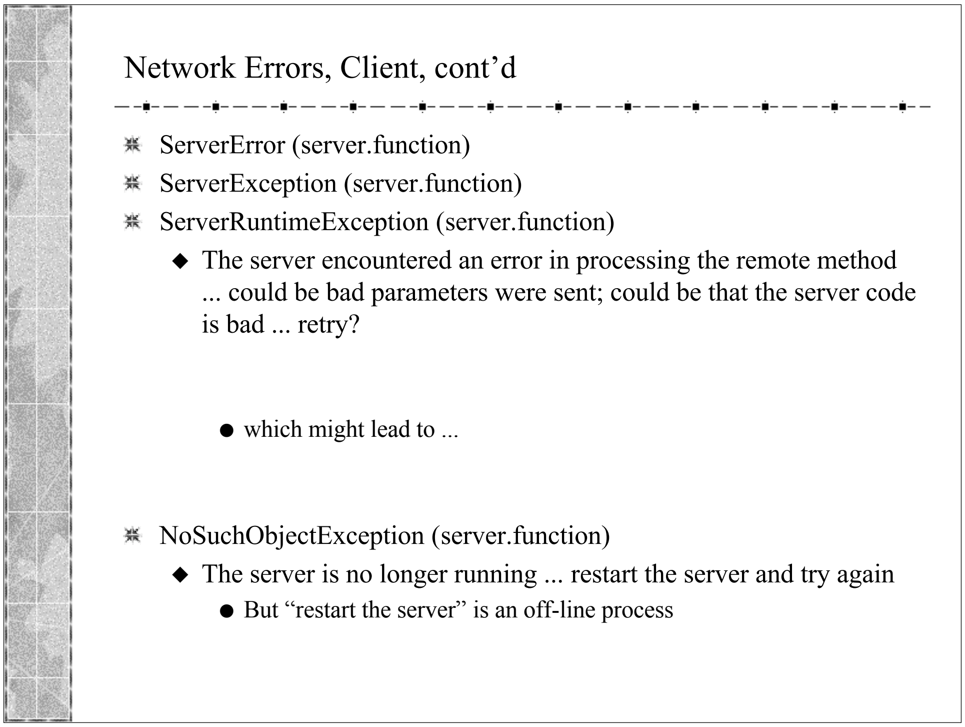
Handling Errors

- While modern networks are fairly robust, it is always a good idea to detect network errors (RemoteException errors) and do something “intelligent” with them
  - Often, you can just retry the network and it will work fine
- If you are doing wide-area (even cross-campus) networking, then you are LIKELY to see some errors ... on both the client and server
  - Again, many of them will go away with a retry
- Note that some network connection routines have long “time-outs”
  - Sometimes a function will wait as much as 5 minutes before saying the network has problems!



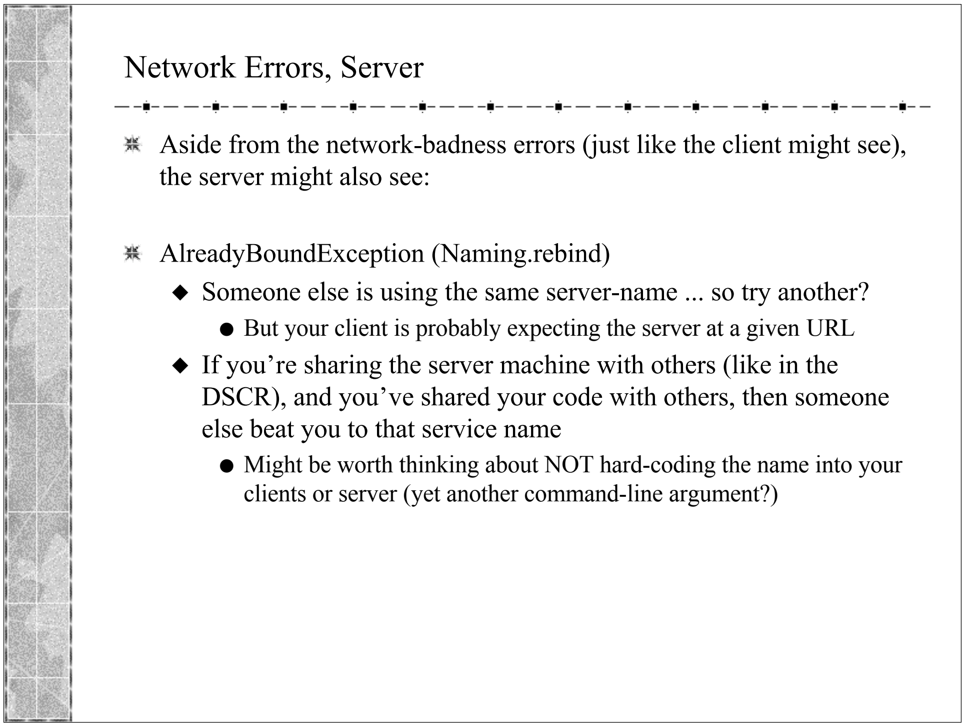
Network Errors, Client

- UnknownHostException (Naming.lookup)
  - Yes, it’s the obvious cause
- ConnectException (Naming.lookup or server.function)
  - The connection was refused by the remote machine, i.e. the network is ok, but the server is refusing to hand out any data
  - Or a network time-out or other network problem ... retry?
- ConnectIOException
  - Possible network problems ... retry?
- MarshalException (server.function)
- UnmarshalException (server.function)
  - Problems when “marshalling” the data to the remote machine ... possibly just a problem with the Serialization of your data-type; but could also be an RMI-version mismatch
  - This is unlikely, but very bad when it happens (the data MIGHT have gotten to the server and been processed, but you can’t tell)



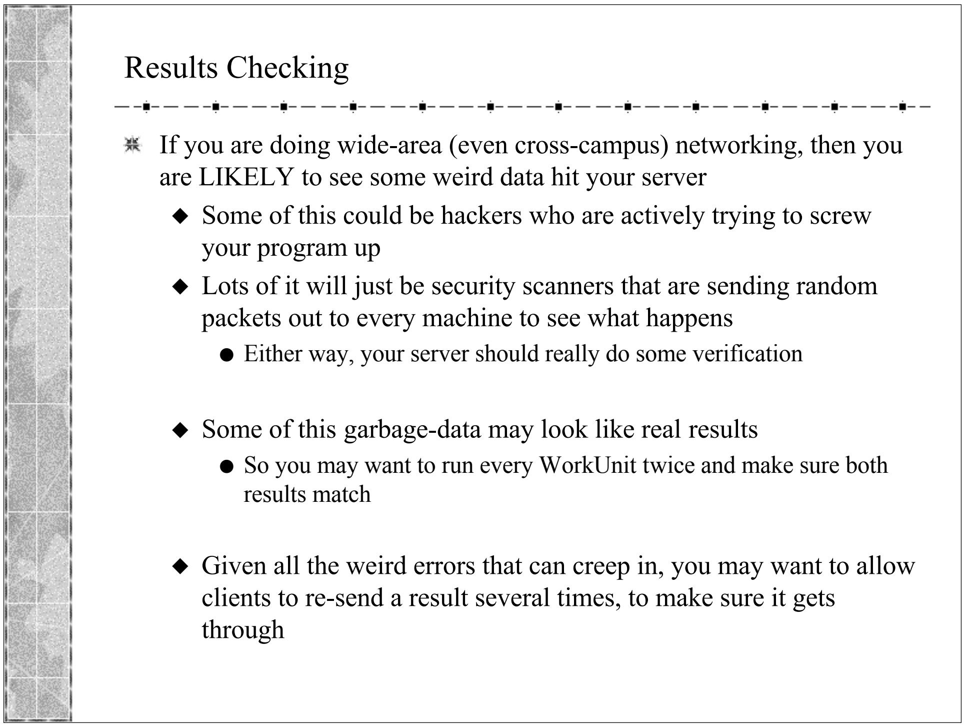
Network Errors, Client, cont’d

- ServerError (server.function)
- ServerException (server.function)
- ServerRuntimeException (server.function)
  - The server encountered an error in processing the remote method ... could be bad parameters were sent; could be that the server code is bad ... retry?
    - which might lead to ...
- NoSuchObjectException (server.function)
  - The server is no longer running ... restart the server and try again
    - But “restart the server” is an off-line process



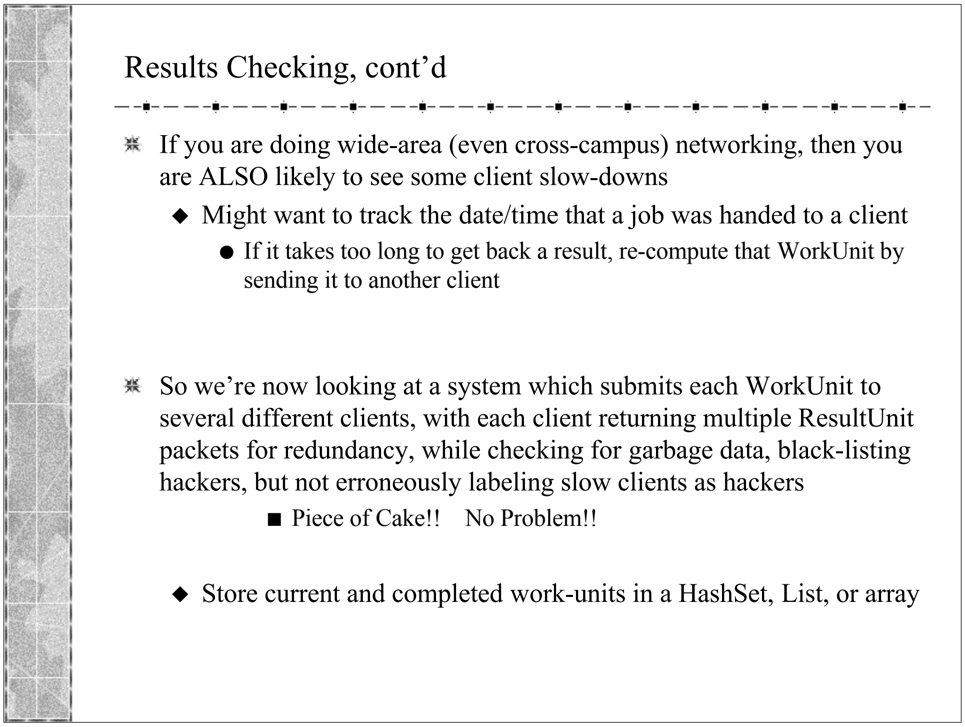
Network Errors, Server

- Aside from the network-badness errors (just like the client might see), the server might also see:
- AlreadyBoundException (Naming.rebind)
  - Someone else is using the same server-name ... so try another?
    - But your client is probably expecting the server at a given URL
  - If you’re sharing the server machine with others (like in the DSCR), and you’ve shared your code with others, then someone else beat you to that service name
    - Might be worth thinking about NOT hard-coding the name into your clients or server (yet another command-line argument?)



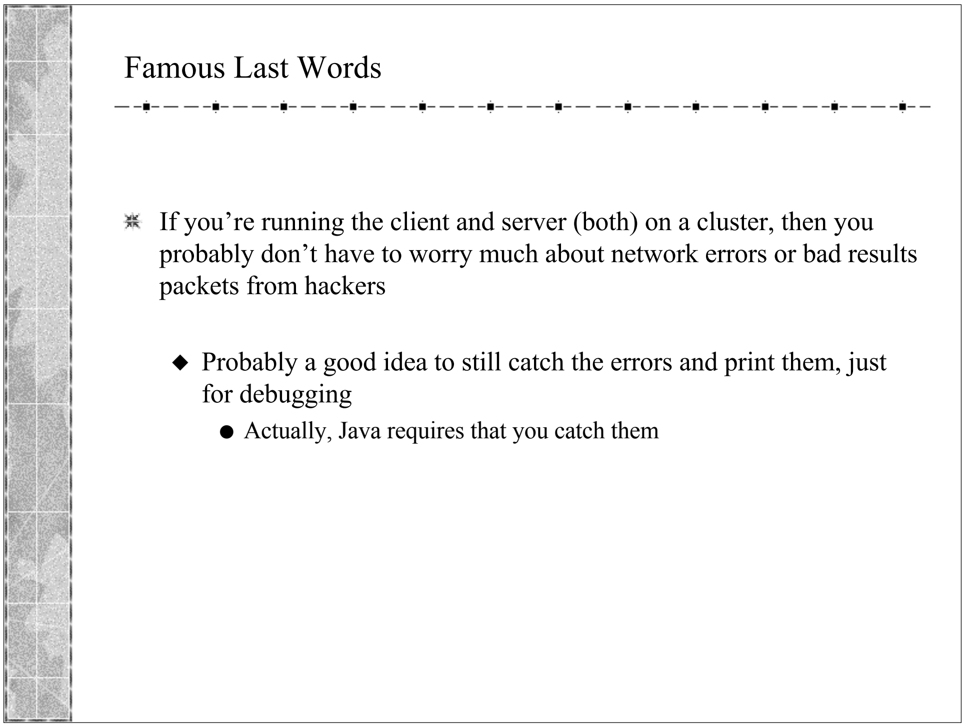
Results Checking

- If you are doing wide-area (even cross-campus) networking, then you are LIKELY to see some weird data hit your server
  - Some of this could be hackers who are actively trying to screw your program up
  - Lots of it will just be security scanners that are sending random packets out to every machine to see what happens
    - Either way, your server should really do some verification
  - Some of this garbage-data may look like real results
    - So you may want to run every WorkUnit twice and make sure both results match
  - Given all the weird errors that can creep in, you may want to allow clients to re-send a result several times, to make sure it gets through



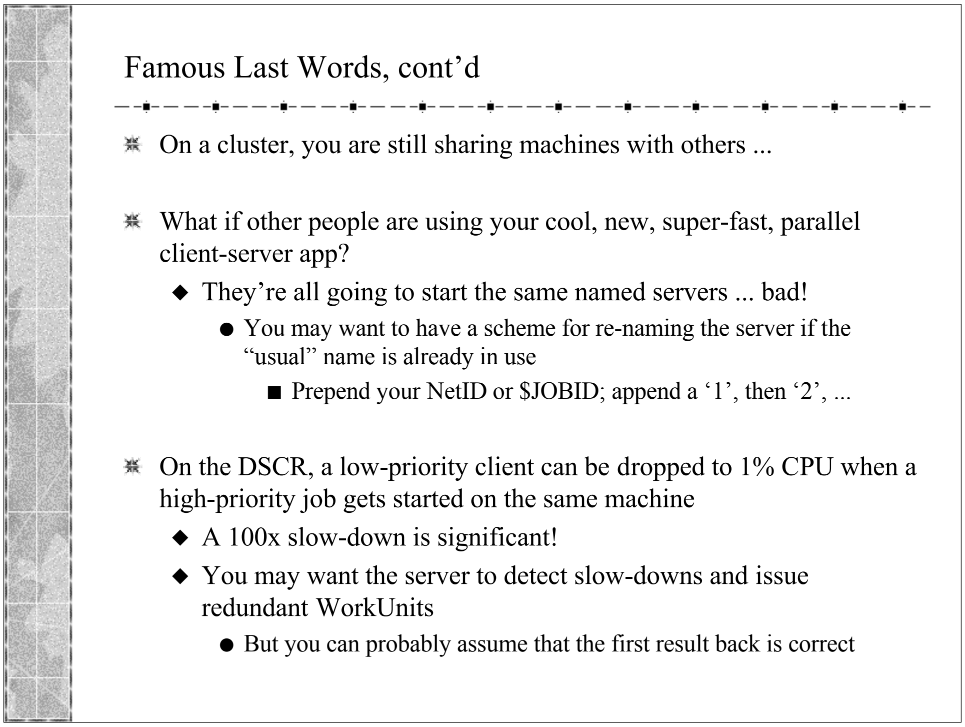
Results Checking, cont'd

- If you are doing wide-area (even cross-campus) networking, then you are ALSO likely to see some client slow-downs
  - Might want to track the date/time that a job was handed to a client
    - If it takes too long to get back a result, re-compute that WorkUnit by sending it to another client
- So we're now looking at a system which submits each WorkUnit to several different clients, with each client returning multiple ResultUnit packets for redundancy, while checking for garbage data, black-listing hackers, but not erroneously labeling slow clients as hackers
  - ■ Piece of Cake!! No Problem!!
  - Store current and completed work-units in a HashSet, List, or array



Famous Last Words

- If you’re running the client and server (both) on a cluster, then you probably don’t have to worry much about network errors or bad results packets from hackers
  - Probably a good idea to still catch the errors and print them, just for debugging
    - Actually, Java requires that you catch them



Famous Last Words, cont’d

- On a cluster, you are still sharing machines with others ...
- What if other people are using your cool, new, super-fast, parallel client-server app?
  - They’re all going to start the same named servers ... bad!
    - You may want to have a scheme for re-naming the server if the “usual” name is already in use
      - Prepend your NetID or $JOBID; append a ‘1’, then ‘2’, ...
- On the DSCR, a low-priority client can be dropped to 1% CPU when a high-priority job gets started on the same machine
  - A 100x slow-down is significant!
  - You may want the server to detect slow-downs and issue redundant WorkUnits
    - But you can probably assume that the first result back is correct

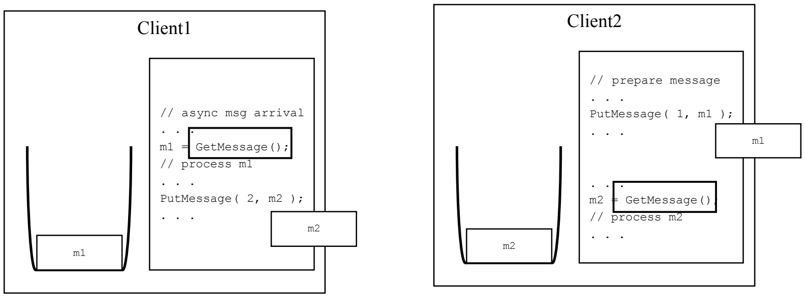
Next Steps

- Add a database back-end for job monitoring
  - Using a HastSet, List, or array means storing data in memory
    - So you could run out of memory on the server machine
  - Store the data to a database (file)
    - Java has support for using those database files as if they were “regular” HashSet, Maps, Lists, etc.
  - JDBC+SQLite
    - JDBC is a set of connector-functions to a “real” database
    - SQLite is a mostly-full-featured SQL server
      - Used in many production apps (Mac OS X)
    - Database will store data to disk
      - So server failure won’t be catastrophic

“Message Passing” with Java-RMI ??

- Java-RMI is more broad than just client-server
  - One could imagine each “client” also acting as a “server”

```
public interface RmiInterface extends Remote {
   public MessageUnit GetMessage() throws RemoteException;
   public int PutMessage( int other_cli, MessageUnit mu ) throws RemoteException;
}
```



```
// async msg arrival
. . .
m1 = GetMessage();
// process m1
. . .
PutMessage( 2, m2 );
. . .
```

```
// prepare message
. . .
PutMessage( 1, m1 );
. . .

. . .
m2 = GetMessage();
// process m2
. . .
```