

Introduction to Networking and Network Programming


Weixiong Rao 饶卫雄

Tongji University 同济大学计算机科学与技术学院

2025 秋季

wxrao@tongji.edu.cn

Goals of class

- 
- Basic understanding of common modern networking technology and terminology
 - Introduction of Network Programming by Java
 - Streams and File Handling
 - Network Programming

Not Goals of Class

- Deep understanding of networking
- Server administration
- Setting up your computer
- How to use email, web, etc.
- Troubleshooting (another Tech Briefing)
- TCP/IP details (another Tech Briefing)

What is a “Network”?

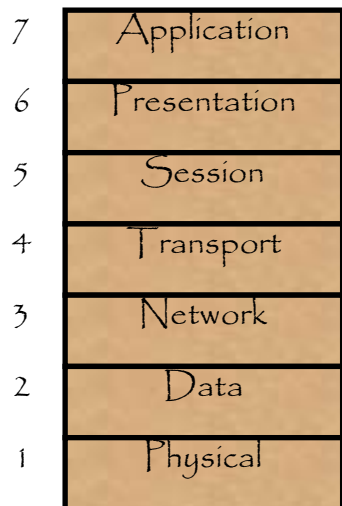
- A **network** is a way to get “stuff” between 2 or more “things”
- Examples: Mail, phone system, conversations, railroad system, highways and roads.



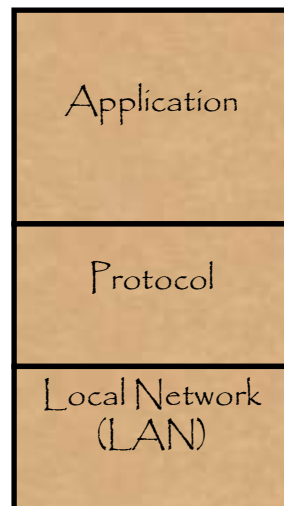
Computer Networking Models

- **Models**, also called **protocol stacks**, represented in layers, help to understand where things go right or wrong.

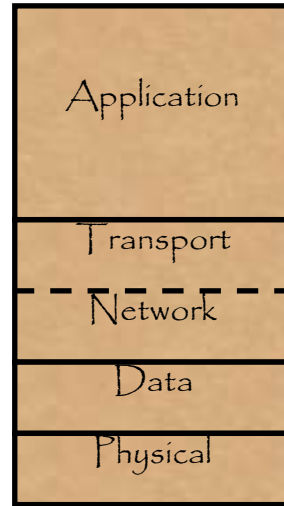
OSI 7-layer model



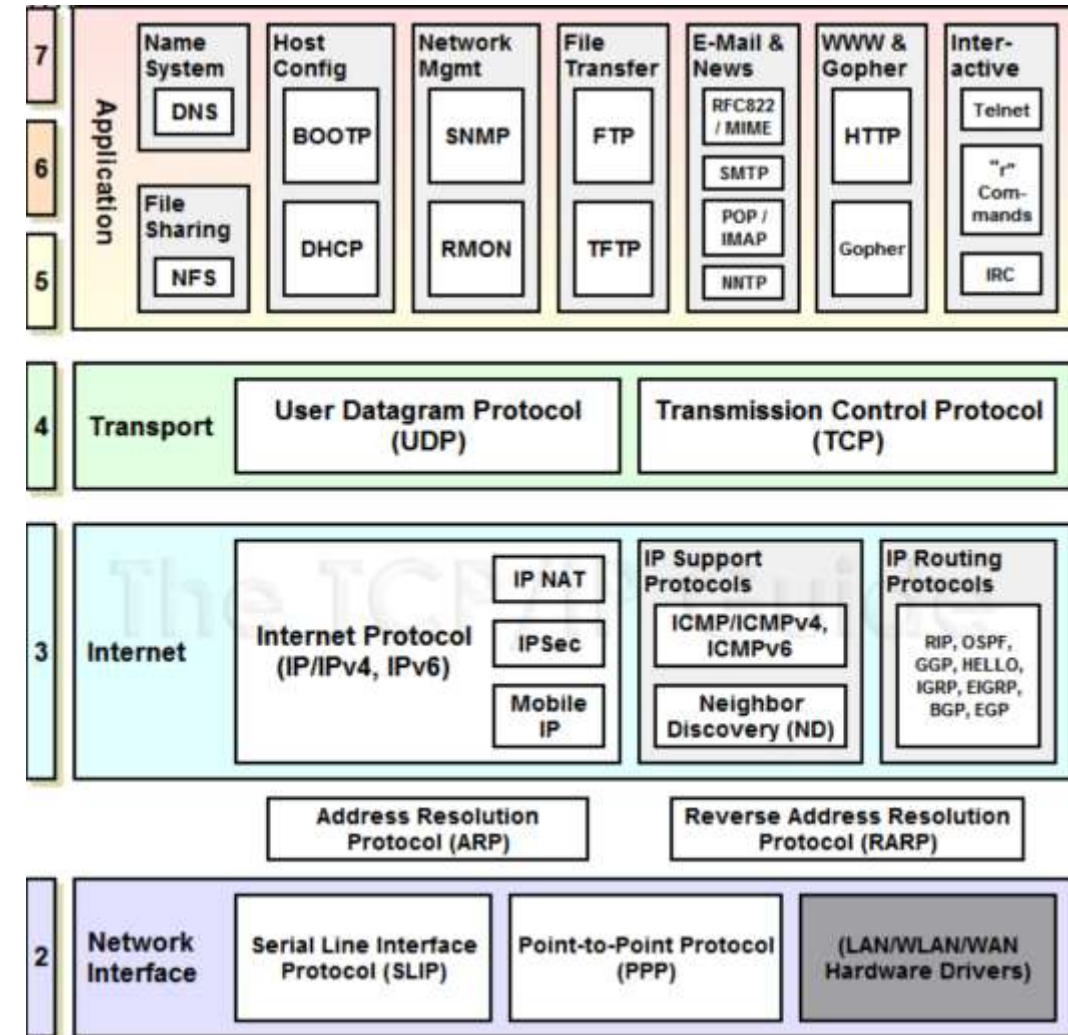
DOD 3-layer model



Simplified 4/5-layer model

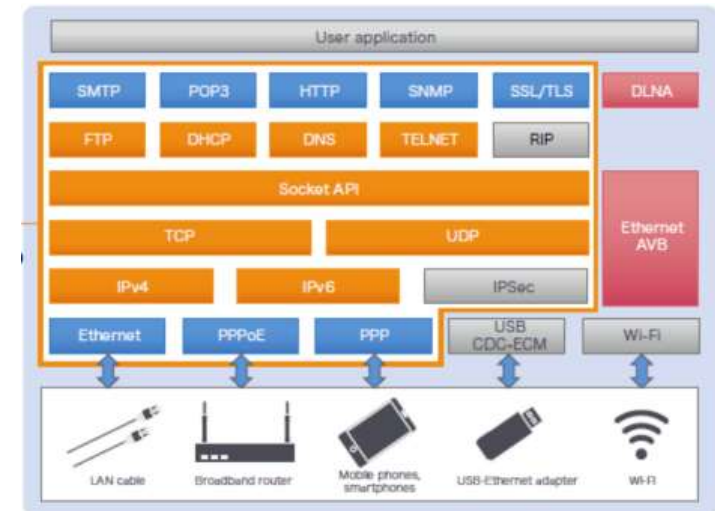
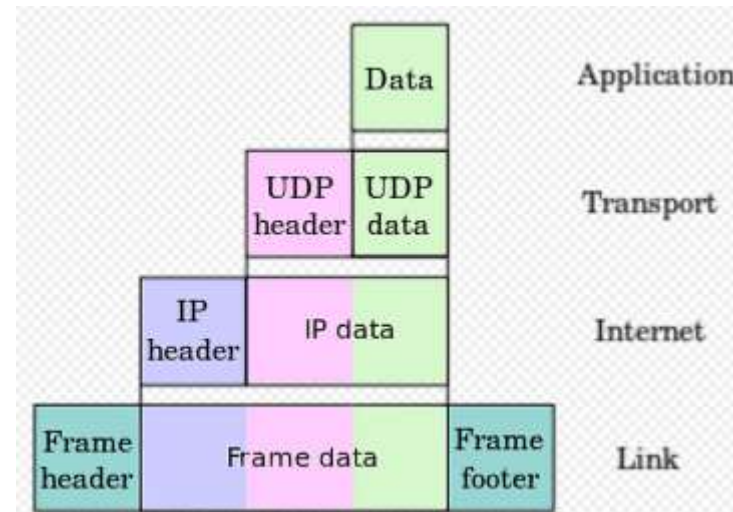
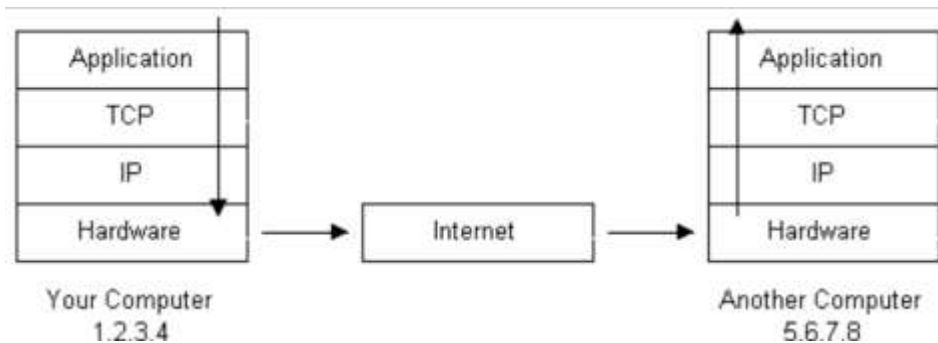


OSI (Open Systems Interconnection) mnemonic: All People Seem To Need Data Processing. If you ever take a test on networking, you'll have to now this, otherwise, use the simplified model.



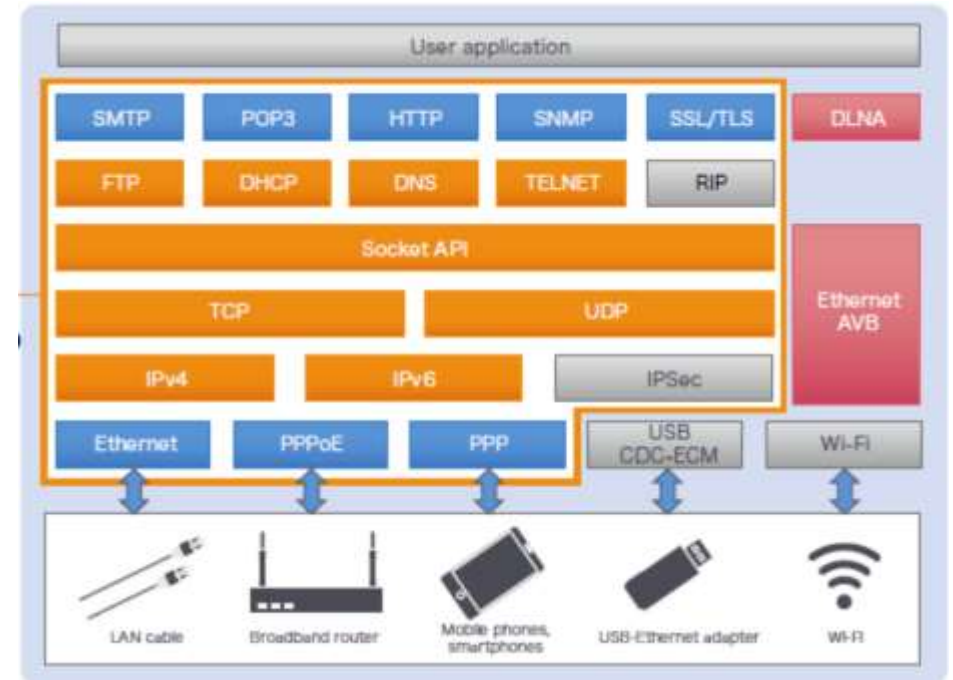
Computer Networking Models

- **Models**, also called **protocol stacks**, represented in layers, help to understand where things go right or wrong.



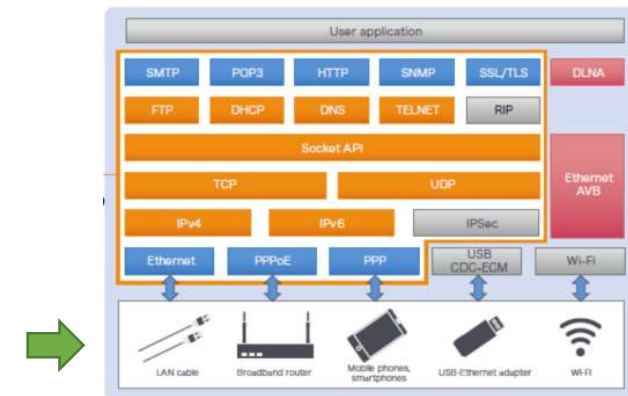
Protocol Concepts

- **Protocols** are sets of **rules**.
- **W**hat do you want to do? (**A**pplication)
- **W**here are you going? (**A**ddressing)
- **HoW** do you get there? (**M**edia types)
- Did you really get there?
(**A**cknowledgments, **E**rror checking)



Physical Layer (Layer 1)

- Nowadays: Pretty much just **Cat 5** (or Cat 5e or Cat6) twisted pair copper wire and microwave (**wireless**).
- Other: **Fiber** (multi-mode or single-mode) **coaxial copper** (thick- and thin-net), Cable Modem, plain phone (DSL), microwaves (wireless ethernet), etc.

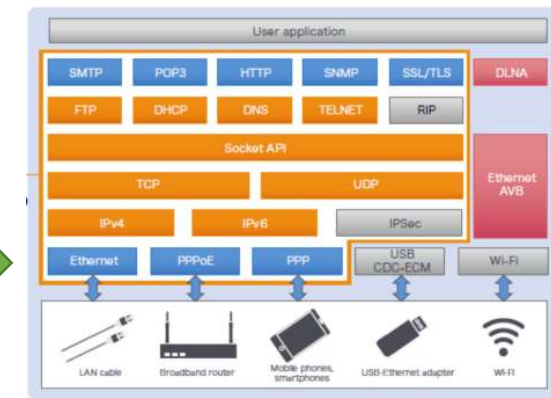


Physical: Wireless

协议	频率	通道宽度	MIMO	最大数据速率 (理论上)
802.11ax	2.4 或 5GHz	20, 40, 80, 160MHz	多用户 (MU-MIMO)	2.4 Gbps ¹
802.11ac wave2	5 GHz	20, 40, 80, 160MHz	多用户 (MU-MIMO)	1.73 Gbps ²
802.11ac wave1	5 GHz	20, 40, 80MHz	单用户 (SU MIMO)	866.7 Mbps ²
802.11n	2.4 或 5 GHz	20, 40MHz	单用户 (SU MIMO)	450 Mbps ³
802.11g	2.4 GHz	20 MHz	不适用	54 Mbps
802.11a	5 GHz	20 MHz	不适用	54 Mbps
802.11b	2.4 GHz	20 MHz	不适用	11 Mbps
传统 802.11	2.4 GHz	20 MHz	不适用	2 Mbps

- Terms: 802.11b, 802.11a, 802.11g (coming soon: 802.16 a.k.a. “WiMax”)
- Uses **microwave radio waves** in the 2.4Ghz (802.11b and g) and 5.4Ghz (802.11a and n) bands to transmit data. These are unregulated frequencies, so other things (cordless phones, etc.) can use the same frequencies, but hopefully one or the other is smart enough to hop frequencies to stay clear of the other. 802.11b and g devices can use the same access points easily. 802.11a requires separate (or dual) antennae.
- For the most part, completely and utterly **insecure**. Very easy to capture someone else's data. Make sure **your application is secure** (SSL, SSH, etc.)
- Although 802.11b at 11Mbps is the slowest (both 802.11a and g claim 54Mbps, 12-20Mbps in practice) it's the **cheapest** and most **ubiquitous**, so you'll still find some. New wireless is 802.11g.

Data Layer (Layer 2)



- The **data layer** takes the **1's and 0's** handed it by the **Network layer** and turns them into some kind of **signal** that can go over the physical layer (**electrical current, light pulses, microwaves**, etc.) It also takes this signal and turns it back into **1's and 0's** to pass up the stack on the receiving end.
- If there might be more than 2 devices on the connection, some form of addressing scheme is required to get the packet to the right destination.

Fiber Distributed Data Interface
- Some data layers: **Token Ring**, **FDDI**, **LocalTalk**, and the overwhelmingly most common data layer protocol: **Ethernet**.

Aloha 算法是最早的多路访问协议之一，其核心在于随机性。通过引入时间同步(时隙Aloha)或更复杂的二进制树形分组机制(如GFSA)，可以显著降低冲突率并提高系统效率。

Data Layer: Ethernet

载波侦听多路访问/冲突检测

- CSMA/CD: Carrier Sense, Multiple Access, Collision Detect. Simple!
- Since Ethernet was designed to be on **shared** media, with 2 or more users, and the “more” part can be very big (that’s the “Multiple Access” part) you have to listen to see if anyone else is talking before you talk (Carrier Sense) and if you and someone else start talking at the same time, notice it (Collision Detect), say “excuse me” stop and try again later. A polite free for all with rules.
- **Ethernet** is **10Mbit** (10 million bits per second) only. **Fast ethernet**, which has nearly the same rules, is **100Mbit** only. **Gigabit ethernet** is 1000Mbit only. Some Network Interface Cards (NIC’s) can speak at 10 or 100 (and sometimes 10 or 100 or 1000) but each end has to be using the same speed or there’s no connection. 10Mbit at one end and 100Mbit at the other end won’t work.

Ethernet: Addressing

- Since there can be many users on an ethernet network, everyone has to have their own unique address.
- This is called the **Media Access Control** (or **MAC**) address, or sometimes ethernet address, physical address, adaptor address, hardware address, etc.
- It's a 12-digit (48 bit) hexadecimal address that is unique to that ethernet adaptor and no other in the world. It can be written as `00:30:65:83:fc:0a` or `0030.6583.fc0a` or `003065:83fc0a` or `00-30-65-83-fc-0a` but they all mean the same thing.
- The first 6 digits are the **Vendor code**, (003065 belongs to Apple), the last 6 are the **individual interface's** own. Like a car's VIN. See http://coffer.com/mac_find/ to look up some vendor codes.

Ethernet: Finding your Address(es)

- On Windows 95/98, from the “run” menu type “[winipcfg](#)”
- On Windows NT, 2000, XP and Vista, open a command window and type “[ipconfig /all](#)” (Vista shows lots of extra junk). Make sure you get the one for the actual ethernet adaptor, not the loopback or PPP!
- On MacOS 9, open the TCP/IP control panel and select “[Get info](#)”
- On MacOS X and most Unix or Unix-like systems, from a terminal, type [ifconfig -a](#).
- Instructions with nice pictures are at <http://www.stanford.edu/services/ess/pc/sunet.html> and <http://www.stanford.edu/services/ess/mac/sunet.html>
 - Just type “ess” in your browser.

Ethernet addresses: now what?

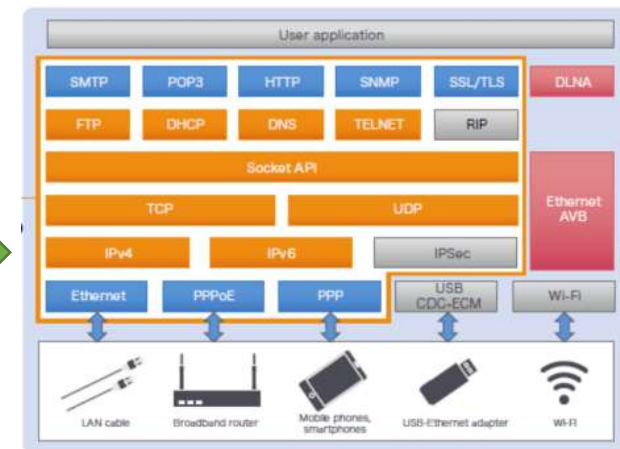
- To send someone a message, start with a **broadcast** (FFFF.FFFF.FFFF) asking “where’s Bob?” Everyone’s supposed to look at broadcasts.
- “Bob” replies, in his reply, he includes his ethernet address. Since every ethernet packet has the destination and sender address listed, “Bob” knows your address (from your broadcast packet) so doesn’t have to start with a broadcast.
- For the rest of the conversation, you’ll put each other’s address as the destination (and yours as the sender), so the conversation can pass along the ethernet media between you.
- Who’s “Bob” and how did he get that name? That’s a **layer 3 (Network)** problem, **layer 2 (Data)** doesn’t care.

Hubs vs. Switches



- **Hubs** are shared media devices. Everyone sees everyone's packets, you're only supposed to pay attention to those specifically directed to you, or to broadcasts. **Not too secure, but cheap.** Most wireless still qualifies as a "hub," while actual wired ethernet hubs are becoming hard to find.
- **Switches** aren't shared, most of the time. The switch pays attention to the packets and makes a list of the "sender" ethernet addresses and makes a table (it removes old data after a while). When a packet comes along whose destination address is in the table (because that host has recently "talked" and identified itself) the packet only goes to that port. Unknown packets and broadcasts still go to all ports, but overall, there are nearly no collisions and is generally more secure. **Switches are now much more common than hubs.**

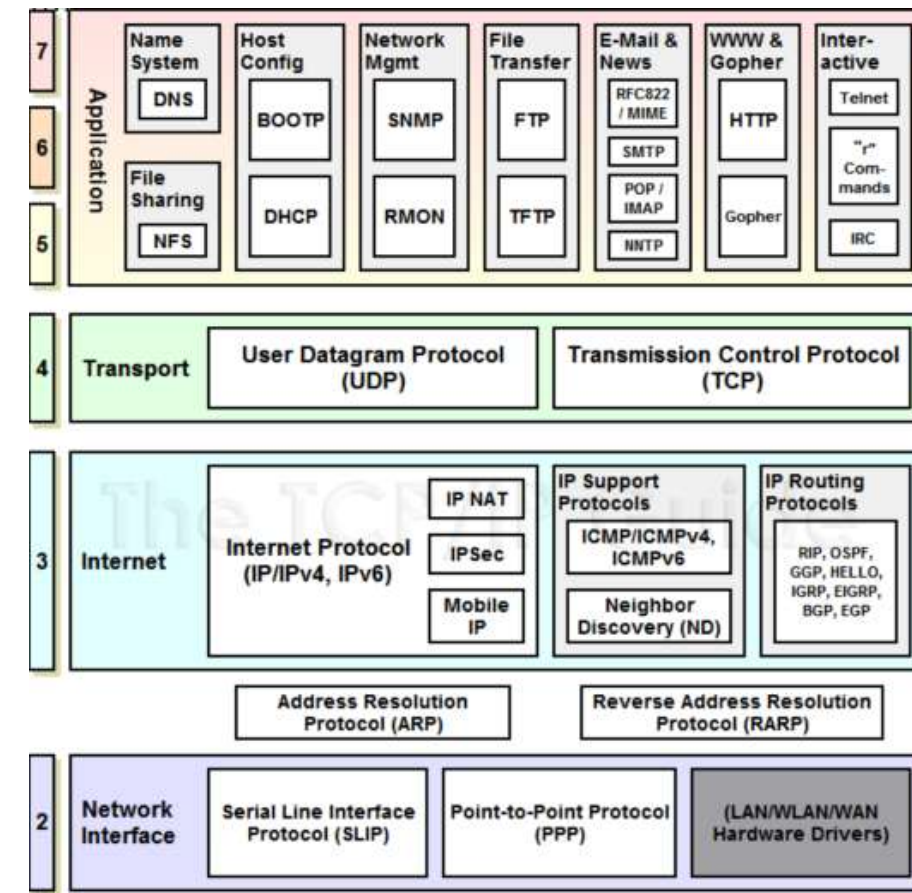
Network Layer (Layer 3)



- Network packets can be **routed**. This means they can be passed from one local network to another. **Data layer packets can't be routed, they're local only**. Your computer can only get data layer packets on its data layer interface, so network layer packets have to be stuffed inside the data layer packets. This is called “encapsulation” and is why a layered model is so handy.
- When you link computers up, via layers 1 (Physical) and 2 (Data) you get a network. When you link networks up, you get an internetwork. You need the Network layer (3) to get data between all the little networks (often called **subnets**) of your **internetwork**.
- Network Layer Protocols: **Internet Protocol (IP)** and some others that aren't used any more (**AppleTalk**, **Netware**, etc.)

Network Layer: IP

- The **Internet Protocol (IP)** is the Network layer protocol used on the Internet! It's so handy that most everyone uses it on all their networks big and small.
- Designed for huge, ever-expanding networks of networks. Works pretty well with unreliable links, routes can be re-built when links go down.
- **ARP: Address Resolution Protocol.** Turns an IP number into an ethernet number, very important. Instead of asking "Who's Bob?" you ask "Who's 172.19.4.15" and if you get a reply, associate the **ethernet address** with the **IP address** in your **arp table**, and now you can keep sending your data to the intended recipient via the correct ethernet address.
- Remember: the only packet you can actually send on ethernet is an ethernet packet, everything else has to be stuffed inside it.

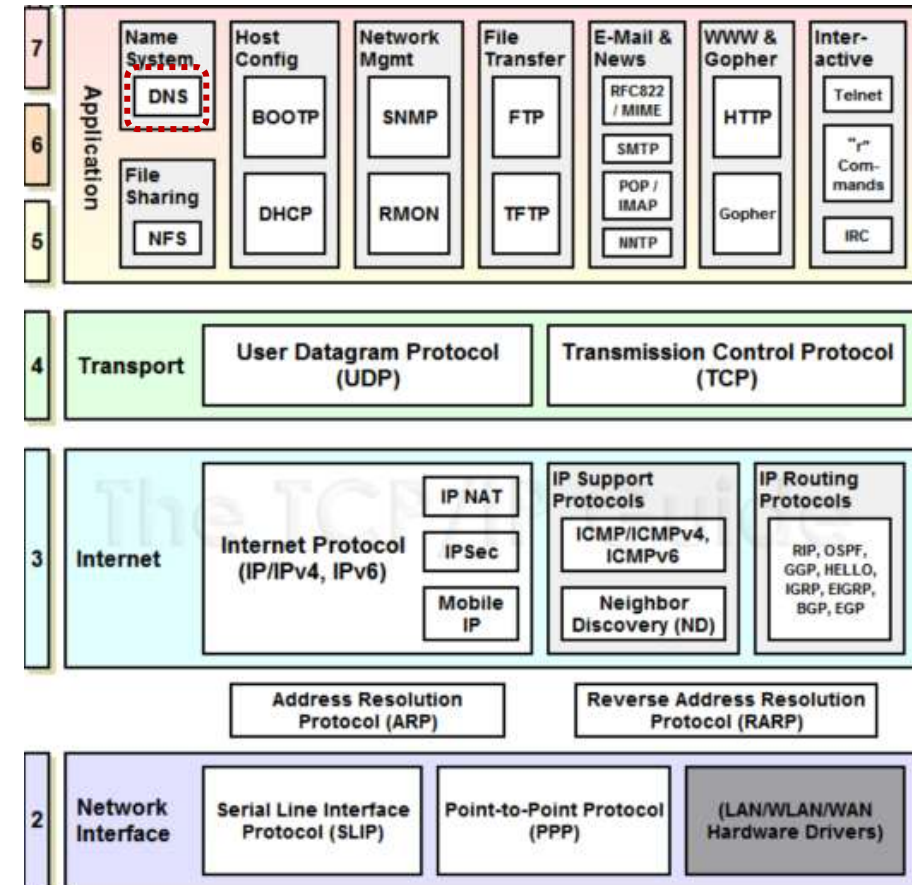


IP Addressing

- **IP addresses** consists of 4 “octets” such as: 171.64.20.23
- Each “octet” consists of numbers between 0 and 255 (or 00 and FF in hex! Don’t ask why ethernet is in hex but IP isn’t, they just are.)
- It works sort of like the phone system, with “area codes” to the left, then “prefix” etc. but more flexible. On campus, your computer will know that “171.64.” means “Stanford” while it will figure out that “20” means “Pine Hall” and will learn that “23” means the computer called “networking.” It does this via subnet masking (in this case, 255.255.255.0), which isn’t covered in this class.
- **Tongji** Network ranges are: 202.120.176.0 through 202.120.191.255.

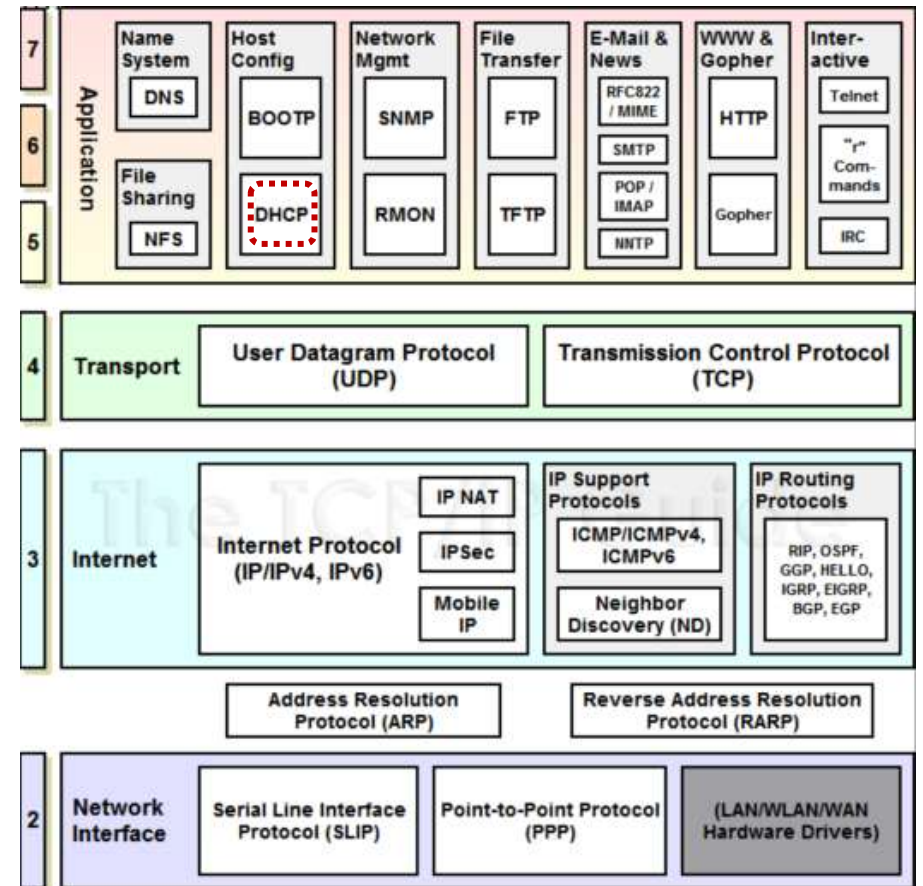
IP: Domain Name Resolution (DNS)

- Since most people find it easier to remember names instead of numbers, IP numbers can and almost always are associated with names.
- Your computer, however, needs a number, so the **Domain Name System** (DNS) exists to make everyone happy.
- A name, such as **www.tongji.edu.cn** tells you the first (or top) level domain (**.cn**, for China) the second level domain (**edu**)... until the actual host's name (**www**).
- If you want the number for a host name **within tongji.edu.cn**, you'll ask one of our DNS servers to give it to you.
- If you need to go **outside tongji.edu.cn**, you'll still ask our servers, but they'll figure out which other server(s) should get your request, send it to them, and will send the reply back to you.



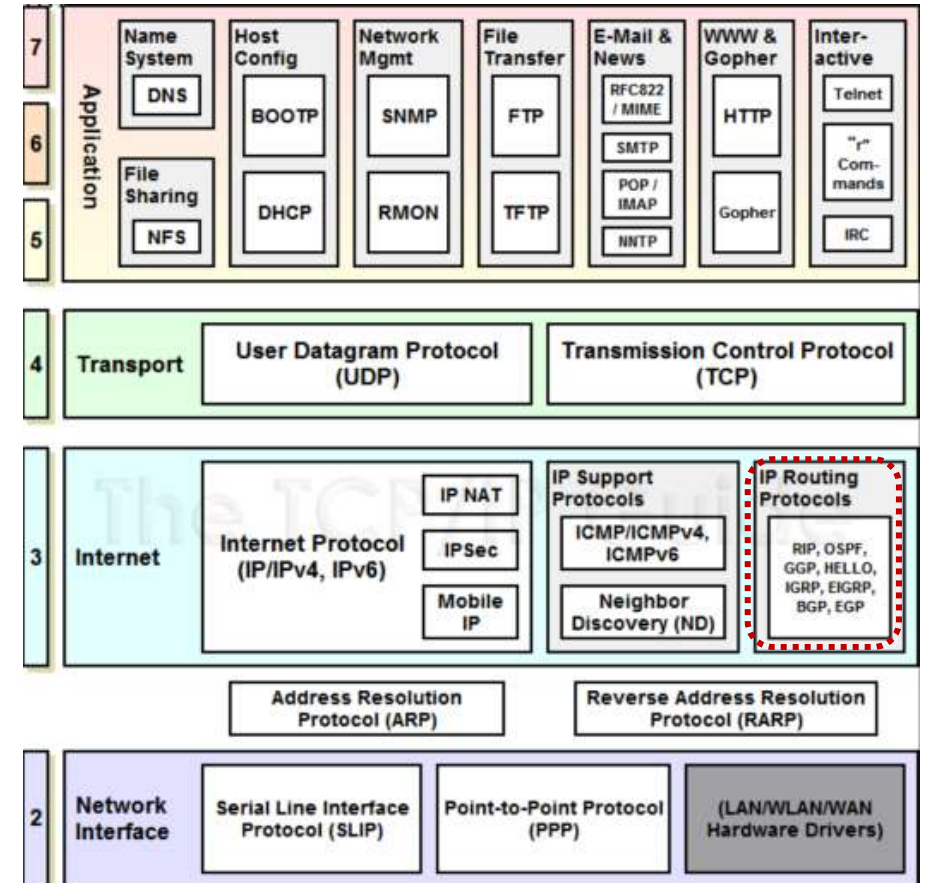
DNS Servers

- Since you need the DNS servers to turn names into numbers, you really need to know the numbers of the DNS servers.
- DHCP (Dynamic Host Configuration Protocol), not covered in this class, can hand this information to you automatically.
- We have others, but these are the most important ones for most campus people.



IP: Routing. “How do you get there from here?”

- ◆ As mentioned before, you can only send ethernet packets out of your ethernet interface, and ethernet packets stay on your local network.
- ◆ You can put an IP (Network layer) packet inside of an ethernet (data layer) packet, but somebody's got to pass it along, and that somebody's a router.
- ◆ Every IP number not on your local network will “belong” to your router in your ARP table.
- ◆ If you want to talk to someone outside your local network, you'll send that ethernet packet to your router's ethernet address and trust that it will work afterwards. It's out of your hands now. You know what's “local” or “not” by the subnet mask.



More routing.

- Routers keep **tables of networks**, often many and often **large**.
- Routers know: 1- Networks directly connected to them (sometimes one or two, sometimes a hundred or more), 2- Networks connected to their “friends and neighbors” and 3- The “default route” for everything else.
- When your ethernet packet arrives at the router, it takes the Network packet (and all its contents), looks at the destination IP number, checks its tables, and sends a new ethernet (or other layer 2) packet (where the “sender” is now the router, not you) out the (hopefully) correct interface.
- That may go to the final host if it's on one of the routers directly connected networks, or to another router, which does the same process, until your packet gets to the router responsible for that local network, who then sends your packet to to the intended host.
- Whether your final destination host is in the next building or on the other side of the world, it works the same way.

It really can't be a networking class without ping and traceroute

- [Ping](#) and [Traceroute](#) are two somewhat useful tools for looking at and learning about your network.
- [Ping](#) sends a small packet to a host which may or may not choose to reply to it, and times how long the packet takes to get back. Lack of a reply doesn't indicate a problem with the host or network.
- [Traceroute](#) asks all routers along the path between you and the destination host if they'd like to respond to you, and times how long each of 3 requests take to get back to you. Some routers may not respond, but may still pass the traceroute packet along, and many hosts will not reply to the traceroute inquiry at all. Lack of a reply doesn't indicate a problem with the host or network.
 - [tracert](#)

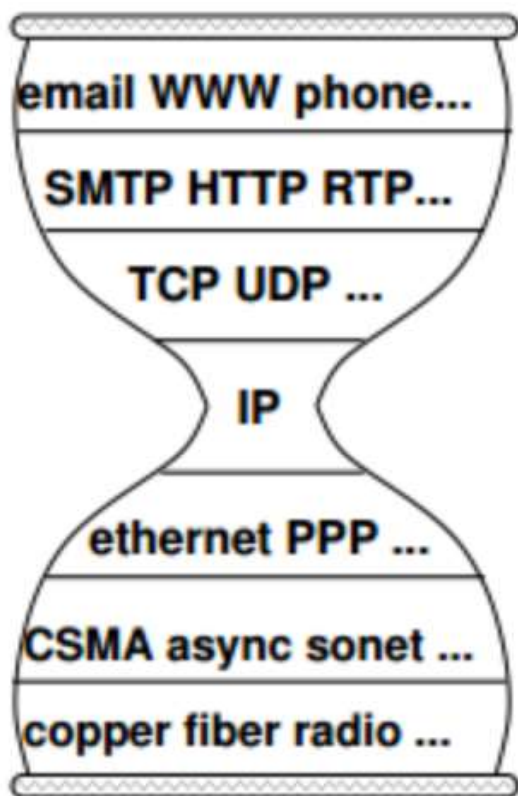
Review.

- What's a network?
- What's a Protocol Stack?
 - What happened to layers 4 through 7?
- What's Cat 5? Cat 5e? What layer are they?
- What's Ethernet? Why do I care?
- What's IP?
- What kind of conversations can my computer have? Who can help it with more conversations?
- What's DNS?
- What's a router do? Why do I care? Does each building have one?

Discussion

参考经典的网络的5层或者7层协议，网络层一直都只有 IP 协议，无论向上有多种应用层协议，向下有多种物理层协议。所以才会说 IP 是网络协议的“细腰”。

- IP层是计算网络的细腰结构



"细腰"设计:

使得底层和高层设计比较简单 → 因为它们之间的通讯全部交给IP完成.
修改IP意味着对因特网的重大改变.

这种设计沿用多年至今, 取得了巨大成功.

但是, 随着社会科技发展, IP不见得很好地能满足人们的需求.

“窄腰”是漏斗, 也是瓶颈

Wireshark工具

- Wireshark是非常流行的网络封包分析软件，小鲨鱼，功能强大
- 截取各种网络封包，显示网络封包的详细信息
- Wireshark是开源软件，运行在Windows和Mac OS上
 - Linux下的抓包工具是 tcpdump。
- 使用wireshark的人必须了解网络协议，否则就看不懂wireshark了。

The world's most popular network protocol analyzer

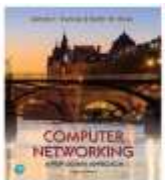
Wireshark常用应用场景

1. 网络管理员会使用wireshark来检查网络问题
2. 软件测试工程师使用wireshark抓包，来分析自己测试的软件
3. 从事socket编程的工程师会用wireshark来调试
4. 运维人员用于日常工作，应急响应等等

总之跟网络相关的东西，都可能会用到wireshark

WIRESHARK LABS

https://gaia.cs.umass.edu/kurose_ross/wireshark.php



Computer Networking: A Top-Down Approach
8th edition

Jim Kurose, Keith Ross
Authors' website

- 通过 wireshark 快速排查客户现场微服务环境部署问题-案例分享
 - <https://developer.aliyun.com/article/1344966>
- 记一次wireshark分析问题实战与总结
 - <https://www.modb.pro/db/225299>

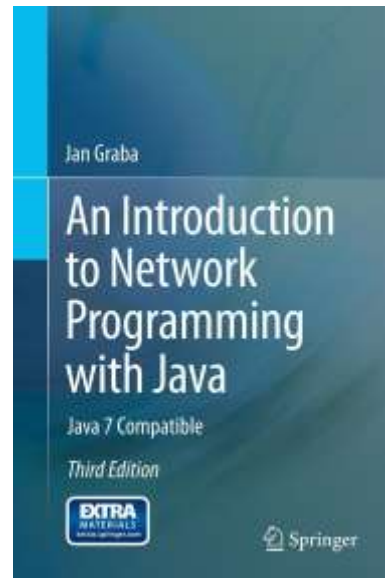
Goals of class

- Basic understanding of common modern networking technology and terminology
- ➔ • Introduction of Network Programming by Java
 - Streams and File Handling
 - Network Programming

<https://docs.oracle.com/javase/tutorial/essential/io/index.html>

An Introduction to Network Programming with Java

<https://link.springer.com/book/10.1007/978-1-4471-5254-5>



Objectives

- After Completing this lecture, the student can understand the following concepts.
- Basic Concepts, Protocols and Terminology
- Streams - Output Streams , Input Streams, Filter Streams
- Buffered Streams, Print Stream
- Readers and Writers
- File Handling - Serial Access Files, File Methods
- Redirection, Command Line Parameters
- Random Access Files
- Scanner class
- Serialization

Basic Concepts

- Client, Server:
- A **server**, provides a service of some kind. This service is provided for clients that connect to the server's host machine specifically for the purpose of accessing the service.
- The machine upon which the server is running is called the host machine.
- A **client** initiates a dialogue with the server.
- Common services provided by servers include the 'serving up' of Web pages (by Web servers).
- To access service provided by Web Servers, the corresponding client programs would be Web Browsers (such as Firefox, Chrome or Internet Explorer).

Client Server Model

- Client –Web Browser
- Server –Web Server



Sockets, Protocol, Ports

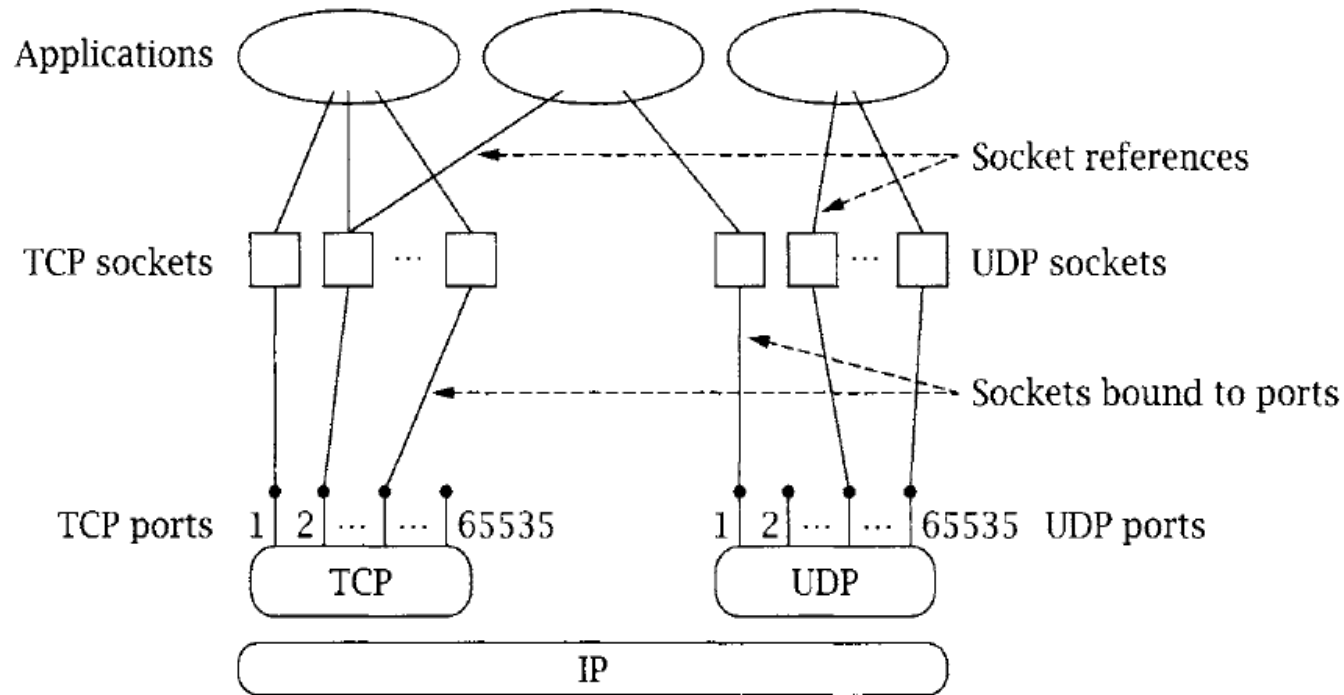
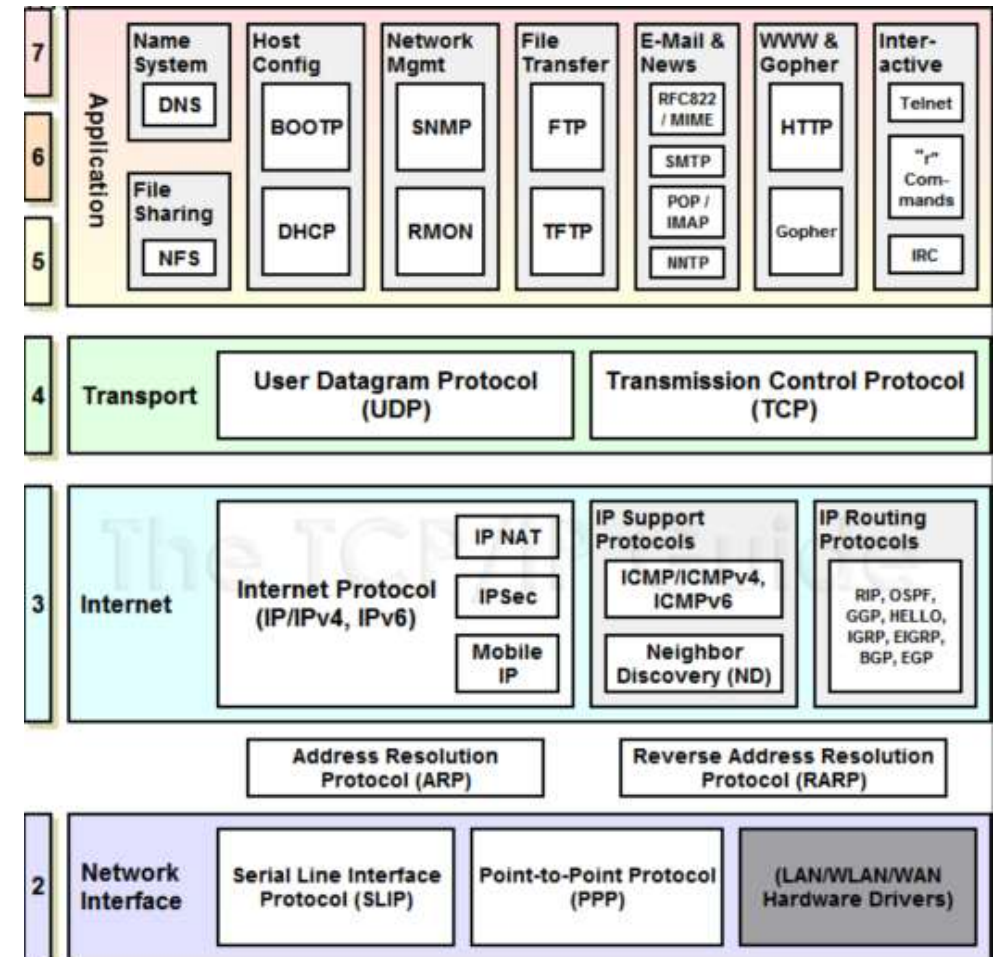


Figure 1.2: Sockets, protocols, and ports.



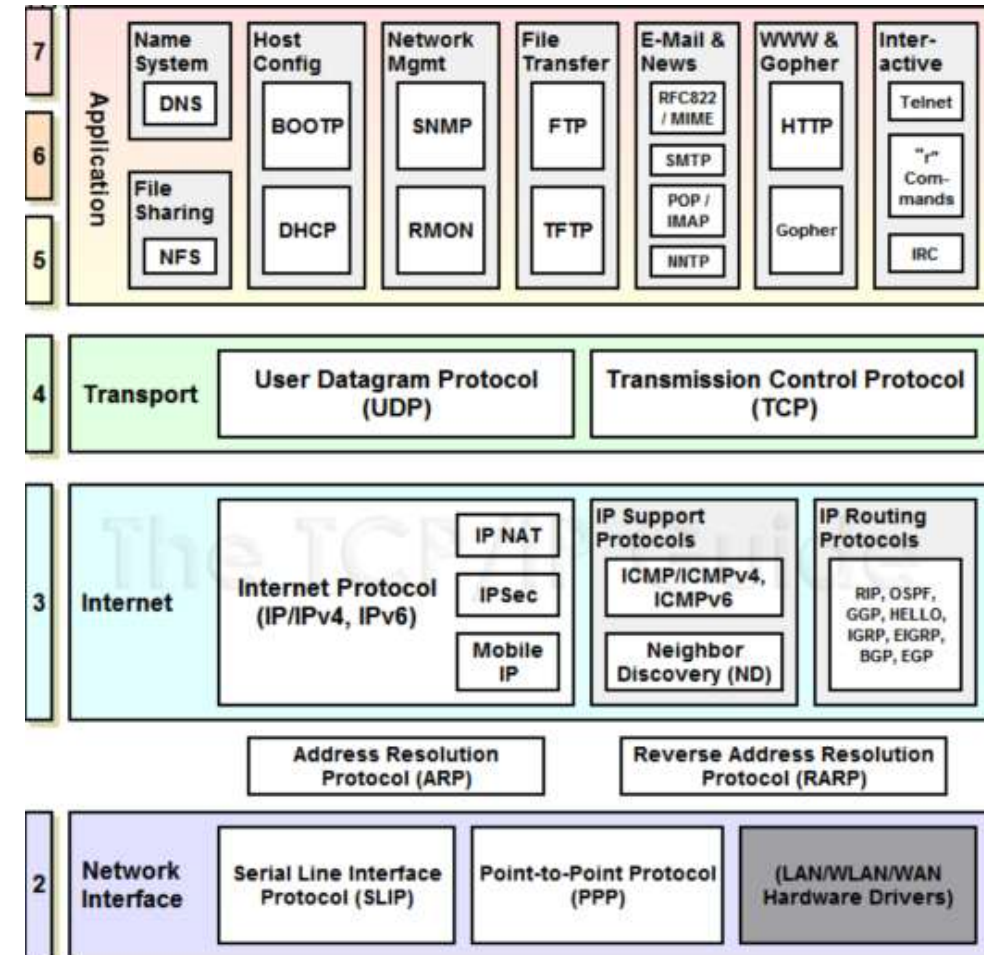
Ports and Sockets

- A **Port** is a **logical connection** to a computer and is identified by a number in the range 1–65535.
- Each **port** may be dedicated to a particular **server/service** .
- Port numbers in the range **1–1023** are normally set aside for the use of specified standard services, often referred to as ‘well-known’ services.
 - For example, port 80 is normally used by Web servers.
- Application programs wishing to use ports for non-standard services should use port numbers in the range of **1024–65535**.
- **Socket** is used to indicate one of the two end-points of a communication link between two processes.
- When a client wishes to make connection to a server, it will create a socket at its end of the communication link.
- Upon receiving the client’s initial request (on a particular port number), the server will create a new socket at its end that will be dedicated to communication with that particular client.

Well-Known Network Services

Table 1.1 Some well-known network services

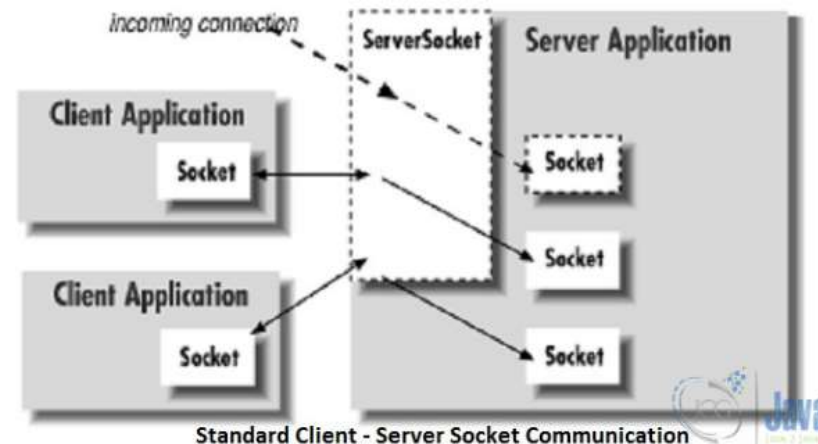
Protocol name	Port number	Nature of service
Echo	7	The server simply echoes the data sent to it. This is useful for testing purposes
Daytime	13	Provides the ASCII representation of the current date and time on the server
FTP-data	20	Transferring files. (FTP uses two ports.)
FTP	21	Sending FTP commands like PUT and GET
Telnet	23	Remote login and command line interaction
SMTP	25	E-mail. (Simple Mail Transfer Protocol.)
HTTP	80	HyperText Transfer Protocol (the World Wide Web protocol)
NNTP	119	Usenet. (Network News Transfer Protocol.)



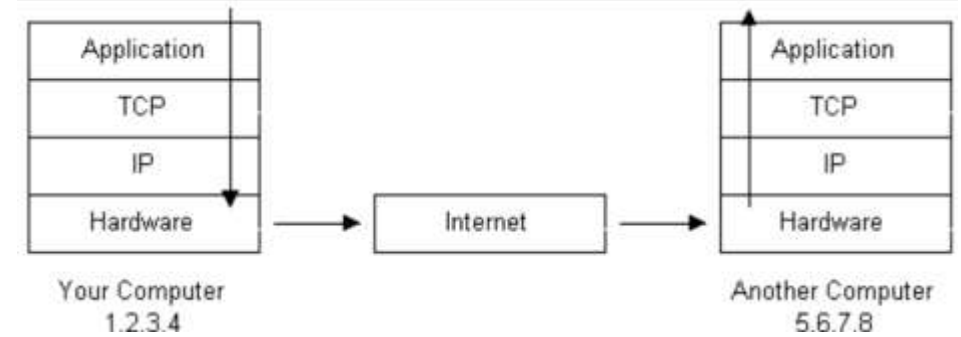
Why Socket ?

- In most applications, there are likely to be **multiple clients** wanting **the same service** at **the same time**.
- A common example of this requirement is that of multiple browsers (quite possibly thousands of them) wanting Web pages from the same server.
- The server, needs some way of distinguishing between clients and keeping their **dialogues** separate from each other. This is achieved via the use of sockets .

Client Server Socket Communication



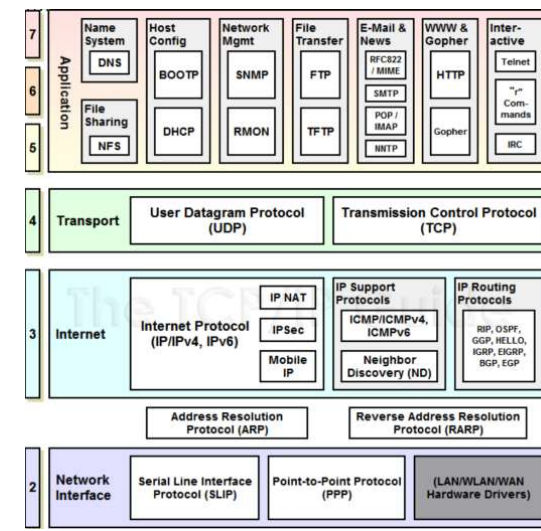
Four Layer Network Model



- When a message is sent by the application layer at one end of the connection, it passes through each of the lower layers. As it does so, each layer adds further protocol data specific to the particular protocol at that level.
- TCP layer breaks up the data packets into TCP segments, adding sequence numbers and checksums.
- IP layer place the TCP segments into IP packets called datagrams and add the routing details.
- The host-to-network layer then converts the digital data into an analog form suitable for transmission over the carrier wire, sends the data and converts it back into digital form at the receiving end.
- At the receiving end, the message travels up through the layers until it reaches the receiving application layer.

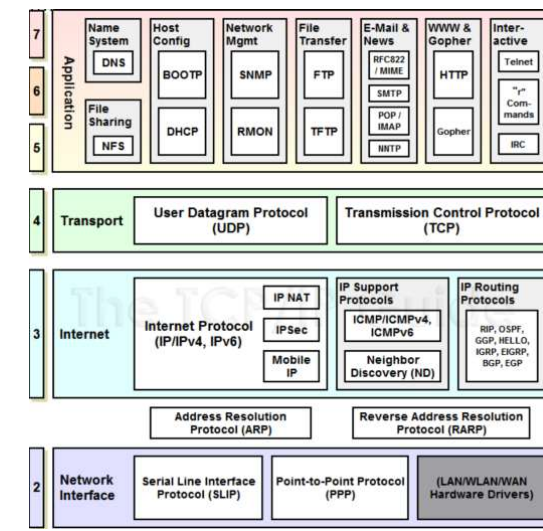
TCP

- TCP is called Transmission Control Protocol.
- TCP is **Reliable** and provides **Acknowledgement**.
- A message sent using TCP is **guaranteed** to be delivered to client.
- TCP also guarantees **order** of message i.e. Message will be delivered to client in the same order, server has sent.
- TCP **retransmits** a packet, if it fails to arrive.
- TCP allows the packets to be **rearranged into their correct sequence** at the receiving end.
- TCP is slow when compared to UDP.
- For some applications like File Transfer, TCP is used.



UDP

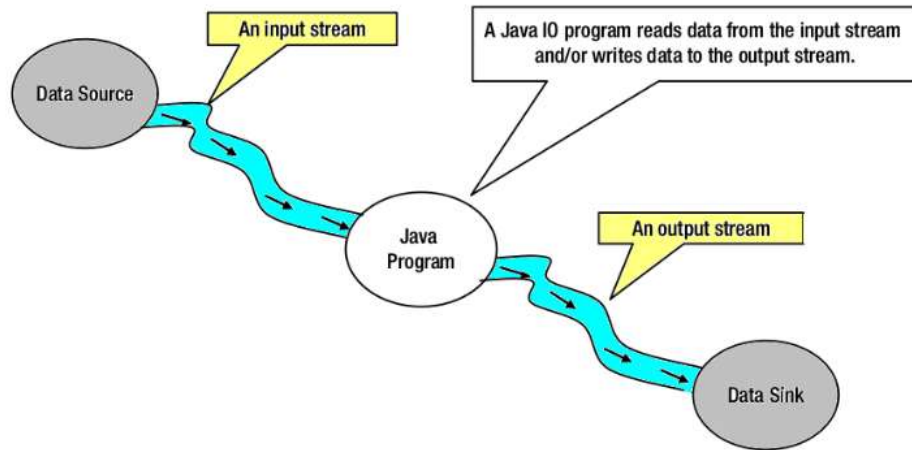
- UDP is called User Datagram Protocol.
- UDP is **Unreliable** and does **not provide Acknowledgement**.
- UDP **doesn't guarantee** that each packet of data will arrive.
- UDP **doesn't guarantee** that packets will be in the **right order**.
- UDP **doesn't re-send** a packet if it fails to arrive.
- UDP **doesn't re-assemble** packets into the correct sequence at the receiving end.
- UDP is **fast** when compared to TCP.
- For some applications including **Streaming** of **Audio** and **Video** Files, UDP is used.



Streams

- I/O in Java is built on streams.
- Input streams read data; Output streams write data.
- Different stream classes, like `java.io.FileInputStream`, read and write particular sources of data.
- Filter streams can be chained to either an input stream or an output stream. Filters can modify the data as it's read or written.
- Readers and writers can be chained to input and output streams to allow programs to read and write text (i.e., characters) rather than bytes.

Input-Output Stream

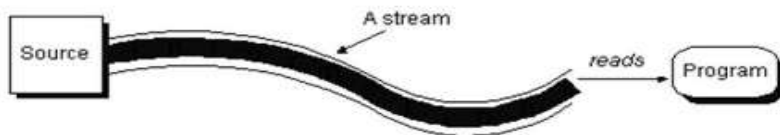


Flow of data using an input/output stream in a Java program

Input Streams vs. Output Stream

- **Java uses streams to facilitate input and output operations**

- **Input streams are used to “read” data**



- **Output streams are used to “write” data**



“Character” Streams
(Reader/Writer)

char
(16-bit)

“Byte” Streams
(InputStream/
OutputStream)

Byte
(8-bit)

Input Stream

Input Source
(keyboard, file,
network, program)

Output Stream

Output Sink
(console, file,
network, program)

Internal Data Formats:

- Text(char): UCS-2
- int, float, double, etc.

External Data Formats:

- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

Output Stream

Java's basic output class is `java.io.OutputStream`:

`public abstract class OutputStream`

Methods of Output Stream:

1. `public abstract void write(int b) throws IOException`
2. `public void write(byte[] data) throws IOException`
3. `public void write(byte[] data, int offset, int length)`
`throws IOException`
4. `public void flush() throws IOException`
5. `public void close() throws IOException`

Subclasses of `OutputStream` use these methods to write data onto particular media. For instance, a `FileOutputStream` uses these methods to write data into a file. A `TelnetOutputStream` uses these methods to write data onto a network connection.

Why Flush Streams ?

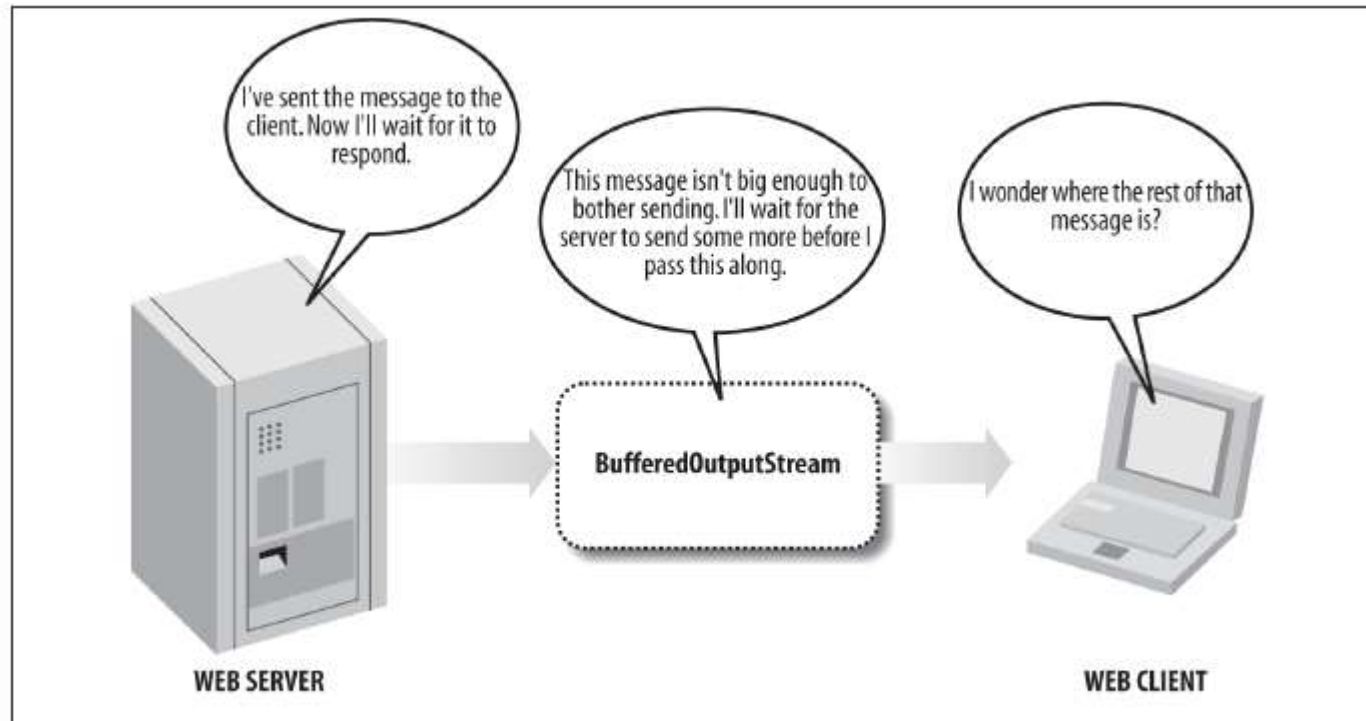


Figure 2-1. Data can get lost if you don't flush your streams

flush() and close() Methods

1. If you are done writing data, it's important to flush the output stream.
2. The **flush()** method force the buffered stream to send its data even if the buffer isn't yet full.
3. You should flush all streams immediately before you close them. Otherwise data left in the buffer when the stream is closed is lost.
4. Failing to flush when you need to can lead to unpredictable, unrepeatable program hangs that are extremely hard to diagnose.

Close() Method:

You should close a stream, by invoking the **close()** method.

Advantages of close() method:

- close() releases any resources associated with the stream, such as file handles or ports.
- If the stream derives from a network connection, closing the stream terminates the connection.
- Once an Output stream is closed, further writes to it throw IOException.
- If you do not close a stream in a long running program, it can leak file handles, network ports and other resources.

Dispose Pattern

- Close the stream in a finally block.
- Declare the stream variable outside the try block.
- Initialize the stream variable inside the try block.
- To avoid NullPointerExceptions, check whether the stream variable is null before closing it.
- Ignore or log any exceptions that occur while closing the stream.
- The above five steps are called dispose pattern; and it's common for any object that needs to be cleaned up before it's garbage collected. You'll see it used not just for streams, but also for sockets, channels, JDBC connections, etc.

Dispose Pattern Example

```
OutputStream out = null; // declare stream variable
try {
    out = new FileOutputStream("/tmp/data.txt"); // initialise variable
    // work with the output stream...
}
catch (IOException ex) {
    System.err.println(ex.getMessage());
} finally {
    if (out != null)
    {
        try {
            out.close();
        } catch (IOException ex) {
            // ignore
        }
    }
}
```

Input Stream

Java's basic input class is `java.io.InputStream`:

public abstract class InputStream

Methods of Input Stream:

1. `public abstract int read() throws IOException`
2. `public int read(byte[] input) throws IOException.`
3. `public int read(byte[] input, int offset, int length) throws IOException`
4. `public long skip(long n) throws IOException`
5. `public int available() throws IOException`
6. `public void close() throws IOException`

Concrete subclasses of `InputStream` use these methods to read data from particular media. For instance, a `FileInputStream` reads data from a file. A `TelnetInputStream` reads data from a network connection.

read(), available(), skip(), close() methods

- The basic method of InputStream is the noargs read() method. This method reads a single byte of data from the input stream's source and returns it as an int from 0 to 255.
- End of stream is signified by returning -1. The read() method waits and blocks execution of any code that follows it until a byte of data is available and ready to be read.
- All three read() methods return -1 to signal the end of the stream.

available() Method:

- If you do not want to wait until all the bytes you need are immediately available, you can use the available() method to determine how many bytes can be read without blocking. This returns the minimum number of bytes you can read. On end of stream, available() returns 0.

skip() Method:

- On rare occasions, you may want to skip over data without reading it. The skip() method accomplishes this task.

close() Method:

- You should close an input stream, by invoking the close() method. This releases any resources associated with the stream, such as file handles or ports. Once an input stream is closed, further read from it throw IOExceptions.

Filter Stream

- The filters come in two versions: the **filter streams**, and the **readers** and **writers**.
- The filter streams still work primarily with raw data as bytes: for instance, by compressing the data or interpreting it as binary numbers. The readers and writers handle the special case of text in a variety of encodings such as UTF-8 and ISO 8859-1.
- Every filter output stream has the same `write()`, `close()`, and `flush()` methods as `java.io.OutputStream`.
- Every filter input stream has the same `read()`, `close()`, and `available()` methods as `java.io.InputStream`.
- Filters are connected to streams by their constructor.
- A `BufferedInputStream` object `bin` is created by passing `fin` as an argument to the `BufferedInputStream` constructor:

```
FileInputStream fin = new FileInputStream("data.txt");  
BufferedInputStream bin = new BufferedInputStream(fin);
```

BufferedOutputStream 默认的缓冲区大小为 **8192 字节** (即 **8 KB**)

Buffered Streams

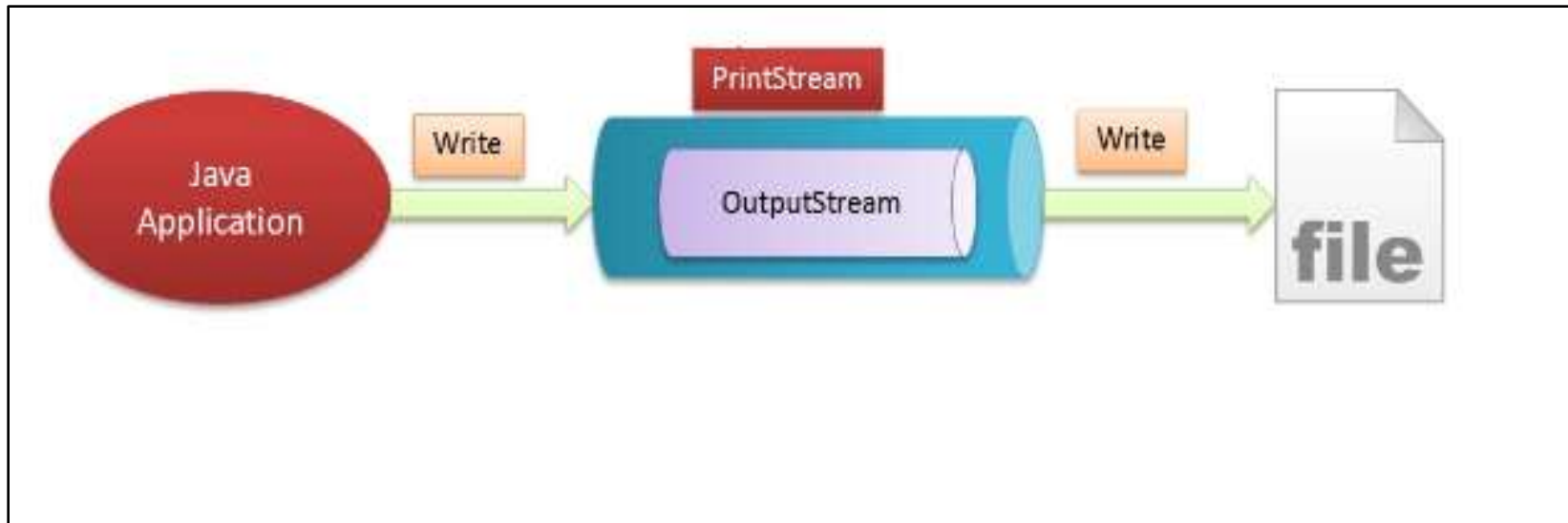
如果需要调整缓冲区大小, 可以在创建 BufferedOutputStream 对象时通过构造方法指定

```
FileOutputStream fos = new  
FileOutputStream("output.txt"); BufferedOutputStream bos  
= new BufferedOutputStream(fos, 16384) //缓冲区大小设置为 16 KB
```

- The **BufferedOutputStream** class stores written data in a buffer (a protected byte array field named buf) until **the buffer is full or the stream is flushed**. Then it writes the data onto the underlying output stream **all at once**. A single write of many bytes is almost always **much faster** than **many small writes**.
- BufferedOutputStream has two constructors.
 - i) **public BufferedOutputStream(OutputStream out)**
 - ii) **public BufferedOutputStream(OutputStream out, int bufferSize)**
- The **BufferedInputStream** class also has a protected byte array named buf that serves as a buffer. When one of the stream's read() methods is called, it first tries to get the requested data from the buffer. Only when the buffer runs out of data does the stream read from the underlying source.
- BufferedInputStream has two constructors.
 - i) **public BufferedInputStream(InputStream in)**
 - ii) **public BufferedInputStream(InputStream in, int bufferSize)**

Print Stream

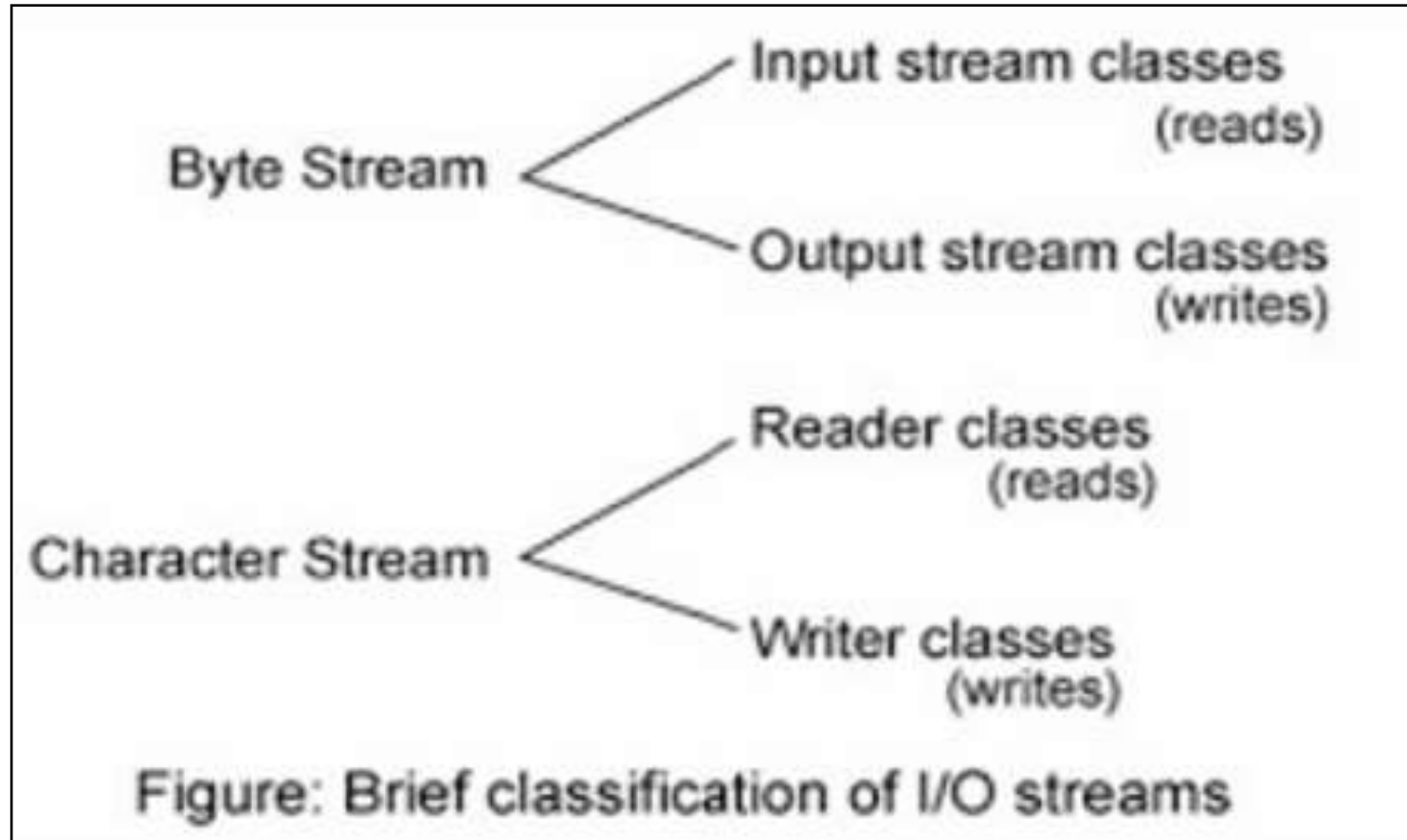
- The PrintStream class is the first filter output stream most programmers encounter because System.out is a PrintStream. However, other output streams can also be chained to print streams, using these two constructors:
 - i) `public PrintStream(OutputStream out)`
 - ii) `public PrintStream(OutputStream out, boolean autoFlush)`



Print Stream (cont..)

- PrintStream has 9 overloaded print() methods and 10 overloaded println() methods:
 - **public void print(boolean b)**
 - **public void print(char c)**
 - **public void print(int i)**
 - **public void print(long l)**
 - **public void print(float f)**
 - **public void print(double d)**
 - **public void print(char[] text)**
 - **public void print(String s)**
 - **public void print(Object o)**
 - **public void println()**
 - **public void println(boolean b)**
 - **public void println(char c)**
 - **public void println(int i)**
 - **public void println(long l)**
 - **public void println(float f)**
 - **public void println(double d)**
 - **public void println(char[] text)**
 - **public void println(String s)**
 - **public void println(Object o)**

Input-Output Stream (cont..)



Readers and Writers

1. The `java.io.Reader` class specifies the API by which characters are read.
2. The `java.io.Writer` class specifies the API by which characters are written.
3. Wherever input and output streams use bytes, readers and writers use Unicode characters.
4. The most important concrete subclasses of `Reader` and `Writer` are the `InputStreamReader` and the `OutputStreamWriter` classes.
5. The `java.io` package provides several raw reader and writer classes that read characters without directly requiring an underlying input stream, including:
 - a) `FileReader`
 - b) `FileWriter`
 - c) `StringReader`
 - d) `StringWriter`

Writers

- The `Writer` class mirrors the `java.io.OutputStream` class. It is abstract and has two protected constructors. Like `OutputStream`, the `Writer` class is never used directly; instead, it is used polymorphically, through one of its subclasses. It has five `write()` methods, a `flush()` and a `close()` method

Methods of Writer Class:

- `protected Writer()`
- `protected Writer(Object lock)`
- `public abstract void write(char[] text, int offset, int length) throws IOException`
- `public void write(int c) throws IOException`
- `public void write(char[] text) throws IOException`
- `public void write(String s) throws IOException`
- `public void write(String s, int offset, int length) throws IOException`
- `public abstract void flush() throws IOException`
- `public abstract void close() throws IOException`

OutputStreamWriter and Readers

- OutputStreamWriter is the most important concrete subclass of Writer. An OutputStreamWriter receives characters from a Java program. It converts these into bytes according to a specified encoding and writes them onto an underlying output stream. Its constructor specifies the output stream to write to and the encoding to use.

**public OutputStreamWriter(OutputStream out, String encoding) throws
UnsupportedEncodingException**

Readers:

- The Reader class mirrors the java.io.InputStream class. It is abstract with two protected constructors. Like InputStream and Writer, the Reader class is never used directly, only through one of its subclasses.

Methods of Reader Class:

- protected Reader()
- protected Reader(Object lock)
- public abstract int read(char[] text, int offset, int length) throws IOException

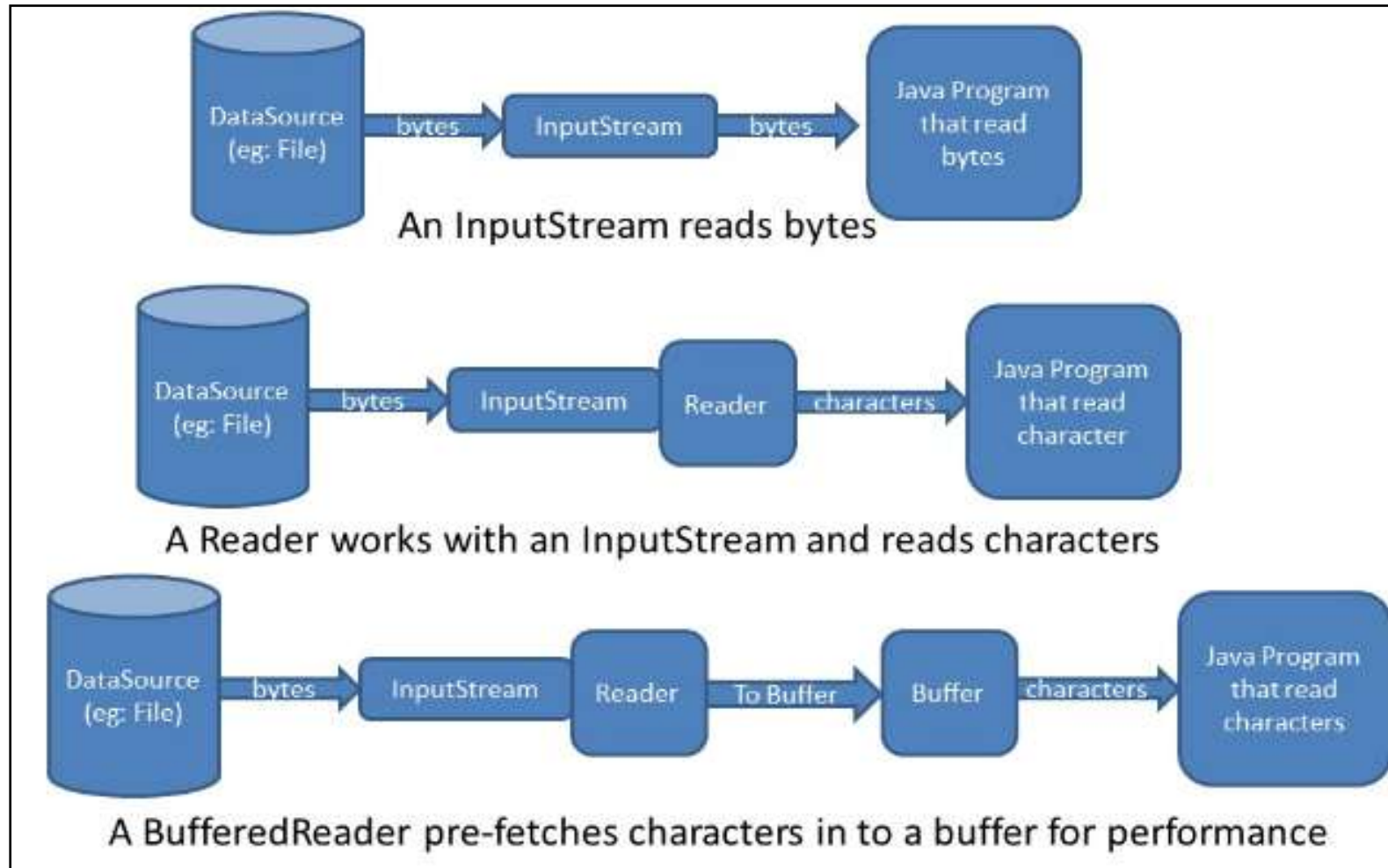
Readers (Cont..)

- `public int read()` throws `IOException`
- `public int read(char[] text)` throws `IOException`
- `public long skip(long n)` throws `IOException`
- `public abstract void close()` throws `IOException`

InputStreamReader:

- `InputStreamReader` is the most important concrete subclass of `Reader`. An `InputStreamReader` reads bytes from an underlying input stream such as a `FileInputStream`. It converts these into characters according to a specified encoding and returns them. The constructor specifies the input stream to read from and the encoding to use:
 - **`public InputStreamReader(InputStream in)`**
 - **`public InputStreamReader(InputStream in, String encoding)`
throws `UnsupportedEncodingException`**

Stream/Reader/BufferedReader



BufferedReader and BufferedWriter

1. The `BufferedReader` and `BufferedWriter` classes are the character-based equivalents of the byte-oriented `BufferedInputStream` and `BufferedOutputStream` classes.
2. The `BufferedInputStream` and `BufferedOutputStream` use an internal array of bytes as a buffer, while the `BufferedReader` and `BufferedWriter` use an internal array of characters.
3. When a program reads from a **BufferedReader**, text is taken from the buffer rather than directly from the underlying input stream or other text source.
4. When a program writes to a **BufferedWriter**, the text is placed in the buffer. The text is moved to the underlying output stream or other target only when the buffer fills up or when the writer is explicitly flushed.
5. `BufferedReader` and `BufferedWriter` have the usual methods associated with readers and writers, like `read()`, `write()`, `close()`.

Constructors of Buffered Reader/Writer

- i) `public BufferedReader(Reader in, int bufferSize)`
- ii) `public BufferedReader(Reader in)`
- iii) `public BufferedWriter(Writer out)`
- iv) `public BufferedWriter(Writer out, int bufferSize)`

- The `BufferedReader` class also has a **`readLine()`** method that reads a single line of text and returns it as a string:

`public String readLine() throws IOException`

- The `BufferedWriter()` class adds one new method not included in its superclass, called **`newLine()`**, also geared toward writing lines:

`public void newLine() throws IOException`

- This method inserts a platform-dependent line-separator string into the output.

PrintWriter

- The PrintWriter class has an almost identical collection of methods to PrintStream.

Methods of PrintWriter class:

- public PrintWriter(Writer out)
- public PrintWriter(Writer out, boolean autoFlush)
- public PrintWriter(OutputStream out)
- public PrintWriter(OutputStream out, boolean autoFlush)
- public void flush()
- public void close()
- public boolean checkError()
- public void write(int c)
- public void write(char[] text, int offset, int length)
- public void write(char[] text)
- public void write(String s, int offset, int length)
- public void write(String s)

PrintWriter (Cont..)

- `public void print(boolean b)`
- `public void print(char c)`
- `public void print(int i)`
- `public void print(long l)`
- `public void print(float f)`
- `public void print(double d)`
- `public void print(char[] text)`
- `public void print(String s)`
- `public void print(Object o)`
- `public void println()`
- `public void println(boolean b)`
- `public void println(char c)`
- `public void println(int i)`
- `public void println(long l)`
- `public void println(float f)`
- `public void println(double d)`
- `public void println(char[] text)`
- `public void println(String s)`
- `public void println(Object o)`

Serial Access Files

1. **Serial access files** are files in which data is stored in physically adjacent locations, often in no particular logical order, with each new item of data being added to the end of the file.
2. The internal structure of a serial file can be either binary or text.
3. A text file requires a `FileReader` object for input and a `FileWriter` object for output.
4. Constructors for these objects are overloaded and may take either of the following arguments.
 - i) a `String` (holding the file name)
 - ii) a `File` object (constructed from file name)

Examples:

File Objects are created first

- a) `File inFile = new File("input.txt");`
- b) `File outFile = new File("output.txt");`

Now, `FileReader` and `FileWriter` objects are created

- c) `FileReader in = new FileReader(inFile);`
- d) `FileWriter out = new FileWriter(outFile);`

Serial Access Files (cont..)

5. Class **File** is contained within package **java.io** , so this package should be imported into any file-handling program.
6. If a program needs both to read from and write to a file, it must declare both a **FileReader** object and a **FileWriter** object. These objects are in package java.io.

Problem with FileReader and FileWriter:

FileReader and FileWriter do not provide sufficient functionality or flexibility for reading and writing data from and to files. To acquire this functionality, we need to :

- i) Wrap a **BufferedReader** object around a **FileReader** object in order to read from a file.
- ii) Wrap a **PrintWriter** object around a **FileWriter** Object in order to write to the file.

Example:

```
BufferedReader input = new BufferedReader(new FileReader("in.txt"));
```

```
PrintWriter output = new PrintWriter(new FileWriter("out.txt"));
```

NOTE: Use methods **readLine()** from class **BufferedReader** and **print(), println()** from class **PrintWriter**. When the processing of a file has been completed, the file should be closed via the **close()** method. Closing the file causes the output buffer to be flushed and any data in the buffer to be written to disc.

Serial Access Files (cont..)

NOTE: When reading numeric values, use the parse methods (parseInt, parseDouble, parseFloat) if any arithmetic or formatting is to be performed on the data.

Program-4.1

NOTE: There is no 'append' method for a serial file in Java. If the file already existed, its initial contents will have been overwritten. If you need to add data to the contents of an existing file, you need to use a FileWriter constructor.

Public FileWriter(String <fileName>, boolean <append>)

Example:

```
FileWriter addFile = new FileWriter("data.txt", true);
```

Program-4.2

Program-4.3

Advantages-Drawbacks of Serial Access Files

Advantages:

1. Simple to handle.
2. Widely used in small scale applications.
3. Used to provide temporary storage in large scale applications.

Drawbacks:

1. Can't go directly to a specific record.
2. To access a **particular record**, it is necessary to **read past all the preceding records**.
3. It is **not possible to add or modify records** within an existing file. (The whole file would have to be re-created).

File Methods

1. **boolean canRead()** - Returns true if file is readable and false otherwise.
2. **boolean canWrite()** - Returns true if file is writeable and false otherwise.
3. **boolean delete()** - Deletes file and returns true/false for success/failure.
4. **boolean exists()** - Returns true if file exists and false otherwise.
5. **String getName()** - Returns name of file.
6. **boolean isDirectory()** - Returns true if object is a directory/folder and false otherwise.(Note that File objects can refer to ordinary files or to directories.)
7. **boolean isFile()** - Returns true if object is a file and false otherwise.
8. **long length()** - Returns length of file in bytes.
9. **String[] list()** - If object is a directory, array holding names of files within directory is returned.
10. **File[] listFiles()** - Similar to previous method, but returns array of File objects.
11. **boolean mkdir()** - Creates directory with name of current File object. Return value indicates success/failure.

Program – 4.4

Scanner

- Before Java SE 5, it was necessary to wrap a `BufferedReader` object around a `FileReader` object in order to read from a file. Likewise, it was necessary to wrap a `PrintWriter` object around a `FileWriter` object in order to write to the file. Now we can wrap a **Scanner object around a File object for input** and a **PrintWriter object around a File object for output**.
- Examples:
 - (i) `Scanner input = new Scanner(new File("inFile.txt"));`
 - (ii) `PrintWriter output = new PrintWriter(new File("outFile.txt"));`
- We can then make use of methods `next`, `nextLine`, `nextInt`, `nextFloat`, ... for input and methods `print` and `println` for output.
- Examples (using objects input and output, as declared above)
 - (i) `String item = input.next();`
 - (ii) `output.println("Test output");`
 - (iii) `int number = input.nextInt();`

Scanner (Cont..)

```
(iv) File inFile = new File("inFile.txt");
    Scanner input = new Scanner(inFile);
(v) File outFile = new File("outFile.txt");
    PrintWriter output = new PrintWriter(outFile);
```

Note: As of Java 5, we must not attempt to read beyond the end-of-file if we wish to avoid the generation of a `NoSuchElementException`. Instead, we have to check ahead to see whether there is more data to be read. This is done by making use of the Scanner class **hasNext** method, which returns a Boolean result indicating whether or not there is any more data.

Example:

```
while (input.hasNext())
{
    .....
    .....
}
```

Redirection

1. By default, the standard input stream “System.in” is associated with the keyboard, while the standard output stream “System.out” is associated with the VDU. If, however, we wish input to come from some other source (such as a text file) or we wish output to go to somewhere other than the VDU screen, then we can redirect the input/output.
2. We use the symbol ‘ < ’ to specify the new source of input and the symbol ‘ > ’ to specify the new output destination.

Example:

```
java ReadData < payroll.txt
```

```
java WriteData > results.txt
```

3. We can use redirection of both input and output with the same program, as the example below shows.

```
java ProcessData < readings.txt > results.txt
```

4. For program ‘ProcessData(.class)’ above, all file input statements will read from file ‘readings.txt’, while all print and println will send output to file ‘results.txt’.

Command Line Parameters

1. When entering the java command into a command window, it is possible to supply values in addition to the name of the program to be executed. These values are called **command line parameters** and are values that the program may make use of.
2. Such values are received by method *main* as an array of *Strings*. If this argument is called *arg*, then the elements may be referred to as *arg[0]*, *arg[1]*, *arg[2]*, etc.
3. Program – 4.5

Random access files

1. They are also called as Direct Access Files.
2. To create a random access file in Java, we create a **RandomAccessFile** object. The constructor takes two arguments:
 - (i) **a string or File object identifying the file.**
 - (ii) **a string specifying the file's access mode** (The mode may be either "**r**" (for read-only access) or "**rw**" (for read-and-write access)).

Example:

```
RandomAccessFile ranFile = new RandomAccessFile("accounts.dat","rw");
```

Note:

1. Before reading or writing a record, it is necessary to position the file pointer. We do this by calling method **seek**, which requires a single argument **specifying the byte position** within the file. The first byte in a file is byte **0**.

Example : `ranFile.seek(500);` //Move to byte 500 (the 501st byte).

2. In order to move to the correct position for a particular record, we need to know two things: (i) the size of records on the file; (ii) the algorithm for calculating the appropriate position.
3. Class **RandomAccessFile** provides the following methods:
 - (i)readInt, (ii) readLong, (iii) readFloat, (iv) readDouble (v) writeInt,
 - (vi) writeLong, (vii) writeFloat, (viii) writeDouble

Program – 4.6, 4.7

Advantages-Disadvantages of Random Access Files

Advantage:

1. Speed and flexibility.
2. Can directly access a specific record.
3. Can update a specific record.

Disadvantage:

1. All the records must be of **same length**.
2. A given string field must be of the same length for all records.
3. Numeric data is not in human-readable form.

Serialization

1. Objects of any class that implements the **Serializable** interface may be transferred to and from **disc files** as **whole objects**, with no need for **decomposition** of those objects.
2. Class **ObjectOutputStream** is used to save entire objects directly to disc.
3. Class **ObjectInputStream** is used to read them back from disc.
4. For output, we wrap an object of class **ObjectOutputStream** around an object of class **FileOutputStream**, which itself is wrapped around a **File object or file name**.

Example: `ObjectOutputStream ostream = new ObjectOutputStream (new
FileOutputStream("personnel.dat"));`

5. For input, we wrap an **ObjectInputStream** object around a **FileInputStream** object, which in turn is wrapped around a **File object or file name**.

Example: `ObjectInputStream istream = new ObjectInputStream(new FileInputStream("personnel.dat"));`

6. Methods **writeObject** and **readObject** are then used for the actual output and input respectively. Any objects read back from file must be **typecast** into their original class.

Example: `Personnel person = (Personnel)istream.readObject();`

Goals of class

- Basic understanding of common modern networking technology and terminology
- Introduction of Network Programming by Java
 - Streams and File Handling
 - ➔ • Network Programming

<https://docs.oracle.com/javase/tutorial/essential/io/index.html>

Next Week

网络编程

- Socket, Client/Server

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

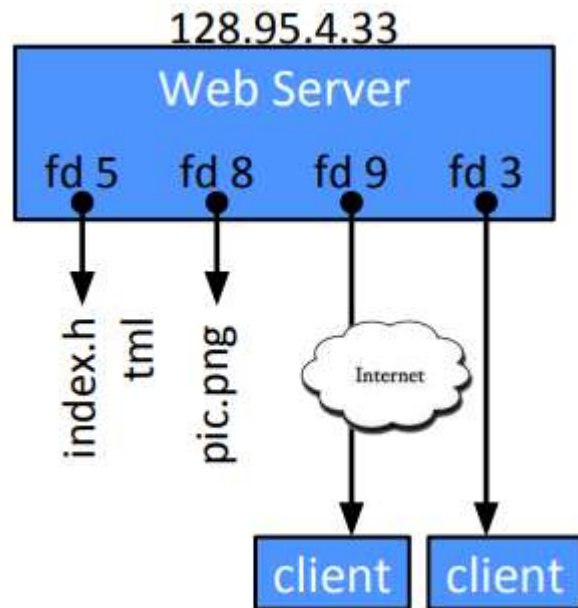
Files and File Descriptors

- 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.

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)

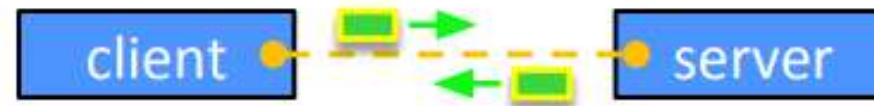
Stream Sockets

- 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:



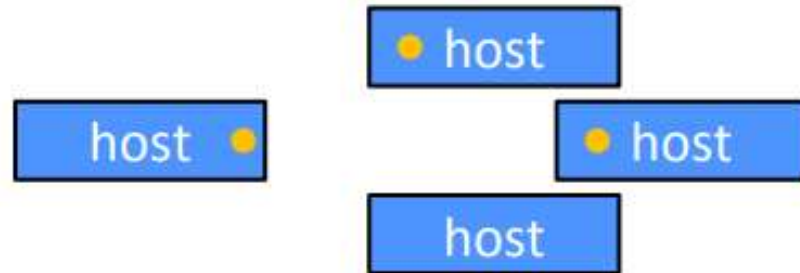
3) Close connection:



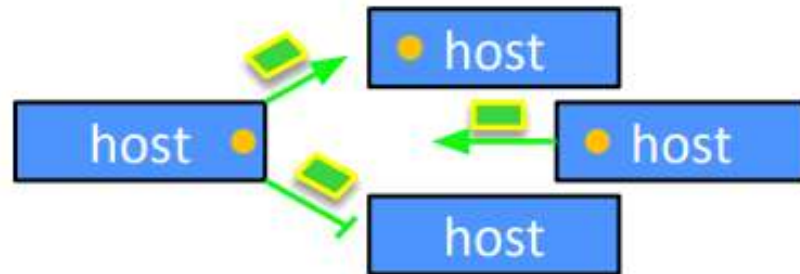
Datagram Sockets

- 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);  
  
}
```

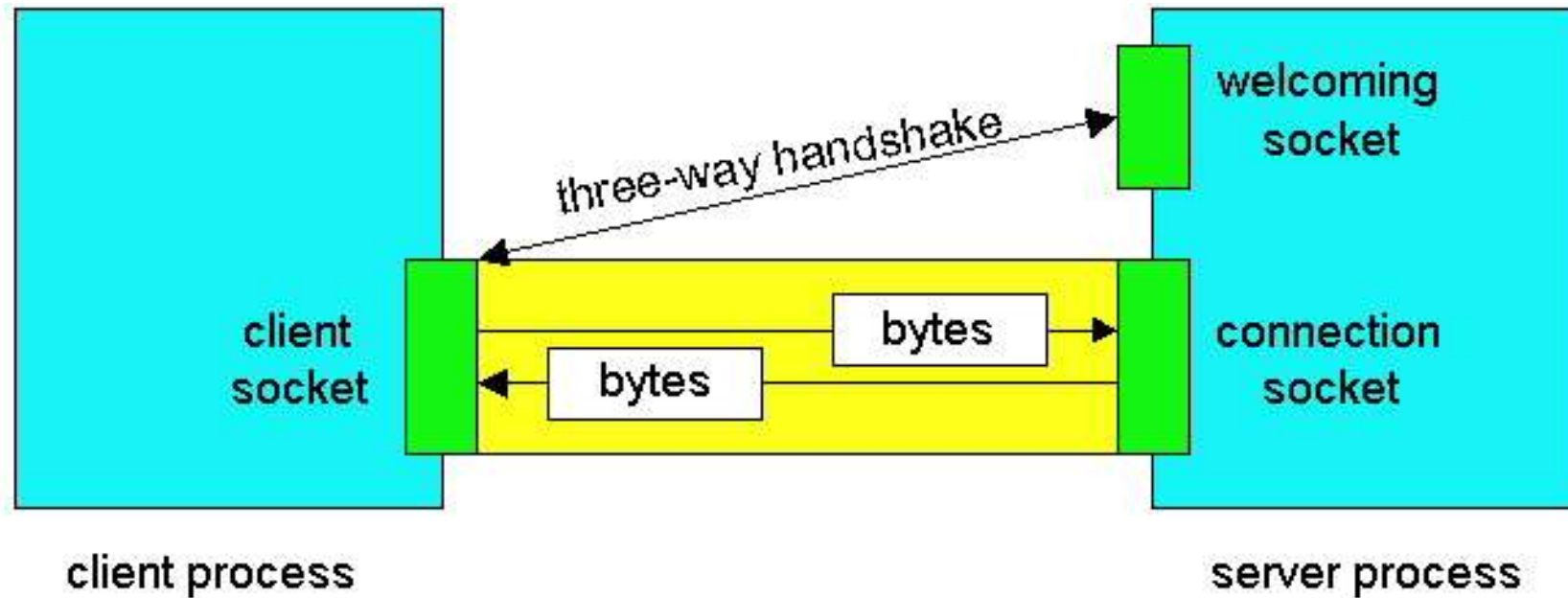
Socket class

- 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 getAddress();
```

```
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();

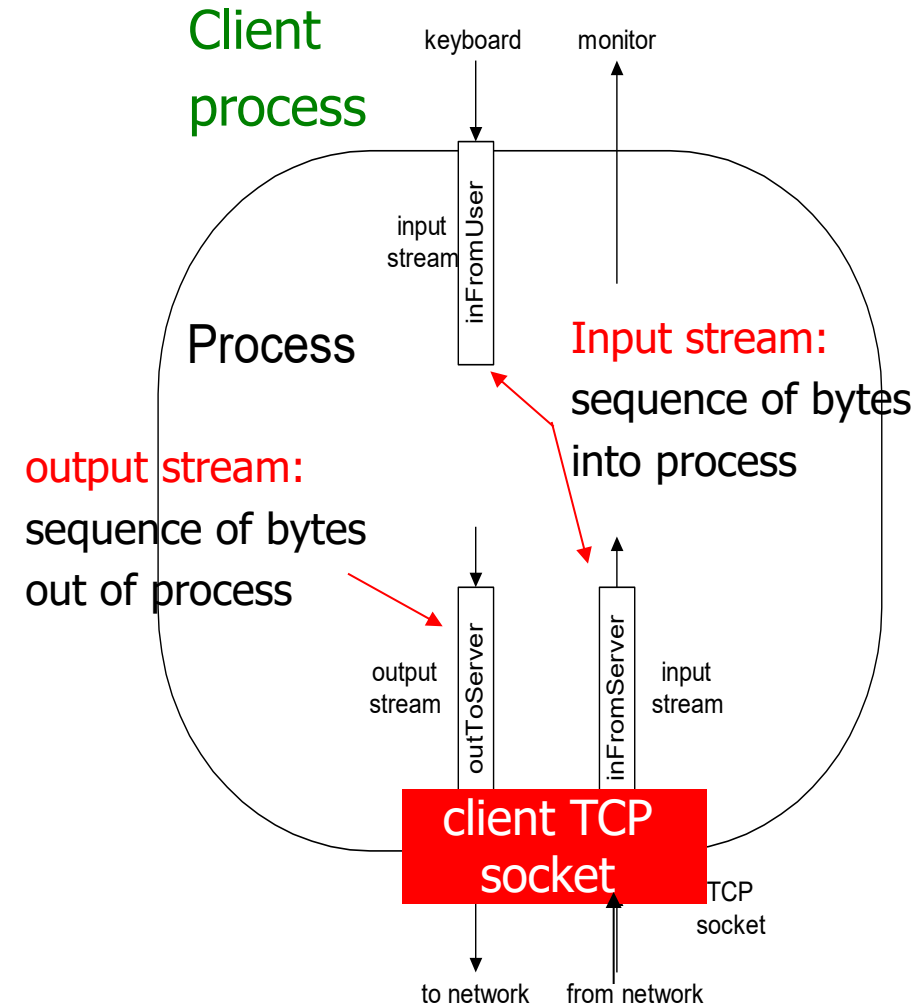
InetAddress getInetAddress();

int getLocalPort();

throw IOException, SecurityException

Socket programming with TCP

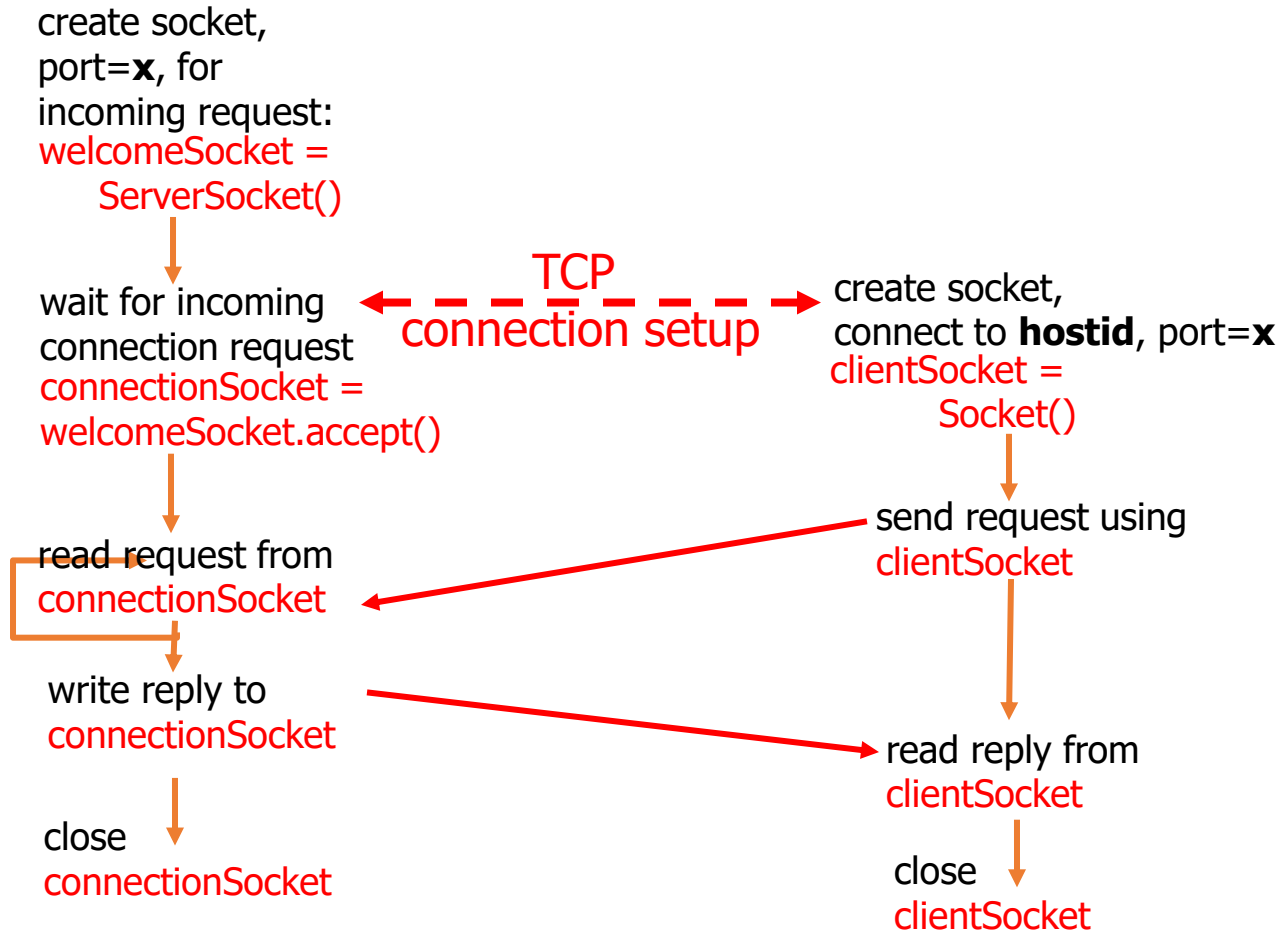
- **Example client-server app:**
 - client reads line from standard input (**inFromUser stream**), sends to server via socket (**outToServer stream**)
 - server reads line from socket
 - server converts line to uppercase, sends back to client
 - client reads, prints modified line from socket (**inFromServer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



Client/server socket interaction: TCP

Server (running on **hostid**)

Client



TCPClient.java

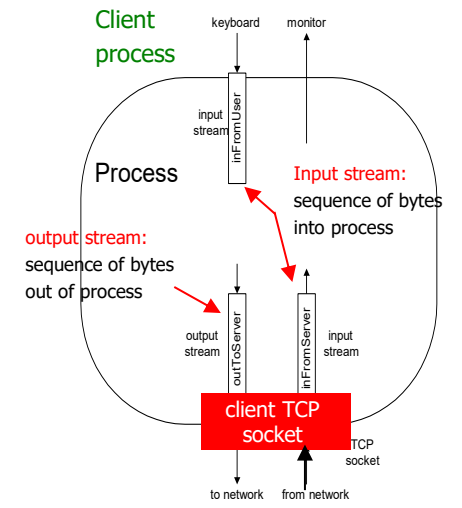
```
import java.io.*;
import java.net.*;
```

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

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

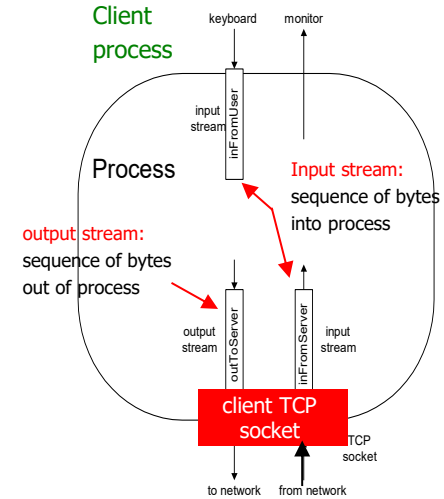
```
Socket clientSocket = new Socket("hostname", 6789);
```

```
        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();  
}  
}
```



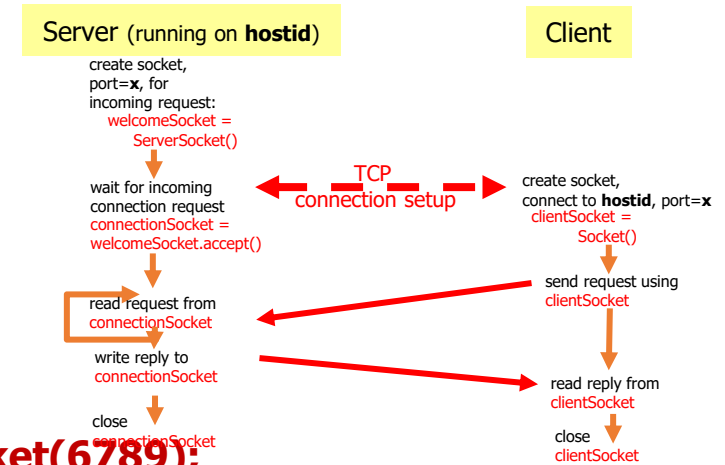
TCPServer.java

```
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

```
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();}
    }
}
```

```
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);
```

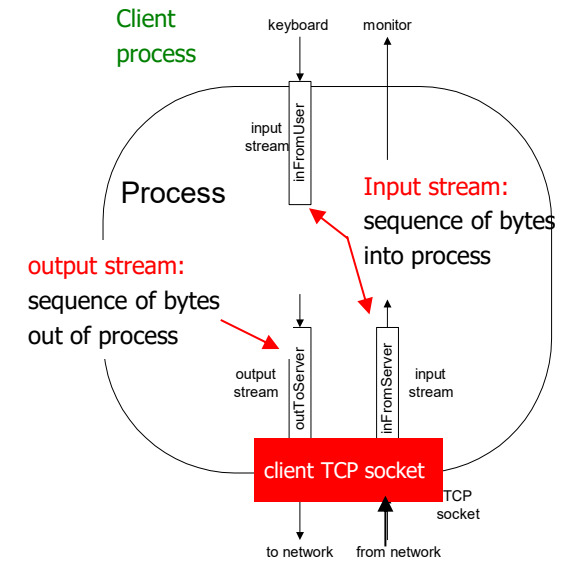
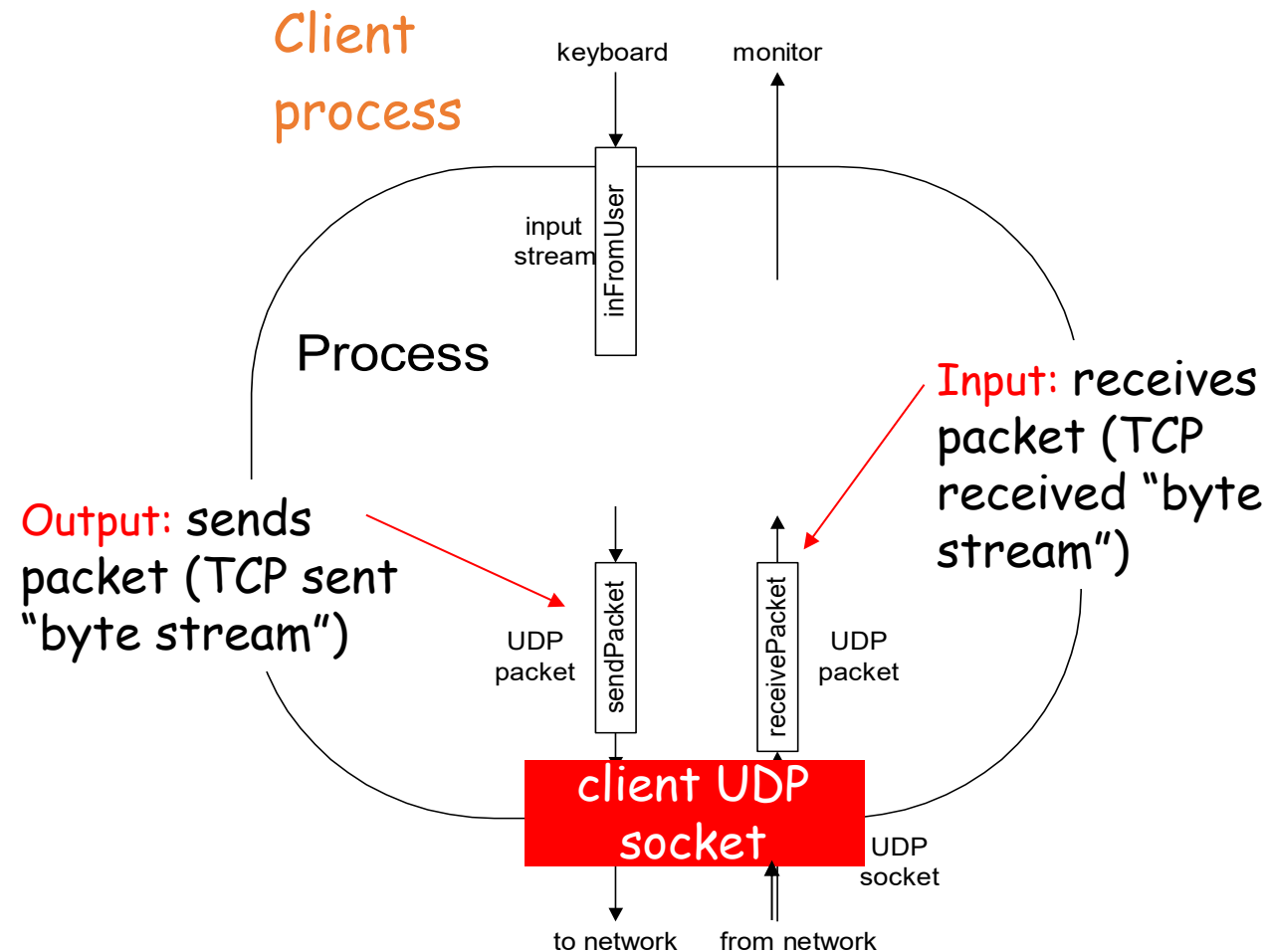
```
void setAddress(InetAddress a);
```

```
void setPort(int port);
```

```
InetAddress getAddress();
```

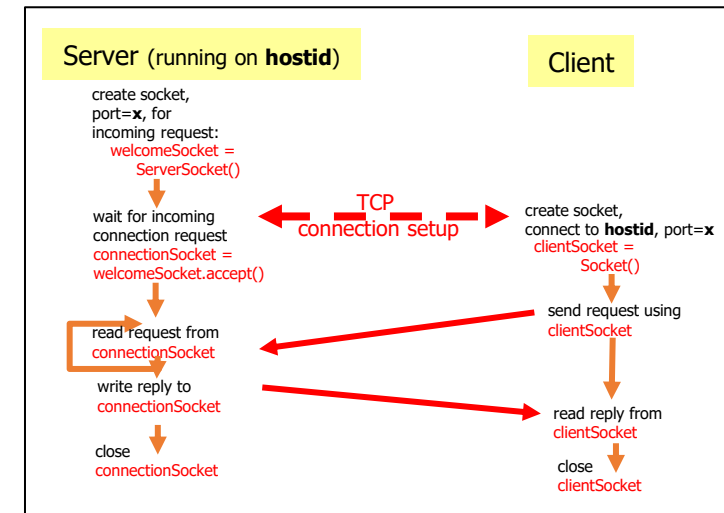
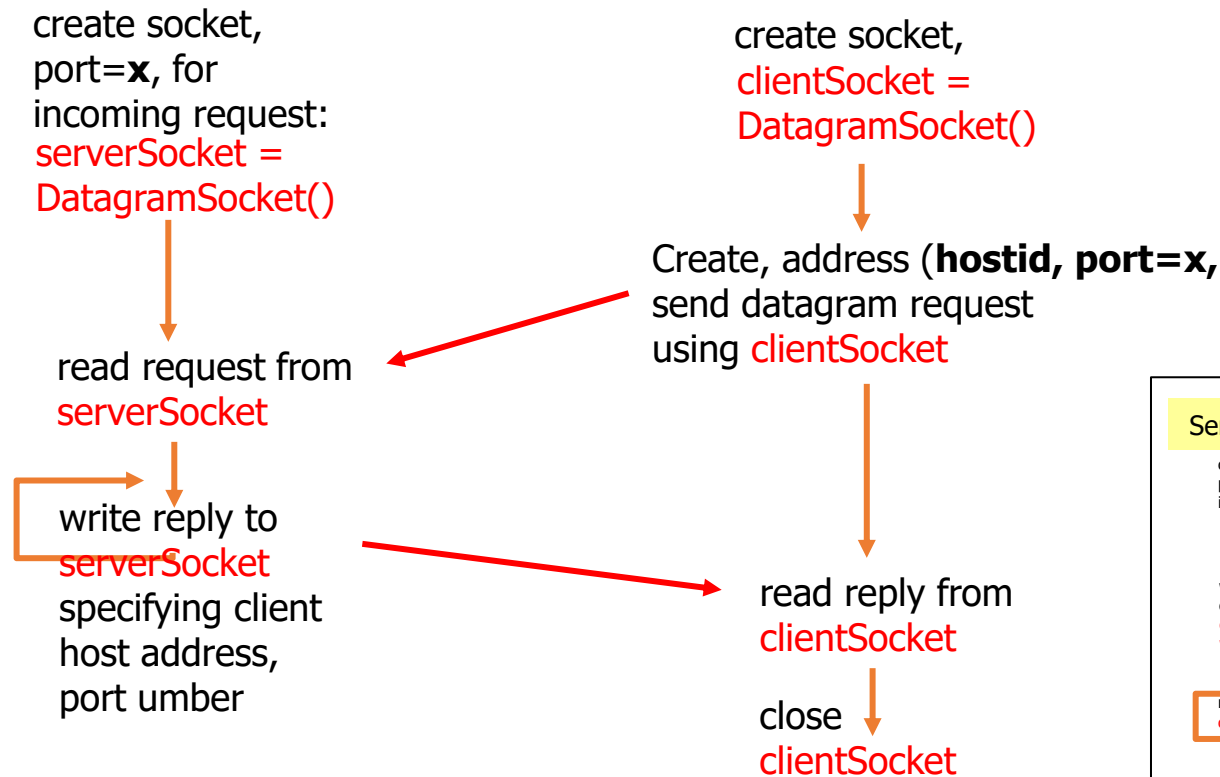
```
int getPort();
```

Example: Java client (UDP)



Client/server socket interaction: UDP

Server (running on **hostid**) Client



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);

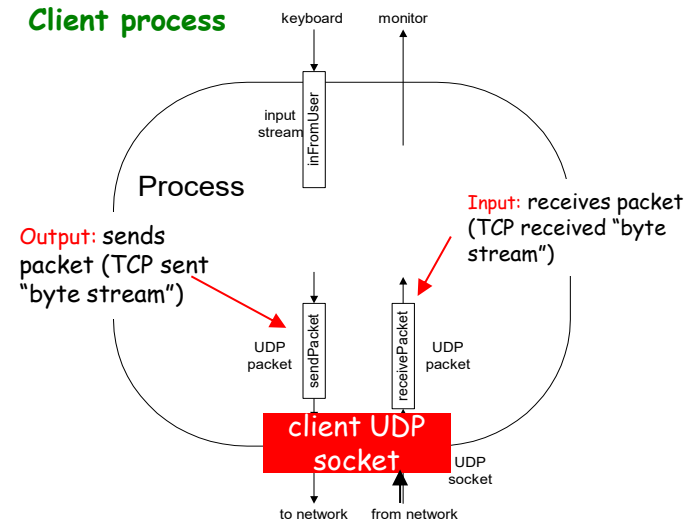
        DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);

        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();
```

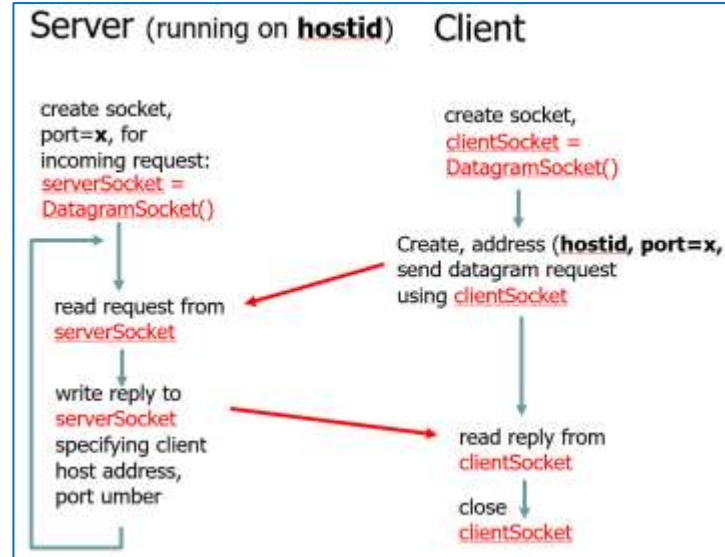
```
            InetAddress IPAddress = receivePacket.getAddress();
```

```
            sendData = capitalizedSentence.getBytes();
            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

编程作业 2

Deadline: Nov 2, 2025