# *SYSC 3303 Real-Time Concurrent Systems*

## Introduction to Internet Protocols



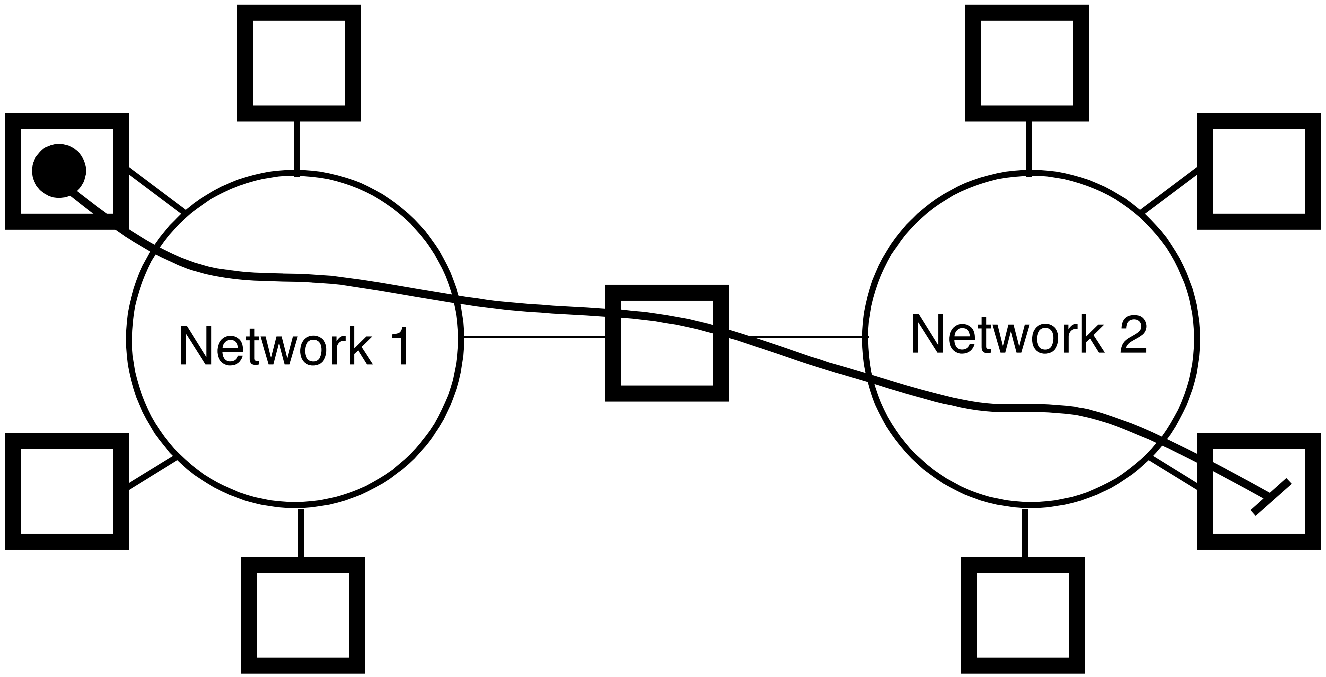
- Reference: Internetworking with TCP/IP, Douglas Comer, Prentice Hall
- revised January 9th, 2016

## *What's In This Set of Slides?*

- An overview of the principles of computer networking, focussing primarily on the Internet UDP/IP protocol
- An introduction to the Java classes that support internetworking via UDP/IP
- The principles of computer communications networks are covered in a much more comprehensive fashion in courses such as SYSC 3502, SYSC 4602, and COMP 3203, so don't expect to become an expert through this course!
- Our goal is to provide just enough background so that we can study the design of computer communications applications as examples of real-time (distributed, concurrent, event-driven) programs

# *Internetworking*

- How can we arrange for communication between different hardware/software platforms which are connected to interconnected networks?



# *Internetworking*

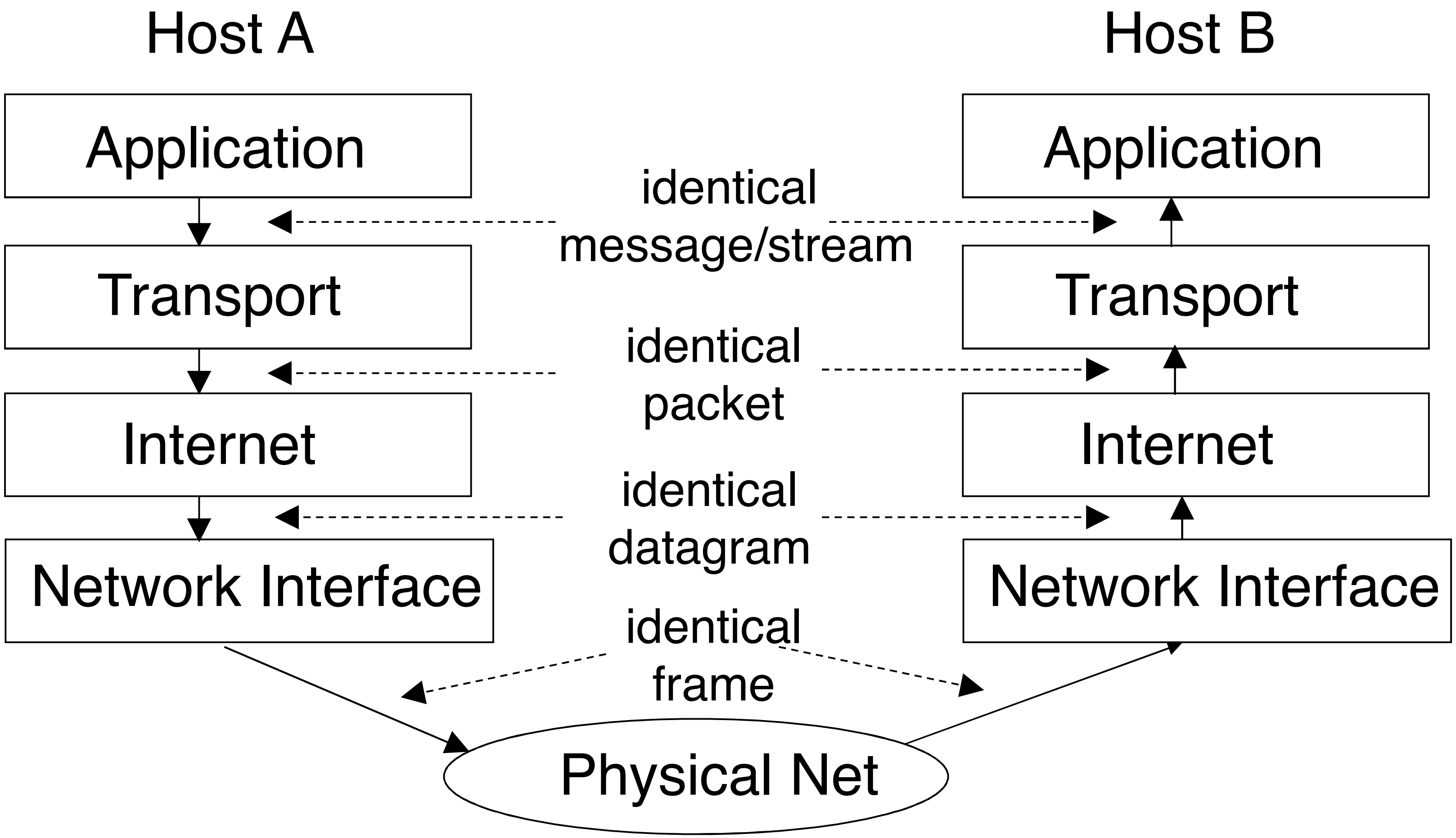
- Research starting in about the mid 1970's led to the development of internetworking architectures, protocol layering models, datagram and stream transport services, and the client-server interaction paradigm
- This set of slides will provide an overview of the internetworking technology commonly known as TCP/IP (after the names of its two main standards), although we'll consider only one of its standard protocols: UDP/IP

## *Conceptual Layers of TCP/IP Protocol Software*

| Conceptual Layer | Information Passed Between Layers |
|---|---|
| Application | |
| | Messages or Streams |
| Transport | |
| | Transport Protocol Packets |
| Internet | |
| | IP Datagrams |
| Network Interface | |
| | Network-Specific Frames |
| Network Hardware | |

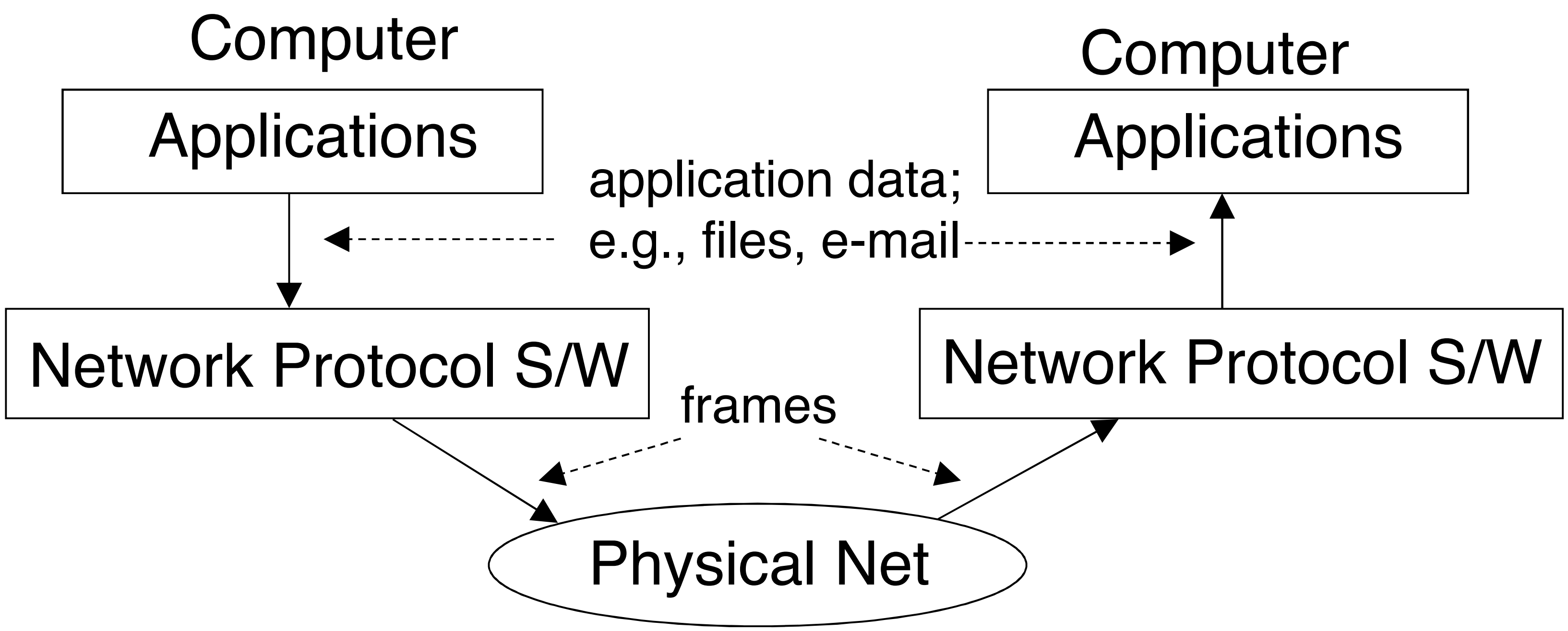
This is sometimes called a *protocol stack*

# *Protocol Layering Principle*



# *Physical Network Technologies*

- Hardware-specific frames travel along a physical network



# ***Example: Ethernet Frame Format***

| Preamble | Destination Address | Source Address | Frame Type | Data | CRC |
|---|---|---|---|---|---|
| 64 bits | 48 bits | 48 bits | 16 bits | 368 -12000 bits | 32 bits |

Frame Header

- The Ethernet interface hardware in each computer is assigned a unique Ethernet address (physical address).
- An address in an Ethernet frame specifies a physical address, a network broadcast address, or a multicast address.

# *Ethernet Frame Format (Details)*

| Preamble | Destination Address | Source Address | Frame Type | Data | CRC |
|---|---|---|---|---|---|
| 64 bits | 48 bits | 48 bits | 16 bits | 368 -12000 bits | 32 bits |

**http://www.inetdaemon.com/tutorials/lan/ethernet/frame_format.html**

| *7 bytes* | *1 byte* | *2 or 6 bytes* | *2 or 6 bytes* | *2 bytes* | *4-1500 bytes* | | | | *4 bytes* |
|---|---|---|---|---|---|---|---|---|---|
| | Start | Dest. | Source | | (Data / Pad) | | | | |
| Preamble | Frame Delimiter | MAC | MAC | Length | DSAP | SSAP | CTRL | NLI | FCS |

**MAC = media access control address**
**FCS = frame check sequence**

**Preamble**

**This is a stream of bits used to allow the transmitter and receiver to synchronize their communication. The preamble is an alternating pattern of 56 ones and zeroes. It is immediately followed by the Start Frame Delimiter.**

**Start Frame Delimiter**

**This is always 10101011 and is used to indicate the beginning of the frame-information.**

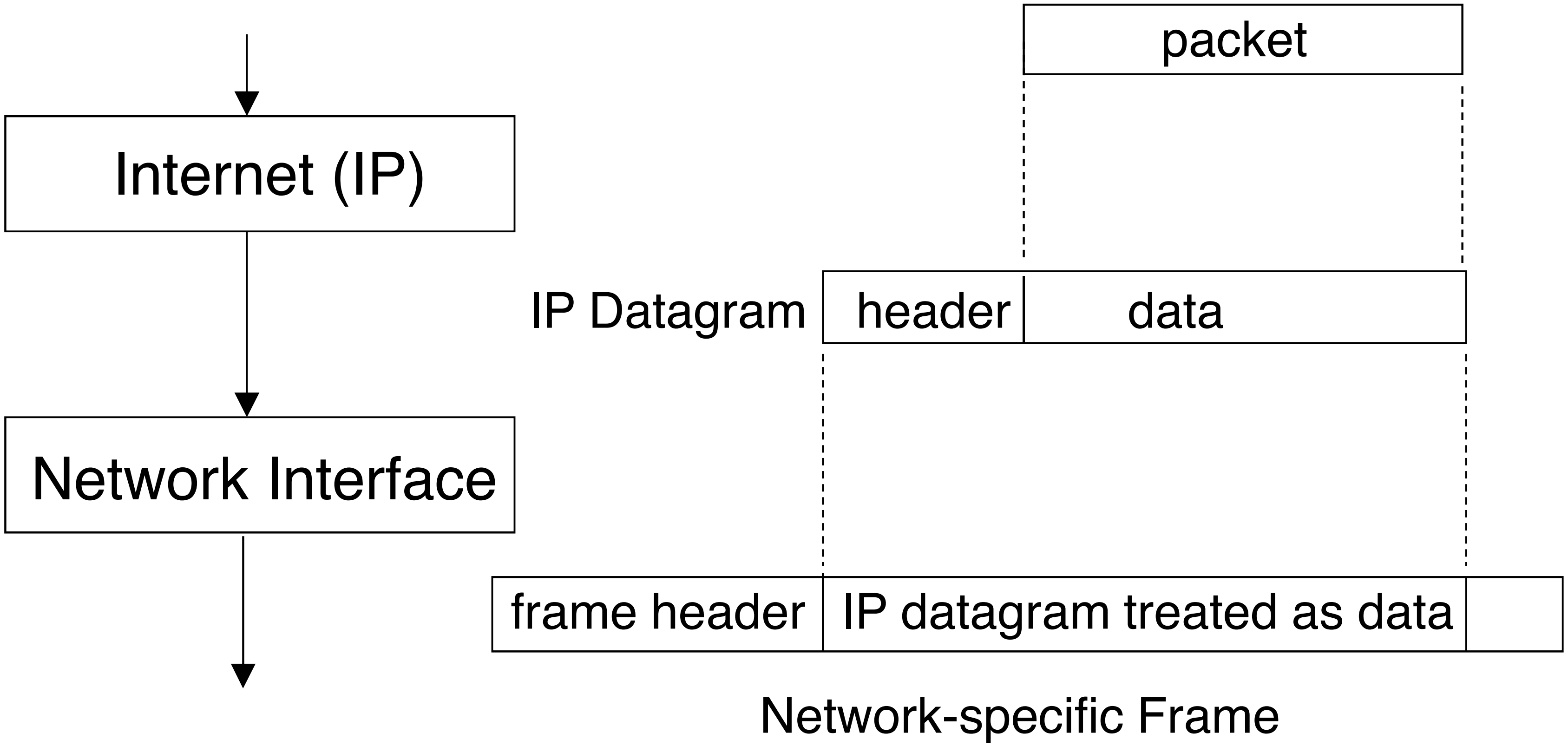
## *Issues*

- Different networks may use different physical network technologies, yet this should be transparent to communicating applications running on a pair of computers
- We need to abstract away from physical network addresses and the details of the frames transmitted on physical networks

# *Internet Addresses*

- Each host on the Internet has a unique Internet address
- Each Internet address specifies a network *connection*
  - it encodes both a network id *and* the id of the host computer on that network
  - dotted-quad format; e.g., 128.10.2.30

# *Internet Protocol (IP) Layer*



# *IP: A Packet Delivery Service*

- The IP layer receives packets of data from the upper layers in the protocol stack, encapsulates each packet in an *IP datagram* , then passes the datagrams to the Network Interface layer for transmission over the network
- IP datagram headers contain the IP addresses of the source and destination hosts (plus other information)
- The Network Interface layer maps IP addresses to physical network addresses
- The IP layer also receives IP datagrams from the Network Interface layer, extracts the packets, and passes the packets to the upper layers

## *IP: A Packet Delivery Service*

- IP is connectionless
  - each packet is treated independently of others
  - a sequence of packets may travel over different paths between a source and a destination
- IP is unreliable
  - delivery of packets is not guaranteed
  - packets may be lost, duplicated, or delivered out of order
  - sender and receiver are not informed of these conditions

# ***Protocol Ports***

- IP addresses specify hosts, not processes running on those hosts
- How do we specify which process at a destination host is to receive a datagram sent by a source host?
- How do we specify which process at a source host sent a datagram received by a destination host?
- Answer: each computer maintains a set of *protocol ports*

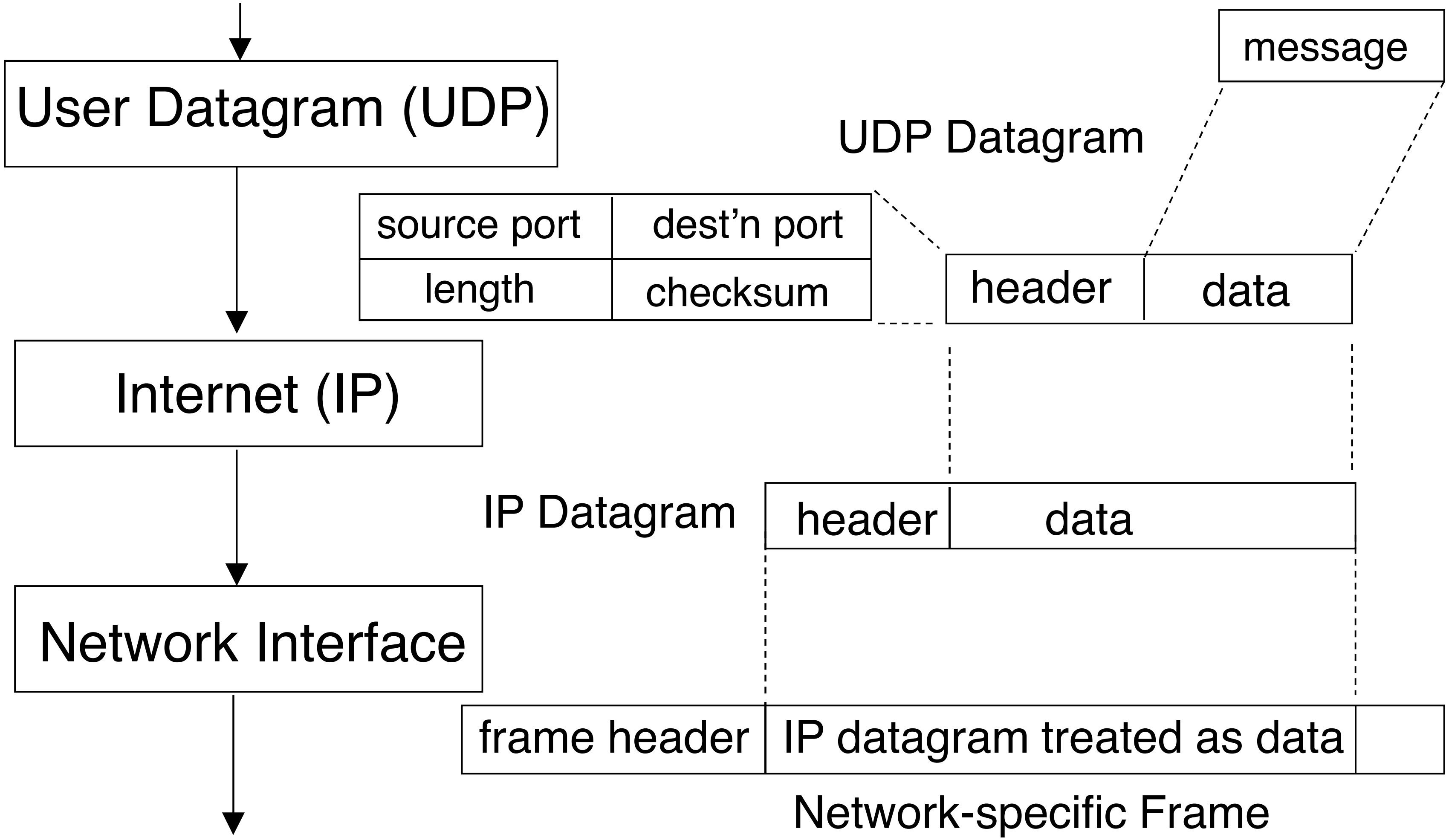
## *Protocol Ports*

- Sending process must specify the
  - Internet address of destination host
  - protocol port # associated with the destination process within that host
  - protocol port # associated with the source process to which replies should be addressed

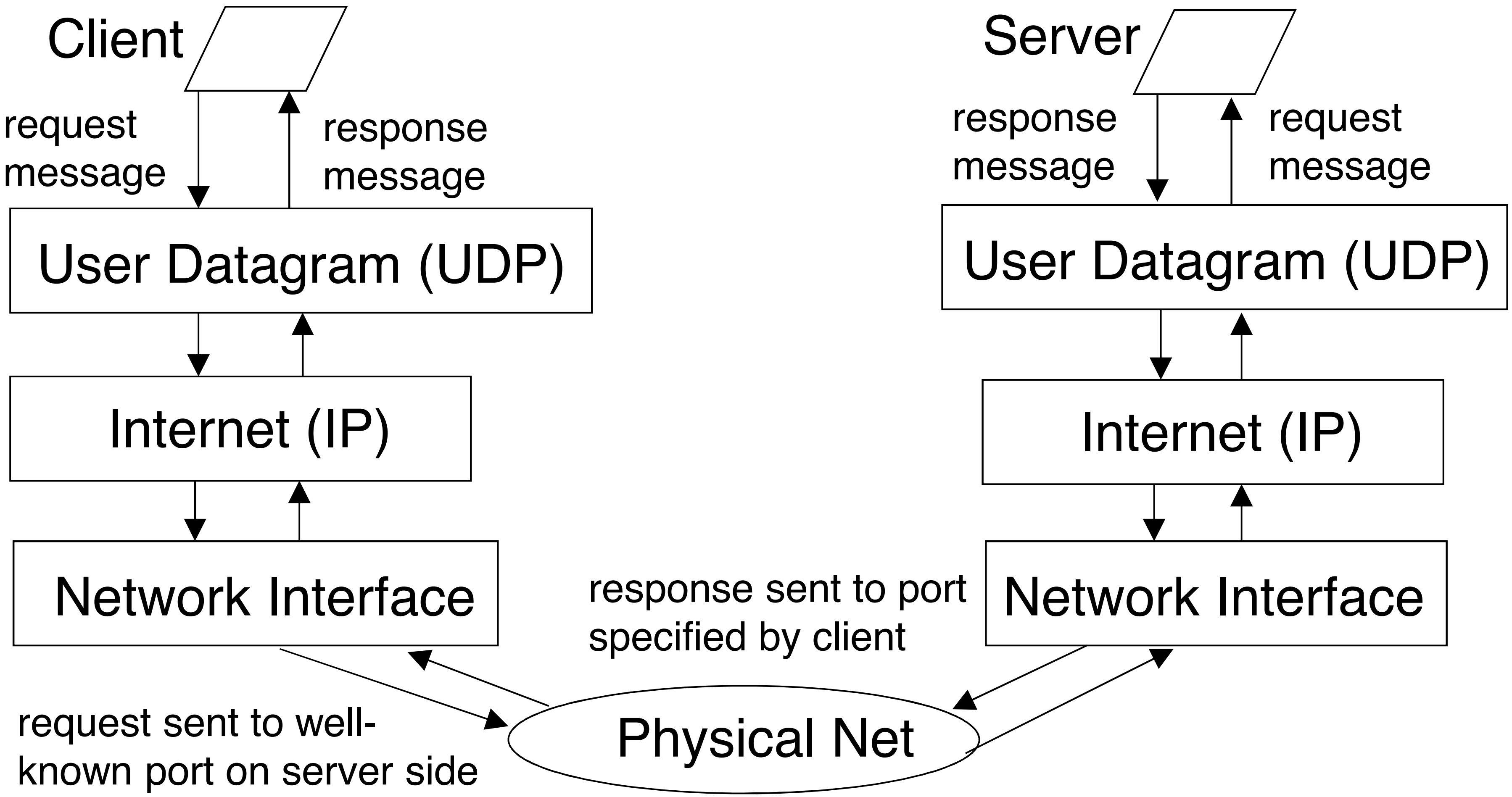
# *User Datagram Protocol (UDP)*

- UDP provides unreliable connectionless delivery service using IP to transport messages among machines
- It adds the ability to distinguish multiple destinations within a computer, via ports
- The UDP layer is an example of a *Transport layer*
- Its Transport Protocol Packets are called *UDP Datagrams*

# User Datagram Protocol (UDP) Layer



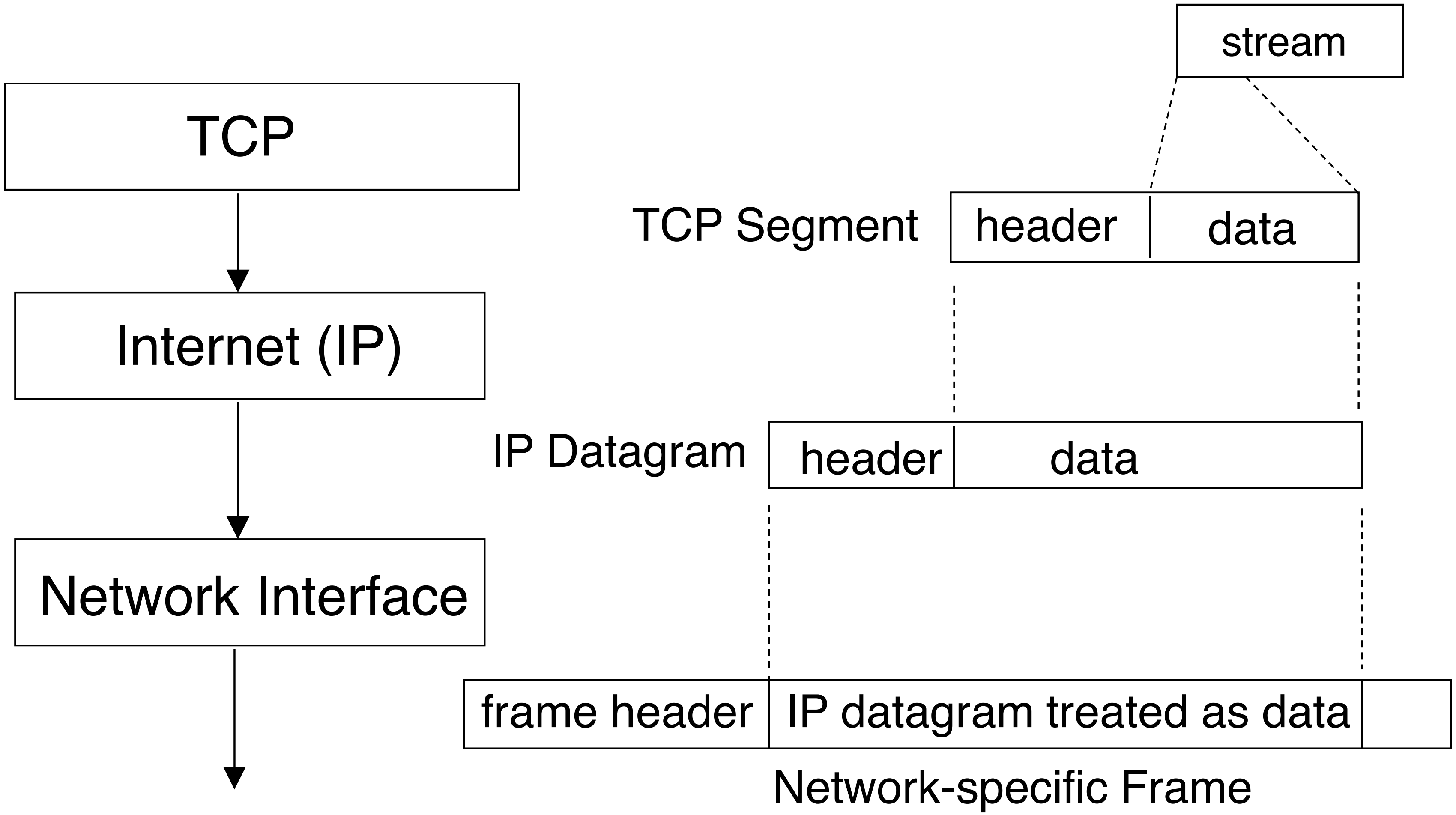
# *Applications: Simple Client-Server Model*



## *Transmission Control Protocol (TCP)*

- The TCP layer is another example of a Transport layer
- TCP provides reliable stream delivery between hosts
- TCP establishes a virtual circuit connection between the source and destination before data transfer begins
- TCP Transport Protocol Packets are called *TCP Segments*

# Transmission Control Protocol (TCP) Layer



# *4.3 BSD Unix Interface to Internet Protocols*

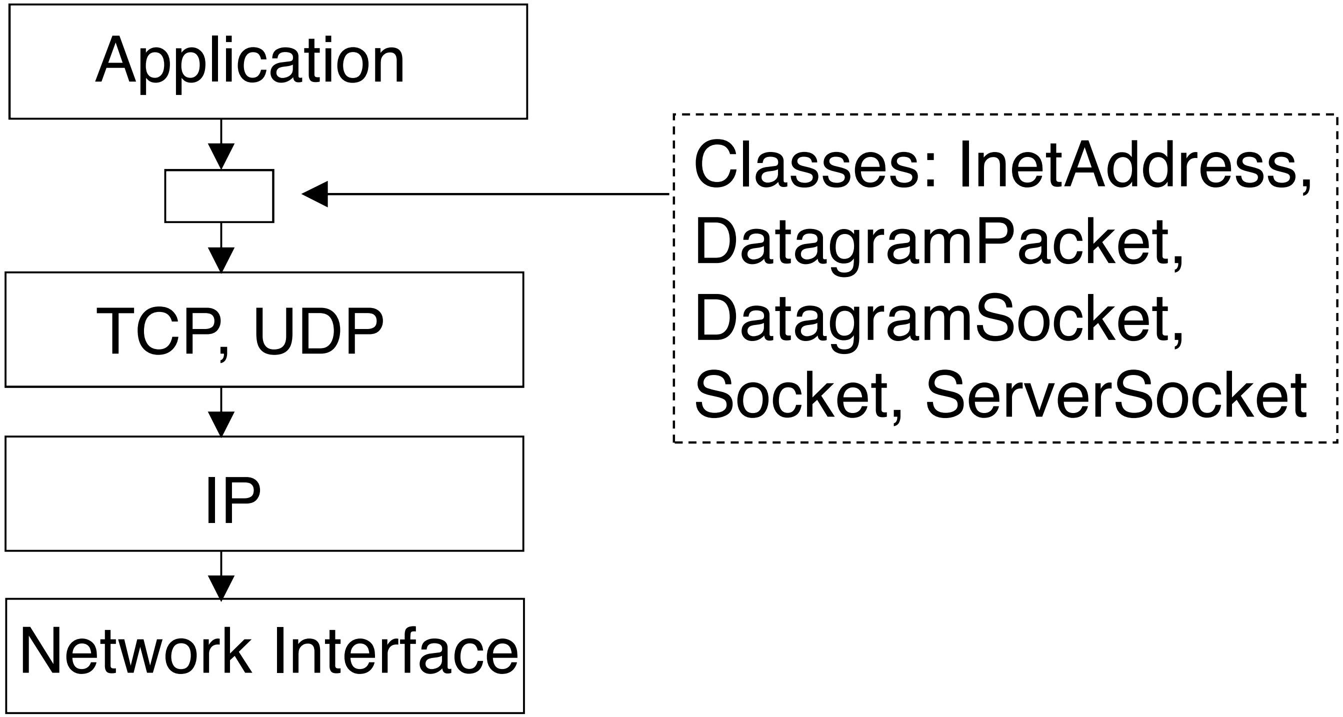
- Socket - an abstraction that is the basis for network I/O
- Analogous to a UNIX file descriptor
- Supports both reliable stream delivery and connectionless datagram delivery
- The BSD socket abstraction heavily influenced the Windows interface to the Internet (a.k.a *Winsock*) and the networking classes provided with the Java SDK (more about this in a few slides)
- The API is procedural, not object-oriented

## *4.3 BSD Unix Interface to Internet Protocols*

- `socket()` (create a new socket)
- `bind()` (bind a socket to a local port)
- `connect()` (establish a connection to a port on a destination host)
- `write()`, `writev()`, `send()`, `sendto()`, `sendmsg()`
- `read()`, `readv()`, `recv()`, `recvfrom()`, `recvmsg()`
- `accept()` (wait for connection request from a client)
- other functions…

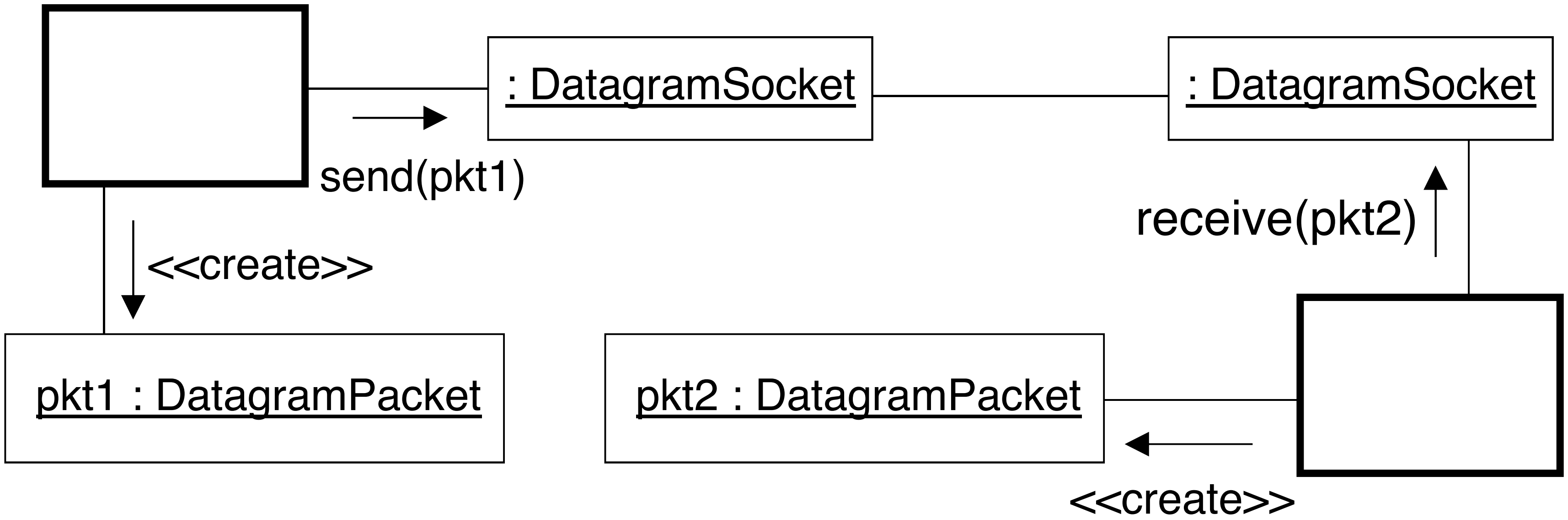
## *Java Support for Internet Protocols*

- Classes in package `java.net` provide an object-oriented view of the TCP/IP protocol stack



# *Java and UDP/IP Networking*

- Java's OO model of UDP/IP: threads create `DatagramPacket` objects, which are transferred between hosts via `DatagramSocket` objects



# *Class* `java.net.InetAddress`

- Class `InetAddress` represents an Internet Protocol (IP) address

*Descriptions of Selected Methods*

`public String getHostAddress()`

- Returns the IP address of the host on which the program is running, as a string in the format "%d.%d.%d.%d" (i.e., dotted-quad notation)

`public String getHostName()`

- Returns the name of the host on which the program is running

# *Class `java.net.InetAddress`*

```
public static InetAddress
getByName(String host)
throws UnknownHostException
```

- Determines the IP address of a host, given the host's name
- The host name can either be a machine name, such as "java.sun.com", or a string representing its IP address, such as "206.26.48.100". A `null` value for `host` means the local host (the host on which the program is running)

# *Class* `java.net.InetAddress`

```
public static InetAddress
getLocalHost()
throws UnknownHostException
```

- Returns the IP address of the local host (the host on which the program is running)

- For more information, see the Java API Specification (online at and downloadable from http://www.oracle.com/technetwork/java/api-141528.html)

## *Class* `java.net.DatagramPacket`

- Class `DatagramPacket` represents a UDP datagram

*Descriptions of Selected Constructors and Methods*

```
public DatagramPacket(byte[] buf,
                      int length)
```

- Constructs a `DatagramPacket` that models a received UDP datagram (packet) whose data "payload" has `length` bytes

# *Class* `java.net.DatagramPacket`

- `buf` is the buffer for holding the data portion of the incoming packet (i.e., header information is not stored in `buf`)
- `length` is the number of bytes of data to receive. The `length` argument does not include the size of the header in the UDP datagram, and must be less than or equal to `buf.length`

## *Class `java.net.DatagramPacket`*

```
public DatagramPacket(byte[] buf,
       int length, InetAddress address, int port)
```

- Constructs a `DatagramPacket` that models a UDP datagram (packet) whose data "payload" has `length` bytes, which is to be sent to the specified port number on the specified host
- `buf` is the buffer of packet data (i.e., header information is not stored in `buf`)

# *Class* `java.net.DatagramPacket`

- `length` is the number of bytes of data to send. The `length` argument does not include the size of the header in the UDP datagram, and must be less than or equal to `buf.length`
- `address` is the destination address
- `port` is the destination port number

## *Class* `java.net.DatagramPacket`

```
public InetAddress getAddress()
```

- Returns the IP address of the machine to which this datagram is being sent or from which this datagram was received

```
public int getPort()
```

- Returns the port number on the remote host to which this datagram is being sent or from which this datagram was received

## *Class* `java.net.DatagramPacket`

`public byte[] getData()`

- Returns the data received or the data to be sent

`public int getLength()`

- Returns the length of the data to be sent or the length of the data received

- There are also methods to set the address, port, data buffer and length of a `DatagramPacket`
- In practice, these do not seem to be as frequently used as the getter methods

## *Class* `java.net.DatagramSocket`

- Class `DatagramSocket` represents a socket for sending and receiving UDP datagrams

*Descriptions of Selected Constructors and Methods*

```
public DatagramSocket()
throws SocketException
```

- Constructs a `DatagramSocket` and binds it to any available port on the local host machine.

```
public DatagramSocket(int port)
throws SocketException
```

- Constructs a `DatagramSocket` and binds it to the specified port on the local host machine

## *Class* `java.net.DatagramSocket`

`public InetAddress getLocalAddress()`

- Returns the IP address of the local host to which the socket is bound

`public int getLocalPort()`

- Returns the port number on the local host to which this socket is bound

`public void close()`

- Closes this `DatagramSocket`

## *Class* `java.net.DatagramSocket`

```
public void send(DatagramPacket p)
throws IOException
```

- Sends a `DatagramPacket` from this socket. The `DatagramPacket` includes information indicating the data to be sent, its length, the IP address of the remote host, and the port number on the remote host

```
public void receive(DatagramPacket p)
throws IOException
```

- Receives a `DatagramPacket` from this socket. This method blocks until a datagram is received

# *Class* `java.net.DatagramSocket`

- When this method returns, the `DatagramPacket`'s buffer is filled with the data received
- The length field of the `DatagramPacket` object contains the length of the received message. If the message is longer than the packet's length, the message is truncated
- The `DatagramPacket` also contains the sender's IP address, and the port number on the sender's machine

## *Class `java.net.DatagramSocket`*

```
public void setSoTimeout(int timeout)
throws SocketException
```

- Enable/disable SO_TIMEOUT with the specified timeout, in milliseconds
- With this option set to a non-zero timeout, a call to `receive()` for this `DatagramSocket` will block for only this amount of time
- The timeout must be >= 0. A timeout of zero is interpreted as an infinite timeout

## *Class* `java.net.DatagramSocket`

- If the timeout expires, a `java.io.InterruptedIOException` is raised
- The option must be enabled prior to entering the blocking operation to have effect

```
public int getSoTimeout()
throws SocketException
```

- Returns the setting for SO_TIMEOUT
- A return value of 0 implies that the option is disabled (i.e., timeout of infinity)

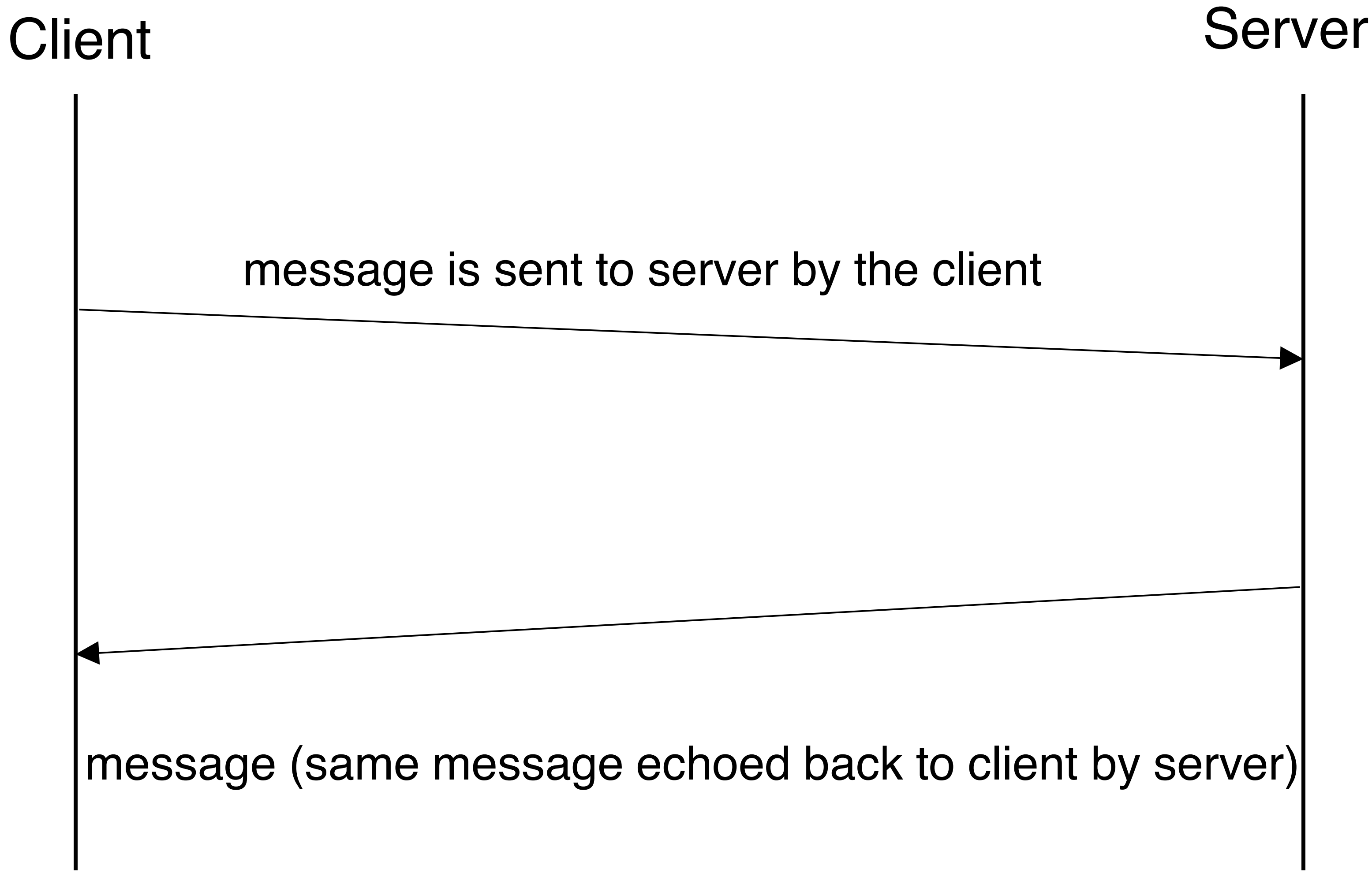
# *Java Support for TCP/IP*

- Class `Socket` implements client sockets
- Class `ServerSocket` implements sockets for servers
- We won’t use these in SYSC 3303

## *Example: An Echo Server & Client*

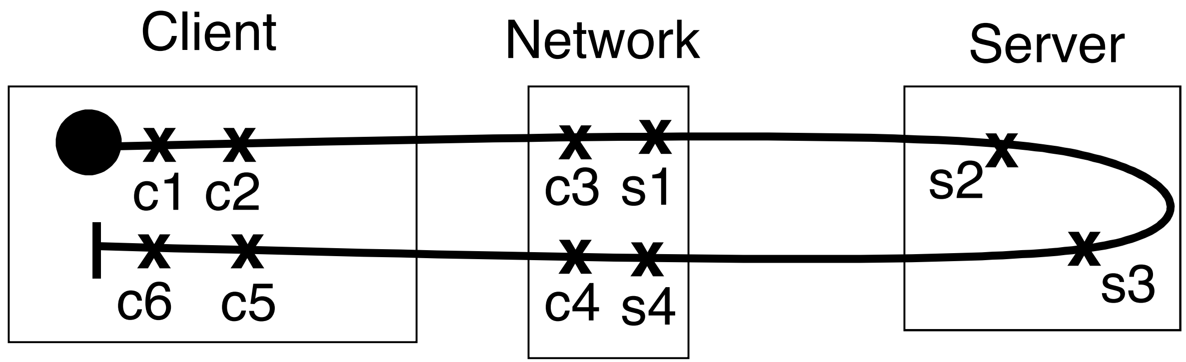
- To demonstrate how the `InetAddress`, `DatagramPacket`, and `DatagramSocket` classes are used, we'll build a simple *echo-server* and a client that uses the server
- The server waits for a message (encapsulated in a UDP datagram) to be received from a client, and echoes the message back to the client

# *Echo Client-Server Timing Diagram*



# *Echo Client-Server UCM*

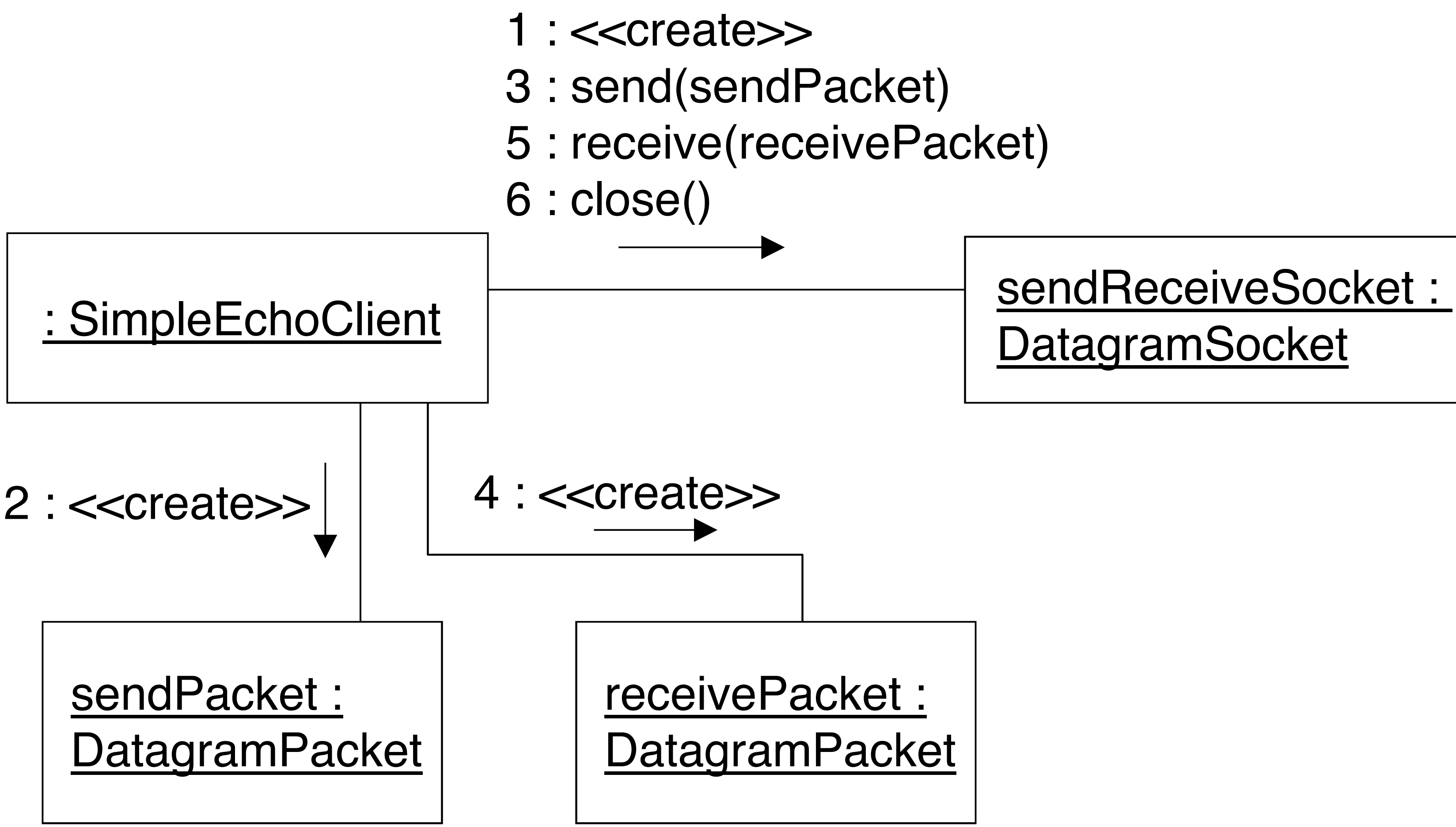
- This UCM makes no commitment to specific UML components or Java classes



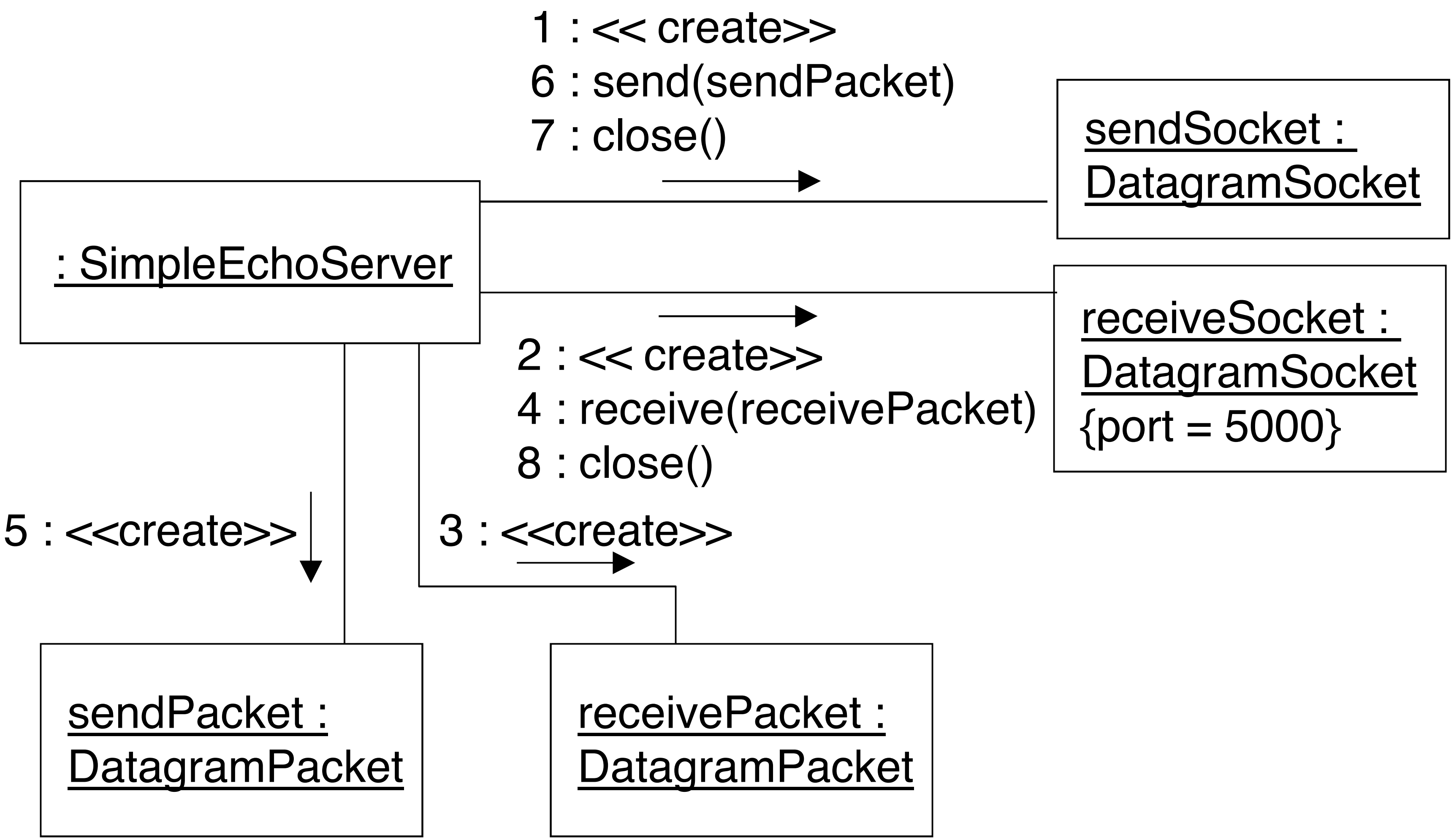

c1 - form message
c2 - create datagram
c3 - send datagram
c4 - receive datagram
c5 - extract message
c6 - print message

s1 - receive datagram
s2 - extract message
s3 - create datagram
s4 - send datagram

# Client Collaboration Diagram



# Server Collaboration Diagram



# *Code Walkthrough*

- The code for this application is in files SimpleEchoClient.java and SimpleEchoServer.java
- The following slides explain the use of the Java networking classes in these programs

## *Client: Create a `DatagramSocket`*

- The client creates a `DatagramSocket` that is bound to any available port on the local host
- This socket will be used by the client to send and receive UDP datagrams

```
DatagramSocket sendReceiveSocket;
try {
   sendReceiveSocket =
            new DatagramSocket();
} catch (SocketException se) {
   se.printStackTrace();
   System.exit(1);
}
```

## *Client: Form the Outgoing Message*

- The message is a printable character string

```
String s = "Anyone there?";
```

- Java stores characters as 16-bit Unicode values, but `DatagramPackets` store messages in byte arrays
- We must convert the `String` into bytes according to the platform's default character encoding, storing the result into a new byte array

```
byte[] msg = s.getBytes();
```

## *Client: Create a* `DatagramPacket`

- Construct a `DatagramPacket` object that contains the message to be sent (a byte array), the length of the byte array, the IP address of the destination host, and the port number on the destination host where the server waits for the datagram

```
DatagramPacket sendPacket;

sendPacket =
   new DatagramPacket(msg, msg.length,
             InetAddress.getLocalHost(),
             5000);
```

## `DatagramPacket` Constructor Arguments

- `msg` - the message to be stored in the data portion of the UDP datagram (a byte array)
- `msg.length` - the length of the byte array
- `InetAddress.getLocalHost()` - the Internet address of the destination host
  - in this example, we want the destination to be the same as the source (i.e., we want to run the client and server on the same computer)
  - `InetAddress.getLocalHost()` returns the Internet address of the local host
- `5000` - the destination port number on the destination host

## *Client: Send the UDP Datagram*

- Give the `DatagramPacket` to the `DatagramSocket` for transmission

```
try {
   sendReceiveSocket.send(sendPacket);
} catch (IOException e) {
   e.printStackTrace();
   System.exit(1);
}
```

- A UDP datagram is formed from the `DatagramPacket`, and is sent to the destination through the port bound to the `DatagramSocket`

# *Client: Create a `DatagramPacket`*

- Construct a `DatagramPacket` for receiving UDP datagrams containing up to 100 bytes of data (i.e., a message up to 100 bytes long)

```
DatagramPacket receivePacket;
byte[] data = new byte[100];
receivePacket = new
   DatagramPacket(data, data.length);
```

## *Client: Receive a UDP Datagram*

```
try {
   sendReceiveSocket.receive(receivePacket);
} catch (IOException e) {
   e.printStackTrace();
   System.exit(1);
}
```

- This method blocks until a datagram is received through the port bound to `sendReceiveSocket`
- When this method returns, its argument (a `DatagramPacket`) has been initialized to reflect the contents of the received datagram

## *Client: Receive a UDP Datagram*

- `receivePacket`'s byte array is filled with the message received from the remote host
  - the `DatagramPacket`'s length field indicates the size of the message (in bytes)
  - if the length of the message sent by the remote host was larger than the length of `DatagramPacket`'s byte array, the message is truncated and the extra data is lost
- The `DatagramPacket` also contains the sender's IP address, and the port number on the sender's machine that was used to send the datagram

## *Client: Process the* `DatagramPacket`

- Print some statistics about the received datagram

```
System.out.println("Client: Packet received:");
System.out.println("From host: " +
                   receivePacket.getAddress());
System.out.println("Host port: " +
                   receivePacket.getPort());
int len = receivePacket.getLength();
System.out.println("Length: " + len);
System.out.print("Containing: " );
```

## *Client: Process the* `DatagramPacket`

- To print the message contained in the received datagram, we need to convert “data” from an array of bytes to a String.

- Form a `String` from the byte array

```
String received =
       new String(data,0,len);
System.out.println(received);
```

## *Client: Close the Socket*

- When the program is finished with the socket (i.e., no more datagrams to send or receive), it should close it so that the resources associated with the socket are returned to the Java runtime system and underlying operating system

```
sendReceiveSocket.close();
```

## *Server: Create* `DatagramSockets`

- The server creates a `DatagramSocket` to echo the received message to the client
  - this `DatagramSocket` is bound to any available port on the local host
- It creates another `DatagramSocket` that is bound to port 5000 on the local host
  - this socket will be used by the server to wait for UDP datagrams from clients

## *Server: Create* `DatagramSockets`

```
DatagramSocket sendSocket, receiveSocket;
try {
   sendSocket = new DatagramSocket();
   receiveSocket =
            new DatagramSocket(5000);
} catch(SocketException se) {
   se.printStackTrace();
   System.exit(1);
}
```

## *Server: Receiving a Datagram*

- Create a `DatagramPacket` and wait for a datagram from the client

```
byte[] data = new byte[100];
receivePacket = new
   DatagramPacket(data, data.length);
try {
   System.out.println("Waiting...");
   receiveSocket.receive(receivePacket);
} catch (IOException e) {
   e.printStackTrace();
   System.exit(1);
}
```

# Server: Echoing

- Form a reply datagram containing the message received from the client, and send it to the client

```
data = receivePacket.getData();
sendPacket = new DatagramPacket(data,
                    receivePacket.getLength(),
                    receivePacket.getAddress(),
                    receivePacket.getPort());
try {
   sendSocket.send(sendPacket);
} catch (IOException e) {
   e.printStackTrace();
   System.exit(1);
}
```

# *Notes About Creating* `sendPacket`

- `data` is a reference to the byte array in the `DatagramPacket` received from the client
  - this array is used as the message for the datagram that will be sent to the client
- `receivePacket.getLength()` returns the length of the received message (which may be less than `data.length` - recall that array `data` is 100 bytes long)
  - this is used as the length of the message in the outgoing datagram

## *Notes About Creating* `sendPacket`

- `receivePacket.getAddress()` returns the IP address of the host that sent the datagram; i.e., the host where the client is running
  – this is used as the destination host to which the outgoing datagram will be sent
- `receivePacket.getPort()` returns the port used by the client to send the datagram
  – this is used as the destination port to which the outgoing datagram will be sent

## *Server: Close the Sockets*

- When the program is finished with the sockets (i.e., no more datagrams to send or receive), it should close them so that the resources associated with the socket are returned to the Java runtime system and underlying operating system

```
sendSocket.close();

receiveSocket.close();
```