4430S
Data Communication and Computer Networks

Dr. WONG Tsz Yeung

Chapter 2, Part 1

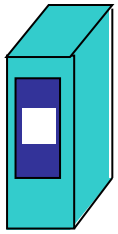
Application Layer & Programming

The playground of all fancy applications.

Pre-requisite: process (service) identification

- Every end host on the Internet is assigned an IP address.
  - Note that a machine can have more than one IP address!

- However, a machine may provide more than one services!
  - How can a client call for a particular service?
  - Such as:
    - **http** service of 137.189.91.192, or
    - **telnet** service of 137.189.91.192.
  - How can a client identify them?



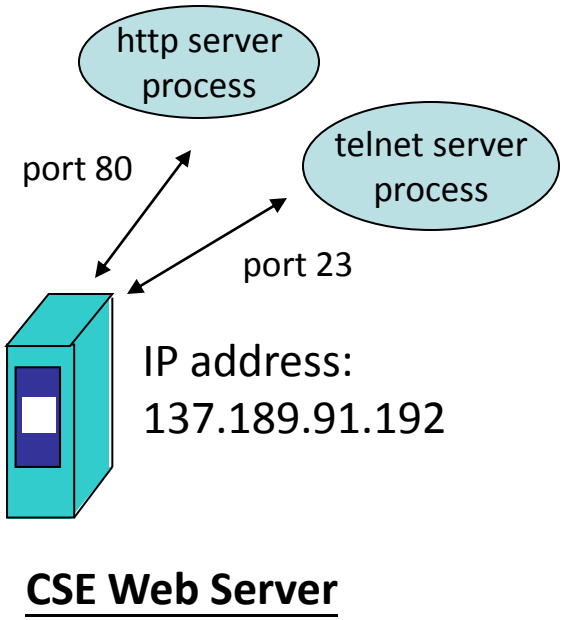
IP address:
137.189.91.192

CSE Web Server

Pre-requisite: process (service) identification

- Well…the communication port is a number, is a logical representation.
  - Different services are identified by different port numbers.
  - HTTP is usually of port 80.
  - telnet is usually of port 23.

- Look up **`/etc/services`** in Linux for details.



Pre-requisite: process (service) identification

- To request for a service, you need to know:
  - the IP address of the remote machine;
  - the port number that the service resides;
  - the "language" that the service is talking in.

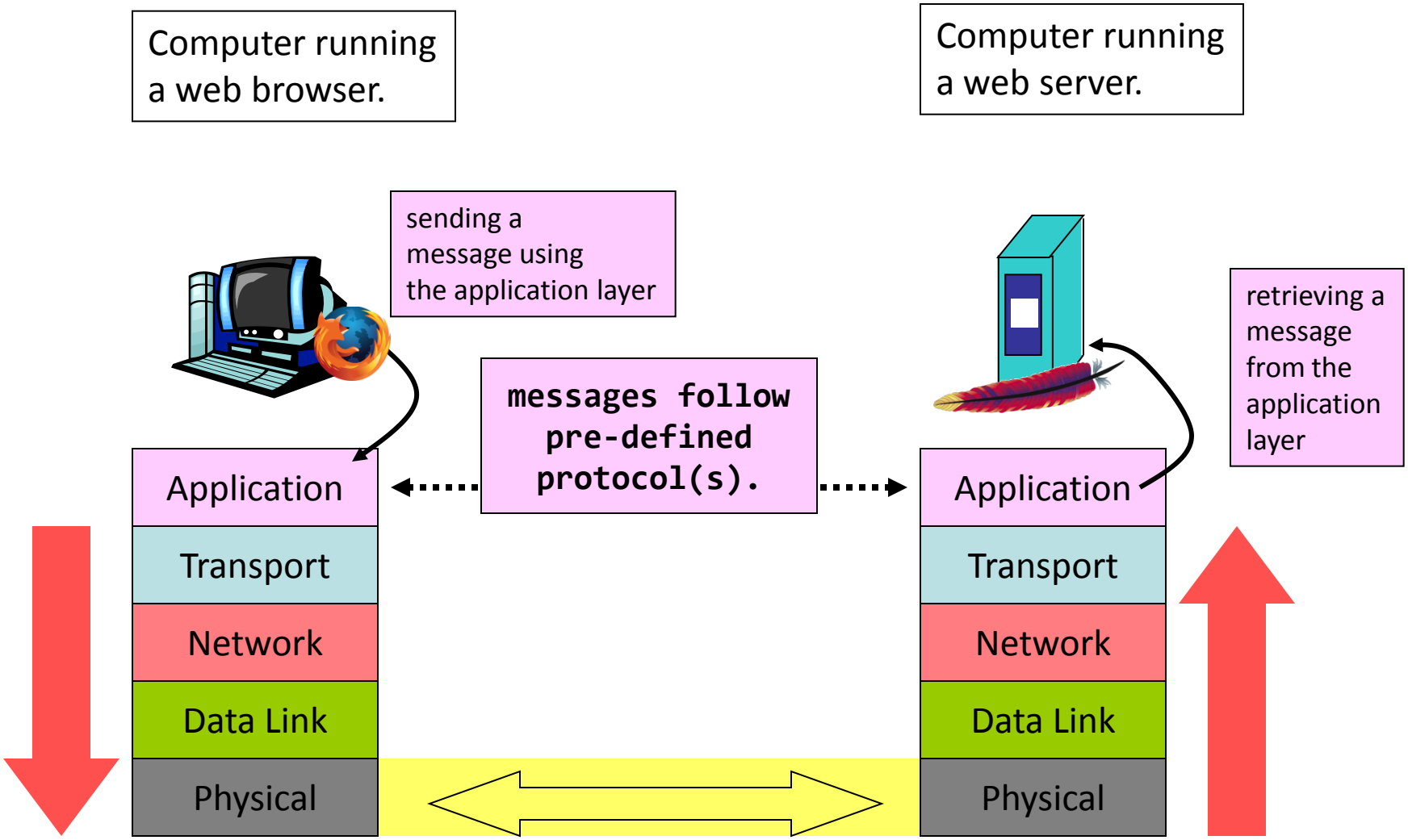
- For example, the web service provided by the CSE web server is on:
  - IP address: 137.189.91.192 (Network layer ID);
  - Port number: 80 (Transport layer ID);
  - Language: HTTP (Application layer content);

- FYI, usually there are two ways for a process to handle another process that **speak alien languages**.
  - Guess! What are the ways?

Pre-requisite: process (service) identification

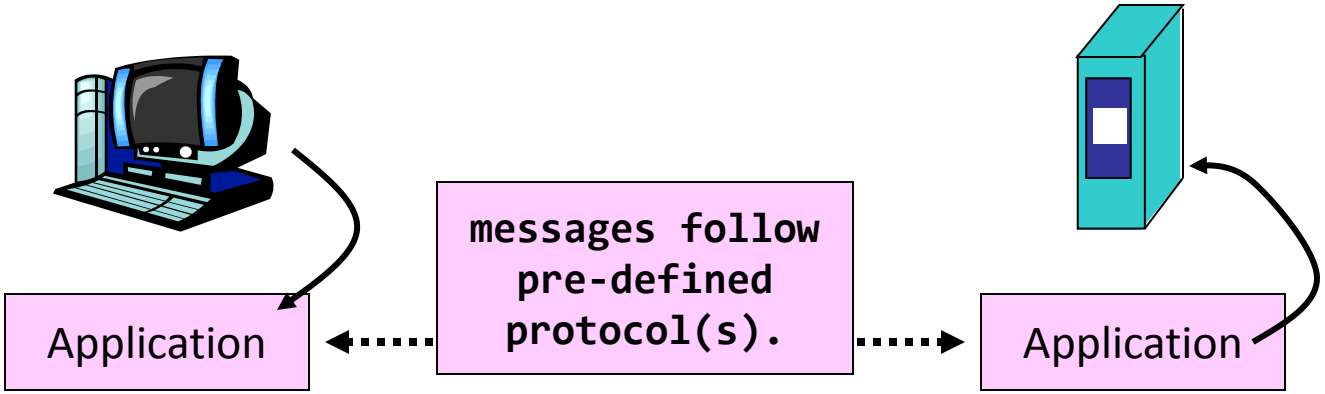
Service Name	Protocol Name	TCP/UDP	Port Number
Web	HTTP	TCP	80
Web with encryption	HTTPS	TCP	443
Secure Shell	SSH	TCP	22
Email sending	SMTP	TCP	24
Domain name service	DOMAIN	UDP	53
Email retrieval	POP3	TCP	110
	IMAP v3	TCP	220
	IMAPS	TCP	993

The Big Picture...



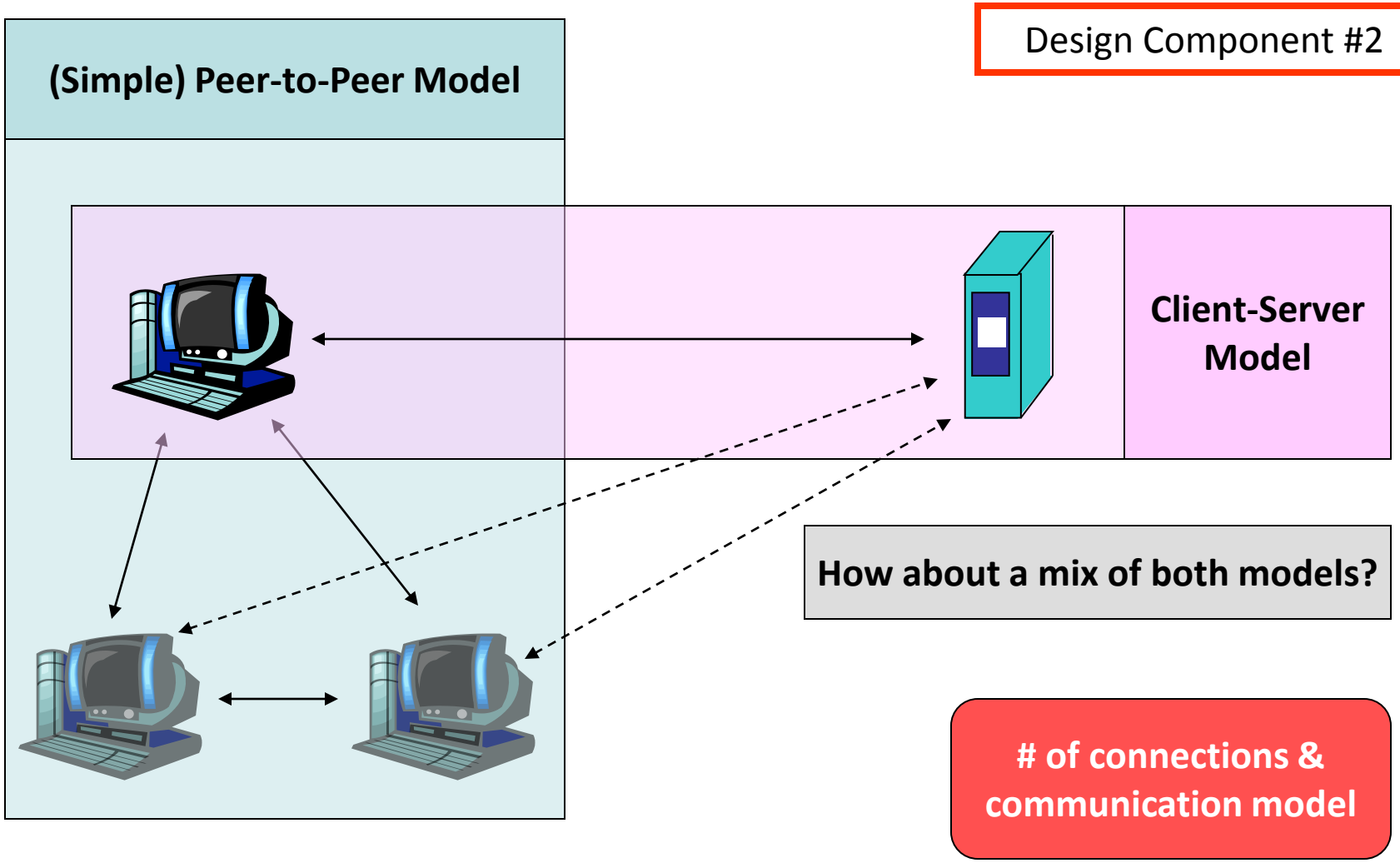
The Big Picture...

Design Component #1



"Medium"	Reliable or not?
Encoding	ASCII, Binary, Compression, Encryption?
Connection type	Persistent or not?
Process Internal	Stateful or Stateless?
Protocol content	What is the syntax?

The Big Picture...

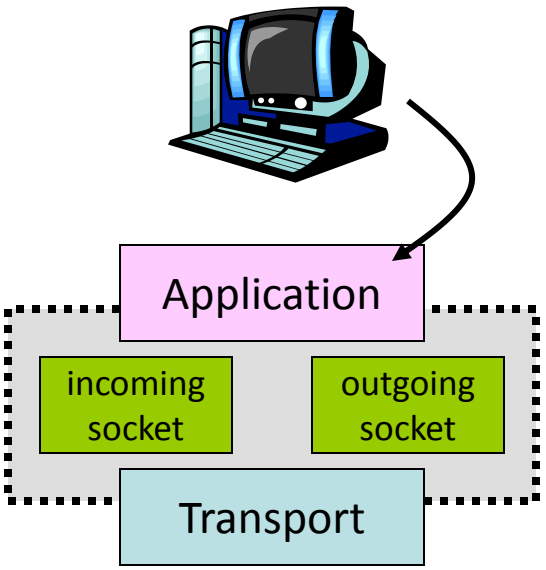


The Big Picture...

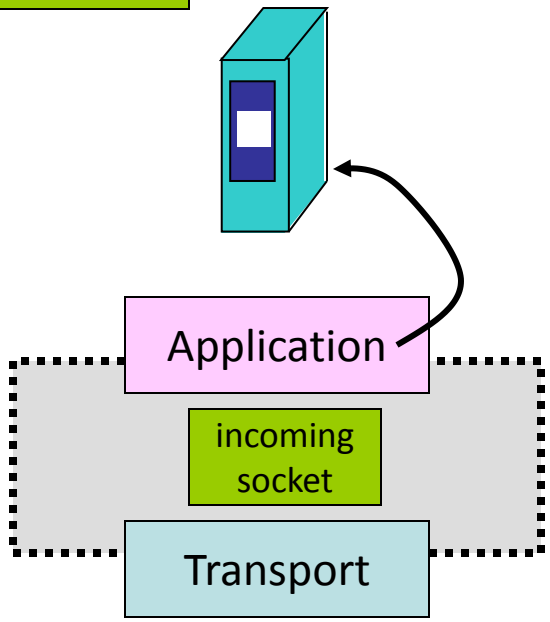
Socket
- is a file descriptor;
- defines a connection;
- has 2 types: incoming & outgoing.

The **model** and the **need** define the # and types of sockets!

Design Component #3



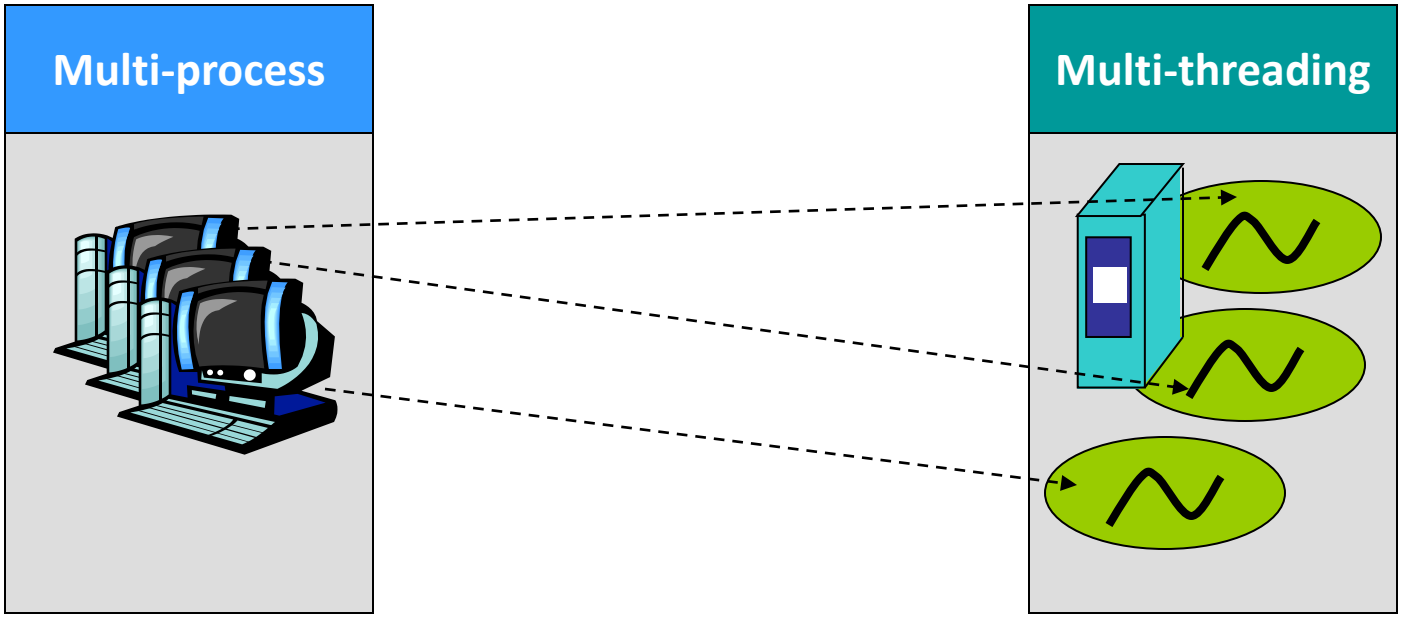
Both incoming & outgoing? This may be a P2P model.



A server usually needs one incoming socket only.

The Big Picture...

Design Component #4

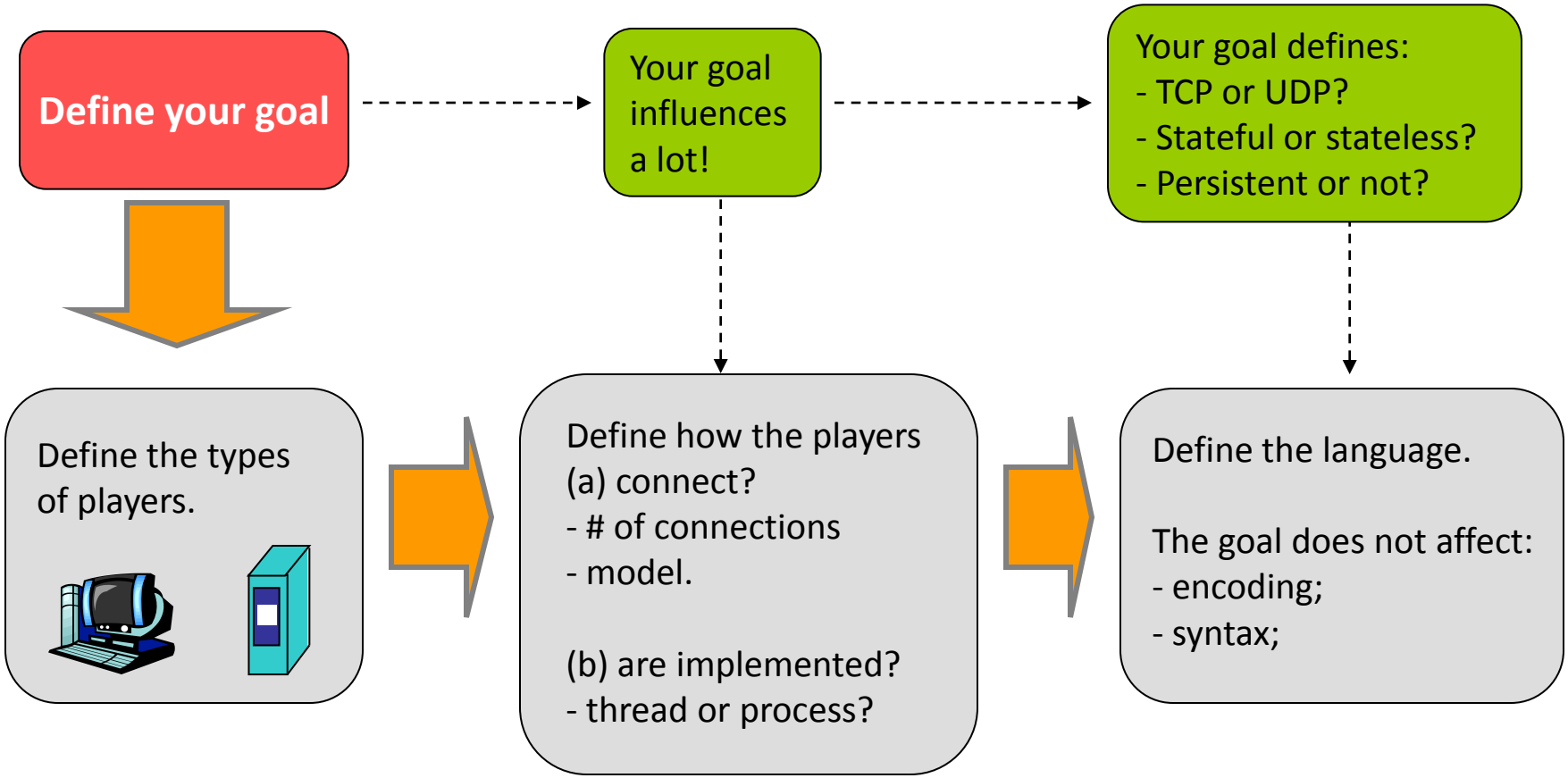


Programming Overheads
I/O Multiplexing – **select()**.
Process Synchronization.

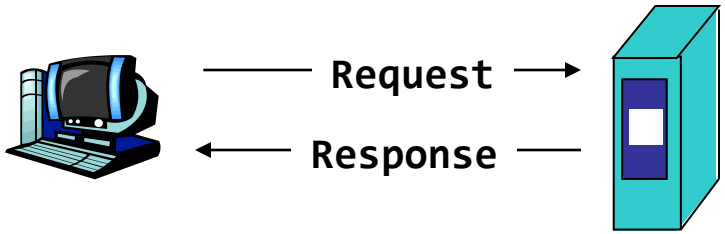
Programming Overheads
File descriptor limit.
Thread Synchronization.

So, what is application layer?

- It is a **top-down design** of an application!



Characteristics #1: state

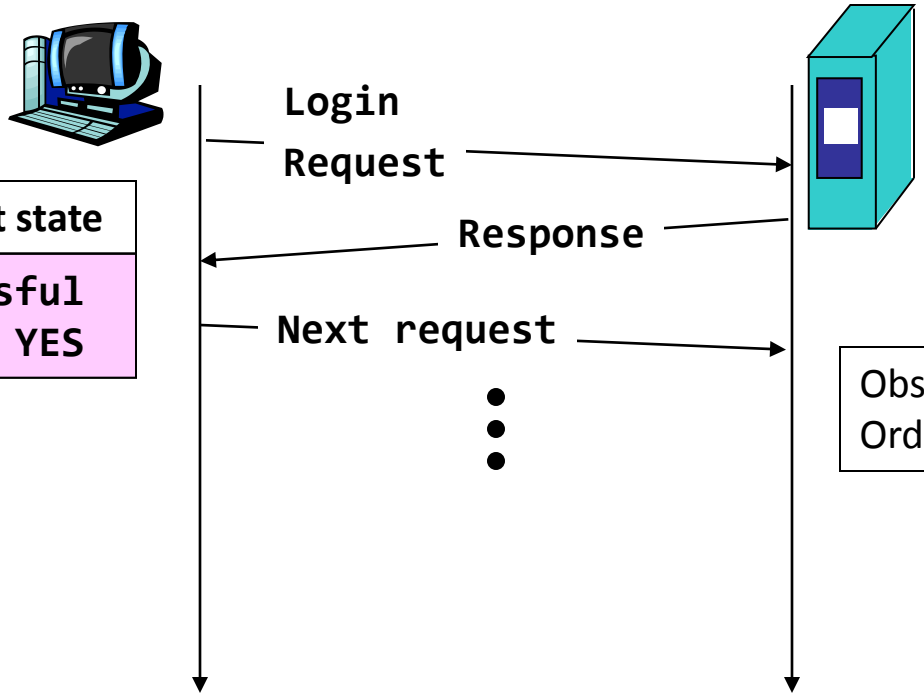


Property #1:
Processes memorize nothing for incoming messages.

Property #2:
Order of request messages is not important.

Stateless Scenario

Stateful Scenario



SSH client state

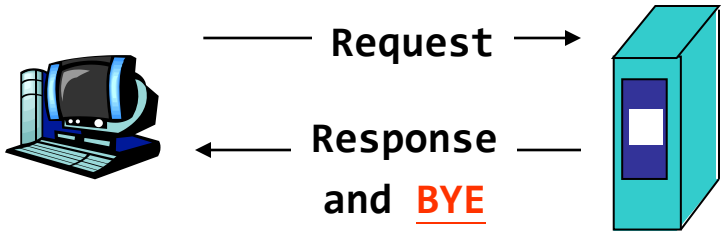
`Successful Login: YES`

SSH server state

`current user: tywong`

Observation:
Order of request messages is important.

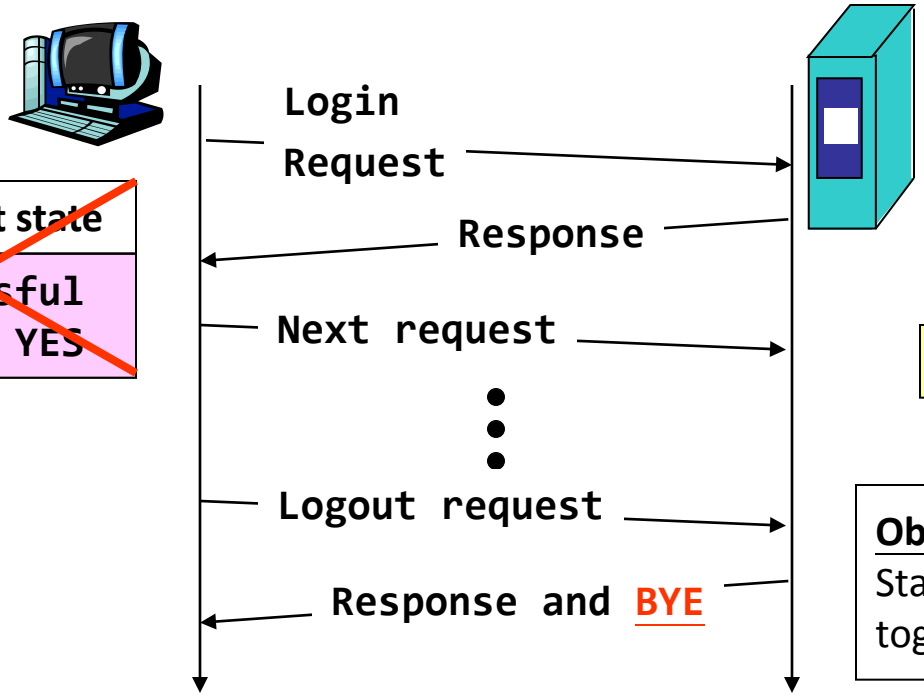
Characteristics #2: persistence



Hang up after the job is done.

Non-Persistent Scenario

Persistent Scenario



~~SSH client state~~
~~Successful Login: YES~~

~~SSH server state~~
~~current user: tywong~~

Hang up upon explicit requests.

Observation.
State and persistence usually come together, not necessarily a requirement.

Characteristics #2: persistence

Non-Persistent Connection	
Advantages	**Disadvantages**
Easy to implement. E.g., no need to implement timeout for idle connections.	Start a TCP connection for each document. The overhead is large for both client and server. E.g. A browser has to open parallel connections to fetch a number of documents at the same time. It is a problem when the number of connections is constrainted.
The server or the client are not required to memorize anything.	Another issue is that non-persistent connection is not suitable for implementing stateful protocols.

Characteristics #2: persistence

Persistent Connection	
Advantages	**Disadvantages**
Less overhead. All the requests can use the same TCP connection.	The request has to be sent one by one: only after a proper response has been received. No pipelining.
Support stateful protocols. E.g., a login procedure can be implemented by using the same TCP connection. **Discussion**: Is it a **must** to have persistent connection when you are logging in?	Garbage collection for idle connections. Else, you'll be running out of available connections.

Characteristics #1 + #2

- Let's fill in the blanks.

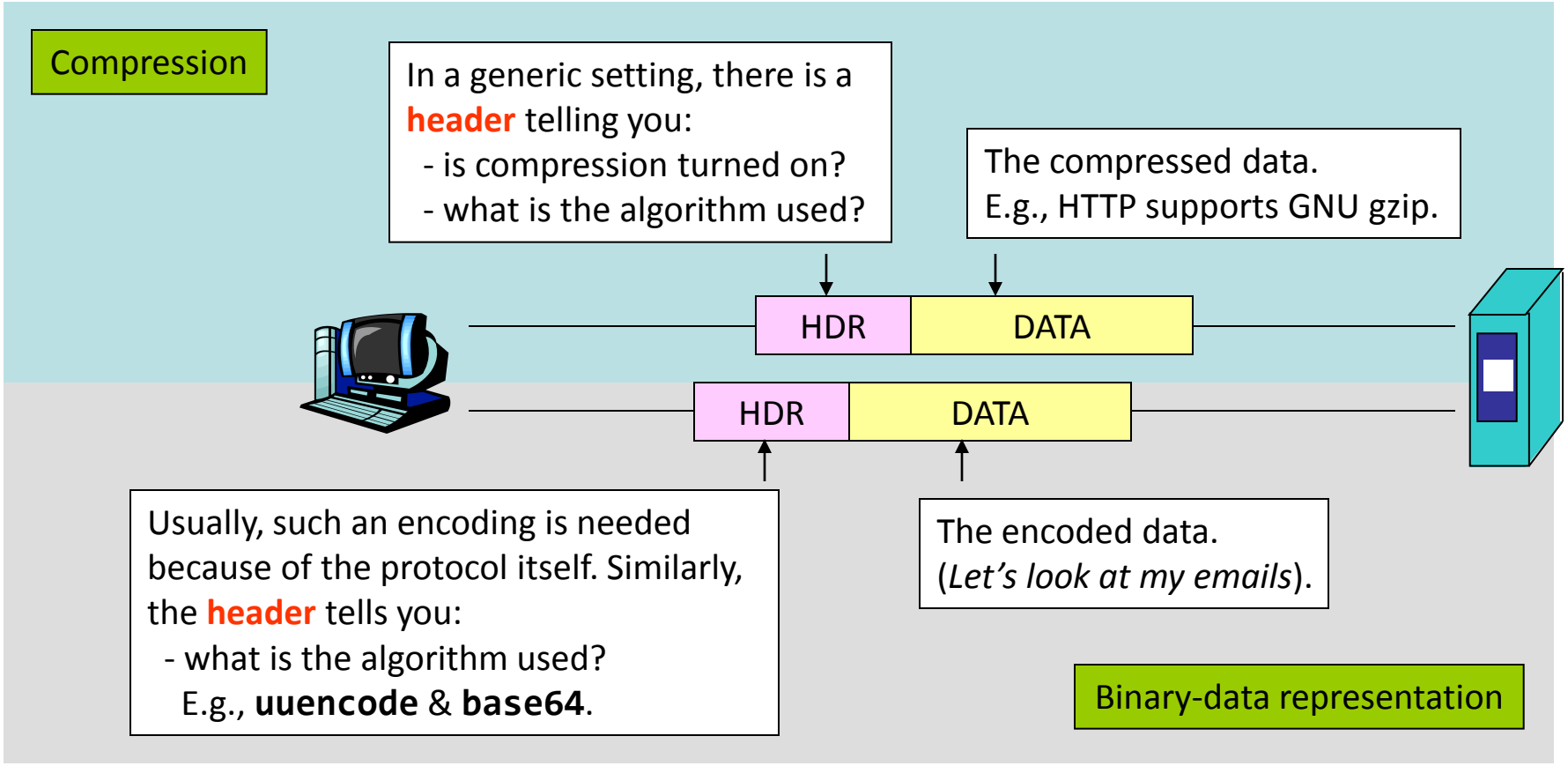
Protocol	Stateful?	Persistence?
HTTP	NO	YES and NO
HTTPS	YES	YES
DNS	NO	NO
FTP	YES	YES
SSH	YES	YES

Can be turning on and off.

For the difference between HTTP and HTTPS, please attend CSCI5470 lectures!

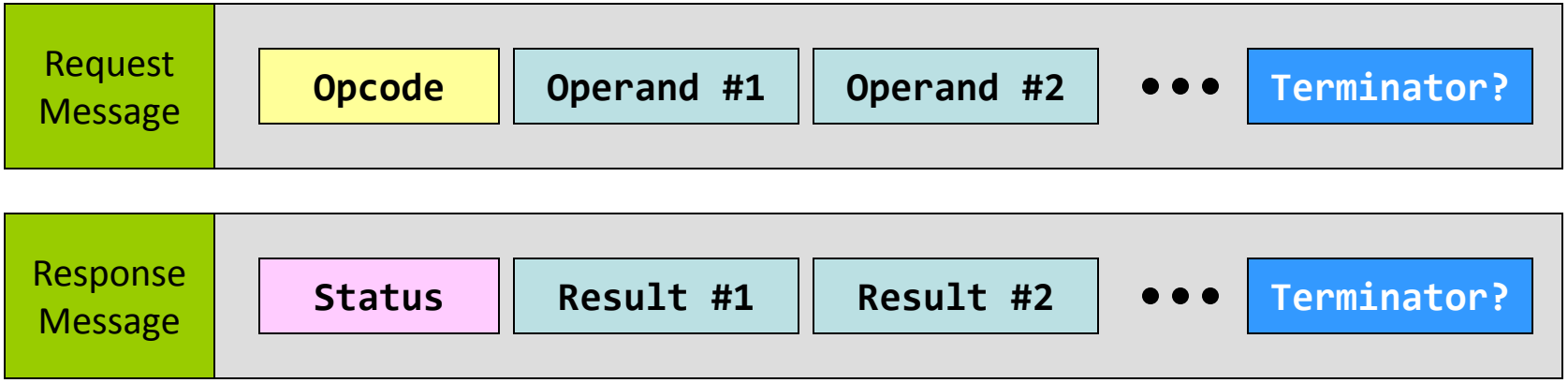
Characteristics #3: encoding

- Usually, encoding comes in two ways:
  - compression & binary-data representation.



Characteristics #4: syntax

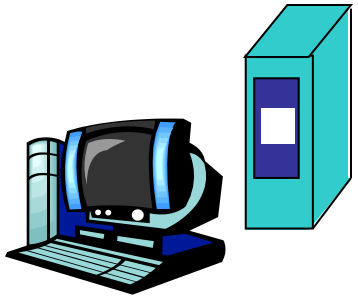
- Protocol-independent design.



Concerns	Description
Delimiter?	Hard coding the length of each field? Or, using a common delimiter to tokenize the message?
ASCII / Binary?	Just the designer's style...
Length Limitation?	E.g., UDP has a soft constraint on the message length.
Terminator?	How do you know the end of a message?

Application layer

- Protocol design:
  - Learn from examples!
    - HTTP



Learn from example #1 – HTTP

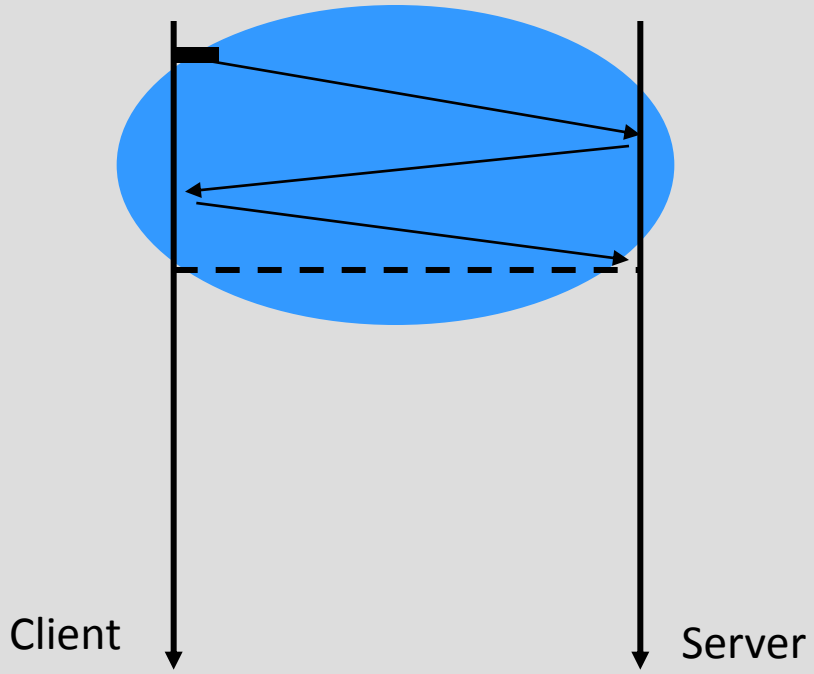
- HTTP – HyperText Transfer Protocol
  - HTTP 1.0 – RFC 1945.
  - HTTP 1.1 – RFC 2616

- A **request-response** protocol: always the client makes the 1st move!

- Characteristics:
  - Since it transfers files, **TCP** is required.
  - It is a **stateless** protocol!
  - It can be a **non-persistent** protocol!
  - Encoding:
    - no encryption (well, HTTPS is another protocol);
    - compression is optional (using gzip).

Learn from example #1 – HTTP

- Sidetrack – what is a connection?
  - A connection is established based on TCP!



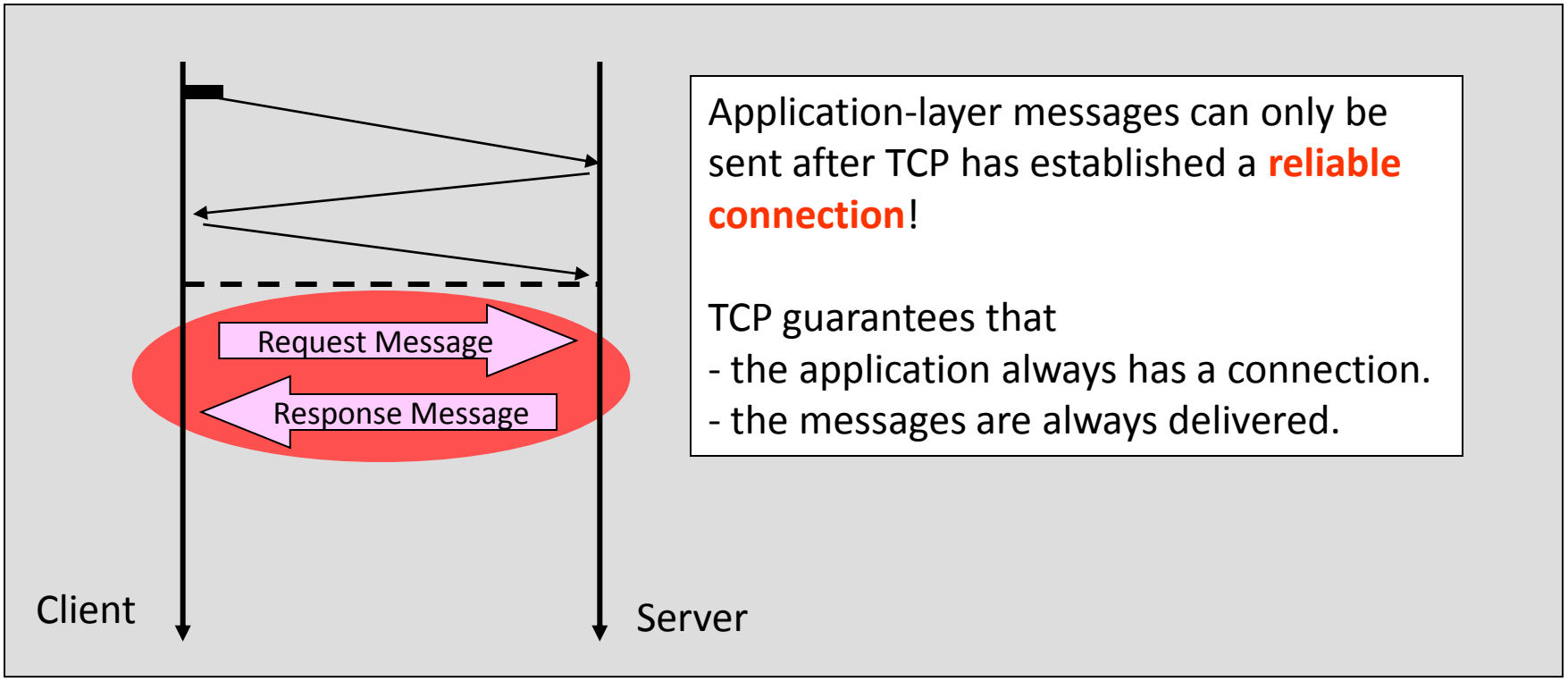
Some TCP things are happening here. We call those steps the **TCP handshaking**.

This marks the start of a connection.

For example, when two strangers meet, before they really start a conversation, they *shake hands and introduce themselves*.

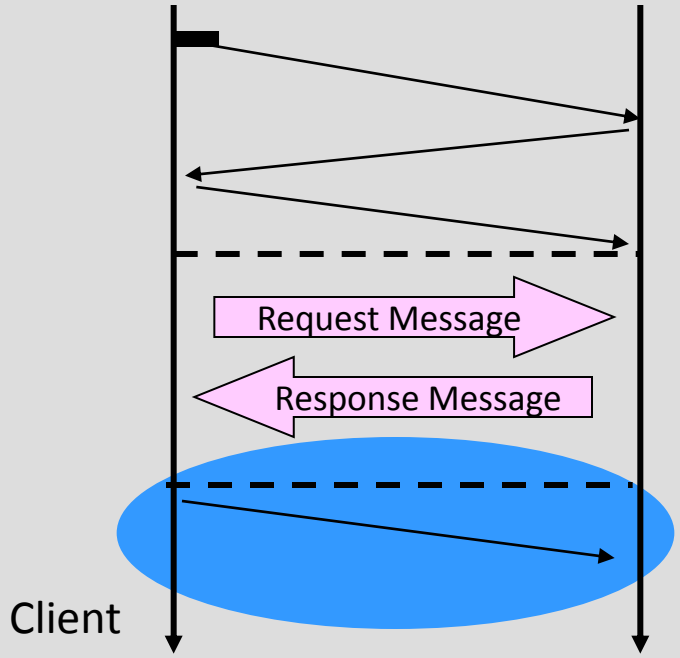
Learn from example #1 – HTTP

- Sidetrack – what is a connection?
  - A connection is established based on TCP!



Learn from example #1 – HTTP

- Sidetrack – what is a connection?
  - A connection is established based on TCP!
  - A connection is closed because of TCP!



Last, the server/the client drops the connection with another TCP handshaking.

Continuing the example, when one of the strangers wants to stop talking, he will shake the other guy's hand, and says "Nice to meet you". Then, he leaves.

Learn from example #1 – HTTP

- Simplified goal: **one-way, reliable file transfer**.
- Players:

Role	Description
Browser (Active role)	- Know the location of a server. - Know the path to the target file on the server. - Connect to the server and retrieve the file.
Web server (Passive role)	- Waiting for connection. - Read the path to a file. - If the path is OK, return the file content. Else, read an error.

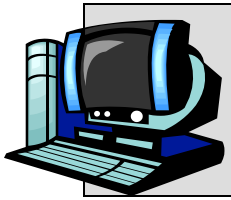
Learn from example #1 – HTTP

- Model: client-server.

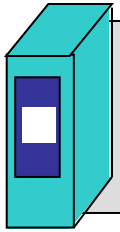
Client Properties	Server Properties
Can be any network addresses.	Fixed and well-known network address.
Execute on-demand: short-lived comparing to the server.	Always executing. When failed, we call that downtime.
Consume services.	Provide services. (That's why it is called "server").
Active role: the one who initiates the communication.	Passive role: the one who waits for communication.

Learn from example #1 – HTTP

- Implementation?



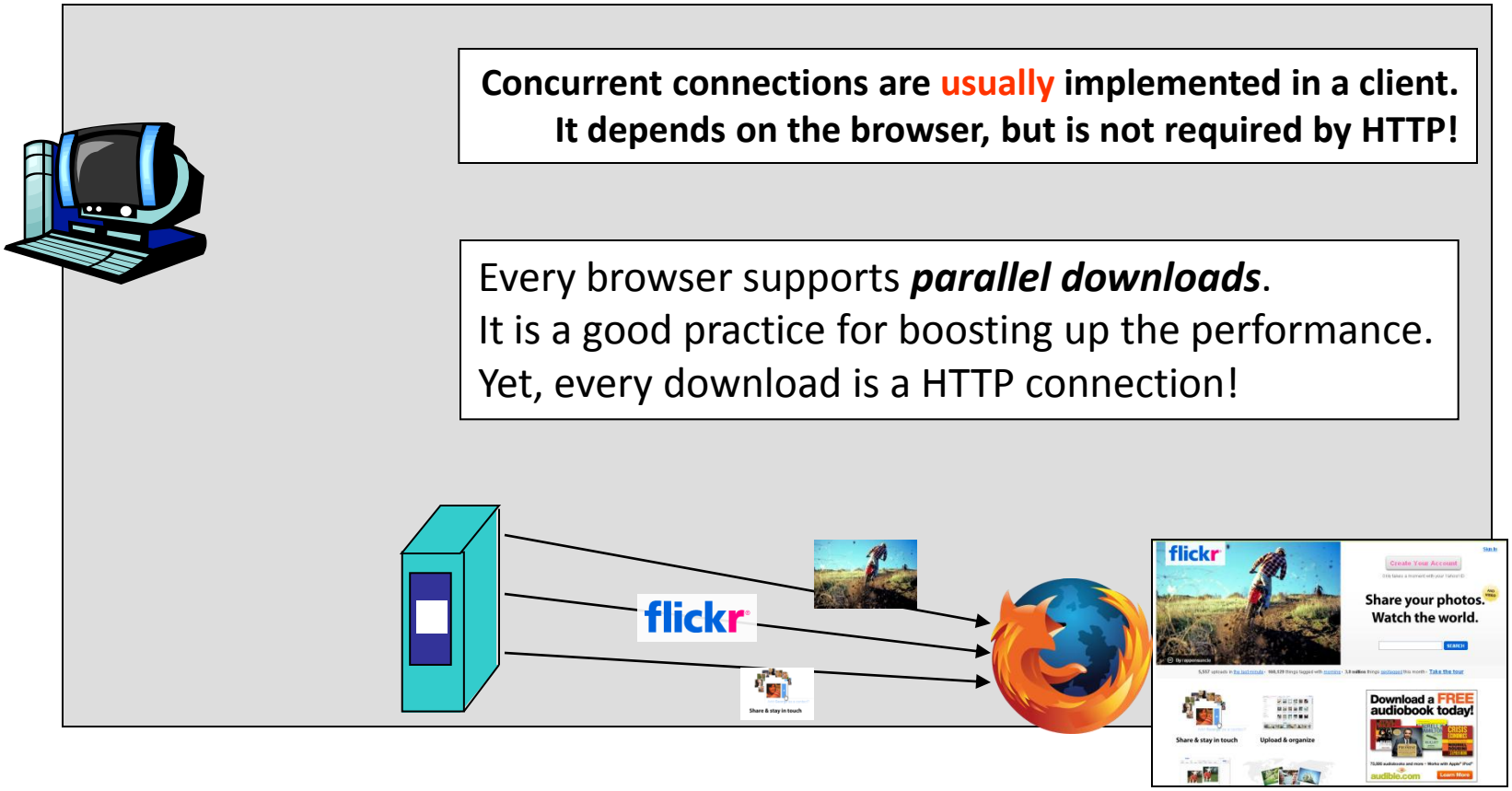
Discussion: Are multiple connections needed? Why?



Discussion: Are multiple connections needed? Why?

Learn from example #1 – HTTP

- Implementation?



Goal → Players → Model & Implementation → Language Definition

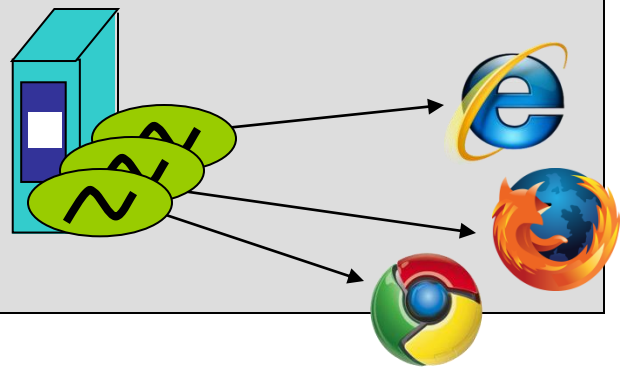
Learn from example #1 – HTTP

- Implementation?

Concurrent connections are always implemented in a server.
It depends on the server, but is not required by HTTP!

Every server must be able to serve multiple clients **at the same time**.
(Think about the consequence if this feature is missing...)

Discussion. What if there is only one instance of server running?
(meaning 1 process with 1 thread.)



Goal → Players → Model & Implementation → Language Definition

HTTP – syntax: message format

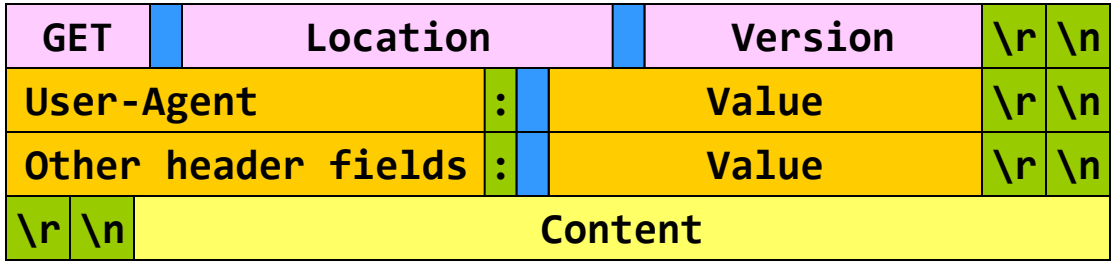
- For both request and response...

HEADERS	Header stores a lot of information of the request. E.g., the location, the HTTP cookies, etc.
CONTENT	Content is the file/data to be transferred.

Concerns	Description
Delimiter?	Space character, line feed + New line "\r\n"
ASCII / Binary?	ASCII header (opcode), Binary operand.
Length Limitation?	No.
Terminator?	No. Length is limited by the "**`content-length`**" header field.

HTTP – syntax: message format

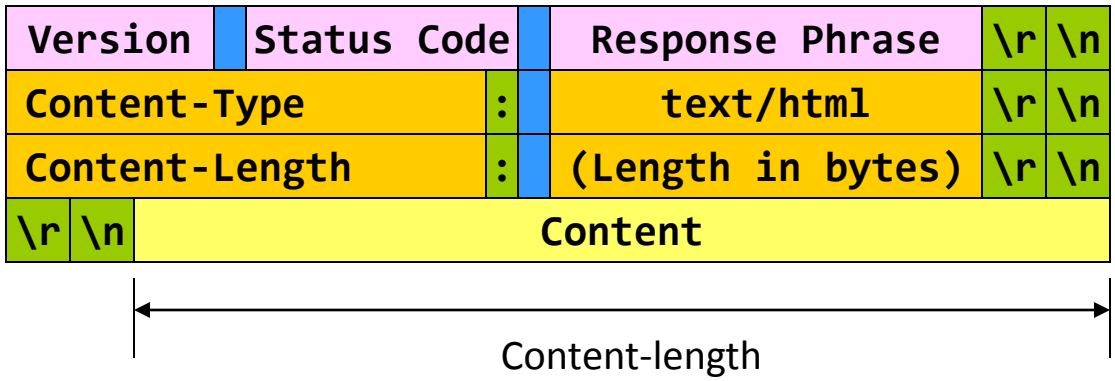
- Request message:



If the fields:
- Content-Type exist, and
- Content-Length > 0,

then,
there exists a content body at the end of the request (response) message.

- Response message:



HTTP – syntax: message format

- Example:

Try persistent connection with the following header line:

`Connection: keep-alive`

```
$ telnet www.cse.cuhk.edu.hk 80
Trying 137.189.91.192...
Connected to fortress.cse.cuhk.edu.hk.
Escape character is '^]'.
GET /~tywong/4430.html HTTP/1.0
User-Agent: Mozilla/4.0

HTTP/1.1 200 OK
Date: Tue, 11 Jan 2011 14:46:03 GMT
Server: Apache
Accept-Ranges: bytes
Vary: Accept-Encoding
Content-Length: 31
Connection: close
Content-Type: text/html

<html>
Welcome to 4430
</html>
Connection closed by foreign host.
$ _
```

Get method request header

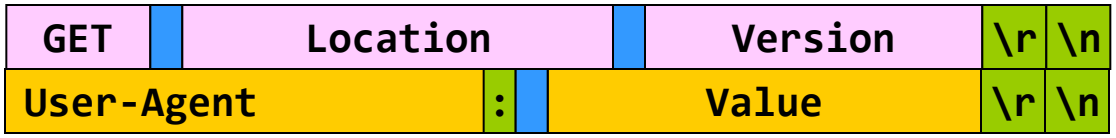
Response Header

Response Content

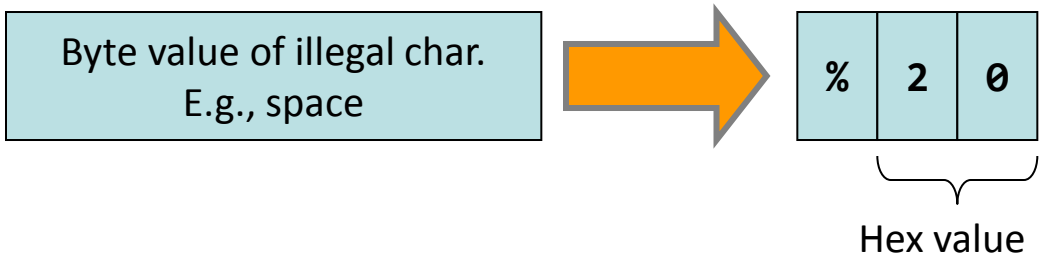
Non-persistent connection!

HTTP – syntax: illegal characters

- The language is quite "*brain-damaging*" because...

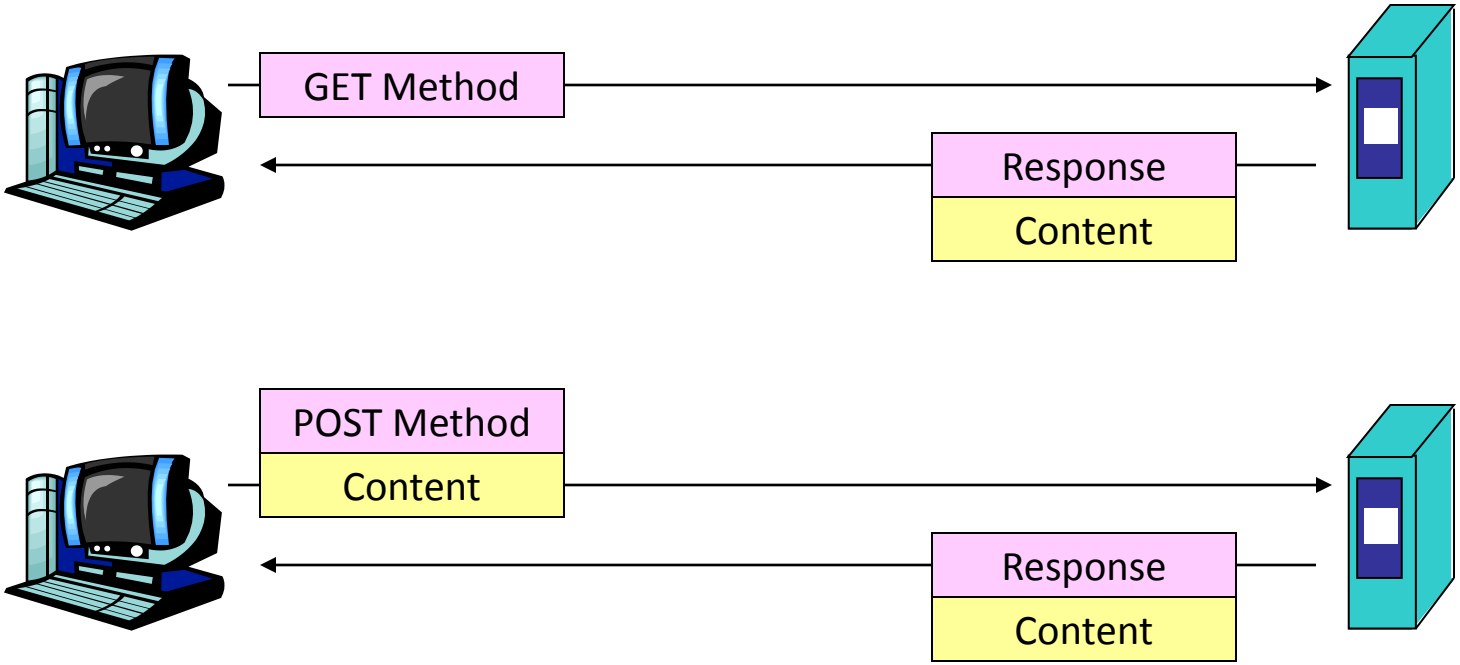


  - There are illegal characters for the header...
    - Space, new line, line feed, `:', etc.
  - So, **encoding** is needed for header values!



HTTP – language: request method

- There are two popular methods:
  - "**GET**": download data.
  - "**POST**": upload data.



HTTP – language: response value

Code	Class	Description
2xx	Successful	Request can be fulfilled.
3xx	Redirection	The object is moved to some where else.
4xx	Client Error	The client is doing something wrong.
5xx	Server Error	The server is not able to process the request successfully.

Status Code	Response Phrase	Description	Example (in one line)
200	OK	Request succeeded. Object is given later in the message.	**NIL**
301	Move Permanently	Object moved. New Location is specified later in the message.	**GET /~tywong**
400	Bad Request	Request is not understood by the server	**GET /~tywong HTTP/1.1**
403	Forbidden	Object is not allowed to be accessed.	**(chmod to 600, then try again)**
404	Not Found	Object is not found on the server.	**GET /~tywong/4430**

HTTP – language: content encoding

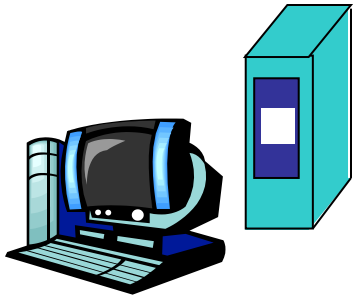
- What is the Content-Type header field?
  - The value of Content-Type is defined in the MIME (Multipurpose Internet Mail Extension) standard
    - see RFCs 2045-2048.
  - This is a formal standard to define
    - the type of a document; and
    - the file extension of a document.
  - Take a look at the file "**`/usr/share/mime/globs`**" in Linux.

HTTP - Summary

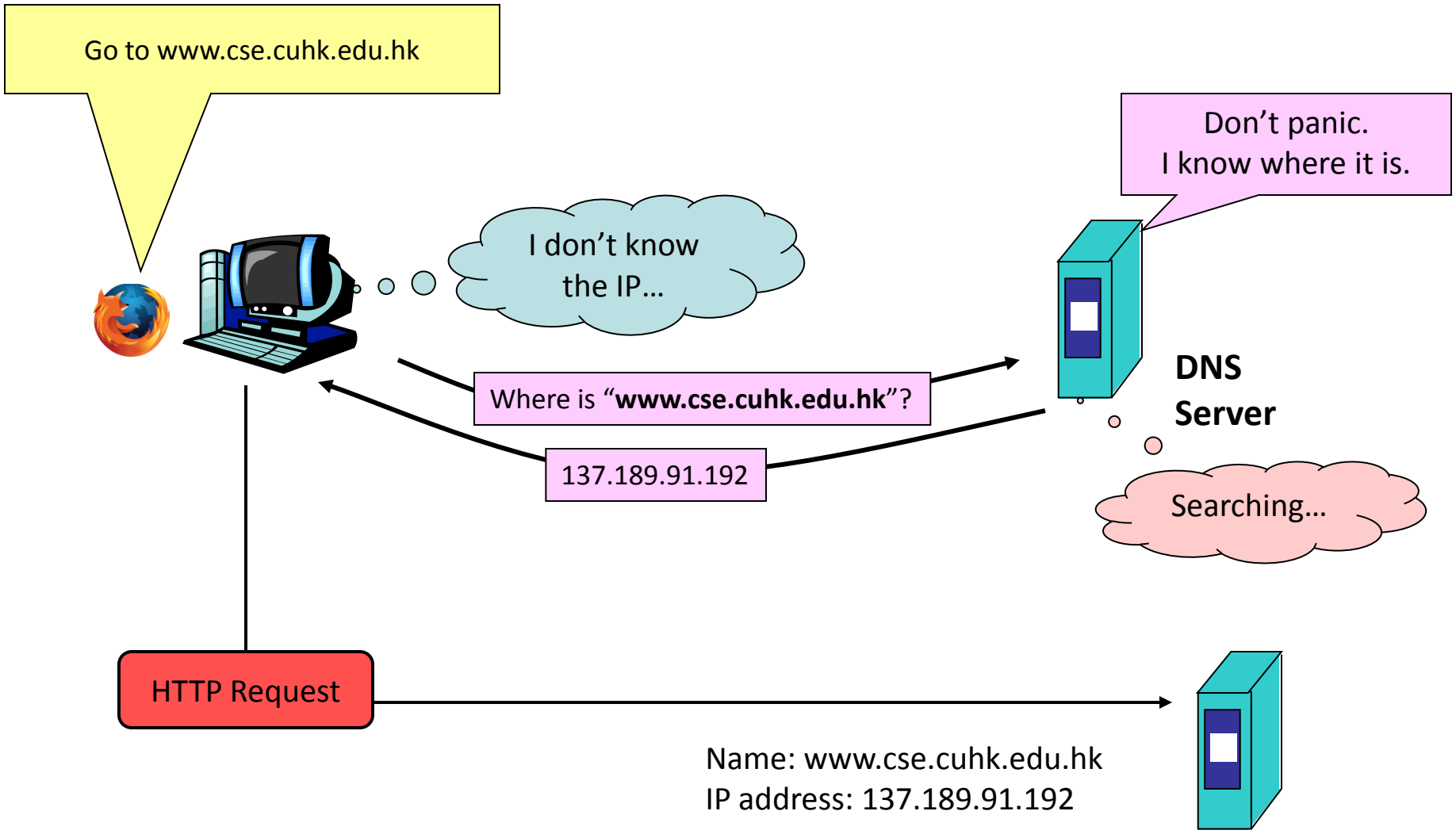
Transport Layer Protocol	TCP
Header encoding	ASCII
Content encoding	ASCII or binary - depends on the "**`Content-Type`**" header field.
Message exchange	Request-response - client is always the first guy to speak.
Connection duration	Depends on the "**`Connection`**" header field. - "**`close`**": closed by the server after response has been sent. - "**`keep-alive`**": closed by the server after the connection is idle for certain second.

Application layer

- Protocol design:
  - Learn from examples!
    - HTTP
    - DNS

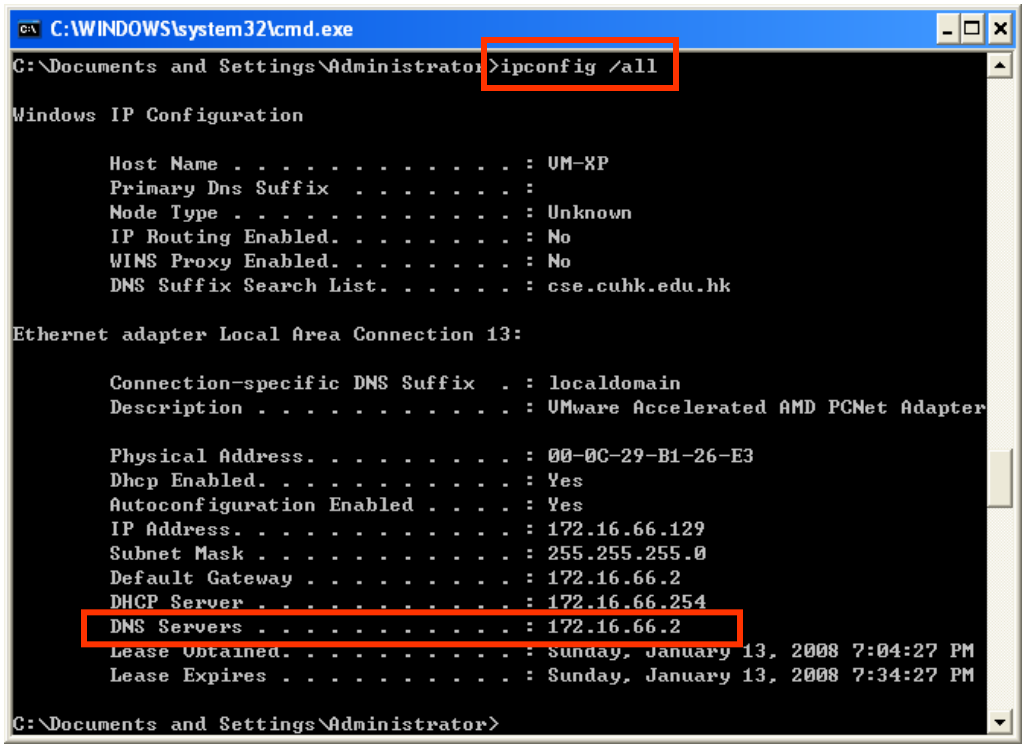


Learn from example #2 – DNS



DNS – What is it?

- Every computer must know about at least one DNS server.



Also, try to take a look at "**`/etc/resolv.conf`**" in Linux and Unix.

DNS - Functions

- Basic functions: translation!
  - Translate hostname to IP address.

```
$ nslookup pc91072.cse.cuhk.edu.hk
Server:         137.189.91.188
Address:        137.189.91.188#53

Name:   pc91072.cse.cuhk.edu.hk
Address: 137.189.91.72
```

result

  - Translate IP address to hostname.

```
$ nslookup 137.189.91.72
Server:         137.189.91.188
Address:        137.189.91.188#53

Name:   pc91072.cse.cuhk.edu.hk
Address: 137.189.91.72
```

result

DNS - Functions

- Additional functions: host aliasing
  - Translate a hostname into its canonical (formal) name.
  - E.g.,
    - The name: **www.google.com** is just an alias.
    - The true name is **www.l.google.com**.

```
$ nslookup www.google.com
Server:         137.189.91.188
Address:        137.189.91.188#53

Non-authoritative answer:
www.google.com  canonical name = www.l.google.com.
Name:   www.l.google.com
Address: 72.14.235.147
Name:   www.l.google.com
Address: 72.14.235.99
Name:   www.l.google.com
Address: 72.14.235.104
```

DNS - Functions

- Additional functions: load distribution.
  - This is to distribute the load of client machines, not the DNS server itself.
  - The DNS server can translate a name to more than one IP addresses.
  - This allows the load distribution of busy machines.

```
$ nslookup www.google.com
Server:         137.189.91.188
Address:        137.189.91.188#53

Non-authoritative answer:
www.google.com  canonical name = www.l.google.com.
Name:   www.l.google.com
Address: 72.14.235.147
Name:   www.l.google.com
Address: 72.14.235.99
Name:   www.l.google.com
Address: 72.14.235.104
```

DNS - Feature

- DNS is a global facility.

- Everyone on this Earth should
  - be able to use the DNS service.
  - have the same result returned from the DNS service.

- How to manage such a big, important facility?

DNS - Feature

- The DNS takes a decentralized approach.

- The advantages are:
  - (1) No single point of failure;
    - Taking down one DNS server will not take down the whole Internet.
  - (2) Traffic distribution;
    - The entire population on Earth is using this single system.
    - It is not possible to have one centralized server to take such a huge workload.

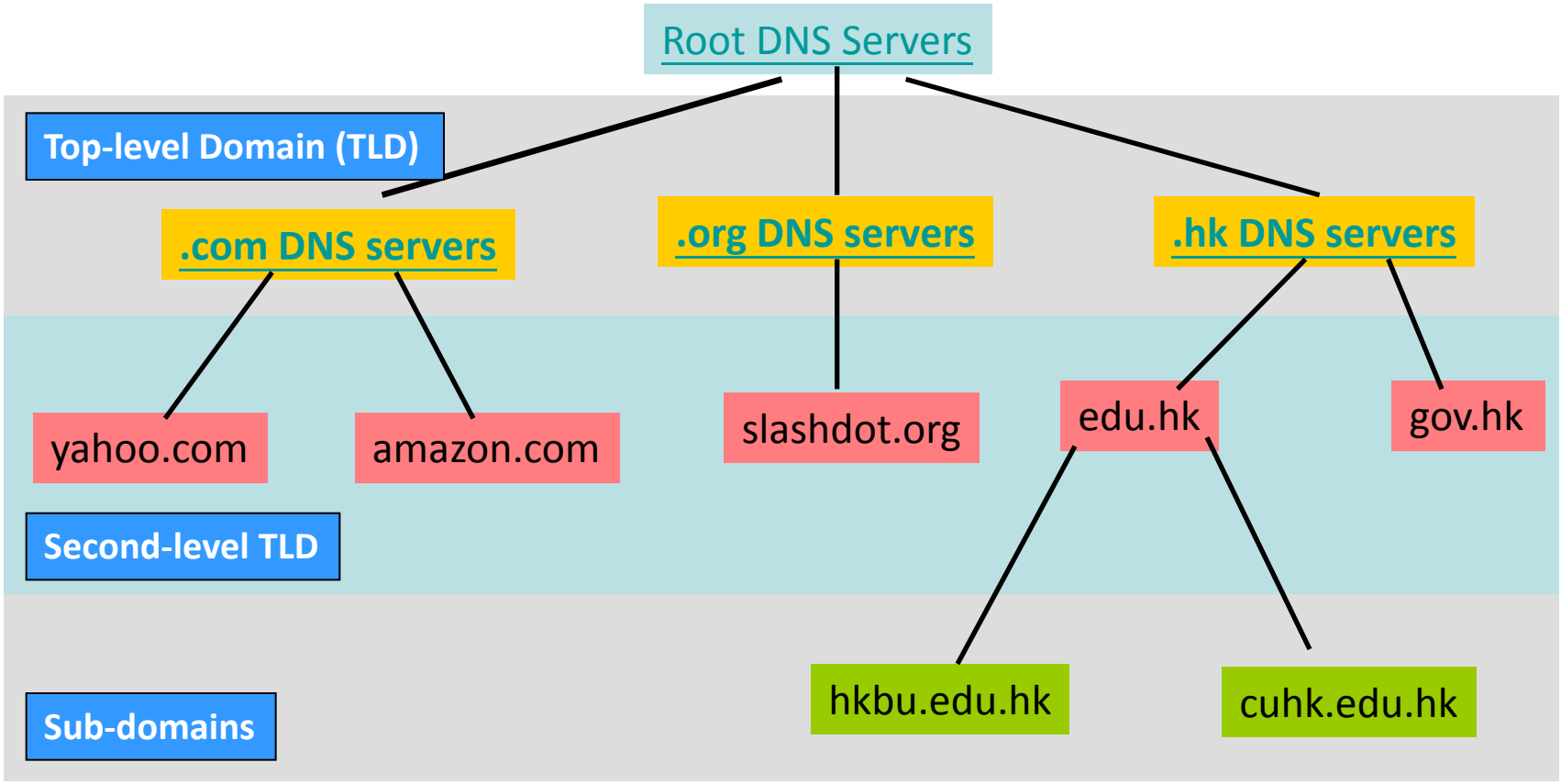
DNS - Feature

- The DNS takes a decentralized approach.

- The advantages are:
  - (3) Geographical advantage.
    - If there is only one or a few centralized DNS servers, a request has to go through a long way in order to reach the server.
    - A local DNS can help solving the problem!.
  - (4) Maintenance.
    - If a single DNS is used, then
      - the DNS database will be huge;
      - updating will be frequent.
    - Every organization should keep its own records only.

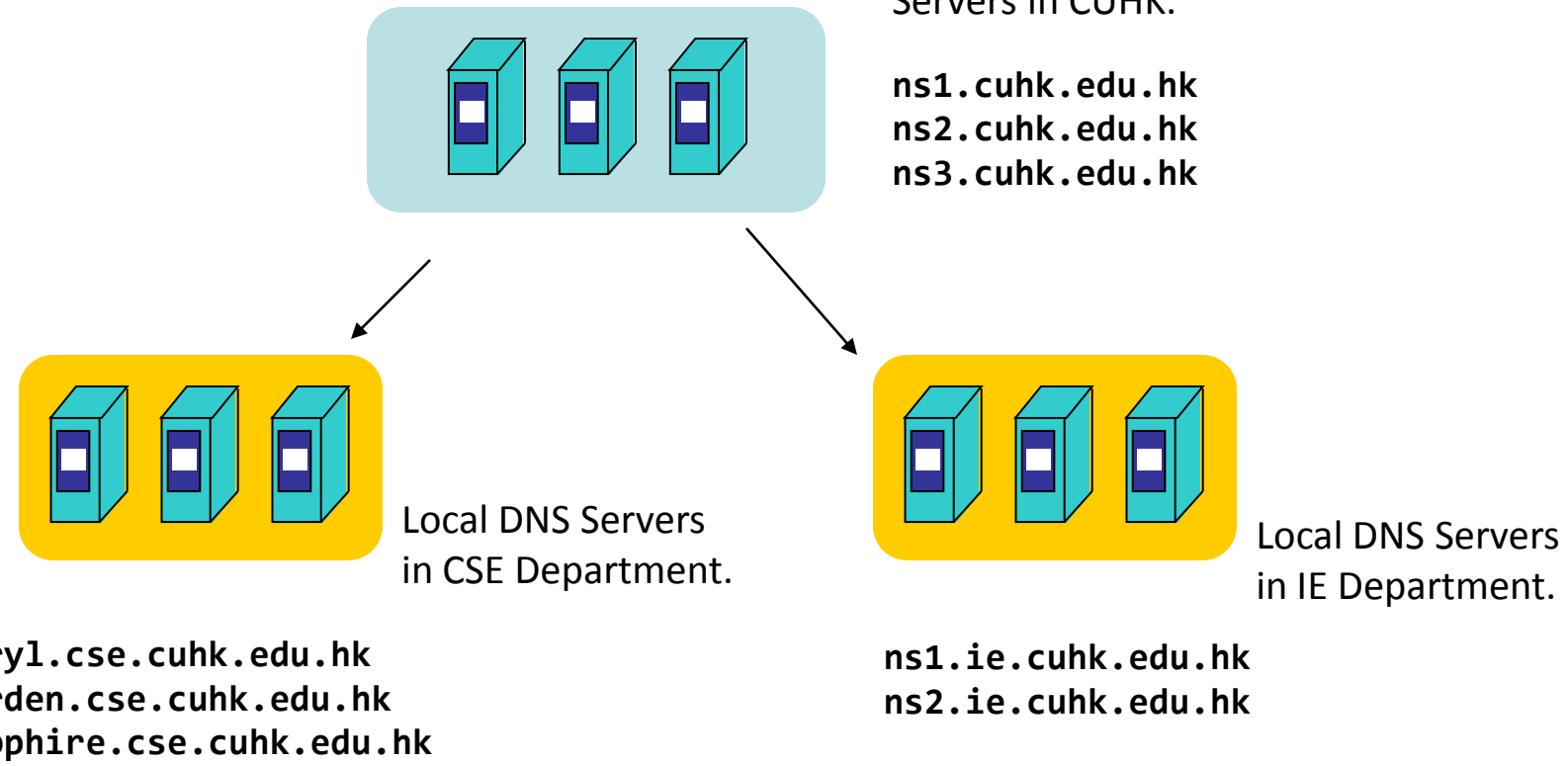
DNS – Server Organization

- How the DNS servers are organized?
  - a layered architecture.



DNS – Server Organization

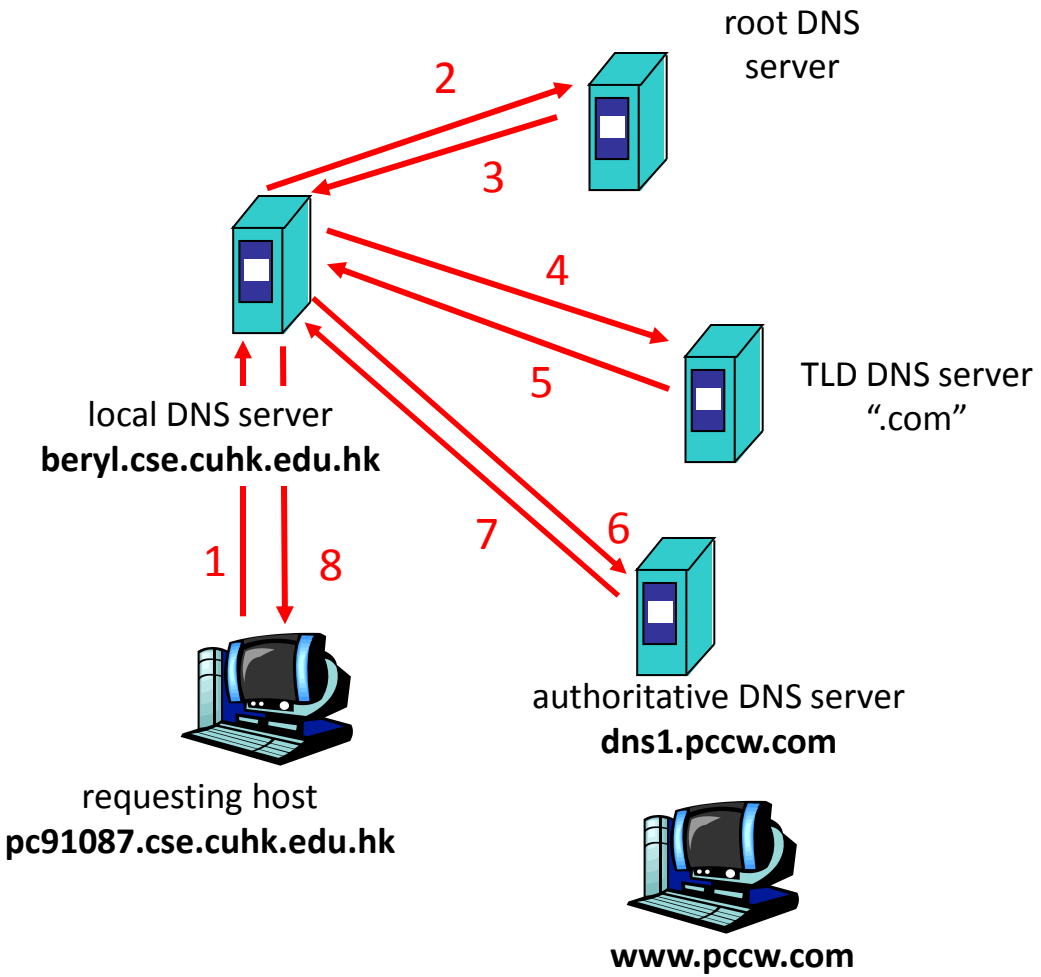
- CUHK Example



DNS – Lookup Example

Looking up: www.pccw.com

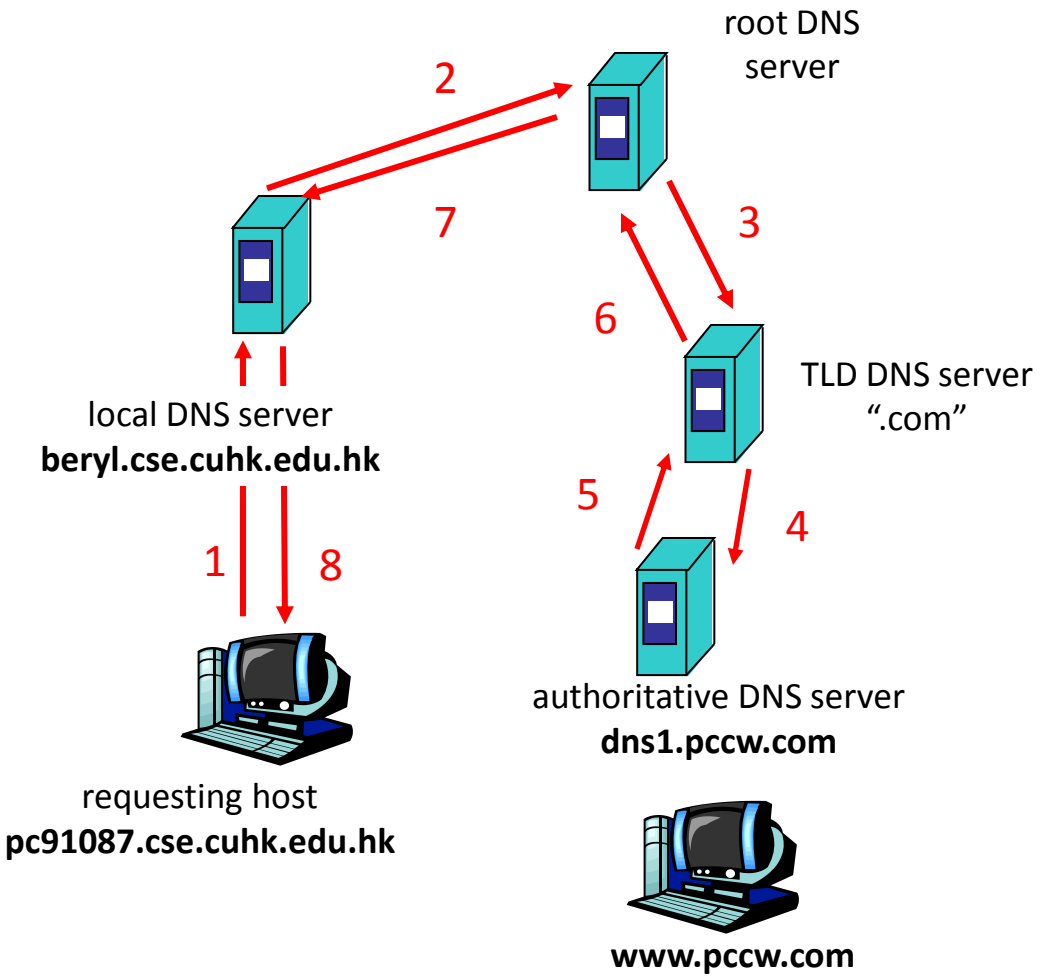
- Iterated query
  - the local DNS server does the following:
  - "I don't know this name, but I ask other servers."
- Not practical.
  - local DNS server has to handle many queries for one translation.



DNS – Lookup Example

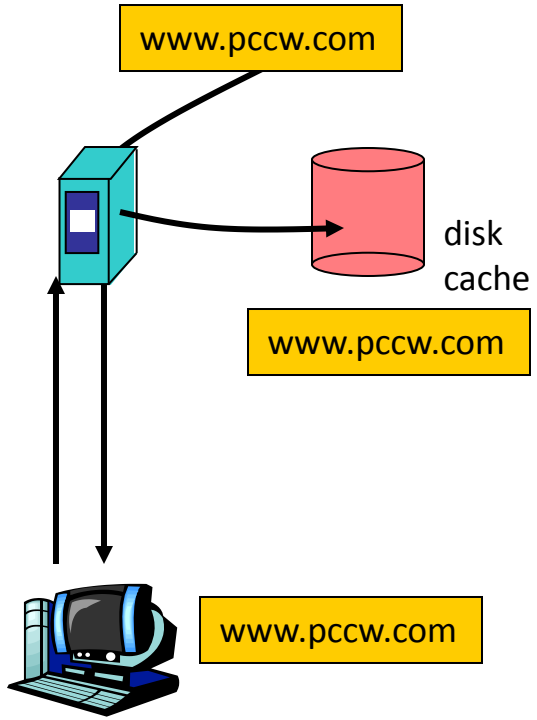
Looking up: www.pccw.com

- Recursive query
  - the local DNS server does the following:
  - "I don't know this name, but I ask my upper servers to search."
- Current practice.
  - Other DNS servers can cache the name record to speed up later queries.



DNS – Caching and Update

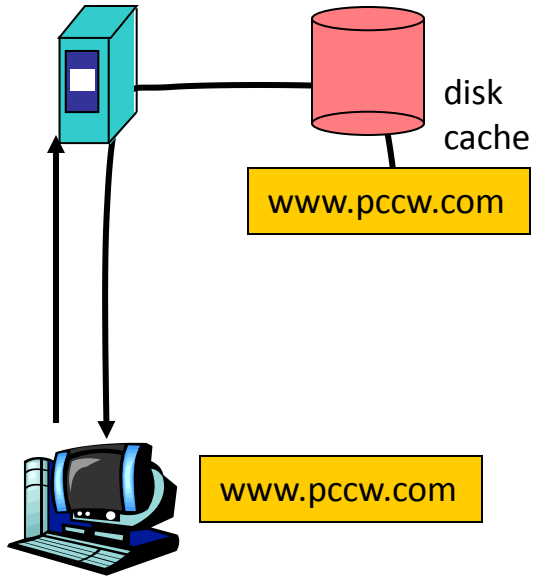
- DNS Cache
  - A cache is a stable storage in a DNS server.
  - The goal is to speed up the DNS queries.
  - After a DNS server has retrieved a name record from other DNS servers, the DNS server stores the record into its cache.



DNS – Caching and Update

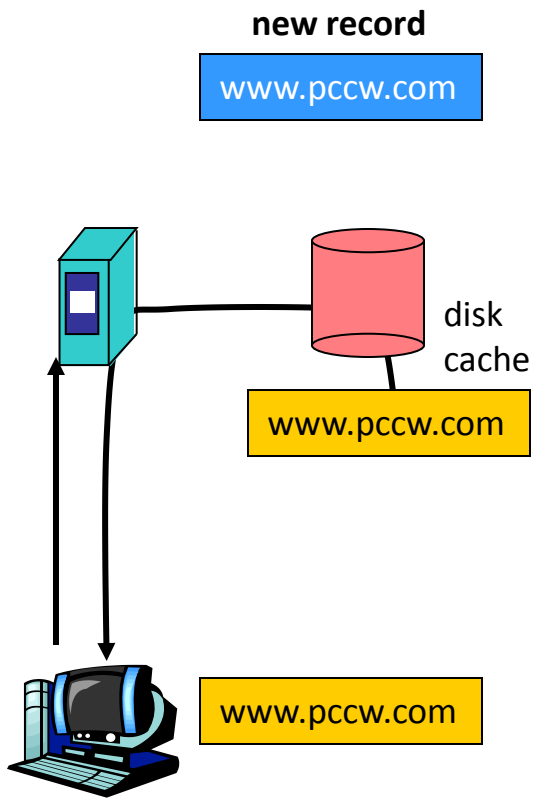
- DNS Cache
  - When the same record is requested, the DNS server uses the record in cache to reply to the request.

- Note that every name record has an expire time. E.g., 1 day.
  - If the record in the cache is expired, the DNS server has to look up for the record recursively again.



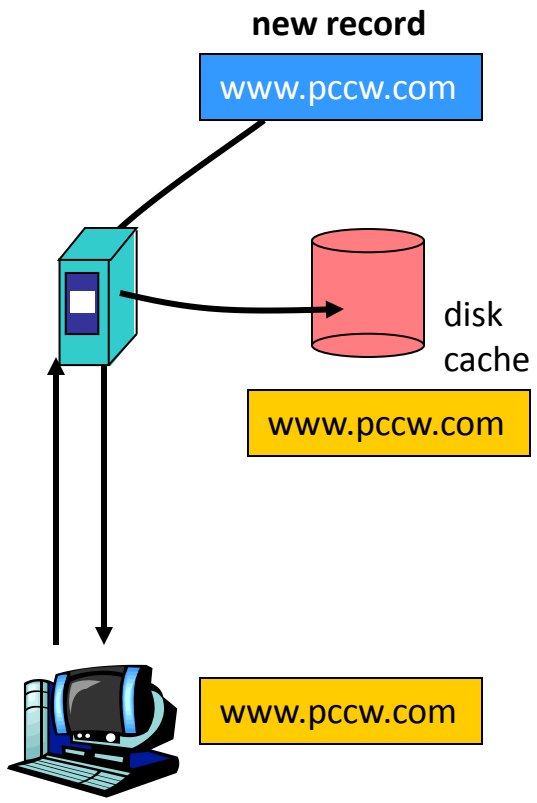
DNS – Caching and Update

- DNS Update
  - When the IP address of a name is updated, what will happen?
  - Since every DNS server has its own cache, the DNS server still uses the record in cache until it is expired.



DNS – Caching and Update

- DNS Update
  - After the record is expired, the new record will be retrieved.
  - We call this DNS update propagation.
  - This process usually takes a couple of days to finish.



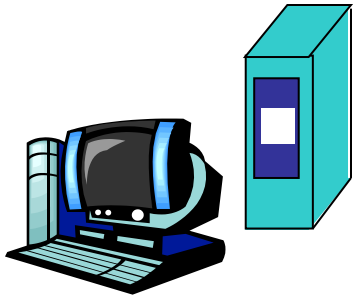
DNS – Summary

Transport Layout Protocol	UDP
Has header?	Yes.
Header encoding	Binary
Content encoding	ASCII + Binary
Message exchange	Request-response - client is always the first guy to speak.
Connection duration	UDP is “connection-less”; no connection at all.

Let’s do the check using wireshark!

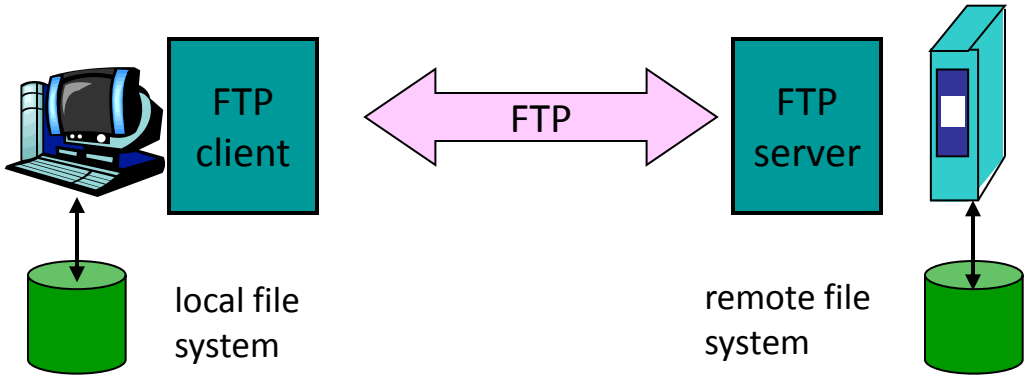
Application layer

- Protocol design:
  - Learn from examples!
    - HTTP
    - DNS
    - FTP



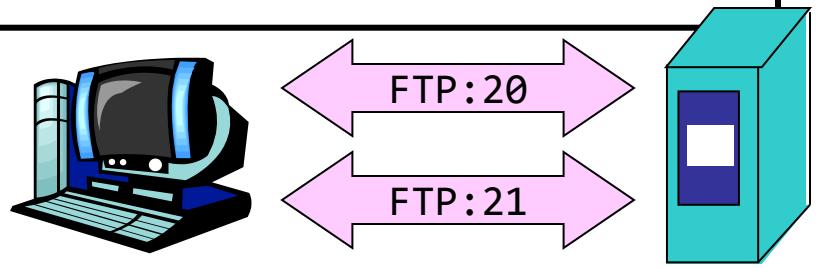
FTP – Introduction

- FTP – File Transfer Protocol
  - A typical client-server model;
  - Goal: to transfer files between the client and the server.
  - TCP connection over ports 20 **and** 21.
  - Standard: RFC 959.



FTP – Features

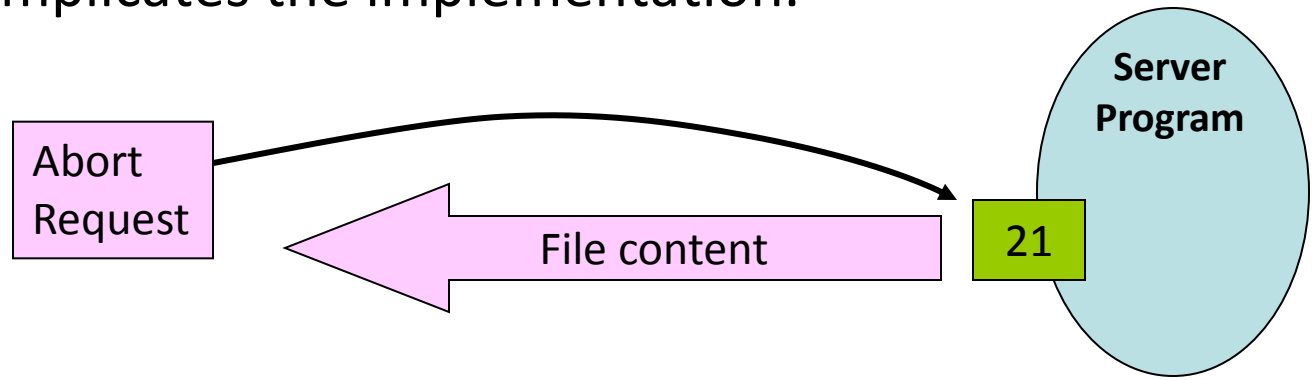
Requires Login	The FTP server reads the password database (i.e., **`/etc/shadow`**) of the server computer. So, FTP server process must be run by root.
Maintain Status	Both the server and the client maintain the login status. E.g., - your identity; - your working directories on both local and remote side.
Two connections	Connection to Port 21: control message. Connection to Port 20: data transfer.



FTP – Why 2 connections?

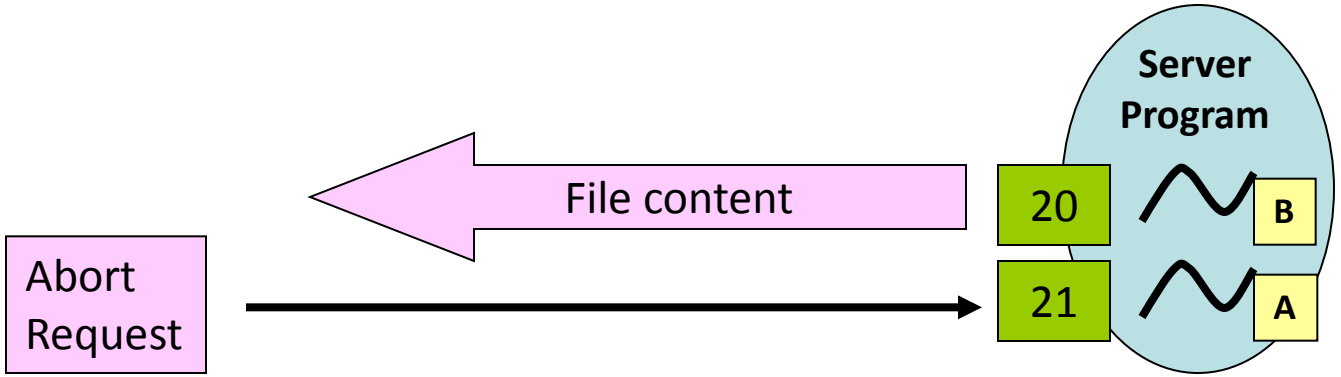
- The catch is: **How to abort gracefully?**
  - Not by killing the client program!

- For example, when the server program is sending a file:
  - If using one TCP connection only, the server program will need to detect
    - whether there are abort requests or not, and at the same time,
    - writing data to the socket.
  - This complicates the implementation.

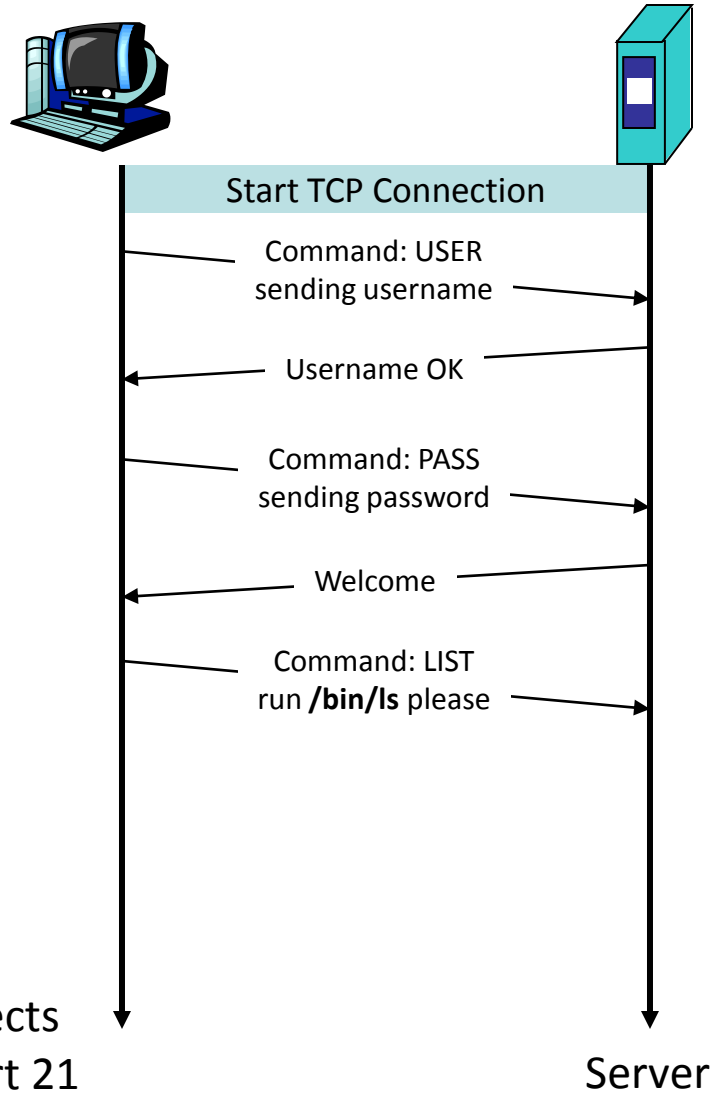


FTP – Why 2 connections?

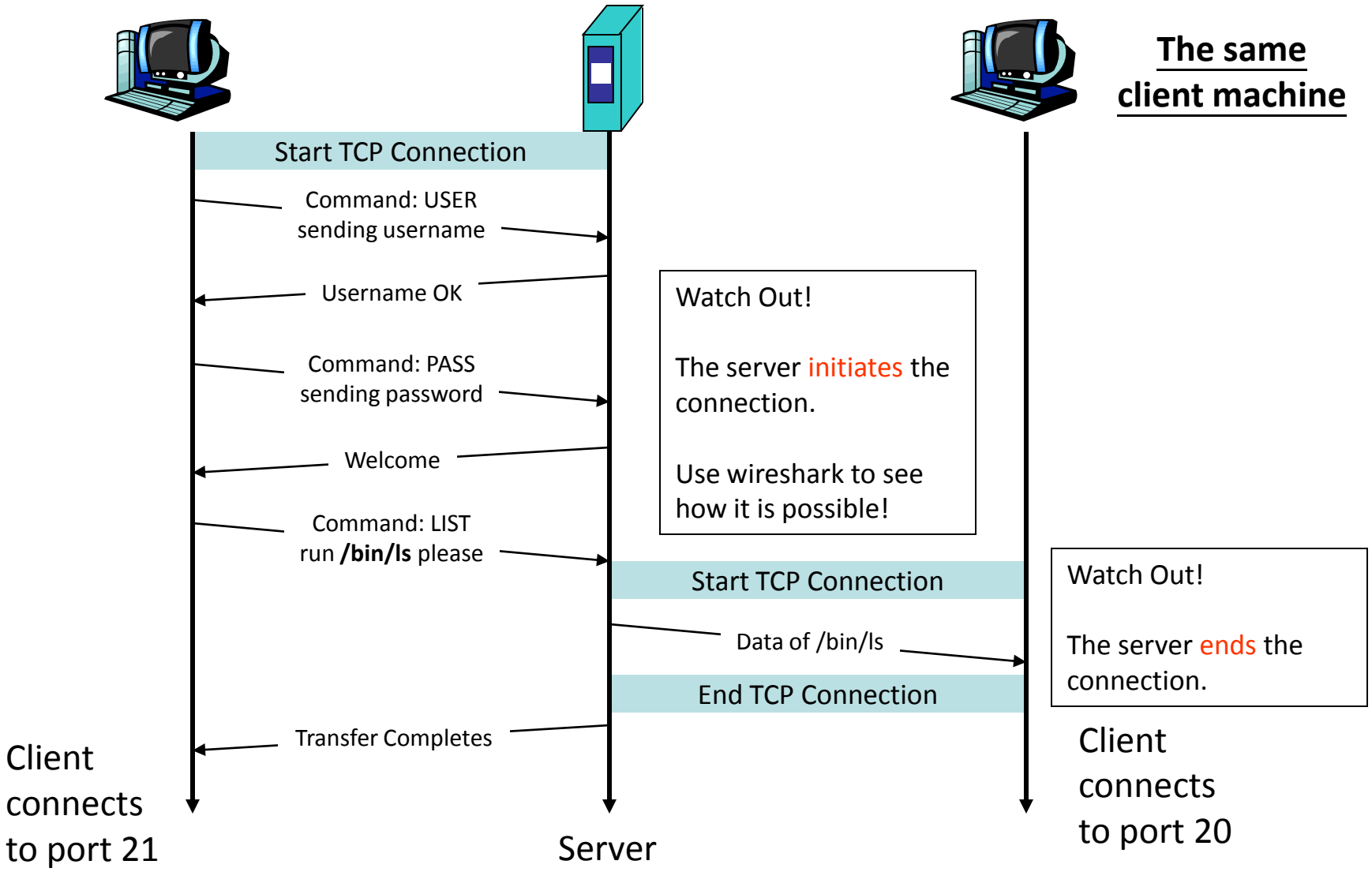
- If two connections are used:
  - The server program will just need to listen on port 21 for incoming request, using thread A.
  - Using thread B to send or receive data.
  - Since reading is a blocking call, it won't take significant overhead in running thread A.



FTP in action



FTP in action



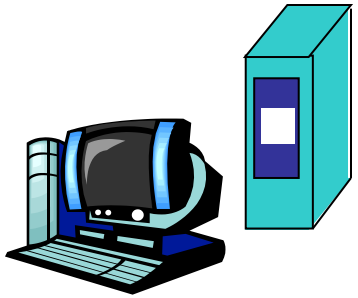
FTP – Summary

Transport Layout Protocol	FTP
Has header?	No. Command-based communication: (command, content) pair. Command is terminated by a space. Content is terminated by a newline character.
Control channel encoding	ASCII
Data channel encoding	Command "**`ascii`**" will send data in ASCII only. Command "**`binary`**" will send data in binary only.
Message exchange	Persistent & stateful connection.
Connection duration	Control channel: until the "**`bye`**" command is sent. Data channel: start on-demand, and close on finishing.

Let's do the check using wireshark!

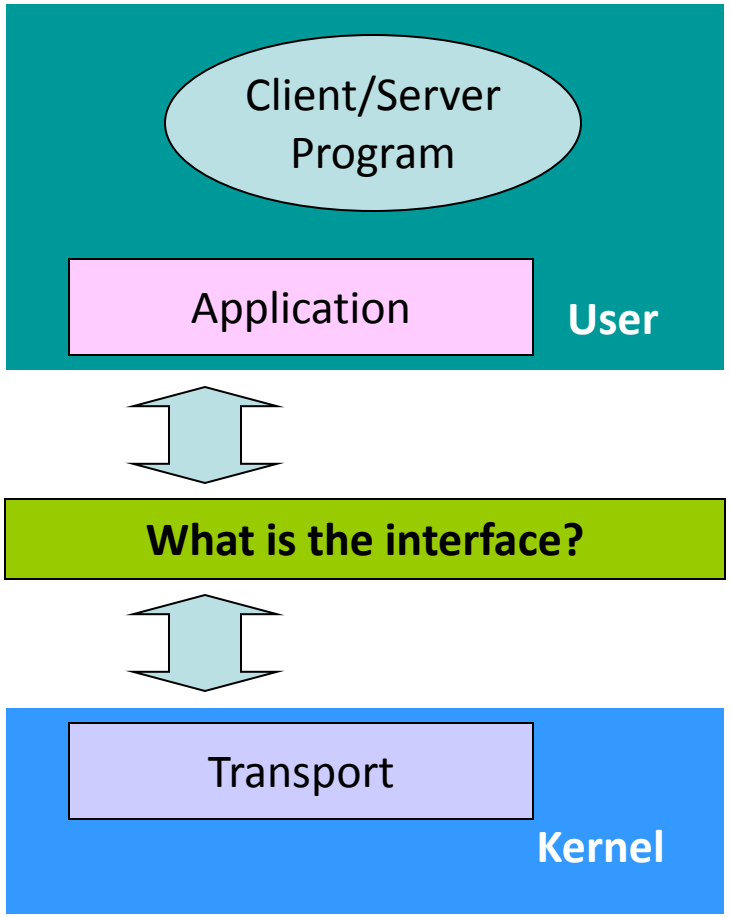
Application layer

- Protocol design:
  - Learn from examples!
- **Socket programming**



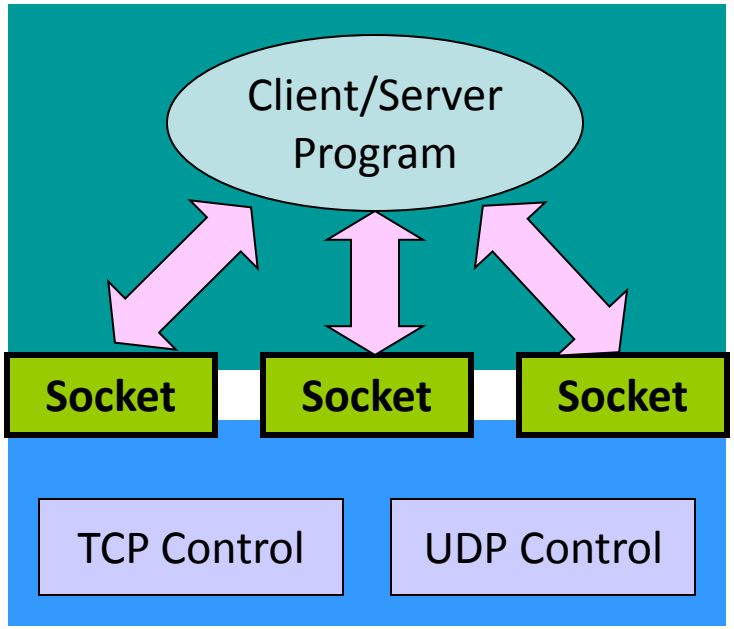
Socket programming – basics

- The interface between the application layer and the transport layer is system calls?
  - Yes!
  - But, we have a better name when working with those network-related system calls.
  - We call it the socket programming.



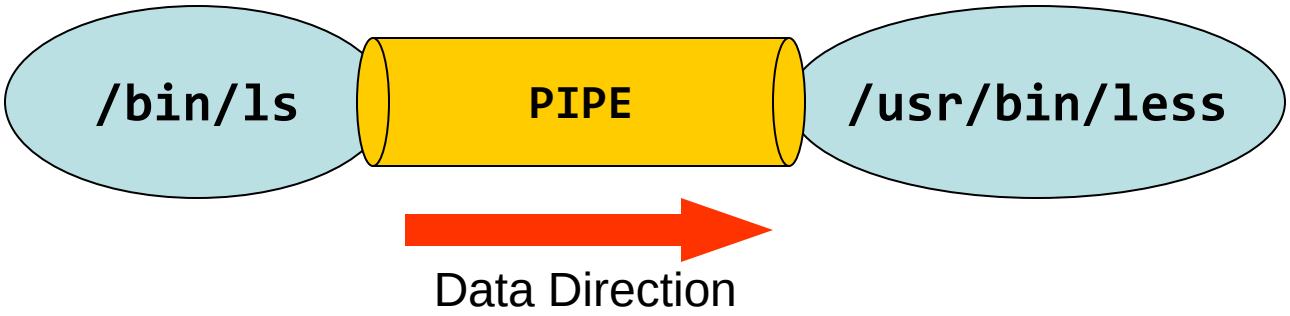
Socket programming – basics

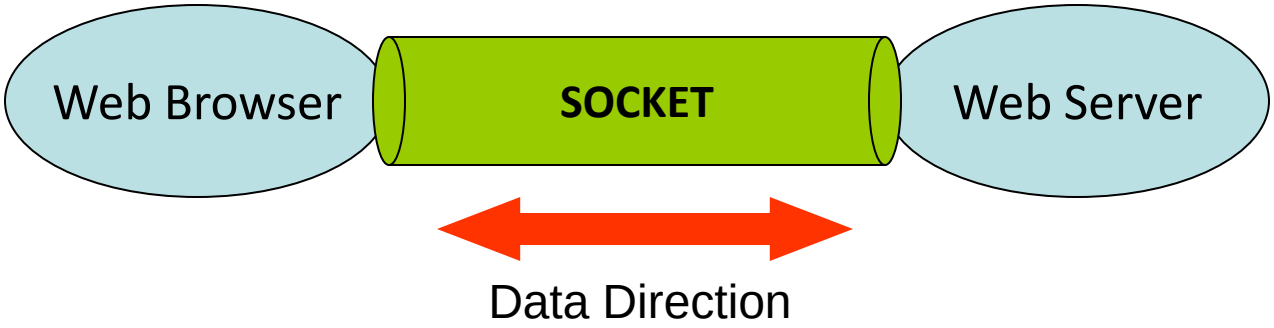
- What is a socket?
  - Socket represents a network resource.
  - It is an interface between the process and the kernel, just like the role of the file descriptor.
  - It is also an interface between the application layer and the network layer!



Socket programming – basics

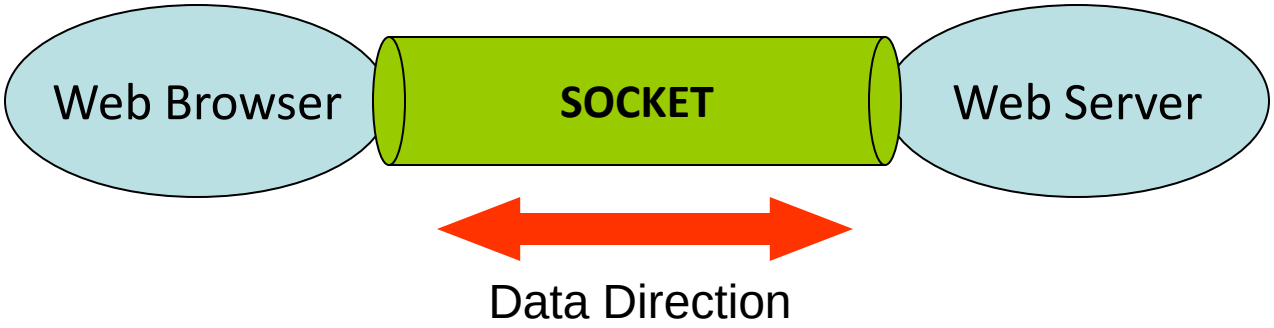
- A socket is a dual-pipe!





Socket programming – basics

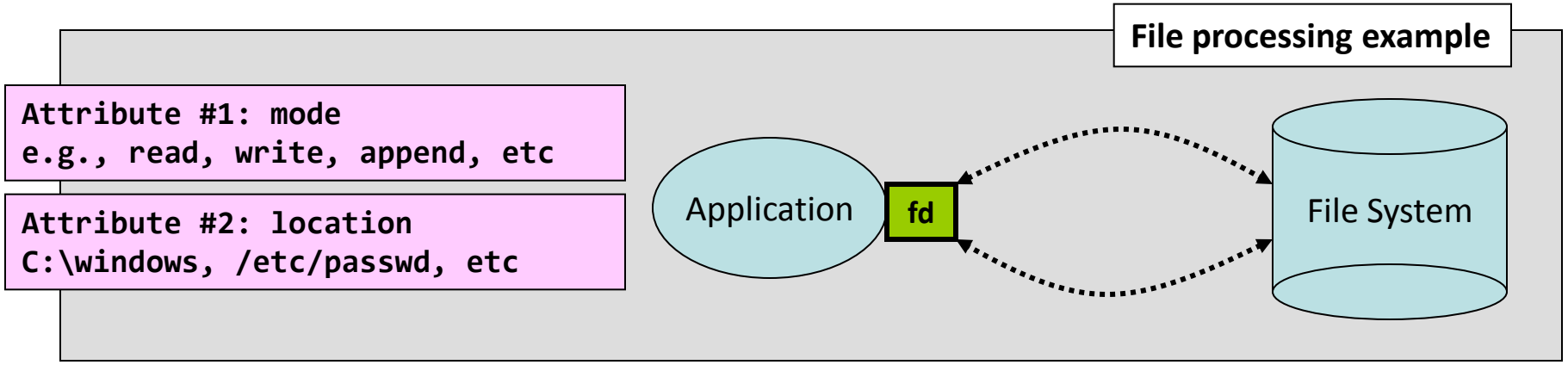
- You can read and write on an opened socket.
  - When you are writing data, you are sending data to the remote host.
  - When you reading data, you are trying to receive data from the remote host.



Socket programming – basics

- Though the underlying layers are handled properly by the kernel, the application still have tell the FS layer some information.

- E.g., when opening a file:
  - an application does not need to handle the FS details, but
  - an application has to tell the kernel:
    - pathname of the target file, and
    - the access mode (read-only, write-only, etc).



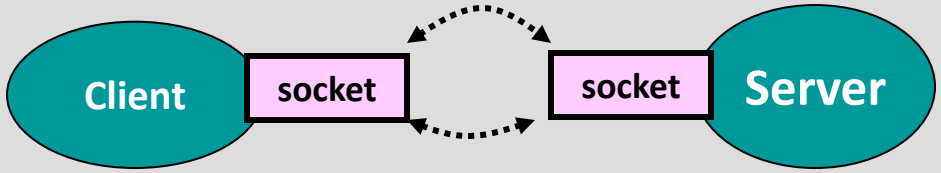
Socket programming – basics

- Though the underlying layers are handled properly by the kernel, the application still have tell the network layer some information.

- When opening a socket:
  - an application has to tell the kernel:
    - IP address of the target server, and
    - Connection mode
      - TCP: Reliable connection or UDP: Unreliable connection.

```
Attribute #1: mode

Choice: TCP or UDP?
```

```
Attribute #2: location

Destination IP address and port number.
```

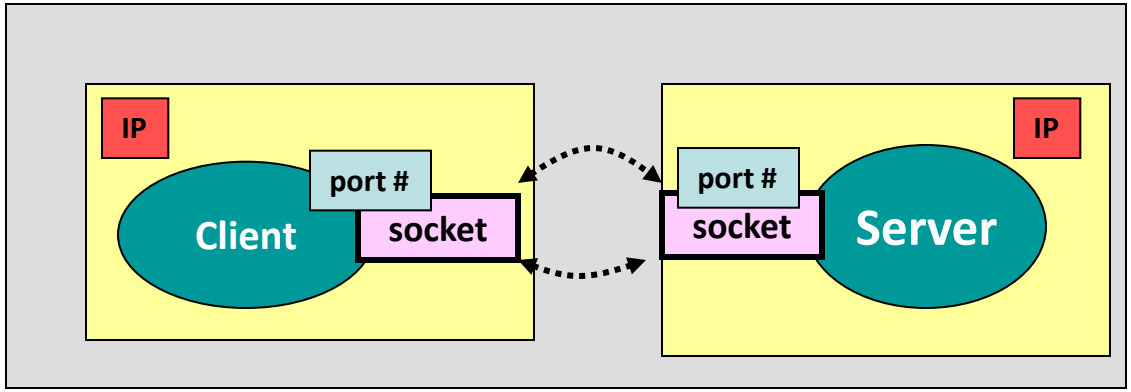


Socket programming – basics

Socket	An application layer entity. It represents a "thing" for connecting two applications.
Port number	A transport layer identifier. It identifies a connection. **A port number is bound to a socket**. An application, e.g., the client, can open more than one port in order to create more connections.
IP address	A network layer identifier. It represents the machine itself, but a machine can have more than one IP address.

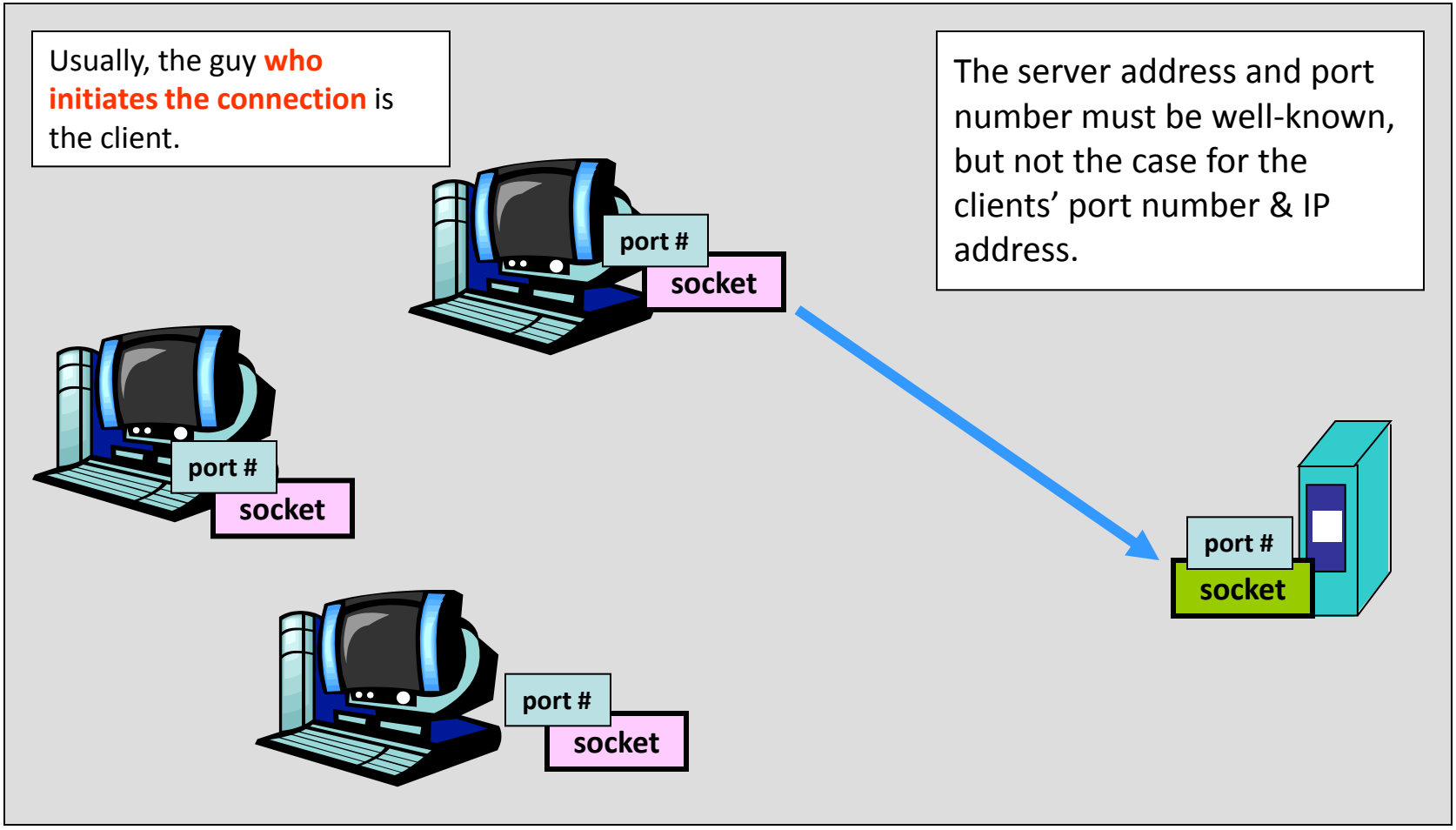
It may be confusing...

An application program needs to know **identifiers of lower layers** of the opposite side in order to talk.



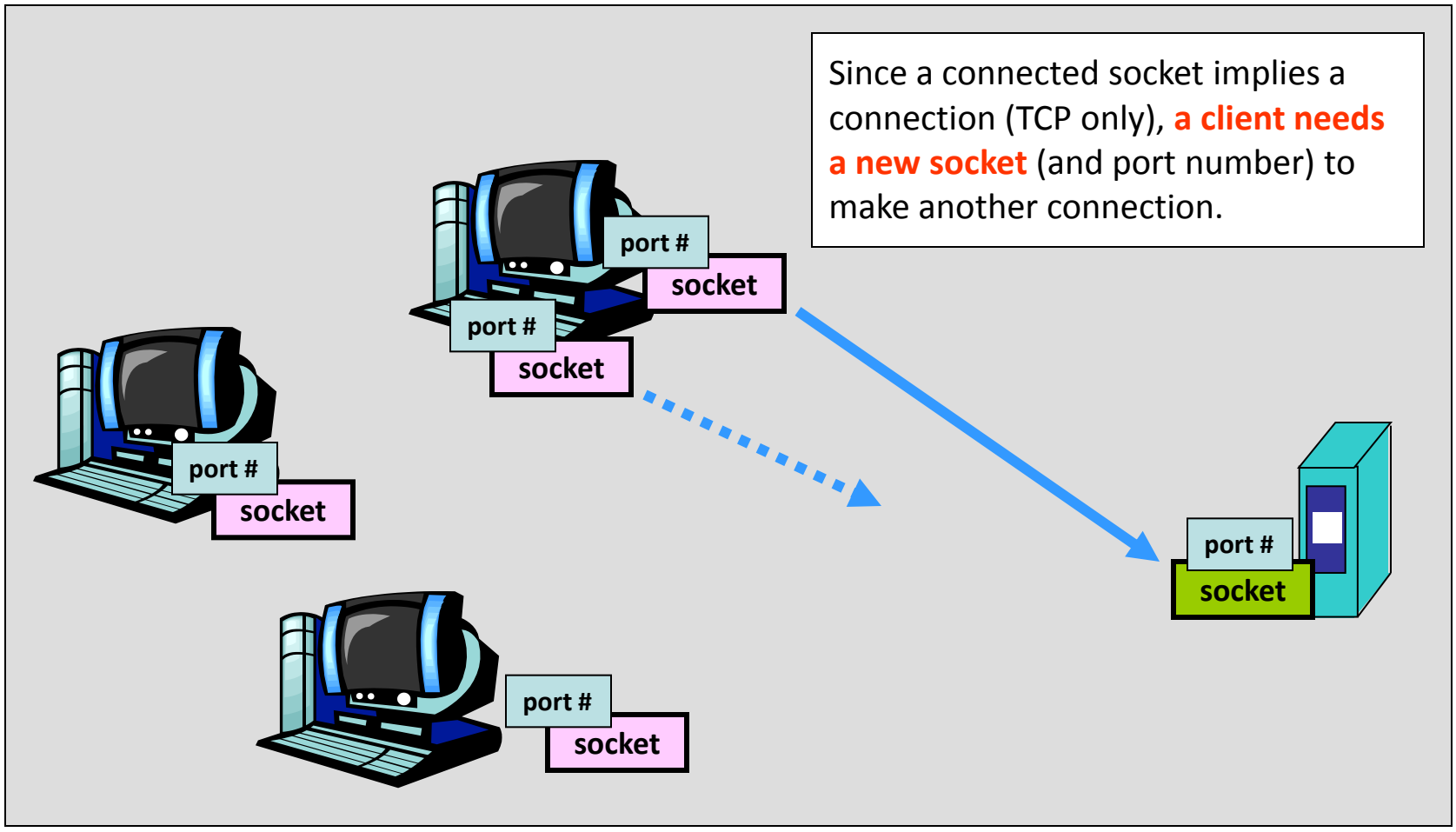
Socket programming – connections

- Many Clients to One Server



Socket programming – connections

- Many Clients to One Server

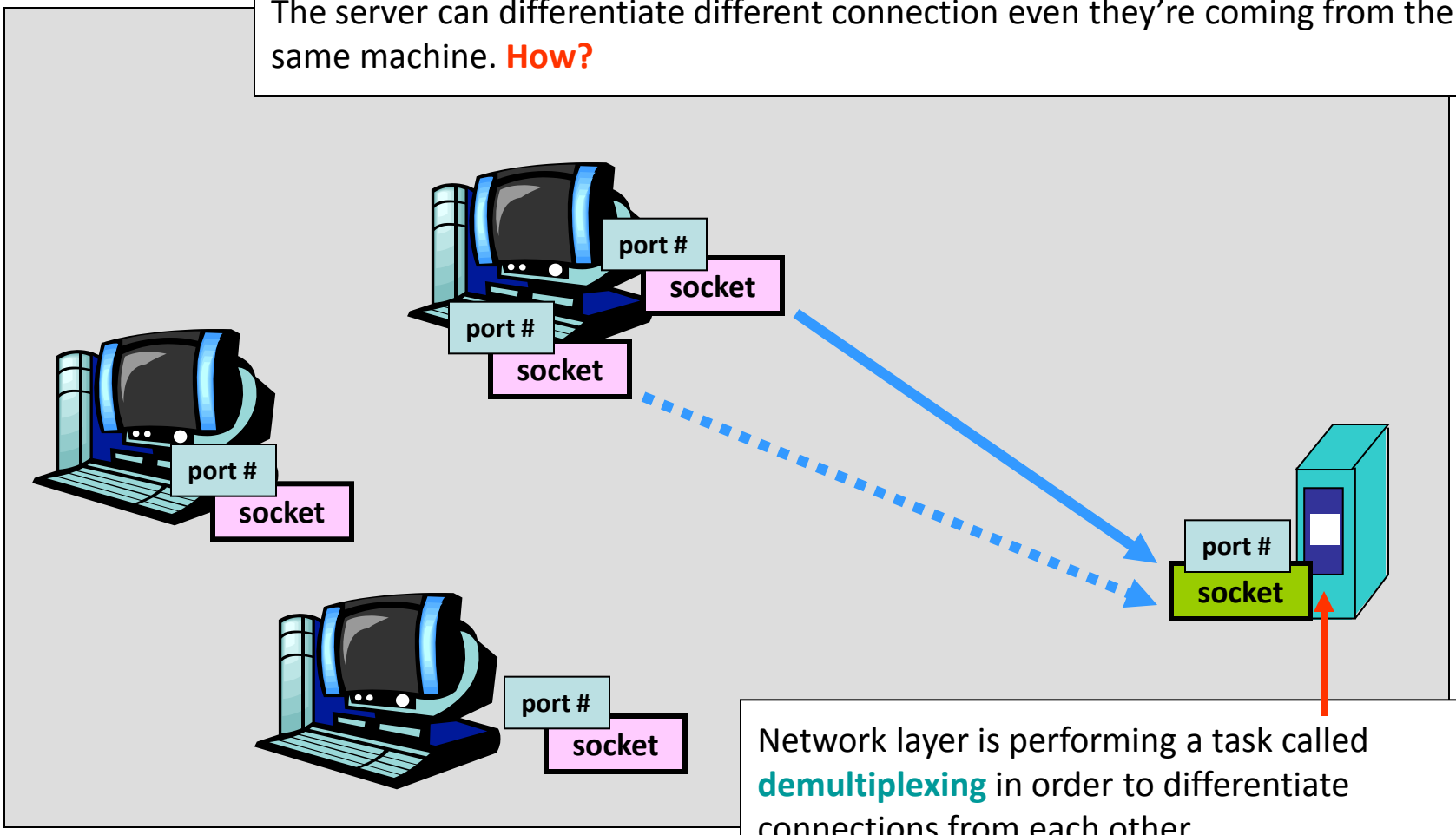


Socket programming – connections

- Many Clie

The server is always using **the same port to accept connections**.

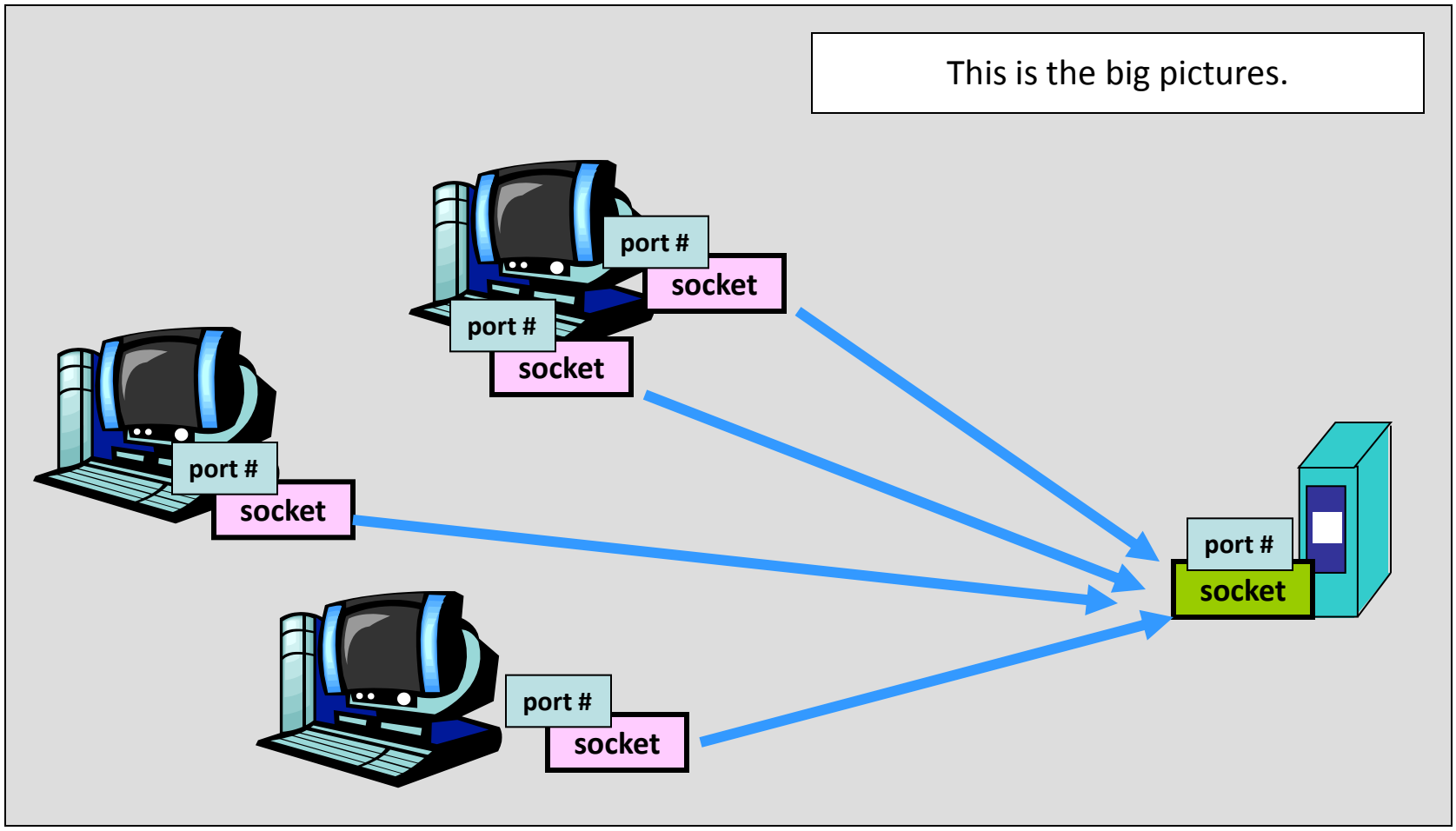
The server can differentiate different connection even they're coming from the same machine. **How?**



Network layer is performing a task called **demultiplexing** in order to differentiate connections from each other.

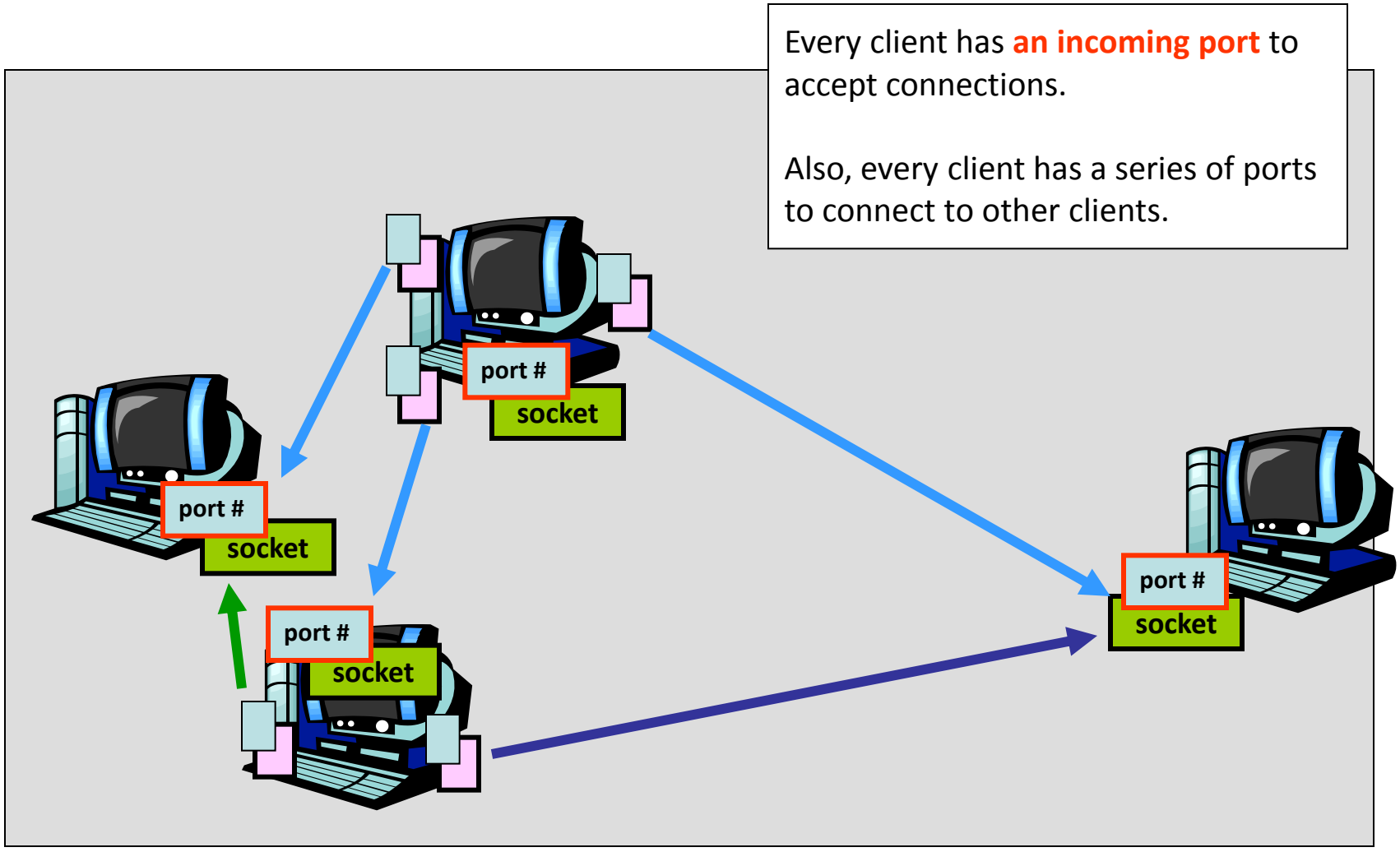
Socket programming – connections

- Many Clients to One Server

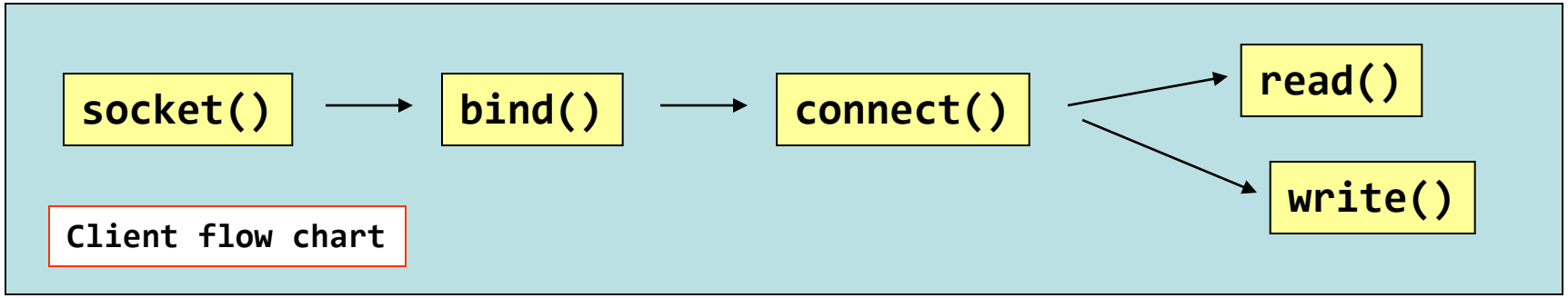


Socket programming – connections

- Peer-to-Peer.



Programming stuffs...client



Step (1). [**socket()**]
- Create a socket.

Step (2). [**bind(); optional**]
- Assign the socket a port number.
- Skip this step and will have a random port number assigned.

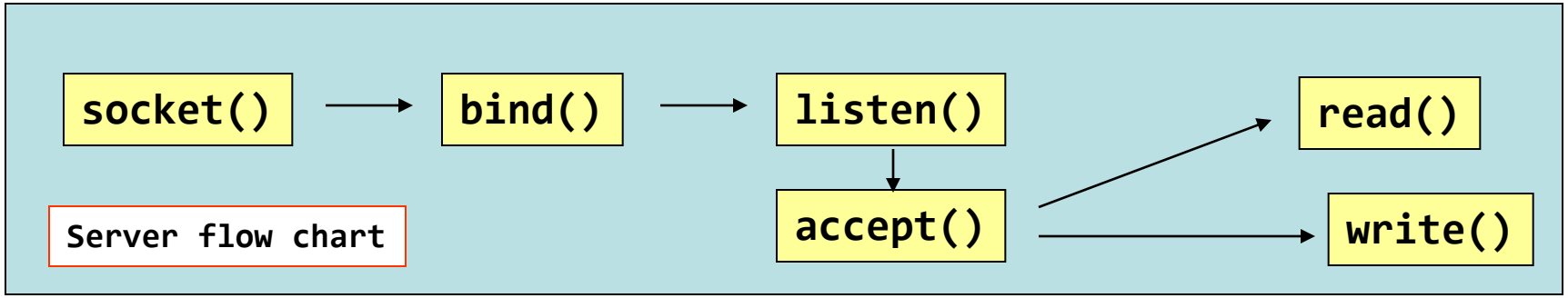
Step (3). [**connect()**]
- Connect to the remote server.
- It is a blocking system call.

read() – to receive data.
write() – to send data.
close() – to close the socket

Important System calls	
Client side	**Server side**
socket()	
bind()*	**bind()**
connect()	**listen() and accept()**

*** means optional.**

Programming stuffs...server



Step (1) & Step(2) [you know them now.]

Step (3). [**listen()**]
- It sets the port to be listening to incoming connections, for TCP only.

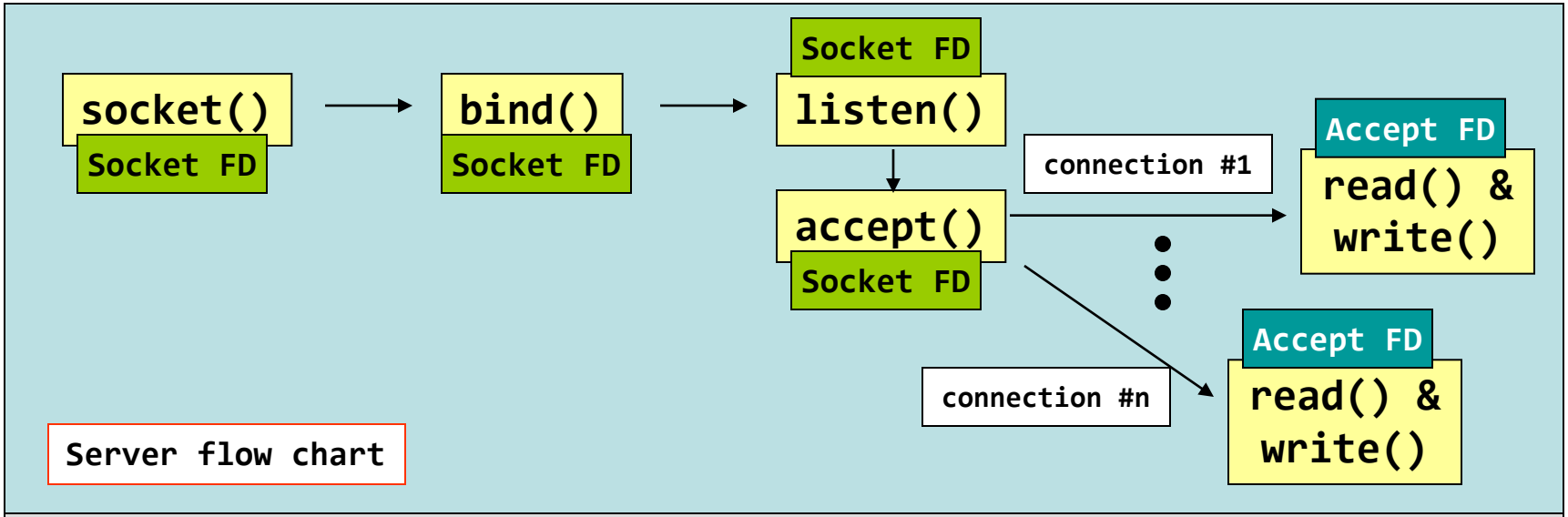
Step (4). [**accept()**]
- Accept incoming connections.
- A blocking system call.

read() – to receive data.
write() – to send data.
close() – to close the socket

Important System calls	
Client side	Server side
socket()	
bind()*	bind()
connect()	listen() and accept()

* means optional.

Programming stuffs...about **accept()**



An interesting thing about **accept()** is the creation of a **new file descriptor**!

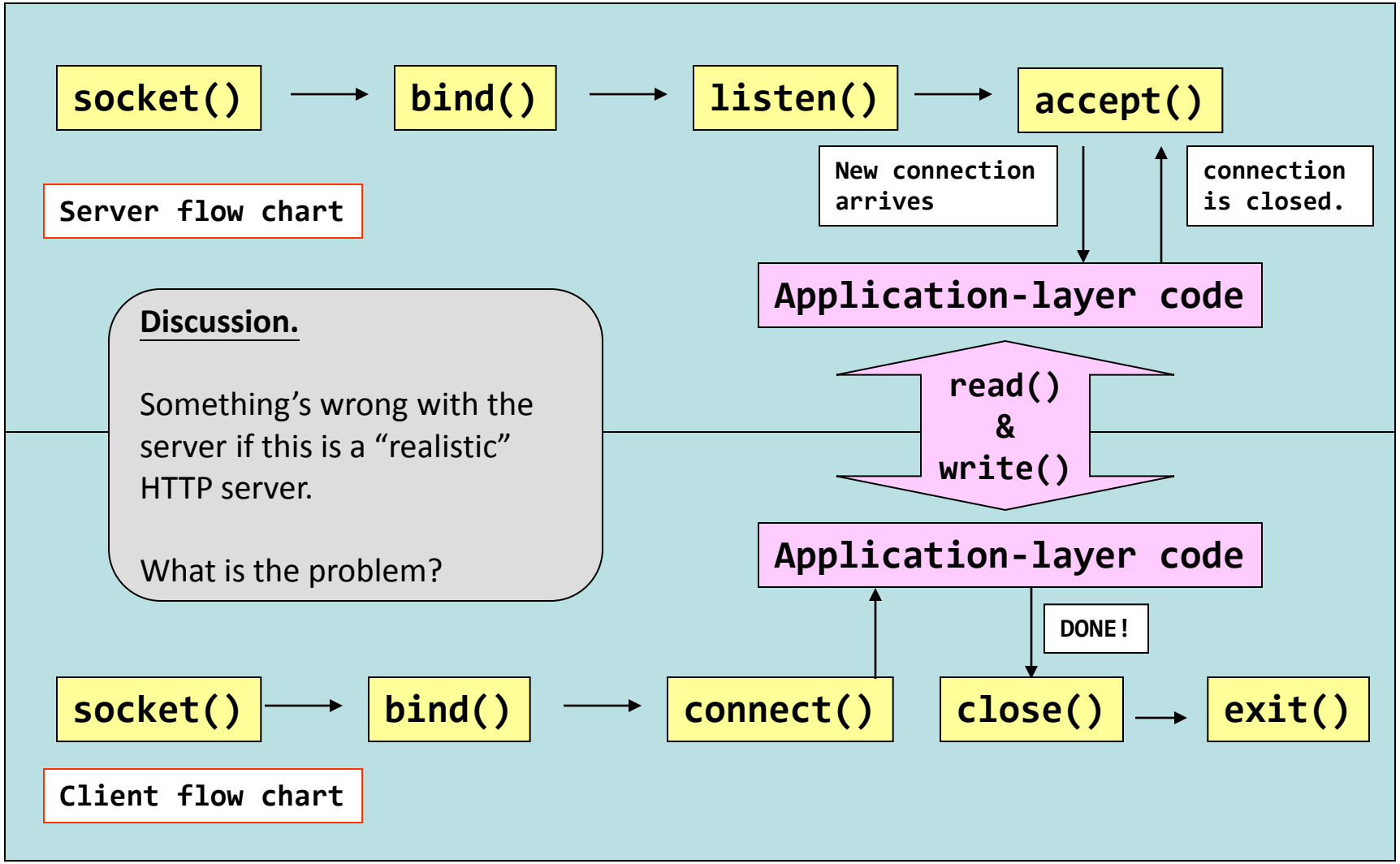
Application layer's point of view.

Good! It provides each connection a new handler and we can distinguish every connection!

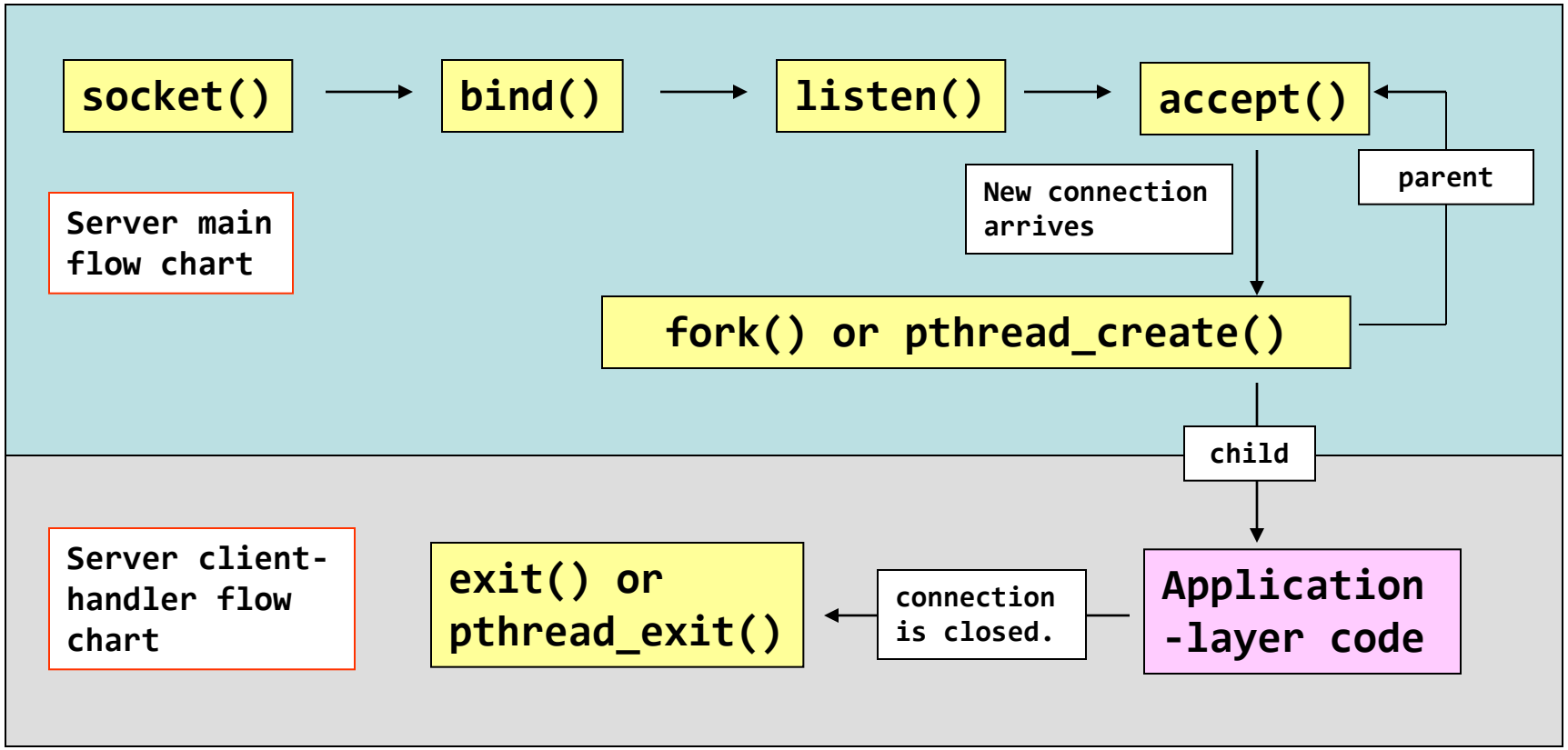
Transport layer's point of view.

Well, for TCP, every FD points to an unique TCP control structure. It is a necessary for reliable data transfer!

Programming stuffs...complete flow?



Programming stuffs...server parallelization



Programming stuffs...`read()` system call

- A sidetrack:
  - Something that we have missed in the OS course...

```
int main(void) {
    char buf[10];
    int ret;
    while(1) {
        ret = read(fileno(stdin), buf, 10);
        printf("ret = %d bytes.\n", ret);
        if(ret <= 0)
            break;
        write(fileno(stdout), buf, ????);
    }
    printf("bye!\n");
}
```

Question 1. Can I hard code the last argument to 10?

Question 2. What is the program output with the following input?

```
abc
efg
^D
```

Programming stuffs...**`read()`** system call

- A sidetrack:
  - Something that we have missed in the OS course...

```
int main(void) {
    char buf[10];
    int ret;
    while(1) {
        ret = read(fileno(stdin), buf, 10);
        printf("ret = %d bytes.\n", ret);
        if(ret <= 0)
            break;
        write(fileno(stdout), buf, ret);
    }
    printf("bye!\n");
}
```

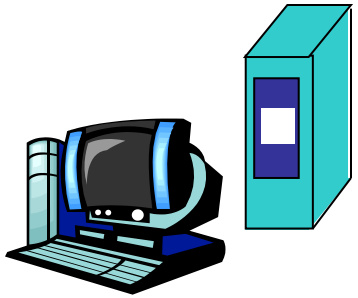
Remember, kernel does not buffer the input for you!!
(Not buffered I/O after all)

So, you must check the return value of the **read-related, direct I/O calls**:
- **`read()`**,
- **`recv()`** [similar to read(), but with lots of useful flags].

Later, we'll explain this effect from the angle of TCP control.

Application layer

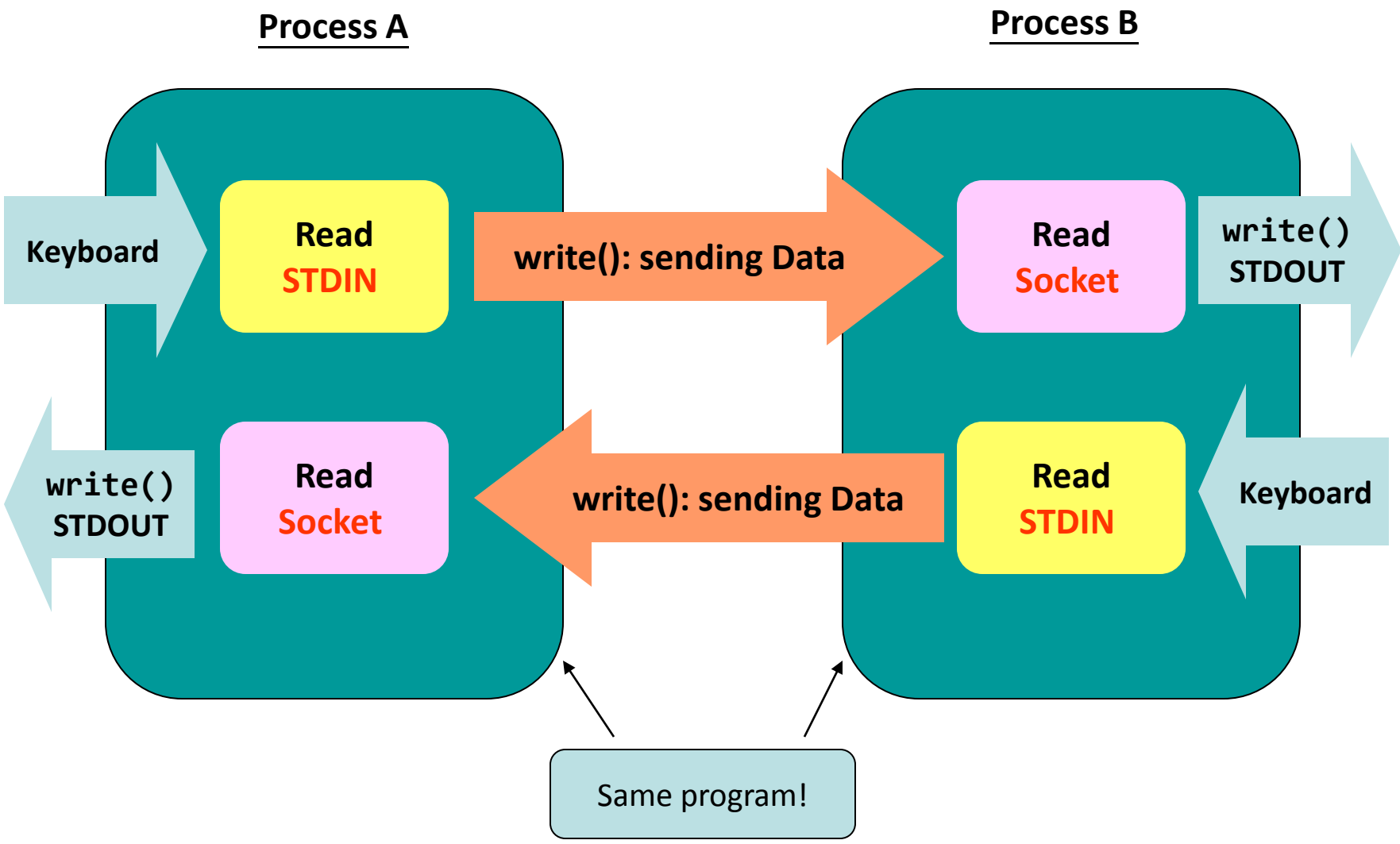
- Protocol design:
  - Learn from examples!
- Socket programming
  - Example: chat room



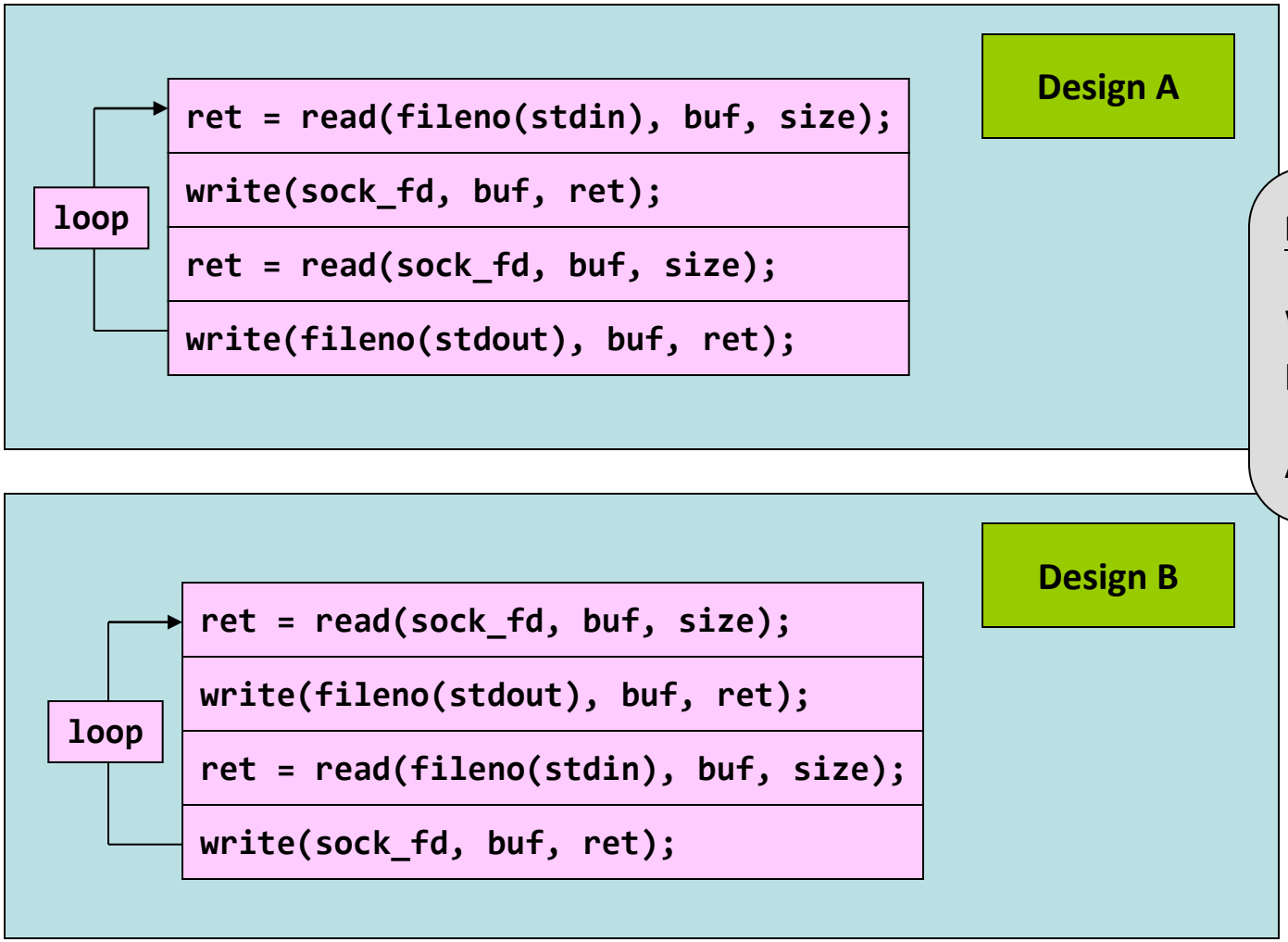
Getting real – chat room

- Requirements:
  - Only two people are involved. Both people use the same chat room program.
  - The program allows the user to type a message using the keyboard, and then the program sends that message to the other side.
  - The program is able to receive messages from the other side, and then displays those messages to the terminal.

Getting real – chat room design



Getting real – chat room design



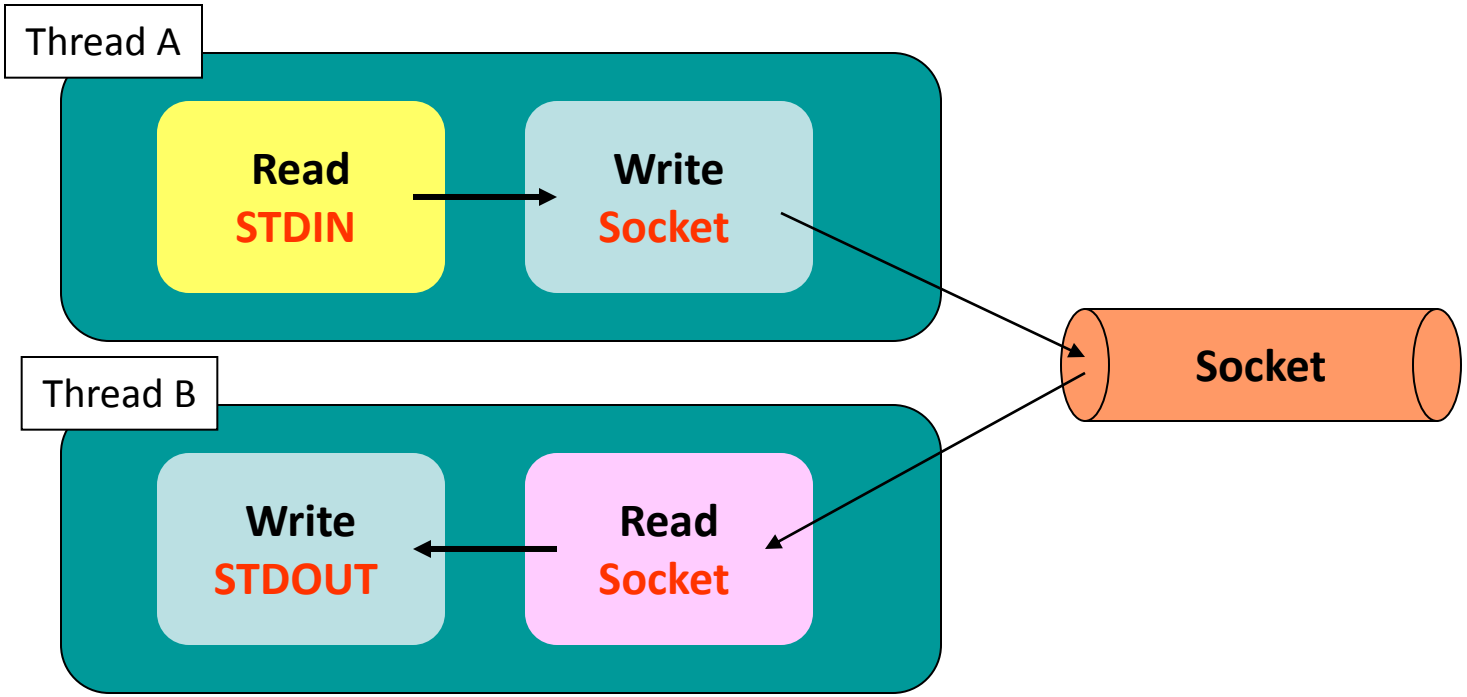
Discussion.

Which one do you prefer?

A or B?

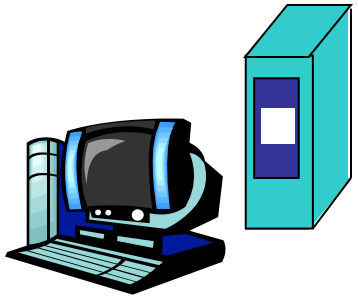
Getting real – chat room with threads?

- Remember, multi-threading is to assign block calls to different threads.
  - But, what if I **hate** multi-threading? Any Plan B?



Application layer

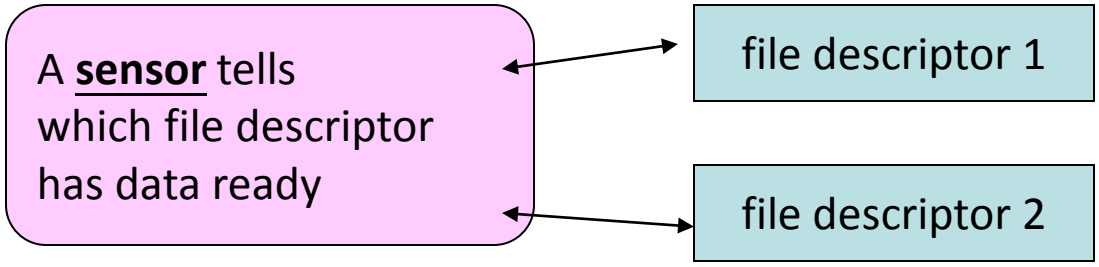
- Protocol design:
  Learn from examples!
- Socket programming
- **I/O multiplexing**



A New Solution?

- Multi-threading? Or, Multi-processes?
  - It works.
  - But, have to take care of problems like process synchronization and mutual exclusion.

- Can I have something like …



I/O Multiplexer – **`select()`**

- **`select()`** is a system call. It checks if:
  - a **`read()`** system call on a set of fds will be blocked;
  - a **`write()`** system call on a set of fds will be blocked;
  - there are exceptions on a set of fds.

```
int select(int nfds,
           fd_set *read_fds,              /* fds for read */
           fd_set *write_fds,            /* fds for write */
           fd_set *except_fds,       /* fds for exceptions */
           struct timevat *timeout);     /* timeout period */
```

`fds – file descriptors.`

I/O Multiplexer – **`select()`**

- **`select()`** is useful in detecting if any one (or ones) of the set of file descriptors has data ready for reading.
  - Therefore, using **`select()`** is usually called I/O multiplexing.
  - as if to aggregate all the **`read()`** system calls into one.

- In our chat room design,
  - using **`select()`** allows the program to sense which fd(s) has data ready **before** the program actually calls **`read()`**.

`select()` – How to Use?

```
#define STDIN     0

fd_set fds;       /* declaration */
FD_ZERO(&fds);    /* initialization */
struct timeval tv;
```

```
struct timeval {
  /* second */
    long tv_sec;
  /* micro-second */
    long tv_usec;
};
```

```
FD_SET(STDIN, &fds);          /* select will monitor STDIN */
select(STDIN+1, &fds, NULL, NULL, NULL);      /* select is blocked */
```

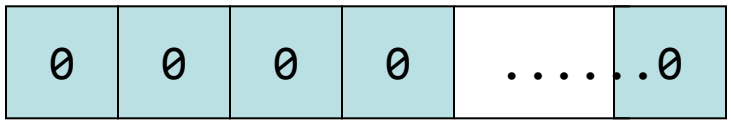
```
FD_SET(STDIN, &fds);          /* select will monitor STDIN */
tv.tv_sec = 0;                /* set the timeout period to 0 sec */
tv.tv_usec = 0;
select(STDIN+1, &fds, NULL, NULL, &tv);       /* select returns immediately */
```

```
FD_SET(STDIN, &fds);          /* select will monitor STDIN */
tv.tv_sec = 5;                /* set the timeout period to 5 sec */
tv.tv_usec = 0;
select(STDIN+1, &fds, NULL, NULL, &tv);       /* select returns after 5 sec */
```

`select()` – What is `fd_set`?

The values are garbage by default.
`FD_ZERO()` initializes all the values to zero.

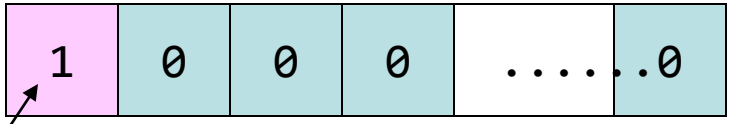
A **`fd_set`** variable is an array:

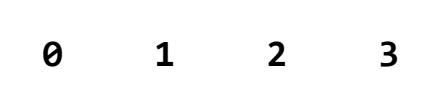


`FD_SETSIZE-1`

file descriptors

The call "**`FD_SET(0, &fds)`**" changes the value of the array element:

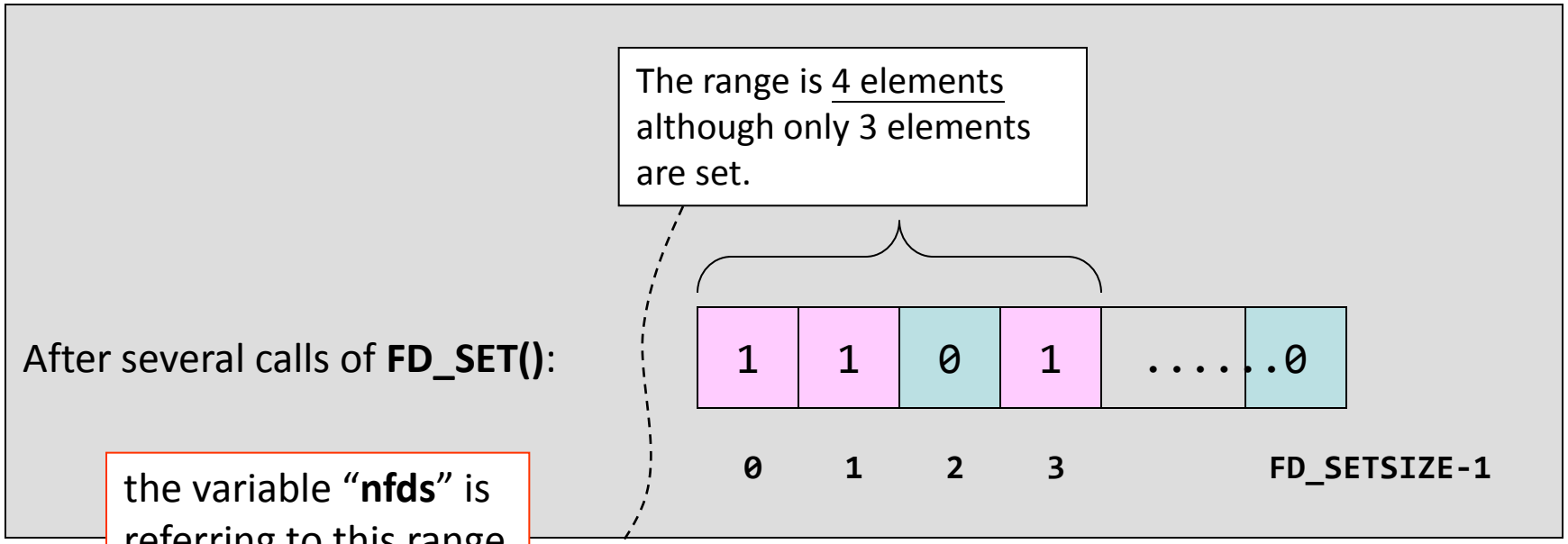




`FD_SETSIZE-1`

In reality, it is NOT as simple as changing the value to 1.

`select()` – What is `fd_set`?




```
int select(int nfds,
           fd_set *read_fds,          /* fds for read */
           fd_set *write_fds,         /* fds for write */
           fdset *except_fds,         /* fds for exceptions */
           struct timevat *timeout);  /* timeout period */
```

select() – What is fd_set?

```
#define MAX(x, y) ((x) > (y) ? (x) : (y))

FD_SET(a, &fds);
FD_SET(b, &fds);
FD_SET(c, &fds);

nfds = MAX( a, MAX(b, c) ) + 1;
select(nfds, &fds, NULL, NULL, NULL);
```

The calculation of nfds.

```
int select(int nfds,
           fd_set *read_fds,          /* fds for read */
           fd_set *write_fds,         /* fds for write */
           fdset *except_fds,         /* fds for exceptions */
           struct timevat *timeout);  /* timeout period */
```

`select()` – What is `fd_set`?

- `fd_set` related marcos:

Function (Macro)	Description
`void FD_CLR(int fd, fd_set *set);`	It clears the given **fd** that is set by `FD_SET()`.
`int FD_ISSET(int fd, fd_set *set);`	It checks if the given **fd** is set in the `fd_set` structure. Returns 0 when false; non-zero when true.
`void FD_SET(int fd, fd_set *set);`	It sets the given **fd** in the `fd_set` structure.
`void FD_ZERO(fd_set *set);`	It clears all the fds in the `fd_set` structure.

`select()` – How to Use?

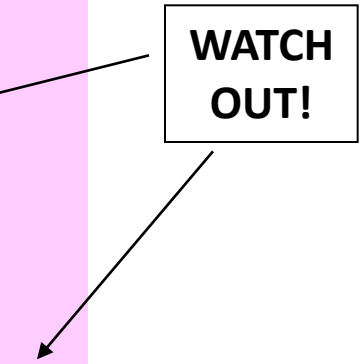
- The return value of **`select()`** is useful:

  -1: error of the system call.

  0: nothing has changed when timeout expires.

  otherwise: the number of fds changed.

```
FD_SET(STDIN, &fds);
tv.tv_sec = 5;
tv.tv_usec = 0;
while(1) {
    rtn = select(STDIN+1, &fds, NULL, NULL, &tv);
    if(rtn == -1) {
        perror("select()");
        exit(1)
    }
    if(rtn == 0) {
        printf("No input after %d sec.\n", tv.tv_sec);
    }
    else
        break;
}
......
```



`select()` – How to Use?

- The timeout value will be changed to the remaining time before **`select()`** returns.
  - E.g., it becomes 0 when there is no change in the monitored fds.

```
FD_SET(STDIN, &fds);
while(1) {
    tv.tv_sec = 5;
    tv.tv_usec = 0;

    rtn = select(STDIN+1, &fds, NULL, NULL, &tv);
    if(rtn == -1) {
        perror("select()");
        exit(1)
    }
    if(rtn == 0) {
        printf("No input after %d sec.\n", tv.tv_sec);
    }
    else
        break;
}
......
```

moved inside the while loop.

`select()` – How to Use?

- The **`fd_set`** variable will also be changed by the **`select()`** system call.
  - If the monitored fds has changed, it will be set.
  - Otherwise, it is not set.

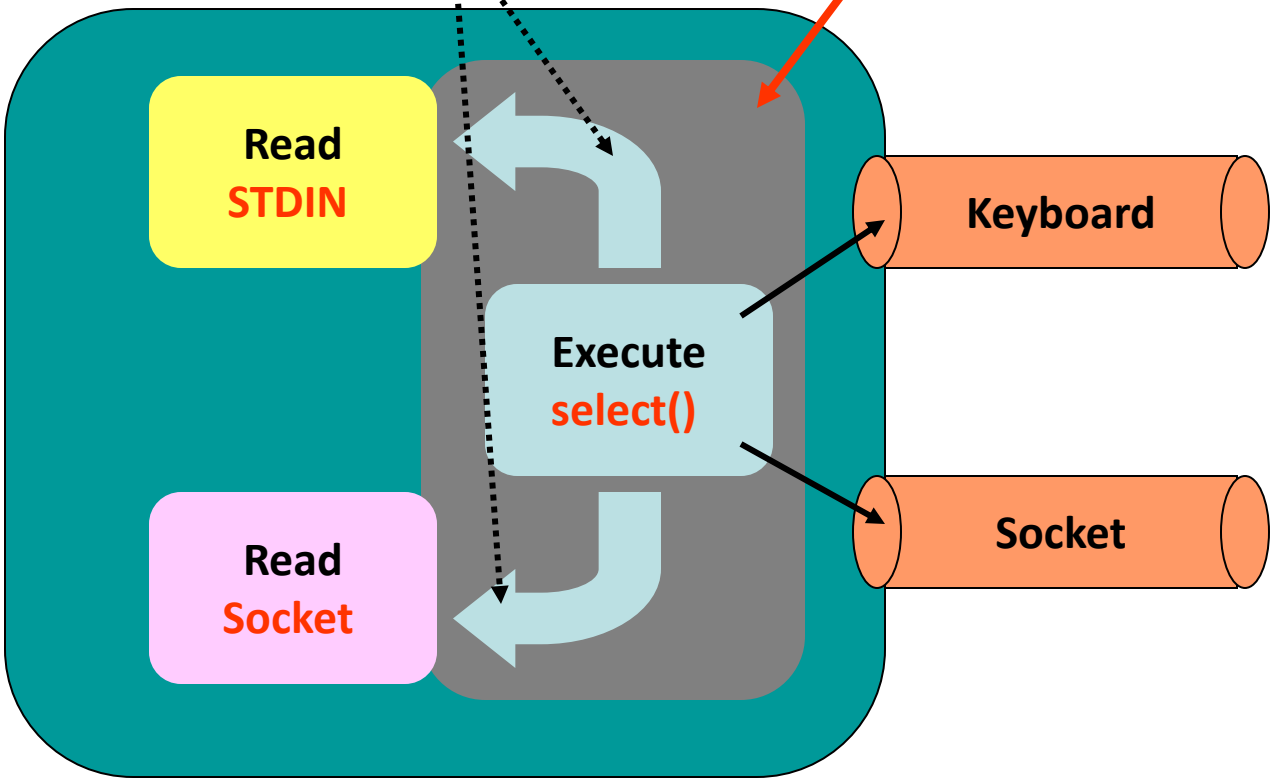
```
while(1) {
    FD_SET(STDIN, &fds);
    tv.tv_sec = 5;
    tv.tv_usec = 0;

    rtn = select(STDIN+1, &fds, NULL, NULL, &tv);
    if(rtn == -1) {
        perror("select()");
        exit(1)
    }
    if(rtn == 0) {
        printf("No input after %d sec.\n", tv.tv_sec);
    }
    else
        break;
}
......
```

moved inside the while loop.

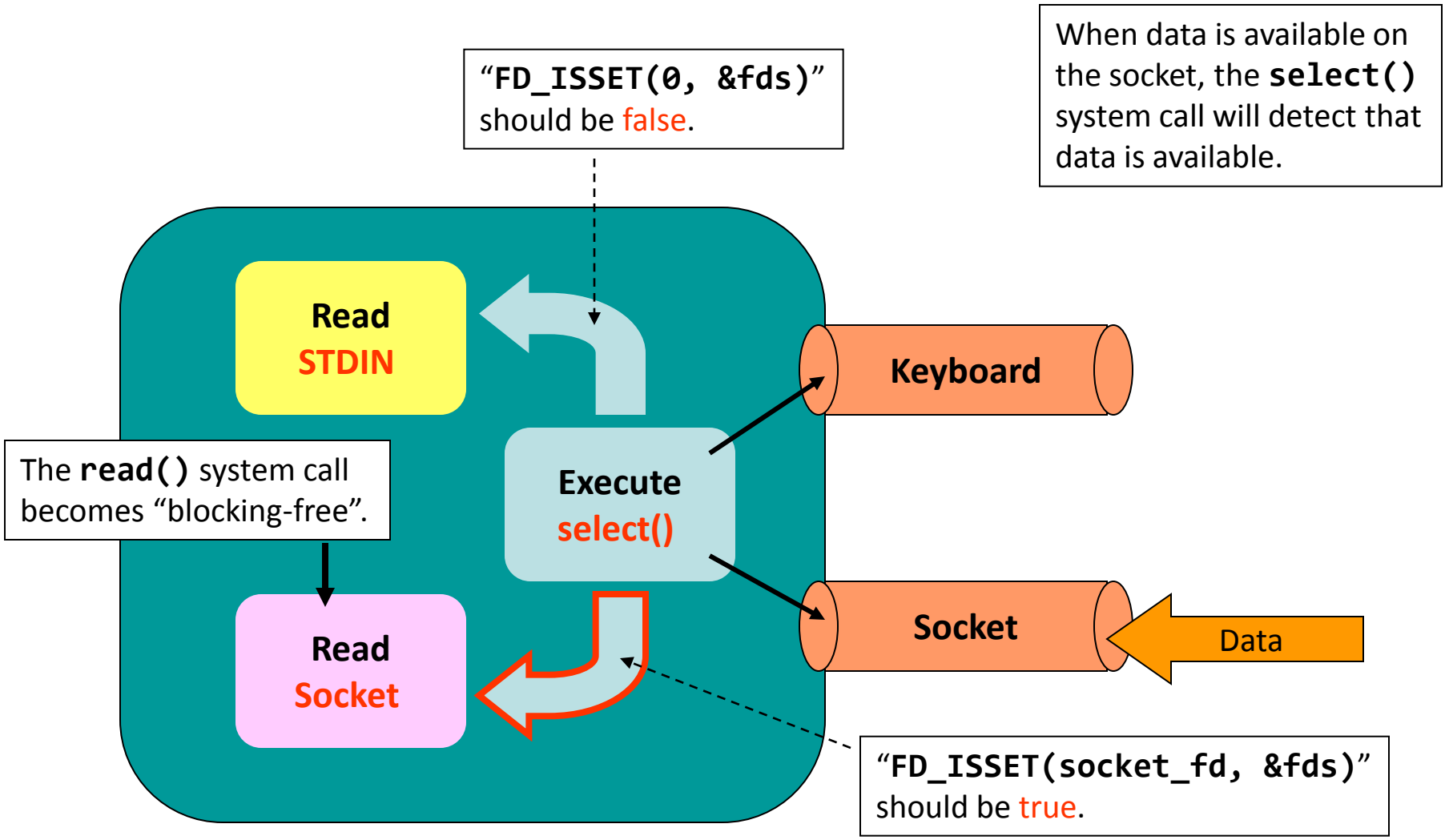
Change in Chat Room Design

The flow of control is decided by the value set by **`select()`**.

The **`select()`** system call is blocked until data is detected on either one of the file descriptors.

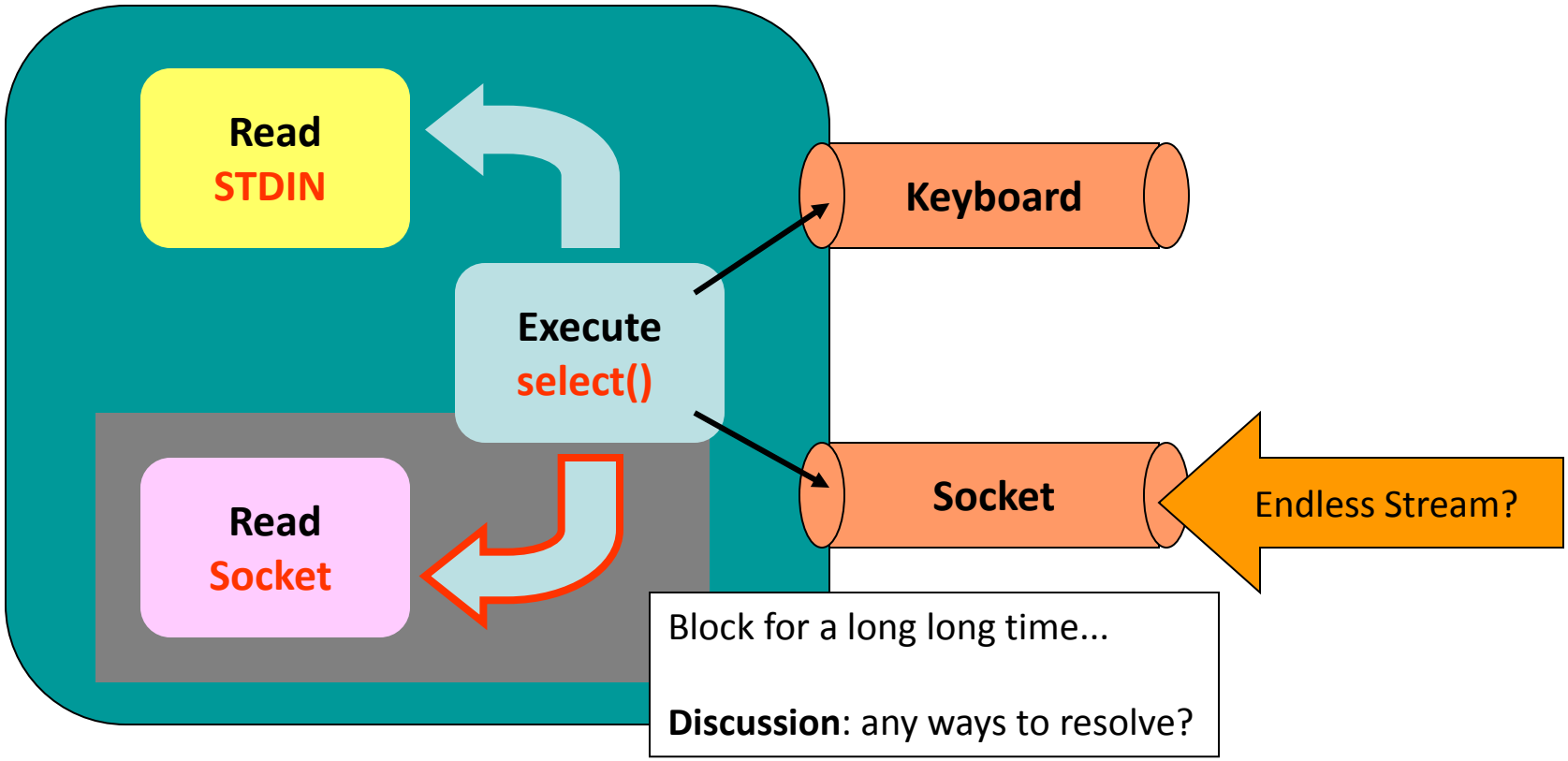


Change in Chat Room Design



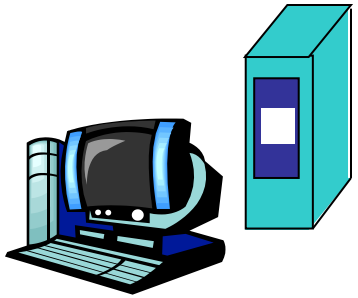
I/O multiplexing – summary

- It should be used with great care...



Application layer

- Protocol design:
  Learn from examples!
- Socket programming
- I/O multiplexing
- **Case study:**
  Apache server



Apache server

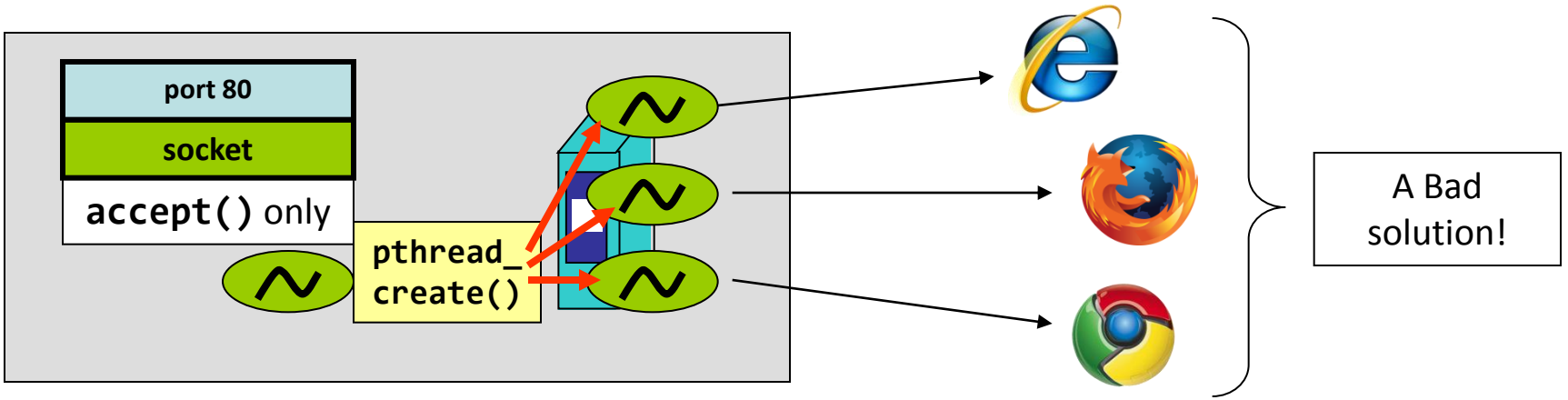
- This is the state-of-the-art, open source, HTTP server.

- Our focus: **performance!**
  - How can Apache support thousands of connections?

- Ingredients:
  - **`fork()`, `pthread_create()`, `pipe()`, `select()`.**

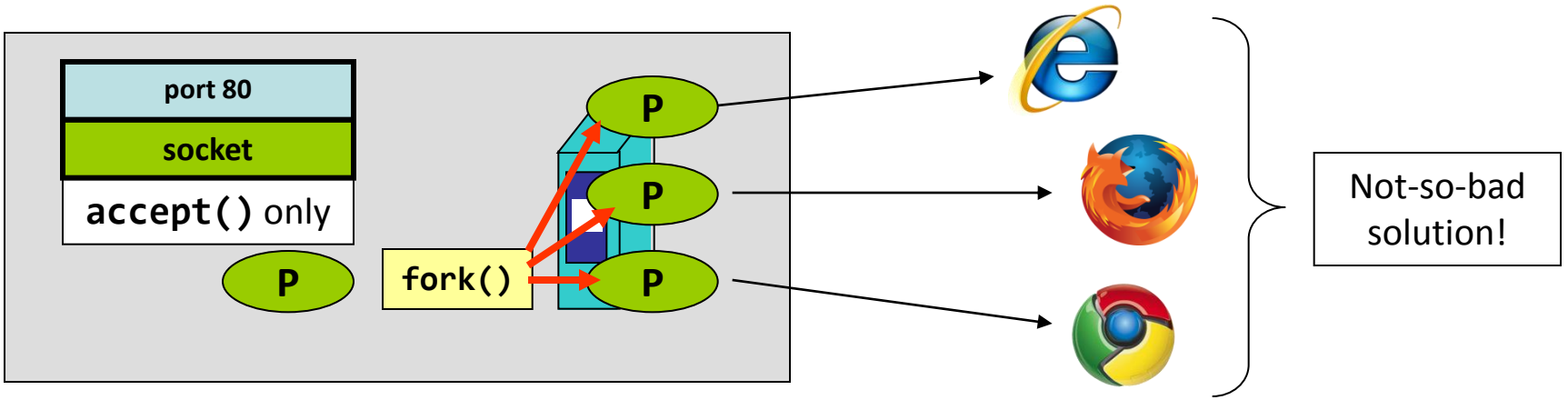
Scalability and Limitation

- Using threads alone?
  - Do you still remember what you've learnt in 3150?
- Addressing space limits the number of threads created.
- There is also a limit on the number of opened files (i.e., connections) per processes.



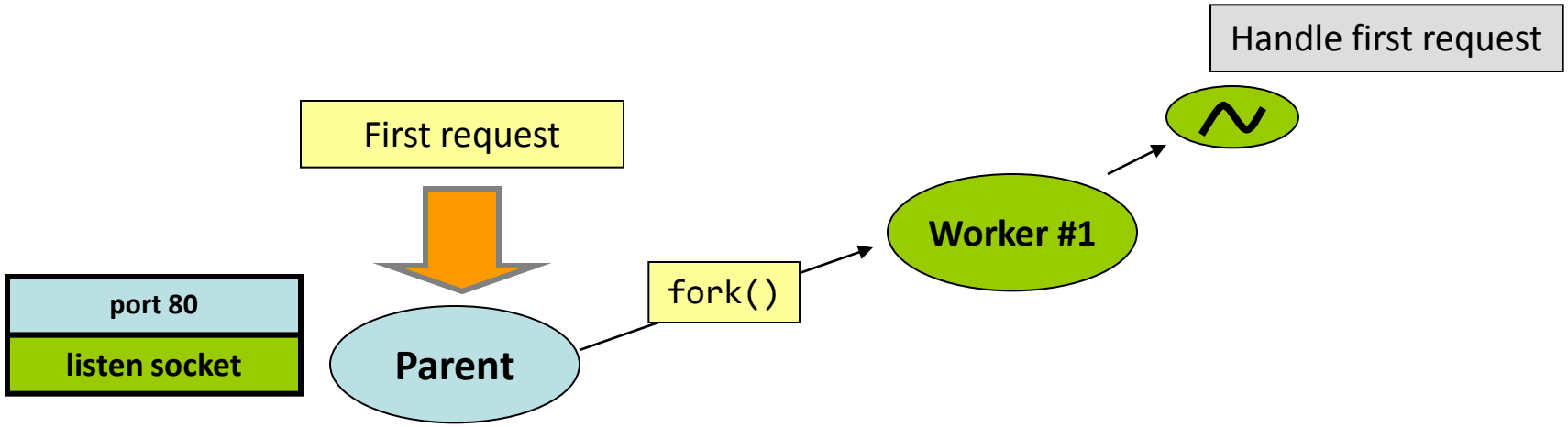
Scalability and Limitation

- Using processes alone?
  - Do you still remember what you've learnt in 3150?
- The number of processes available is limited.
- A heavy burden on memory consumption.



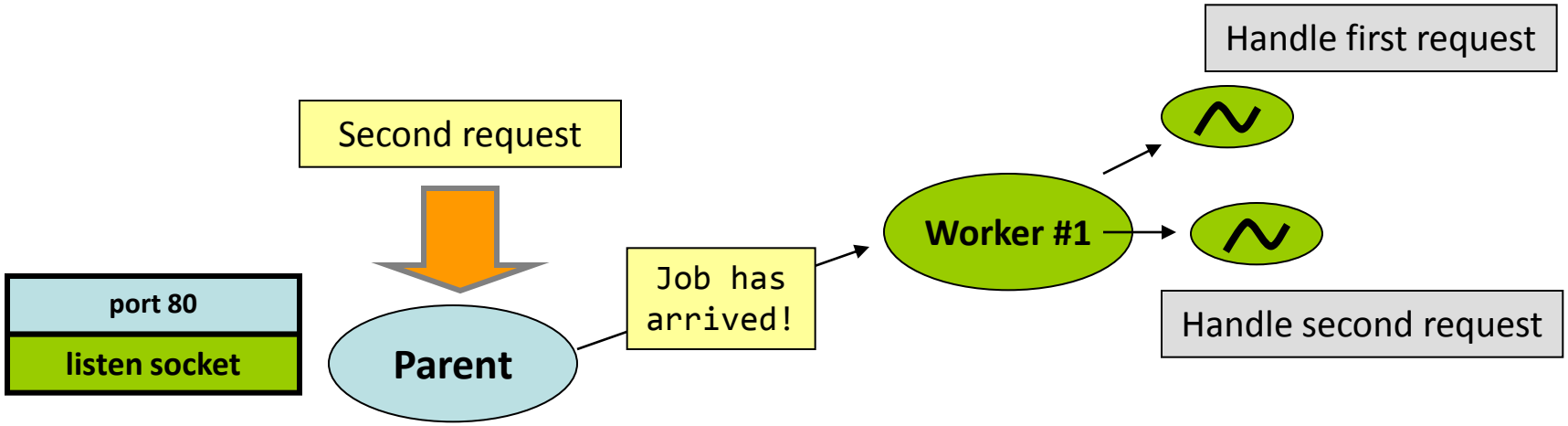
Apache's approach – Step (1)

- **`fork() + pthread_create()`**



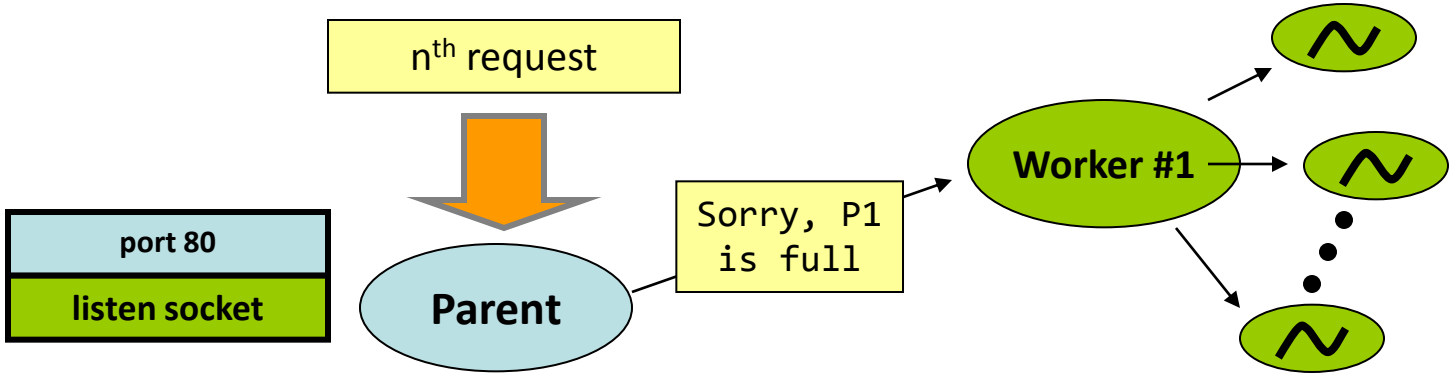
Apache's approach – Step (2)

- **`fork() + pthread_create()`**



Apache's approach – Step (3)?

- **`fork() + pthread_create()`**

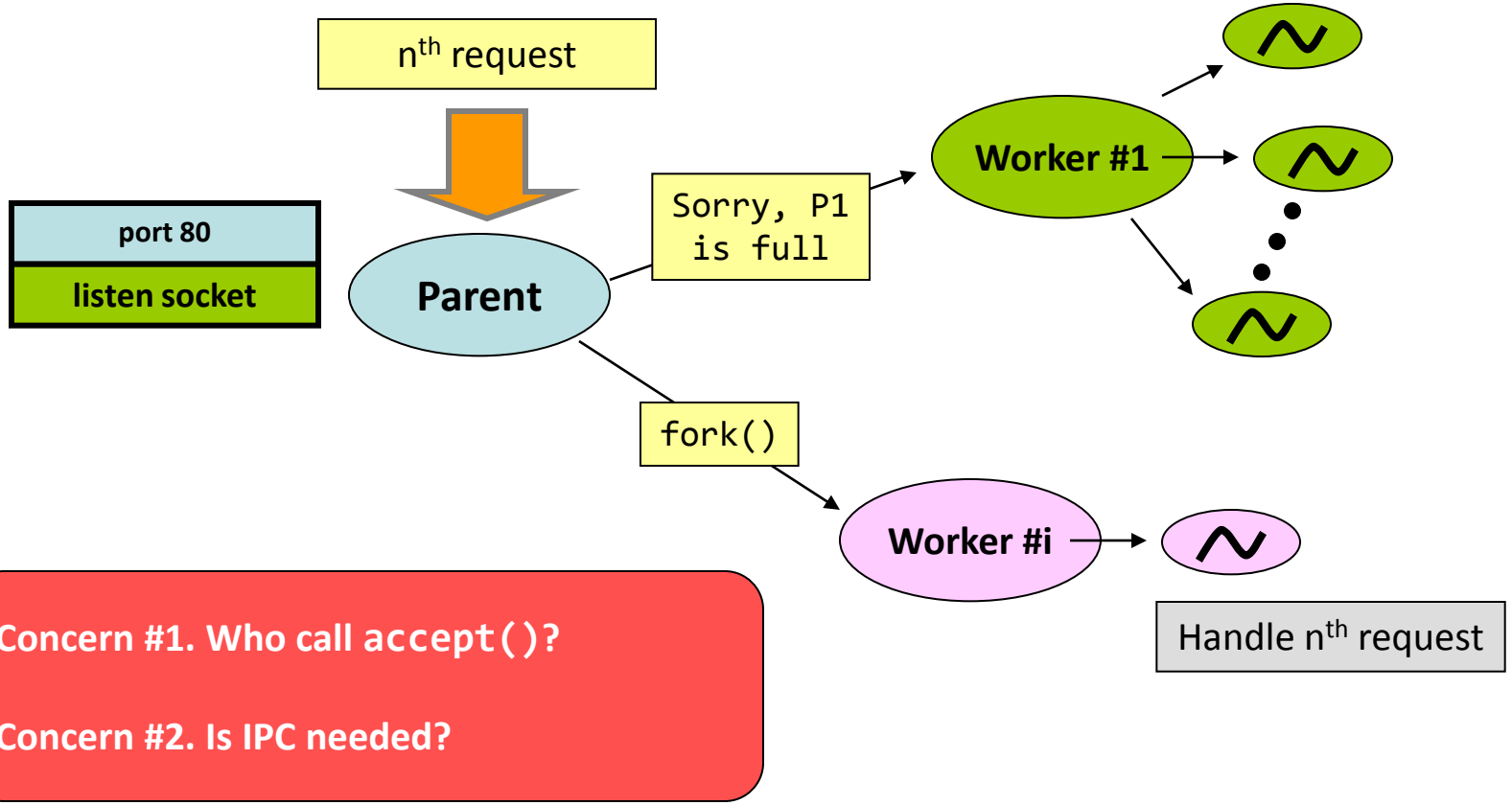


There are several reasons for a worker to reject a request:

(1) All FDs are used up!
(2) No more threads are available!

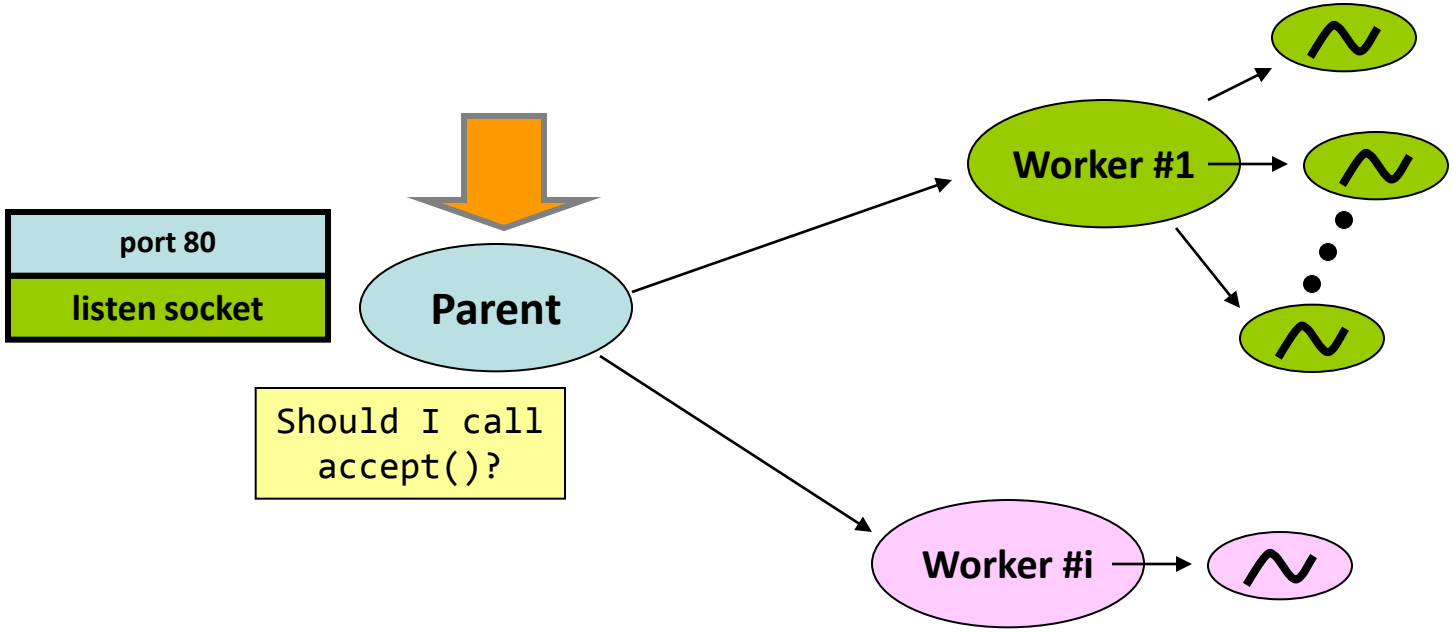
Apache's approach – Step (3)!

- **`fork() + pthread_create()`**



Apache's approach – concern #1

- Where is **`accept()`**?

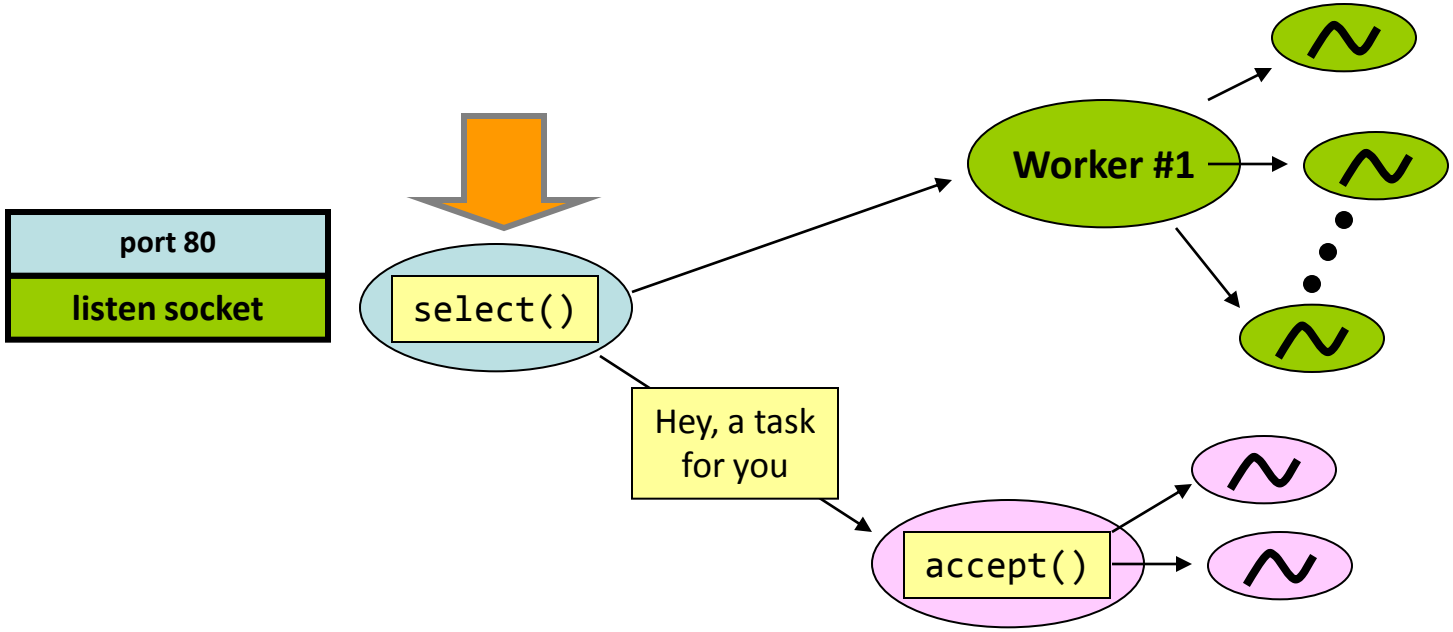


No! **`accept()`** produces a file descriptor for the caller process only.
P1 and P2 cannot use the new file descriptor!

Obviously, P1 and P2 call **`accept()`**! Either one calls or both call?

Apache's approach – concern #1

- Where is **`accept()`**? Get help from **`select()`**!



- Using **`select()`** is to detect if any request arrives at the listen socket.

- Since **`fork()`** is used, P1 and P2 both have the **listen socket**. So, both can call **`accept()`**!

Apache's approach – concern #2

- How about the IPC? Many ways! E.g.,
  - 2 uni-directional pipes;
  - 1 socket between two processes, etc.

- Basically, the IPC has to guarantee two things:
  - The parent process can ask a worker process to take the task, and
  - The worker process may reject an assigned task, e.g., max. # of threads reached!
    - Or, the management is in the parent's hand.
    - It's up to your implementation!

Do you want to know more?

- It provides such a service using a module called Multi-Processing Module, MPM.
  - Two modes: **prefork and worker**.

Prefork	Non-threading, processes only.
Worker	Processes with threads.

http://httpd.apache.org/docs/2.0/mod/prefork.html

http://httpd.apache.org/docs/2.0/mod/worker.html