

# Middleware and Web Services

## Lecture 4: Application Server Services

**doc. Ing. Tomáš Vitvar, Ph.D.**

tomas@vitvar.com • @TomasVitvar • <http://vitvar.com>



Czech Technical University in Prague

Faculty of Information Technologies • Software and Web Engineering • <http://vitvar.com/courses/mdw>



Modified: Thu Nov 23 2017, 11:20:08  
Humla v0.3

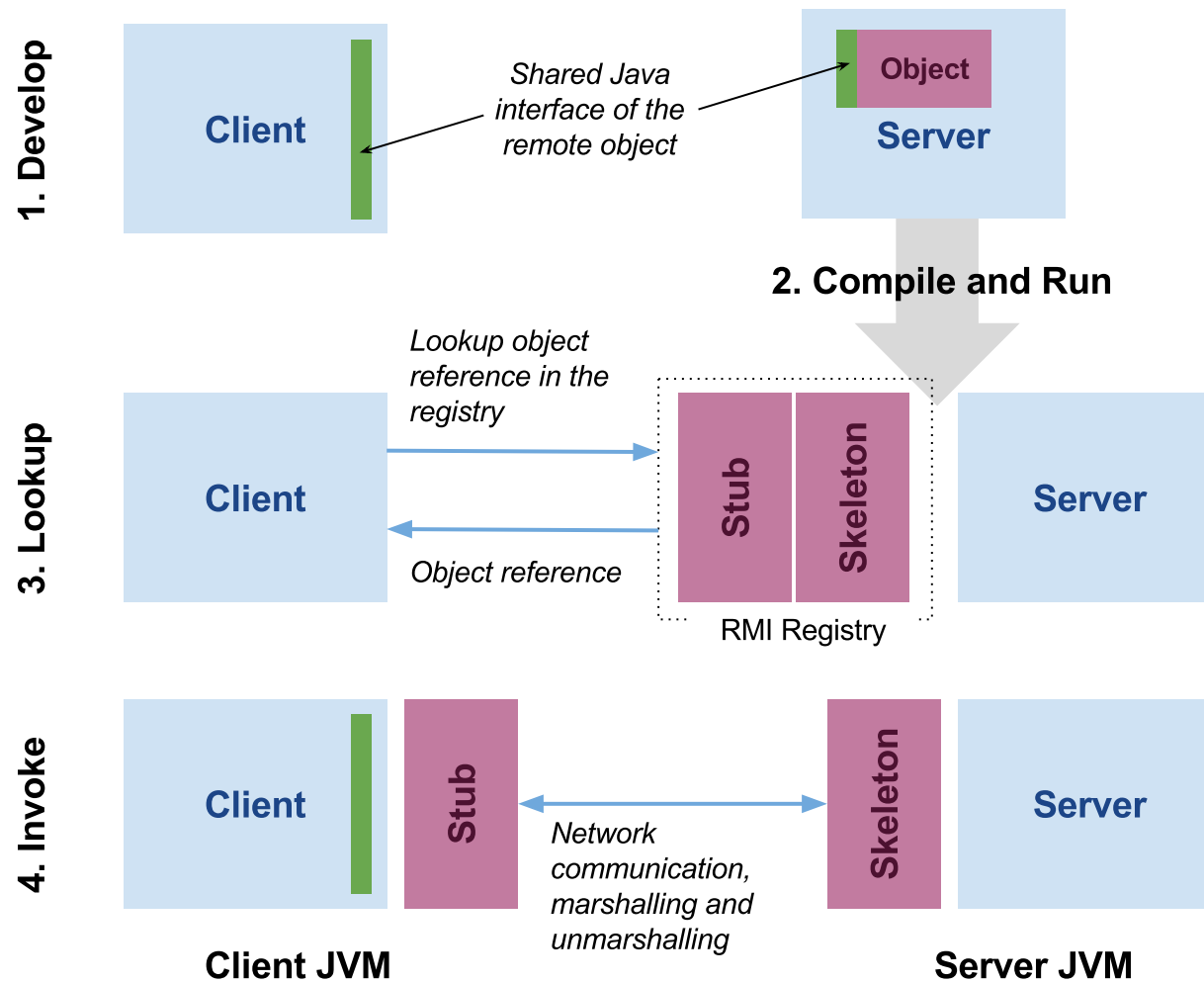
# Overview

- Java Technologies and Services
  - *Remote Method Invocation*
  - *Java Database Connectivity*
  - *Java Naming and Directory Interface*
  - *Application Server and JNDI*
  - *Two-phase Commit*

# Overview

- Communication among Java-based applications
  - *Methods of a Java class can be invoked by other Java class remotely*
  - *Uses Java Remote Method Protocol (JRMP)*
    - *Java-specific application protocol over TCP/IP*
  - *Basis for JEE technologies, such as JMS*
- Terminology
  - **Client** – *a program that invokes a remote method*
  - **Server** – *a program that exports a remote object*
  - **Stub** – *a representation of the client-side object for communication*
  - **Skeleton** – *a representation of the server-side object for communication*
  - **Registry** – *a component that holds a stub*
  - **Marshalling/Unmarshalling** – *a process of transforming memory representation of the object to a form suitable for network transmission and vice-versa*

# Architecture



# RMI Implementation in Java – Interface

- Shared interface

```
1  import java.rmi.Remote;  
2  import java.rmi.RemoteException;  
3  
4  // shared interface between a client and a server to  
5  // invoke methods on the remote object  
6  public interface HelloRMIIInterface extends Remote {  
7      public String calculate(int a, int b) throws RemoteException;  
8  }
```

- RMI Server

```
1  import java.rmi.Naming;  
2  import java.rmi.RemoteException;  
3  import java.rmi.RMISecurityManager;  
4  import java.rmi.server.UnicastRemoteObject;  
5  import java.rmi.registry.LocateRegistry;  
6  
7  public class Server extends UnicastRemoteObject implements HelloRMIIInterface {  
8  
9      // implementation of the interface method  
10     public int calculate(int a, int b) throws RemoteException {  
11         return a+b;  
12     }
```

# RMI Implementation in Java – Server

- RMI Server (cont.)

```
1  // start the server and register the object in the rmi registry
2  public static void main(String args[]) {
3      try {
4          // install a security manager (uses a security policy)
5          if (System.getSecurityManager() == null) {
6              RMISecurityManager sm = new RMISecurityManager();
7              System.setSecurityManager(sm);
8          }
9
10         // create rmi registry
11         LocateRegistry.createRegistry(1099);
12
13         // create remote object
14         Server obj = new Server();
15
16         // Bind the object instance to the name "HelloRMI"
17         // 0.0.0.0 denotes the service will listen on all network interfaces
18         Naming.rebind("//0.0.0.0/HelloRMI", obj);
19
20         System.out.println("RMI server started, " +
21             "listening to incoming requests...");
22     } catch (Exception e) {
23         System.out.println("Error occurred: " + e.getLocalizedMessage());
24     }
25 }
26
```

# RMI Implementation in Java – Client

- RMI Client

```
1  import java.rmi.Naming;
2
3  public class Client {
4
5      public static void main(String args[]) throws Exception {
6          // get a reference to the remote object
7          // assuming the server is running on the localhost
8          HelloRMIInterface o = (HelloRMIInterface)
9              Naming.lookup("//localhost/HelloRMI");
10
11          // call the object method
12          System.out.println(o.calculate(6, 4));
13      }
14  }
```

# Overview

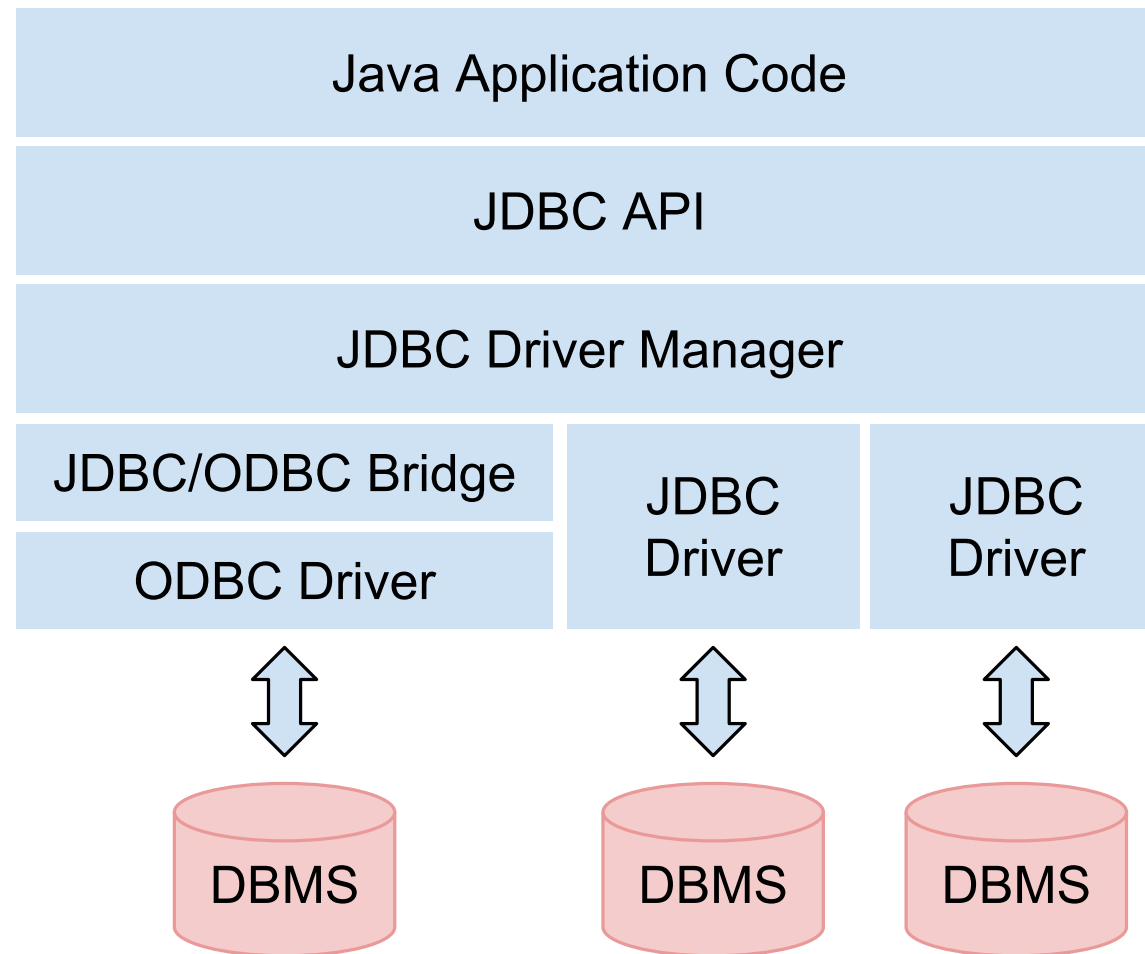
- Java Technologies and Services
  - *Remote Method Invocation*
  - *Java Database Connectivity*
  - *Java Naming and Directory Interface*
  - *Application Server and JNDI*
  - *Two-phase Commit*



# Overview

- Uniform API to access any kind of tabular data
  - *No need to deal with specific APIs of DBMS vendors*
- JDBC components
  - *JDBC API*
    - *defines methods to execute SQL statements and retrieve results*
  - *Driver Manager*
    - *provides drivers that provide access to a specific DBMS*
    - *they implement a specific protocol to access the DBMS*
  - *JDBC-ODBC Bridge*
    - *a software bridge which provides access via ODBC drivers*
    - *ODBC driver is a driver in C for accessing DBMS*

# JDBC Architecture



# JDBC Example Implementation in Java

- JDBC Client

```
1  import java.sql.*;
2
3  public class JDBCClient {
4
5      public static void main(String args[]){
6          // database url
7          String db_url = "jdbc:oracle:thin:@czns20sr:33001:XE";
8
9          // username and password
10         String username = "myUsername";
11         String password = "myPassword";
12
13         try {
14             // Register JDBC driver
15             Class.forName("oracle.jdbc.driver.OracleDriver");
16
17             // Open a connection
18             Connection con = DriverManager.getConnection(
19                 db_url, username, password);
20
21             // Create and execute query statement
22             Statement stmt = con.createStatement();
23             String sql = "SELECT id, first, last, age FROM Employees";
24             ResultSet rs = stmt.executeQuery(sql);
25         }
```

# JDBC Example Implementation in Java

- JDBC Client (cont.)

```
25         // Loop and extract received data
26         while (rs.next()) {
27             int id = rs.getInt("id");
28             int age = rs.getInt("age");
29             String first = rs.getString("first");
30             String last = rs.getString("last");
31         }
32
33         // Release the connections
34         rs.close();
35         stmt.close();
36         conn.close();
37     } catch (SQLException se) {
38
39         //Handle errors for JDBC
40         se.printStackTrace();
41     } catch (Exception e) {
42
43         //Handle errors for Class.forName
44         e.printStackTrace();
45     }
46 }
47 }
48 }
```

# Overview

- Java Technologies and Services
  - *Remote Method Invocation*
  - *Java Database Connectivity*
  - *Java Naming and Directory Interface*
  - *Application Server and JNDI*
  - *Two-phase Commit*

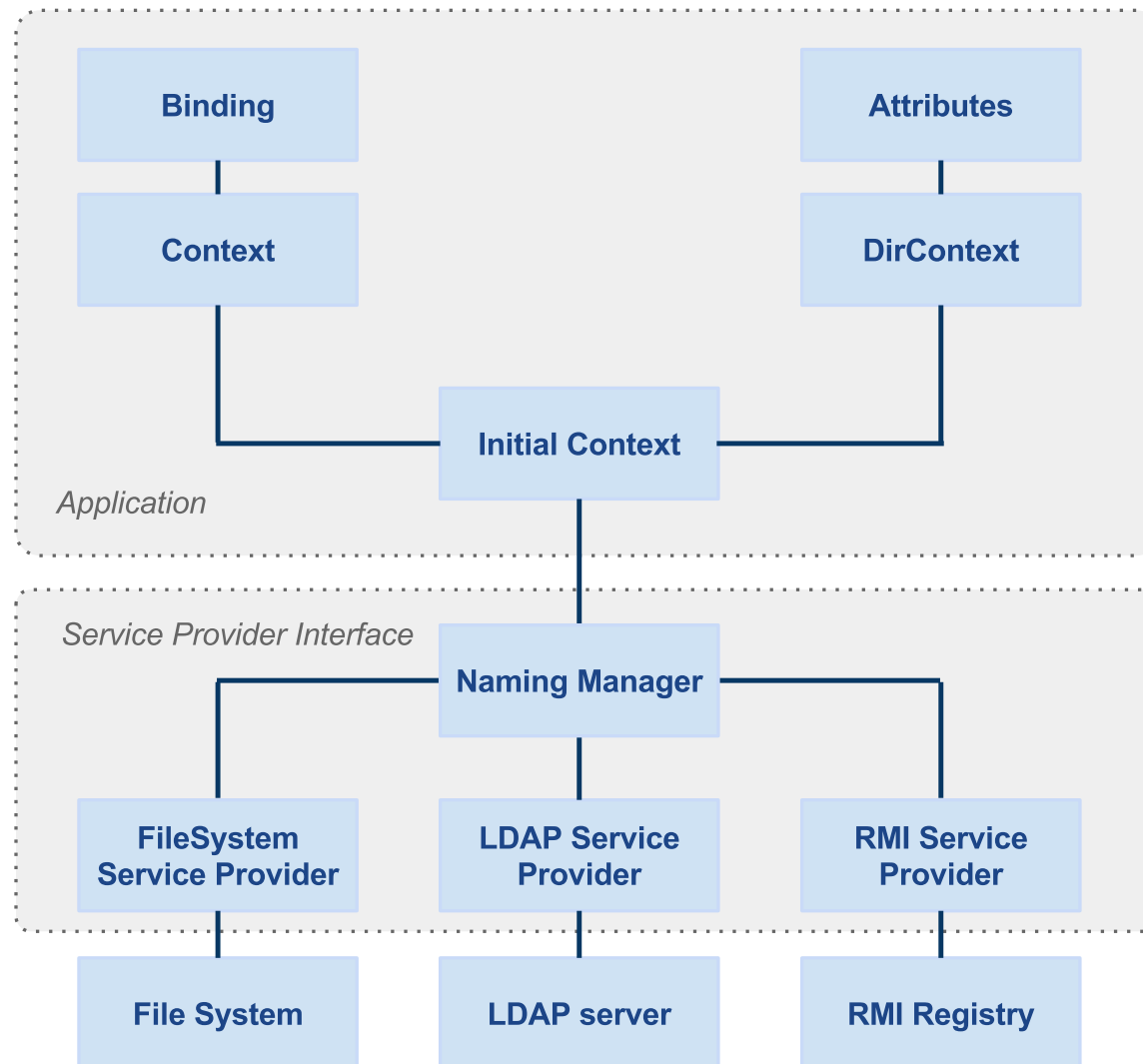
# Overview

- Objectives
  - *Allows to access objects by names in various directory systems and their attributes*
  - *Independent of any specific directory service implementation*
  - *Enables to distribute Java objects across various systems in the environment*
- Terminology
  - *Binding – association between a name and a object*
  - *Context – a set of bindings*
- JNDI Provides:
  - *a mechanism to bind an object to a name.*
  - *a directory lookup interface*
  - *a pluggable service provider interface (SPI) – any directory service implmentation can be plugged in*

# JNDI Packages

- Naming Package
  - *interfaces to access naming services*
  - *Context: looking up, binding/unbinding, renaming, objects*
- Directory Package
  - *allows to retrieve attributes of objects, and to search for objects*
- Event Package
  - *allows for event notification in naming and directory services*
  - *For example, object was added, object changed, etc.*
- Other packages
  - *LDAP – allows to access LDAP services*
  - *Service Provider Interface – allows to develop various naming/directory services*

# JNDI Architecture





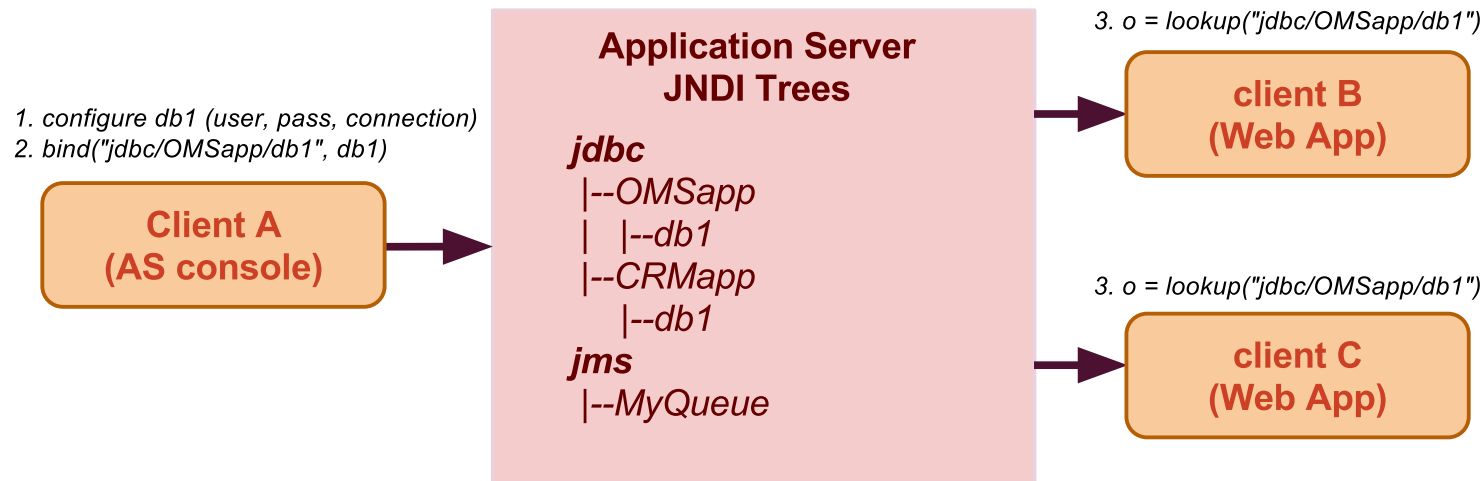
# Application Server and JNDI

- Distribution of objects
  - *Application Server provides central directory for various kinds of objects*
    - *Datasources, JMS queues and topics, etc.*
  - *Clients store objects in the central directory*
    - *Administrator configures objects using Application Server Console or via AS API*
  - *Clients retrieve objects from the central directory*
- Benefits
  - *replication of objects across clients*
  - *central configuration of objects' parameters*
  - *scalability – allowing/disabling connections as required*

# Overview

- Java Technologies and Services
  - *Remote Method Invocation*
  - *Java Database Connectivity*
  - *Java Naming and Directory Interface*
  - *Application Server and JNDI*
  - *Two-phase Commit*

# Application Server and JNDI



- Example Scenario

- Client A creates a datasource, configures it and registers it in the JNDI tree
  - Client A is a Admin server console app; this task is performed by the administrator
- Client B and C lookup the object under specific JNDI name and retrieves the object from the tree
  - They get the object from the tree and use it to connect to the DB
  - They do not need to know any DB specific details, the object is pre-configured from the server

# Example Datasource in Weblogic

Settings for order-db

Configuration

Targets

Monitoring

Control

Security

Notes

General

Connection Pool

Oracle

ONS

Transaction

Diagnostics

Identity Options

Click the **Lock & Edit** button in the Change Center to modify the settings on this page.


Save

Applications get a database connection from a data source by looking up the data source on the Java Naming and provides the connection to the application from its pool of database connections.


This page enables you to define general configuration options for this JDBC data source.


Name:

order-db


 **JNDI Name:**

jdbc/order-db

☐  **Row Prefetch Enabled**

 **Row Prefetch Size:**

48

 **Stream Chunk Size:**

256

Save

# Targets

- Object
  - *A service provided by the application server, e.g. datasources, JMS queue, SAF*
- Types of services
  - *Pinned services*
    - *Objects targeted to a single server only*
  - *Cluster services*
    - *Objects targeted to all servers in the cluster*

# Example Target Configuration

**Settings for order-db**

Configuration **Targets** Monitoring Control Security Notes

Click the **Lock & Edit** button in the Change Center to modify the settings on this page.

Save

This page allows you to select the servers or clusters on which you would like to deploy this JDBC data source.

**Servers**  
☐ SOA\_AdminServer

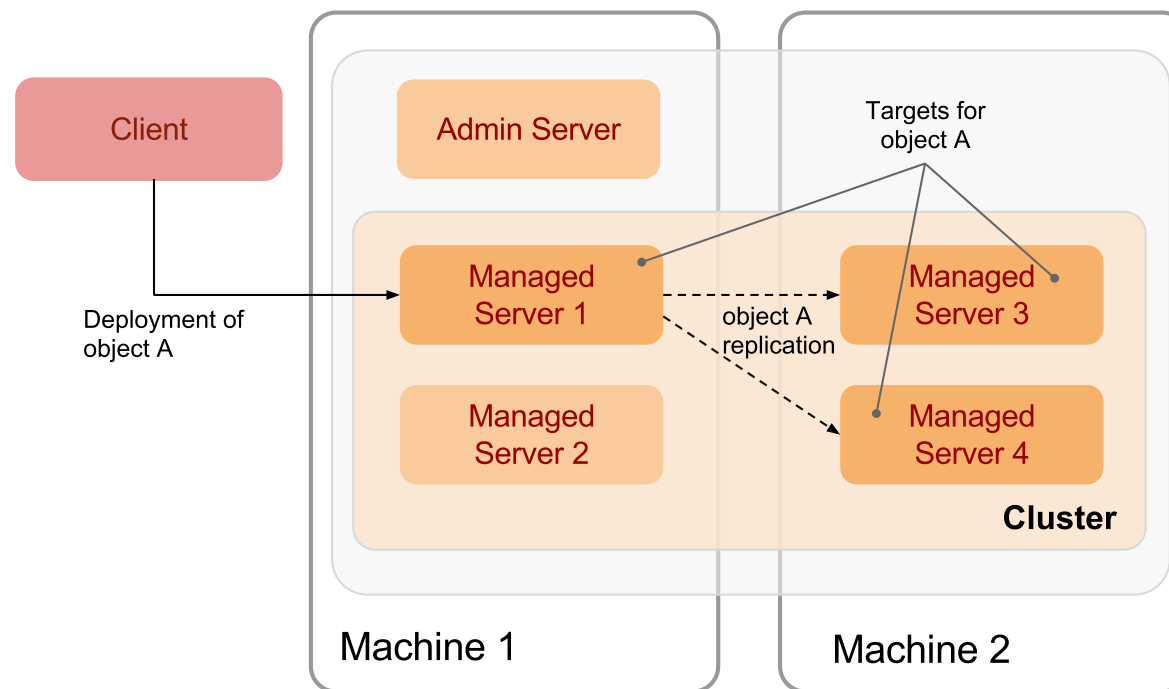
**Clusters**  
☒ SOA\_Cluster  
    ☒ All servers in the cluster  
    ☐ Part of the cluster  
        ☐ WLS\_SOA1  
        ☐ WLS\_SOA2  
        ☐ WLS\_SOA3  
  
☐ WSM\_PM-Cluster  
    ☐ All servers in the cluster  
    ☐ Part of the cluster  
        ☐ WLS\_WSM1

Save

Click the **Lock & Edit** button in the Change Center to modify the settings on this page.

# Deployment to Cluster

- Deployment of an object
  - Client deploys to one managed server in the cluster
  - Object gets replicated to its targets
    - Targets can be configured for the object, usually all servers but can be selected servers



# Cluster-wide JNDI Tree

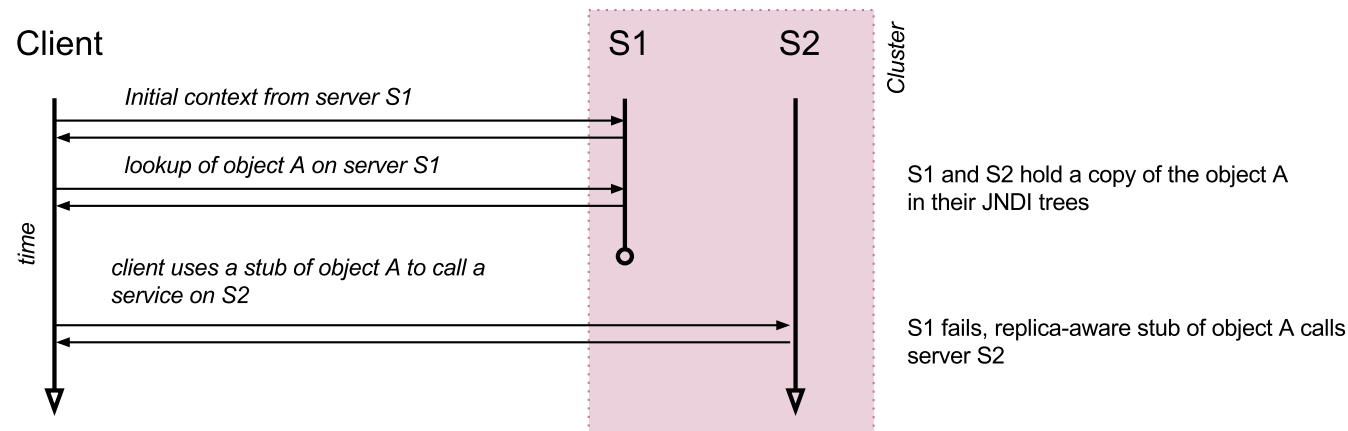
- Cluster
  - *Every managed server has its own JNDI tree*
  - *Servers in a cluster sync up their JNDI trees as per the target configuration*
    - *A stub of the object appears in every managed server's JNDI tree*
    - *They use **JNDI replication service** (see Lecture 6)*
- When a client retrieves an object from the tree
  1. *Client connects to the cluster using the cluster address*
  2. *Client creates an initial context (represents a naming service)*
  3. *Client uses the initial context to lookup objects*
  4. *Client uses the stub of the object to call the service*



# Object Failover

- Failover
  - *Failover = ability to locate an object on another server that holds a copy of the object without impact on the performance and configuration*

## Replica-aware stub of object A, failover in cluster



- *A client gets a stub of the object by calling **lookup** on the context*
- *A client uses the stub of the object to access the object on the server*
- *When a server fails, replicate-aware stub calls the next server that holds the object copy*

# Example JNDI Tree on Weblogic Server

## ORACLE WebLogic Server® Administration Console

### JNDI Tree Structure

#### WLS\_SOA1

- AGMetadataService
- AGQueryService
- ⊕ AuditServiceBean#oracle
- BPelaActivityManagerBean
- BPelaAuditTrailBean
- BPelaInstanceManagerBean
- BPelaProcessManagerBean
- BPelaServerManagerBean
- BPelaTestInstanceManager
- ⊕ BPMAnalytics#oracle
- ⊕ BPMJMSServer\_auto\_1@jms
- ⊕ BPMJMSServer\_auto\_2@jms
- ⊕ BPMJMSServer\_auto\_3@jms
- BPMNActivityManagerBean
- BPMNInstanceManagerBean
- BPMNProcessManagerBean
- CompositeMetadataServiceBean
- DiagnosticService
- ⊕ eis
- ExalogicOptimizedFileAdapter
- FileAdapter

### Settings for jdbc.order-db

#### Overview

#### Security

This page displays details about this bound object.

**Binding Name:** jdbc.order-db

**Class:** weblogic.jdbc.common.internal.RmiDataSource\_1036\_WLStub

**Hash Code:** 399376309

**toString Results:** ClusterableRemoteRef(-2550923414591209501S:czfmwapp03-vf:  
[8001,8001,-1,-1,-1,-1]:soa\_domain:WLS\_SOA1 [-2550923414591209501S:czfmwapp03-vf:  
[8001,8001,-1,-1,-1,-1]:soa\_domain:WLS\_SOA1/290, 3468331851939647561S:czfmwapp04-vf:  
[8001,8001,-1,-1,-1,-1]:soa\_domain:WLS\_SOA2/290, -2757926207220082473S:czfmwapp05-vf:  
[8001,8001,-1,-1,-1,-1]:soa\_domain:WLS\_SOA3/290])/290

# JNDI Implementation in Java

- Lookup for bound object

```
1  import javax.naming.InitialContext;
2  import java.util.*;
3  import javax.sql.*;
4
5  ...
6
7  Properties p = new Properties();
8
9  // configure the service provider url: "t3://localhost:7001"
10 p.put(Context.PROVIDER_URL,
11       "t3://czfmwapp03-vf:8001,czfmwapp04-vf:8001,czfmwapp05-vf:8001");
12
13 // configure the initial context factory.
14 // we use WebLogic context factory
15 p.put(Context.INITIAL_CONTEXT_FACTORY,
16       "weblogic.jndi.WLInitialContextFactory");
17 InitialContext ctx = new InitialContext(p);
18
19 dataSource =
20     (DataSource) ctx.lookup("jdbc/order-db");
21
22 // invoke the object method
23 Connection c = dataSource.getConnection();
24
```

# Overview

- Java Technologies and Services
  - *Remote Method Invocation*
  - *Java Database Connectivity*
  - *Java Naming and Directory Interface*
  - *Application Server and JNDI*
  - *Two-phase Commit*

# Overview

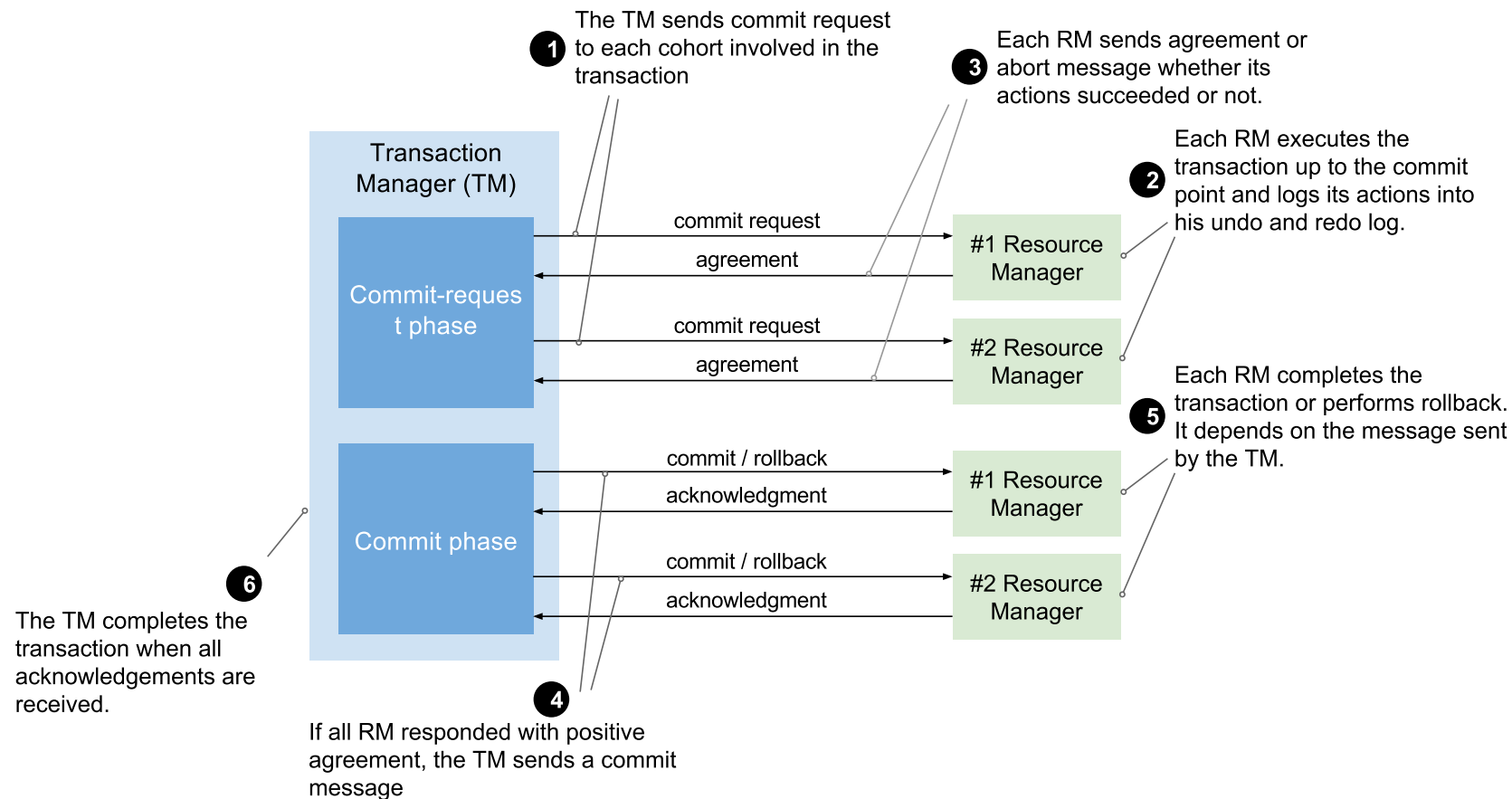
- Coordination of a distributed transaction
  - *All transaction operations are completed across multiple resources; or none is completed*
  - *Able to deal with many types of failures (process, network, communication)*
- Terminology
  - **Transaction Manager** – *manages transactions, coordinates decisions for commit or rollback, and coordinates failure recovery*
  - **Resource Manager** – *manages an access to a resource that participates in the transaction, e.g. DBMS, JMS*
  - **Agreement** – *an agreement message send by the Resource Manager, whether the operation was processed successfully*
  - **Acknowledgment** – *a message about a status of the operation execution*
  - **Rollback** – *operation that returns the Resource Manager state to its pre-transaction state.*

# X/Open – eXtended Architecture (XA)

- Standard for executing distributed transactions
  - *Specifies how the coordinator will roll up the transaction against involved different systems.*
  - *Based on the Two-phase Commit protocol.*
  - *Defines interface between the coordinator and each system node.*
  - *Single transaction access to multiple resources (e.g. message queues, databases, etc.)*
  - *Defined in the eXtended Architecture Specification [↗](#)*
- Wide technological support
  - *Java Transaction API (JTA) [↗](#) – distributed transactions in a Java environment.*
  - *Supported in the Oracle Service Bus through a JMS queue.*
  - *MySQL Relational Database Management System (since v5.0)*

# Two-phase Commit

- Two-phase commit scenario



# XA Example Implementation in Java

- Distributed Transaction

```
1  import java.sql.*;
2  import javax.sql.*;
3  import javax.naming.*;
4  import java.util.*;
5
6  public class Server {
7
8      public static void main(String args[]) {
9
10         // Initialize context
11         Hashtable parms = new Hashtable();
12         parms.put(Context.INITIAL_CONTEXT_FACTORY,
13             "weblogic.jndi.WLInitialContextFactory");
14         parms.put(Context.PROVIDER_URL, "t3://localhost:7001");
15         InitialContext ctx = new InitialContext(parms);
16
17         // Perform a naming service lookup to get UserTransaction object
18         javax.transaction.UserTransaction usertx;
19         usertx = (UserTransaction) ctx.lookup("java:comp/UserTransaction");
20
21         try {
22             //Start a new user transaction.
23             usertx.begin();
```



# XA Example Implementation in Java

- Distributed Transaction (cont.)

```
24 // Establish a connection with the first database
25 javax.sql.DataSource data1;
26 data1=(javax.sql.DataSource)ctx.lookup("java:comp/env/jdbc/DataBase1");
27 java.sql.Connection conn1 = data1.getConnection();
28 java.sql.Statement stat1 = conn1.createStatement();
29 // Establish a connection with the second database
30 javax.sql.DataSource data2;
31 data2=(javax.sql.DataSource)ctx.lookup("java:comp/env/jdbc/DataBase2");
32 java.sql.Connection conn2 = data2.getConnection();
33 java.sql.Statement stat2 = conn2.createStatement();
34
35 // Execute update query to both databases
36 stat1.executeUpdate(...);
37 stat2.executeUpdate(...);
38
39 // Commit the transaction
40 // Apply the changes to the participating databases
41 usertx.commit();
42
43 //Release all connections and statements.
44 stat1.close();
45 stat2.close();
46 conn1.close();
47 conn2.close();
```

# XA Example Implementation in Java

- Distributed Transaction (cont.)

```
48      // Catch any type of exception
49      catch (java.lang.Exception e) {
50          try {
51              e.printStackTrace();
52
53              // Rollback the transaction
54              usertx.rollback();
55              System.out.println("The transaction is rolled back.");
56          } catch (java.lang.Exception ex) {
57              e.printStackTrace();
58              System.out.println("Exception is caught. Check stack trace.");
59          }
60      }
61  }
```