

Bule Hora University
Faculty of Engineering and Technology
Department of Information technology

Year: III

Semester: II

Subject: Introduction to Distributed System

Course Code: ITec3102

Lecture Notes Unit III - Naming and Threads

1.1 Naming and Name Services
1.2 Remote Invocation
1.3 Processes and Threads

1.1. Naming and Name Services

1.1.1. Names or Codes, or Numbers

- Names (when meaningful) are easier to remember than codes or numbers.
- Number (or sequence codes) are more useful for structuring data and locating resources by a program..
- Example: IPv4

1.1.2. Naming Services

- In a distributed system, names are used to refer to a wide variety of resources such as:
 - Computers, services, remote objects, and files, as well as users.
- Naming is fundamental issue in DS design as it facilitates communication and resource sharing.

Definition

In a Distributed System, a Naming Service is a specific service whose aim is to provide a consistent and uniform naming of resources, thus allowing other programs or services to localize them and obtain the required metadata for interacting with them.

- The major operation that a name service supports is to resolve names.

Key benefits

- Resource localization- Resources are accessed using *identifier* or *reference*
- Uniform naming
- Device independent address (e.g., you can move domain name/web site from one server to another server seamlessly).

The role of names and name services

- Currently, different name systems are used for each type of resource:

resource name identifies

file pathname file within a given file system

process process id process on a given computer

port port number IP port on a given computer

- Uniform Resource Identifiers (URI) offer a general solution for any type of resource. There two main classes:

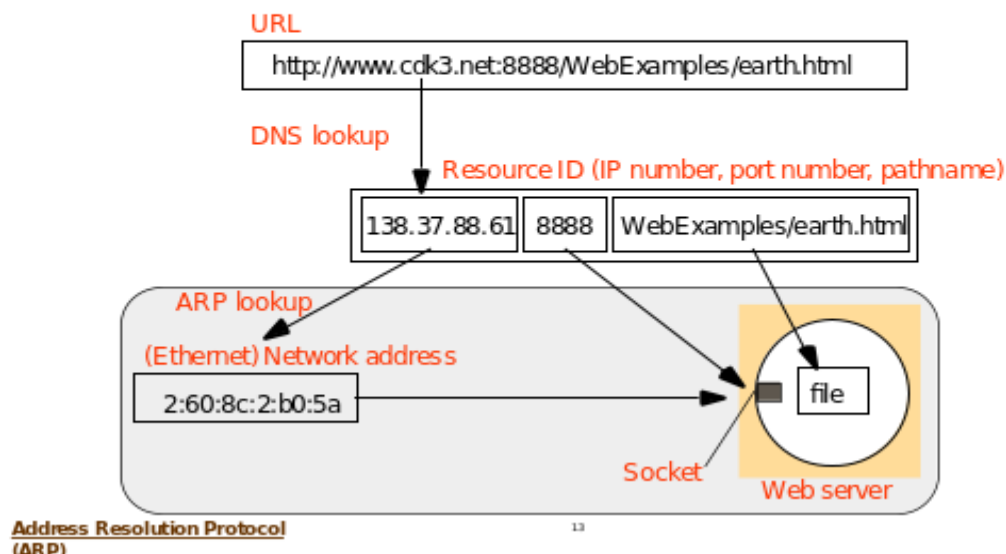
URL Uniform Resource Locator (URL)

- typed by the protocol field (http, ftp, nfs, etc.)
- part of the name is service-specific
- resources cannot be moved between domains

URN Uniform Resource Name (URN)

- requires a universal resource name lookup service - a DNS-like system for all resources

Composed naming domains used to access a resource from a URL

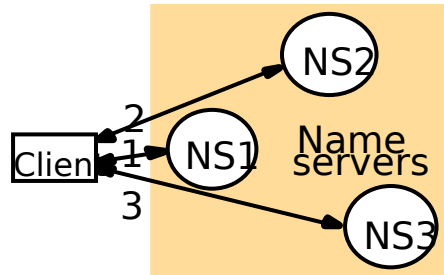


Navigation

Navigation is the act of chaining multiple Naming Services in order to resolve a single name to the corresponding resource.

Iterative navigation

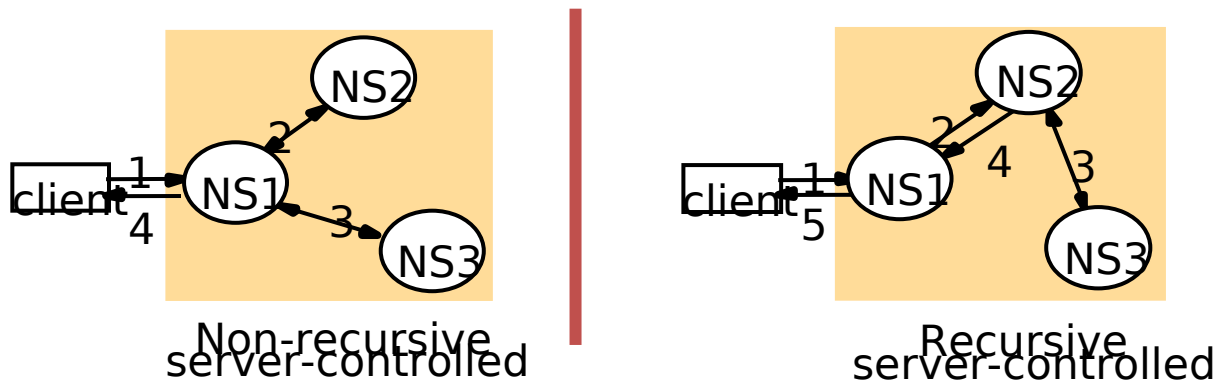
The client computer must determine which server this is.



A client iteratively contacts name servers NS1-NS3 in order to resolve a name

Server controlled navigation

- Recursive:
 - ♦ it is performed by the naming server
 - ♦ the server becomes like a client for the next server
- Non recursive:
 - ♦ it is performed by the client or the first server
 - ♦ the server bounces back the next hop to its client



1.1.3. Domain name system

- Rapidly resolves domain names to IP addresses

Basic DNS algorithm for name resolution (domain name -> IP number)

Look for the name in the local cache

Try a superior DNS server, which responds with:

- another recommended DNS server
- the IP address (which may not be entirely up to date)

1.1.4. Directory services

Directory services can help with above situation: they store collections of bindings and attributes and also looks up entries that match attribute-based specs.

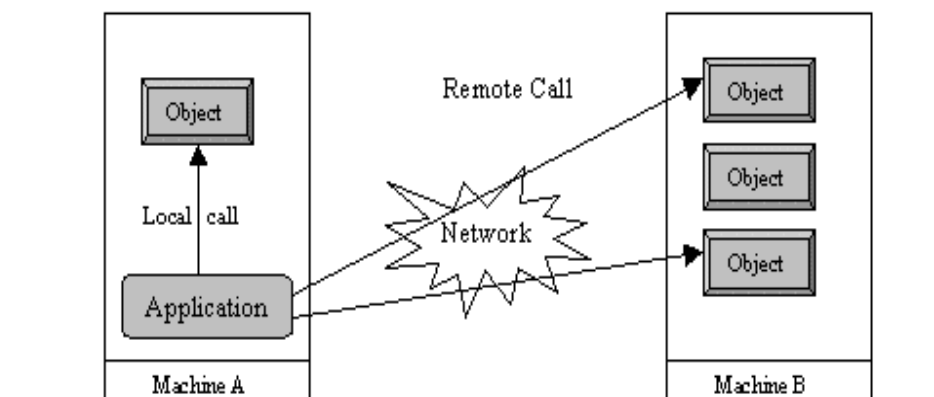
1.2. Remote Invocation

-invoking the method (service) available in remote computer.

Some remote method concepts are

1. RPC (Remote Procedure Call)
2. Remote Method Invocation (RMI)

Local vs. Remote method invocation



1.2.1. RMI

- RMI is the action of invoking a method of a remote interface on a remote object.
- It is used to develop applications that communicate between virtual machines
- RMI is a means of doing distributed computing
- RMI hides the underlying mechanism of transporting method arguments and return values across the network.

Example

- Server returns sum of two numbers
- Client calls the add() method of the server and passes two numbers
- We will need four files
 - AddServerIntf.java
 - AddServerImpl.java
 - AddServer.java
 - AddClient.java

Programming with RMI

Programming with RMI involves

- Define a remote interface
- Provide an implementation of the remote interface
- Develop a Server
- Develop a client
- Generate stubs and skeletons
- Start the RMI registry
- Run the client and server

Step1: Defining a remote interface

AddServerIntf.java

- A *remote* interface is an interface that declares a set of methods that may be invoked from a remote Java virtual machine.
- Must extend `java.rmi.Remote`
- All methods must throw `RemoteException`

```
import java.rmi.*;

public interface AddServerIntf extends Remote
{
    double add (double d1, double d2) throws RemoteException;
}
```

Step 2: Providing an implementation of the remote interface

- Implement the methods that can be invoked remotely
- The implementation class must extend `UnicastRemoteObject` that "exports" the Server Object.

AddServerImpl.java

```
import java.rmi.*;
import java.rmi.server.*;
public class AddServerImpl extends UnicastRemoteObject implements
AddServerIntf
```

```

    {
        public AddServerImpl() throws RemoteException{
            public double add (double d1, double d2) throws
RemoteException
            {
                return d1+d2;
            }
        }
    }

```

Step 3: Developing a Server

- The remote object must register itself with the RMI naming registry

```

import java.rmi.*;

public class AddServer
{
    public static void main(String[] args)
    { try{
        AddServerImpl addServerImpl=new AddServerImpl();
        Naming.rebind("AddServer",addServerImpl);
    } catch(Exception e){System.out.println(e);}
    }
}

```

Step 4: Developing a client

- A reference to the remote object is obtained by the client by looking up the registry
- The method is invoked using the reference.

```

import java.rmi.*;

public class AddClient
{
    public static void main(String[] args)
    {
        try{
            String addServerURL = "rmi://" + args[0] + "/AddServer";

            AddServerIntf addServerIntf=
            (AddServerIntf)Naming.lookup(addServerURL);

```

```

        double d1=Double.valueOf(args[1]).doubleValue();
        double d2=Double.valueOf(args[2]).doubleValue();
        System.out.println("Sum= "+addServerIntf.add(d1,d2));
    } catch(Exception e){System.out.println(e);}
}
}

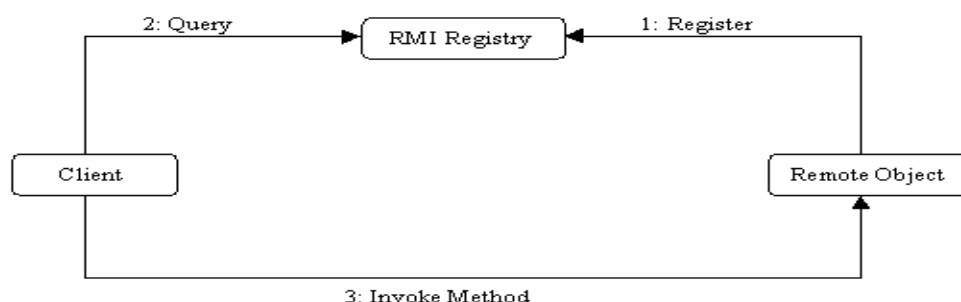
```

Step 5: Generate stubs and skeletons

- Use the rmic compiler
- `rmic AddServerImpl`
- Stub
 - resides on client
 - does marshalling and unmarshalling
 - Provides interface of the server
- Skeleton
 - Resides on server
 - does marshalling and unmarshalling

Step 6: Start the RMI registry

- Simple Name Repository. It is a naming service that allows clients to obtains references to remote objects
- Server binds a name with an object implementation
- Client can query the registry for checking the availability of a server object



Step 6: Run Server Program and Client Programs

Run the server program then run the client program in order to get the result

1.3. Processes and Threads

⌘ Process (program in execution):

- Unit of resource management for operating system.
- Execution environment:
 - Address space (A unit of management a process's virtual memory).
 - Thread synchronization and communication resources (e.g. semaphores).
 - Computing resources (file systems, windows, etc.)
- Expensive to create and manage.

⌘ Threads (lightweight process):

- Individual and separate unit of execution that is part of a process
- multiple threads can work together to accomplish a common goal
- Arise from the need for concurrent activities to share resources within one process.
 - Enable to overlap computation with input and output.
 - Allow concurrent processing of client requests in servers – each request handled by one thread.
- Easier to create and destroy.

Advantages of threads

- easier to program
 - 1 thread per task
- can provide better performance
 - thread only runs when needed
- multiple threads can share resources
- utilize multiple processors if available

Disadvantages

- multiple threads can lead to deadlock
 - much more on this later
- overhead of switching between threads

Creating Threads (method 1)

- extending the Thread class
 - must implement the *run()* method
 - thread ends when *run()* method finishes
 - call *start()* to get the thread ready to run

Example:

```
class Output extends Thread {  
    private String toSay;  
    public Output(String st) {
```

```

        toSay = st;
    }
    public void run() {
        try {
            for(;;) {
                System.out.println(toSay);
                sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
}
class Program {
    public static void main(String [] args) {
        Output thr1 = new Output("Hello");
        Output thr2 = new Output("There");
        thr1.start();
        thr2.start();
    }
}

```

Creating Threads (method 2)

- implementing Runnable interface
 - virtually identical to extending Thread class
 - must still define the *run()* method
 - setting up the threads is slightly different

Example

```

class Output implements Runnable {
    private String toSay;
    public Output(String st) {
        toSay = st;
    }
    public void run() {
        try {
            for(;;) {
                System.out.println(toSay);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
}

```

```

class Program1 {
    public static void main(String [] args) {
        Output out1 = new Output("Hello");
        Output out2 = new Output("There");
        Thread thr1 = new Thread(out1);
        Thread thr2 = new Thread(out2);
        thr1.start();
        thr2.start();
    }
}

```

States of Java Threads

