- Welcome to 334.
- People
  - Ulrich Speidel (supervisor), Xinfeng Ye
- Assessment
  - 4 assignments with a combined weight of 15%
  - one test with a weight of 25%
  - one exam with a weight of 60%
  - you must pass both practical and theory to pass the course
- Schedule
  - Week 1 - 2 (Xinfeng Ye)
  - Week 3 - 4 (Ulrich Speidel)
  - Week 5 – 6 (Xinfeng Ye)
  - Week 7 – 8 (Ulrich Speidel)
  - Week 9 – 10 (Xinfeng Ye)
  - Week 11 – 12 (Ulrich Speidel)
  - The even numbered weeks' (e.g. week 2, 4, etc.) Wednesday 4:30pm lectures are in-class on-demand tutorials.

- Xin Feng Ye
- Office
  - 303.589 (City)
- Office Hours (during my lecturing period)
  - Mon 5:30pm – 6:00pm (Tamaki)
  - Wed 5:30pm – 6:00pm (Tamaki)
  - or in my city office

Assignment Marking

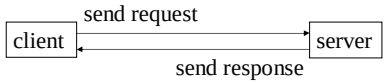
- All assignments carry equal weight, i.e., 3.75% of your final mark.
- Each assignment will carry a specific number of **points**, typically 100 points.
  - Getting 60 or more of the assignment points gives you full marks (3.75% of the total course marks) for the assignment.
  - Scoring more points does not give you any extra marks, but it gives you a better preparation for test and exam.
  - Marking is based on block-box marking

Recommended Readings

- RMI
  - Tutorials on Sun's web site
  - Tutorials that come with J2SE 6 download
- Servlets and JSP
  - Core SERVLETS and JAVASERVER PAGES, Volume 1: Core Technologies, by Marty Hall and Larry Brown
    A Sun Microsystems Press/Prentice Hall PTR Book
    ISBN 0-13-009229-0

Distributed Systems Middleware

- A complex system consists of software components running on different machines.
- To make the system work, the components on different machines must communicate with each other.
- The communication need protocols to exchange data/transfer control.



Sockets

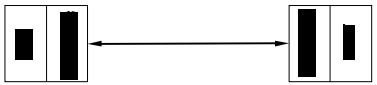
- Basic inter-machine communication model
  - find out IP addresses
  - make connection
  - exchange data

Lots of Complexities...

- How does client locate server? Server object(s)?
- What if server location moves/multiple servers?
- What if multiple clients/concurrent access?
- What protocol/language on client? Server?
  - How “serialise”/”deserialise” data for transport?
  - How does client invoke server function?

Middleware for Distributed Systems

- A middleware can be regarded as a software that connects two otherwise separate applications.
- A middleware for distributed systems is responsible for handling the communication between the software components running on different machines.
- A middleware also provides mechanisms for registering and discovering services in the system.



Java Remote Method Invocation (RMI)

- The Java Remote Method Invocation (RMI) system allows an object running in one Java Virtual Machine (VM) to invoke methods on an object running in another Java VM.
- RMI provides for remote communication between programs written in the Java programming language.
- A primary goal of RMI was to allow programmers to develop distributed Java programs (i.e. programs running on different machines) with the same syntax and semantics used for non-distributed programs.

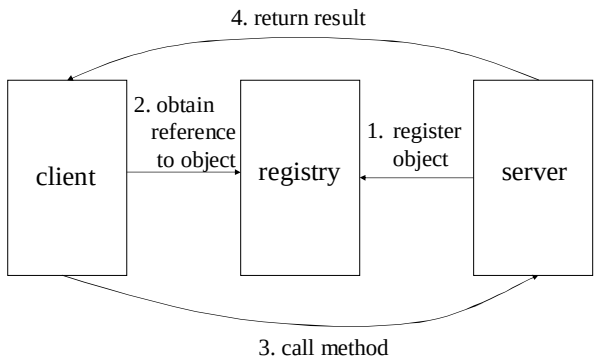
References on RMI

- Sun provides on-line tutorials on RMI http://java.sun.com/docs/books/tutorial/rmi/TOC.html
- You can also read the RMI tutorial that comes with the J2SE 6.0 download
- Compared with previous versions, there are some differences in writing RMI applications in J2SE 6.0.
  - we use J2SE 6.0

An Overview of RMI Applications

- RMI applications are often comprised of two separate parts: a server and a client.
- A typical server application
  - creates some objects, called remote objects
  - makes references to remote objects accessible
  - waits for clients to invoke methods on these remote objects
- A typical client application gets a remote reference to remote objects in the server and then invokes methods on them.
  - The execution of the methods of the remote objects are carried out on the server

- RMI provides the mechanism by which the server and the client communicate and pass information back and forth.
- RMI provides a simple naming facility, the rmiregistry, for
  - Server to register remote objects
  - Client to discover references to the remote objects



Writing an RMI Application

- writing an RMI server
  - define server interface
  - implement server interface
  - set up server objects
- creating a client program
  - obtain a reference to a remote object
  - manipulate the object

Writing an RMI Server

- An account object represents some kind of bank account. We use RMI to export it as a remote object so that remote clients, e.g. ATMs, personal banking software running on a PC) can access it and carry out operations.
- The server is comprised of an interface and a class.
  - The interface provides the definition for the methods that can be called from the client.
  - The class provides the implementation.
- Writing an RMI server consists of two tasks:
  - Define the interface
  - Write a class to implement the interface

Server interface

- The interface extends java.rmi.Remote to be an RMI object.
- All the methods in the interface must throw java.rmi.RemoteException.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Account extends Remote {
   public String getName() throws RemoteException;
   public int getBalance() throws RemoteException;
   public void withdraw(int amt) throws RemoteException;
   public void deposit(int amt) throws RemoteException;
   public void transfer(int amt, Account src) throws RemoteException;
}
```

must extend this to be an RMI object

all the methods must throw this exception

Implementing server interface

- The implementation class should implement all the methods in the interface.
- The implementation can implement methods that are not defined in the interface. However, these methods cannot be called by the clients of the remote (server) objects.


```
import java.rmi.RemoteException;
public class AccountImpl implements Account {
        private int balance; // account balance
        private String name; // name of the account holder
        public AccountImpl(String name) throws RemoteException {
                this.name = name;
        }
        public String getName() throws RemoteException {return name;}
        public int getBalance() throws RemoteException {return balance;}
        public void withdraw(int amt) throws RemoteException {balance -= amt;}
        public void deposit(int amt) throws RemoteException {balance += amt;}
        public void transfer(int amt, Account src) throws RemoteException {
                src.withdraw(amt);
                this.deposit(amt);
        }
}
```


Setting up server objects

- create server (remote) objects on the server
- export the objects to RMI runtime (the middleware)
- register the object with a name service

```
// contains methods for accessing name service
import java.rmi.Naming;

// contains methods for manipulating server objects
import java.rmi.server.UnicastRemoteObject;
public class RegAccount {
  public static void main(String[] args) {
    try {
      // create a server (remote) object
      AccountImpl account = new AccountImpl("X");

      // export the server object to the RMI runtime
      // the server object listens on a port assigned by JVM
      Account stub = (Account)
            UnicastRemoteObject.exportObject(account,0);
```



```
      // register the object with a name server
      // the server object is given name "X" on the name server
      // the name server is at port 8081
      Naming.rebind("//localhost:8081/X",stub);

      // the server object is ready to be called
      System.out.println("Account registerd.");
    }
    catch (Exception e) {
      System.out.println("Error in RegAccount");
      e.printStackTrace();
    }
  }
}
```


Creating a Client Program

- Regarding the use of the remote (i.e. server) object, a client program needs to carry out the following two tasks:
  - Look up the remote object
  - Manipulate the remote object using the methods specified in the server interface

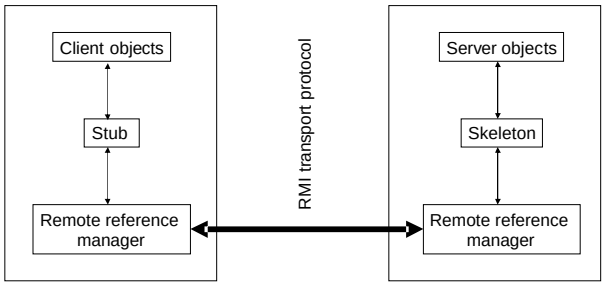

```
// contains methods for accessing name service
import java.rmi.Naming;
public class AccountClient {
  public static void main(String[] args) {
    try {
      // look up the server object with name "X"
      Account xAccount =
            (Account)Naming.lookup("//localhost:8081/X");

      // call the getBalance method to display account balance
      System.out.println("Balance of account is: "+
                  xAccount.getBalance());

      // deposit money to the account
      xAccount.deposit(1234);
```

```
            // display the new balance
            System.out.println("Deposit 1234 into "+
                                    xAccount.getName());
            System.out.println("Balance of account is: "+
                                    xAccount.getBalance());
        }
        catch (Exception e) {
            System.out.println("Error in AccountClient");
            e.printStackTrace();
        }
    }
}
```


Compiling and Running

- Compile all the classes and interfaces
  - javac *.java
- Start the name server
  - rmiregistry 8081
- Create and register the account object
  - java RegAccount
- Run the client
  - java AccountClient

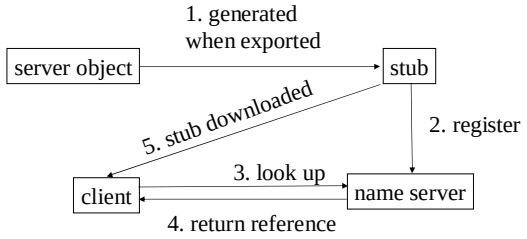
RMI Architecture



- When a client calls a method on a remote object, the corresponding method in the stub is called.
- The stub marshals the arguments in the method call into serialized form. There are three possible cases:
  - An argument is a Remote object: forwards the reference to the object
  - An argument is a primitive data type or a Serializable object: serialize the argument
  - Neither of the above: raise an exception

- On the client side, the remote reference manager converts the stub request to low-level protocol messages.
- On the server side, the remote reference manager converts the low-level protocol messages into a format that the skeleton can understand.
- The skeleton unmarshals the arguments and calls the appropriate method on the actual server object.
- If there are information to be passed back to the client, the skeleton marshals the information and forwards them to the client side. The stub on the client side would unmarshal the information and pass them to the client.

How does a client get the stub?




```
AccountImpl account = new AccountImpl("X");
Account stub = (Account) UnicastRemoteObject.exportObject(account,0);
Naming.rebind("//localhost:8081/X",stub);
```


The registry and naming services

- When you start rmiregistry, you can specify a port number. By default, rmiregistry listens to port 1099.
- Once the RMI registry is running, you register remote objects with it using one of the classes:
  - java.rmi.registry.LocateRegistry
  - java.rmi.Naming
  - java.rmi.registry.Registry

Some useful methods

- java.rmi.registry.LocateRegistry
  - createRegistry
    - Start your own registry service
  - getRegistry
    - Obtain a reference to a registry service either on localhost or on a specified host
- java.rmi.registry.Registry
  - bind, rebind, unbind

```
Registry reg =
        LocateRegistry.getRegistry(8081);
reg.rebind("X",account);
```

On server:

```
LocateRegistry.createRegistry(8081);
Registry reg = LocateRegistry.getRegistry(8081);
reg.rebind("X",stub);
```

On client:

```
Registry reg = LocateRegistry.getRegistry("localhost", 8081);
Account xAccount = (Account)reg.lookup("X");
```


- java.rmi.Naming
  - This class can be used to bind an object to a known registry
    - bind, rebind, unbind
  - This class lets a client look up local and remote objects using URL-like naming syntax.
    - //host:port/object-name
  - On server

    ```
    Naming.rebind("//localhost:8080/X",account);
    ```
  - On client

    ```
    Account xAccount =
    (Account)Naming.lookup("//localhost:8080/X");
    ```


JDBC

- Load the JDBC driver.
- Define the connection URL.
- Establish the connection.
- Create a statement object.
- Execute a query or update.
- Process the results.
- Close the connection.

DB Connection Pool

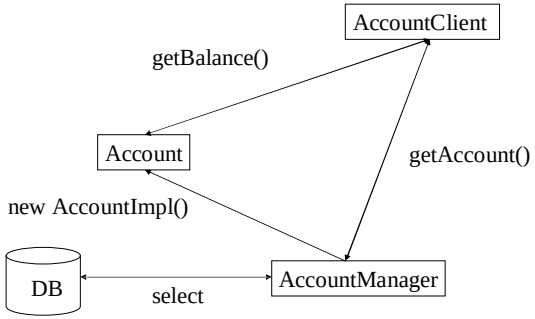
- Opening a connections to a database is a time-consuming process.
- To make the access to DBs more efficient, the connections to DBs should be reused.
- Refer to Chapter 17&18 of *Core SERVLETS and JAVASERVER PAGES*

DBConn class

- Hold a set of connection to a DB
- DBConn()
  - DB driver, location, user name, password, number of connections in the pool
- makeConnection()
  - obtain a connection to the DB, called by DBConn()
  - declared as private
- getConnection()
  - obtain a connection from the connection pool
- releaseConnection()
  - return a connection to the connection pool
- closeAllConnection()
  - release all the DB connections in the connection pool


```
// obtain a connection from the DB connection pool
public synchronized Connection getConnection() {
   Connection conn=null;
   try {
     // there are still connections available in the pool
     if (!available.isEmpty()) {
            // obtain the connection
            conn = available.lastElement();
            // record the connection as no longer available
            available.remove(conn);
             busy.addElement(conn);
     }
```


Integrating the Account Example with DB




```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface AccountManager extends Remote {
   // retrieve an account from the DB according to the account's name
   public Account getAccount(String name) throws RemoteException;
}
```

AccountManagerImpl class

- AccountManagerImpl
  - create a DB connection pool,
    - DBConn

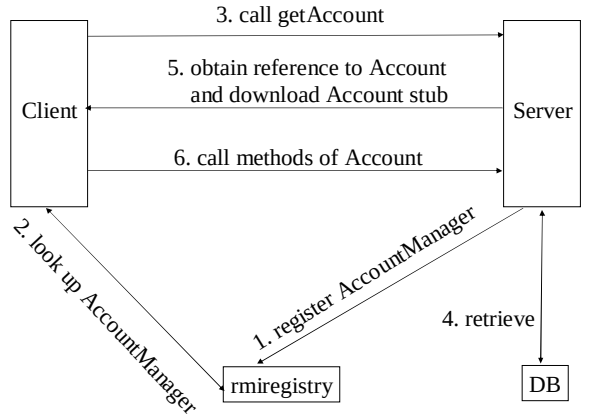
- getAccount
  - obtains a connection from the DB connection pool
    - getConnection()
  - retrieve account details from the DB
    - The DB table account has two columns, name and balance.
    - SQL statement for retrieving the account details of a given user: select * from account where name='X'
    - Java statements for querying a DB: getConnection, createStatement, executeQuery
  - construct an Account object
    - account = new AccountImpl(accountName,balance);
  - make the object a RMI remote object
    - stub = (Account) UnicastRemoteObject.exportObject(account, 0);
  - return the DB connection back to the connection pool
    - releaseConnection
  - return the reference to the server object back to the caller
    - return stub

RegAccountManager class

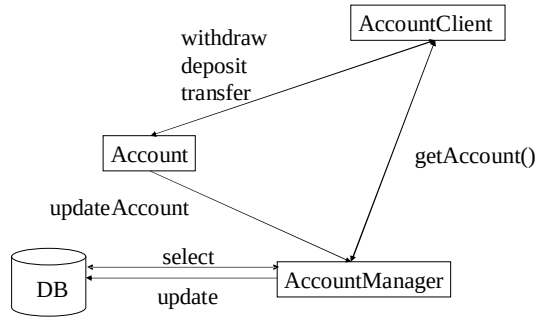
- Create an AccountManagerImpl object
- Make the object a RMI remote object
  - AccountManager stub = (AccountManager) UnicastRemoteObject.exportObject(accountManager, 0);
- Create a RMI registry
  - LocateRegistry.createRegistry(8081);
  - No need to start rmi registry manually
- Register the RMI object with the registry

BankClient class

- Look up the AccountManager object
  - AccountManager manager = (AccountManager)Naming.lookup("//localhost:8081/manager");
- Obtains references to some Account objects
  - Account xAccount = manager.getAccount("X");
  - Account yAccount = manager.getAccount("Y");
- Manipulate the Account objects
  - getBalance, transfer



Improvement to the Account Example



AccountManager Interface

- Account objects and the AccountManager objects reside at the same location.
  - updateAccount does not need to be provided as a method that can be accessed remotely
  - AccountManager Interface remains the same
- The class that implements AccountManager needs to implement the updateAccount method
  - This method can only be accessed locally, i.e. cannot be accessed by client at a different location.

AccountManagerImpl2 class

- Same as the AccountManagerImpl apart from the discussions below.
- updateAccount
  - obtains a DB connection from the connection pool
  - update the account details in the DB
    - SQL statement for updating a record of a given client
      - update account set balance=X where name='Y'
  - return the DB connection to the connection pool
- getAccount
  - when an Account object is created, a reference to the AccountManager should be passed to the Account object (see explanation later)
    - account = new AccountImpl2(accountName,balance, this);

AccountImpl2 class

- Same as AccountImpl apart from the discussion below.
- AccountImpl2
  - The constructor should receive a reference to the AccountManagerImpl2 object. This is to allow the Account object call the updateAccount method of the AccountManagerImpl2.
    - AccountImpl2(String name, int balance, AccountManagerImpl2 accountManager)
    - this.accountManager = accountManager;
- withdraw, deposit
  - call the updateAccount method of the AccountManager to write the changes back to DB
    - accountManager.updateAccount(this);

Remote Method Arguments and Return Values

- The arguments and the return values of a remote method are either primitive data types, e.g int, or objects which implement java.io.Serializable interface, or references to remote objects.
- The server does not necessarily know the concrete implementation of the objects being passed in. As a consequence, the server's JVM might also need to download the relevant classes when a remote method call is made.

Download Classes Dynamically

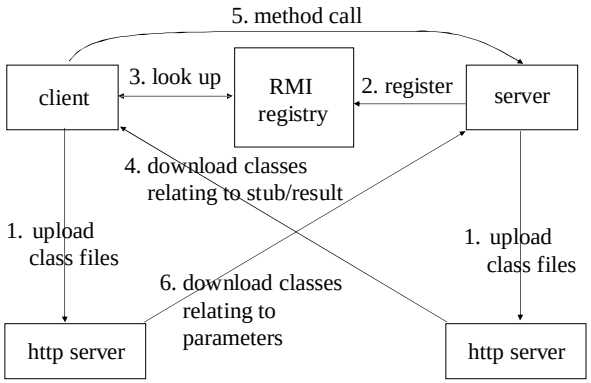
- JVM can dynamically download Java software from any URL, e.g. a web server.
- A codebase is a place, from which to load classes into a virtual machine.
  - CLASSPATH is a "local codebase", because it is the list of places on disk from which you load local classes.
  - java.rmi.server.codebase property value represents one or more URL locations from which classes needed during the execution of the RMI applications can be downloaded.
- The classes needed to execute remote method calls should be made accessible from a network resource, such as an HTTP or FTP server.
- java.rmi.server.codebase can be specified when a program is started
  - java -Djava.rmi.server.codebase=http://localhost:8080/rmi/ex6/ ComputeClient

The need for downloading classes dynamically (1)

- When a client makes a method call, there are three possible cases:
  - All of the method parameters (and return value) are primitive data types, so the remote object knows how to interpret them. Thus, there is no need to check its CLASSPATH or any codebase.
  - At least one remote method parameter or the return value is an object, for which the remote object can find the class definition locally in its CLASSPATH.
  - The remote method receives an object instance, for which the remote object cannot find the class definition locally in its CLASSPATH.
    - The class of the object sent by the client will be a subtype of the declared parameter type.
    - In this case the class need to be downloaded to the server.

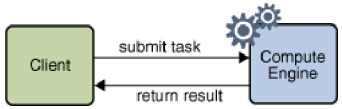
The need for downloading classes dynamically (2)

- When a client receives a stub, the stub uses classes which cannot be found in the client's CLASSPATH. In this case the class need to be downloaded to the client.



Compute Engine

(http://java.sun.com/docs/books/tutorial/rmi/)



- A client can submit a task to the server (computer engine) for execution.
  - The submitted task is executed on the server
- The server provides a (remote) interface for client to submit a task .
- An interface is defined to specify the task to be submitted to the server.
  - This interface is available on both client and server
  - The task submitted by the client implements the interface.
  - The interface is non-remote.


```
public class Adder
{
   private int i, j;
   public Adder(int i, int j) {
      this.i = i;
      this.j = j;
   }
   public Integer execute() {
      return (new Integer(i+j));
   }
}
```

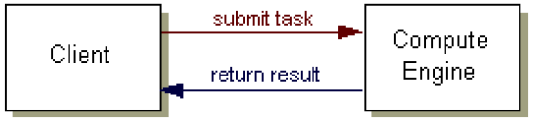
```
public class Multiplier
{
   private double i, j;
   public Adder(double i, double j) {
      this.i = i;
      this.j = j;
   }
   public Double execute() {
      return (new Double(i+j));
   }
}
```

Argument (task) Interface

```
import java.io.Serializable;
public interface Task<T> extends Serializable {
   T execute();
}
```

- Task is not a remote object
- The Task interface must extend Serializable to allow the task to be sent to the server.
- In order to allow development on server and client site, the server interface and the argument interface should be available on both sites.

Server interface (remote)




```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Compute extends Remote {
   <T> T executeTask(Task<T> t) throws RemoteException;
}
```

Java generic types:
http://www-128.ibm.com/developerworks/edu/j-dw-java-generics-i.html

Server Implementation

- The task submitted by the client is a subtype of the Task<T> interface. The class needs to be downloaded by the server at run time.
- In order for a JVM to attempt to load classes remotely, a security manager has to be installed to allow remote class loading.
  - System.setSecurityManager(new RMISecurityManager())


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

public class ComputeImpl implements Compute
{
   public ComputeImpl() throws RemoteException {
      // set up security manager to allow class downloading
      System.setSecurityManager(new RMISecurityManager());
   }

   public <T> T executeTask(Task<T> t) {
      // execute the submitted job
      return t.execute();
   }
}
```

Create Server Object

- create a remote object
  - ComputeImpl ce = new ComputeImpl();
- export the remote object
  - UnicastRemoteObject.exportObject
- create a RMI registry
  - LocateRegistry.createRegistry
- register with the RMI registry
  - rebind
  - The reference to the stub

Policy files

- When a compute engine object is created, the security manager of the object will read a policy file to determine which actions are allowed for the compute engine.
- The file below allows the engine to accept connections and make connections on any non-privileged port.

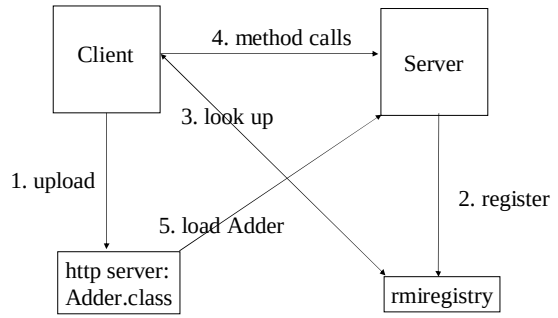
```
grant {
permission java.net.SocketPermission "*:1024-65535", "accept, connect";
};

java -Djava.security.policy=mypolicy RegCompute
```


Client Implementation

- Write a concrete task implementation.
  - The class specifies the task to be sent to the server.
  - The execute method should contain the code that carry out the computation.
- Upload the task class to a web server for server to download during its execution.
- Write client application
  - Create a task.
  - Look up the compute engine.
  - Submit the task to the compute engine through the remote interface.
  - When start the client, specify the value of java.rmi.server.codebase
    - java -Djava.rmi.server.codebase=http://localhost:8080/rmi/ex6/ComputeClient


```
public class Adder implements Task<Integer>
{
        private static final long serialVersionUID = 334L;
        private int i, j;
        public Adder(int i, int j) {
                this.i = i;
                this.j = j;
        }
        public Integer execute()
        {
                return (new Integer(i+j));
        }
}
```

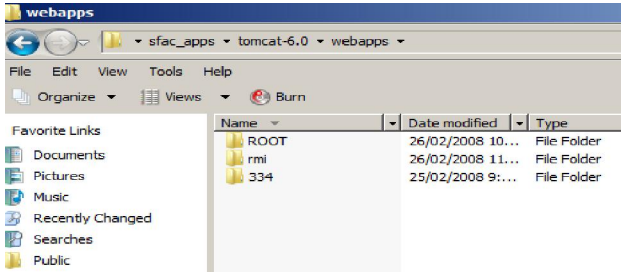
```
public class ComputeClient {
   public static void main(String[] args) {
         try {
            // look up the compute engine
            Compute ce = (Compute)Naming.lookup("//localhost:8081/ce");
            // create a task
            Adder adder = new Adder(1,2);
            // send the task to the server
            Integer result = (Integer)ce.executeTask(adder);
            System.out.println("Result is: "+result.intValue());
         }
```

Tomcat in the lab

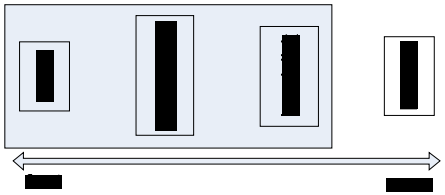
- download rmi-classes.zip file from http://www.cs.auckland.ac.nz/compsci334s1t/resources/rmi-classes.zip
- unpack the file and store it at H:\sfac-apps\tomcat-6.0\webapps
- Start Tomcat in the lab
  - Start menu à programs à development à development environment à Apache Tomcat

- download rmi-classes.zip file from http://www.cs.auckland.ac.nz/compsci334s1t/resources/rmi-classes.zip
- unpack the file and store it at H:\\sfac_apps\tomcat-6.0\webapps



Performance Tuning

- Many applications are time-critical.
  - It is important to make your application as efficient as possible.
- How is the efficiency of a program affected?

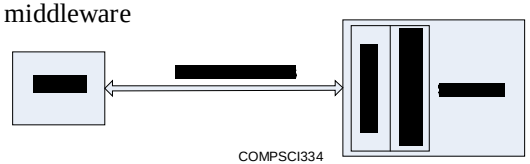


How to make a program run efficiently

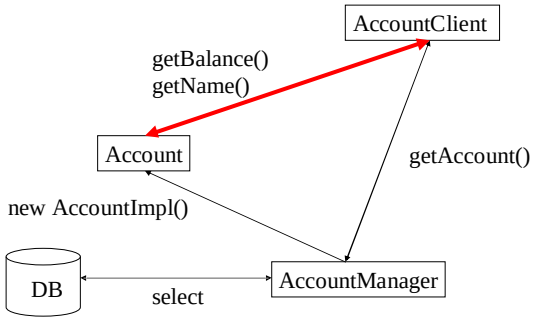
- Reduce the amount of operations involving the network
  - Only access a remote service when it is necessary
    - Send data to the machine on which the processing occurs
  - Avoid sending a large amount of data over the network
    - Process the data at the place that the data is stored
  - There is a trade-off between processing data locally and remotely
- Have as much data in the cache as possible

Process data remotely

- Process data remotely means the data will be processed at the location that the data is stored
  - Pass reference of data to the applications
  - Invoke methods on data object to execute the operations remotely
- Pros: avoid transmitting a large amount of data across the network
- Cons: there are overhead associated with the middleware



- In our banking examples, the AccountManager returns a reference of an account object to the client
  - In AccountManager
    - public Account getAccount(String name) throws RemoteException;
  - public interface Account extends Remote
- All the operations on the Account object are remote operations



Process data locally

- Process data locally means that data is sent to the client and being processed on the client's site
- Pros: avoids the overheads associated with the remote operations
- Cons: data transmission delays

- Modify our banking examples, so that the AccountManager returns an account object to the client
  - In AccountManager
    - public Account getAccount(String name) throws RemoteException
  - Account is defined as
    - public class Account implements Serializable
- Account object is returned to the client
  - All the operations on the Account object are carried out on the client machine


```
Balance of X's account is: 2468
Time to complete: 16 milliseconds

C:\data\work\Teaching\334\rmi\2008examples\ex7\byvalue>
```

```
Balance of X's account is: 2468
Time to complete: 157 milliseconds

C:\data\work\Teaching\334\rmi\2008examples\ex7\byreference>
```

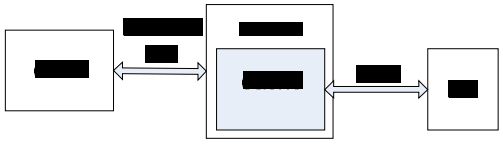
Caching within applications

- Access DB could be a time-consuming operation
  - OS and middleware overheads
  - DB might reside on a different machine
- If an application needs to use the data repeatedly and the data are not shared by other applications, the data can be cached by the application.
  - The data will be loaded into the CPU cache or main memory when the application is executed.
    - Apart from the first access to the data, data will be served from the CPU cache or memory
  - Application needs to manage the data
    - Check whether the data exist in the cache before retrieving the data from the DB
    - Before the application terminates, write the modified data back to the DB

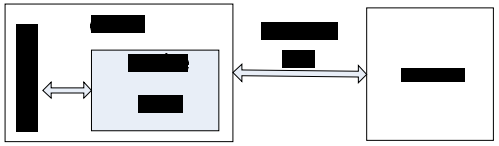
- Pros: improve the efficiency of some applications
- Cons: complicate the programming task
  - The application becomes complicated

Implementing caching

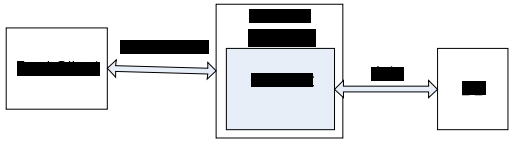
- Server side
  - The remote object implementation caches the data
    - Improve the efficiency by avoiding needless DB access
      - Reduce the load of the DB
    - The client is not aware of the existence of the cache



- Client side
  - The application running on the client manages the cache
    - Data need to be retrieved from the server first



Server side caching



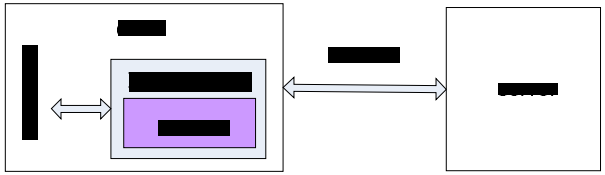
- Re-write AccountManagerImpl in the previous banking example
- Create a cache
  - private Hashtable<String,Account> accountCache = new Hashtable<String,Account>();
- For all the operations, before accessing the DB, checks the cache for the requested data. For example, for getAccount(String name)
  - Try to retrieve the object from the cache
    - account = accountCache.get(name);
  - Check whether account is null
  - If account is not null, account refers to the account object that we want.
    - Return this reference to the client

- If the object does not exist in the cache, retrieve the account information from the DB
  - String query = "select * from account where name="'+name+'"';
  - ResultSet result = statement.executeQuery(query);
- Constructs an object
  - account = new AccountImpl(accountName,balance);
  - stub = (Account) UnicastRemoteObject.exportObject(account, 0);
- Store the object in the cache
  - accountCache.put(name,stub);
- The client implementation is the same as before


```
Balance of X's account is: 2468
Time to complete: 297 milliseconds

C:\data\work\Teaching\334\rmi\2008examples\ex8\serverside>
```

```
Balance of X's account is: 2468
Time to complete: 125 milliseconds

C:\data\work\Teaching\334\rmi\2008examples\ex8\nocache>
```

Client side caching



- When getAccount is called, an Account object to returned to the client application.
- Define a LocalManager class to manage the cached data
- The client application interacts with the server through LocalManager
  - LocalManager should provide the same set of method as the remote AccountManager object
    - public Account getAccount(String name)
    - The method is not an RMI remote method
    - The Account object being returned is a local object

- The client application, BankClient, creates a LocalManager object
  - LocalManager localManager = new LocalManager();
- The client application interacts with the remote AccountManager through the LocalManager object
  - account = localManager.getAccount("X");
- Once a reference to an Account object is obtained, the client application can manipulate the object
  - The Account object is not a remote object

- The LocalManager maintains a cache
  - private Hashtable<String, Account> accountCache = new Hashtable<String, Account>();
- The LocalManager needs to retrieve the account information from the remote server.
  - remoteManager = (AccountManager) Naming .lookup("//localhost:8081/manager");
- When the client application wants to retrieve an Account object, the LocalManager needs to check to see whether the object exists in the cache first.
  - localAccount = accountCache.get(name);
  - if (localAccount != null)
- If the Account object does not exist in the cache, the LocalManager obtains the object from the remote server and stores the object in the cache.
  - localAccount = remoteManager.getAccount(name);
  - accountCache.put(name, localAccount);

```
Time to complete: 31 milliseconds

C:\data\work\Teaching\334\rmi\2008examples\ex8\clientside>
```

```
Balance of X's account is: 2468
Time to complete: 125 milliseconds

C:\data\work\Teaching\334\rmi\2008examples\ex8\nocache>
```