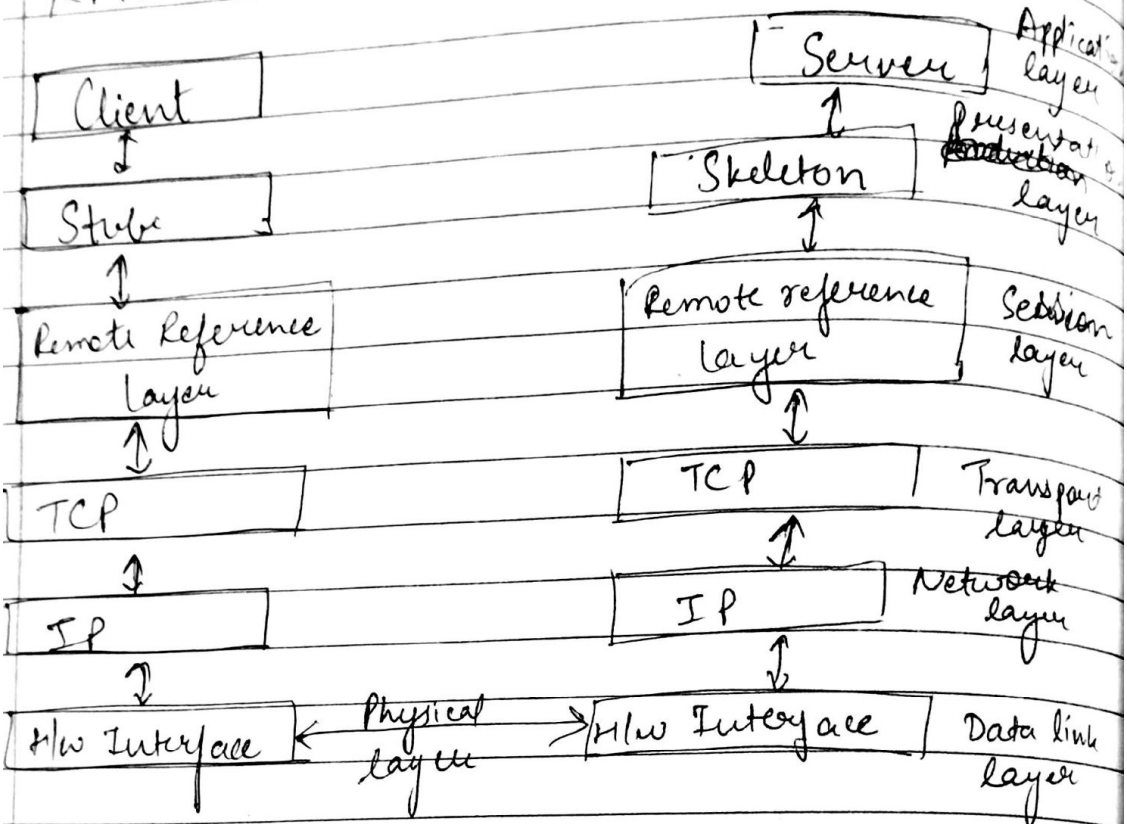
Remote Method Invocation

It is used when we want to access method of object of remote machine

RMI Architecture



It works on 3 layers.

Stub | Skeleton layer

This layer is responsible to route the method calls made by the client to the interface reference and redirect these calls to remote objects

Stub: Stubs are client side components that acts as a proxy for remote object.

Foll. are the functions performed by stub.

1. It initiates connection with remote JVM having the desired remote object.
2. It marshalls the parameters to remote JVM.
Marshalling is converting (data) contained bit string to byte string and vice versa is unmarshalling ;i.e it writes and transmits data in the form of bytes to remote JVM.
3. It unmarshalls the return value or exception returned i.e return value from remote object is converted back from byte stream to data stream format.
4. This data value is then returned to caller.

Skeleton: It is a server side component that communicates with remote object fell. are the functions performed by skeleton:
(i) The marshalled parameters sent by stub are unmarshalled by skeleton.
(ii) Skeleton invokes the method on actual remote object of ~~invocation~~ implementati
(iii) It marshalls result obtained from remote method implementation that means return value or exception returned is converted to byte string and transmitted over RMI link to the client stub.

2. Remote Reference layer: defines the invocation semantics of RMI. It provides the object that represents the handle to remote object using java remote method protocol. With the help of this layer single proxy can send method request to multiple implementations simultaneously.

3. Transport layer: It is a binary data protocol that sends remote object requests. It makes use of TCP/IP for communication b/w 2 JVMs which is connection oriented protocol. Generally transport layer prefers to use multiple TCP/IP connections b/w client & server but in RMI mechanism transport layer multiplexes several virtual connections within single TCP/IP connection.

4th Jan

Creating an RMI Application:

1.) Create the Remote Interface
2.) Provide implementation of interface
3.) Complete implementation class using rmic tool to create stub/skeleton class.
4.) Start rmi registery
5.) Create & Run Server class
6.) Create client class

First Run Server
↳ Run Client.

1) Creating RMI interface [Adder.java]

```
import java.rmi.*;
public interface Adder extends Remote
{
public int add(int x, int y) throws RemoteException;
}
```

2) Implementing interface [AdderRemote.java]

```
import java.rmi.*;
import java.rmi.server.*;
public class AdderRemote extends UnicastRemoteObject implements Adder
{
AdderRemote() throws RemoteException
{
super();
}
public int add(int x, int y)
{
return (x+y);
}
}
```

To Run:

rmic AdderRemote

Start rmi registry.

// 3 Registering Remote object [myServer.java]

```
import java.rmi.*;
import java.rmi.registry.*;
public class myServer
{
public static void main(String arg[])
{
try
{
Adder stub = new AdderRemote();
Naming.rebind("rmi://localhost:1234/sta",stub);
}
catch(Exception e)
{
System.out.println(e.getMessage());
}
}
```

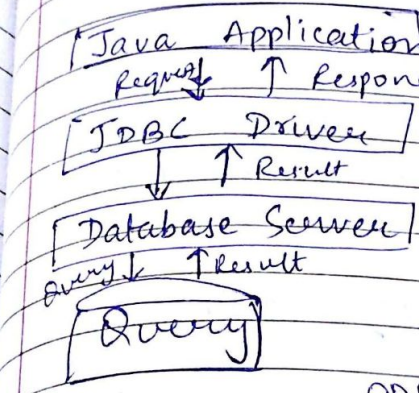
4. // Creating client [myClient.java]

```
import java.rmi.*;
public class myClient
{
public static void main(String arg[])
{
try
{
Adder stub = (Adder) Naming.lookup("rmi://
localhost:1234/sta");
System.out.println(stub.add(34,3));
}
catch(Exception e)
{
System.out.println(e.getMessage());
}
}
```

DELTA Page No. 7
Date 10/01/2017

JDBC (Java DataBase Connectivity)

JDBC Architecture

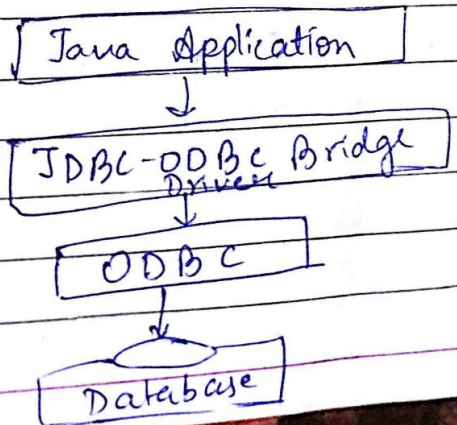


ODBC - Open Database connectivity

JDBC Driver: specification classifies JDBC Drivers into 4 types:

Type 1: JDBC-ODBC Bridge driver
Type 2: Native API / Partly Java Driver
Type 3: Net Protocol All Java Driver
Type 4: Pure Java Driver

JDBC-ODBC Bridge Driver: This driver translate JDBC calls into ODBC calls and sends them to ODBC driver ODBC is a generic API. In this case database must be present on same client machine.

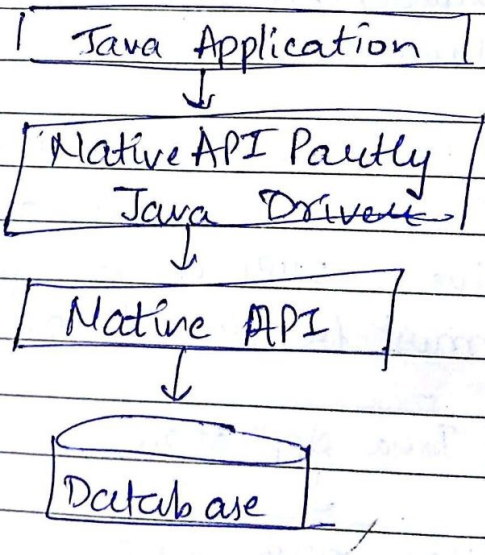


Merits: This driver can be used to access any database.

Demerits:
(i) It is slowest driver because calls are sent to ODBC driver and then to database connectivity interface
(ii) Not suitable for large applications
(iii) For this type of driver database must be present on same machine.

Type 2 Native API/Partly Java Driver

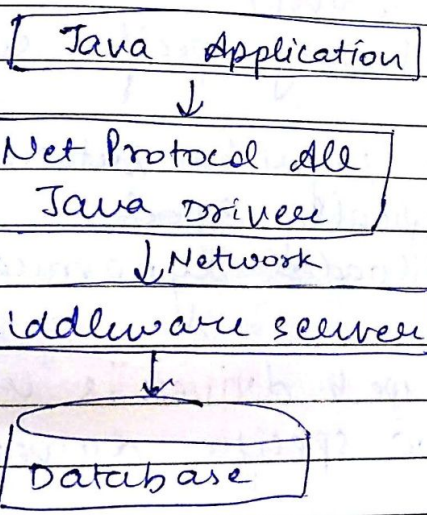
This driver translates all JDBC calls into database-specific calls, so this driver directly communicates with database server so JDBC calls are after converting to database specific calls are given to Native API



Merits: As JDBC calls are converted directly into database specific calls. It gives better performance than type 1 driver.

Demerits
1) Library of reqd. database must be loaded on client machine
2) can't be used for internet applications.
3) If some modification is done in database then same modification must be done in Native API also.

Type 3 Net Protocol All Java Driver: This type of driver is used to access middleware server which then translates the request to database specific connectivity interface and then the request is given to database server. This driver is fully written in java.



Merits: 1) It is portable because it is fully written in java and can be used for internet applications.

(ii) The performance of this driver can be optimized.
(iii) No need to keep library of reqd database on client machine.
(iv) This driver supports many advance feature such as load balancing, caching & logging
(v) Using this driver multiple databases can be accessed using single driver.

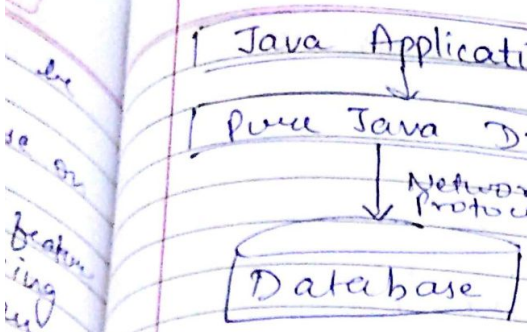
Demerits: Middleware server applications need to be installed.

Type 4: Pure Java Driver

→ This driver converts JDBC calls to network protocol directly understood by database.
→ This driver is also fully implemented in Java.

Merits: same as above(i)
(ii) No need to install any specific software on client m/c.
(iii) No translation layer in between ~~perfo~~ so performance is typically good.
iv) It can be downloaded dynamically.

Demerit: When type 4 driver is used then for each database a specific driver will be needed.



Creating JDBC Application

1) Import packages
2) Loading / registering the Driver
3) Creating / opening connection with the database.
4) Executing query through Statement object.
5) Executing from ResultSet Object
6) Display data
7) Closing Connection.

eg

EMPLOYEE

ID	Age	firstname	Last Name
101	31	Ramesh	Kumar

```
//step1: Import packages
  import java.sql.*;
  public class FirstExample
  {
Static final String JDBC-DRIVER = "com.mysql.jdbc.Driver";
static final String URL = "jdbc:mysql://localhost/EMP";
Static final String User = "username";
static final String Pass = "password";
```

```
public static void main(String args[]).
{
Connection con = null;
Statement stmt;
try // Step2: Loading driver.
{
Class.forName(JDBC-DRIVER);
System.out.println("connecting to database");

// Step3: Opening Connection
    con = DriverManager.getConnection
                    (URL, User, Pass);
// Step 4: ~~Opening Connection~~ Execute Query
Stmt ~~Stmt~~ = con.CreateStatement();
String query = "Select * from Employee";
ResultSet rs = ~~stmt~~ stmt.executeQuery(query);

// Step5: Extract ~~Data~~ Data from Result Set
    while(rs.next())
    {
    int id = rs.getInt("id");
    int age = rs.getInt("age");
    String first = rs.getString("firstname");
    String last = rs.getString("lastname");
    System.out.println("ID:"+id);
                       ("Age:"+age);
                       ("Firstname:"+first);
                       ("Lastname:"+last);
    }
rs.close();
Stmt.close();
Con.close();
```

```
}
catch(Exception e)
{
Sop (e.getMessage());
}}
```

Creating JDBC App to Access Derby Database:

In derby db we use private static string dbURL
= "jdbc.derby://localhost:1527/myDB;