# Client / Server Programming

COMP476
Networked Computer Systems

## Client-Server Paradigm

- *Server* application is "listener"
  - Waits for incoming message
  - Performs service
  - Returns results
- *Client* application establishes connection
  - Sends message to server
  - Waits for return message

## Characteristics of Client

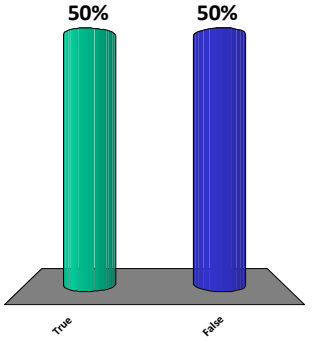
- Arbitrary application program
  - Becomes client when network service is needed
  - Also performs other computations
- Invoked directly by user
- Runs locally on user's computer
- Initiates contact with server
- Can access multiple services

## Characteristics of Server

- Special purpose application dedicated to providing network service
- Starts at system initialization time
- Runs on a remote computer (usually centralized, shared computer)
- Waits for service requests from clients; loops to wait for next request
- Will accept requests from arbitrary clients; provides one service to each client

## Web servers need to be implemented on large expensive server machines.

1. True
2. False




## Two Basic Communication Paradigms

- The Internet supports two basic communication paradigms:
  - Stream Transport in the Internet
  - Message Transport in the Internet

| Stream Paradigm | Message Paradigm |
|---|---|
| Connection-oriented | Connectionless |
| 1-to-1 communication | Many-to-many communication |
| Sequence of individual bytes | Sequence of individual messages |
| Arbitrary length transfer | Each message limited to 64 Kbytes |
| Used by most applications | Used for multimedia applications |
| Built on TCP protocol | Built on UDP protocol |

**Figure 3.1** The two paradigms that Internet applications use.

© 2009 Pearson Education Inc.

## Stream Transport in the Internet

- Stream denotes a paradigm in which a sequence of bytes flows from one application program to another
- Internet's mechanism arranges two streams, one in each direction
- The stream mechanism transfers a sequence of bytes without attaching meaning to the bytes and without inserting boundaries
- A sending application can choose to generate one byte at a time, or can generate blocks of bytes
- The network chooses the number of bytes to deliver at any time
  - the network can choose to combine smaller blocks into one large block or can divide a large block into smaller blocks

© 2009 Pearson Education Inc.

## Message Transport in the Internet

- In a message paradigm, the network accepts and delivers messages
- Each message delivered to a receiver corresponds to a message that was transmitted by a sender
  - the network never delivers part of a message, nor does it join multiple messages together
  - if a sender places exactly *n* bytes in an outgoing message, the receiver will find exactly *n* bytes in the incoming message

© 2009 Pearson Education Inc.

## Message Transport in the Internet

- Message service does not make any guarantees
- So messages may be
  - Lost (i.e., never delivered)
  - Duplicated (more than one copy arrives)
  - Delivered out-of-order
- A programmer who uses the message paradigm must insure that the application operates correctly
  - even if packets are lost or reordered
- Most applications require delivery guarantees. Programmers tend to use the stream service except in special situations

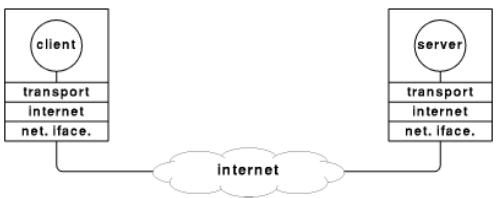
© 2009 Pearson Education Inc.

## Message Exchanges

- Typically, client and server exchange messages:
  - Client sends request, perhaps with data
  - Server send response, perhaps with data
- Client may send multiple requests; server sends multiple responses
- Server may send multiple response - consider streaming audio

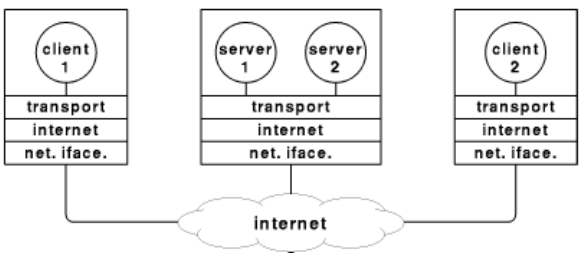
## Transport Protocols and Client-Server Paradigm

- Clients and servers exchange messages through transport protocols; e.g., TCP or UDP
- Both client and server must have same protocol stack and both interact with transport




## Multiple Services on One Computer

- Servers run as independent processes and can manage clients simultaneously



## Multiple Services on One Computer

- Can reduce costs by sharing resources among multiple services
- Reduces management overhead - only one computer to maintain
- One server can affect others by exhausting computer resources
- Failure of single computer can bring down multiple servers

## Server Identification

- How does a client identify a server?
- The Internet protocols divide identification into two pieces:
  - An identifier for the computer on which a server runs
  - An identifier for a service on the computer
- Identifying a computer?
  - Each computer is assigned a unique 32-bit identifier known as an Internet Protocol address (IP address)
  - To make server identification easy for humans, each computer is also assigned a name
  - Thus, a user specifies a name such as *www.ncat.edu* rather than an integer address

© 2009 Pearson Education Inc.

## Service Identification

- Each service available in the Internet is assigned a unique 16-bit identifier known as a protocol port number (or port number)
  - Examples, email → port number 25, and the web → port number 80
- When a server begins execution
  - it registers with its local OS by specifying the port number for its service
- When a client contacts a remote server to request service
  - the request contains a port number

© 2009 Pearson Education Inc.

## Internet Port Numbers

- Applications are identified by a 16 bit integer number known as a port number.
- Internet ports do NOT refer to plugs in the back of the machine.
- The full address of an application is

  InternetName:port
- Applications bind to a port number to receive data sent to that port.

## Well Known Ports

- Port numbers under 2K are reserved for specific "well known" application servers
  - 21 ftp
  - 23 telnet
  - 79 finger
  - 80 HTTP web servers
  - 443 HTTPS secure web servers
  - 17 Quote of the Day

## *Lesser Known* Ports

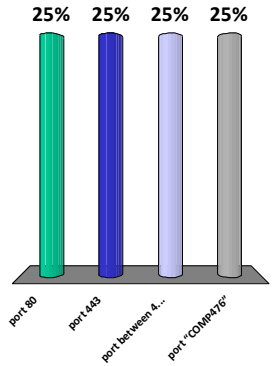
- Well Known Ports are only used by servers.
- Servers for non-standard applications use higher numbered ports.
- Applications accessing a server use a higher numbered port.
- When a program connects to a remote system, it is automatically assigned a port.

## Binding of Server Location

- How and when does a client application learn the location of a service?
- Goals:
  - portability - allow the application to be used on different systems
  - load balancing - select server with lowest utilization
  - failure recovery - select a different server if original fails
  - efficiency – avoid many messages or broadcasts

## When writing your class program, it should bind to

1. port 80
2. port 443
3. port between 4095 and 65535
4. port "COMP476"



## Finding a Service

- Write server name in code
- Read a file of server addresses
- Broadcast request for a server
- Ask a human
- Name server

## Client-Server Interactions

- Clients can access multiple services
- Clients may access different servers for one service
- Servers may become clients of other servers

## Client-Server Summary

- *Client-server* paradigm used in almost every distributed computation
  - *Client* requests service when needed
  - *Server* waits for client requests
- Clients and servers use *transport protocols* to communicate
- Often, but not always, there is an *application protocol*

## `java.net` package

- Provides the classes for implementing networking applications
- Sending and receiving data across a network is similar to writing and reading from a file
- The java.net package can be roughly divided into high level and low level Application Programmer Interfaces (API)

## java.net Low Level API

- The Low Level API deals with the following abstractions:
  - *Addresses*, which are networking identifiers, like IP addresses.
  - *Sockets*, which are basic bidirectional data communication mechanisms.
  - *Interfaces*, which describe network interfaces.

## java.net High Level API

- The High Level API deals with the following abstractions:
  - *URIs*, which represent Universal Resource Identifiers.
  - *URLs*, which represent Universal Resource Locators.
  - *Connections*, which represents connections to the resource pointed to by *URLs*.

## Java Socket Classes

- Sockets are means to establish a communication link between machines over the network.
- **Socket** is a TCP client API, and will typically be used to connect to a remote host.
- **ServerSocket** is a TCP server API, and will typically accept connections from client sockets.
- **DatagramSocket** is a UDP endpoint API and is used to send and receive datagram packets.

## Class java.net.Socket

- This class implements TCP client sockets (also called just "sockets"). A socket is an endpoint for communication between two machines.
- Sockets create streams that can be used exactly like file streams.
- Reading and writing to a socket is identical to reading and writing to a file.

## Socket Constructor

```
public Socket(String host, int
  port)
         throws
  UnknownHostException,
  IOException
```

- Creates a stream socket and connects it to the specified port number on the named host.
- Other constructors are available

## Methods

```
public InputStream getInputStream(
             throws IOException
```

- Returns an input stream for this socket.

```
public OutputStream getOutputStream()
             throws IOException
```

- Returns an output stream for this socket.

## java.net.ServerSocket

- Creates a TCP socket for use by a server program.
- A server socket waits for requests to come in over the network.

## ServerSocket Constructor

```
public ServerSocket(int port)
             throws IOException
```

- Creates a server socket on a specified port.
- A port of 0 creates a socket on any free port.

## ServerSocket Methods

```
public Socket accept()
             throws IOException
```

- Listens for a connection to be made to this socket and accepts it.
- The method blocks until a connection is made.
- Returns a new Socket for communicating with the client.

## Accepting Connections

- When a client connects to a ServerSocket, a new Socket is created on the server
- This new Socket is used to communicate with the client
- Typically a new thread on the server would communicate with the client
- The original ServerSocket can do another accept to wait for another client

## UDP Sockets

- **DatagramSocket** is a class to create a Java socket that uses UDP.
- **DatagramSocket** objects send and receive objects of the **DatagramPacket** class.
- A **DatagramPacket** object contains the data transmitted along with the address of the sender or destination.

## Java Security

- Java applets running in a browser can only connect to the server that hosts the applet.
- Java applications generally have no restrictions.
- Experience shows that network programming is easiest on a PC with Java.

## Sample Java Client

```
public class Tclient {
   final static String IPname = "whatever.ncat.edu";
   final static int      serverPort   = 4567;
   public static void main(String[] args) {
      java.net.Socket                sock = null;
      java.io.PrintWriter             pw   = null;

      java.io.BufferedReader        br   = null;
```

```
try {
      sock = new java.net.Socket(IPname,serverPort);
      pw   = new java.io.PrintWriter(sock.getOutputStream(),true);
      br   = new java.io.BufferedReader(new
               java.io.InputStreamReader(sock.getInputStream()));
      pw.println("Message from the client");
      String answer = br.readLine();
      System.out.println("Response from the server >" + answer);
      pw.close();
      br.close();
      sock.close();
} catch (Exception e) {
```

## Sample Java Server

```
public class Tserver {
   final static int serverPort   = 4567;
   public static void main(String[] args) {
      java.net.ServerSocket   sock = null;
      java.net.Socket            clientSocket = null;
      java.io.PrintWriter       pw   = null;
      java.io.BufferedReader br   = null;
```

```
try {
   sock = new java.net.ServerSocket(serverPort);
   clientSocket = sock.accept();
   pw   = new java.io.PrintWriter(
                        clientSocket.getOutputStream());
   br   = new java.io.BufferedReader(new
        java.io.InputStreamReader( clientSocket.getInputStream()));
   String msg = br.readLine();
   System.out.println("Message from the client >" + msg);
   pw.println("Got it!");
   pw.flush();
   pw.close();
   br.close();
   clientSocket.close();
   sock.close();
} catch (Throwable e) {
```

# Read a URL with High Level API

```
try {
   /* Create a URL and get an InputStream for it. */
   java.net.URL webFile = new
java.net.URL("http://www.acme.com/myPage.html");
   java.io.InputStream inFile = webFile.openStream();
   /* Wrap the InputStream in a BufferedReader */
   java.io.InputStreamReader inRdr = new java.io.InputStreamReader(inFile);
   java.io.BufferedReader bufRead = new java.io.BufferedReader(inRdr);
   /* Read and print all lines of the file. */
   String html = bufRead.readLine();
   while (html != null) {
       System.out.println(html);
       html = bufRead.readLine();
   }
   bufRead.close();
} catch (Exception e) { // catch any error and print details
   System.out.println(e.getMessage());
}
```