Last Class: Network Overview

- =>Processes in a distributed system all communicate via a message exchange.

Physical reality: packets	**Abstraction: messages**
limited size	arbitrary size
unordered (sometimes)	ordered
unreliable	reliable
machine to machine	process to process (routing)
asynchronous	synchronous
insecure	secure

- Unless otherwise stated, assume a reliable end-to-end message delivery.

Today: Distributed Systems

What gets harder when we move from a stand alone system to a distributed environment?

- resource sharing
- timing (e.g., synchronization)
- critical sections
- deadlock detection and recovery
- failure recovery

Resource Sharing

There are many mechanisms for sharing (hardware, software, data) resources.

- **Data Migration:** moving the data around
- **Computation Migration:** move the computation to the data
- **Job Migration**: moving the job (computation and data) or part of the job

=> The fundamental tradeoff in resource sharing is to complete user instructions as fast and as cheaply as possible. (Fast and cheap are usually incompatible.)

Computation versus Communication

- If communication is fast and cheap, we can utilize all the resources in the distributed environment.
- If communication is slow and expensive, we should do most processing locally.
- Reality is usually in the middle somewhere.

=> we need a quantitative analysis to decide where the cutoffs are.

Data Migration

Data Migration may occur when process at site A accesses a file at site B.

1. Copy file B to process A
   - Costly if the file is large
   - Data must be converted to A's data format
   - Multiple copies can cause *consistency* problems
   - All subsequent accesses at A are local
2. Keep file at B, access file remotely from A
   - Saves file transfer cost
   - Converting the file from A's format to B's may be difficult to do in pieces
   - Single copy of file, so no consistency problems
   - Single "file service center" for file at B may be a performance bottleneck

Computation Migration

Computation Migration may occur when it is more efficient to transfer the computation itself rather than data.

Example: a small program which produces a short summary of a large file, say "wc" or a database query

- Remote Procedure Calls (**RPC**): suppose A wants to access file at site B. B provides a predefined process.

  A sends a message to the predefined process at B, which performs the requested action, and sends the result back to A.

Job Migration

- **Job Migration:** perform the job (or parts of the job) at remote sites by moving the data *and* computation.
  - **Load balancing:** even workload across the distributed system
  - **Computational speedup:** concurrent (parallel) execution of parts of the job.
  - **Hardware preference:** job may match a given piece of hardware somewhere in the system.
  - **Software preference:** job may require software only available on a specific site. For example, site specific license of expensive software.
  - User interaction: may want to hide migration from the user. For example, in load balancing. May want the user to specify migration (hardware/software preferences)

Client/Server Model

- One of the most common models for structuring distributed computation is by using the *client/server* paradigm.
  - A *server* is a process or collection of processes that provide a service, e.g., name service, file service, database service, etc.
  - The server may exist on one or more nodes.
  - A *client* is a program that uses the service.
  - A client first binds to the server, i.e., locates it in the network and establishes a connection.
  - The client then sends the server a request to perform some action. The server sends back a response.
  - RPC is one common way this structure is implemented.

Remote Procedure Call

Basic idea:

- Servers export procedures for some set of clients to call.
- To use the server, the client does a procedure call.
- OS manages the communication.

Remote Procedure Call: Implementation Issues

For each procedure on which we want to support RPC:

- The RPC mechanism uses the procedure *signature* (number and type of arguments and return value)
  - to generate a client stub that bundles up the RPC arguments and sends it off to the server, and
  - to generate the server stub that unpacks the message, and makes the procedure call.

Remote Procedure Call: Implementation Issues

Client Stub:

build message
send message
wait for response
unpack reply
return result

Server Stub:

create threads
loop
    wait for a command
    unpack request parameters
    call procedure with thread
    build reply with result(s)
    send reply
end loop

Comparison between RPC and a regular procedure call

- Name of procedure
- Parameters
- Result
- Return address

Remote Procedure Call

- How does the client know the right port?
  - The binding can be static - fixed at compile time.
  - Or the binding can be dynamic - fixed at runtime.
- In most RPC systems, dynamic binding is performed using a name service.
  - When the server starts up, it exports its interface and identifies itself to a network name server
  - The client, before issuing any calls, asks the name service for the location of a server whose name it knows and then establishes a connection with the server.

Example: Remote Method Invocation (RMI) in Java

- Java provides the following classes/interfaces:
  - **Naming:** class that provides the calls to communicate with the remote object registry
  - public static void bind(String name, Remote obj) - Binds a server to a name.
  - public static Remote lookup(String name) - Returns the server object that corresponds to a name.
- **UnicastRemoteObject:** supports references to non-replicated remote objects using TCP, exports the interface automatically when the server object is constructed
- Java provides the following tools:
  - **rmiregistry** server-side name server
  - **rmic:** given the server interface, generates client and server stubs that create and interpret packets

Example: Server in Java

- **Server**
  - Defines an interface listing the signatures of methods the server will satisfy
  - Implements each of the methods in the interface
  - Main program for server:
    - Creates one or more server objects - normal constructor call where the object being constructed is a subclass of RemoteObject
    - Registers the objects with the remote object registry
- **Client**
  - Looks up the server in the remote object registry
  - Uses normal method call syntax for remote methods
  - Should handle RemoteException

Example: Hello World Server Interface

Declare the methods that the server provides:

```
package examples.hello;

// All servers must extend the Remote interface.
public interface Hello extends java.rmi.Remote {

    // Any remote method might throw RemoteException.
    // Indicates network failure.
    String sayHello() throws java.rmi.RemoteException;
}
```


Example: Hello World Server

```
package examples.hello;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl extends UnicastRemoteObject implements Hello
{
    public HelloImpl() throws RemoteException {
     // The superclass constrctor exports the interface and gets a port
      super();
 }

 public String sayHello() throws RemoteException {
     // This is the "service" provided.
     return  "Hello World!";
 }
```

Example: Hello World Server (contd)

```
public static void main(String args[])
 {
    // Create and install a security manager
    System.setSecurityManager(new RMISecurityManager());

    // Construct the server object.
    HelloImpl obj = new HelloImpl();

    // Register the server with the name server.
    Naming.rebind("//myhost/HelloServer", obj);
 }
}
```


Example: Hello World Client

```
package examples.hello;

import java.awt.*;
import java.rmi.*;

public class HelloApplet extends java.applet.Applet {
    String message = "";

    // The init method begins the execution of the applet on the client
    // machine that is viewing the Web page containing the reference
    // to the applet.
    public void init() {
      try {
          // Looks up the server using the name server on the host that
          // the applet came from.
          Hello obj = (Hello)Naming.lookup(
                          "//" + getCodeBase().getHost() + "/HelloServer");
```

Example: Hello World Client (contd)

```
            // Calls the sayHello method on the remote object.
          message = obj.sayHello();
      } catch (RemoteException e) {
        System.out.println("HelloApplet RemoteException caught");
      }
    }

    public void paint(Graphics g) {
      // The applet will write the string returned by the remote method
      // call on the display.
      g.drawString(message, 25, 50);
    }
}
```


Summary

- Data, computation, job migration
- Client-Server Model
- Mechanism: RPC
  - Most common model for communications in distributed applications.
  - RPC is essentially language support for distributed programming.
  - Relies on a stub compiler to automatically produce client/server stubs from the signatures.
  - RPC is commonly used *even on a single node* for communication between applications running in different address spaces.