RMI 2

Java Concurrency

1. Object locks
   - Synchronize on receiver
   - Structure is "static" -- not possible to write code with unbalanced acquire / release
   - Classically, this is known as a "monitor"
2. Wait / Notify system
   - Beware of dropped notify
3. Semaphore
   - Use for counting
   - Notice, structure is no longer static -- can write unbalanced code
4. One "GUI" lock for all GUI messages

Concurrency Conclusions

Challenge: future hardware will, effectively, have multiple processors, so explicitly threaded software will have the potential to run faster. However, writing explicitly threaded software is difficult.

Classical concurrency

GUI concurrency
- e.g. ThreadWeb -- high latency / networking type activities
- e.g. Camera download / decompress
- Theme: fast CPU connected to slow network, disk, camera, etc.

Snappy GUI Illusion

Elapsed time speed

Responsive "snappy" GUI speed -- keep the GUI thread separate

Visual progress feedback speed

NeXT 68030 machines -- pre-emptive GUI seemed fast vs. DOS / Macintosh of its day, though its hardware was lame.

RMI Example

FooRemote

Messages implemented by the server

-doit()

-install(Pipe)

FooServer

Implements the messages in FooRemote

A single instance is allocated

Calls naming.rebind() to start listening

FooClient

Uses Naming.lookup() to get a FooRemote
Sends the doit() message to the FooRemote (which tunnels over to the FooServer and executes against it)

PipeRemote

PipeServer / PipeRemote system : communication from PipeRemote -> PipeServer
Client allocates a PipeServer, and installs it over on the server side
In this case, the server has the stub and the client has the real one
When the server sends a message to its stub, it executes on the client

rtable.policy

Used by the java job on both client and server side

```
grant {
  permission java.net.SocketPermission "*:1024-65535","connect,accept";
  permission java.net.SocketPermission "*:80", "connect";
  permission java.awt.AWTPermission "*";
  permission java.lang.RuntimePermission "*";
  permission java.io.FilePermission "<<ALL FILES>>", "read,write";
};
```

Server Side

```
elaine32:~/java/rmi> rmiregistry 32456 &
[1] 23903
elaine32:~/java/rmi> rjava FooServer 32456
2001-05-15 09:49:24.465 FooServer: server bound
2001-05-15 09:49:33.1 FooServer: doit start
2001-05-15 09:49:33.105 FooServer: serverInternal
2001-05-15 09:49:33.106 FooServer: doit done
2001-05-15 09:49:33.392 FooServer: doit start
2001-05-15 09:49:33.392 FooServer: serverInternal
2001-05-15 09:49:33.416 FooServer: doit done
2001-05-15 09:49:38.11 FooServer: doit start
2001-05-15 09:49:38.112 FooServer: serverInternal
2001-05-15 09:49:38.112 FooServer: doit done
2001-05-15 09:49:38.347 FooServer: doit start
2001-05-15 09:49:38.347 FooServer: serverInternal
2001-05-15 09:49:38.362 FooServer: doit done
^C
```

Client Side

```
elaine39:~/java/rmi> rjava FooClient elaine32:32456
2001-05-15 09:49:33.003 Client: received from server -- hello there
2001-05-15 09:49:33.289 PipeServer: got message Server says hello
elaine39:~/java/rmi> rjava FooClient elaine32:32456
2001-05-15 09:49:38.006 Client: received from server -- hello there
2001-05-15 09:49:38.237 PipeServer: got message Server says hello
elaine39:~/java/rmi>
```

/ / FooRemote.java

```
// FooRemote.java
// The interface exposed by the FooServer -- the client
// sends these on the client end, and they happen on the server
// end.

import java.rmi.*;
import java.rmi.server.*;

public interface FooRemote extends java.rmi.Remote {
   // Run the doit() operation on the server
   public String[] doit() throws RemoteException;

   // Send a pipe over to the server
   public void install(PipeRemote pipe) throws RemoteException;

   public static final String SERVICE = "nickFoo";
}
```

/ / FooClient.java

```
// FooClient.java

import java.rmi.*;
import java.math.*;

public class FooClient {
   public static void main(String args[]) {
      if (System.getSecurityManager() == null) {
         System.setSecurityManager(new RMISecurityManager());
      }
      try {
         String name = "//" + args[0] + "/" + FooRemote.SERVICE;

         // Get the stub for the server object
         FooRemote foo = (FooRemote) Naming.lookup(name);

         // Send the server a message
         String[] result  = (String[]) foo.doit();
         Log.print("Client: received from server -- " + result[0] + result[1]);


         // Create a pipe here and send it to the server
         PipeRemote pipe = new PipeServer();
         foo.install(pipe);

         // This will call us back on the pipe
         foo.doit();


         // Remove their ref back to us, so we can exit
         foo.install(null);

         System.exit(0);

      } catch (Exception e) {
```

```
         System.err.println("FooClient exception: " +
            e.getMessage());
         e.printStackTrace();
      }

      Log.print("Client: done");
   }
}
```

/ / FooServer.java

```
// FooServer.java
// Demonstrates RMI
// The client invokes doit() which runs on the server
// The client can send us a pipe object, which we use
// to send objects back to the client

import java.rmi.*;
import java.rmi.server.*;


public class FooServer extends UnicastRemoteObject
      implements FooRemote
{
   PipeRemote pipe;

    public FooServer() throws RemoteException {
      super();
      pipe = null;
    }

    public String[] doit() throws RemoteException {
      Log.print("FooServer: doit start");

      serverInternal();

      if (pipe != null) pipe.send("Server says hello");

      // We'll try this later -- send executable content
      // back to the client
      // if (pipe != null) pipe.sendRunnable(new MyRunnable());

      // Make a little array to return
      // (arrays and strings automatically serializable)
      String[] result = new String[2];
      result[0] = "hello";
      result[1] = " there";

      Log.print("FooServer: doit done");
      return(result);
    }

    private void serverInternal() {
      Log.print("FooServer: serverInternal");
    }
```

```
   // Sent by the client to give us a pipe
   public void install(PipeRemote pipe) throws RemoteException {
      this.pipe = pipe;
   }


   public static void main(String[] args) {
      if (args.length != 1) {
         System.out.println("Need port number");
         System.exit(0);
      }

      if (System.getSecurityManager() == null) {
         System.setSecurityManager(new RMISecurityManager());
      }
      String name = "//localhost:" + args[0] +
         "/" + FooRemote.SERVICE;
      try {
         FooRemote impl = new FooServer();
         Naming.rebind(name, impl);
         Log.print("FooServer: server bound");
      } catch (Exception e) {
         System.err.println("FooServer exception: " + e.getMessage());
         e.printStackTrace();
      }
   }
}

// A separate runnable object. We create one of these and send
// back to the client on the pipe. It runs on the client.
class MyRunnable implements Runnable, java.io.Serializable {
   public void run() {
      int i = 1;
      i = i + 1;
      System.out.println("This code sent from the server " + i);
   }
}
```

Running

```
--Build

1) Build with javac *.java
2) rmic FooServer PipeServer

--Run
alias rjava "java -Djava.security.policy=rtable.policy"

server% rmiregistry 32456 &      // run the registry
server% rjava FooServer 32456    // start the server

client% rjava FooClient elaine33:32456 // or whatever the server host is


The machinery here will send the MyRunnable
through the filesystem -- some additional steps
are required with the Class Loader so that
MyRunnable.class is sent over the RMI network.
```

/ / PipeRemote.java

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

// A creates one of these and sends it to B.
// B can then invoke the send() operation on theirs,
// and it is received on the instance held by A.
// Sending a runnable sends code back to A -- the object
// must be serializable.

public interface PipeRemote extends java.rmi.Remote {
   public void send(String message) throws RemoteException;
   public void sendRunnable(Runnable runnable) throws RemoteException;
}
```

/ / PipeServer.java

```
// PipeServer.java
import java.rmi.*;
import java.rmi.server.*;

public class PipeServer extends UnicastRemoteObject
   implements PipeRemote {

   public PipeServer() throws RemoteException {
      super();
   }

   public void send(String message) throws RemoteException {
      Log.print("PipeServer: got message " + message);
   }

   public void sendRunnable(Runnable runnable) throws RemoteException {
      Log.print("PipeServer: got runnable");
      // Run the thing they sent us in its own thread,
      // or could just call runnable.run()
      Thread thread = new Thread(runnable);
      thread.start();
   }
}
```

/ / Log.java

```
// Log.java -- print messages with the current time
import java.util.*;
import java.sql.*;
public class Log {
   public static void print(String string) {
      java.util.Date date = new java.util.Date();  // now
      Timestamp time = new Timestamp(date.getTime());
      System.out.println(time.toString() + " " + string);
   }
}
```