Communication Models

1. Remote Procedure Call (RPC)
   - Client/Server application
2. Remote Method Invocation (RMI)
   - Distributed objects
3. Message-Oriented Middleware (MOM)
   - High-level message-queuing
4. Streams
   - Communication of continuous media (having timing constraints)
Layers, interfaces, and protocols in the OSI model.
A typical message as it appears on the network.
Lower-Level Protocols (1)

- **Physical layer:**
  - contains the specification and implementations of bits, and their transmission between sender and receiver.
  - standardizing electrical, mechanical and signaling interfaces
  - Example: RS-232-C

- **Data link layer:**
  - prescribes the transmission of a group of bits, called frames, with error correction and flow control.
Data Link Layer: Example

Discussion between a receiver and a sender in the data link layer.
Lower-Level Protocols (2)

Network layer: describes how packets in a network of computers are to be routed.
- Connectionless IP (Internet Protocol)
- Connection-oriented virtual channel in ATM networks.

Observation: for many distributed systems, the lowest-level interface (of any interest) is that of network layer.
Transport Layer

Important: The transport layer provides the actual communication facilities for most distributed systems.

Responsibility: deliver messages without loss;
- The messages are broken into smaller pieces, and each piece is assigned a sequence number.
- if necessary retransmit messages.

Standard Internet protocols:
- TCP: connection-oriented, reliable, stream-oriented
- UDP: connectionless, unreliable (best-effort) datagram communication; essentially IP with minor additions.
Client-Server TCP

a) Normal operation of TCP
b) Transactional TCP (T/TCP) (for C/S interaction)
**Application Layer**

**Observation:** Many application protocols are directly implemented on top of transport protocols (bypassing session and presentation layers); hence doing a lot of application independent work.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>News</th>
<th>FTP</th>
<th>WWW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protocol</td>
<td>NNTP</td>
<td>FTP</td>
<td>HTTP</td>
</tr>
<tr>
<td>Encoding</td>
<td>7-bit + MIME</td>
<td>7-bit text + 8-bit binary</td>
<td>8-bit + content type</td>
</tr>
<tr>
<td>Naming</td>
<td>Newsgroup</td>
<td>Host + path</td>
<td>URL</td>
</tr>
<tr>
<td>Distribution</td>
<td>Pull</td>
<td>Pull</td>
<td>Pull</td>
</tr>
<tr>
<td>Replication</td>
<td>Flooding</td>
<td>Caching + DNS tricks</td>
<td>Caching + DNS tricks</td>
</tr>
<tr>
<td>Security</td>
<td>None (PGP)</td>
<td>Username + password</td>
<td>Username + password</td>
</tr>
</tbody>
</table>
Middleware Layer

Middleware is invented to provide *common* services and protocols that can be used by many different applications:

- A rich set of communication protocols
- Marshalling and unmarshalling data
- Naming protocols, so that different applications can easily share resources
- Security protocols
- Scaling mechanisms (support for replication & caching)
- Commit protocols
- Locking protocol
- Reliable multicasting
- Many others
Middleware Protocols

An adapted reference model for networked communication.

1. Physical protocol
2. Data link protocol
3. Network protocol
4. Transport protocol
5. Middleware protocol
6. Application protocol

Application independent
Remote Procedure Call (RPC)

The procedures such as send and receive do not conceal the communication. More sophisticated is allowing programs to call procedures located on other machines.

- Basic RPC operation
- Parameter passing
- Variations
Conventional Procedure Call

```
int   read (int fd, char *buf, int bytes);
```

a) Parameter passing in a local procedure call: the stack before the call to read

b) The stack while the called procedure is active

**Diagram:**

- **Main program’s local variables**
- **Call-by-value**
- **Call-by-reference**
- **Call-by-copy/restore**
Local Procedure Call Principles

1. Push parameter values of the procedure on stack
2. Call procedure
3. Use stack for local variables
4. Pop results (parameters)

**Principle:** “communication” with local procedure is handled by copying data to/from the stack.
Basic RPC Operation

**Goal:** to make a remote procedure call look as much as possible like a local one

**Observations:**
- Application developers are familiar with simple local procedure call model
- Well-engineered procedures operate in isolation (black box)
- There is no fundamental reason not to execute procedures on a remote machine

**Conclusion:** communication between caller & callee can be hidden by using procedure-call mechanism
Client and Server Model

Principle of RPC between a client and server program.
Key Components in RPC

Client stub:
A piece of code that transforms the procedure call to a request message that is to be delivered to the receiver by the underlying network services
- It blocks until the reply comes back

Server stub:
A piece of code that transforms requests coming in over the network into local procedure calls
- The server stub is blocked waiting for incoming messages.
Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client
Steps involved in doing remote computation through RPC

20
Parameter marshalling: Packing parameters into a message. However, there is more than just wrapping parameters into a message:

- Client and server machines may have different data representations (think of byte ordering)
- Wrapping a parameter means transforming it into a sequence of bytes
- Client and server have to agree on the same encoding:
  - How are basic data values represented (integers, floats, characters)
  - How are complex data values represented (arrays, unions)
- Client and server need to properly interpret messages, transforming them into machine-dependent representations
Passing Value Parameters

Two parameters to the RPC:

- `int 5;`
- `string "VELI";`

a) Pentium

b) on SPARC after receipt

c) after inversion
Something to Think About

How to pass pointers?
RPC Protocol

- RPC protocol governs two things:
  1. The format of the parameters and return values
  2. The actual communication protocol the caller & the callee agree
     - For example, TCP/IP or UDP

- Once the protocol is defined, clients and server stubs needs to be implemented.
Interface

- An interface consists of a collection of procedures that can be called by a client, and which are implemented by a server.
- An interface is generally available in the same programming language as the one in which the client or server is written (not necessarily though).
- It, often, looks like a ANSI function declaration
- Interfaces are often specified by means of an Interface Definition Language (IDL).
- An interface specified in an IDL, is then subsequently compiled into a client and server stub (e.g. rmic generates client stub given an interface in Java RMI)
Variations: Asynchronous RPC

**Essence:** client needs not to block when there is nothing to return.

*a* traditional RPC

*b* asynchronous RPC
Asynchronous RPC with Return Value

Deferred synchronous RPC: A client and server interacting through two asynchronous RPCs

One-way RPC: Client does not even wait for an acknowledgement of server’s acceptance of the request. Naturally, unreliable.
RPC in practice

- **Goal:** Let the developer concentrate on only the client- and server-specific code; let the RPC system (generators and libraries) do the rest.

- **Distributed Computing Environment (DCE) is developed by OSF (The Open Group).**

- DCE is a true middleware system.

- It uses client-server model.

- **DCE Services:**
  - Distributed file service
  - Directory service (to keep track of resources)
  - Security service (for protection of resources)
  - Distributed time service (for global synchronization of system clocks)
Reading Assignments

1. Read the pages 77-88 of the textbook for Doors

2. Read the pages 80-85 of the textbook for DCE RPC
Remote Method Invocation

- Distributed objects
- Remote method invocation
- Parameter passing
Distributed Objects (1)

- Data (state) and operations are encapsulated in an object.
- Operations on object data are implemented as methods, and are accessible through interfaces.
- Object offers only interfaces to its clients.
- Object server is responsible for a collection of objects.
- Client stub (proxy) implements interface on the client side.
- Server stub (skeleton) handles un/marshalling and method invocation (probably returning results).
- The state of the objects are not distributed → remote objects.
Common organization of a remote object with client-side proxy
Object References

Systemwide object reference: Having an object reference allows a client to bind to a remote object:

- Binding is the resolving the object reference to an actual object
- Distributed objects support systemwide object references
- Object references can be freely passed between processes, for example, as parameter to method invocations.
- Reference denotes a server, object, and communication protocol
- Binding results in a proxy being placed in the process’s address space
- Client loads associated stub code
- Stub is instantiated and initialized for specific object
Client-to-Object Binding: Implicit

- **Implicit binding**: invoke methods directly on the referenced object

```c
distr_object * obj_ref;  // Declare a systemwide object reference

obj_ref = ...;           // Initialize the reference to a distributed object

obj_ref->do_something(); // Implicitly bind and invoke a method
```
**Client-to-Object Binding: Explicit**

```c
distr_object* obj_ref; // Declare a systemwide object reference
local_object* obj_ptr; // Declare a pointer to local objects
obj_ref = ...; // Initialize the reference to a distributed object

obj_ptr = bind(obj_ref); // Explicitly bind and obtain a pointer to the local proxy

obj_ptr->do_something(); // Invoke a method on the local proxy
```

- **Explicit binding:** Client must first explicitly bind to object before invoking its method
Implementing Object References (1)

- Object reference must contain sufficient information for a client to bind to an object.
- A simple object reference would include \textit{network address of the machine and endpoint (port)} + \textit{which object}.
- Not good if the machine crashes or the object relocates.
- A local daemon per machine listening to a well-known port and that keeps track of server-to-endpoint assignments
  - only good when the endpoint is changed
  - server ID in object reference is necessary
Implementing Object References (2)

- **Solution:**
  - A location server keeps track of the machine where an object’s server is currently running
    - Object reference contains the network *address of the location server* + *systemwide identifier for the server*
Implementing Object References (3)

- Object reference may even contain protocol information
  - TCP or UDP,
  - protocols for marshalling parameters.
- Reference may contain an URL pointing to an implementation file (implementation handle)
  - May refer to a complete implementation of a proxy that the client can dynamically load
  - We need only a standard protocol for dynamically loading, installing and subsequently instantiating code.
  - Flexible
Remote Method Invocation

**Static RMI:** Uses a predefined interface definitions (Java)
- Interfaces of an object are known when the client application is being developed.
- If interfaces change, the client application must be recompiled
- Object-specific stubs are used.

**Dynamic RMI:** Application selects at runtime which method it will invoke at a remote object
- Interfaces can be inspected at runtime, method invocation can be dynamically constructed.
- Application must be developed to support any possible interface.
- Dynamic invocation look like

```java
invoke (object, method, input_parameters, output_parameters);
```
Dynamic vs. Static RMI

Example:

**Static Invocation:**

```
fobject.append(int i)
```

**Dynamic Invocation:**

```
invoke(fobject, id(append), int i)
```
Parameter Passing (1)

Object as a parameter: Two cases:

- **Object-by-reference**: when it refers to a remote object, an object reference is passed when invoking a remote method.

- **Object-by-value**: reference refers to a local object
  - It is in the address space of the client
  - The state and the methods of the object are to be marshaled
  - Server unmarshalls the object; creating a copy of the original object

This distinction may lead to some problems due to the transparency

- In Java, reference to a local object and reference to a remote object are of different data types. But similarly treated.
The situation when passing an object by reference or by value.
Java RMI: Object model

- Distributed objects are integral part of the language
- Distributed object is defined as remote object, whose state resides on a single remote machine
- Interfaces are implemented by means of a proxy
- A proxy appear as a local object in the client’s address space
- Remote objects cannot be cloned (neither their proxies at each client) (can be cloned only by the server)
- Each object can be constructed as a monitor, by declaring a method to be synchronized
  - Access to object’s internal data is completely serialized.
  - protecting remote objects through synchronized methods is generally not possible.
Java RMI: Remote Object Invocation

- Any primitive or object type can be passed as a parameter to an RMI, provided that the type can be marshaled.
- In Java terminology, objects must be **serializable**.
- Platform dependent objects (e.g. file descriptors, sockets) are not serializable.
- Local objects are passed by value, remote objects by reference.
- Object reference contains:
  - network address,
  - endpoint of the server
  - local identifier for the actual object in the server's address space
  - protocol stack for communication
Components of Remote Object

- Each object in Java is an instance of a class which contains an implementation of one or more interfaces
- **Server-class**: implementation of server-side code
  - Contains implementation of the object (description of the object’s state, implementation of the methods on the state)
  - Server side stub, *skeleton*, is generated from the interface specification of the object (obsolete now)
- **Client-class**: implementation of client-side code
  - Contains the client-side stub, called *proxy*, generated from the interface specification of the remote object
  - Remote object reference (i.e. network address of the server, the port number etc.) is always stored as part of the state of a proxy.
  - Proxies are serializable since they are, in essence, local objects
Marshalling a Proxy

- A proxy can be used as a reference to a remote object
- Marshalling a proxy may be inefficient since it can lead to very large references
- In Java, when marshalling a proxy, an implementation handle is generated
- Implementation handle specifies precisely which classes are needed to construct the proxy
- Some of these classes may first need to be downloaded from a remote site
- Since Java code is completely portable passing proxies is possible
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}

import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl
    extends UnicastRemoteObject implements Hello {

    public HelloImpl() throws RemoteException {
        super();
    }

    public String sayHello() {
        return "Hello World!";
    }
}
public static void main(String args[]) {
    // Create and install a security manager
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new RMISecurityManager());
    }
    try {
        HelloImpl obj = new HelloImpl();
        // Bind this object instance to the name
        // "HelloServer"
        Naming.rebind("//localhost/HelloServer", obj);

        System.out.println("HelloServer bound in registry");
    } catch (Exception e) {
        System.out.println("HelloImpl err: " + e.getMessage());
        e.printStackTrace();
    }
}
---- HelloImpl_stub.java File ------

// Stub class generated by rmic, do not edit.
// Contents subject to change without notice.
public final class HelloImpl_Stub
    extends java.rmi.server.RemoteStub
    implements hello.Hello, java.rmi.Remote
{
    private static final long serialVersionUID = 2;
    private static java.lang.reflect.Method $method_sayHello_0;
    static {
        try {
            $method_sayHello_0 =
                hello.Hello.class.getMethod("sayHello", new java.lang.Class[] { });
        } catch (java.lang.NoSuchMethodException e) {
            throw new java.lang.NoSuchMethodError(
                "stub class initialization failed");
        }
    }
}
```java
// constructors
public HelloImpl_Stub(java.rmi.server.RemoteRef ref) {
    super(ref);
}

// methods from remote interfaces implementation of sayHello()
public java.lang.String sayHello() throws java.rmi.RemoteException {
    try {
        Object $result = ref.invoke(this, $method_sayHello_0, null, 6043973830760146143L);
        return ((java.lang.String) $result);
    } catch (java.lang.RuntimeException e) {
        throw e;
    } catch (java.rmi.RemoteException e) {
        throw e;
    } catch (java.lang.Exception e) {
        throw new java.rmi.UnexpectedException("undeclared checked exception", e);
    }
}
```
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;
import java.util.Date;

public class HelloClient {

    public static void main(String[] argv) {
        String serverName = "";
        String message = "blank";

        Hello obj = null;
    }
}
System.setSecurityManager(new RMISecurityManager());
    if (argv.length != 1) {
        try {
            serverName = java.net.InetAddress.getLocalHost().getHostName();
        } catch (Exception e) {
            e.printStackTrace();
        }
    } else {
        serverName = argv[0];
    }
    if (serverName == "") {
        System.out.println("usage: java SimpleRMIClient <IP address of host running RMI server>");
        System.exit(0);
    }
try {
    // explicitly bind server object to object in client
    obj = (Hello) Naming.lookup("//" + serverName + "/HelloServer");
    // Print success message
    System.out.println("RMI connection successful");
    // Call method on server and put result into String message
    message = obj.sayHello();
    // Print message on server
    System.out.println("Message on server is " + message);
}

catch(Exception e) {
    System.out.println("Exception occurred: " + e);
    System.exit(0);
}
Java RMI: Example (9)

Server side:
- HelloImpl.java
- Hello.java (interface)
- HelloImpl_stub.java (needed for registry)
- Skeleton is deprecated in version 1.2

Client side:
- HelloClient.java (client object invoking remote method)
- HelloImpl_stub.java (proxy) - can be downloaded remotely
- Hello.java (interface)