Designing RMI Applications

José M. Vidal

Wed Mar 17 09:42:52 EST 2004

We present some of the ideas from:

- William Grosso, Java RMI[1], 2002. Chapters 5-10, 12-17, 19-20, 22.

Code examples were downloaded from and belong to O'Reilly[2].

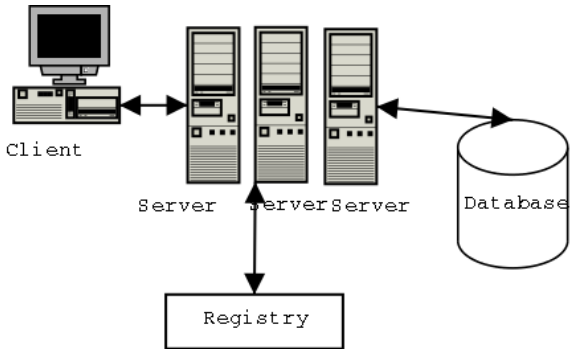
1 Sketch a Rough Architecture

- The first step in designing a distributed system is to sketch a rough architecture of your system.

1. Figure out what you are going to build. Do your requirements analysis.
2. Find a basic use case that will motivate the rough architecture.
3. Figure out what you can safely ignore for now (often scalability and security).
4. Determine which design decisions are forced by the environment.
5. Narrow down the servers to as few as possible.

2 Bank Example

- We will use as an example the construction of an ATM application.
- The basic use case is
  1. User walks up to ATM and enters password.
  2. If password is correct user is allowed to make transactions until card is removed.
  3. User is given choices: get balance, withdraw money, deposit money, transfer money.
  4. After choosing an action the user is given a list of valid accounts for that choice.
  5. After a few transactions the user leaves.
- The environmental constraints are that we must use the legacy database and the ATMs are physically distributed.



- **Client** responsible for managing interaction with user, via GUI.
- **Stub** implicitly handles connection.
- **Registry** maintains mapping of human-readable names to server stubs and returns them when asked.
- **Skeletons and launch code** do the work on the database.
- **Servers** handle the business logic by running the skeletons, communicate with database.
- **Database system** is responsible for long-term persistence and integrity of important data.

3 Two Choices

- We will consider two possible design choices. Which one is better?

```
//One instance of Bank for all clients
class Bank {
  public Money getBalance(Account account) throws RemoteException;
  public void makeDeposit(Account account)
    throws RemoteException, NegativeAmountException;
  public void makeWithdrawal(Account account, Money amount)
    throws RemoteException, OverdraftException, NegativeAmountException;
}

//One instance of Account for each account
class Account {
  public Money getBalance() throws RemoteException;
  public void makeDeposit(Money amount)
    throws RemoteException, NegativeAmountException;
  public void makeWithdrawal(Money amount)
    throws RemoteException, OverdraftException, NegativeAmountException;
}
```

4 Choosing

- Does each instance of the server require a shared or scarce resource?
- How well does the server replicate or scale to multiple machines?
- Can a single server handle a typical client interaction?
- How easy is it to make a server handle multiple simultaneous clients?
- How easy is it to tell whether the code is correct?
- How fatal is server failure?
- How easy is it to add new functionality?
- So, `Account` seems to be the preferred implementation for this problem

4.1 Does Each Instance Require Shared Resource?

- Memory, in general, is not an issue because the amount needed is linear with the number of clients active. Also, we can use a factory if needed.
- Sockets are not a problem because RMI re-uses sockets between two JVMs, even if different objects (thanks to `RemoteRef`).
- The only possibility might be a log file, but then both choices would have to share one logfile.
- *Advantage: neither*

4.2 How Well Server Scales or Replicates to Multiple Machines?

- Applicable when our needs grow beyond one JVM.
- We can think of a load-balancing server which distributes clients among a set of machines. We would need to divide our objects among the machines.
- The `Account` is easy. Since there are many of them we can re-write our launcher to register some of them with one machine, some with another, and so on.
- The `Bank` is very hard to spread out since two clients with two Banks (from different servers) could be trying to access the same account.
- *Advantage: Account*

4.3 Is Single Server Enough?

- If the client accesses multiple accounts serially its not a problem because RMI shares sockets, so the costs are the same in either case.
- If the client does a transfer from one account to another the Account classes will have trouble (it will be hard to implement) performing the single database operation needed.
- *Advantage: Bank*

4.4 How to Handle Multiple Simultaneous Clients?

- The smaller and simpler the server is, the easier it is to make it safely handle multiple clients.
- *Advantage: Account*

4.5 Is Code Correct?

- In general, the smaller the server (code) the easier to tell whether it is correct.
- *Advantage: Account*

4.6 How Fatal is Server Crash?

- If Bank crashes, all clients are affected.
- If an Account crashes, only that client is affected.
- *Advantage: Account*

4.7 How Easy Adding Functionality?

- Requirements are constantly changing.
- Smaller server are easier to modify and extend.
- *Advantage: Account*

5 Writing the Interface

- Should we pass method objects?
- Should we pass objects as arguments or use primitive values?
- Should we receive objects as return values or receive primitive values?
- Do individual method calls waste bandwidth?
- Is each conceptual operation a single method call?
- Does the interface expose the right amount of metadata?
- Have we identified a reasonable set of distributed exceptions?

5.1 Should We Pass Method Objects?

```
public interface Account extends Remote {
  public Money getBalance() throws RemoteException;
  public void postTransaction(Transaction transaction)
    throws RemoteException, TransactionException;
}
```

- Passing a method object will allow the addition of new functionality without changing the interface.
- Using methods lets the compiler catch more errors.
- Using methods makes the code easier to read.
- Using methods allows us to introduced focused exceptions.

5.2 Pass Objects As Arguments or Use Primitive Values?

- Objects are bigger than primitive values.
- Adding another property (data member) to an existing object is easy. The only code that needs modification is the object.
- Using the `Money` class seems a bit silly since we end up just sending an integer number of cents.
- However, there is an implicit assumption that we are talking about dollars. To make our ATM work across borders we will need the `Money` class.

5.3 Return Values: Objects or Primitive Values?

- The reason for returning objects is that you only get to return one thing.
- Remember, no call-by-reference of non-remote types with remote objects.
- You should usually prefer to return objects, or nothing.
- boolean return values are a possible exception. But, first consider if there is something else that needs returning and if exceptions might be a better solution (they usually are, but people are lazy and don't want to write another class).

5.4 Do Individual Method Calls Waste Bandwidth?

- Serialization makes deep copies. This can lead to many objects being serialized and sent over.
- Object types that can be of arbitrary size (e.g., a `Vector` or `ArrayList`) are usually a bad choice for a return type.
  - They degrade perceived client performance.
  - They increase performance variance.
  - They involve a large, all-at-once network hit.
  - They involve a larger single-client commitment on the part of the server.
  - They penalize client mistakes.

5.5 Is Each Conceptual Operation A Single Method Call?

- An interface that is good for a local object because it has many small methods might be awful for a remote object because method calls are expensive.
- For a remote object try to make a method call for each "usage" of the object by grouping together smaller methods into one.
- If implementing an iterator, it should have an argument that tells it how many to fetch at a time. These would be stored in the local iterator object and handed out one-by-one with `getNext` call.

5.6 Does The Interface Expose The Right Amount Of Metadata?

- **Functional methods** make the server do something.
- **Descriptive methods** give more information about the server.
- Descriptive methods are useful because they allow the client to validate that the server can handle an operation, and they help the client when choosing from among various servers.
- Our Account interface could give information about the owner, account number, type of account, etc.

5.7 Have We Identified A Reasonable Set Of Distributed Exceptions?

- You want to return all the information the client needs to proceed.
- In the bank example, could try to withdraw more money than he has, or he could try to withdraw a negative amount.
- We have handled these two cases explicitly with `OverdraftException` and `NegativeAmountException`.

- `OverdraftException` can also tell if the withdrawal succeeded.

```
public class OverdraftException extends Exception {
  public boolean withdrawalSucceeded;
  public OverdraftException(boolean withrawalSucceeded){
    this.withrawalSucceeded  = withrawalSucceeded;
  }
  public boolean didWithdrawalSucceed(){
    return withrawalSucceeded;
  }
}
```

- We probably also need exception for when a null is sent as argument, and when the argument is exactly 0.
- In general, the server should throw as specific exceptions as possible, while the client will probably only catch the higher-level exceptions.

6 Building Data Objects

- Once you have the interfaces, the data objects should be easy to discern.
- Data objects don't have many functional methods, mostly just descriptive (getX) methods.

```
import java.io.*;


public class Money extends ValueObject {
  protected int _cents;

  public Money(Integer cents) {
    this (cents.intValue());
  }

  public Money(int cents) {
    super (cents + " cents.");
    _cents = cents;
  }

  public int getCents() {
    return _cents;
  }

  public void add(Money otherMoney) {
    _cents += otherMoney.getCents();
  }

  public void subtract(Money otherMoney) {
    _cents -= otherMoney.getCents();
  }

  public boolean greaterThan(Money otherMoney) {
    if (_cents > otherMoney.getCents()) {
```

```
      return true;
    }
    return false;
  }

  public boolean isNegative() {
    return _cents < 0;
  }

  public boolean equals(Object object) {
    if (object instanceof Money) {
      Money otherMoney = (Money) object;

      return (_cents == otherMoney.getCents());
    }
    return false;
  }
}
```

7 Server Implementation

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

public class Account_Impl extends UnicastRemoteObject implements Account {
  private Money _balance;
  public Account_Impl(Money startingBalance)
    throws RemoteException {
    _balance = startingBalance;
  }

  public Money getBalance()
    throws RemoteException {
    return _balance;
  }

  public void makeDeposit(Money amount)
    throws RemoteException, NegativeAmountException {
    checkForNegativeAmount(amount);
    _balance.add(amount);
    return;
  }

  public void makeWithdrawal(Money amount)
    throws RemoteException, OverdraftException, NegativeAmountException {
    checkForNegativeAmount(amount);
    checkForOverdraft(amount);
    _balance.subtract(amount);
    return;
  }

  private void checkForNegativeAmount(Money amount)
    throws NegativeAmountException {
```

```
    int cents = amount.getCents();

    if (0 > cents) {
      throw new NegativeAmountException();
    }
  }

  private void checkForOverdraft(Money amount)
    throws OverdraftException {
    if (amount.greaterThan(_balance)) {
      throw new OverdraftException(false);
    }
    return;
  }
}
```

- Extends `UnicastRemoteObject`

8 Server Implementation 2

```
import java.rmi.server.*;
/*
 The only difference between this and Account_Impl is that
 Account_Impl extends UnicastRemote.
 */
public class Account_Impl2 implements Account {
  private Money _balance;
  public Account_Impl2(Money startingBalance)
    throws RemoteException {
    _balance = startingBalance;
  }

  public Money getBalance()
    throws RemoteException {
    return _balance;
  }

  public void makeDeposit(Money amount)
    throws RemoteException, NegativeAmountException {
    checkForNegativeAmount(amount);
    _balance.add(amount);
    return;
  }

  public void makeWithdrawal(Money amount)
    throws RemoteException, OverdraftException, NegativeAmountException {
    checkForNegativeAmount(amount);
    checkForOverdraft(amount);
    _balance.subtract(amount);
    return;
  }
```

```
  /** We must define this function */
  public boolean equals(Object object) {
    // three cases. Either it's us, or it's our stub, or it's
    // not equal.

    if (object instanceof Account_Impl2) {
      return (object == this);
    }
    if (object instanceof RemoteStub) {
      try {
        RemoteStub ourStub = (RemoteStub) RemoteObject.toStub(this);

        return ourStub.equals(object);
      } catch (NoSuchObjectException e) {
        // we're not listening on a port, therefore it's not our
        // stub
      }
    }
    return false;
  }

  /** We must define this function */
  public int hashCode() {
    try {
      Remote ourStub = RemoteObject.toStub(this);

      return ourStub.hashCode();
    } catch (NoSuchObjectException e) {
    }
    return super.hashCode();
  }

  private void checkForNegativeAmount(Money amount)
    throws NegativeAmountException {
    int cents = amount.getCents();

    if (0 > cents) {
      throw new NegativeAmountException();
    }
  }

  private void checkForOverdraft(Money amount)
    throws OverdraftException {
    if (amount.greaterThan(_balance)) {
      throw new OverdraftException(false);
    }
    return;
  }
}
```

- We must call `exportObject` after creating one of these.
- We must implement `equals()` and `hashCode()`. These are tricky to implement. The code here gets around that by creating a stub of itself and letting that stub to the work.

- If you want to implement a remote object that must extend another class, but still want the ease of extending `UnicastRemoteObject`, you can use a **tie server**.
- A **tie server** extends `UnicastRemoteObject` and implements the remote interface. The implementation, however, simply forwards all method calls to the real server (cf. Adapter pattern[3]).
- The real server can then extend some other class.

9 Launch Code

- When should servers be launched?
- When should servers be shutdown?
- When should server save their state to a persistent store?
- The launch script looks like:

```
start rmiregistry
start java -Djava.security.manager -Djava.security.policy="d:
textbackslashjava.policy"
  com.ora.rmibook.chapter9.applications.ImplLauncher Bob 10000 Alex 1223
```

- The actual launcher is:

```
public class ImplLauncher {
  public static void main(String[] args) {
    Collection nameBalancePairs = getNameBalancePairs(args);
    Iterator i = nameBalancePairs.iterator();

    while (i.hasNext()) {
      NameBalancePair nextNameBalancePair = (NameBalancePair) i.next();

      launchServer(nextNameBalancePair);
    }
  }

  private static void launchServer(NameBalancePair serverDescription) {
    try {
      Account_Impl newAccount = new Account_Impl(serverDescription.balance);

      Naming.rebind(serverDescription.name, newAccount);
      System.out.println("Account " + serverDescription.name + " successfully launched.");
    } catch (Exception e) {
    }
  }

  private static Collection getNameBalancePairs(String[] args) {
    int i;
    ArrayList returnValue = new ArrayList();

    for (i = 0; i < args.length; i += 2) {
      NameBalancePair nextNameBalancePair = new NameBalancePair();
```

```
      nextNameBalancePair.name = args[i];
      int cents = (new Integer(args[i + 1])).intValue();

      nextNameBalancePair.balance = new Money(cents);
      returnValue.add(nextNameBalancePair);
    }
    return returnValue;
  }

  private static class NameBalancePair {
    String name;
    Money balance;
  }
}
```

10 Building the Client

- Client should consume as few server resources as possible, for as short a time as is reasonable.
- Don't host connections to a server you are not using. Set that remote reference to null as soon as possible.
- Validate arguments on the client side whenever possible. Increase response time. Reduce server load.
- Most of the client is usually GUI code.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.rmi.*;

public class BankClient {
  public static void main(String[] args) {
    (new BankClientFrame()).show();
  }
}

public class BankClientFrame extends ExitingFrame {
  private JTextField _accountNameField;
  private JTextField _balanceTextField;
  private JTextField _withdrawalTextField;
  private JTextField _depositTextField;
  private Account _account;

  protected void buildGUI() {
    JPanel contentPane = new JPanel(new BorderLayout());

    contentPane.add(buildActionPanel(), BorderLayout.CENTER);
    contentPane.add(buildBalancePanel(), BorderLayout.SOUTH);
    setContentPane(contentPane);
    setSize(250, 100);
  }
```

```
private void resetBalanceField() {
  try {
    Money balance = _account.getBalance();

    _balanceTextField.setText("Balance: " + balance.toString());
  } catch (Exception e) {
    System.out.println("Error occurred while getting account balance\n" + e);
  }
}

private JPanel buildActionPanel() {
  JPanel actionPanel = new JPanel(new GridLayout(3, 3));

  actionPanel.add(new JLabel("Account Name:"));
  _accountNameField = new JTextField();
  actionPanel.add(_accountNameField);
  JButton getBalanceButton = new JButton("Get Balance");

  getBalanceButton.addActionListener(new GetBalanceAction());
  actionPanel.add(getBalanceButton);
  actionPanel.add(new JLabel("Withdraw"));
  _withdrawalTextField = new JTextField();
  actionPanel.add(_withdrawalTextField);
  JButton withdrawalButton = new JButton("Do it");

  withdrawalButton.addActionListener(new WithdrawAction());
  actionPanel.add(withdrawalButton);
  actionPanel.add(new JLabel("Deposit"));
  _depositTextField = new JTextField();
  actionPanel.add(_depositTextField);
  JButton depositButton = new JButton("Do it");

  depositButton.addActionListener(new DepositAction());
  actionPanel.add(depositButton);
  return actionPanel;
}

private JPanel buildBalancePanel() {
  JPanel balancePanel = new JPanel(new GridLayout(1, 2));

  balancePanel.add(new JLabel("Current Balance:"));
  _balanceTextField = new JTextField();
  _balanceTextField.setEnabled(false);
  balancePanel.add(_balanceTextField);
  return balancePanel;
}

private void getAccount() {
  try {
    _account = (Account) Naming.lookup(_accountNameField.getText());
  } catch (Exception e) {
    System.out.println("Couldn't find account. Error was \n " + e);
    e.printStackTrace();
  }
```

```
  return;
}

private void releaseAccount() {
  _account = null;
}

private Money readTextField(JTextField moneyField) {
  try {
    Float floatValue = new Float(moneyField.getText());
    float actualValue = floatValue.floatValue();
    int cents = (int) (actualValue * 100);

    return new PositiveMoney(cents);
  } catch (Exception e) {
    System.out.println("Field doesn't contain a valid value");
  }
  return null;
}

private class GetBalanceAction implements ActionListener {
  public void actionPerformed(ActionEvent event) {
    try {
      getAccount();
      resetBalanceField();
      releaseAccount();
    } catch (Exception exception) {
      System.out.println("Couldn't talk to account. Error was \n " + exception);
      exception.printStackTrace();
    }
  }
}


private class WithdrawAction implements ActionListener {
  public void actionPerformed(ActionEvent event) {
    try {
      getAccount();
      Money withdrawalAmount = readTextField(_withdrawalTextField);

      _account.makeWithdrawal(withdrawalAmount);
      _withdrawalTextField.setText("");
      resetBalanceField();
      releaseAccount();
    } catch (Exception exception) {
      System.out.println("Couldn't talk to account. Error was \n " + exception);
      exception.printStackTrace();
    }
  }
}


private class DepositAction implements ActionListener {
  public void actionPerformed(ActionEvent event) {
```

```
      try {
        getAccount();
        Money depositAmount = readTextField(_depositTextField);

        _account.makeDeposit(depositAmount);
        _depositTextField.setText("");
        resetBalanceField();
        releaseAccount();
      } catch (Exception exception) {
        System.out.println("Couldn't talk to account. Error was \n " + exception);
        exception.printStackTrace();
      }
    }
  }
}
```

11 Serialization

- In Java we **serialization** refers to the process of turning an object into something that can be written to a stream.
- More generally, we call it **marshalling**.

11.1 Using Serialization

- You can place any object into a stream by simply

  ```
  FileOutputStream underlyingStream = new FileOutputStream("/tmp/file");
  ObjectOutputStream serializer = new ObjectOutputStream(underlyingStream);
  serializer.writeObject(serializableObject);
  ```

- Later, you can read it with

  ```
  FileInputStream underlyingStream = new FileInputStream("/tmp/file");
  ObjectInputStream deserializer = new ObjectInputStream(underlyingStream);
  Object deserializedObject = deserializer.readObject();
  //hmmmm, must typecast deserializerObject but, to what?
  ```

11.2 Making a Class Serializable

1. Implement the `Serializable` interface.
   (a) Add `implements Serializable` to class definition.
2. Make sure that instance-level locally defined state is serialized properly.
3. Make sure that superclass state is properly serialized.
4. Override `equals()` and `hashCode()`.

11.2.1 Ensure Serialization of Instance-Level State

- The non-static member variables contain the instance-level state.
- If they are primitive or serializable then its OK.
- Your classes might not be serializable. Also, Java classes such as `ArrayList` are not serializable!
- Your choices are:

1. Make the variable `transient`:

   private transient Object myobject[];

   so it will not get serialized (be careful!)

2. Declare which variables should be stored by defining a special variable:

   private static final ObjectSreamField[] serialPersistentFields =
   { **new ObjectStreamField**("size", Integer.Type), ... };

3. Do your own serialization by implementing

   private void writeObject(java.io.**ObjectOutputStream** out) **throws IOException**;
   private void readOject(java.io.**ObjectInputStream** in)
   throws IOException, **ClassNotFoundException**;

   these methods can invoke `defaultWriteObject()` to invoke default serialization on the non-transient members.

11.2.2 Ensure Superclass State is Handled Correctly

- If superclass is serializable then relax.
- Else if superclass maintains state then you can
  1. Use `serialPersistentFields` to tell serialization about the parent's fields that need to be serialized.
  2. Write your own `writeObject()` and `readObject()`.
- Also, it must have a zero-argument constructor (for de-serialization).

11.2.3 Override equals() and hashCode()

- The default implementations use the object's location.
- After being deserialized, you have two identical copies which you might want to be `equals()` and produce the same `hashCode()`.

12 Threading

- Start with code that works for a single client.
- Ensure data integrity.
- Minimize time in synchronized blocks.
- Be careful when using container classes.
- Use containers to mediate inter-thread communication.
- Immutable objects are automatically threadsafe.
  - `String`
- Always have a safe way to stop your threads.
  - Release all locks and resources before quitting.
- Background threads should have low priority.
- Pay careful attention to what you serialize.
  - Serialization takes time and objects could change in the meantime.
- Use threading to reduce response-time variance.
  - Starting a new thread can allow you to return to client quickly (e.g., web server).
- Limit the number of objects a thread touches.
  - Make it easier to debug.
- Acquire locks in a fixed order.
- User worker threads to prevent deadlocks.

12.1 Ensure Data Integrity

- Avoid either excessive or inadequate synchronization.
- We can try to synch everything:

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

public class Account_Impl extends UnicastRemoteObject implements Account {
  private Money _balance;

  public synchronized Money getBalance()
    throws RemoteException {
    return _balance;
  }

  public synchronized void makeDeposit(Money amount)
    throws RemoteException, NegativeAmountException {
    checkForNegativeAmount(amount);
    _balance.add(amount);
    return;
```

```
  }

  public synchronized void makeWithdrawal(Money amount)
    throws RemoteException, OverdraftException, NegativeAmountException {
    checkForNegativeAmount(amount);
    checkForOverdraft(amount);
    _balance.subtract(amount);
    return;
  }
}
```

- It is possible to check balance then fail to withdraw that because someone else withdrew the money just after.
- Client should maintain lock.

12.2 Client Maintains Lock

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

public class Account2_Impl extends UnicastRemoteObject
  implements Account2 {
  private Money _balance;
  private String _currentClient;

  public Account2_Impl(Money startingBalance)
    throws RemoteException {
    _balance = startingBalance;
  }

  /** The client can use this to get a lock on the account */
  public synchronized void getLock()
    throws RemoteException, LockedAccountException {
    if (false == becomeOwner()) {
      throw new LockedAccountException();
    }
    return;
  }

  /** The client can use this to release a lock on the account */
  public synchronized void releaseLock() throws RemoteException {
    String clientHost = wrapperAroundGetClientHost();

    if ((null != _currentClient) && (_currentClient.equals(clientHost))) {
      _currentClient = null;
    }
  }

  private boolean becomeOwner() {
    String clientHost = wrapperAroundGetClientHost();

    if (null != _currentClient) {
```

```
      if (_currentClient.equals(clientHost)) {
        return true;
      }
    } else {
      _currentClient = clientHost;
      return true;
    }
    return false;
  }

  private void checkAccess() throws LockedAccountException {
    String clientHost = wrapperAroundGetClientHost();

    if ((null != _currentClient) && (_currentClient.equals(clientHost))) {
      return;
    }
    throw new LockedAccountException();
  }

  private String wrapperAroundGetClientHost() {
    String clientHost = null;

    try {
      clientHost = getClientHost();
    } catch (ServerNotActiveException ignored) {
    }
    return clientHost;
  }

  public synchronized Money getBalance()
    throws RemoteException, LockedAccountException {
    checkAccess();
    return _balance;
  }

  public synchronized void makeDeposit(Money amount)
    throws RemoteException, LockedAccountException, NegativeAmountException {
    checkAccess();
    checkForNegativeAmount(amount);
    _balance.add(amount);
    return;
  }

  public synchronized void makeWithdrawal(Money amount)
   throws RemoteException, OverdraftException, LockedAccountException, NegativeAmoun-
tException {
    checkAccess();
    checkForNegativeAmount(amount);
    checkForOverdraft(amount);
    _balance.subtract(amount);
    return;
  }

  private void checkForNegativeAmount(Money amount)
```

```
    throws NegativeAmountException {
    int cents = amount.getCents();

    if (0 > cents) {
      throw new NegativeAmountException();
    }
  }

  private void checkForOverdraft(Money amount)
    throws OverdraftException {
    if (amount.greaterThan(_balance)) {
      throw new OverdraftException(false);
    }
    return;
  }
}
```

- This increases the number of method calls the client must make.
- Its vulnerable to partial failure. What if the client that has the lock dies?

12.3 Using a Lock Expiry Thread

```
import java.rmi.*;
import java.rmi.server.*;
/*
 Has timer-based lock management on server-side
 */

public class Account3_Impl extends UnicastRemoteObject implements Account3 {
  private static final int TIMER_DURATION = 120000; // Two minutes
  private static final int THREAD_SLEEP_TIME = 10000; // 10 seconds

  private Money _balance;
  private String _currentClient;
  private int _timeLeftUntilLockIsReleased;

  public Account3_Impl(Money startingBalance)
    throws RemoteException {
    _balance = startingBalance;
    _timeLeftUntilLockIsReleased = 0;
    new Thread(new CountDownTimer()).start();
  }

  public synchronized Money getBalance()
    throws RemoteException, LockedAccountException {
    checkAccess();
    return _balance;
  }

  public synchronized void makeDeposit(Money amount)
    throws RemoteException, LockedAccountException, NegativeAmountException {
    checkAccess();
```

```
    checkForNegativeAmount(amount);
    _balance.add(amount);
    return;
  }

  public synchronized void makeWithdrawal(Money amount)
   throws RemoteException, OverdraftException, LockedAccountException, NegativeAmoun-
tException {
    checkAccess();
    checkForNegativeAmount(amount);
    checkForOverdraft(amount);
    _balance.subtract(amount);
    return;
  }

  private void checkAccess() throws LockedAccountException {
    String clientHost = wrapperAroundGetClientHost();

    if (null == _currentClient) {
      _currentClient = clientHost;
    } else {
      if (!_currentClient.equals(clientHost)) {
        throw new LockedAccountException();
      }
    }
    resetCounter();
    return;
  }

  private void resetCounter() {
    _timeLeftUntilLockIsReleased = TIMER_DURATION;
  }

  private void releaseLock() {
    if (null != _currentClient) {
      _currentClient = null;
    }
  }

  private String wrapperAroundGetClientHost() {
    String clientHost = null;

    try {
      clientHost = getClientHost();
    } catch (ServerNotActiveException ignored) {
    }
    return clientHost;
  }

  private void checkForNegativeAmount(Money amount)
    throws NegativeAmountException {
    int cents = amount.getCents();

    if (0 > cents) {
```

```
    throw new NegativeAmountException();
   }
  }

  private void checkForOverdraft(Money amount)
   throws OverdraftException {
   if (amount.greaterThan(_balance)) {
    throw new OverdraftException(false);
   }
   return;
  }

  /** The expire thread */
  private class CountDownTimer implements Runnable {
   public void run() {
    while (true) {
     try {
      Thread.sleep(THREAD_SLEEP_TIME);
     } catch (Exception ignored) {
     }
     synchronized (Account3_Impl.this) {
      if (_timeLeftUntilLockIsReleased > 0) {
       _timeLeftUntilLockIsReleased -= THREAD_SLEEP_TIME;
      } else {
       releaseLock();
      }
     }
    }
   }
  }
}
```

- Lock remains as long as client executes an operation every two minutes or more.
- This code is more complicated.
- The threads are expensive. There is one per account. We could have one thread check all instances (see book).

12.4 Minimize Time in Synchronized Blocks.

- Synchronize around the smallest possible block of code.
- Don't synchronize across device accesses. If many clients use this method, they will all stop when the device slows down. This also creates a queue.
  - One solution is to use a background thread which does a "pull" of the information, instead of the "push".

12.5 Be Careful When Using Container Classes.

- `Vector` and `Hashtable` claim to be threadsafe because they synchronize every method.
- But, this does not mean that they are really safe.
- Is the following thread safe?

```
import java.util.*;
public synchronized void insertIfAbsent(Vector vector, Object object){
  if (vector.contains(object)) {
    return;
  }
  vector.add(object);
}
```

- No. It assumes there is only one instance of the class.
- The right way is to do:

```
import java.util.*;

public void insertIfAbsent(Vector vector, Object object){
  synchronized (vector){
    if (vector.contains(object)) {
      return;
    }
    vector.add(object);
  }
}
```

12.6 Use Containers To Mediate Inter-thread Communication.

- If one thread needs to feed data to another thread it is often useful to use a queue.
- The publishing thread can always add to the end of the queue. The consumer thread takes from the head of the queue.
- We replace a synchronous call with asynchronous calls.
- Return values are no longer meaningful. We might need a callback mechanism (e.g., out-of-paper).

13 Testing

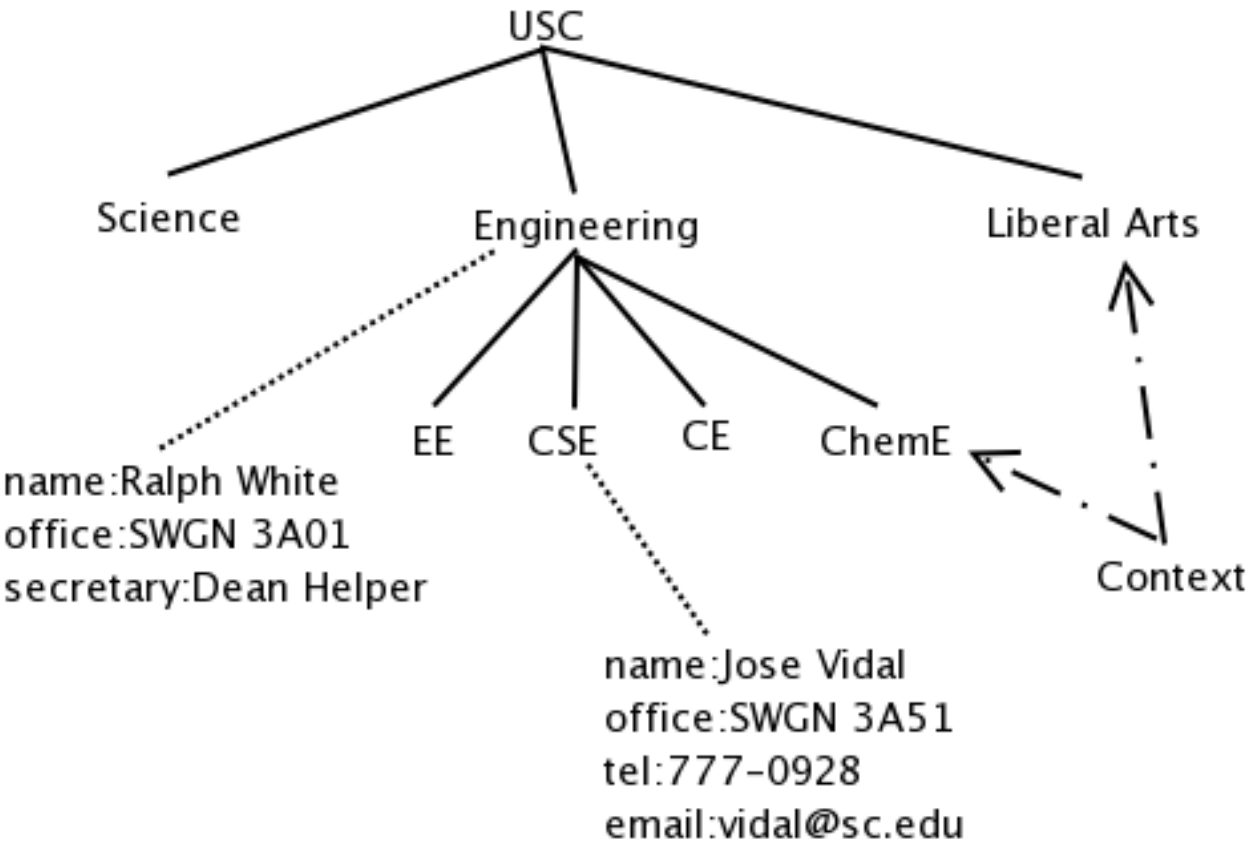
- Correctness:
  - Can client connect to server?
  - Can client perform remote calls?
  - Does server do the right thing?
- Scalability
  - Does it work with more than one client?
  - Is performance acceptable under normal load?
  - Can it handle peak loads?
  - How does it degrade with increased load?
  - How does it behavior over long-term? leaks?

13.1 Test Strategy

1. Build tests objects that test unit functionality, e.g., one function.
2. Build aggregate tests for entire use case.
3. Build threaded *tester* that does many of these, in sequence.
4. Build a container that launches many testers. Simulate users.
5. Build a reporting mechanism.
6. Run many tests.
7. Profile performance using tester containers.

Note:
I realize that these instructions are overkill for your typical problem set project. However, any real-world project will likely be ten times larger and involve five times as many developers. Unit testing and automated testing are **indispensable** for any real project. Frequent automated testing forms the basis of all major software development companies' culture. In fact, many of them have nightly builds/tests. If any bugs are found then fixing it becomes the developer's first priority.

14 Naming Services



- The rmiregistry is limited to one flat search space of services on one machine!
- Scalable systems can extend the DNS idea by adding attributes.

- Most popular example is LDAP[4].
- It defines nested **contexts**. Elements can be added to any context and have attributes and values.
- Search proceeds bottom-up. Assumes most searches answered locally.

Note:
LDAP provides an excellent example of the kind of sophisticated naming service that will likely be needed to replace the rmiregistry's limited search ability. If we envision a world of different remote objects offered by different companies then the need for this type of service is obvious. Web services have been trying to provide a solution to this same problem via the use of UDDI and WSDL (later in class).

15 RMI Garbage Collection

- The JVM uses reference counting for local garbage collection. When number of references is zero then it can be collected.
- No time guarantee.
- RMI keeps track of number of active clients.
- Clients *lease* objects for fixed periods (10 minutes, `java.rmi.dgc.leasevalue`)
- Clients must renew or lease expires and object gets collected.

16 RMI Logging

- The *standard log* is turned on with

  java -Djava.rmi.server.logCalls=true ...

  or with

  System.getProperties().put("java.rmi.server.logCalls","true");

  and you must set the destination with

  FileOutputStream logFile = **new FileOutputStream**("/tmp/file");
  RemoteServer.setLog(logFile);

  Log is very verbose.

16.1 Specialized Logs

- They are: transport, proxy, tcp, dgc, and loader.
- Set them with

  FileOutputStream transportLogFile = **new FileOutputStream**("/tmp/tlog");
  LogStream.log("transport").setOutputStream(transportLogFile);

and similarly for the other types.

- You can then determine how much information goes into them by setting these six properties to either: silent, brief, or verbose.
  1. sum.rmi.server.dgcLevel
  2. sun.rmi.server.logLevel
  3. sun.rmi.loader.logLevel
  4. sun.rmi.transport.logLevel
  5. sun.rmi.transport.tcp.logLevel
  6. sun.rmi.transport.proxy.logLevel

17 Security Policy

"Making a distributed system secure is a mindnumbingly difficult task."–William Grosso

- Keep API as simple as possible.
- Grant your code (downloaded code) the minimal permissions needed.
- The `SecurityManager` tells JVM what the code can't do.

17.1 Security Manager Permissions

- *AWT permissions*: accesClipboard, accessEventQueue, listenToAllAwtEvents, readDisplayPixels, showWindowWithoutWarningBanner, and createRobot (pretends to be user).

```
grant{
  permission java.awt.AWTPermission "showWindowWithoutWarningBanner";};
```

- *File permissions* can be: read, write, execute, and delete.

```
grant {
  permissions java.io.FilePermission "/tmp/-", "read, write, delete";};
```

- is a recursive wildcard, * is local wildcard.

- *Socket permissions* cover resolve, connect, listen, and accept.

```
grant {
  permission java.net.SocketPermission "*.sc.edu:1024-", "accept,
  connect";};
```

- *Property permissions* allow read, write to system properties ("-Dsun.com.property=false")

```
grant {
  permission java.util.PropertyPermission "java.version", "read";}
```

17.2 The Security Manager

- *If there is none, then there is no security checks!*
- You can install one with

  java -Djava.security.manager application

  or in the code with

  System.setSecurityManager(**new RMISecurityManager**());

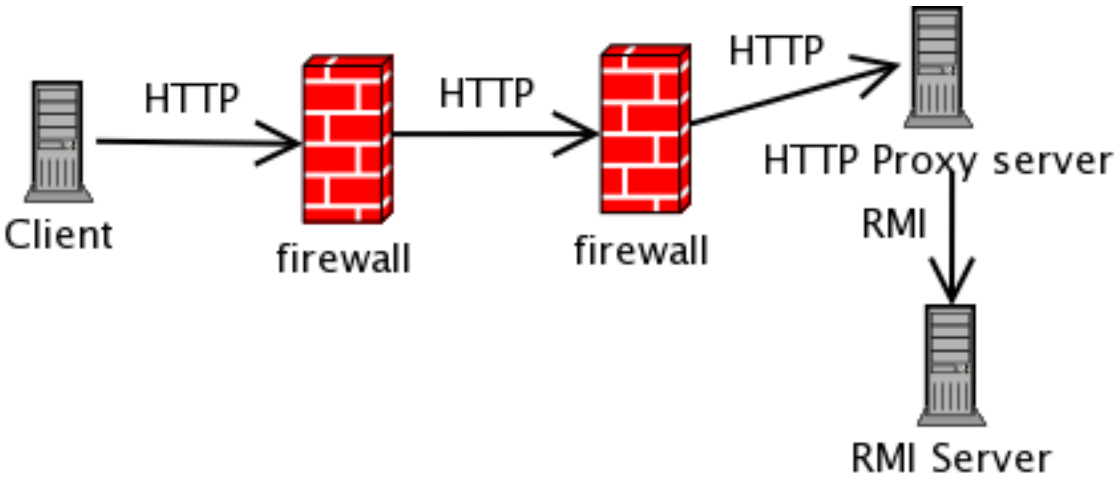
  this last is a much better idea for your RMI code.
- In Java there are three policy files that are used.
  1. Global policy file. (`...jre/lib/security/java.policy`)
  2. User-specific policy file (`HOME/.java.policy`)
  3. Application-specific set with

     java -Djava.security.policy="java.policy"
- The general format is

  grant [signedBy Name] [codebase URL] {...};

  where Name and URL can be anything.
- The program `policytool` provides a gui for editing this file.

18 HTTP Tunneling



- When an RMI client (stub, even registry stub) tries to contact a server, it will try:

1. Contact directly using JRMP.
2. Make direct HTTP connection to server, encapsulate method call in HTTP request.

3. Assume firewall is proxy server, ask it to forward the request to appropriate port on server. Firewall forwards request as HTTP request.

4. Connect to port 80 on server machine and send request to URL beginning with `/cgi-bin/java-rmi.cgi`. Hope request gets forwarded to proper port on server machine.

5. Connect to port 80 of firewall machine and send request to URL beginning with `/cgi-bin/java-rmi.cgi`. Hope request gets forwarded to server.

Notes

[1] http://www.amazon.com/exec/obidos/ASIN/1565924525/multiagentcom/
[2] http://www.oreilly.com/catalog/javarmi/
[3] http://www.wikipedia.org/wiki/Adapter_pattern
[4] http://www.wikipedia.org/wiki/LDAP

This talk is available at `http://jmvidal.cse.sc.edu/talks/rmidesign`