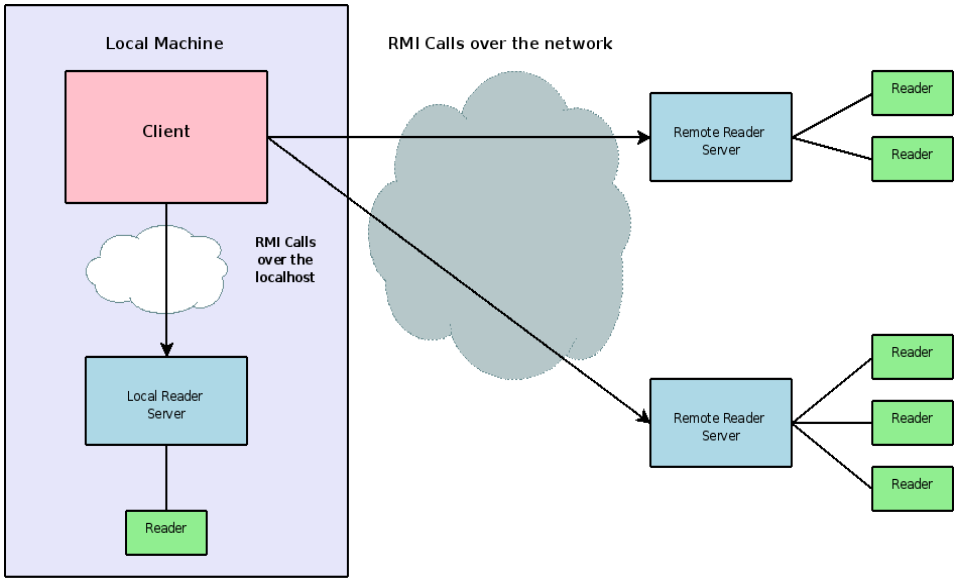
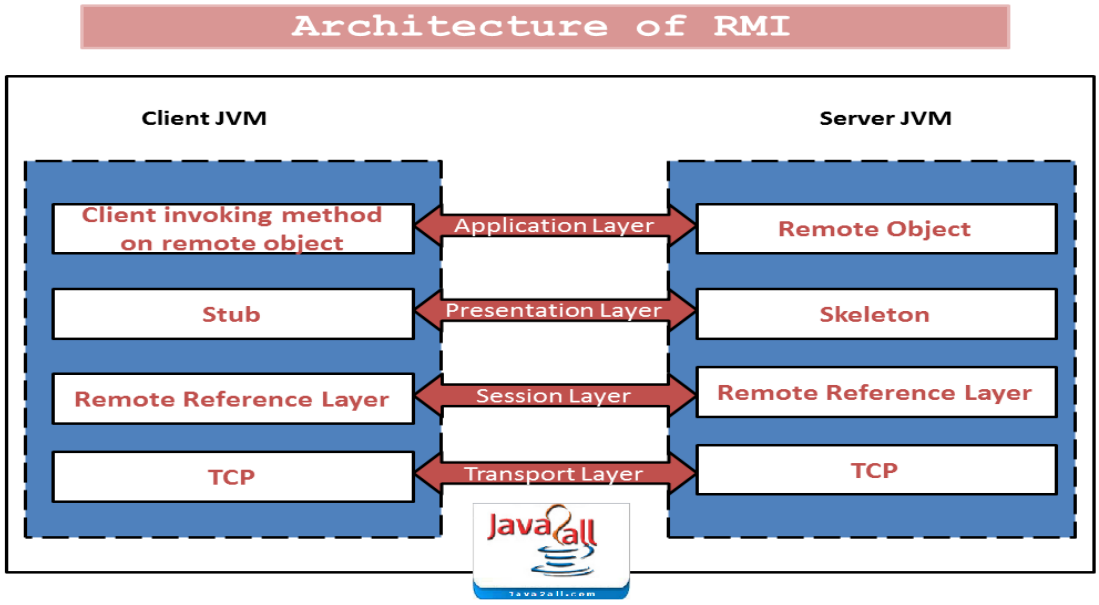
RMI Architecture:

The RMI Architecture (System) has a **FOUR** layer,
(1) Application Layer
(2) Proxy Layer
(3) Remote Reference Layer
(4) Transport Layer

Block Diagram for RMI



RMI Architecture Diagram:



(1) Application Layer:

It's a responsible for the actual logic (implementation) of the client and server applications.

Generally at the server side class contain implementation logic and also apply the reference to the appropriate object as per the requirement of the logic in application.

(2) Proxy Layer:

It's also called the **"Stub/Skeleton layer"**.

A Stub class is a client side proxy handles the remote objects which are getting from the reference.

A Skeleton class is a server side proxy that set the reference to the objects which are communicates with the Stub.

(3) Remote Reference Layer (RRL):

It's a responsible for manage the references made by the client to the remote object on the server so it is available on both JVM (Client and Server).

The Client side RRL receives the request for methods from the Stub that is transferred into byte stream process called serialization (Marshaling) and then these data are send to the Server side RRL.

The Server side RRL doing reverse process and convert the binary data into object. This process called deserialization or unmarshaling and then sent to the Skeleton class.

(4) Transport Layer:

It's also called the "**Connection layer".**

It's a responsible for the managing the existing connection and also setting up new connections.

So it is a work like a link between the RRL on the Client side and the RRL on the Server side.

JDBC Driver Types

JDBC drivers are divided into four types or levels. The **different types of jdbc drivers** are:

Type 1: JDBC-ODBC Bridge driver (Bridge)

Type 2: Native-API/partly Java driver (Native)

Type 3: All Java/Net-protocol driver (Middleware)

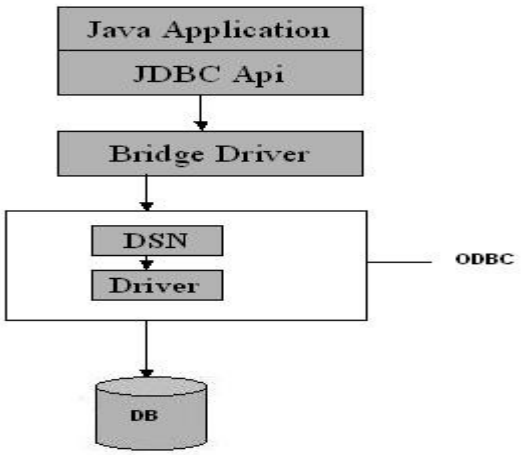
Type 4: All Java/Native-protocol driver (Pure)

4 types of jdbc drivers are elaborated in detail as shown below:

Type 1 JDBC Driver

JDBC-ODBC Bridge driver

The Type 1 driver translates all JDBC calls into ODBC calls and sends them to the ODBC driver. ODBC is a generic API. The JDBC-ODBC Bridge driver is recommended only for experimental use or when no other alternative is available.



Type 1: JDBC-ODBC Bridge

Advantage

The JDBC-ODBC Bridge allows access to almost any database, since the database's ODBC drivers are already available.

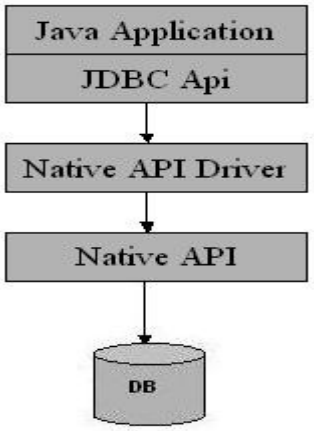
Disadvantages

1. Since the Bridge driver is not written fully in Java, Type 1 drivers are not portable.
2. A performance issue is seen as a JDBC call goes through the bridge to the ODBC driver, then to the database, and this applies even in the reverse process. They are the slowest of all driver types.
3. The client system requires the ODBC Installation to use the driver.
4. Not good for the Web.

Type 2 JDBC Driver

Native-API/partly Java driver

The distinctive characteristics of type 2 jdbc drivers are that Type 2 drivers convert JDBC calls into database-specific calls i.e. this driver is specific to a particular database. Some distinctive characteristic of type 2 jdbc drivers are shown below. Example: Oracle will have oracle native api.



Type 2: Native API/ Partly Java Driver

Advantage

The distinctive characteristic of type 2 jdbc drivers are that they are typically offer better performance than the JDBC-ODBC Bridge as the layers of communication (tiers) are less than that of Type
1 and also it uses Native api which is Database specific.

Disadvantage

1. Native API must be installed in the Client System and hence type 2 drivers cannot be used for the Internet.

2. Like Type 1 drivers, it's not written in Java Language which forms a portability issue.

3. If we change the Database we have to change the native api as it is specific to a database

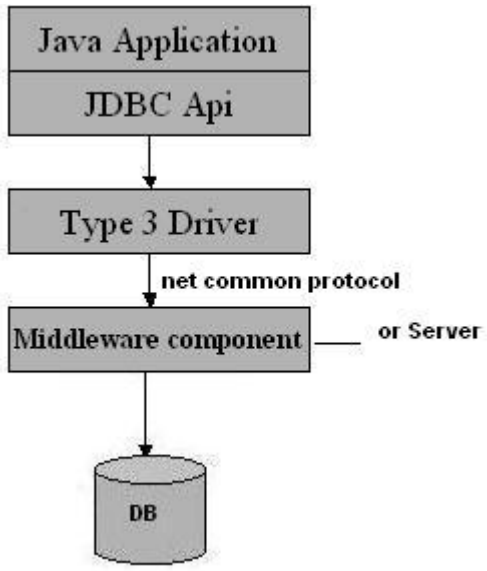
4. Mostly obsolete now

5. Usually not thread safe.

Type 3 JDBC Driver

All Java/Net-protocol driver

Type 3 database requests are passed through the network to the middle-tier server. The middle-tier then translates the request to the database. If the middle-tier server can in turn use Type1, Type 2 or Type 4 drivers.



Type 3: All Java/ Net-Protocol Driver

Advantage

1. This driver is server-based, so there is no need for any vendor database library to be present on client machines.

2. This driver is fully written in Java and hence Portable. It is suitable for the web.

3. There are many opportunities to optimize portability, performance, and scalability.

4. The net protocol can be designed to make the client JDBC driver very small and fast to load.

5. The type 3 driver typically provides support for features such as caching (connections, query results, and so on), load balancing, and advanced
system administration such as logging and auditing.

6. This driver is very flexible allows access to multiple databases using one driver.

7. They are the most efficient amongst all driver types.

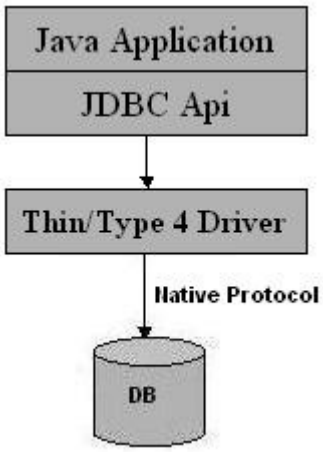
Disadvantage

It requires another server application to install and maintain. Traversing the record set may take longer, since the data comes through the backend server.

Type 4 JDBC Driver

Native-protocol/all-Java driver

The Type 4 uses java networking libraries to communicate directly with the database server.



Type 4: Native-protocol/all-Java driver

Advantage

1. The major benefit of using a type 4 jdbc drivers are that they are completely written in Java to achieve platform independence and eliminate deployment administration issues.

Basic steps to use a database in Java

1. Establish a **connection**
2. Create JDBC **Statements**
3. Execute **SQL** Statements
4. GET **ResultSet**
5. **Close** connections

1. Establish a connection

- **import java.sql.*;**
- **Load the vendor specific driver**
  - Class.forName("oracle.jdbc.driver.OracleDriver");
    - Ques: What do you think this statement does, and how?
    - Ans: Dynamically loads a driver class, for Oracle database
- **Make the connection**
  - Connection con = DriverManager.getConnection("jdbc:oracle:thin:@oracle-prod:1521:OPROD", username, passwd);
    - Ques: What do you think this statement does?
    - Ans: Establishes connection to database by obtaining a *Connection* object

2. Create JDBC statement(s)

- Statement stmt = con.createStatement() ;
- Creates a Statement object for sending SQL statements to the database

3. Executing SQL Statements

- String createLehigh = "Create table Lehigh " + "(SSN Integer not null, Name VARCHAR(32), " + "Marks Integer)";

stmt.**executeUpdate**(createLehigh);

- String insertLehigh = "Insert into Lehigh values “ + “ (123456789,abc,100)";

  stmt.**executeUpdate**(insertLehigh);

4. Get ResultSet

String queryLehigh = "select * from Lehigh";

ResultSet rs = Stmt.**executeQuery**(queryLehigh);

//What does this statement do?

while (rs.next()) {

int ssn = rs.getInt("SSN");

String name = rs.getString("NAME");

int marks = rs.getInt("MARKS");

}

5. **Close connection**

- stmt.close();
- con.close();

RMI Methods:

lookup ()

public static Remote lookup(String name)
throws NotBoundException,
MalformedURLException,
RemoteException

Returns a reference, a stub, for the remote object associated with the specified name.

Parameters:
name - a name in URL format (without the scheme component)

Returns:
a reference for a remote object

Bind method:

public static void bind(String name,
Remote obj)
throws AlreadyBoundException,
MalformedURLException,
RemoteException
Binds the specified name to a remote object.
Parameters:
name - a name in URL format (without the scheme component)
obj - a reference for the remote object (usually a stub)
Throws:
AlreadyBoundException - if name is already bound
MalformedURLException - if the name is not an appropriately formatted URL
RemoteException - if registry could not be contacted
AccessException - if this operation is not permitted (if originating from a non-local host, for example)

Unbind method:

public static void unbind(String name)
throws RemoteException,
NotBoundException,
MalformedURLException
Destroys the binding for the specified name that is associated with a remote object.
Parameters:
name - a name in URL format (without the scheme component)
Throws:

NotBoundException - if name is not currently bound
MalformedURLException - if the name is not an appropriately formatted URL
RemoteException - if registry could not be contacted
AccessException - if this operation is not permitted (if originating from a non-local host, for example)

Rebind method:

public static void rebind(String name, Remote obj)
throws RemoteException,
MalformedURLException

Rebinds the specified name to a new remote object. Any existing binding for the name is replaced.

Parameters:
name - a name in URL format (without the scheme component)
obj - new remote object to associate with the name

Throws:
MalformedURLException - if the name is not an appropriately formatted URL
RemoteException - if registry could not be contacted
AccessException - if this operation is not permitted (if originating from a non-local host, for example)

List method:

public static String[] list(String name)
throws RemoteException, MalformedURLException

Returns an array of the names bound in the registry. The names are URL-formatted (without the scheme component) strings. The array contains a snapshot of the names present in the registry at the time of the call.

Parameters:
name - a registry name in URL format (without the scheme component)

Returns:
an array of names (in the appropriate format) bound in the registry

Throws:
MalformedURLException - if the name is not an appropriately formatted URL
RemoteException - if registry could not be contacted.