分布式计算

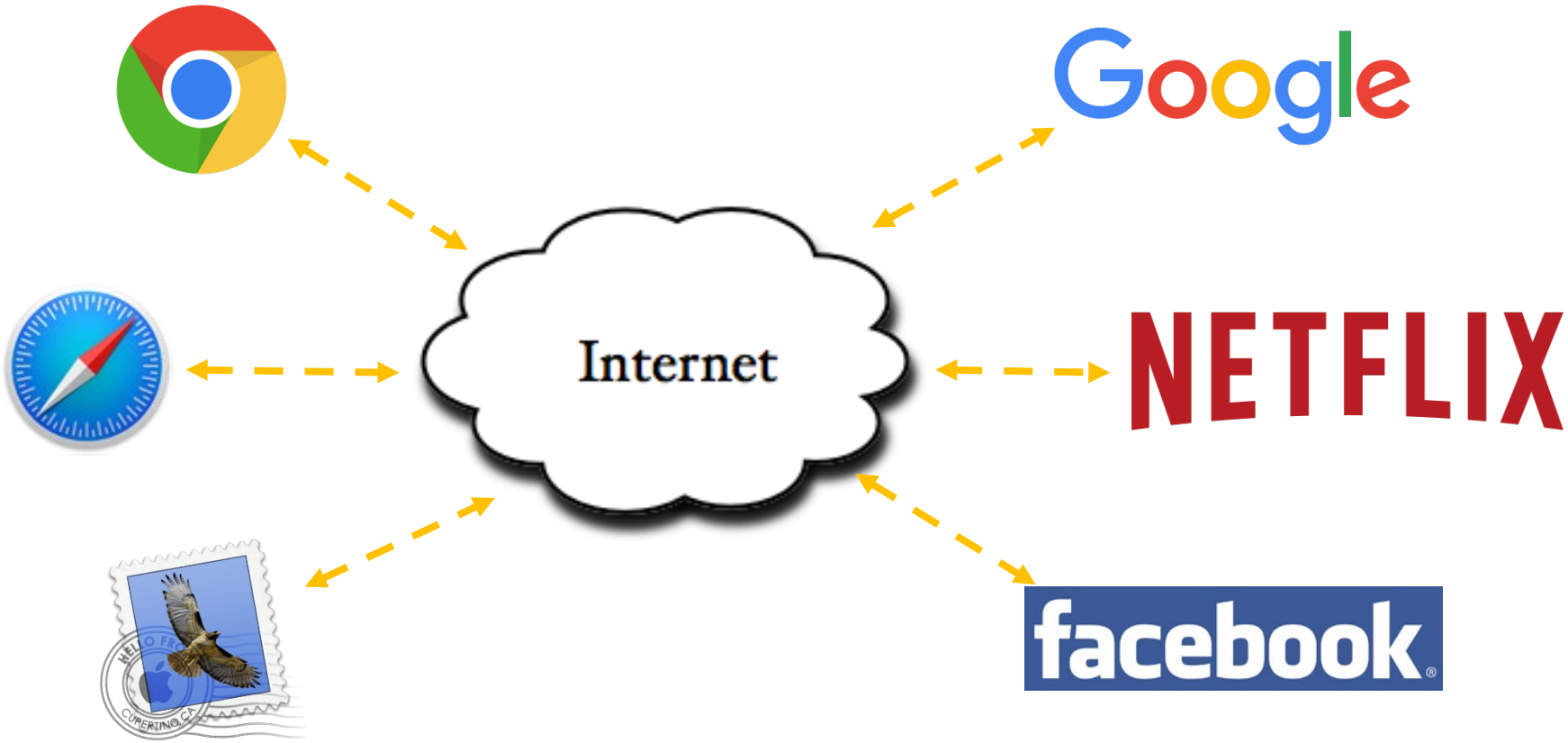
Java网络编程

Weixiong Rao 饶卫雄
Tongji University 同济大学软件学院
2023 秋季
wxrao@tongji.edu.cn

网络编程

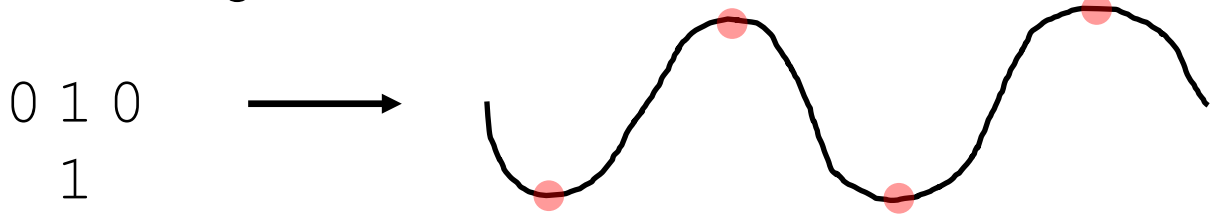
- 网络基础知识
- Socket, Client/Server

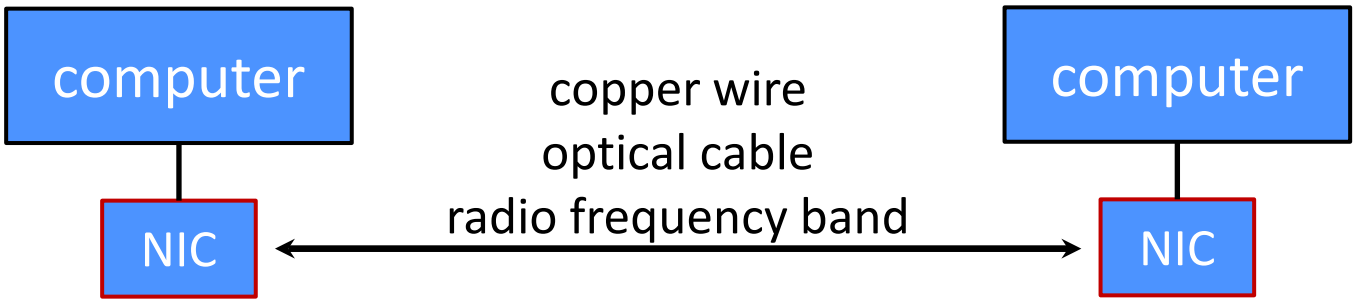
Networks From 10,000 ft



The Physical Layer

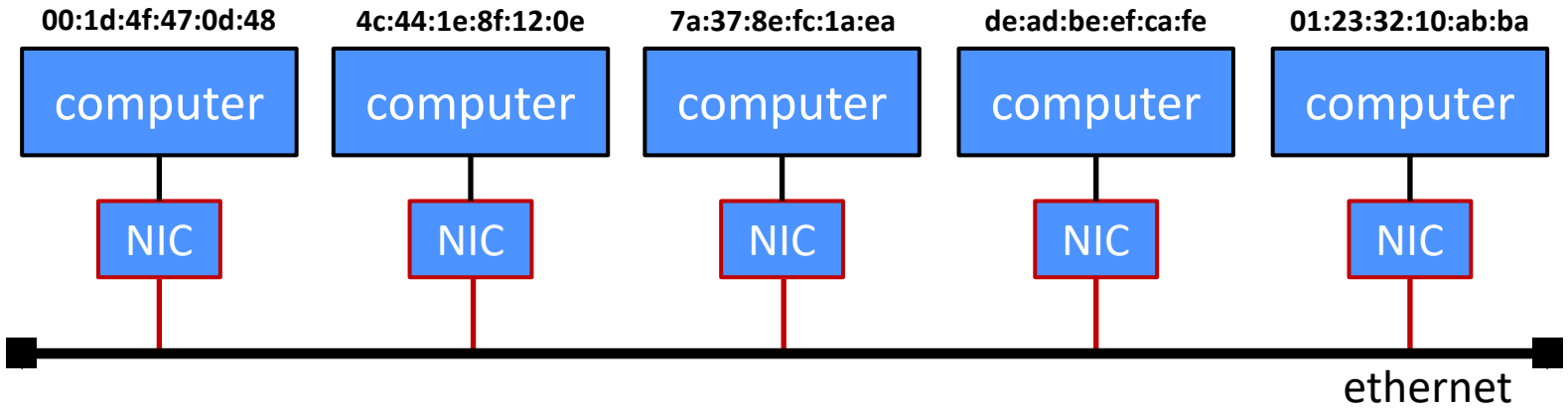
- Individual bits are modulated onto a wire or transmitted over radio
  - Physical layer specifies how bits are encoded at a signal level
  - Many choices, e.g., encode “1” as +1v, “0” as -0v; or “0”=+1v, “1”=-1v, …

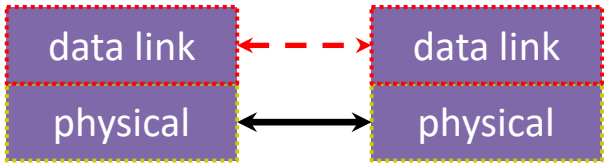


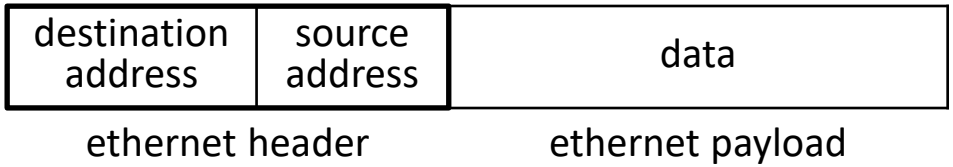


The Data Link Layer

- Multiple computers on a LAN contend for the network medium
  - Media access control (MAC) specifies how computers cooperate
  - Link layer also specifies how bits are “packetized” and network interface controllers (NICs) are addressed

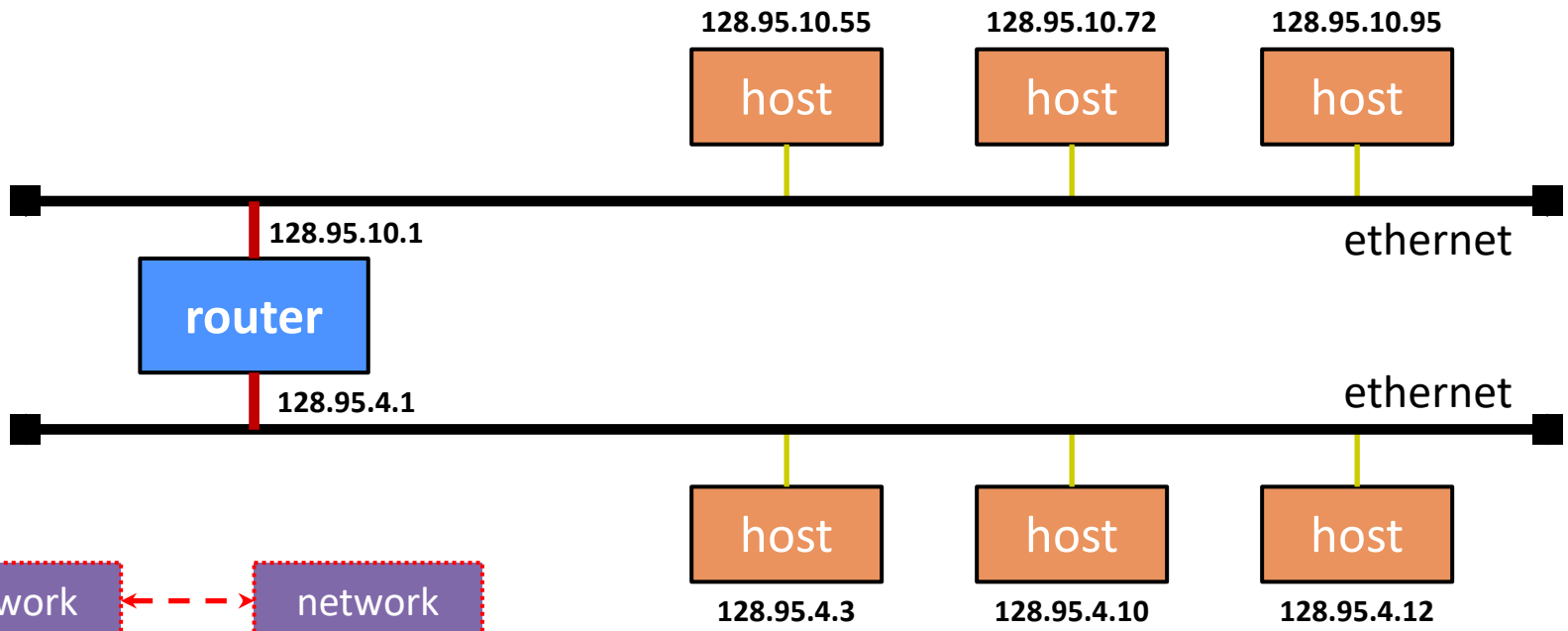


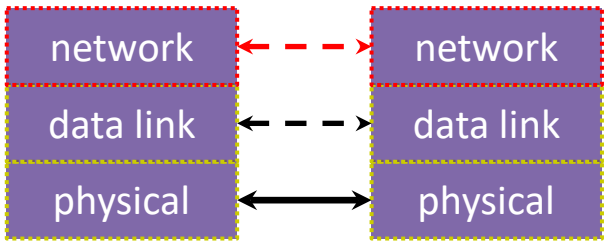




The Network Layer (IP)

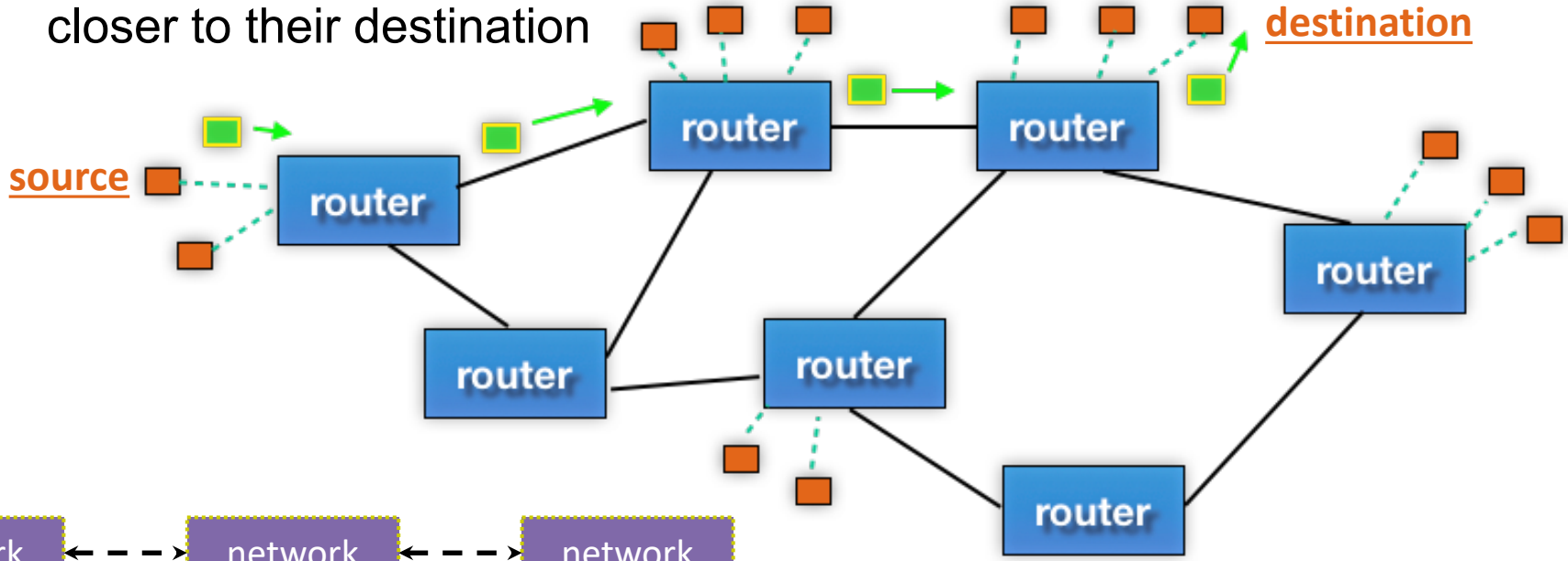
- Internet Protocol (IP) routes packets across multiple networks
  - Every computer has a unique IP address
  - Individual networks are connected by routers that span networks

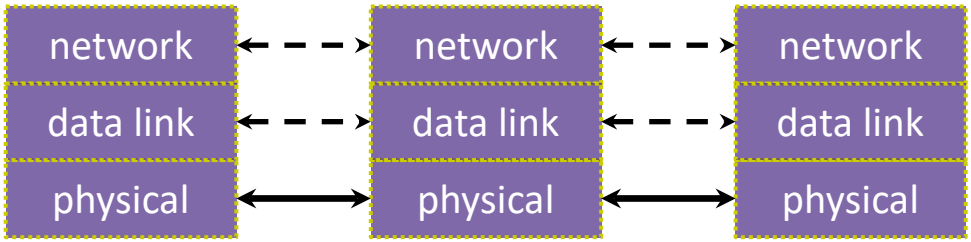




The Network Layer (IP)

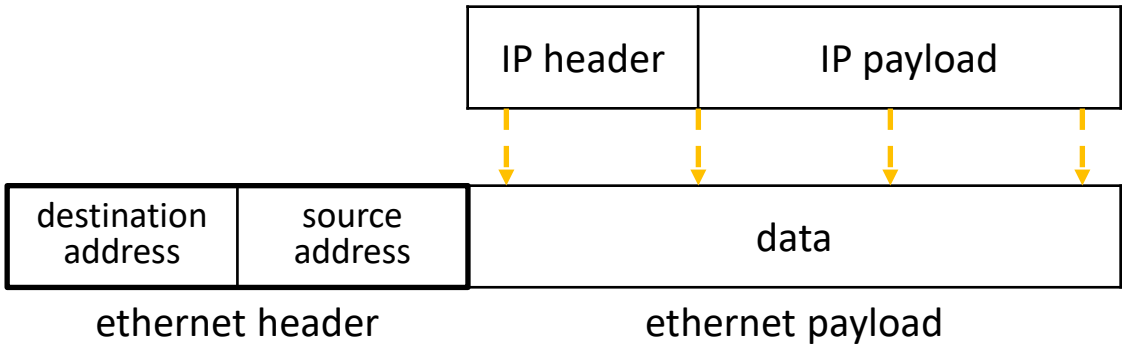
- There are protocols to:
  - Let a host map an IP to MAC address on the same network
  - Let a router learn about other routers to get IP packets one step closer to their destination

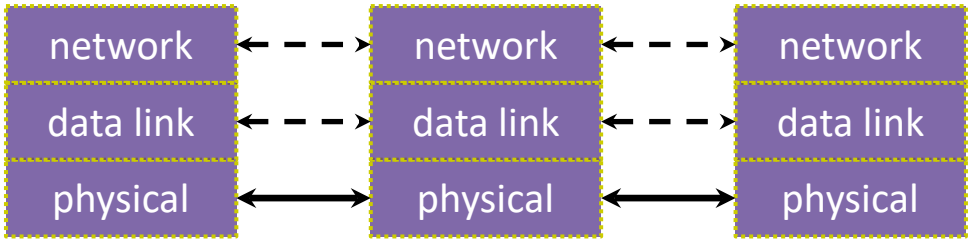




The Network Layer (IP)

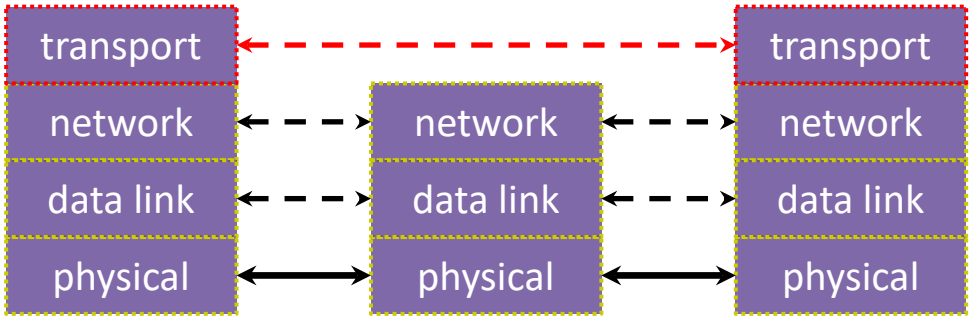
- Packet encapsulation:
  - An IP packet is encapsulated as the payload of an Ethernet frame
  - As IP packets traverse networks, routers pull out the IP packet from an Ethernet frame and plunk it into a new one on the next network





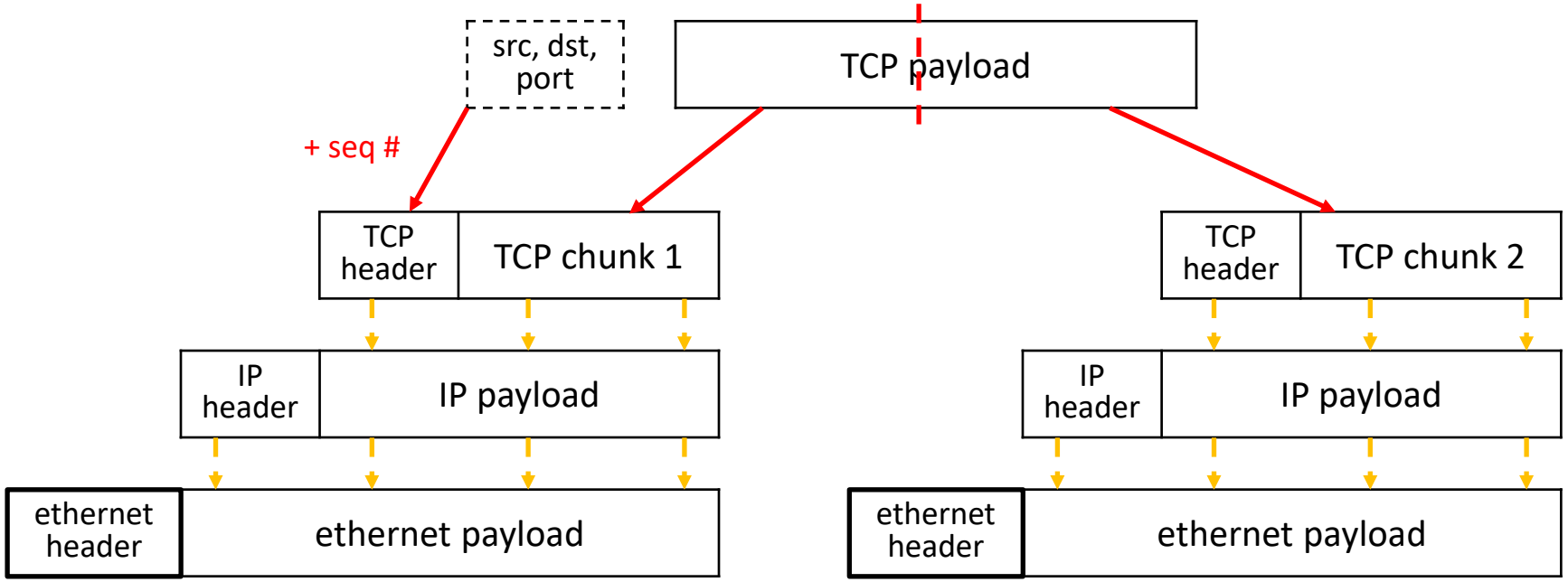
The Transport Layer (TCP)

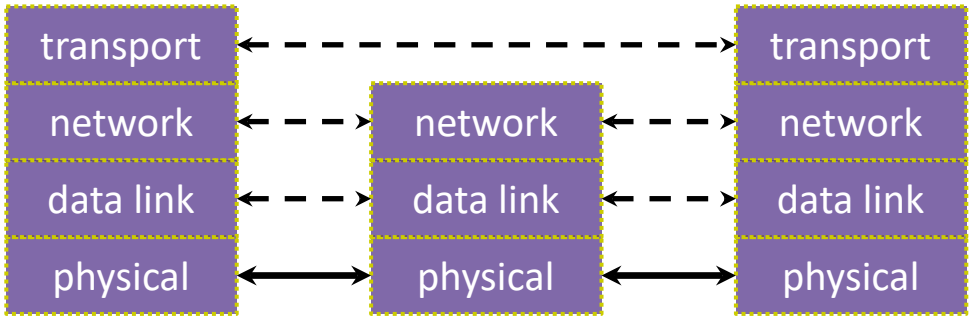
- Transmission Control Protocol (TCP):
  - Provides applications with reliable, ordered, congestion-controlled byte streams
    - Sends stream data as multiple IP packets (differentiated by sequence numbers) and retransmits them as necessary
    - When receiving, puts packets back in order and detects missing packets
  - A single host (IP address) can have up to 216 = 65,535 “ports”
    - Kind of like an apartment number at a postal address (your applications are the residents who get mail sent to an apt. #)



The Transport Layer (TCP)

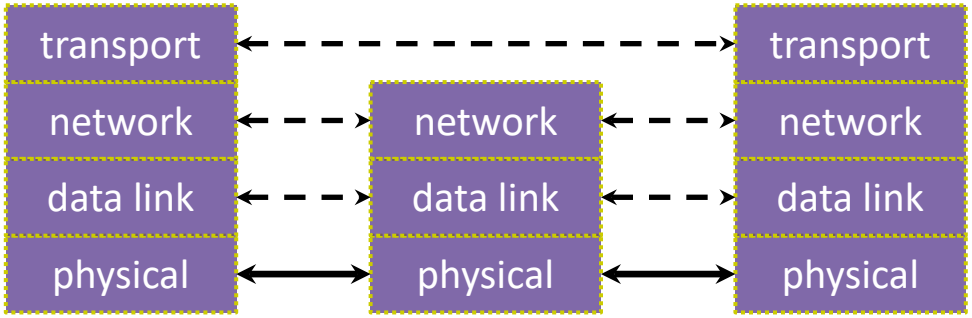
- Packet encapsulation – one more nested layer!





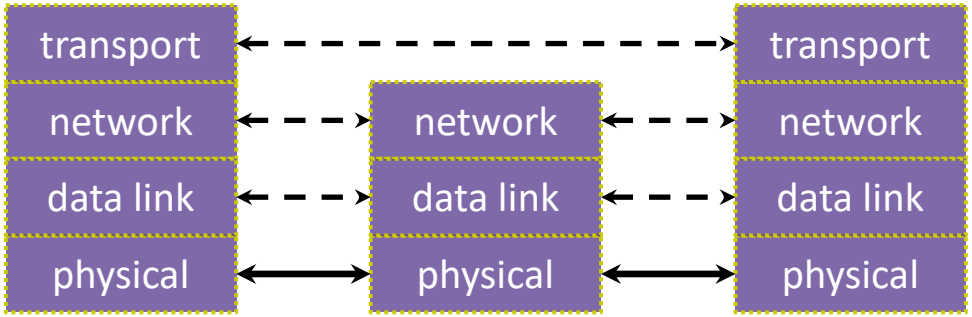
The Transport Layer (TCP)

- Applications use OS services to establish TCP streams:
  - The “Berkeley sockets” API
    - A set of OS system calls
  - Clients `connect()` to a server IP address + application port number
  - Servers `listen()` for and `accept()` client connections
  - Clients and servers `read()` and `write()` data to each other



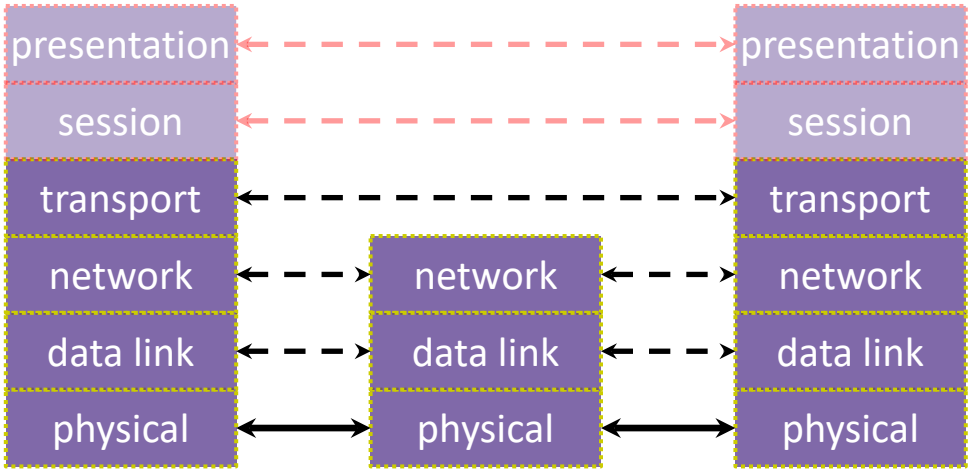
The Transport Layer (UDP)

- User Datagram Protocol (UDP):
  - Provides applications with *unreliable* packet delivery
  - UDP is a really thin, simple layer on top of IP
    - Datagrams still are fragmented into multiple IP packets



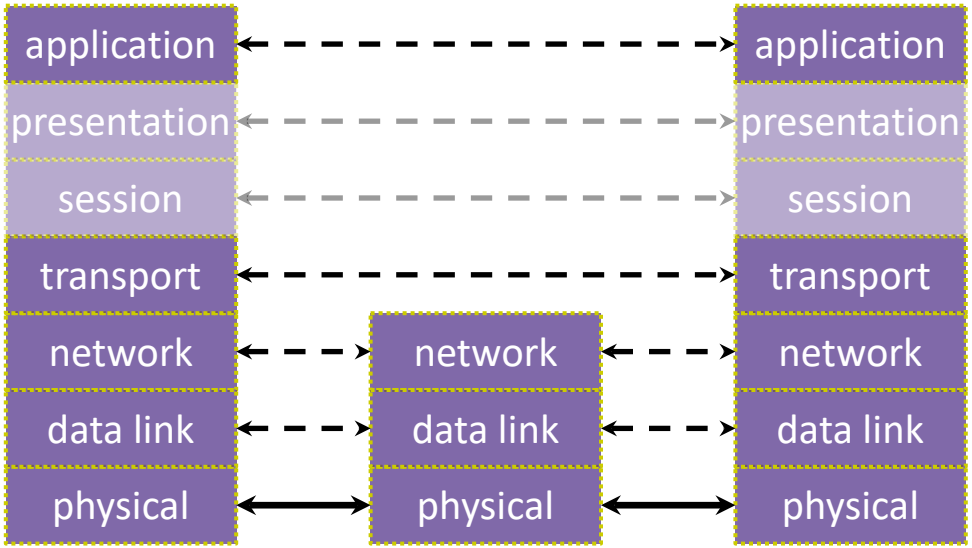
The (Mostly Missing) Layers 5 & 6

- Layer 5: Session Layer
  - Supposedly handles establishing and terminating application sessions
  - Remote Procedure Call (RPC) kind of fits in here
- Layer 6: Presentation Layer
  - Supposedly maps application-specific data units into a more network-neutral representation
  - Encryption (SSL) kind of fits in here



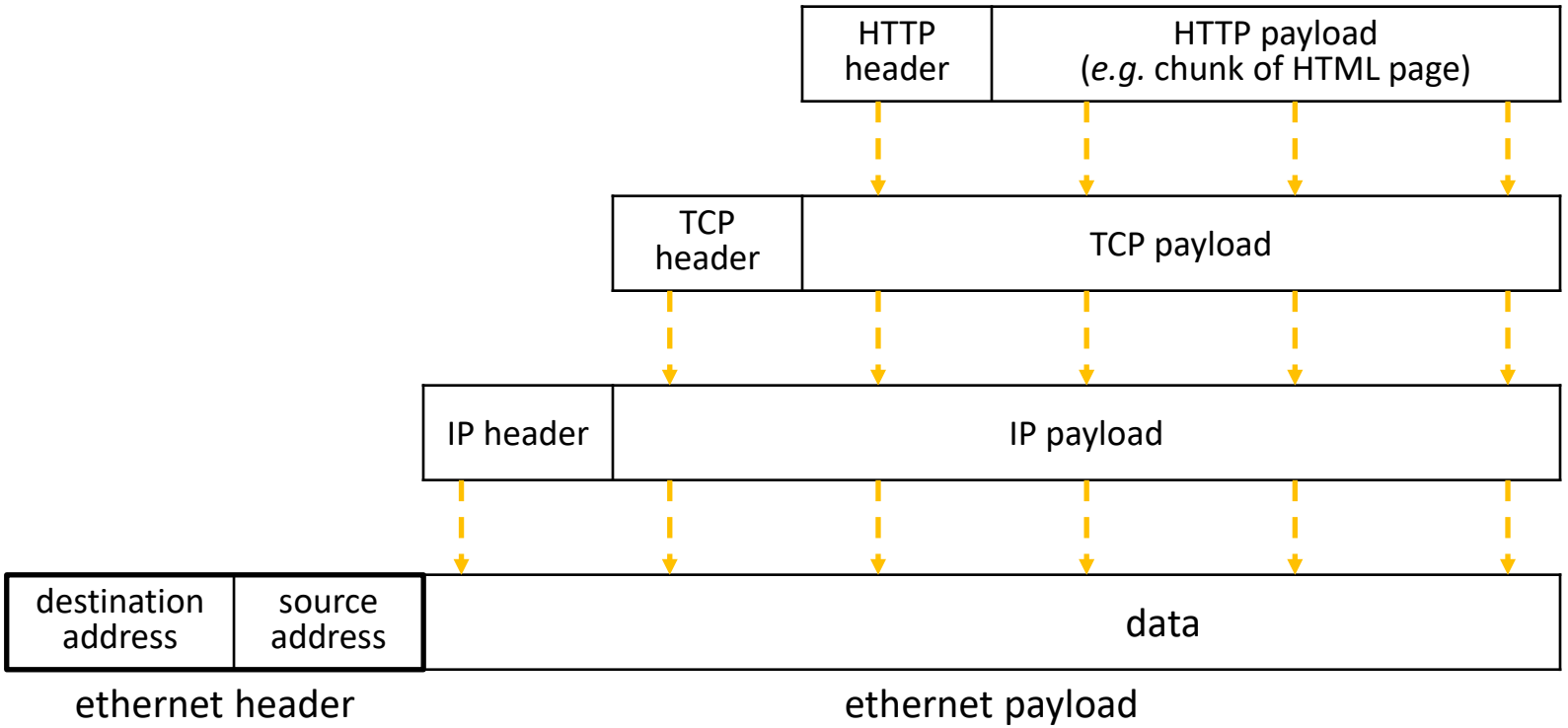
The Application Layer

- Application protocols
  - The format and meaning of messages between application entities
  - Example: HTTP is an application-level protocol that dictates how web browsers and web servers communicate
    - HTTP is implemented *on top of* TCP streams



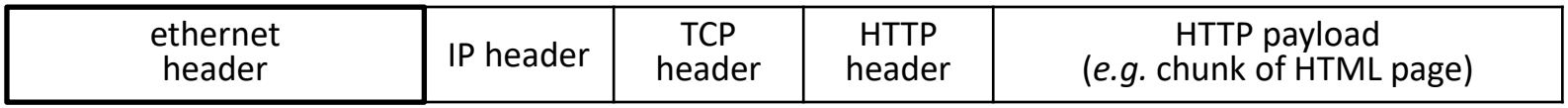
The Application Layer

- Packet encapsulation:



The Application Layer

- Packet encapsulation:



The Application Layer

- Popular application-level protocols:
  - **DNS:** translates a domain name (*e.g.* www.google.com) into one or more IP addresses (*e.g.* 74.125.197.106)
    - Domain Name System
    - An hierarchy of DNS servers cooperate to do this
  - **HTTP:** web protocols
    - Hypertext Transfer Protocol
  - **SMTP, IMAP, POP:** mail delivery and access protocols
    - Secure Mail Transfer Protocol, Internet Message Access Protocol, Post Office Protocol
  - **SSH:** secure remote login protocol
    - Secure Shell
  - **bittorrent:** peer-to-peer, swarming file sharing protocol

网络编程

- 网络基础知识
- Socket, Client/Server

https://docs.oracle.com/javase/tutorial/networking/index.html

Files and File Descriptors

- Remember open(), read(), write(), and close()?
  - POSIX system calls for interacting with files
  - open() returns a file descriptor
    - An integer that represents an open file
    - This file descriptor is then passed to read(), write(), and close()
  - Inside the OS, the file descriptor is used to index into a table that keeps track of any OS-level state associated with the file, such as the file position.

POSIX的故事

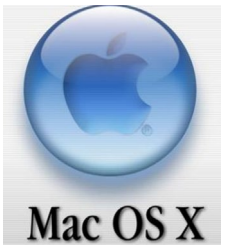
https://unix.org/version3/online.html

- 可移植操作系统接口Portable Operating System Interface of UNIX → POSIX
- POSIX是IEEE为要在各种UNIX操作系统上运行的软件而定义的一系列API标准的总称，POSIX.1 已经被国际标准化组织（International Standards Organization，ISO）所接受，被命名为 ISO/IEC 9945-1:1990 标准。
- POSIX 标准的制定最后投票敲定阶段大概是 1991~1993 年间，而此时正是 Linux 刚刚起步的时候，这个 UNIX 标准为 Linux 提供了极为重要的信息，使得 Linux 能够在标准的指导下进行开发，并能够与绝大多数 UNIX 操作系统兼容。
- 在最初的 Linux 内核源码（0.01版、0.11版）中就已经为 Linux 系统与 POSIX 标准的兼容做好了准备工作。
- 在 Linux 0.01 版内核 /include/unistd.h 文件中就已经定义了几个有关 POSIX 标准要求的符号常数，而且 Linus 在注释中已写道：“OK，这也许是个玩笑，但我正在着手研究它呢”。
- 正是由于Linux支持POSIX标准，无数可以在unix上运行的程序都陆续的移植到Linux上，而此时unix因为版权问题，官司打的不可开交，使得Linux后来者居上

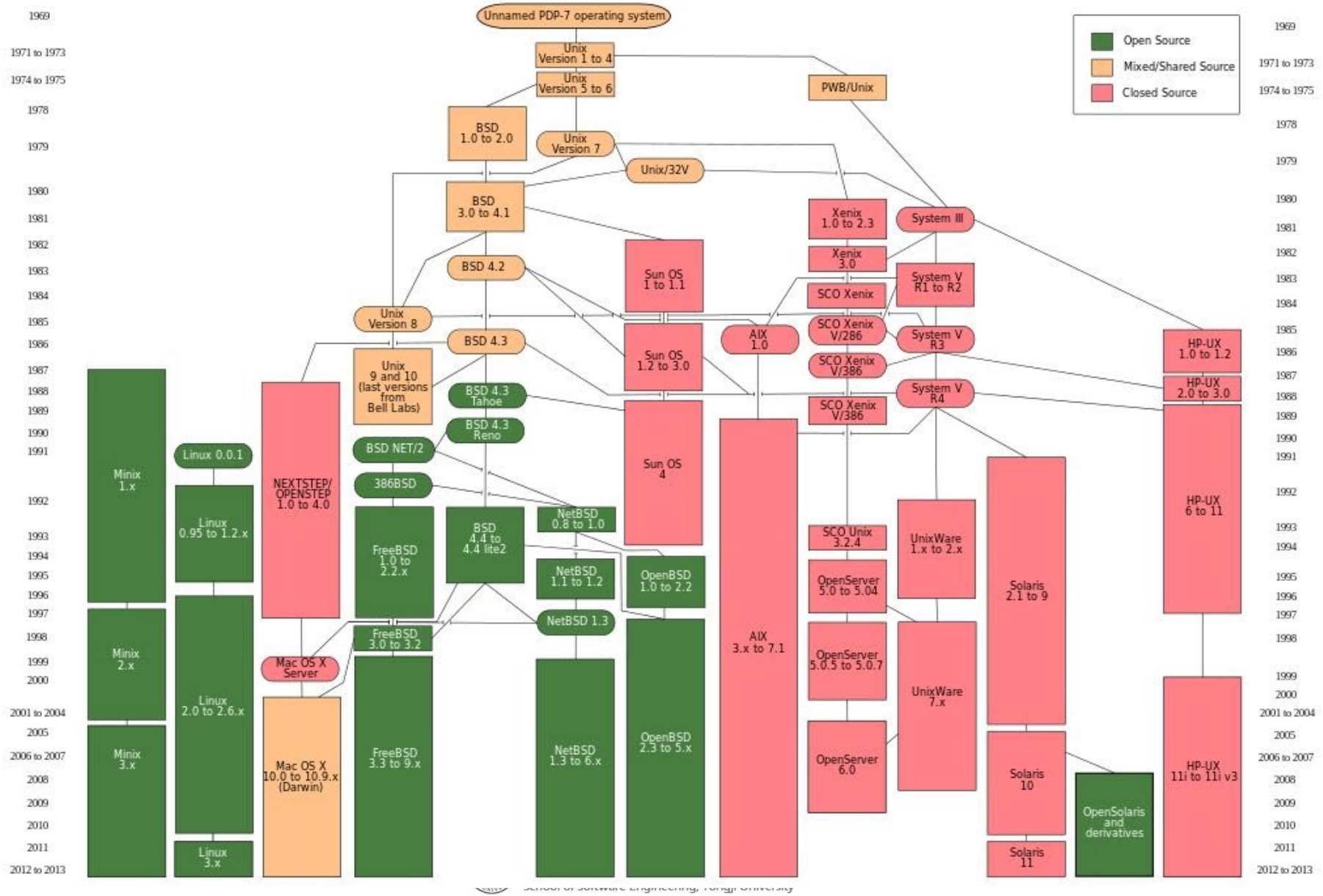
POSIX历史

- 1974年，贝尔实验室正式对外发布Unix。因为涉及到反垄断等各种原因，加上早期的Unix不够完善，于是贝尔实验室以慷慨的条件向学校提供源代码，所以Unix在大专院校里获得了很多支持并得以持续发展。
- 于是出现了独立开发的与Unix基本兼容但又不完全兼容的OS→Unix-like OS。
  - 美国加州大学伯克利分校的Unix4.xBSD(Berkeley Software Distribution)。
  - 贝尔实验室发布的自己的版本，称为System V Unix。
  - 其他厂商的版本，比如Sun Microsystems的Solaris系统,则是从这些原始的BSD和System V版本中衍生而来。
  - 20世纪80年代中期，Unix厂商试图通过加入新的、往往不兼容的特性来使它们的程序与众不
- 局面非常混乱，麻烦也就随之而来了。
  - 为了提高兼容性和应用程序的可移植性，阻止这种趋势， IEEE(电气和电子工程师协会)开始努力标准化Unix的开发，后来由 Richard Stallman命名为“Posix”。
- **谁遵循这个标准呢？**

UNIX



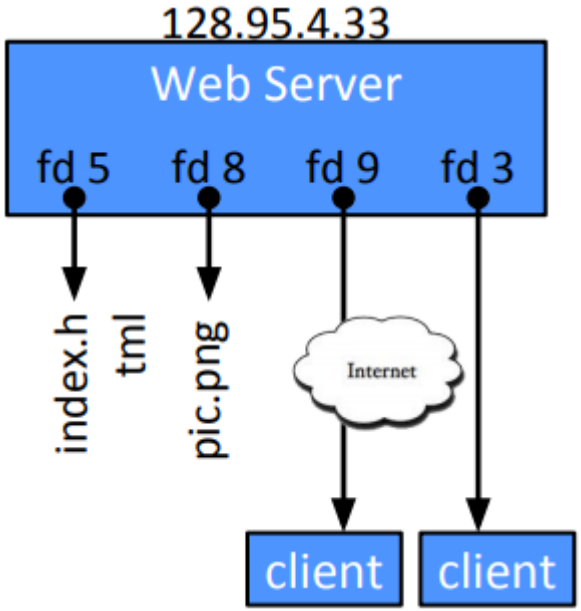
POSIX历史


Open Source
Mixed/Shared Source
Closed Source
1969
1971 to 1973
1974 to 1975
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001 to 2004
2005
2006 to 2007
2008
2009
2010
2011
2012 to 2013
Unnamed PDP-7 operating system
Unix Version 1 to 4
Unix Version 5 to 6
PWB/Unix
BSD 1.0 to 2.0
Unix Version 7
Unix/32V
BSD 3.0 to 4.1
Xenix 1.0 to 2.3
System III
Xenix 3.0
BSD 4.2
Sun OS 1 to 1.1
System V R1 to R2
SCO Xenix
Unix Version 8
BSD 4.3
AIX 1.0
SCO Xenix V/286
System V R3
HP-UX 1.0 to 1.2
Sun OS 1.2 to 3.0
SCO Xenix V/386
HP-UX 2.0 to 3.0
Unix 9 and 10 (last versions from Bell Labs)
BSD 4.3 Tahoe
System V R4
SCO Xenix V/386
BSD 4.3 Reno
Minix 1.x
Linux 0.0.1
NEXTSTEP/ OPENSTEP 1.0 to 4.0
BSD NET/2
Sun OS 4
386BSD
HP-UX 6 to 11
Linux 0.95 to 1.2.x
FreeBSD 1.0 to 2.2.x
BSD 4.4 to 4.4 lite2
NetBSD 0.8 to 1.0
SCO Unix 3.2.4
UnixWare 1.x to 2.x
Solaris 2.1 to 9
NetBSD 1.1 to 1.2
OpenBSD 1.0 to 2.2
OpenServer 5.0 to 5.04
AIX 3.x to 7.1
Minix 2.x
FreeBSD 3.0 to 3.2
NetBSD 1.3
OpenServer 5.0.5 to 5.0.7
UnixWare 7.x
Mac OS X Server
Linux 2.0 to 2.6.x
NetBSD 1.3 to 6.x
OpenBSD 2.3 to 5.x
Minix 3.x
Mac OS X 10.0 to 10.9.x (Darwin)
FreeBSD 3.3 to 9.x
OpenServer 6.0
Solaris 10
OpenSolaris and derivatives
HP-UX 11i to 11i v3
Linux 3.x
Solaris 11

Networks and Sockets

- UNIX likes to make all I/O look like file I/O
  - You use read() and write() to communicate with remote computers over the network!
  - A file descriptor use for network communications is called a **socket**
  - Just like with files:
    - Your program can have multiple network channels open at once
    - You need to pass a file descriptor to read() and write() to let the OS know which network channel to use

File Descriptor Table



OS's File Descriptor Table for the Process

File Descriptor	Type	Connection
0	pipe	stdin (console)
1	pipe	stdout (console)
2	pipe	stderr (console)
3	TCP socket	local: 128.95.4.33:80 remote: 44.1.19.32:7113
5	file	index.html
8	file	pic.png
9	TCP socket	local: 128.95.4.33:80 remote: 102.12.3.4:5544

Types of Sockets

- Stream sockets
  - For connection-oriented, point-to-point, reliable byte streams
    - Using TCP, SCTP, or other stream transports
- Datagram sockets
  - For connection-less, one-to-many, unreliable packets
    - Using UDP or other packet transports
- Raw sockets
  - For layer-3 communication (raw IP packet manipulation)

Stream Sockets

- Typically used for client-server communications
  - Client: An application that establishes a connection to a server
  - Server: An application that receives connections from clients
  - Can also be used for other forms of communication like peer-to-peer

1) Establish connection:

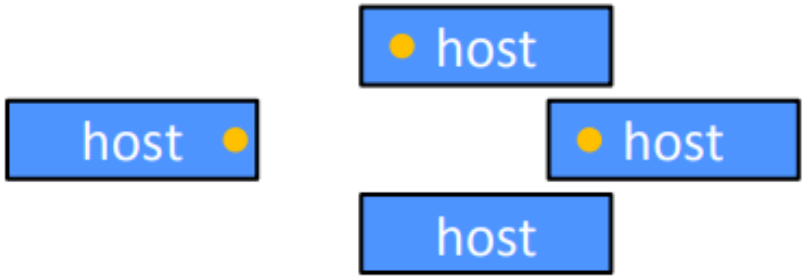
2) Communicate:

3) Close connection:

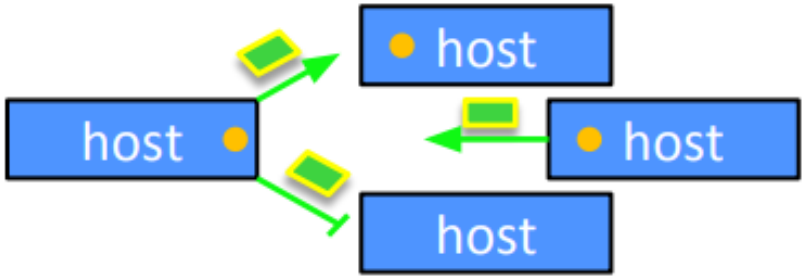
Datagram Sockets

- Often used as a building block
  - No flow control, ordering, or reliability, so used less frequently
  - e.g. streaming media applications or DNS lookups

1) Create sockets:



2) Communicate:



The Sockets API

- Berkeley sockets originated in 4.2BSD Unix (1983)
  - It is the standard API for network programming
    - Available on most OSs
  - Written in C
- POSIX Socket API
  - A slight update of the Berkeley sockets API
    - A few functions were deprecated or replaced
    - Better support for multi-threading was added

Socket API: Client TCP Connection

- We'll start by looking at the API from the point of view of a client connecting to a server over TCP
- There are five steps:

1) Figure out the IP address and port to which to connect
2) Create a socket
3) Connect the socket to the remote server
4) read() and write() data using the socket
5) Close the socket

Java Sockets Programming

- The package java.net provides support for sockets programming (and more).

- Typically you import everything defined in this package with:

```
import java.net.*;
```

Classes

InetAddress

Socket

ServerSocket

DatagramSocket

DatagramPacket

InetAddress class

- static methods you can use to create new InetAddress objects.
  - getByName(String host)
  - getAllByName(String host)
  - getLocalHost()

```
InetAddress x = InetAddress.getByName(
                    "www.tongji.edu.cn");
```

- Throws `UnknownHostException`

Sample Code: Lookup.java

- Uses InetAddress class to lookup hostnames found on command line.

```
> java Lookup www.tongji.edu.cn
www.tongji.edu.cn:192.168.66.4
```

```
try {

  InetAddress a = InetAddress.getByName(hostname);

  System.out.println(hostname + ":" +
                     a.getHostAddress());

} catch (UnknownHostException e) {

  System.out.println("No address found for " +
                     hostname);

}
```

Socket class

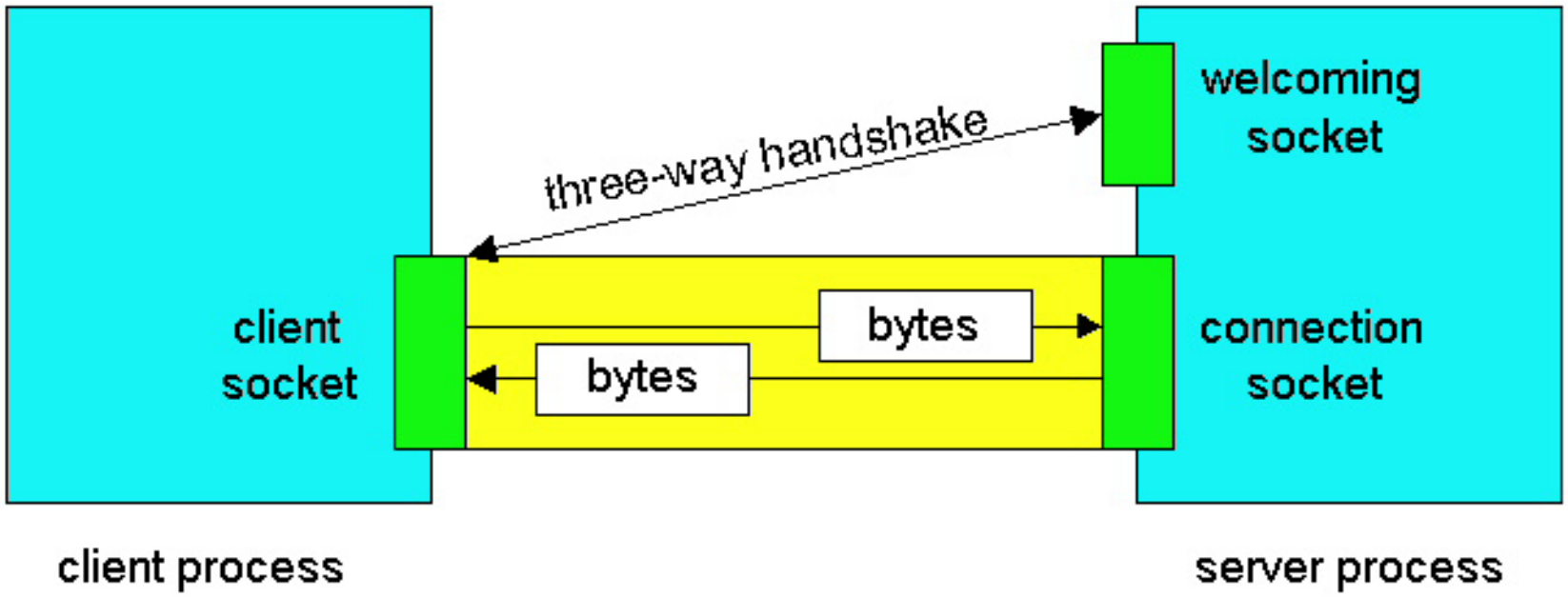
- Corresponds to active TCP sockets only!
  - client sockets
  - socket returned by accept();
- Passive sockets are supported by a different class:
  - ServerSocket
- UDP sockets are supported by
  - DatagramSocket

JAVA TCP Sockets

- java.net.Socket
  - Implements client sockets (also called just “sockets”).
  - An endpoint for communication between two machines.
  - Constructor and Methods
    - **Socket** (String host, int port): Creates a stream socket and connects it to the specified port number on the named host.
    - InputStream **getInputStream**()
    - OutputStream **getOutputStream**()
    - close()

- java.net.ServerSocket
  - Implements server sockets.
  - Waits for requests to come in over the network.
  - Performs some operation based on the request.
  - Constructor and Methods
    - **ServerSocket** (int port)
    - Socket **Accept**(): Listens for a connection to be made to this socket and accepts it. This method blocks until a connection is made.

Sockets



Client socket, welcoming socket (passive) and connection socket (active)

Socket Constructors

- Constructor creates a TCP connection to a named TCP server.
  - There are a number of constructors:

```
Socket(InetAddress server, int port);

Socket(InetAddress server, int port,
       InetAddress local, int localport);

Socket(String hostname, int port);
```

Socket Methods

```
void close();
InetAddress getInetAddress();
InetAddress getLocalAddress();
InputStream getInputStream();
OutputStream getOutputStream();
```

- Lots more (setting/getting socket options, partial close, etc.)

Socket I/O

- Socket I/O is based on the Java I/O support
  - in the package `java.io`
- InputStream and OutputStream are abstract classes
  - common operations defined for all kinds of InputStreams, OutputStreams…

InputStream Basics

```
// reads some number of bytes and
// puts in buffer array b
int read(byte[] b);

// reads up to len bytes
int read(byte[] b, int off, int len);
```

Both methods can throw IOException.

Both return –1 on EOF.

OutputStream Basics

```
// writes b.length bytes
void write(byte[] b);

// writes len bytes starting
// at offset off
void write(byte[] b, int off, int len);
```

Both methods can throw IOException.

ServerSocket Class (TCP Passive Socket)

- Constructors:

```
ServerSocket (int port);

ServerSocket (int port, int backlog);

ServerSocket (int port, int backlog,
                        InetAddress bindAddr);
```

ServerSocket Methods

Socket accept();

void close();

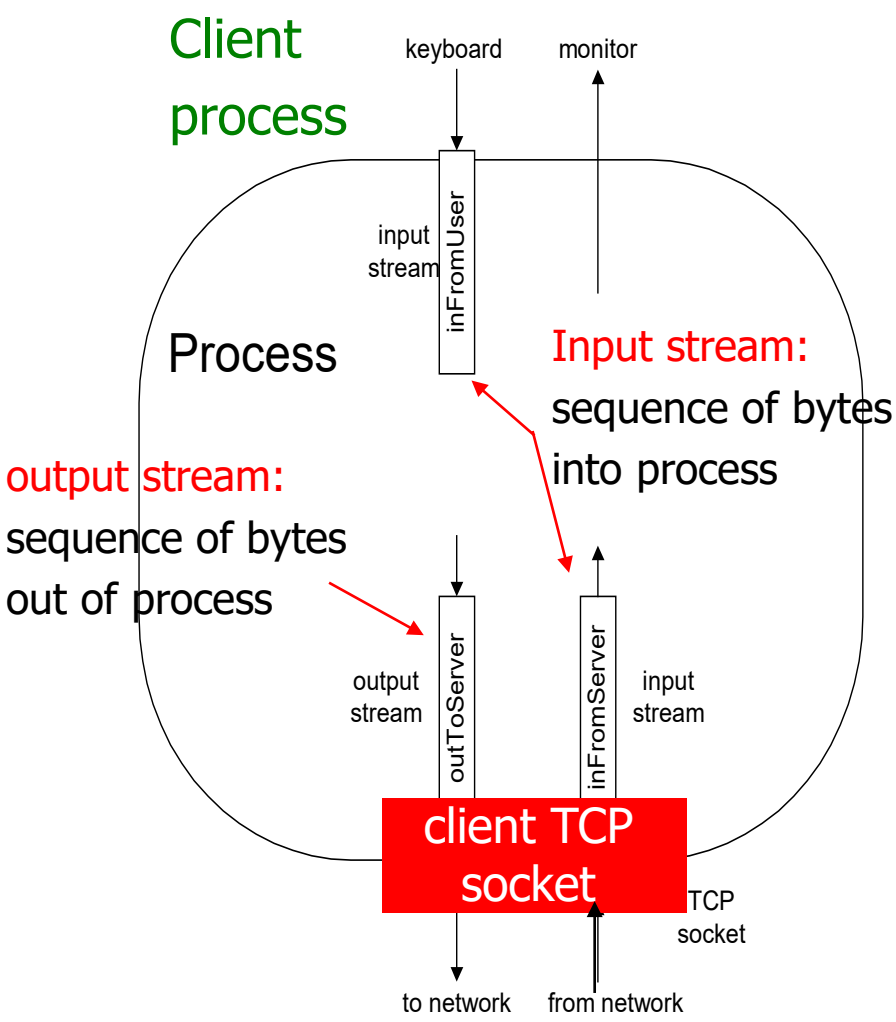
InetAddress getInetAddress();

int getLocalPort();

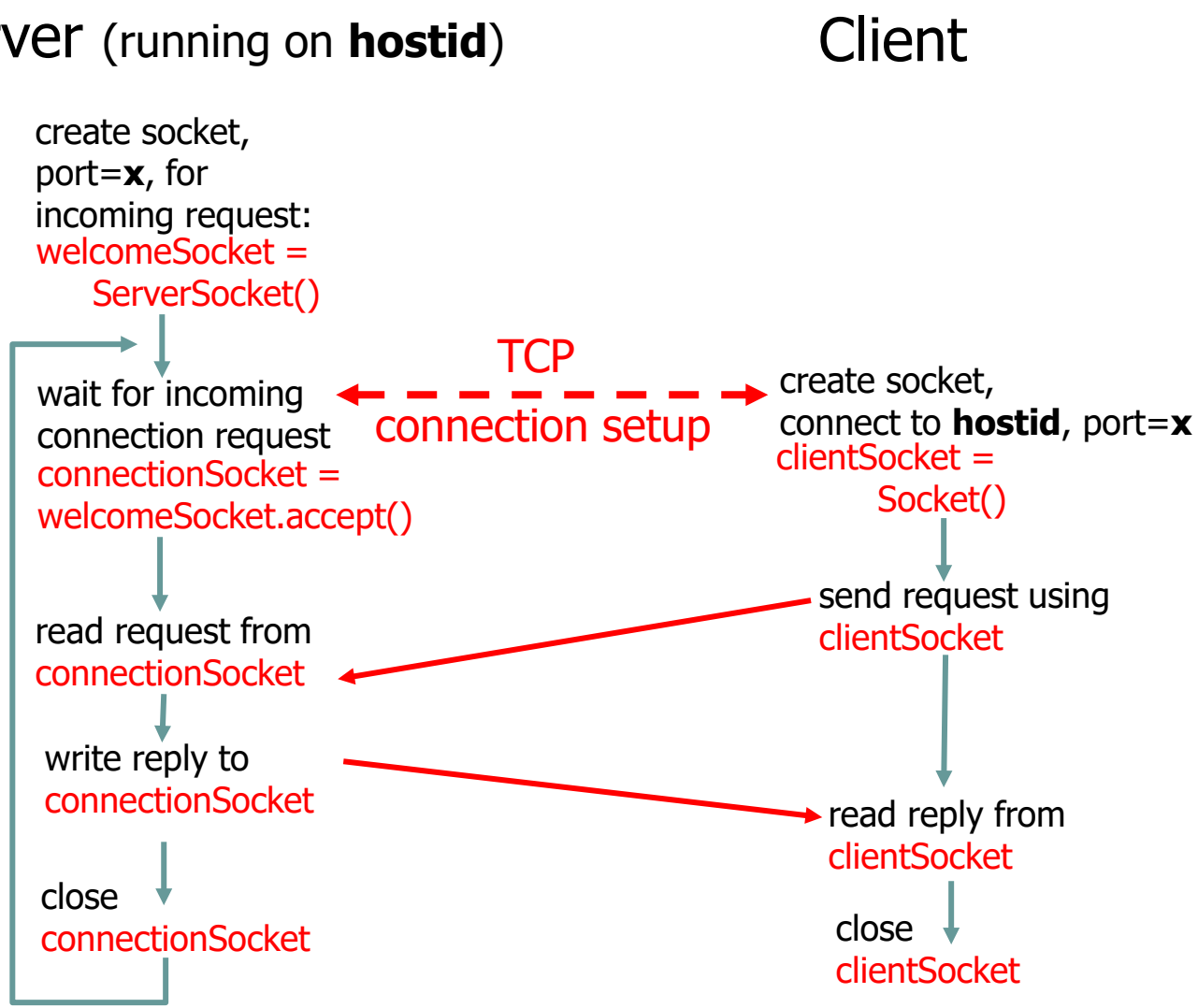
throw IOException, SecurityException

Socket programming with TCP

- Example client-server app:
- client reads line from standard input (inFromUser stream) , sends to server via socket (outToServer stream)
- server reads line from socket
- server converts line to uppercase, sends back to client
- client reads, prints modified line from socket (inFromServer stream)

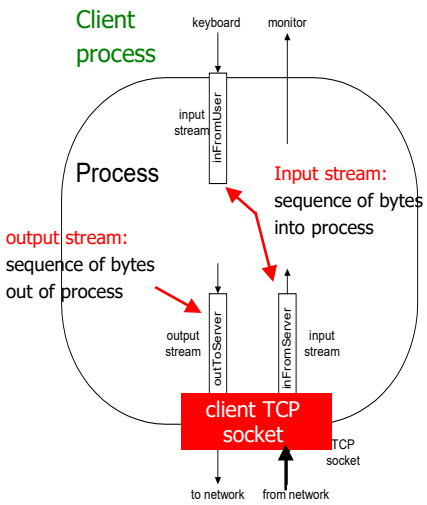


Client/server socket interaction: TCP



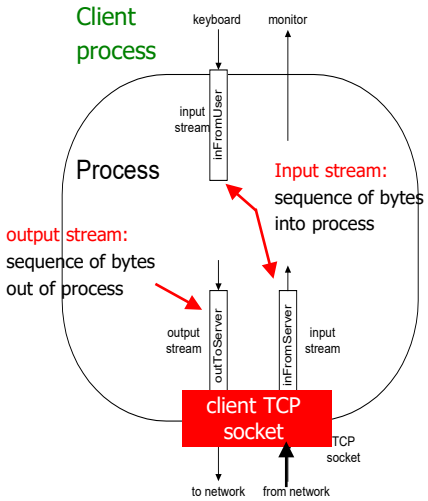
TCPClient.java

```
import java.io.*;
import java.net.*;

class TCPClient {
    public static void main(String argv[]) throws Exception
    {
        String sentence;
        String modifiedSentence;

        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));

        Socket clientSocket = new Socket("hostname", 6789);

        DataOutputStream outToServer =
                    new DataOutputStream(clientSocket.getOutputStream());
```

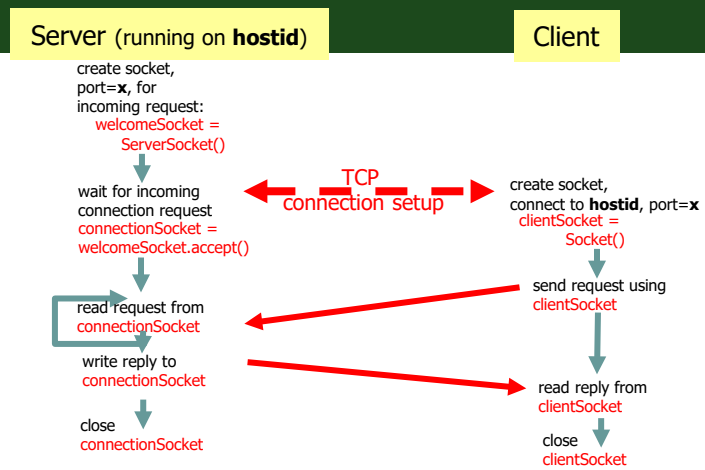


TCPClient.java

```
BufferedReader inFromServer =
        new BufferedReader(new
        InputStreamReader(clientSocket.getInputStream()));

sentence = inFromUser.readLine();

outToServer.writeBytes(sentence + '\n');

modifiedSentence = inFromServer.readLine();

System.out.println("FROM SERVER: " + modifiedSentence);

clientSocket.close();

    }
}
```



TCPServer.java




```
import java.io.*;
import java.net.*;
class TCPServer {
    public static void main(String argv[]) throws Exception
        {
          String clientSentence;
          String capitalizedSentence;
            ServerSocket welcomeSocket = new ServerSocket(6789);

            while(true) {
              Socket connectionSocket = welcomeSocket.accept();
              BufferedReader inFromClient = new BufferedReader(new
                    InputStreamReader(connectionSocket.getInputStream()));
              DataOutputStream  outToClient =
                    new DataOutputStream(connectionSocket.getOutputStream());
                clientSentence = inFromClient.readLine();
                capitalizedSentence = clientSentence.toUpperCase() + '\n';
                outToClient.writeBytes(capitalizedSentence);

            }
        }
    }
```

Sample Echo Server

TCPEchoServer.java

Simple TCP Echo server.

Based on code from:

TCP/IP Sockets in Java

https://github.com/Jonikiro/TCPEchoApp

TCPEchoServer.java

```java
public class TCPEchoServer {
    public static void main(String[] args) {
        try {
            /* Create ServerSocket, assign it port 12900, give it a
             *queue of 100, and bind it to IP address of localhost. */
            ServerSocket serverSocket = new ServerSocket(12900, 100,
                InetAddress.getByName("localhost"));
            System.out.println("Server started at: " + serverSocket);

            // Keep accepting clients in infinite loop
            while (true) {
                System.out.println("Waiting for a connection...");

                // Accept a connection and assign it to normal Socket
                final Socket activeSocket = serverSocket.accept();

                System.out.println("Received a connection from " +
                    activeSocket);

                // Create a new thread to handle the new connection
                Runnable runnable =
                    () -> handleClientRequest(activeSocket);
                new Thread(runnable).start();
            }
        } catch (IOException ex) {ex.printStackTrace();}
    }
```

```java
    public static void handleClientRequest(Socket socket) {
        BufferedReader socketReader = null;
        BufferedWriter socketWriter = null;

        try {
            // Create a buffered reader/writer for the socket
            socketReader = new BufferedReader(
                new InputStreamReader(socket.getInputStream()));
            socketWriter = new BufferedWriter(
                new OutputStreamWriter(socket.getOutputStream()));

            String inMsg = null;
            while ((inMsg = socketReader.readLine()) != null) {
                System.out.println("Received from client: " + inMsg);

                // Echo the received message to the client
                String outMsg = inMsg;
                socketWriter.write(outMsg + "\n");
                socketWriter.flush();
            }
        } catch (IOException ex) {ex.printStackTrace();}
        finally {
            try {
                socket.close();
            } catch (IOException ex) {ex.printStackTrace();}
        }
    }
}
```

TCPEchoClient.java

```
public class TCPEchoClient {
    public static void main(String[] args) {
        Socket socket = null;
        BufferedReader socketReader = null;
        BufferedWriter socketWriter = null;

        try {
            /* Create a socket that will connect to localhost
             * at port 12900. Note that server must also be
             * running at localhost and 12900. */
            socket = new Socket("localhost", 12900);
            System.out.println("Started client socket at " +
                socket.getLocalSocketAddress());

            // Create buffered reader/writer using IO streams
            socketReader = new BufferedReader(
                new InputStreamReader(socket.getInputStream()));
            socketWriter = new BufferedWriter(
                new OutputStreamWriter(socket.getOutputStream()));

            // Create buffered reader for user's input
            BufferedReader consoleReader =
                new BufferedReader(new InputStreamReader(System.in));
            String promptMsg = "Please enter a message (Bye to quit):";
            String outMsg = null;

            System.out.print(promptMsg);
            while ((outMsg = consoleReader.readLine()) != null) {
                if (outMsg.equalsIgnoreCase("bye")) {
                    break;
                }

                socketWriter.write(outMsg + "\n");
                socketWriter.flush();

                String inMsg = socketReader.readLine();
                System.out.println("Server: " + inMsg);

                System.out.println();
                System.out.print(promptMsg);
            }
        } catch (IOException ex) {ex.printStackTrace();}
        finally {
            if (socket != null) {
                try {
                    socket.close();
                } catch (IOException ex) {ex.printStackTrace();}
            }
        }
    }
}
```

UDP Sockets

- DatagramSocket class
- DatagramPacket class needed to specify the payload
  - incoming or outgoing

Socket Programming with UDP

- UDP
  - Connectionless and unreliable service.
  - There isn't an initial handshaking phase.
  - Doesn't have a pipe.
  - transmitted data may be received out of order, or lost

- Socket Programming with UDP
  - No need for a welcoming socket.
  - No streams are attached to the sockets.
  - the sending hosts creates "packets" by attaching the IP destination address and port number to each batch of bytes.
  - The receiving process must unravel to received packet to obtain the packet's information bytes.

JAVA UDP Sockets

- In Package java.net
  - java.net.DatagramSocket
    - A socket for sending and receiving datagram packets.
    - Constructor and Methods
      - DatagramSocket (int port): Constructs a datagram socket and binds it to the specified port on the local host machine.
      - void receive (DatagramPacket p)
      - void send (DatagramPacket p)
      - void close ()

DatagramSocket Constructors

- DatagramSocket ();
- DatagramSocket (int port);
- DatagramSocket (int port, InetAddress a);
- All can throw SocketException or SecurityException

Datagram Methods

```
void connect(InetAddress, int port);

void close();

void receive(DatagramPacket p);

void send(DatagramPacket p);
```

Lots more!

Datagram Packet

- Contain the payload
  - a byte array
- Can also be used to specify the destination address
  - when not using connected mode UDP

DatagramPacket Constructors

For receiving:

```
DatagramPacket( byte[] buf, int len);
```

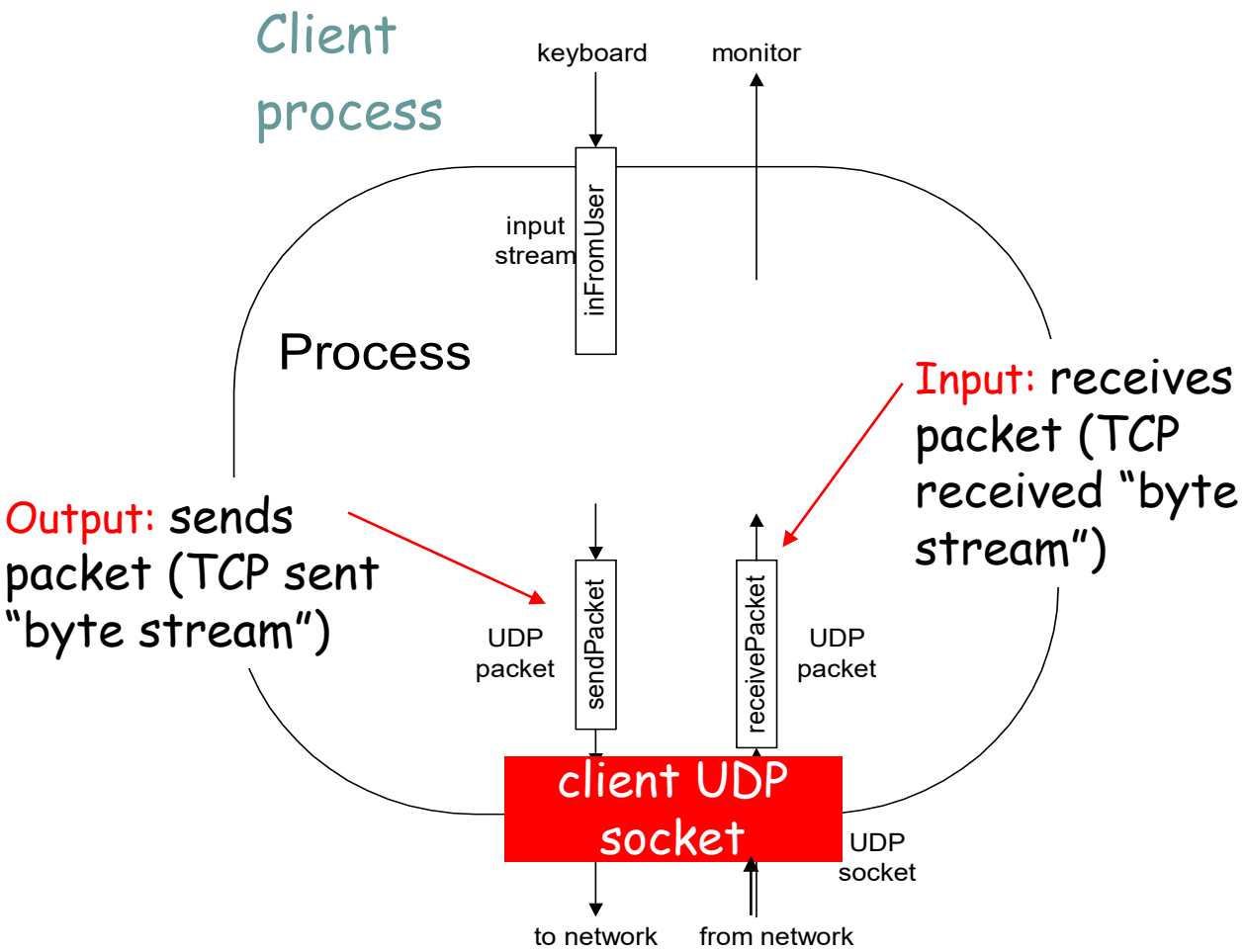
For sending:

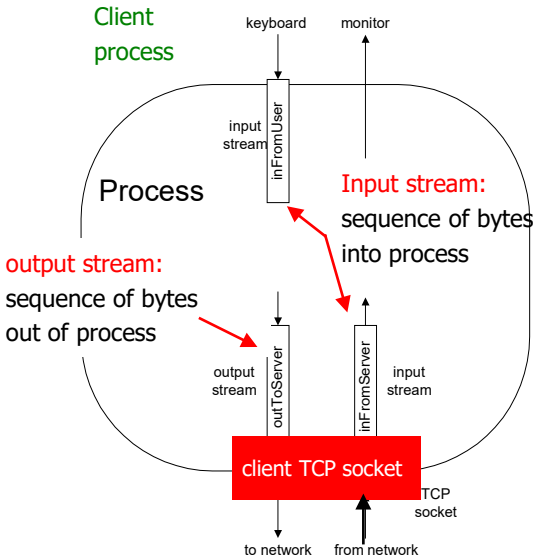
```
DatagramPacket( byte[] buf, int len
                InetAddress a, int port);
```

DatagramPacket methods

```
byte[] getData();
void setData(byte[] buf);

void setAddress(InetAddress a);
void setPort(int port);

InetAddress getAddress();
int getPort();
```

Example: Java client (UDP)





Client/server socket interaction: UDP

Server (running on **hostid**) Client

create socket, port=**x**, for incoming request: serverSocket = DatagramSocket()

read request from serverSocket

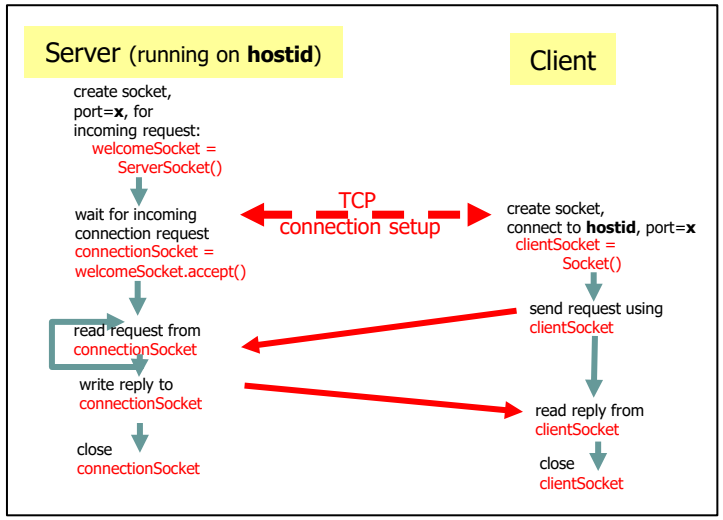
write reply to serverSocket specifying client host address, port umber

create socket, clientSocket = DatagramSocket()

Create, address (**hostid, port=x,** send datagram request using clientSocket

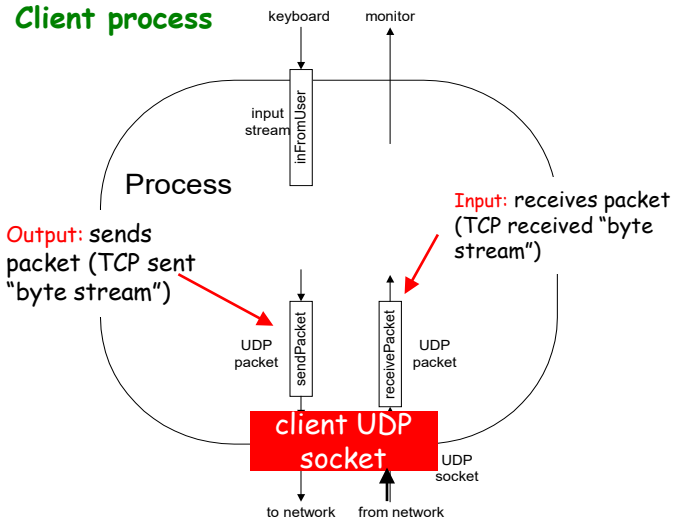
read reply from clientSocket

close clientSocket

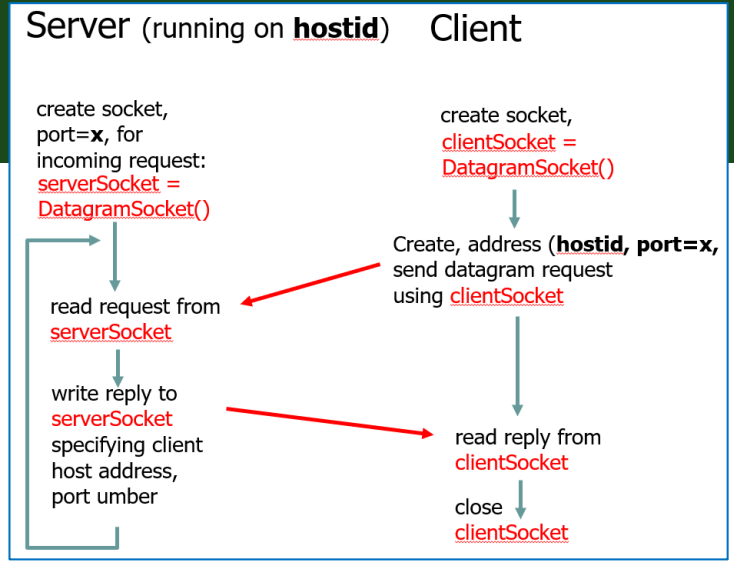


UDPClient.java

```
import java.io.*;
import java.net.*;

class UDPClient {
   public static void main(String args[]) throws Exception
   {

     BufferedReader inFromUser =
       new BufferedReader(new InputStreamReader(System.in));

     DatagramSocket clientSocket = new DatagramSocket();

     InetAddress IPAddress = InetAddress.getByName("hostname");

     byte[] sendData = new byte[1024];
     byte[] receiveData = new byte[1024];
     String sentence = inFromUser.readLine();
     sendData = sentence.getBytes();

     DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, 9876);

       clientSocket.send(sendPacket);

       DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);

       clientSocket.receive(receivePacket);

       String modifiedSentence = new String(receivePacket.getData());

       System.out.println("FROM SERVER:" + modifiedSentence);

       clientSocket.close();
     }
}
```



UDPServer.java




```
import java.io.*;
import java.net.*;

class UDPServer {

    public static void main(String args[]) throws Exception
     {
        DatagramSocket serverSocket = new DatagramSocket(9876);

        byte[] receiveData = new byte[1024];
        byte[] sendData  = new byte[1024];

        while(true)
          {

           DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);

           serverSocket.receive(receivePacket);

           String sentence = new String(receivePacket.getData());
           String capitalizedSentence = sentence.toUpperCase();
           sendData = capitalizedSentence.getBytes();

           InetAddress IPAddress = receivePacket.getAddress();

           int port = receivePacket.getPort();

           DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, port);

           serverSocket.send(sendPacket);

          }
      }
}
```

Sample UDP code

UDPEchoServer.java

Simple UDP Echo server.

Test using nc as the client (netcat):

```
> nc -u hostname port
```

Socket functional calls

socket (): Create a socket
bind(): bind a socket to a local IP address and port #

listen(): passively waiting for connections
connect(): initiating connection to another socket
accept(): accept a new connection

Write(): write data to a socket
Read(): read data from a socket
sendto(): send a datagram to another UDP socket
recvfrom(): read a datagram from a UDP socket

close(): close a socket (tear down the connection)

Java URL Class

- Represents a Uniform Resource Locator
  - scheme (protocol)
  - hostname
  - port
  - path
  - query string

Parsing

- You can use a URL object as a *parser*:

```
URL u = new URL("http://www.tongji.edu.cn");

System.out.println("Proto:" + u.getProtocol());

System.out.println("File:" + u.getFile());
```

URL construction

- You can also build a URL by setting each part individually:

```
URL u = new URL("http",
                "www.tongji.edu.cn");

System.out.println("URL:" + u.toExternalForm());

System.out.println("URL: " + u);
```

Retrieving URL contents

- URL objects can retrieve the documents they refer to!
  - actually this depends on the protocol part of the URL.
  - HTTP is supported
  - File is supported (“file://c:\foo.html”)
  - You can get “Protocol Handlers” for other protocols.

- There are a number of ways to do this:

```
Object getContent();

InputStream openStream();

URLConnection openConnection();
```

Getting Header Information

- There are methods that return information extracted from response headers:

```
String getContentType();

String getContentLength();

long getLastModified();
```

URLConnection

- Represents the connection (not the URL itself).
- More control than URL
  - can write to the connection (send POST data).
  - can set request headers.
- Closely tied to HTTP