Lecture 14b

Remote Method Invocation

OBJECTIVES

In this lecture you will learn:

- **Understand how RMI works**
- **Learn the process of developing RMI applications**
- **Know the differences between RMI and socket-level programming**
- **Develop three-tier applications using RMI**
- **Use callbacks to develop interactive applications**

Outline

14.1 RMI Basics

RMI is the Java Distributed Object Model for facilitating communications among distributed objects.

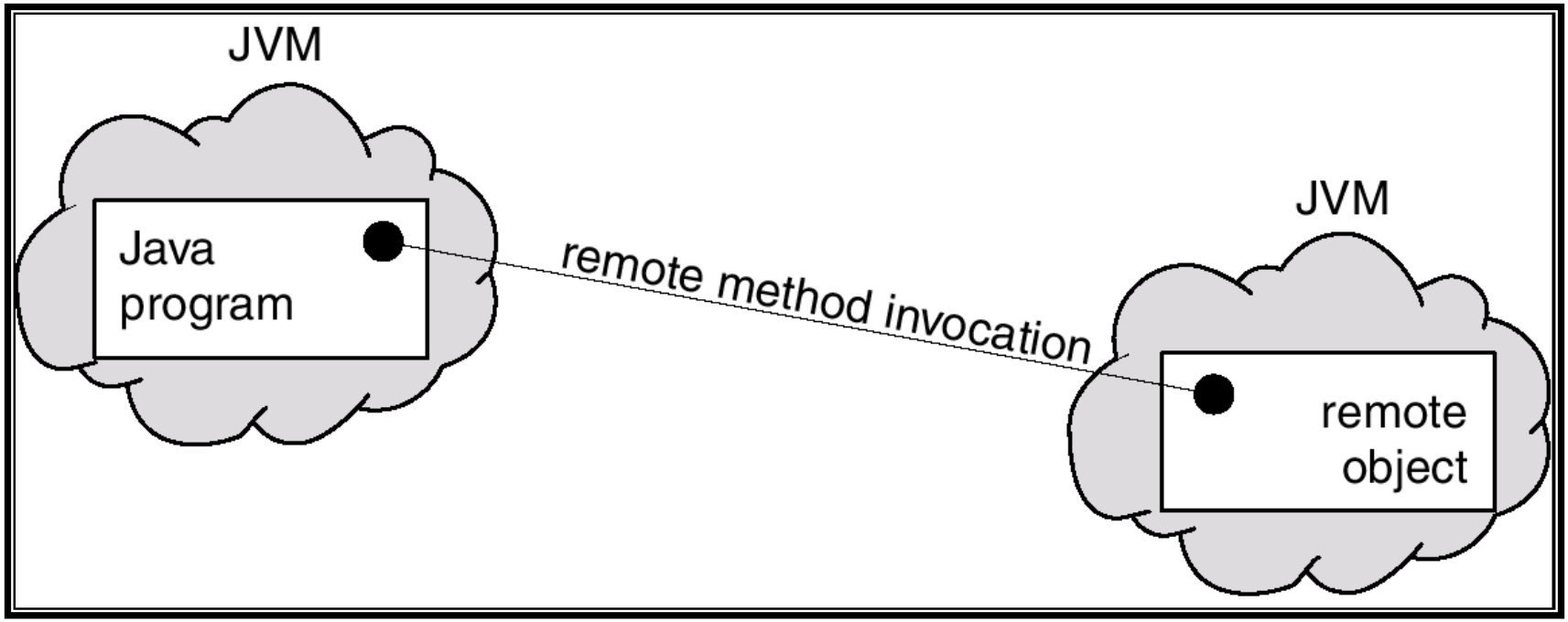
RMI is a higher-level API built on top of sockets. Socket-level programming allows you to pass data through sockets among computers.

RMI enables you not only to pass data among objects on different systems, but also to invoke methods in a remote object

Remote Method Invocation

Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.

RMI allows a Java program on one machine to invoke a method on a remote object.



14.2 The Differences between RMI and RPC

RMI is similar to Remote Procedure Calls (RPC) in the sense that both RMI and RPC enable you to invoke methods, but there are some important differences.

- **With RPC, you *call a standalone procedure*.**
- **With RMI, you invoke a method within a specific object.**
- **RMI can be viewed as object-oriented RPC.**

14.2 The Differences between RMI and Traditional Client/Server Approach

- **A RMI component can act as both a client and a server, depending on the scenario in question.**

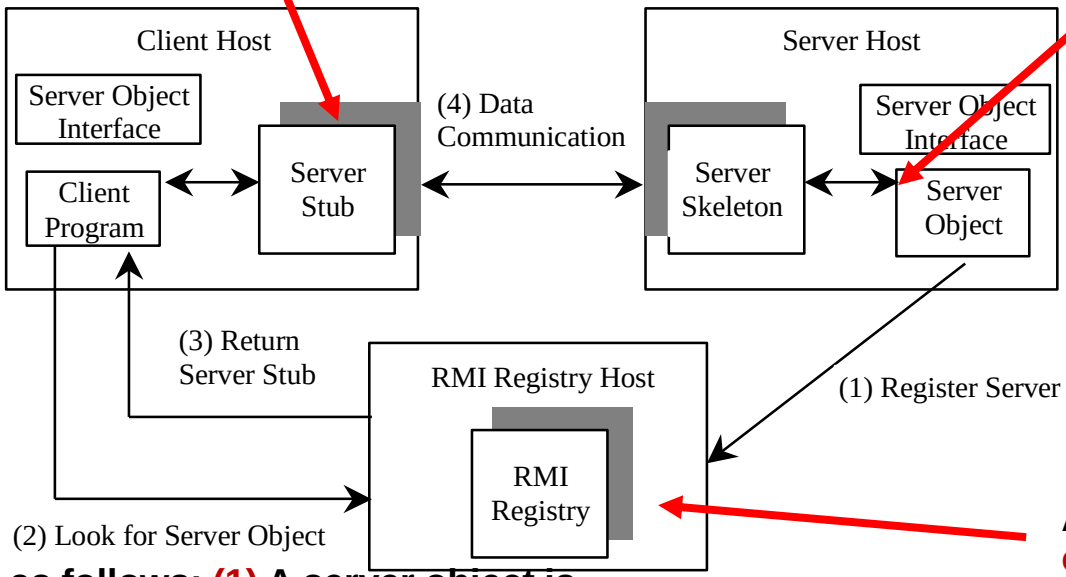
- **A RMI system can pass functionality from a server to a client and vice versa. A client/server system typically only passes data back and fourth between server and client.**

14.3 How does RMI work?

An **object that resides on the client host** and serves as a surrogate for the remote server object.

A subinterface of `java.rmi.Remote` that **defines the methods** for the server object.

An **object that resides on the server host**, communicates with the stub and the actual server object.

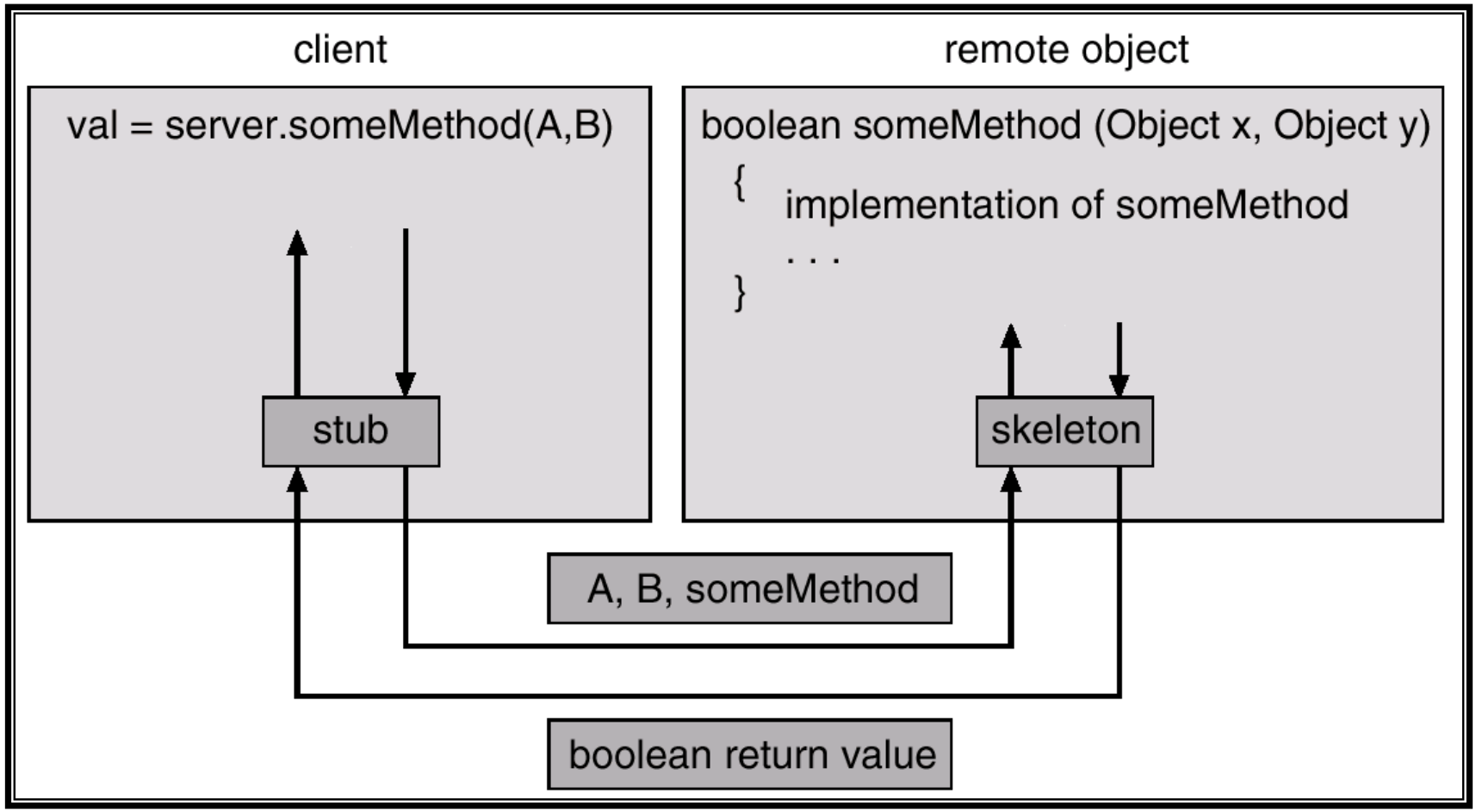


An **instance of the server object interface.**

A **utility that registers remote objects** and provides naming services for locating objects.

RMI works as follows: (1) A server object is registered with the RMI registry; (2) A client looks through the RMI registry for the remote object; (3) Once the remote object is located, its stub is returned in the client; (4) The remote object can be used in the same way as a local object. The communication between the client and the server is handled through the stub and skeleton.

Marshalling Parameters



14.4 Passing Parameters

When a client invokes a remote method with parameters, passing parameters are handled under the cover by the stub and the skeleton.

Let us consider three types of parameters:

1. Primitive data type. A parameter of primitive type such as `char`, `int`, `double`, and `boolean` is *passed by value* like a local call

14.4 Passing Parameters, cont

2. Local object type. Local object (reference) type. A parameter of local object type (reference type) such as `java.lang.String` is also passed by value. In this case, the value passed is the memory address of the object. In a remote call, there is no way to pass the object reference because the address on one machine is meaningless to a different Java VM. Any object can be used as a parameter in a remote call as long as the object is serializable. *The stub serializes the object parameter and sends it in a stream across the network*. The skeleton deserializes stream into an object.

14.4 Passing Parameters, cont

3. Remote object type. *Remote objects are passed differently from the local objects*. When a client *invokes a remote method with a parameter of some remote object type*, the *stub of the remote object is passed*. The server receives the stub and *manipulates the parameter through the stub*.

14.5 RMI Registry

- ***How does a client locate the remote object*? RMI registry provides the registry services for the server to register the object and for the client to locate the object.**
- **You can use several overloaded `static` `getRegistry()` methods in the `LocateRegistry` class to return a reference to a `Registry`. Once a `Registry` is obtained, you can `bind` an object with a `unique name` in the registry using the `bind` or `rebind` method or `locate an object` using the `lookup` method.**

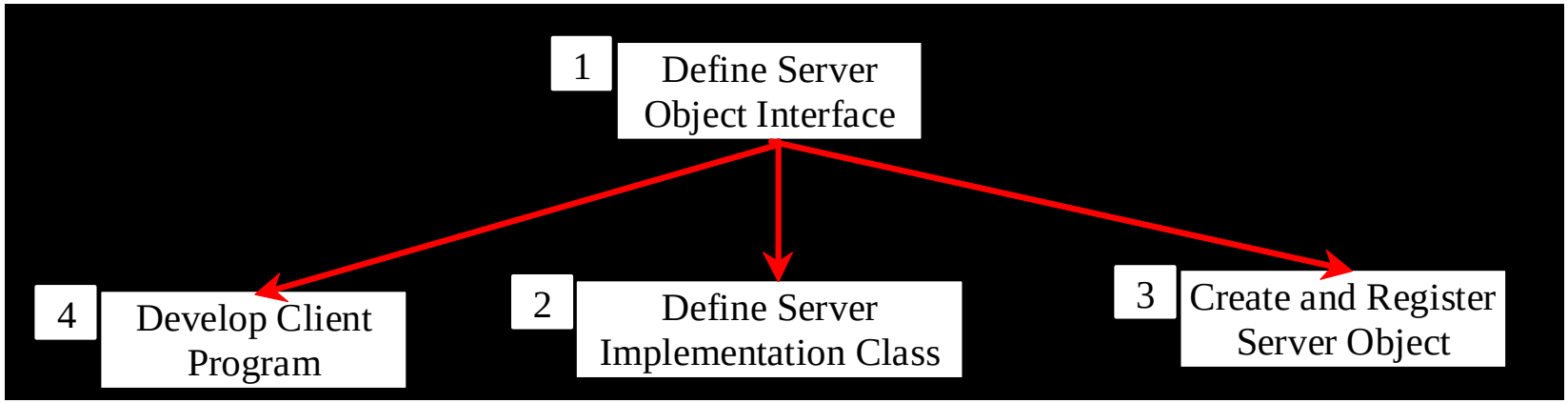
14.5 RMI Registry

java.rmi.registry.LocateRegistry	
+getRegistry(): Registry	Returns a reference to the remote object Registry for the local host on the default registry port of 1099.
+getRegistry(port: int): Registry	Returns a reference to the remote object Registry for the local host on the specified port.
+getRegistry(host: String): Registry	Returns a reference to the remote object Registry on the specified host on the default registry port of 1099.
+getRegistry(host:String, port: int): Registry	Returns a reference to the remote object Registry on the specified host and port.

14.5 RMI Registry: Binding Objects

java.rmi.registry.Registry	
+bind(name: String, obj: Remote): void	Binds the specified name with the remote object.
+rebind(name: String, obj: Remote): void	Binds the specified name with the remote object. Any existing binding for the name is replaced.
+unbind(name: String): void	Destroys the binding for the specified name that is associated with a remote object.
+list(name: String): String[]	Returns an array of the names bound in the registry.
+lookup(name: String): Remote	Returns a reference, a stub, for the remote object associated with the specified name.

14.6 Developing RMI Applications



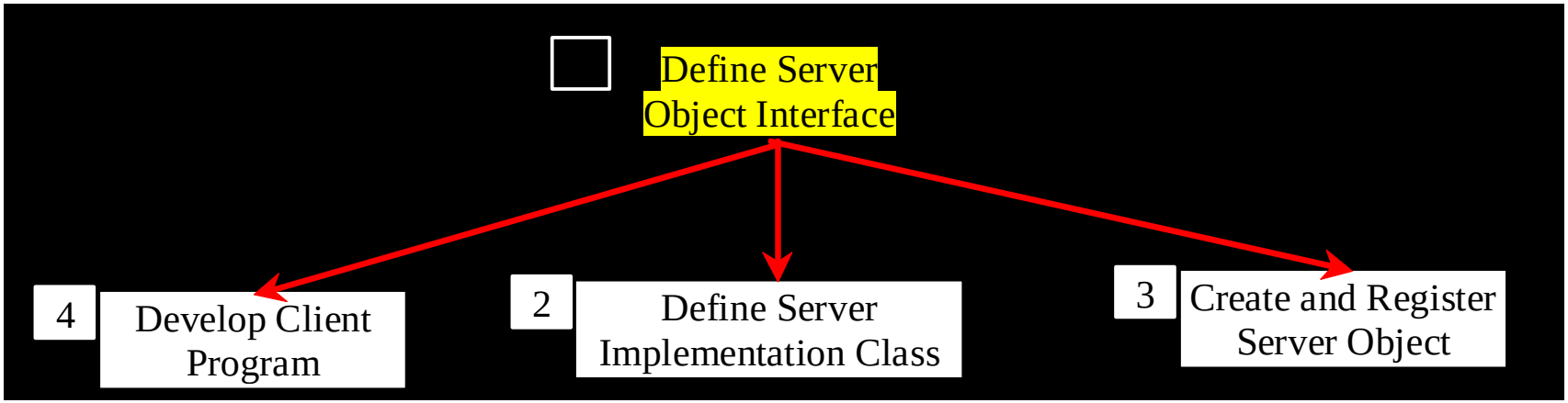
Step 1: Define Server Object Interface

1. **Define a server object interface that serves as the contract between the server and its clients, as shown in the following outline:**

```
public interface ServerInterface extends Remote {
  public void service1(...) throws   RemoteException;
  // Other methods
}
```

A server object interface must extend the `java.rmi.Remote` interface.

Step 1: Define Server Object Interface



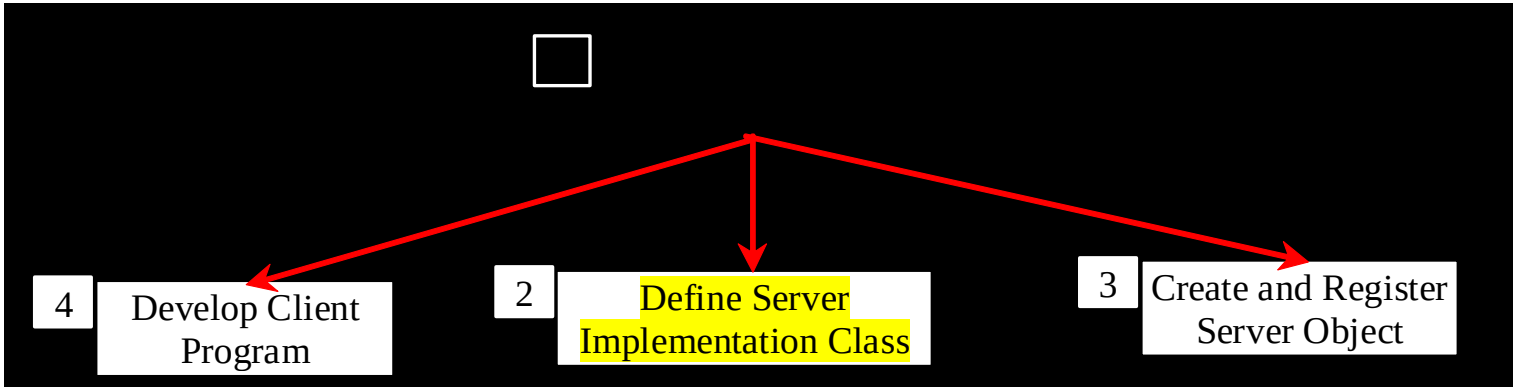
Step 2: Define Server Implementation Object

2. Define a class that implements the server object interface, as shown in the following outline:

```
public class ServerInterfaceImpl extends UnicastRemoteObject
                                 implements ServerInterface
 {
    public void service1(...) throws RemoteException {
      // Implement it
    }
    // Implement other methods
 }
```

The server implementation class must **extend** the `java.rmi.server.UnicastRemoteObject` class. The `UnicastRemoteObject` class provides *support for point-to-point active object references using TCP* streams.

Step 2: Define Server Implementation Object

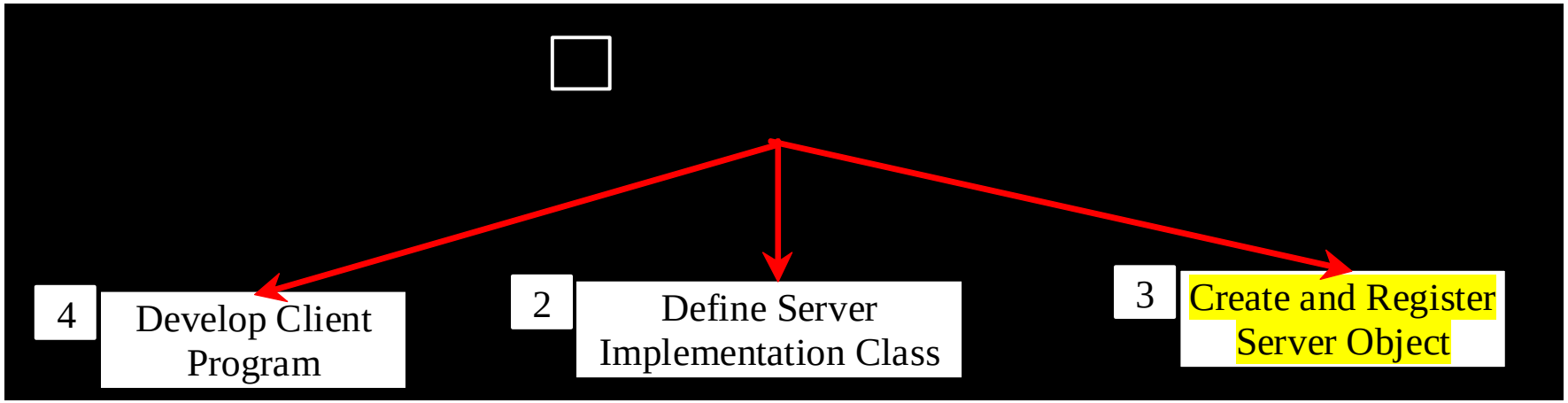


Step 3: Create and Register Server Object

3. Create a server object from the server implementation `class` and register it with an `RMI` registry:

```
ServerInterface obj = new ServerInterfaceImpl(...);
Registry registry       =
                    LocateRegistry.createRegistry(1099);
registry.rebind("RemoteObjectName", obj);
```


Step 3: Create and Register Server Object

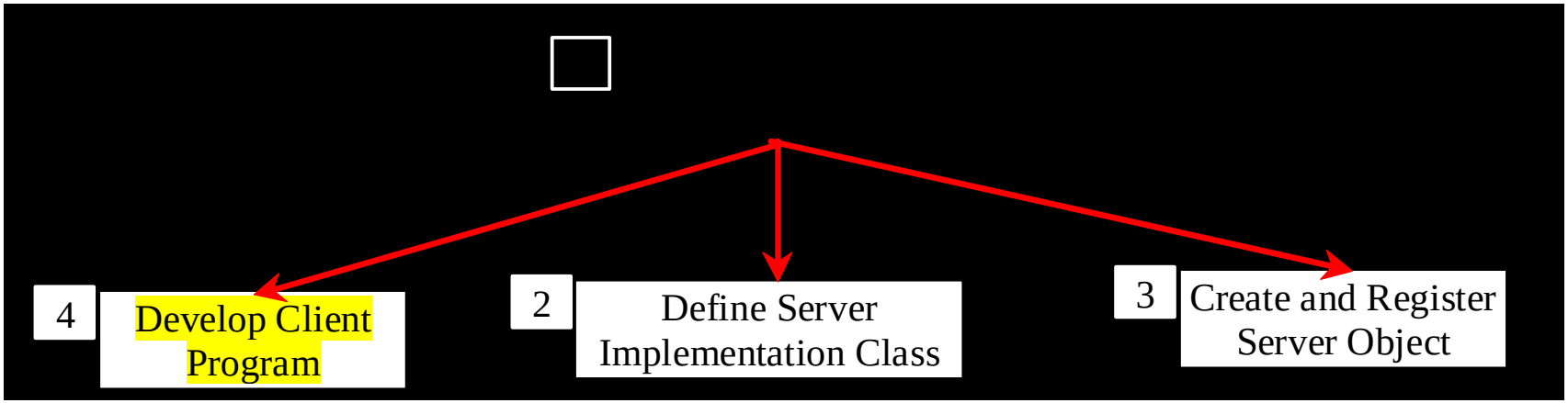


Step 4: Develop Client Program

4. Develop a client that locates a remote object and invokes its methods, as shown in the following outline:

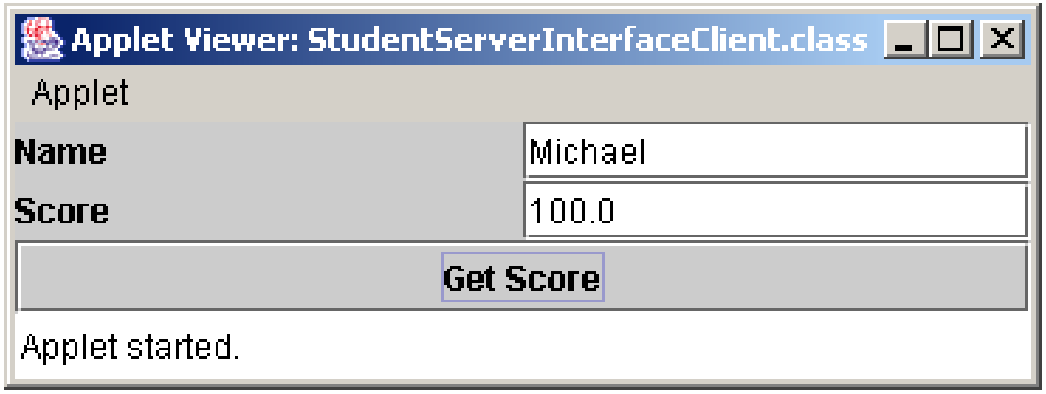
```
Registry registry = LocateRegistry.getRegistry(host);
ServerInterface server =
      (ServerInterface)registry.lookup("RemoteObjectName");
server.service1(...);
```


Step 4: Develop Client Program



14.7 Example: Retrieving Student Scores from an RMI Server

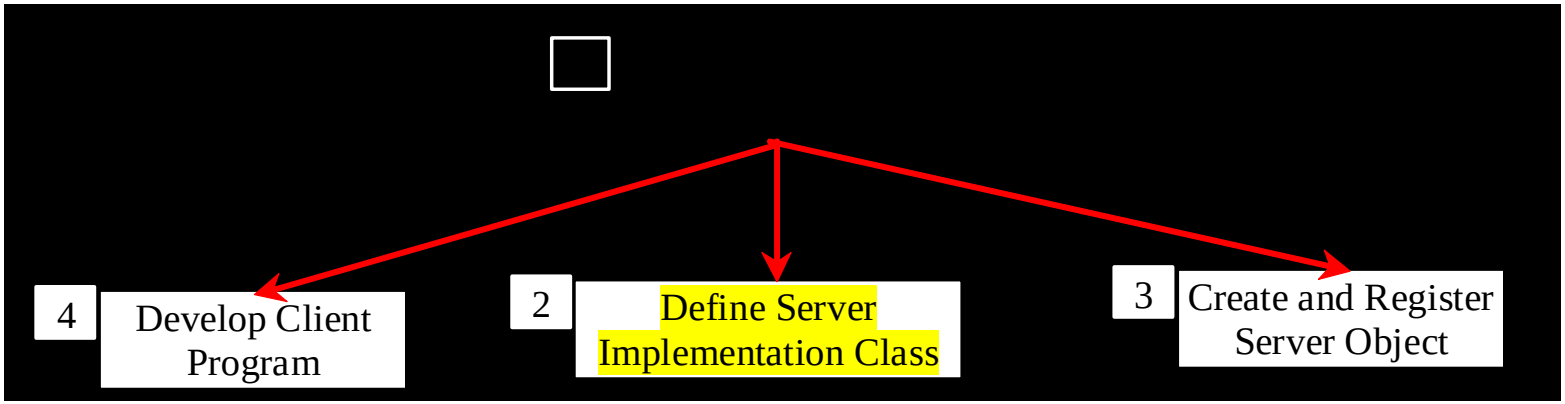
- **Problem (RMIsample1): : This example creates a client that retrieves student scores from an RMI server.**



14.7 Step 1: Define Server Object Interface

1. Create a server interface named StudentServerInterface. The interface tells the client how to invoke the server's findScore method to retrieve a student score.

StudentServerInterface



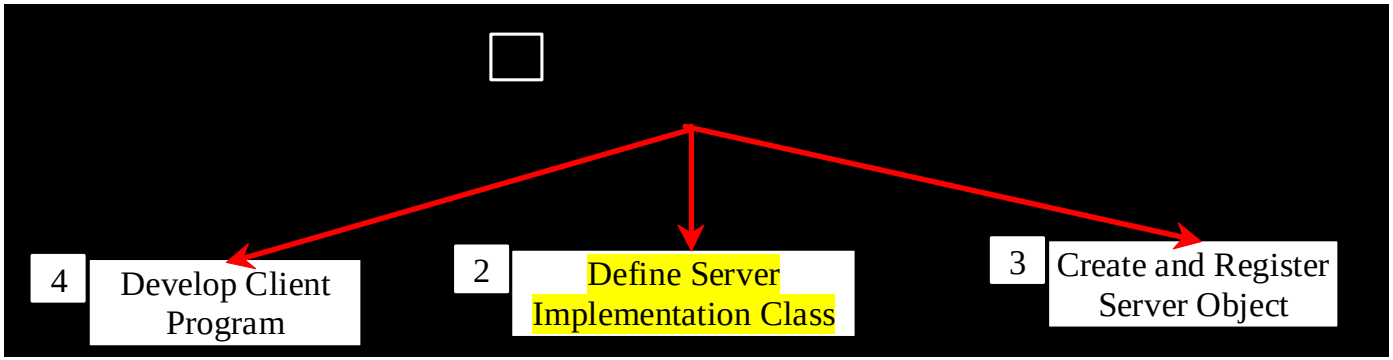
Step 1: Define Server Object Interface

```
 1  import java.rmi.*;
 2
 3  public interface StudentServerInterface extends Remote {
 4  /**
 5   * Return the score for specified the name
 6   * @param   name   the student name
 7   * @return  an double score or -1 if the student is not found
 8   */
 9    public double findScore(String name) throws RemoteException;
10 }
```


14.7 Step 2: Define Server Implementation Object

- **2. Create server implementation named StudentServerInterfaceImpl that implements StudentServerInterface. The findScore method returns the score for a specified student. This method returns -1 if the score is not found.**

StudentServerInterfaceImp



14.7 Step 2: Define Server Implementation Object

```
1  public class StudentServerInterfaceImpl extends UnicastRemoteObject
2                                           implements StudentServerInterface {
3   // Stores scores in a map indexed by name
4   private HashMap scores = new HashMap();
5
6   public StudentServerInterfaceImpl() throws RemoteException {
7     initializeStudent();
8   }
9
10  /** Initialize student information */
11  protected void initializeStudent() {
12    scores.put("John", new Double(90.5));
13    scores.put("Michael", new Double(100));
14    scores.put("Michelle", new Double(98.5));
15  }
16
```

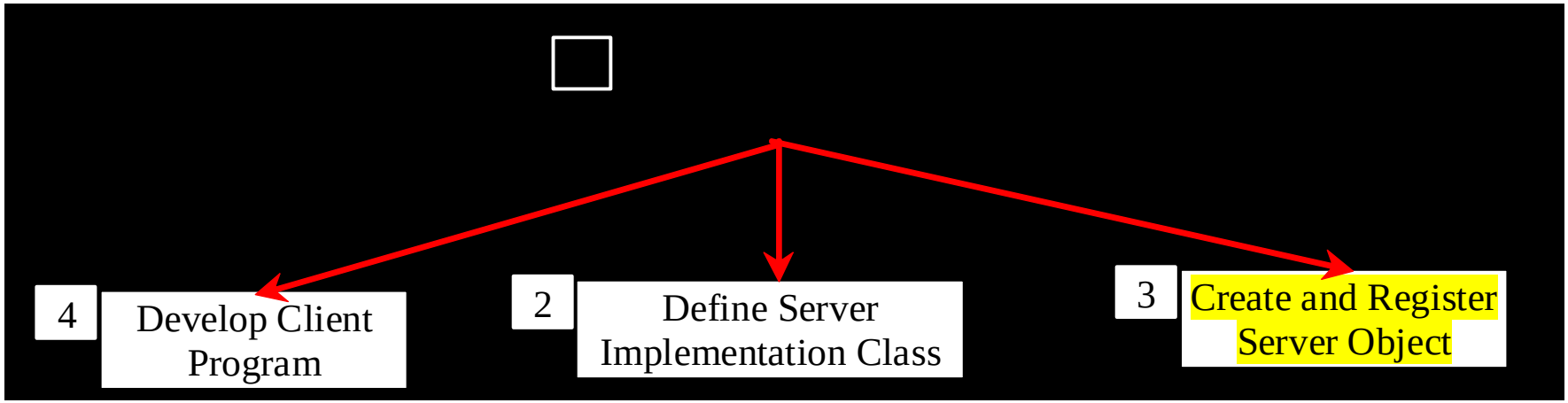

14.7 Step 2: Define Server Implementation Object

```
17  /** Implement the findScore method from the Student interface */
18  public double findScore(String name) throws RemoteException {
19    Double d = (Double)scores.get(name);
20
21    if (d == null) {
22      System.out.println("Student " + name + " is not found ");
23      return -1;
24    }
25    else {
26      System.out.println("Student " + name + "\'s score is "
27        + d.doubleValue());
28      return d.doubleValue();
29    }
30  }
```


14.7 Step 3: Create and Register Server Object

3. Create a server object from the server implementation class and register it with an RMI registry.

RegisterWithRMIServer



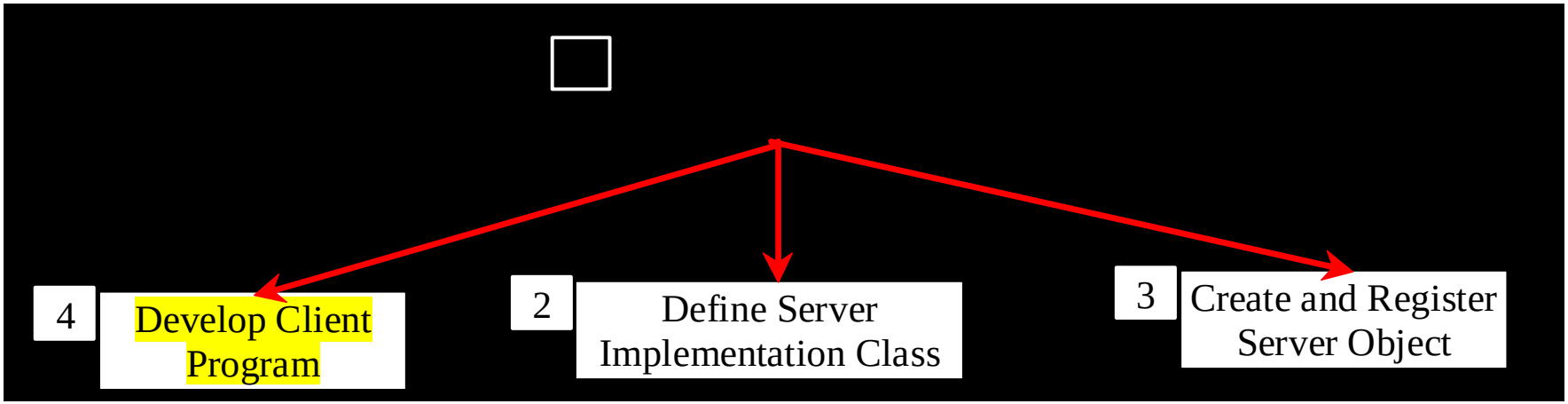
Step 3: Create and Register Server Object

```
 1 import java.rmi.registry.*;
 2
 3 public class RegisterWithRMIServer {
 4  /** Main method */
 5  public static void main(String[] args) {
 6    try {
 7      StudentServerInterface obj = new StudentServerInterfaceImpl();
 8      Registry registry = LocateRegistry.createRegistry(1099);
 9      registry.rebind("StudentServerInterfaceImpl", obj);
10      System.out.println("Student server " + obj + " registered");
11    }
12    catch (Exception ex) {
13      ex.printStackTrace();
14    }
15  }
16 }
```


14.7 Step 4: Develop Client Program

- **4. Create a client as an applet named StudentServerInterfaceClient. The client locates the server object from the RMI registry, uses it to find**

StudentServerInterfaceClient



Step 4: Develop Client Program

```
1  public class StudentServerInterfaceClient extends JApplet {
2   // Declare a Student instance
3   private StudentServerInterface student;
4
5   private boolean isStandalone; // Is applet or application
6
7   private JButton jbtGetScore = new JButton("Get Score");
8   private JTextField jtfName = new JTextField();
9   private JTextField jtfScore = new JTextField();
10
11  public void init() {
12    // Initialize RMI
13    initializeRMI();
14
15    JPanel jPanel1 = new JPanel();
16    jPanel1.setLayout(new GridLayout(2, 2));
17    jPanel1.add(new JLabel("Name"));
18    jPanel1.add(jtfName);
19    jPanel1.add(new JLabel("Score"));
20    jPanel1.add(jtfScore);
21
22    add(jbtGetScore, BorderLayout.SOUTH);
23    add(jPanel1, BorderLayout.CENTER);
24
25    jbtGetScore.addActionListener(new ActionListener() {
26      public void actionPerformed(ActionEvent evt) {
27        getScore();
28      }
29    });
30  }
```


Step 4: Develop Client Program

```
31
32  private void getScore() {
33    try {
34      // Get student score
35      double score = student.findScore(jtfName.getText().trim());
36
37      // Display the result
38      if (score < 0)
39        jtfScore.setText("Not found");
40      else
41        jtfScore.setText(new Double(score).toString());
42    }
43    catch(Exception ex) {
44      ex.printStackTrace();
45    }
46  }
```


Step 4: Develop Client Program

```
64   /** Main method */
65   public static void main(String[] args) {
66     StudentServerInterfaceClient applet =
67                                             new StudentServerInterfaceClient();
68     applet.isStandalone = true;
69     JFrame frame = new JFrame();
70     frame.setTitle("StudentServerInterfaceClient");
71     frame.add(applet, BorderLayout.CENTER);
72     frame.setSize(250, 150);
73     applet.init();
74     frame.setLocationRelativeTo(null);
75     frame.setVisible(true);
76     frame.setDefaultCloseOperation(3);
77   }
78 }
```


Step 4: Develop Client Program

```
47
48  /** Initialize RMI */
49  protected void initializeRMI() {
50    String host = "";
51    if (!isStandalone) host = getCodeBase().getHost();
52
53    try {
54      Registry registry = LocateRegistry.getRegistry(host,1099);
55      student = (StudentServerInterface) registry.lookup("StudentServerInterfaceImpl");
56
57      System.out.println("Server object " + student + " found");
58    }
59    catch(Exception ex) {
60      System.out.println(ex);
61    }
62  }
63
```


14.7 Run Example

- 1. Start RMI Registry by typing "`start rmiregistry`" at a DOS prompt from the `book` directory. By default, the `port number 1099` is used by `rmiregistry`. To use a different `port` number, simply type the command "`start rmiregistry portnumber`" at a DOS prompt

Start RMI

2. Start `RegisterWithRMIServer` using the following command at the current directory:

```
java RegisterWithRMIServer
```

Register Object with RMI Registry

(steps 1 & 2 are not necessary when running the application in Netbeans)

3. Run `StudentServerInterfaceClient` as an application.

Run

14.8 RMI vs. Socket-Level Programming

- **RMI enables you to program at a higher level of abstraction. It hides the details of socket server, socket, connection, and sending or receiving data. It even implements a multithreading server under the hood, whereas with socket-level programming you have to explicitly implement threads for handling multiple clients.**

14.8 RMI vs. Socket-Level Programming

- **RMI applications are scalable and easy to maintain. You can change the RMI server or move it to another machine without modifying the client program except for resetting the URL to locate the server. (To avoid resetting the URL, you can modify the client to pass the URL as a command-line parameter.) In socket-level programming, a client operation to send data requires a server operation to read it. The implementation of client and server at the socket-level is tightly synchronized.**

14.8 RMI vs. Socket-Level Programming

- **RMI clients can directly invoke the server method, whereas socket-level programming *is limited to passing values*.**
- **Socket-level programming is very primitive. *Avoid using it to develop client/server applications*. As an analogy, socket-level programming is like programming in assembly language, while RMI programming is like programming in a high-level language.**

14.9 Developing Three-Tier Applications Using RMI

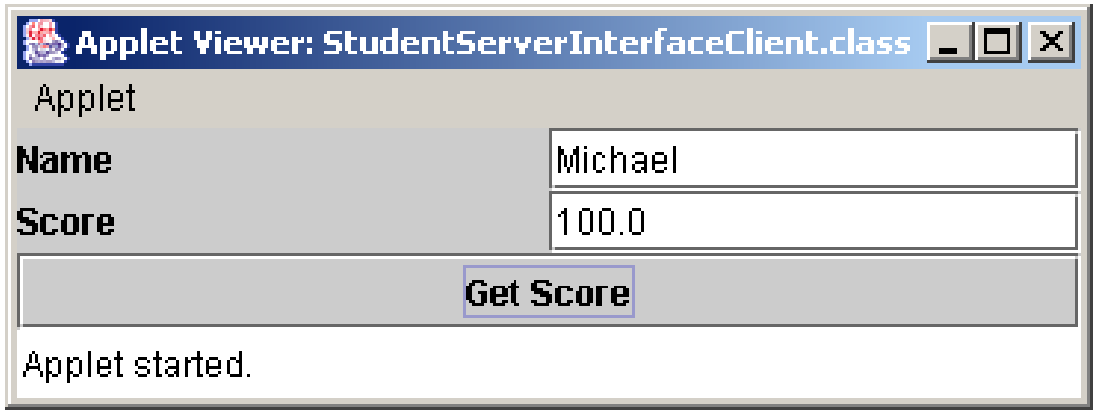
- **Three-tier applications have gained considerable attention in recent years, largely because of the demand for more scalable and load-balanced systems to replace traditional two-tier client/server database systems. A centralized database system not only handles data access but also processes the business rules on data. Thus, a centralized database is usually heavily loaded because it requires extensive data manipulation and processing. In some situations, data processing is handled by the client and business rules are stored on the client side. It is preferable to use a middle tier as a buffer between a client and the database. The middle tier can be used to apply business logic and rules, and to process data to reduce the load on the database.**

14.9 Developing Three-Tier Applications Using RMI

- **A three-tier architecture does more than just reduce the processing load on the server. It also provides access to multiple network sites. This is especially useful to Java applets that need to access multiple databases on different servers, since an applet can only connect with the server from which it is downloaded.**

14.9 Example: Retrieving Student Scores on a Database Using RMI

Problem (RMIsample2): This example rewrites the preceding example to find scores stored in a database rather than a hash map. In addition, the system is capable of blocking a client from accessing a student who has not given the university permission to publish his/her score. An RMI component is developed to serve as a middle tier between client and database; it sends a search request to the database, processes the result, and returns an appropriate value to the client.

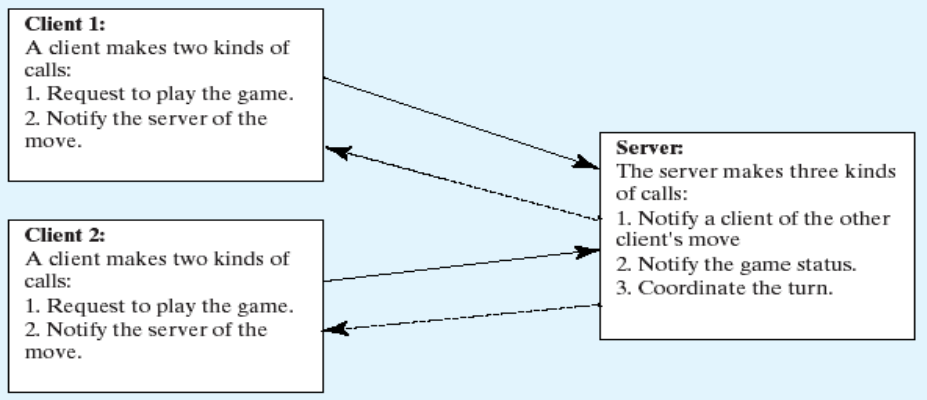


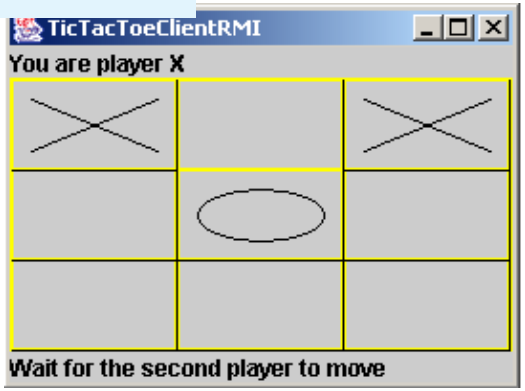
14.10 RMI Call Backs

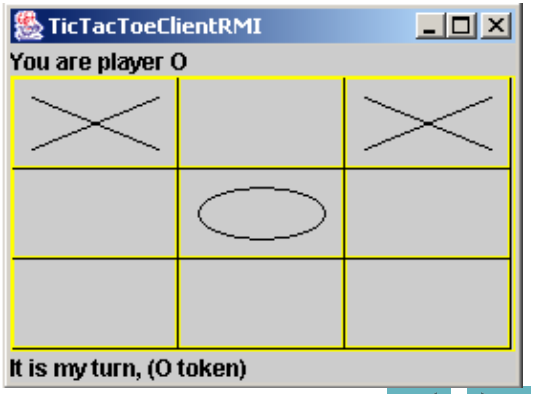
In a traditional client/server system, *a client sends a request to a server, and the server processes the request and returns the result to the client.* The server cannot invoke the methods on a client. One of the important benefits of RMI is that it supports *callbacks*, which enable the server to invoke the methods on the client. With the RMI callback feature, you can develop interactive distributed applications.

14.10 Example: Distributed TicTacToe Using RMI

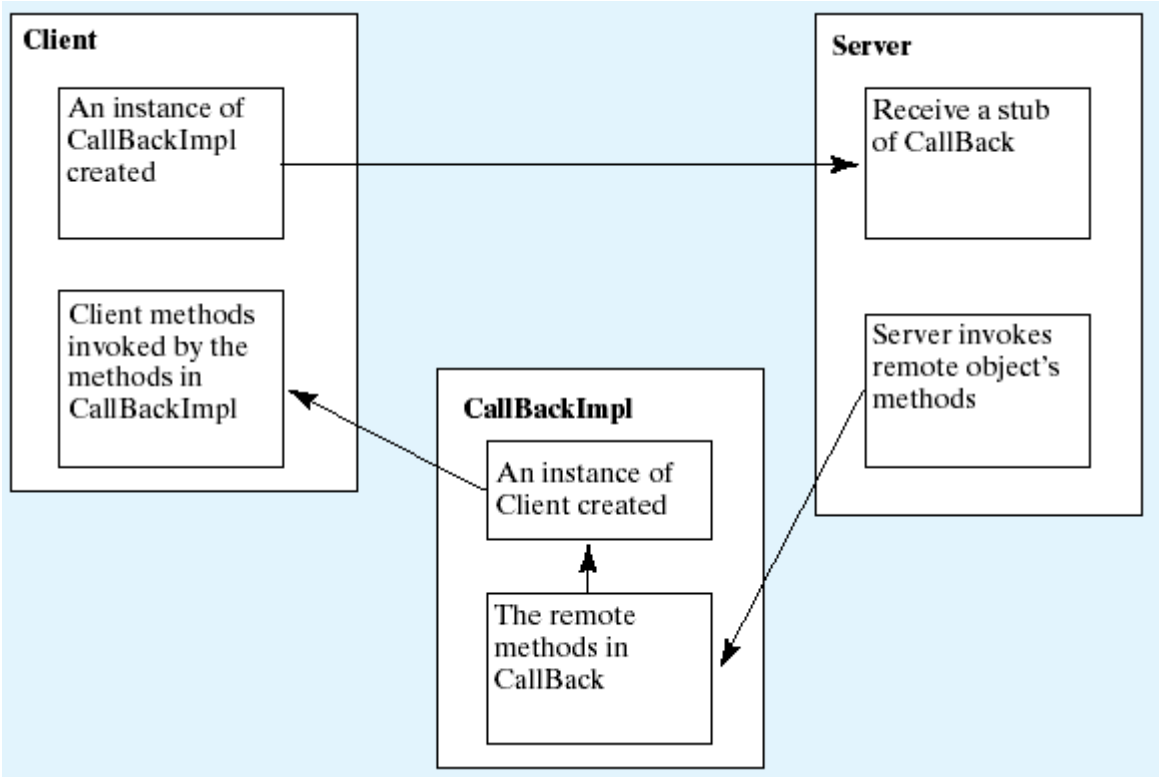
Example (RMIsample3): , "Distributed TicTacToe Game," was developed using stream socket programming. Write a new distributed TicTacToe game using the RMI.







14.10 Example: Distributed TicTacToe Using RMI



TicTacToeInterface

CallBack

TicTacToeImpl

CallBackImpl

TicTacToeClientRMI

Define Server Object Interface

```
 1 import java.rmi.*;
 2
 3  public interface TicTacToeInterface extends Remote {
 4  /**
 5   * Connect to the TicTacToe server and return the token.
 6   * If the returned token is ' ', the client is not connected to
 7   * the server
 8   */
 9     public char connect(CallBack client) throws RemoteException;
10
11  /** A client invokes this method to notify the server of its move*/
12    public void myMove(int row, int column, char token)
13                                        throws RemoteException;
14 }
```


Define Server Implementation Object

```
 1public class TicTacToeImpl extends UnicastRemoteObject
 2                           implements TicTacToeInterface {
 3  // Declare two players, used to call players back
 4  private CallBack player1 = null;
 5  private CallBack player2 = null;
 6
 7  // board records players' moves
 8  private char[][] board = new char[3][3];
 9
10  /** Constructs TicTacToeImpl object and exports it on default port.
11   */
12  public TicTacToeImpl() throws RemoteException {
13    super();
14  }
15
16  /** Constructs TicTacToeImpl object and exports it on specified
17   * port.
18   * @param port The port for exporting
19   */
20  public TicTacToeImpl(int port) throws RemoteException {
21    super(port);
22  }
23
```


Define Server Implementation Object

```
24  /**
25   * Connect to the TicTacToe server and return the token.
26   * If the returned token is ' ', the client is not connected to
27   * the server
28   */
29  public char connect(CallBack client) throws RemoteException {
30    if (player1 == null) {
31      // player1 (first player) registered
32      player1 = client;
33      player1.notify("Wait for a second player to join");
34      return 'X';
35    }
36    else if (player2 == null) {
37      // player2 (second player) registered
38      player2 = client;
39      player2.notify("Wait for the first player to move");
40      player2.takeTurn(false);
41      player1.notify("It is my turn (X token)");
42      player1.takeTurn(true);
43      return 'O';
44    }
45    else {
46      // Already two players
47      client.notify("Two players are already in the game");
48      return ' ';
49    }
50  }
```


Define Server Implementation Object

```
51
52  /** A client invokes this method to notify the server of its move*/
53  public void myMove(int row, int column, char token) throws RemoteException
54 {
55    // Set token to the specified cell
56    board[row][column] = token;
57
58    // Notify the other player of the move
59    if (token == 'X')
60      player2.mark(row, column, 'X');
61    else
62      player1.mark(row, column, 'O');
63
64    // Check if the player with this token wins
65    if (isWon(token)) {
66      if (token == 'X') {
67        player1.notify("I won!");
68        player2.notify("I lost!");
69        player1.takeTurn(false);
70      }
71      else {
72        player2.notify("I won!");
73        player1.notify("I lost!");
74        player2.takeTurn(false);
75      }
76    }
77    else if (isFull()) {
78      player1.notify("Draw!");
79      player2.notify("Draw!");
80    }
81    else if (token == 'X') {
82      player1.notify("Wait for the second player to move");
83      player1.takeTurn(false);
84      player2.notify("It is my turn, (O token)");
85      player2.takeTurn(true);
86    }
87    else if (token == 'O') {
88      player2.notify("Wait for the first player to move");
89      player2.takeTurn(false);
90      player1.notify("It is my turn, (X token)");
91      player1.takeTurn(true);
92    }
93  }
```


Create and Register Server Object

```
118  /** Check if the board is full */
119  public boolean isFull() {
120    for (int i = 0; i < 3; i++)
121      for (int j = 0; j < 3; j++)
122        if (board[i][j] == '\u0000')
123          return false;
124
125    return true;
126  }
127
128  public static void main(String[] args) {
129    try {
130      TicTacToeInterface obj = new TicTacToeImpl();
131      Registry registry = LocateRegistry.createRegistry(1099);
132      registry.rebind("TicTacToeImpl", obj);
133      System.out.println("Server " + obj + " registered");
134    }
135    catch (Exception ex) {
136      ex.printStackTrace();
137    }
138  }
139}
```


Develop Client Program

```
 1
 2  public interface CallBack extends Remote {
 3  /** The server notifies the client for taking a turn */
 4  public void takeTurn(boolean turn) throws RemoteException;
 5
 6  /** The server sends a message to be displayed by the client */
 7  public void notify(java.lang.String message)
 8                                          throws RemoteException;
 9
10  /** The server notifies a client of the other player's move */
11  public void mark(int row, int column, char token)
12                                           throws RemoteException;
13}
```


Develop Client Program – the CallBack interface Implementation

```
 1  public class CallBackImpl  extends UnicastRemoteObject
 2                             implements CallBack {
 3  // The client will be called by the server through callback
 4  private TicTacToeClientRMI thisClient;
 5
 6  /** Constructor */
 7  public CallBackImpl(Object client) throws RemoteException {
 8    thisClient = (TicTacToeClientRMI)client;
 9  }
10
11  /** The server notifies the client for taking a turn */
12  public void takeTurn(boolean turn) throws RemoteException {
13    thisClient.setMyTurn(turn);
14  }
15
16  /** The server sends a message to be displayed by the client */
17  public void notify(String message) throws RemoteException {
18    thisClient.setMessage(message);
19  }
20
21  /** The server notifies a client of the other player's move */
22  public void mark(int row, int column, char token)
23      throws RemoteException {
24    thisClient.mark(row, column, token);
25  }
26}
```


Develop Client Program – the CallBack interface Implementation

```
47  /** Initialize RMI */
48  protected boolean initializeRMI() throws Exception {
49    String host = "";
50    if (!isStandalone) host = getCodeBase().getHost();
51
52    try {
53      Registry registry = LocateRegistry.getRegistry(host);
54      ticTacToe = (TicTacToeInterface)registry.lookup("TicTacToeImpl");
55      System.out.println("Server object " + ticTacToe + " found");
56    }
57    catch (Exception ex) {
58      System.out.println(ex);
59    }
60
61    // Create callback for use by the server to control the client
62    CallBackImpl callBackControl = new CallBackImpl(this);
63
64    if (
65      (marker = ticTacToe.connect((CallBack)callBackControl)) != ' ')
66    {
67      System.out.println("connected as " + marker + " player.");
68      jlblIdentification.setText("You are player " + marker);
69      return true;
70    }
71    else {
72      System.out.println("already two players connected as ");
73      return false;
74    }
75  }
```


14 Summary

Problem 1

Implement a simple remote calculator service and use it from a client program. The calculator service should provide basic arithmetic operations.