Lecture 6

S4 - Core Distributed Middleware Programming in JEE

presentation

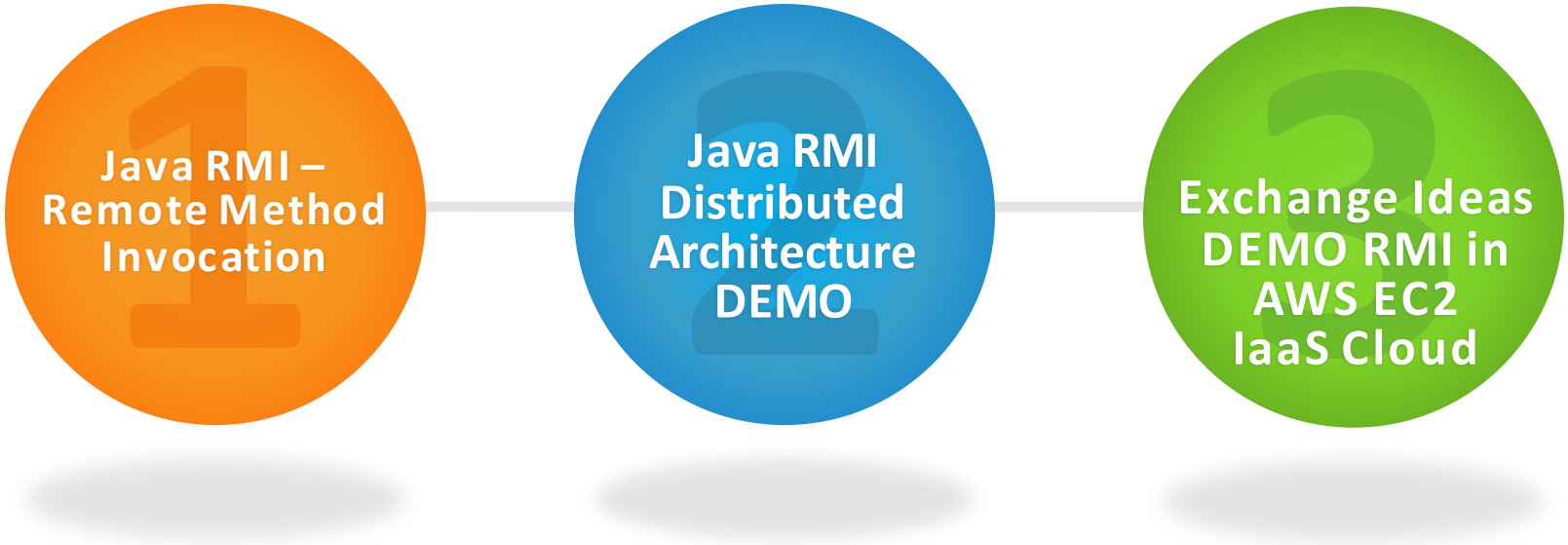
DAD – Distributed Applications Development

Cristian Toma

D.I.C.E/D.E.I.C – Department of Economic Informatics & Cybernetics
www.dice.ase.ro

Cristian Toma – Business Card

Agenda for Lecture 6



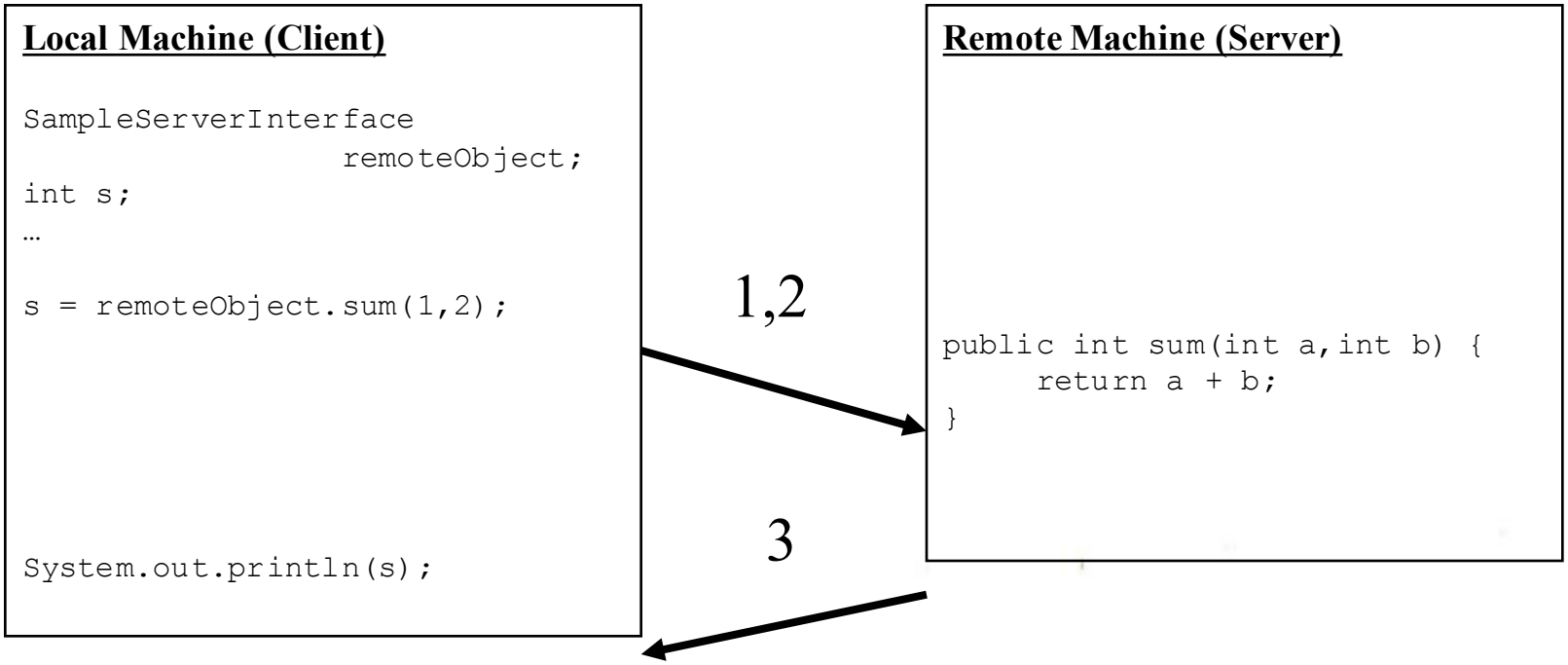
DAD Section 4 - JRMI Samples, JRMP network analysis, JRMI Stubs/Skeleton

Java Remote Method Invocation

1. Java RMI Overview Technology

Java RMI – Remote Method Invocation

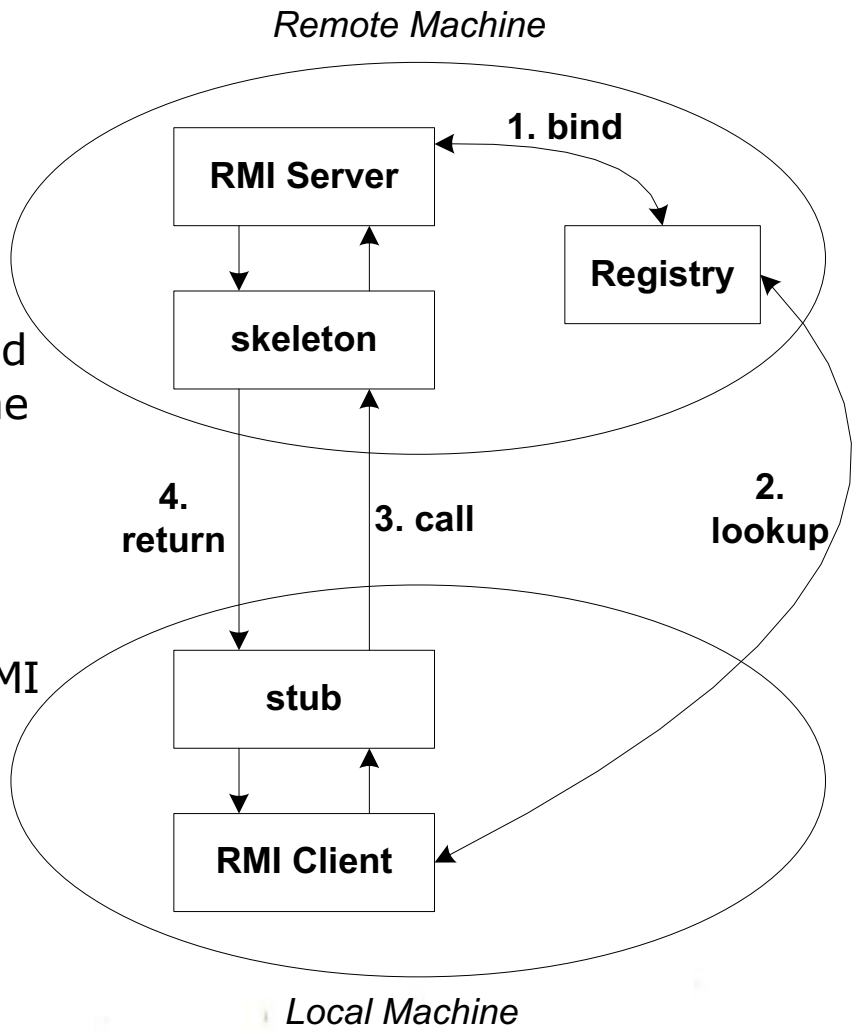
- RMI Overview
- Java RMI allows the programmers to invoke/call procedures/methods from a class inside in a remote virtual machine exactly as it is in the local virtual machine.



1. Java RMI Architecture

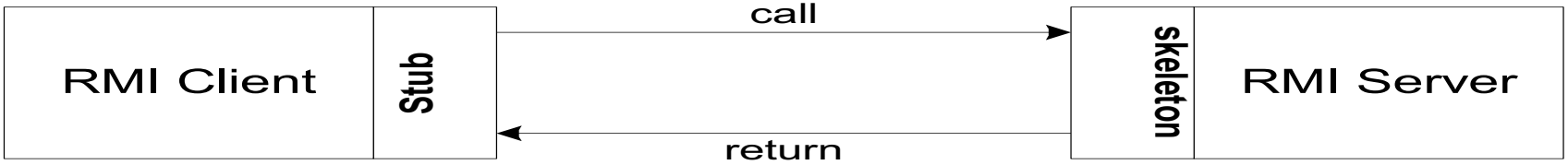
RMI Architecture

- RMI Server must register its name and address in the RMI Registry program – **bind**
- RMI Client is looking for the address and the name of the RMI server object in the RMI Registry program – **lookup**
- RMI Stub serializes and transmits the parameters of the call in sequence – ***"marshalling"*** to the RMI Skeleton. RMI Skeleton de-serializes and extracts the parameters from the received call – ***"unmarshalling".*** In the end, the RMI Skeleton calls the method inside the server object and send back the response to the RMI Stub through "marshalling" the return's parameter.



1. Java RMI Technology

JRMI Skeleton/Stub & Stub Approach



- **The client invokes a remote method after obtaining the reference to the server object through JRMI registry program from the server. The client delegates the involved method invocation sockets and protocol to the JRMI Stub class/instance-object.**

- **JRMI Stub is responsible for calling the method, using the parameter marshaling technique in order to transfer them to the JRMI Skeleton class/instance-object. The JRMI Skeleton class/instance-object from the server side sends the returned response to the JRMI Stub, so the stub must un-marshaling the result and must pass it to the client object.**

- **Technically, the JRMI Stub opens socket to the server, "marshaling" the serializable objects parameters to the server and receive the result from the JRMI Skeleton.**

- **JRMI Skeleton has a method that executes the remote calls from the stubs and it use "un-marshaling" technique in order to extract the parameters of the call and in order to instruct the JRMI server object to run the invoked method with the received parameters**

1. Java RMI Technology

Developing JRMI System

*in readme.txt:

DEVELOPMENT of the RMI SERVER:

1. Defining the remote interface
2. Developing the Java class for instantiation of the JRMI server object – implementing the interface from the step 1.
3. Developing the Java main server program
4. Compiling the Java server classes source code and generating the JRMI Skeleton & JRMI Stub classes using *rmic* utility program

DEVELOPMENT of the RMI CLIENT:

5. Developing the Java client program
6. Copying the Java compiled byte-code files for JRMI Stub and for the remote interface – from the server side to the client side
7. Compiling the client Java source code file together with the files from step 6.

RUNNING RMI SERVER:

8. Start the JRMI registry program.
9. Start the server program.

RUNNING RMI CLIENT:

10. Start the client program

JRMI Development Steps

Step 1: Defining the remote interface

- Remote interface between the client and server objects.

```
/* SampleServerInterface.java */
import java.rmi.*;

public interface SampleServerInterface extends Remote
{
  public int sum(int a,int b) throws RemoteException;
}
```

JRMI Development Steps

Step 2: Development of the remote object implementing the remote interface

- JRMI server object is "unicast remote server" => inheriting the class `java.rmi.server.UnicastRemoteObject`.
- JRMI server object implements the interface from the step 1.

```
/* SampleServerImpl.java */
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
public class SampleServerImpl extends UnicastRemoteObject
                           implements SampleServerInterface {
  SampleServerImpl() throws RemoteException
  { super(); }
```

JRMI Development Steps

Step 2: Development of the remote object implementing the remote interface

```
/* SampleServerImpl.java */
  public int sum(int a,int b) throws RemoteException
  {
     return a + b;
  }
}
```

JRMI Development Steps

Step 3: Development of the main server program

The main JRMI server program, activate the RMISecurityManager in order to protect its own resources in the network and to expose only the items specified in the security policies from the *java.policy* file.

The main server program creates the JRMI server object from the class created in step 2 and implements the interface from the step 1.

The JRMI server must register the object in JRMI registry utility program – **bind()** or **rebind()**.

JRMI Development Steps

Step 3: Development of the main server program

```
/* SampleServerProgMain.java */
  public static void main(String args[])
  {
      try
      {
        System.setSecurityManager(new RMISecurityManager());
        //set the security manager

        //create a local instance of the JRMI server object
        SampleServerImpl Server = new SampleServerImpl();

    //put the local instance in the registry
    Naming.rebind("rmi://localhost:1099/SAMPLE-SERVER" , Server);
        ...
```

JRMI Development Steps

Step 4: Development of the client program

- The JRMI client program activates the RMISecurityManager in order to expose to the JRMI server only the parts specified in the ***java.policy*** file.

- The client program should obtain the reference to the remote object in order to invoke a remote method from the server object. The JRMI clients receive the reference to the remote server object after interrogation of the JRMI registry application – using the **lookup()** method from ***java.rmi.Naming***
- The JRMI server object name is like an URL:
  rmi://server_registry_host:port/server_rmi_name
  rmi://127.0.0.1:1099/SAMPLE-SERVER
- The default port used by the JRMI registry application is 1099.
- The name specified in URL, "server_rmi_name", must be the same as the one used by the JRMI server when registered – *bind()* into JRMI registry application. For instance, here, the name is "SAMPLE-SERVER"

- The call of the remote object is, from the syntax point of view, the same as calling a local method but using the a client object with the remote interface as data type (`SampleServerInterface remoteObject`).

JRMI Development Steps

Step 5: The client program development

```
import java.rmi.*;
import java.rmi.server.*;
public class SampleClient
{
   public static void main(String[]  args)
   {
      // set the security manager for the client
      System.setSecurityManager(new RMISecurityManager());
      //get the remote object from the registry
      try
        {
          System.out.println("Security Manager loaded");
          String url = "rmi://localhost:1099/SAMPLE-SERVER";
 SampleServerInterface remoteObject = (SampleServer)Naming.lookup(url);
          System.out.println("Got remote object");
 System.out.println(" 1 + 2 = " + remoteObject.sum(1,2) );
        }
        catch (RemoteException exc) {
...
```

Java RMI Security Policy File

- In Java, an application actions take into account the privileges required by Java Virtual Machine - JVM - java.exe to the OS-Operating System. The JVM is instructed by a Java Policy file. Usually is in $JAVA_HOME/jre/lib/security folder or can be passed as parameter to the JVM through *-Djava.security.policy=policy.all …* option:

```
grant {
  permission java.security.AllPermission;
};
```

- A modified sample for Java Policy file permission:

```
grant {
 permission java.io.filePermission "d:/tmp", "read", "write";
  permission java.net.SocketPermission
  "somehost.somedomain.com:999","connect";
  permission java.net.SocketPermission "*:1024-
  65535","connect,request";
  permission java.net.SocketPermission "*:80","connect";
};
```

Java RMI Security Policy File

1. It permits the Java class to read and write files from the "d:/tmp" folder and subfolders.

2. It permits the Java class to establish network connections with "somehost.somedomain.com" in the 999 port.

3. It permits the Java class to accept the network connections from any computer as long as the requests are coming for the network ports greater than 1024

4. It permits the Java class to establish the network connections to any computer from the network as long as the requests are for port 80 from the server side, in common way for HTTP protocol.

Necessary Items for running Java RMI

1. The firewall should be configured both on the server and on the client side

2. The security policy files *java.policy* should be configured both on the server and on the client side (e.g. in command line or IDE/Eclipse or IntelliJ ***-Djava.security.policy=./policy.all***)

3. The programs run taking into account the security policies described in the *java.policy* file and JVM heap resize – option -Xms1000000000

4. The RMI name service (**rmiregistry**) should be started into the server root directory.

Section Conclusion

Fact: **DAD core is based on Java RMI**

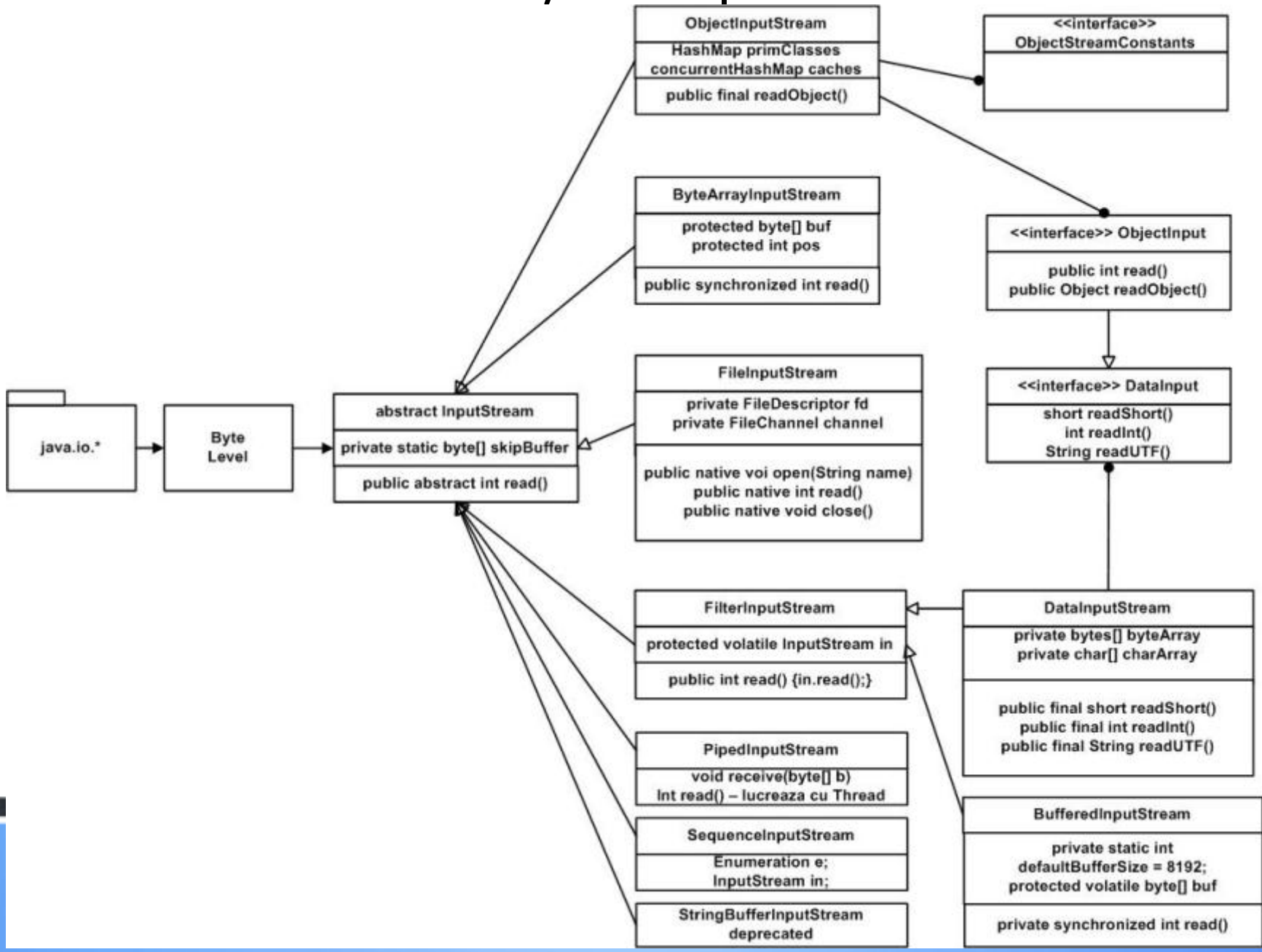
In few **samples** it is simple to remember: Java RMI Architecture with JRMP protocol analysis in real time plus the core actions for distributed computing and systems:

Picture processing within a RMI Cluster.

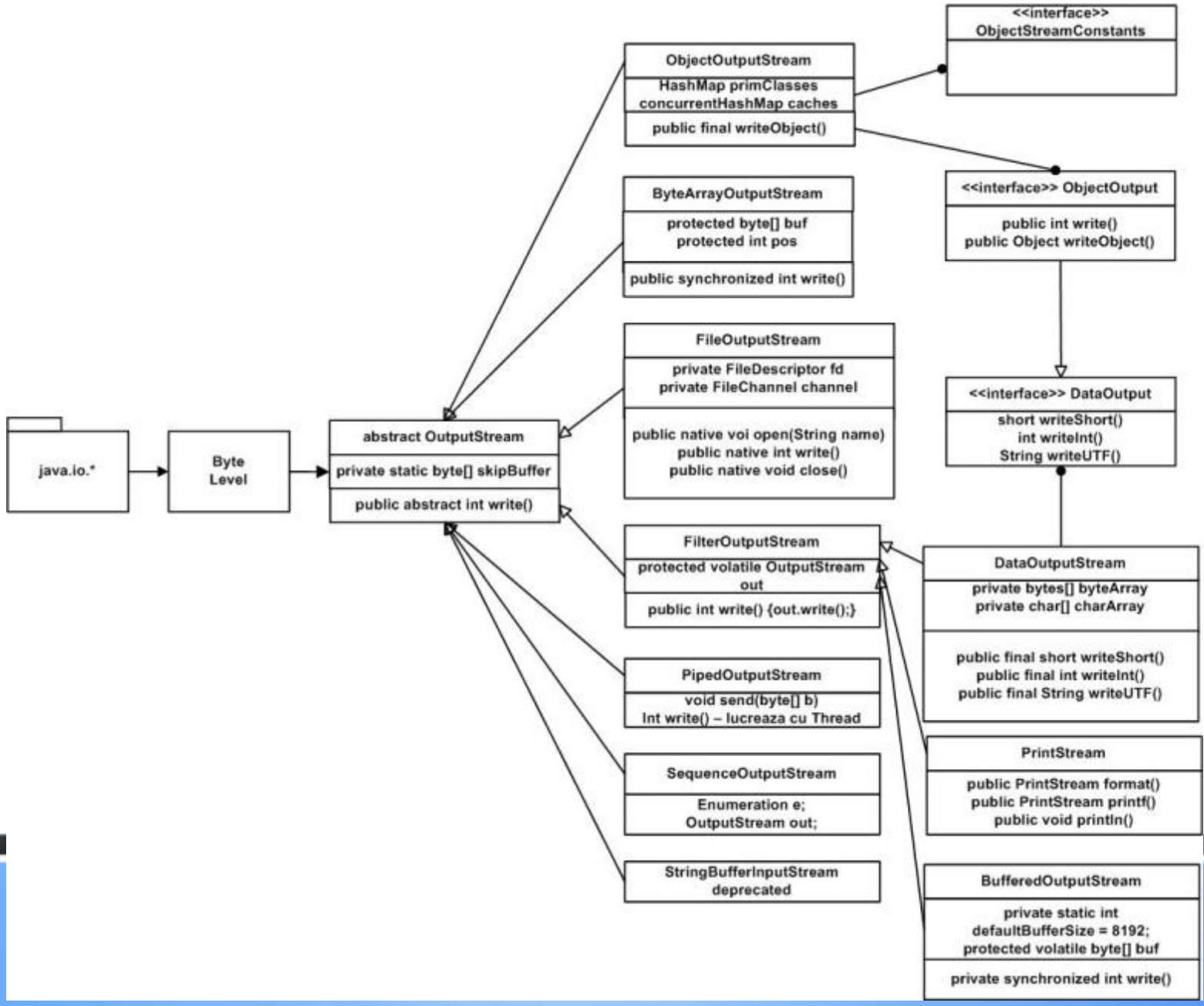
Recap of I/O & Core Middleware Technologies for Distributed Computing / Distributed App Development

Java RMI Remote Method Invocation DEMO

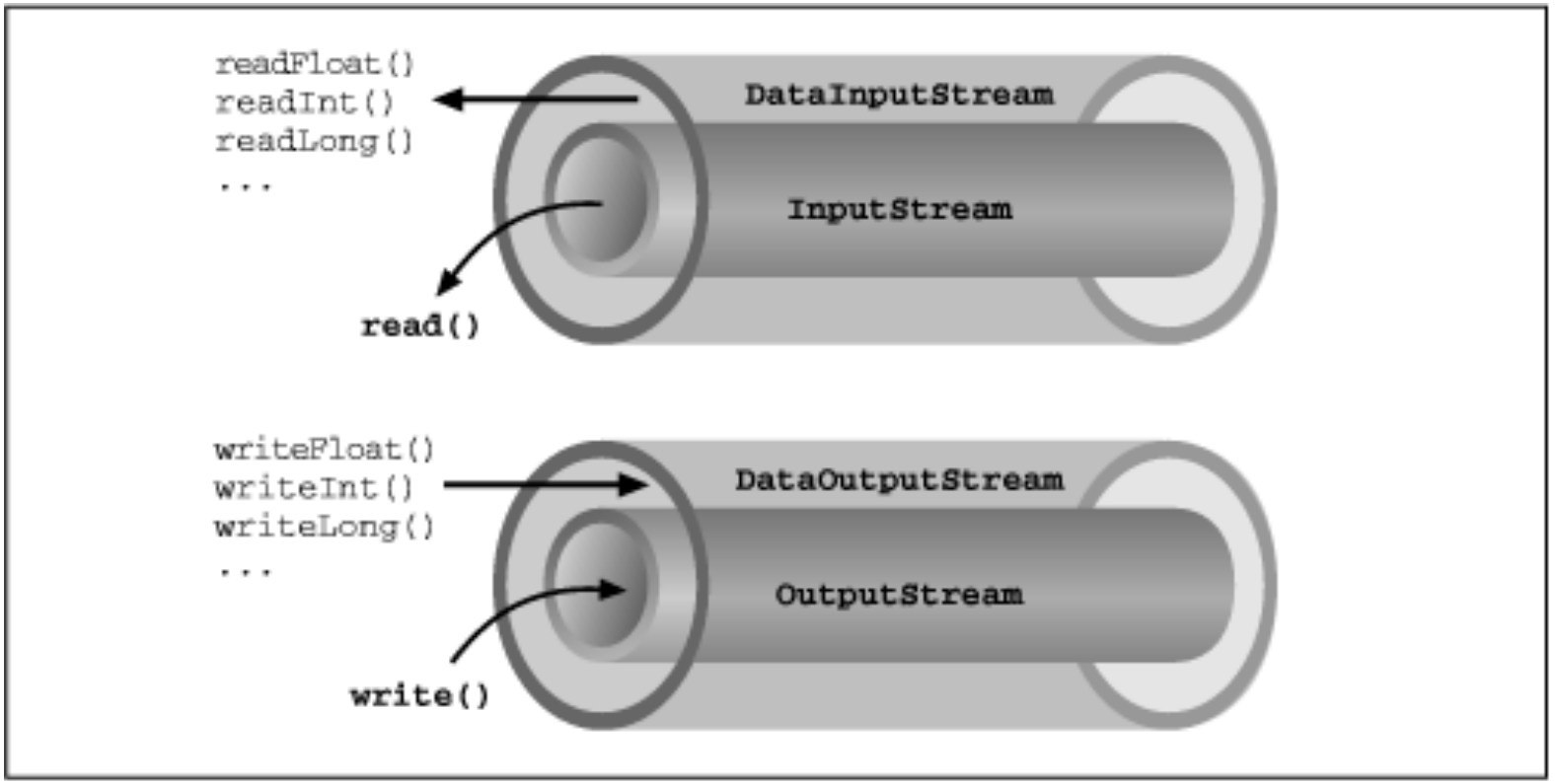
2.1 Java I/O – Input Stream



2.1 Java I/O – Output Stream

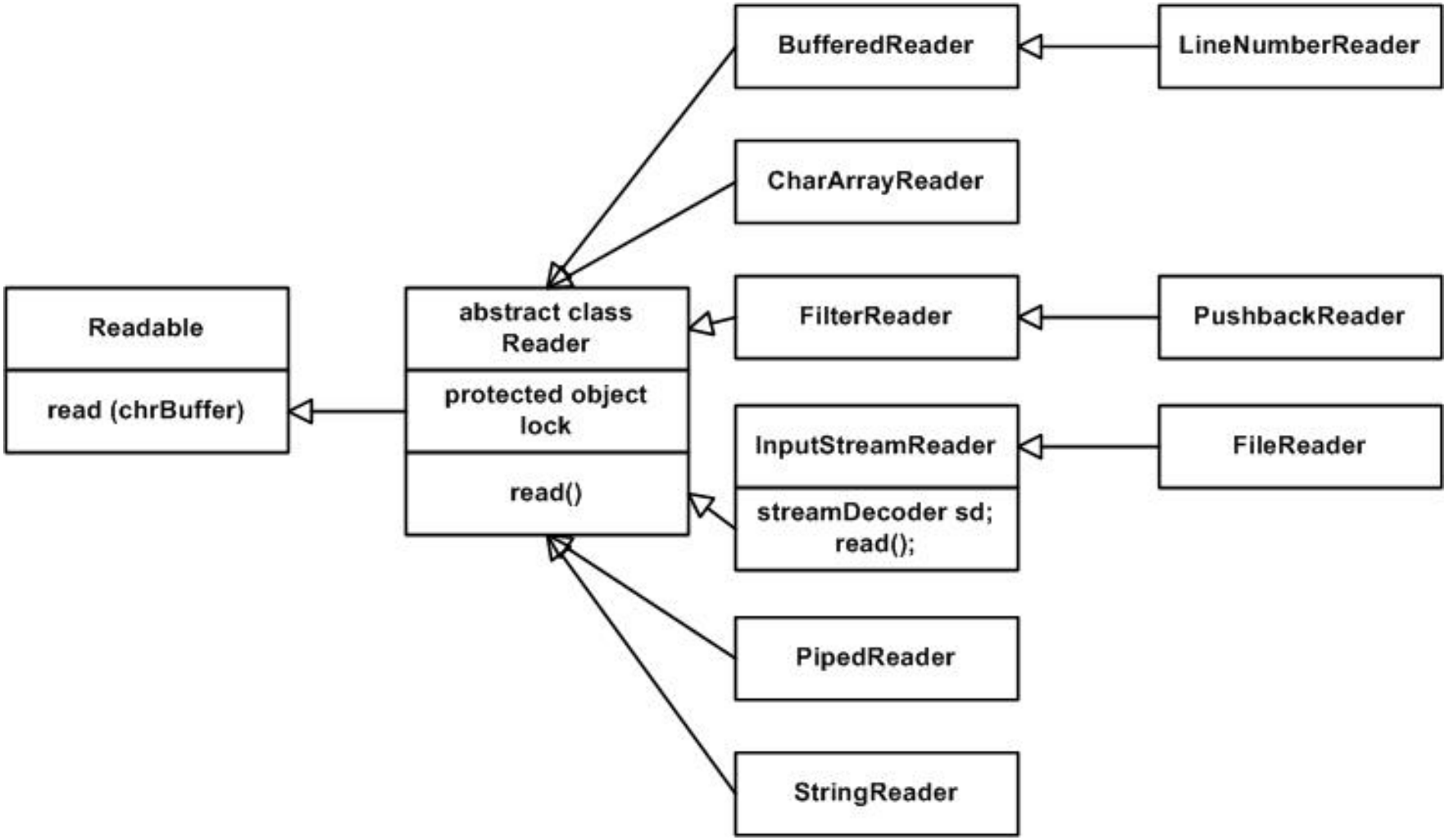


2.1 Java I/O – Streams Encapsulation

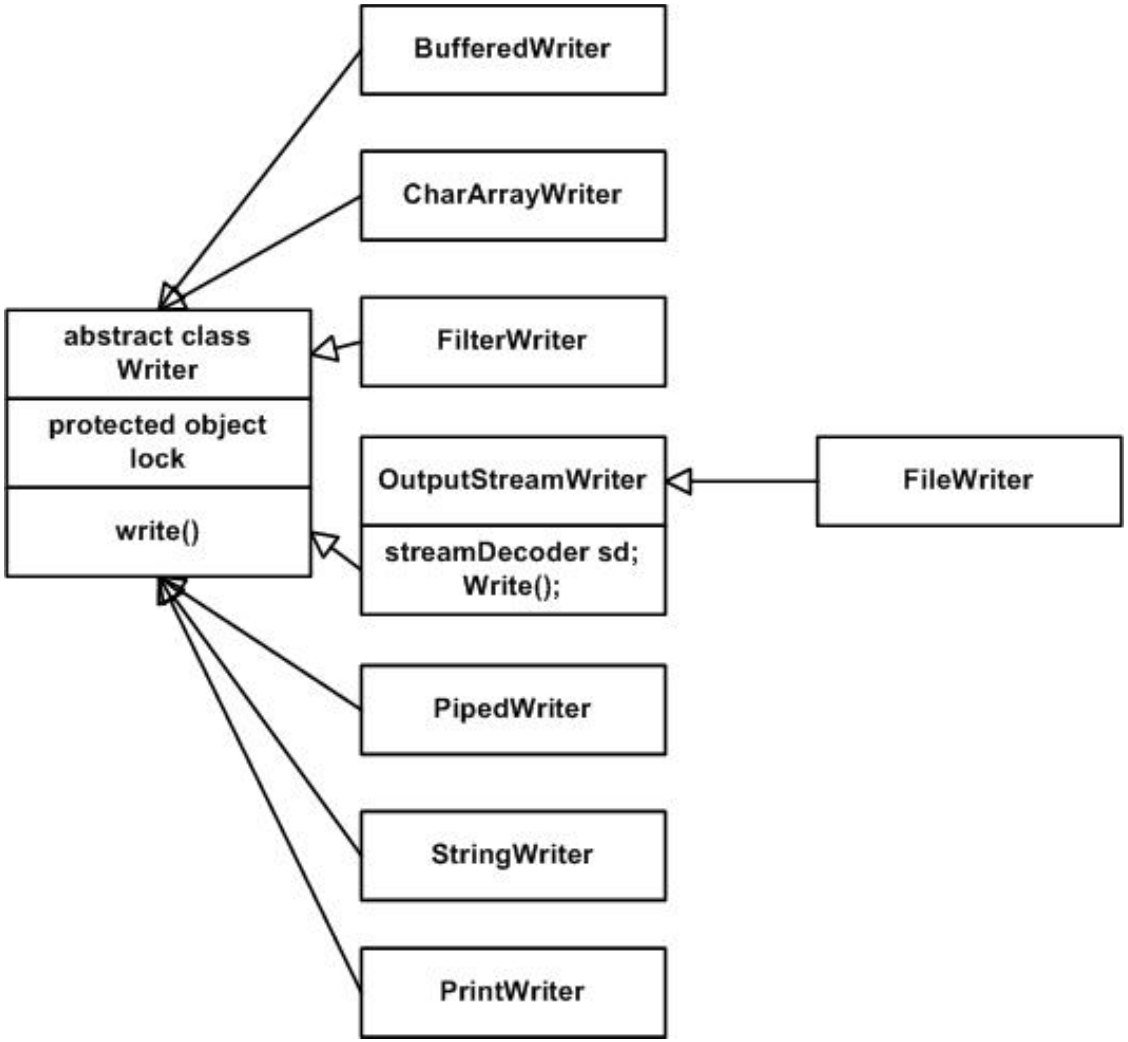


http://doc.sumy.ua/prog/java/exp/ch10_01.htm

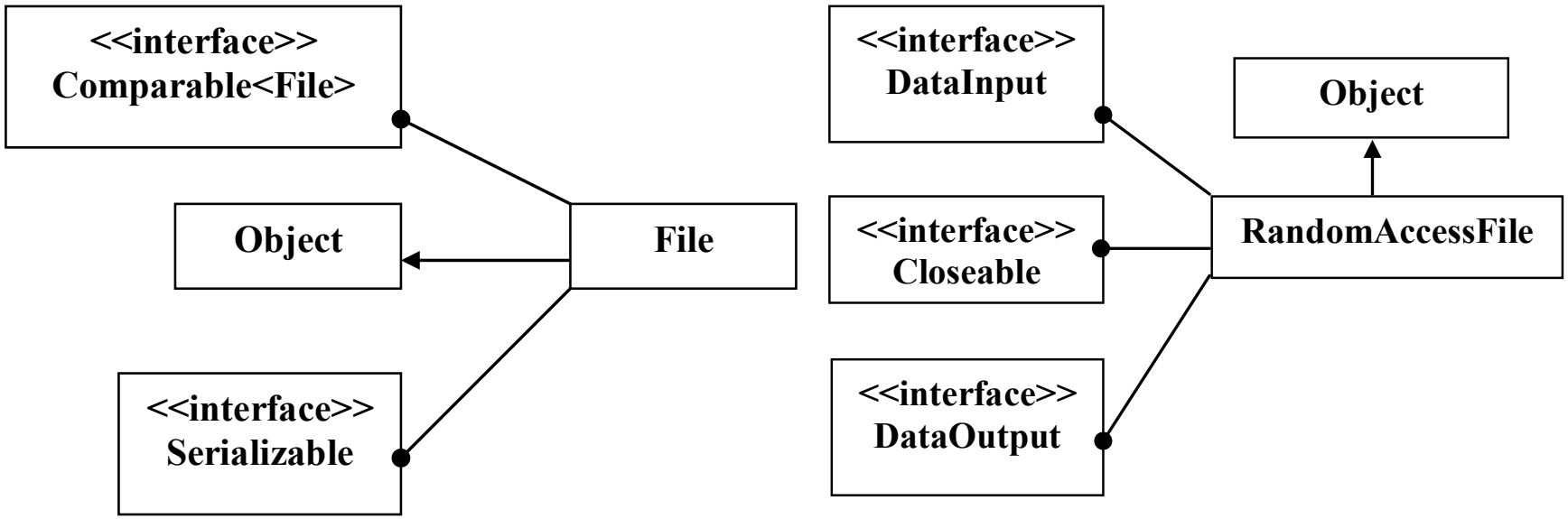
2.1 Java I/O – char level reading



2.1 Java I/O – char level writing

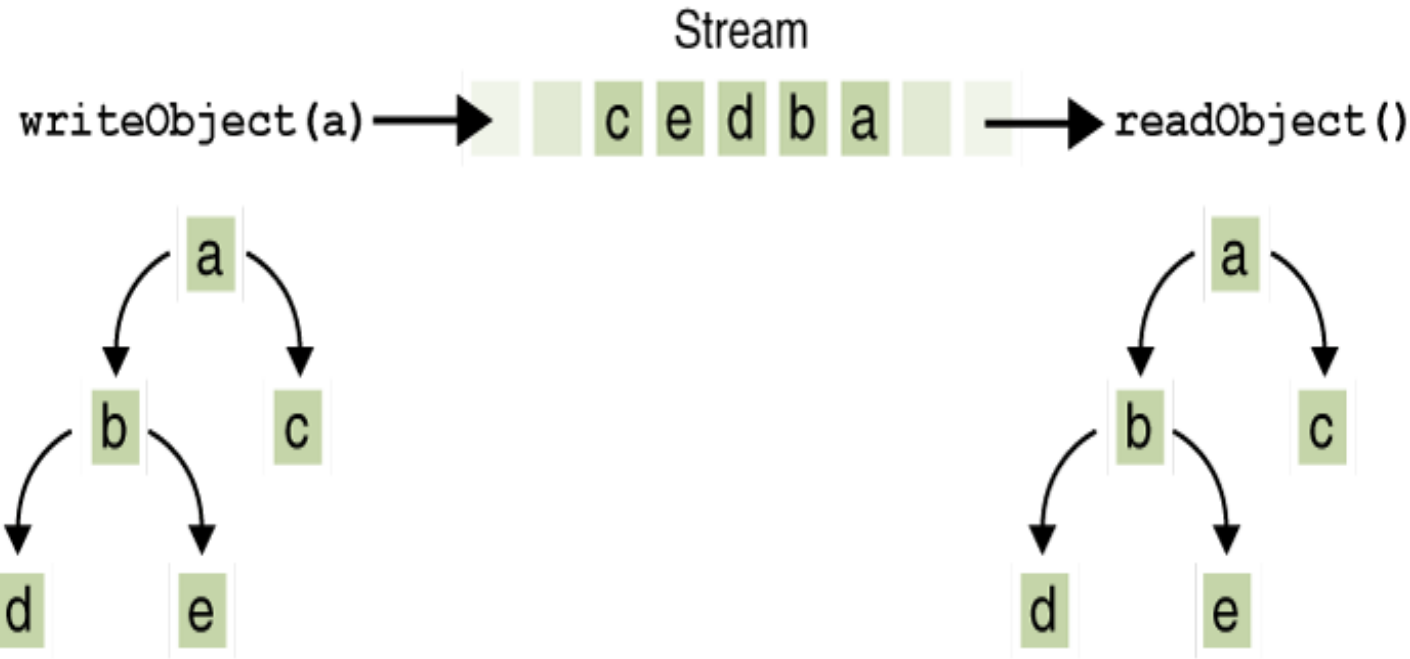


2.1 Java I/O – File Access



2.1 Java I/O – Serialization

This is demonstrated in the following figure, where `writeObject` is invoked to write a single object named **a**. This object contains references to objects **b** and **c**, while **b** contains references to **d** and **e**. Invoking `writeobject(a)` writes not just **a**, but all the objects necessary to reconstitute **a**, so the other four objects in this web are written also. When **a** is read back by `readObject`, the other four objects are read back as well, and all the original object references are preserved.



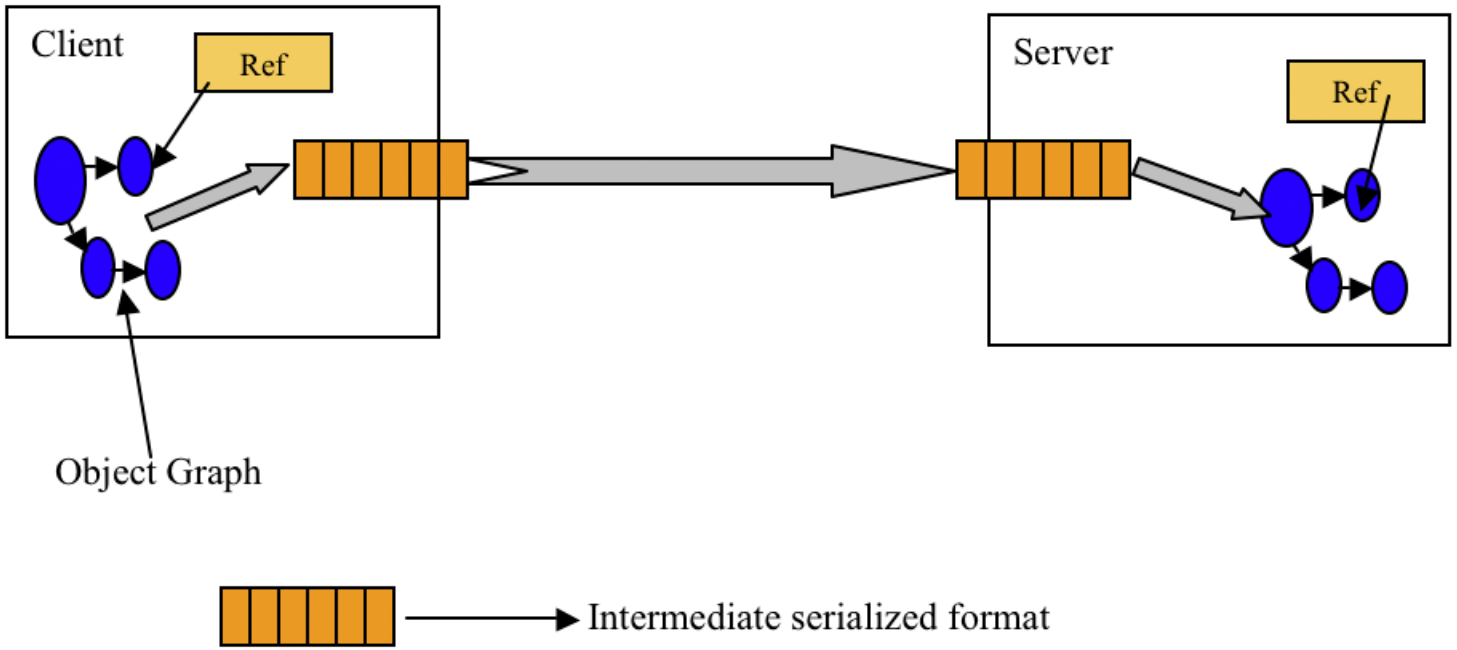
I/O of multiple referred-to objects

2.1 Java I/O – Serialization

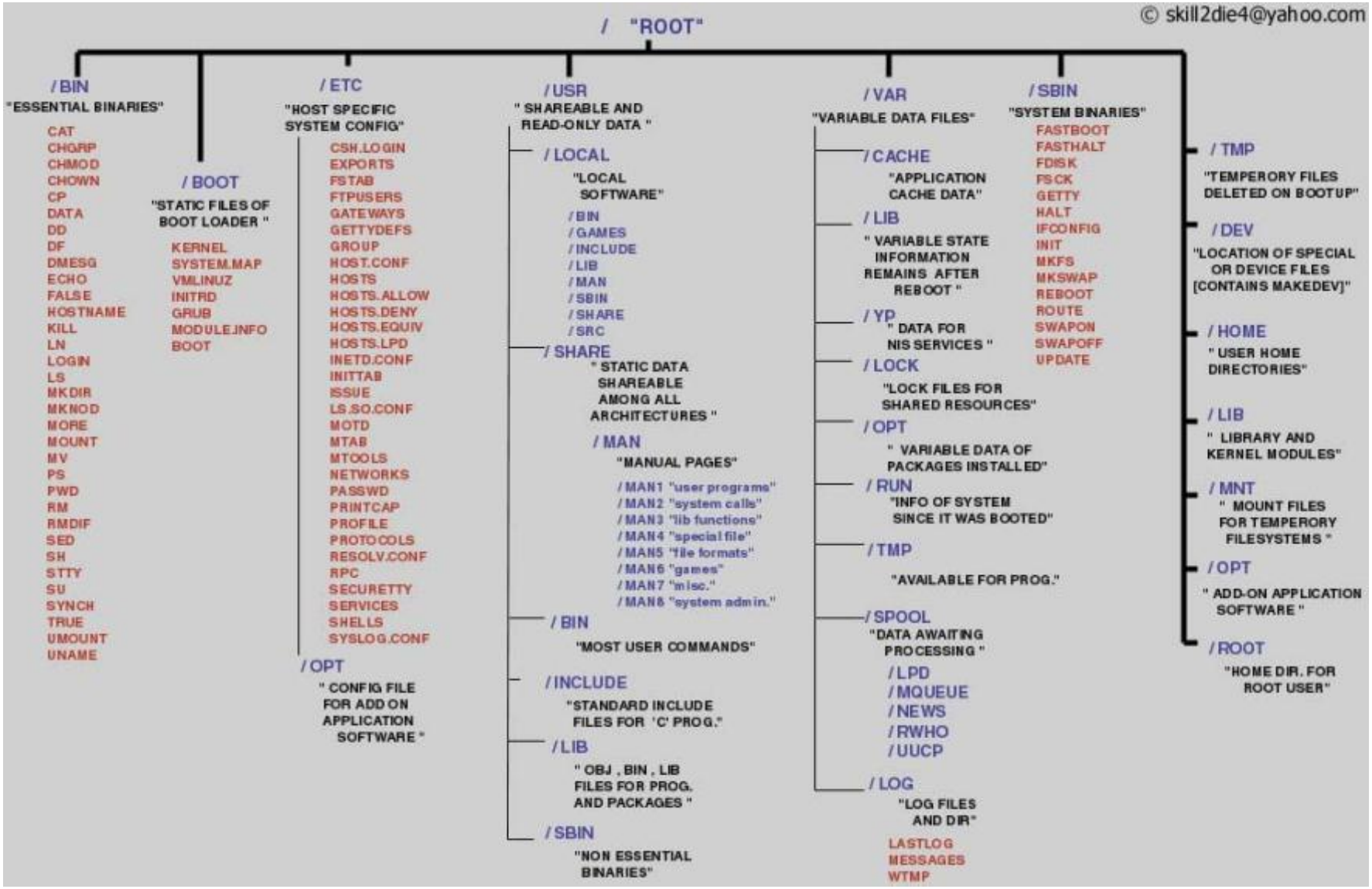
What is going to be saved and restored by serialization in Java?

- **Non-static fields? Static fields?**
- **Transient fields?**
- **Private and public fields and/or methods?**
- **Prototype / signature of the methods and / or the implementation of the methods?**

http://www.javaworld.com/community/node/2915



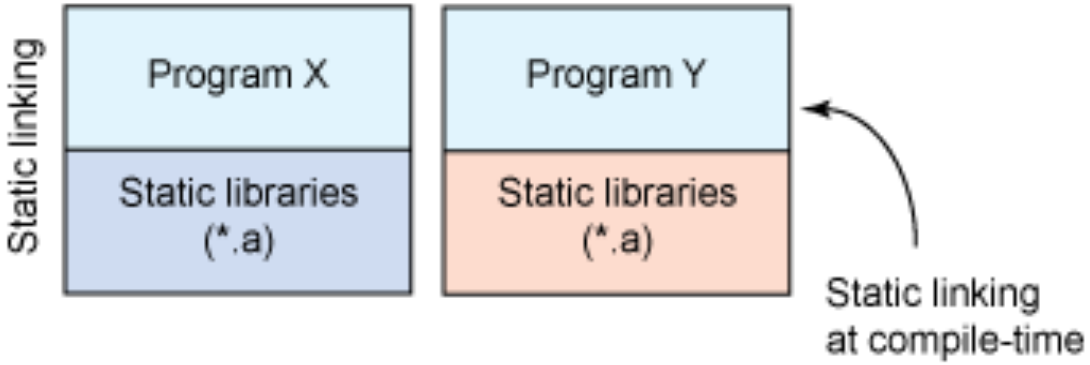
2.2 JNI – Linux Directories

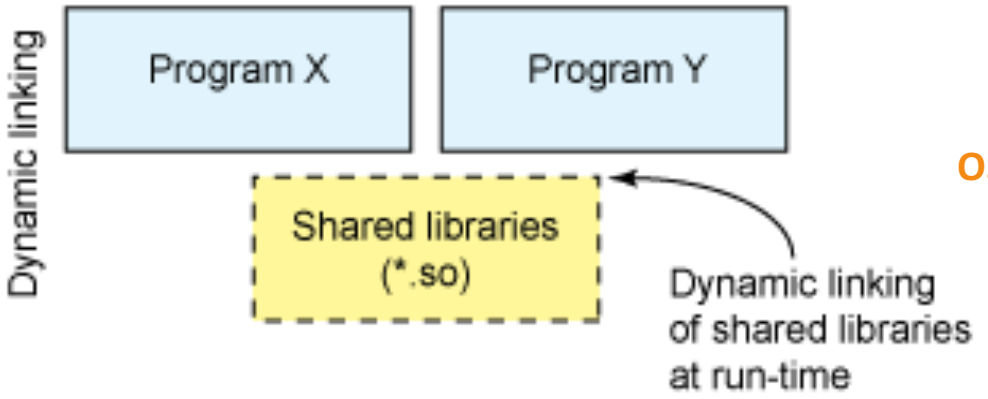


http://freshtutorial.com/file-structure-linux/

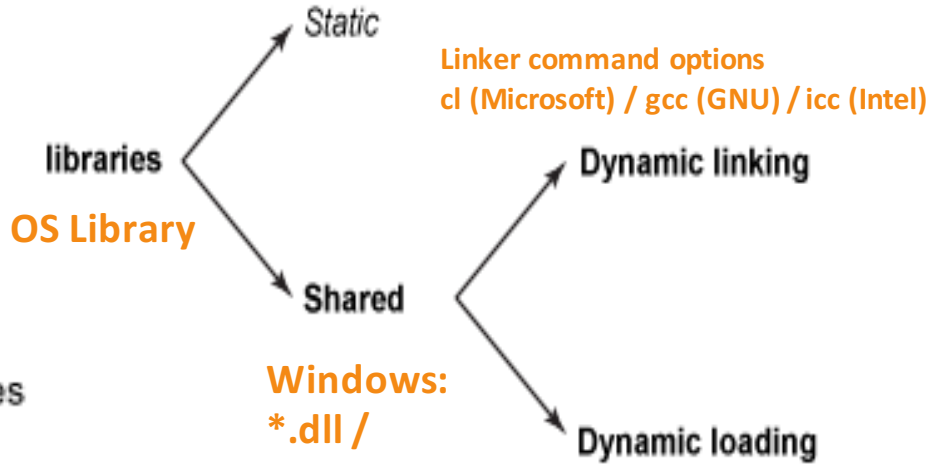
2.2 JNI – Linux vs. Win libraries

http://www.ibm.com/developerworks/linux/library/l-dynamic-libraries/





Windows: *.lib / Linux: *.a



Linker command options
cl (Microsoft) / gcc (GNU) / icc (Intel)

OS Library

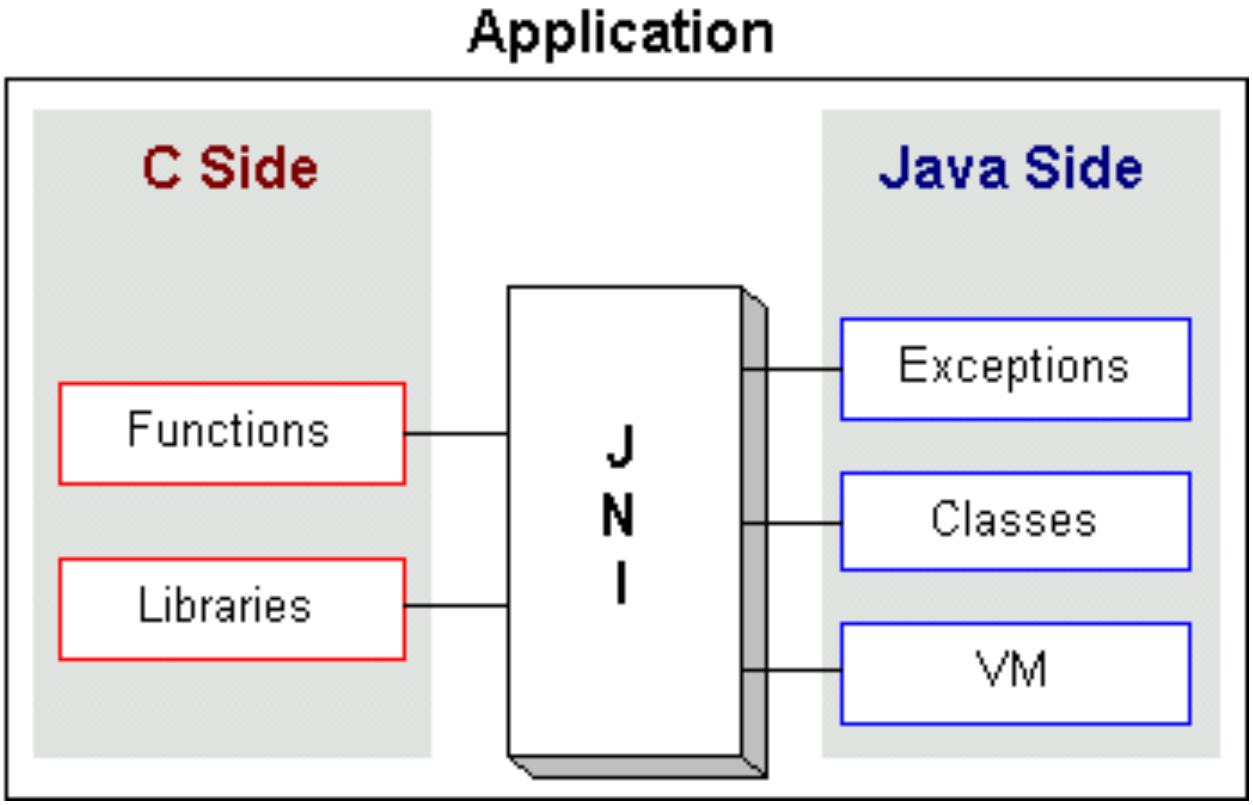
Windows: *.dll /

Linux: *.so

@ Run-time

```
/* Open the shared object */
dl_handle = dlopen( lib, RTLD_LAZY );
if (!dl_handle) {
  printf( "!!! %s\n", dlerror() );
  return;
}
```

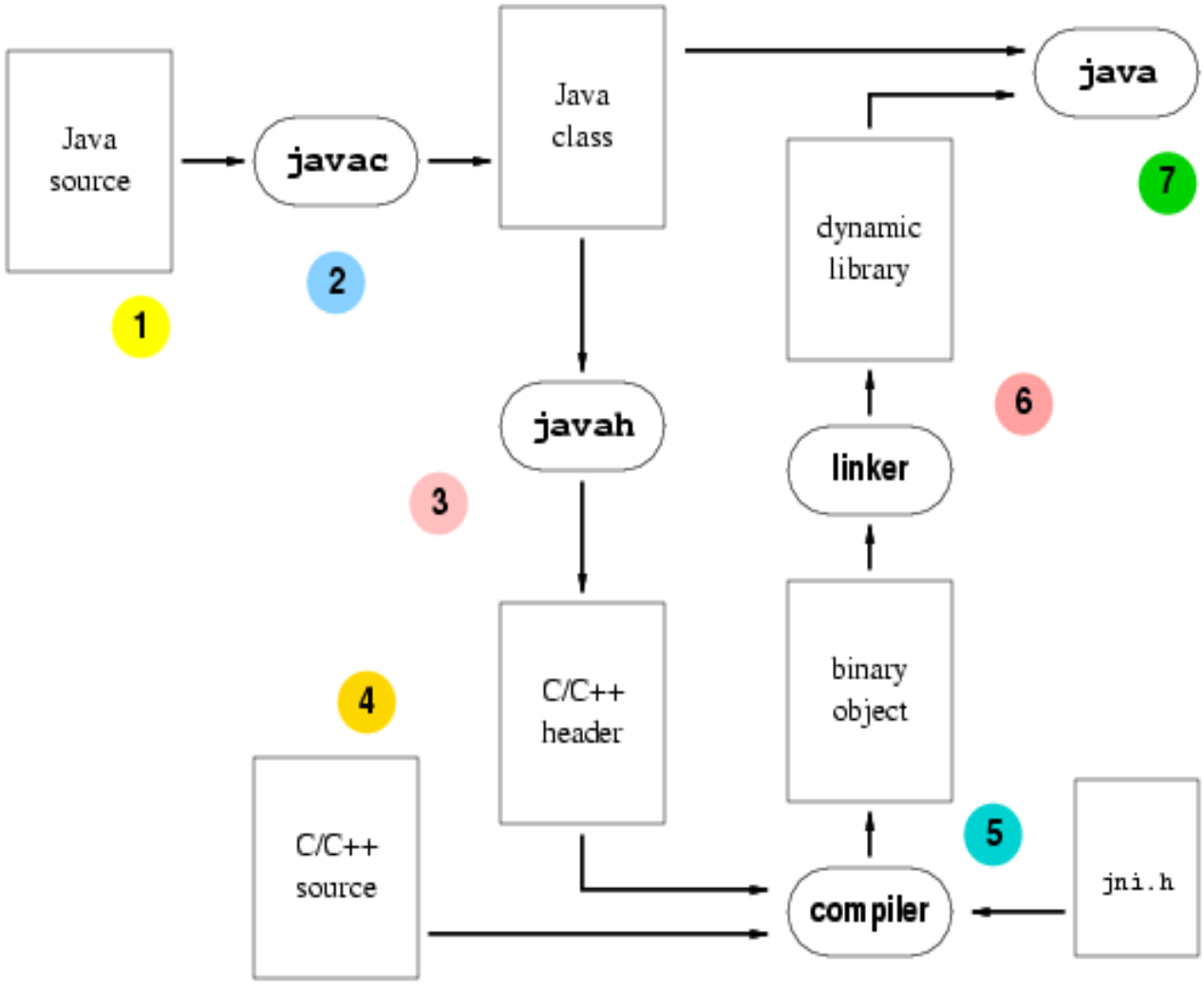
2.2 Java Native Interface



http://www.iam.ubc.ca/guides/javatut99/native1.1/concepts/index.html

2.2 Java Native Interface

http://cs.fit.edu/~ryan/java/language/jni.html



1. Create Java source code with native methods

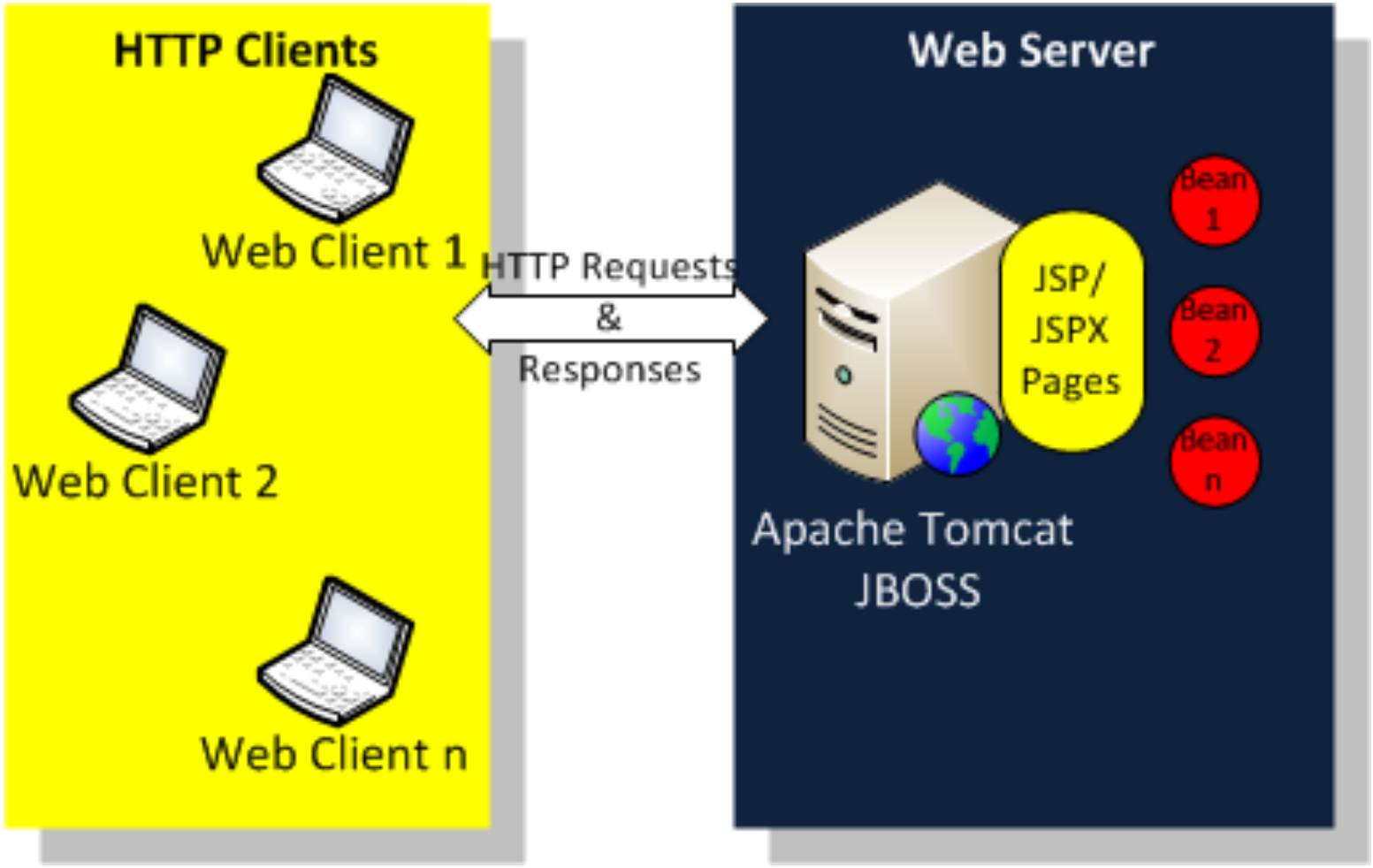
native *return type* method (*arguments*);

2. Compile Java source code and obtain the class files
3. Generate C/C++ headers for the native methods; javah gets the info it needs from the class files
4. Write the C/C++ source code for the native method using the function prototype from the generated include file and the typedefs from include/jni.h
5. Compile the C/C++ with the right header files
6. Use the linker to create a dynamic library file
7. Execute a Java program that loads the dynamic library

static {
System.loadLibrary("*dynamic libary*"); }

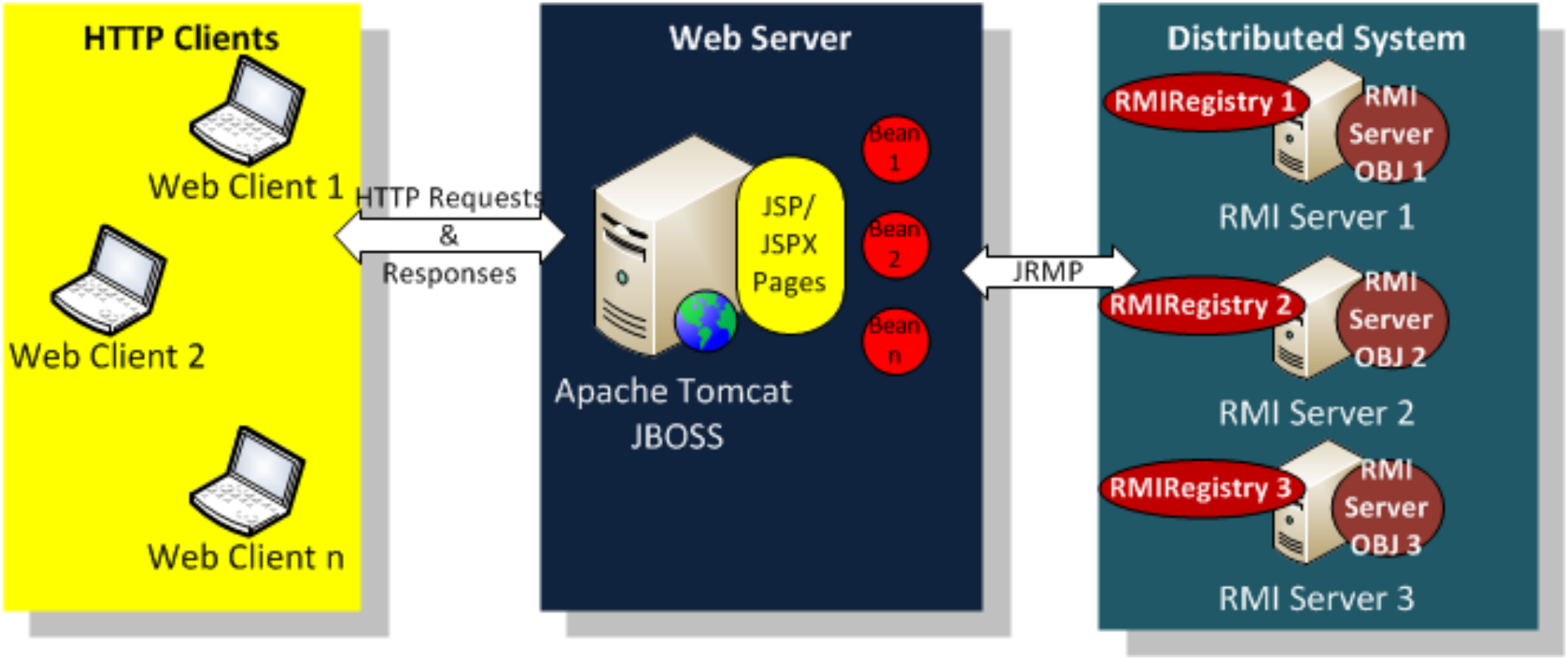
2. Advanced Java RMI DEMO

Java RMI DEMO – Web Local Processing WITHOUT RMI



2. Advanced Java RMI DEMO

Java RMI DEMO – Web Local Processing WITH RMI And in Amazon AWS EC2/EC3 IaaS Cloud



Section Conclusions

Java RMI – Remote method Invocation Technology is the base of Core Distributed Middleware Programming in JEE

EJB – Enterprise Java Beans may communicate each-other via Java RMI

Java RMI network protocol – JRMP is not open

Java RMI subs/skeleton are involved in network communication, objects serialization and deserialization – marshaling/un-marshaling procedure

Java RMI is using the "name service" provided by 'rmiregistry' application from $JAVA_HOME/bin

Java RMI DEMO for pictures processing – local & distributed IN AMAZON AWS EC2

Java RMI DEMO
for easy sharing

Distributed Application Development

Communicate & Exchange Ideas

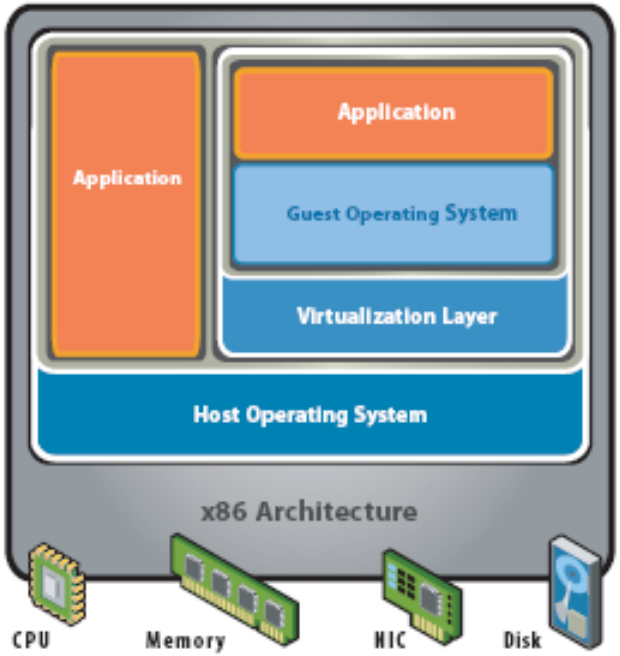
Questions & Answers!

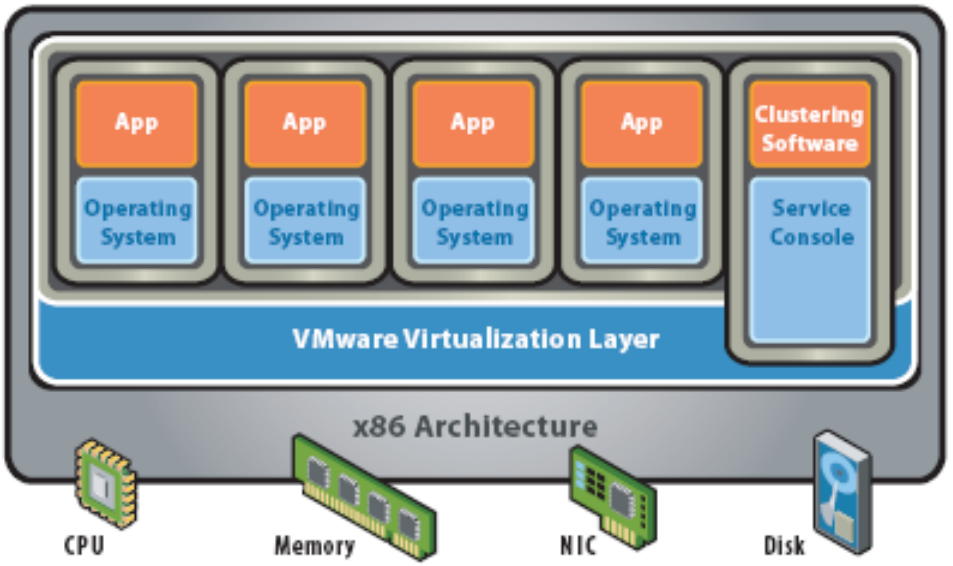
But wait…

There's More!

Distributed Systems: Cloud Architecture

Cloud Concepts – Intro – Virtualization Overview





Hosted Architecture

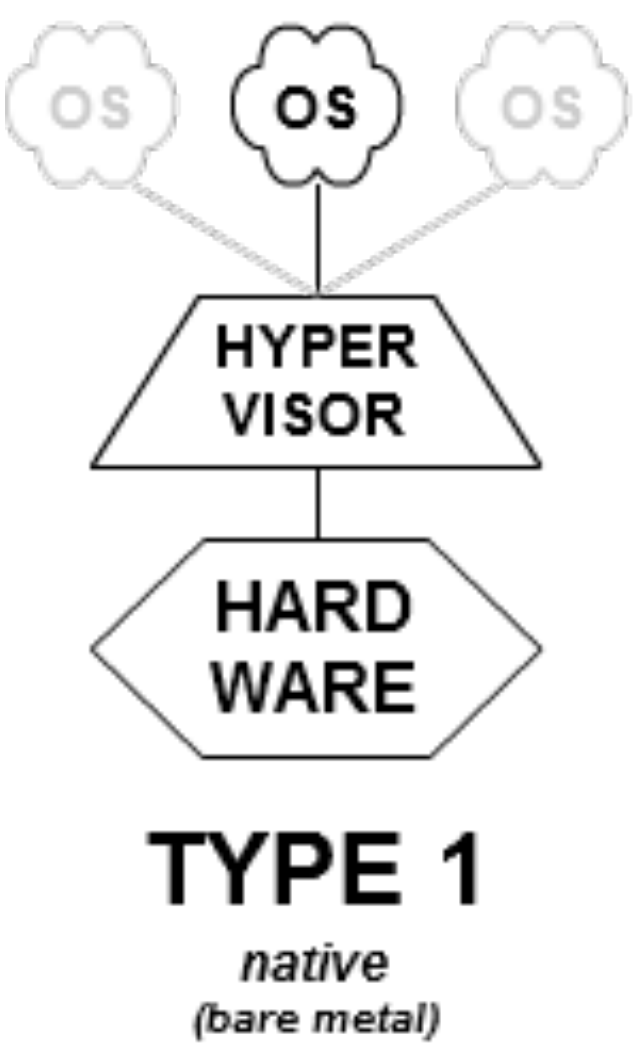
- Installs and runs as an application
- Relies on host OS for device support and physical resource management

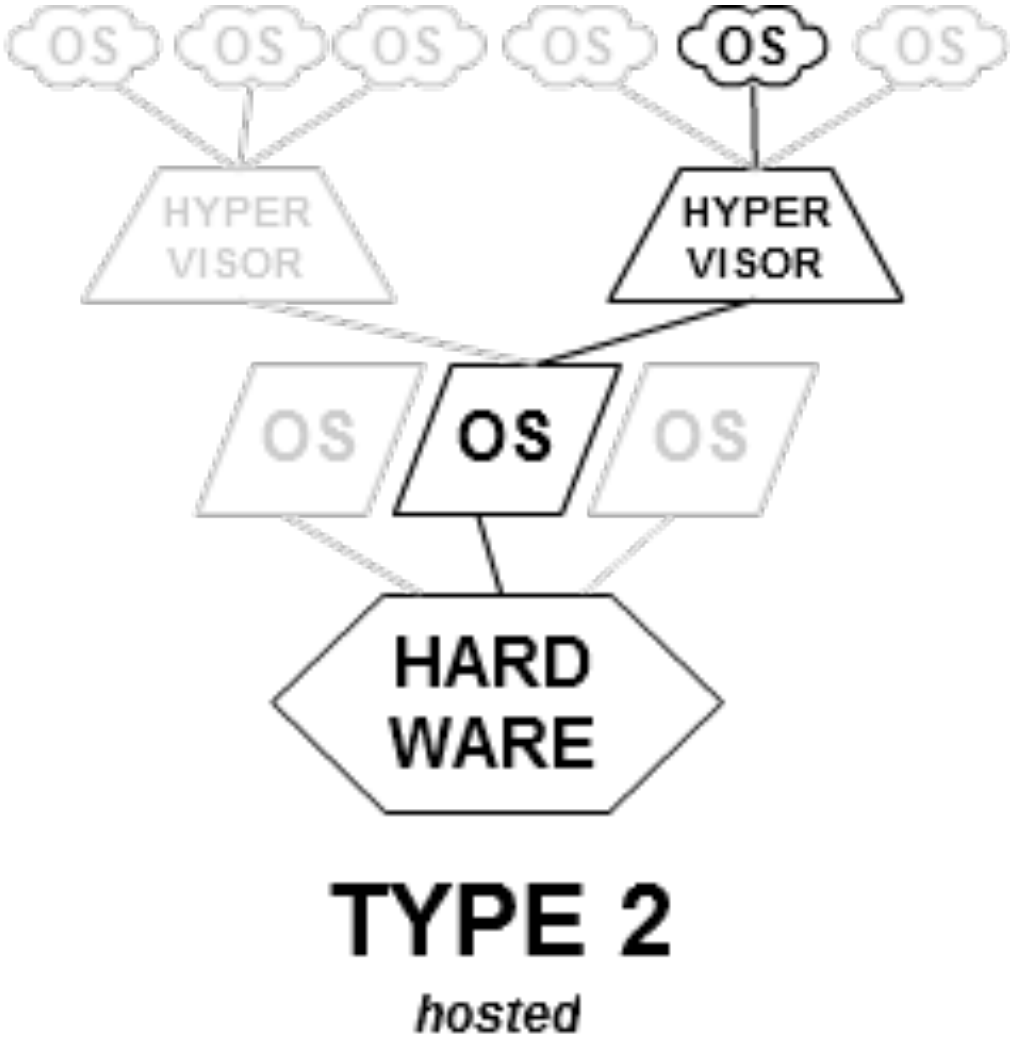
Bare-Metal (Hypervisor) Architecture

- Lean virtualization-centric kernel
- Service Console for agents and helper applications

Figure 2: Virtualization Architectures

Cloud Concepts – Intro – Virtualization Overview

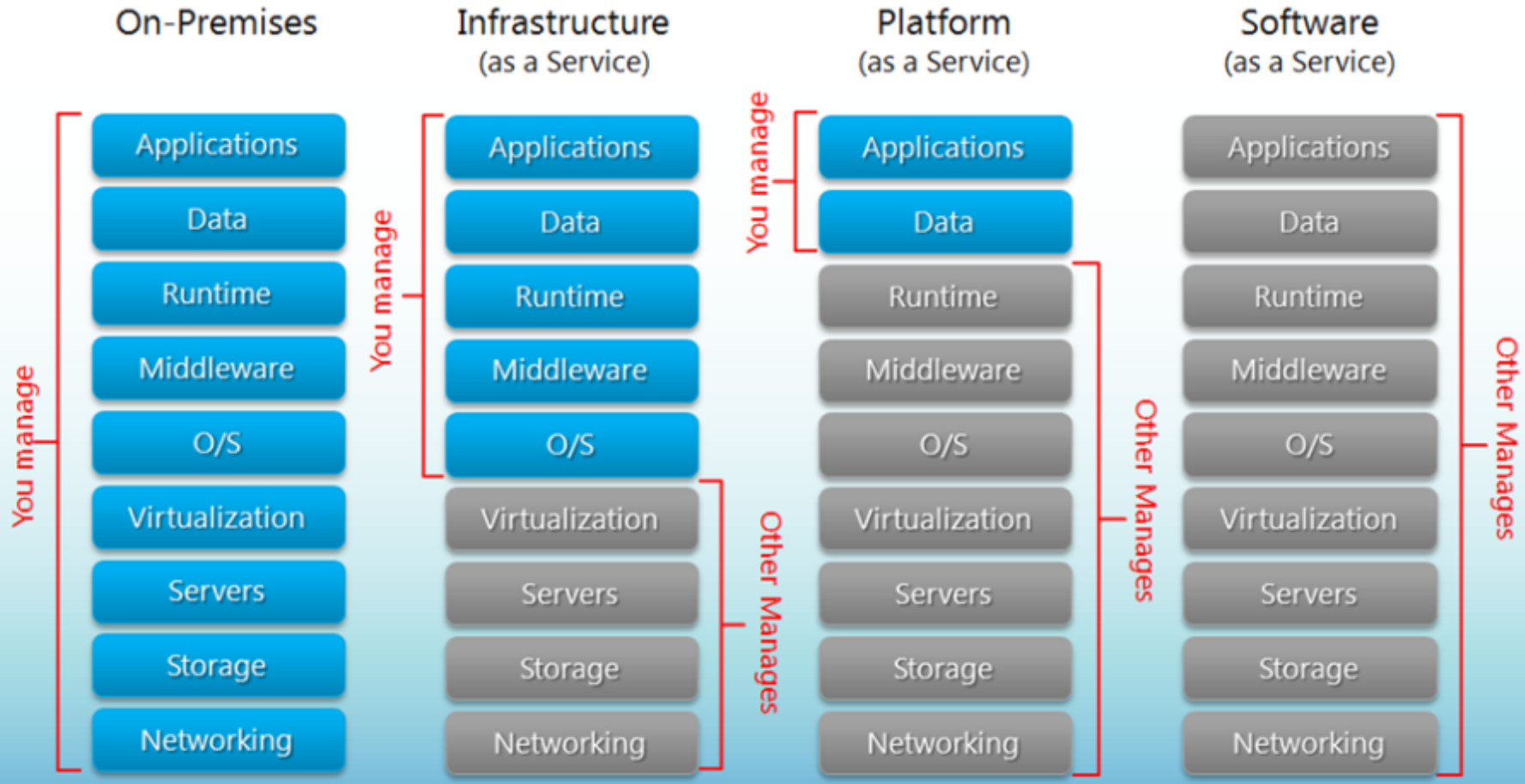




1.3 Distributed Systems: Cloud

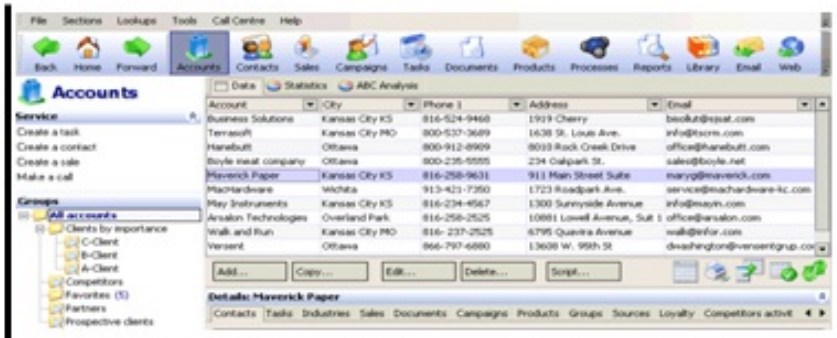
Separation of Responsibilities

Cloud Concepts – Intro – IaaS, PaaS, SaaS



Cloud Concepts – Intro – IaaS, PaaS, SaaS

SaaS



SalesForce.com, Google Apps

PaaS

Tomcat App Server

Java VM

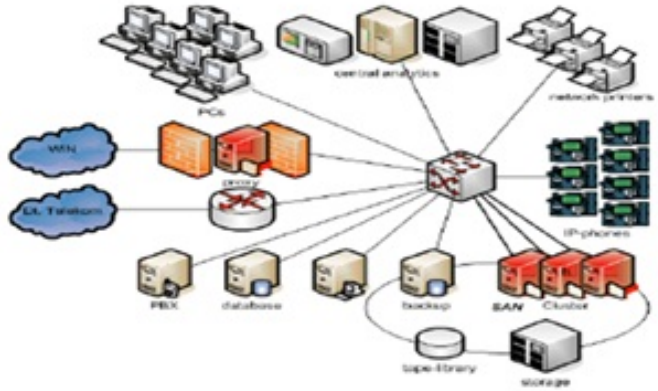
MySQL DB

Ubuntu Linux

Google App Engine for: Java, Ruby, Python & GO

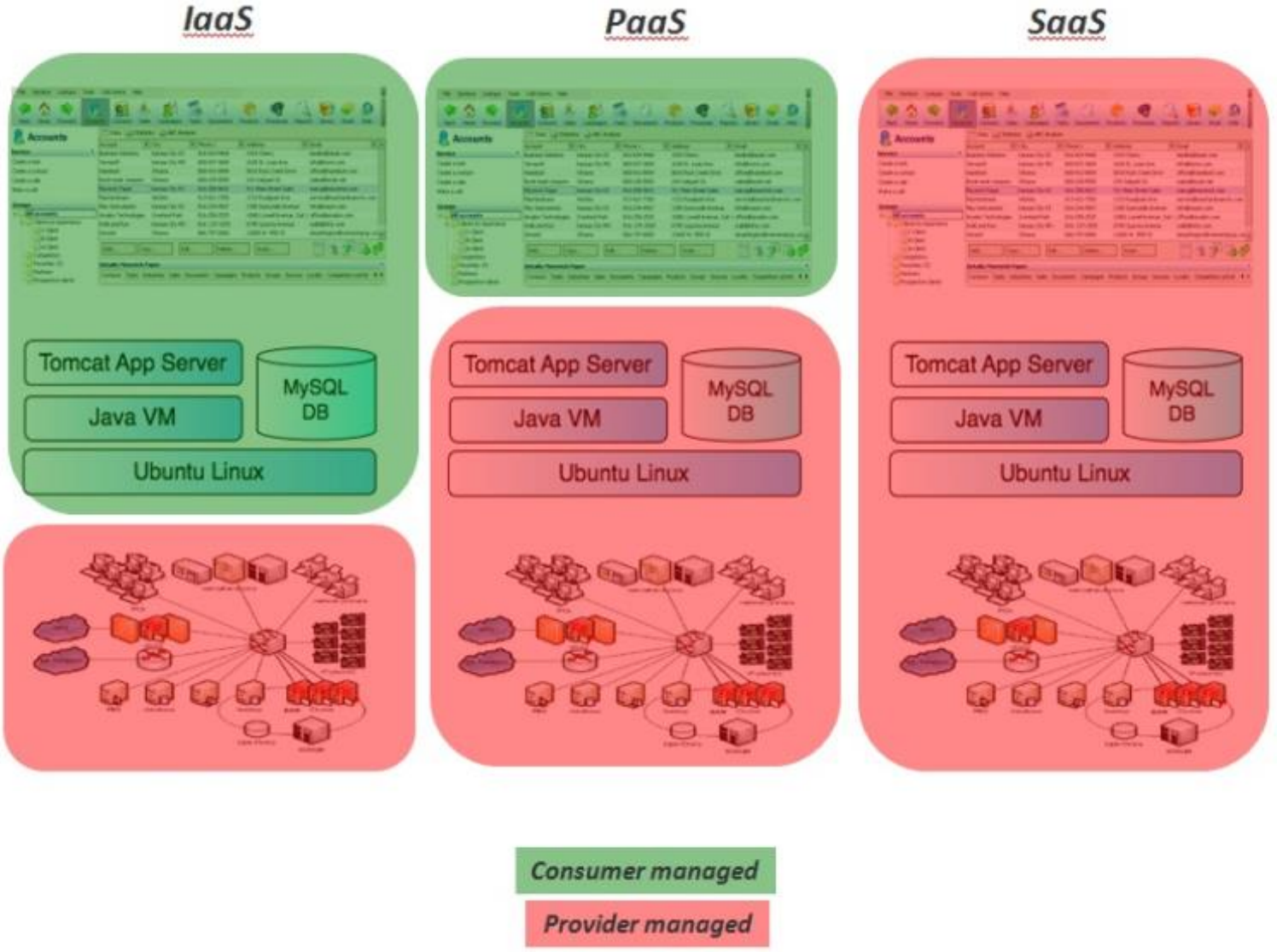
VMForce.com, MS Azure

IaaS



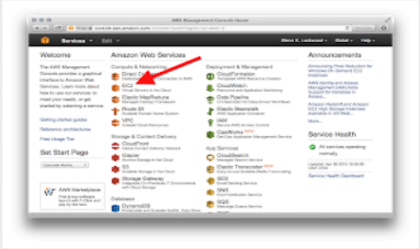
vCloud Express/Datacenter, Amazon EC2

Cloud Concepts – Intro – IaaS, PaaS, SaaS



Step 1: Establish an EC2 Account

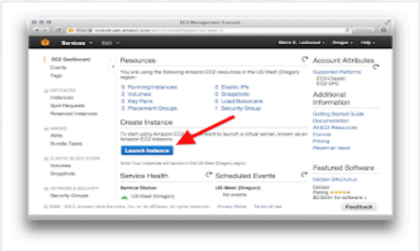
Go to http://aws.amazon.com/ and click "My Account / Console" at the top right, then navigate to the "AWS Management Console." You'll be asked to log in, and if you already have an Amazon.com account, you can use that one. You are then prompted to sign up for AWS, which is a process that requires a credit card number on file to charge for instances, and a robocall to verify your identity. Once this is done, find your way to the AWS Management Console and click the "EC2" option:



This should take you to the EC2 Dashboard from where you can launch VMs.

Step 2: Create your Instances

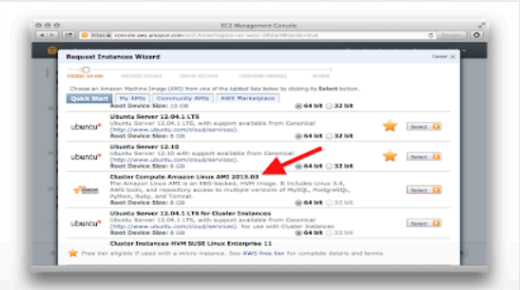
Click the big blue "Launch Instances" button to get into the wizard.



I used the Classic Wizard.

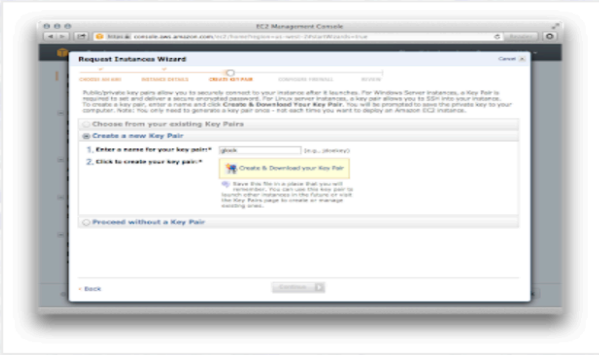
Select a VM Image

The first thing the wizard asks you to do is select an AMI (Amazon Machine Image). Scroll down and choose the Cluster Compute Amazon Linux ABI.



Create a Key Pair

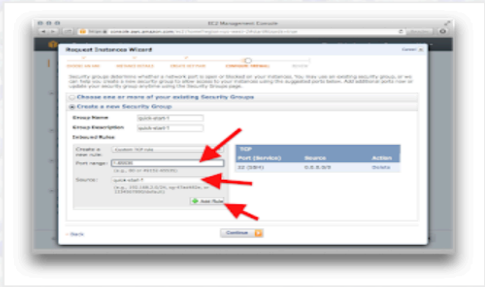
The next step, creating your key pair, is critical to being able to get into your instances once they boot up.



Give your key an arbitrary name, then download it as a .pem file. This is the privatekey that will allow you to ssh into the instance once it's up; if you don't have this, you will simply be locked out of your own instance.

Set Proper Security Group

The next step of customizing your cluster's security group is also essential.



The default behavior of the wizard is to have you create a new security group called "quick-start-1" that will open port 22 (SSH) to the world but block all traffic everywhere else. You need to let all instances (nodes) within your cluster communicate freely for MPI to work, so you must add a new rule to this security group that opens all ports (1-65535) to all other instances within the same security group (quick-start-1). Once you enter these two parameters, you must then click the Add Rule button or this rule will not be applied and MPI will not work!

Launch Your Instances

Thanks!

DAD – Distributed Application Development
End of Lecture 6