

# CS 305 Lab Tutorial Lab 4

## Advanced HTTP & Socket Programming

Dept. Computer Science and Engineering  
Southern University of Science and Technology

Part A.

# Advanced HTTP

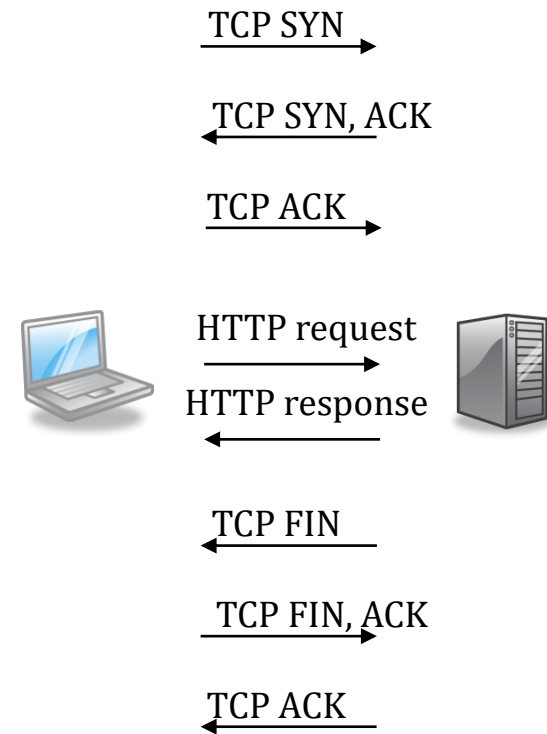
# Part A.1

## Connection and transfer encoding

- Connection management
  - Persistent connection, parallel connection
  - Connection: close
- Content-Length vs. Chunked transfer encoding
  - Reducing latency of response

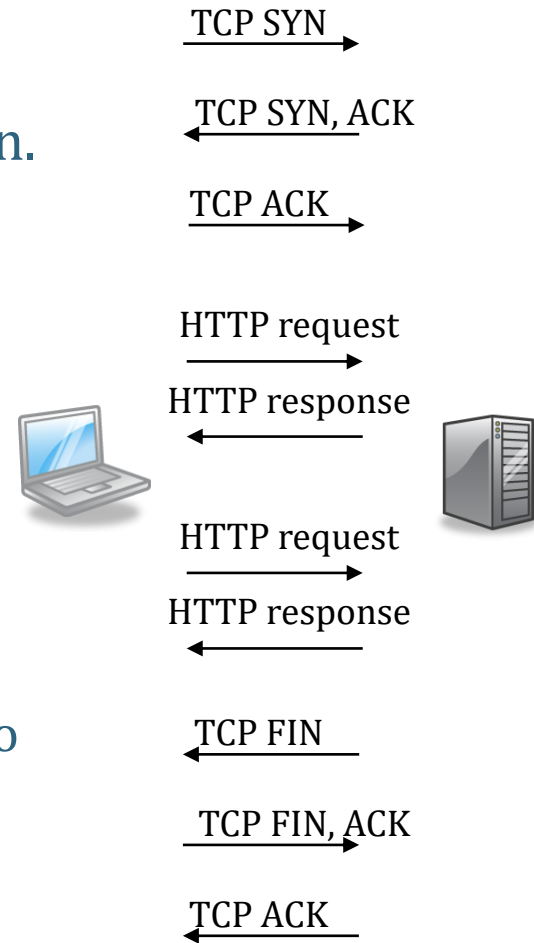
# Problem in HTTP/1.0 connection

- HTTP/1.0 uses a new connection for each HTTP transaction
- Making TCP connection is slow.
  - takes three packets to establish
  - takes three packets to close
- A web page typically contains many embedded images. HTTP/1.0 would make many TCP connections to load a web page.
  - Slow page loading



# Persistent and Parallel Connection

- **Persistent connection:** multiple requests and responses are sent through one TCP connection.
  - Default in HTTP/1.1
  - Browsers keep TCP connection after page load.  
Why?
- **Parallel connection:** A web browser opens several TCP connections to a web site and downloads components of a web page concurrently.
  - using tcp.srcport and tcp.dstport in wireshark to trace a http session

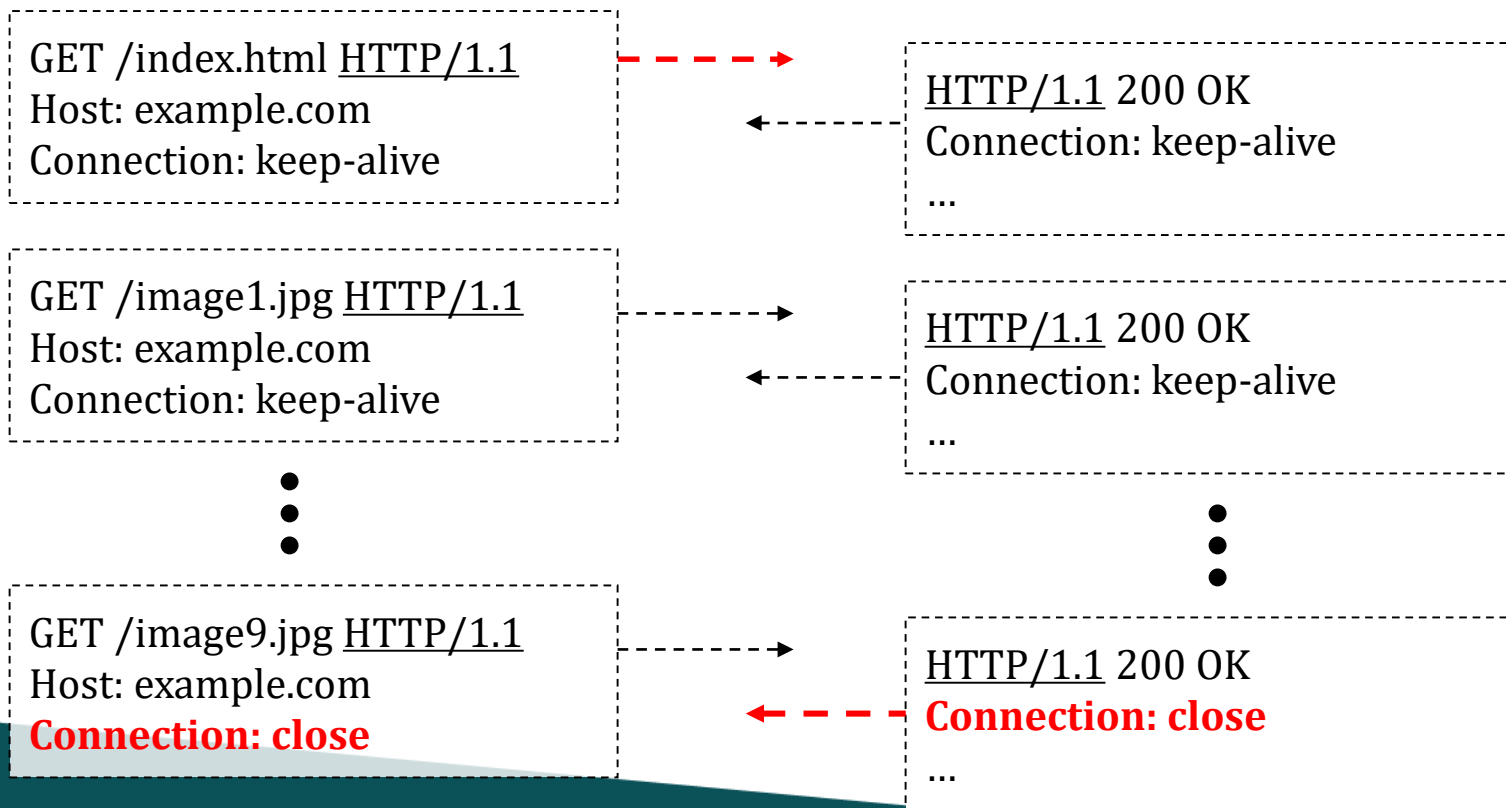


# Connection:

- The Connection: header indicates whether to keep or close the current connection.
- **Connection: keep-alive.** Default, may be omitted.
- **Connection: close**
  - In a request, the client asks the server to close the connection after sending the response
  - In a response, the server indicates that it will close the connection after sending this response

# Persistent connection

*The client creates a connection before sending the first request. Subsequent requests and responses are transferred in this TCP connection.*



# Multiple messages in a connection

- A browser receives multiple responses in a TCP connection.
- To break the byte stream into messages, it must know the end of each message.
- One solution is that the server declares **Content-Length** for each response.

HTTP/1.1 200 OK

Content-Type: ...

**Content-Length: 100**

... content of first resource ...

... 100 bytes ...

HTTP/1.1 200 OK

Content-Type: ...

**Content-Length: 200**

... content of second resource ...

... 200 bytes ...

HTTP/1.1 200 OK

**Connection: close**

Content-Type: ...

**Content-Length: 120**

... content of third resource ...

... 120 bytes ...



# Problems of Content-Length

- **Content-Length** header is sent before the message body.
- If a resource is generated dynamically by a server-side script (e.g. ASPX, PHP), the web server can determine Content-Length only after the script finishes execution.
  - The server has to buffer the whole response first, and can only start sending the response afterwards
- Efficiency problems:
  - Larger memory overhead
  - Slower response time

# Chunked Transfer-Encoding

- **Chunked transfer-encoding** enables a web server to start transmitting beginning parts of a response while it is still generating the rest.
  - Does not send **Content-Length**. Sends **Transfer-Encoding: chunked** instead.
  - A long response body is broken into several pieces called chunks.
  - Before sending each chunk, the server sends its length in hexadecimal.
  - After sending the last chunk, the server sends a 0.

## Response with Content-Length

HTTP/1.1 200 OK

Date: Wed, 19 Mar 2008 01:46:57 GMT

Content-Type: text/plain

**Content-Length: 42**

abcdefghijklmnopqrstuvwxyz1234567890abcdef

## Response with chunked body

HTTP/1.1 200 OK

Date: Wed, 19 Mar 2008 01:46:57 GMT

Content-Type: text/plain

**Transfer-Encoding: chunked**

**1a**

abcdefghijklmnopqrstuvwxyz

**10**

1234567890abcdef

**0**

*Length of a chunk in Hex*

*Length of a chunk in Hex*

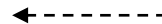
*Length of 0 means no more  
chunks*

# Partial Content

- How to retrieve a slice of resource?
  - Request:
    - Range: <unit>=<range-start>-<range-end>  
e.g. Range: bytes=200-1000, 2000-6576, 19000-
  - Response:
    - HTTP/1.1 206 Partial Content
    - Content-Range: <unit> <range-start>-<range-end>/<size>  
e.g. Content-Range: bytes 21010-47021/47022
- References:
  - <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Range>
  - <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Range>
  - <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/206>

# Example:

**GET** /video.mp4 HTTP/1.1  
Host: example.com  
Connection: keep-alive  
**Range:** bytes=1900-2900



HTTP/1.1 **206 Partial Content**  
Connection: keep-alive  
**Content-Range:** bytes 1900-2900/4702  
...

# Part A.2

## State, session and security

- Session management
  - HTTP is stateless
  - Cookies
  - As in common web framework (e.g. asp.net, php)
  - Session hijacking
- Encryption and SSL
  - Secured login vs. full-session HTTPS
  - Partially secure web page

# HTTP is stateless

- Statelessness means that every HTTP request happens in complete isolation.
- When the client makes an HTTP request, it includes all information necessary for the server to fulfill that request.
- The server never relies on information from previous requests. If that information was important, the client would have sent it again in this request.
  - A web server does not retain info between processing of requests from a user session
  - The client and server do not need to maintain a common state

(from O'Reilly RESTful web service)

# Example

- A browser keeps sending same (or similar) headers to a web server in a series of requests, e.g.
  - Host
  - User-agent
  - Content negotiation
- Each request contain the information from the full URL (absolute URL).

```
GET /wiki/Internet HTTP/1.1
Host: en.wikipedia.org
User-Agent: Mozilla/5.0 ... Firefox/3.5.3
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
```

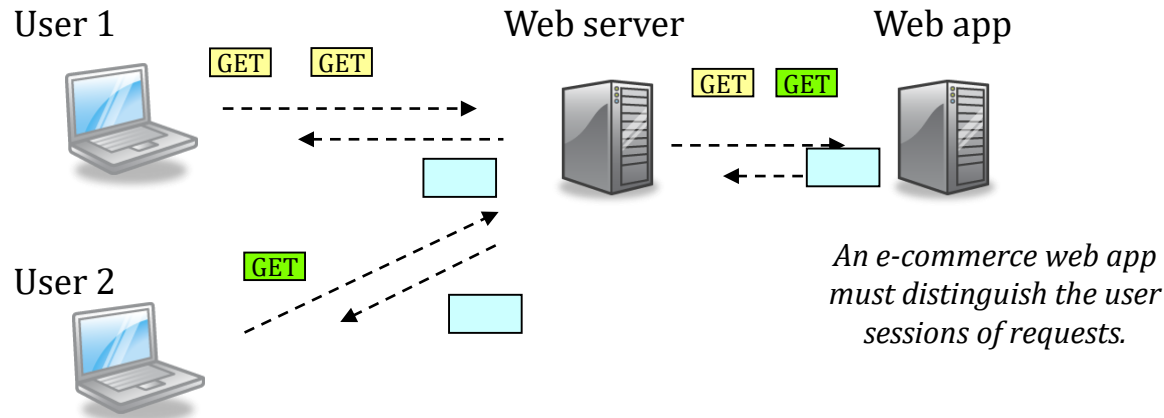


# HTTP over parallel connections

- A client does not need to use a single TCP connection for requests and responses to a server
  - The client can disconnect and reconnect anytime without breaking a session
  - The client can use several TCP connections to send requests and responses.
- Possibly no association between TCP connections and user sessions:
  - The web app cannot assume requests coming from one TCP connection belong to the same user session
  - Requests from one user session may be delivered in several TCP connection.

# Session Management in Web App

- Although HTTP is stateless, web app needs to maintain states in processing requests from a user session
  - e.g. Has a user logged in?  
Which requests come from the user?
- The client needs to attach session identifier in each request

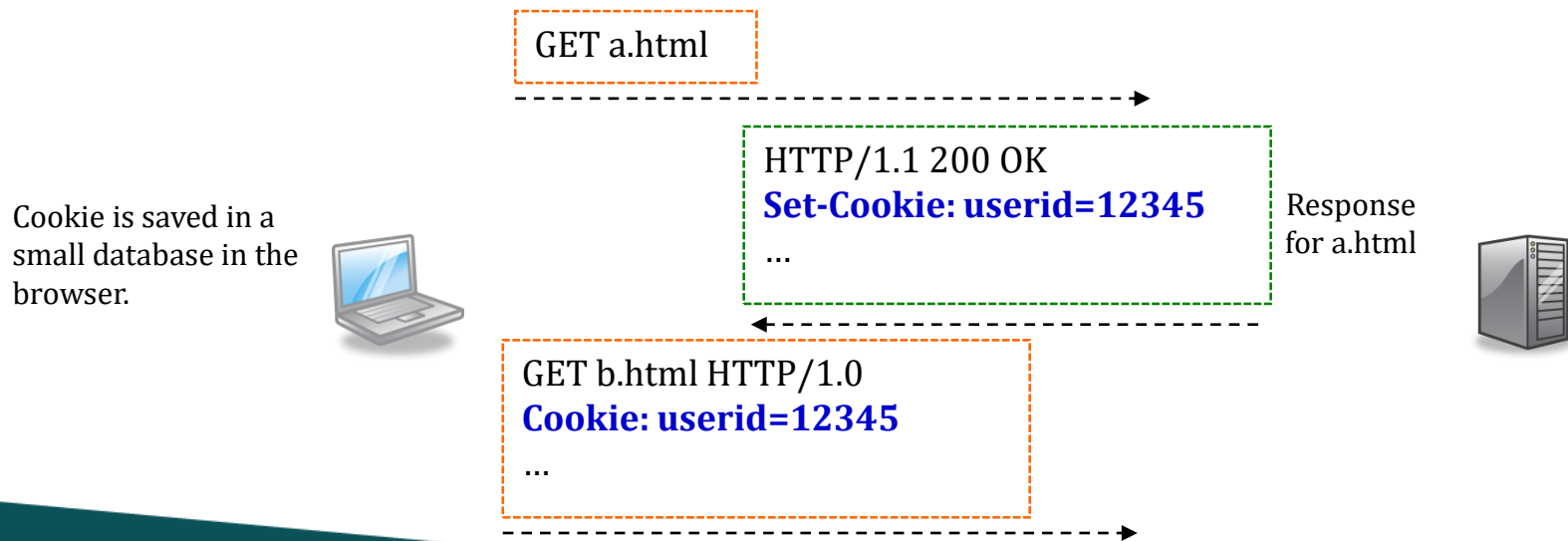


# Session Management

- A web app has to track a user's progress from one request to another
- Each request has to include some data to identify a user session
- Common approaches:
  - Cookies
  - Hidden form field (<input type="hidden">)
  - Query string

# HTTP Cookies

- Cookies are small pieces of data a web server asks a client to keep and send back in future requests.
  - Servers add header **Set-Cookie: name=value** in response
  - Clients add header **Cookie: name=value** in future requests



# Example: Typical use of Session id



POST login.php HTTP/1.1  
...  
user=philip&pw=123456



HTTP/1.1 200 OK  
**Set-Cookie: sessionid=3a4b5e**  
...

The user logs in by entering user name and password in an HTML form. The web app then creates a session in memory and returns a session ID.

GET home.php HTTP/1.1  
**Cookie: sessionid=3a4b5e**  
...

HTTP/1.1 200 OK  
...

GET checkmail.php HTTP/1.1  
**Cookie: sessionid=3a4b5e**  
...

HTTP/1.1 200 OK  
...

The browser attaches the cookie in all future requests to the web app. The web app can use the session id to look up application state (e.g. current user, 'session variables' )

# Cookie attributes

- A web server can restrict the scope of a cookie with attributes:
  - **expires** : date/time after which this cookie can be deleted
    - If not set, the cookie is deleted when user quits the browser
  - **path, domain** : the client should only include this cookie for requests in this domain and URL under this path
    - If not set, the default is the domain and path of the response
  - **secure**: a secure cookie may only be sent through SSL

HTTP/1.1 200 OK

**Set-Cookie: userid=12345; expires=Fri, 31-Dec-2010  
23:59:59 GMT; path=/; domain=.example.com**

...

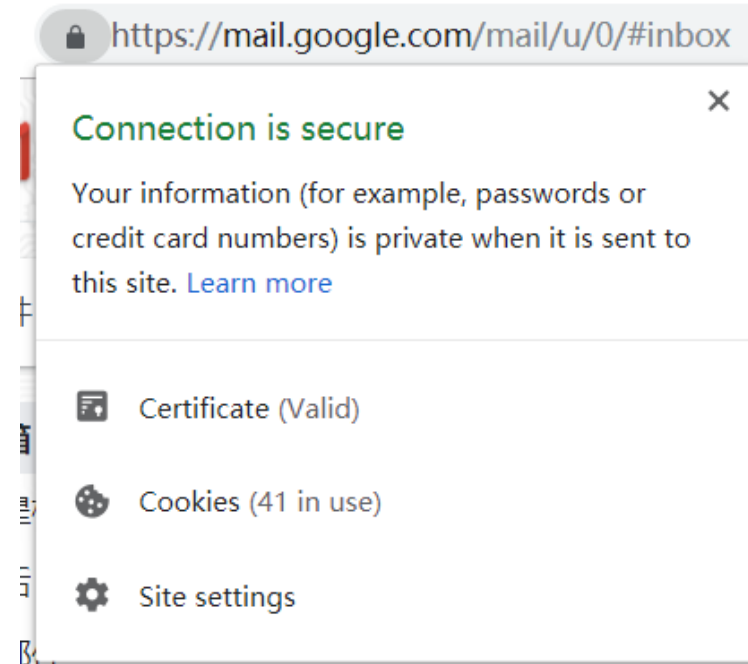
# Encryption for HTTP

- **TLS (Transport Layer Security, aka. SSL)** are cryptographic protocols that encrypt data transmitted over a TCP connection
  - Common versions: TLS1.2, TLS1.3
  - Can run different protocols over TLS, e.g. HTTP, SMTP, IMAP
- Two purposes:
  - Prevent eavesdropping and tampering
    - e.g. Only the client and server of an HTTP transaction can read the request/response
  - Verify the authenticity of the server
    - The server has a valid digital certificate issued by a certificate authority known by the browser

# HTTPS

<https://mail.google.com/mail/#inbox>

- Need to install/trust a digital certificate in the web server
- https – runs HTTP over a secured TCP connection
  - Use port 443
  - Usually TLS1.2
- A secure HTTP transaction
  - Attackers cannot read the request and response
  - Proxy (including cache servers) cannot read the messages either





# Partially secure web page

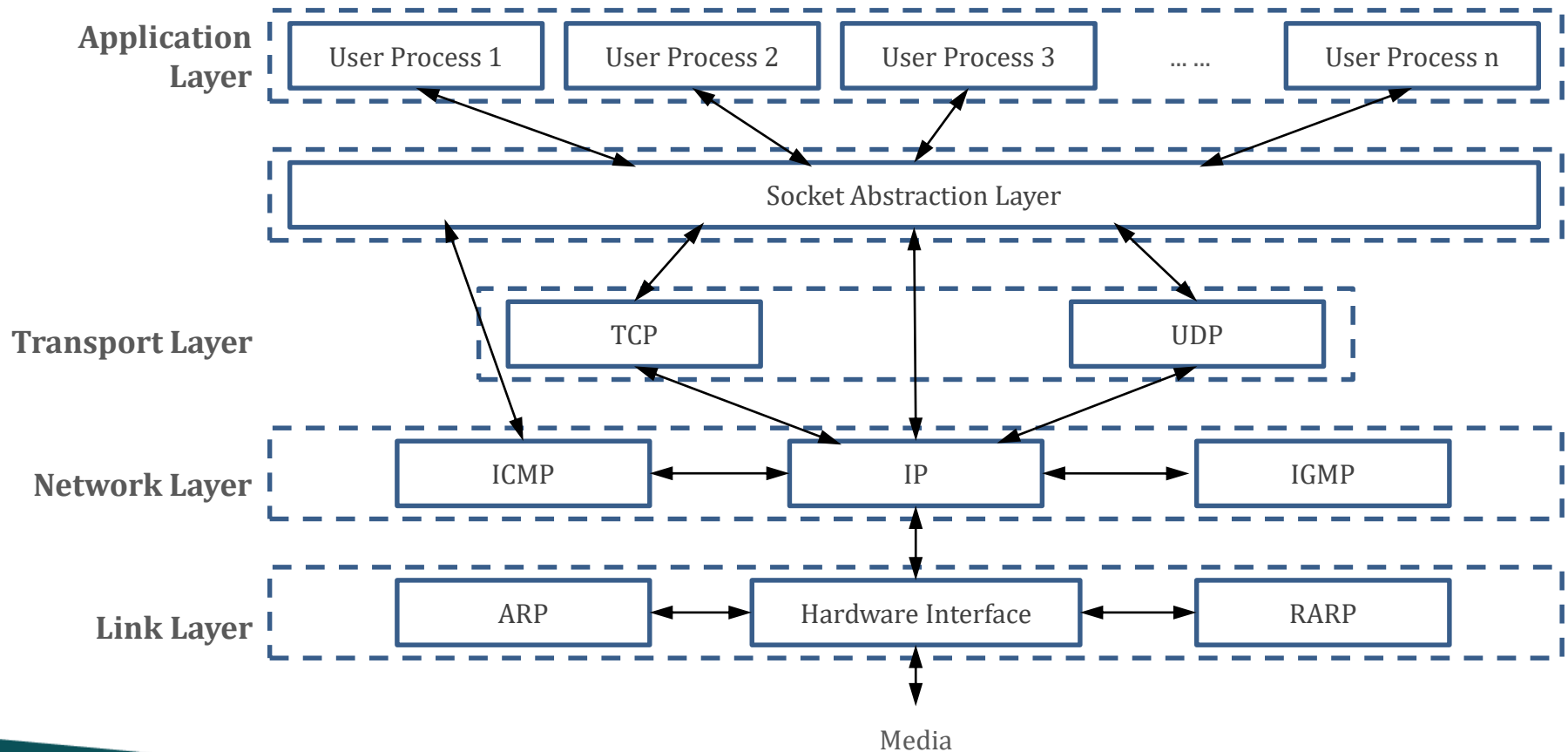
- A secure web page (https) that refers to unsecure resources (http)
  - e.g. the HTML page is using https, but the images inside are using http only
  - Unsecure resources may be modified and then added to the supposedly secure HTML page
    - Very serious if these are JavaScript files
  - HTTP requests to unsecure resources may contain cookies and eavesdropped by attackers
    - Problem solved by 'secure' attribute of cookies

✖ Mixed Content: The page at 'https://[redacted] index.html' was loaded over HTTPS, but requested an insecure resource 'http://player.[redacted]'. This request has been blocked; the content must be served over HTTPS.

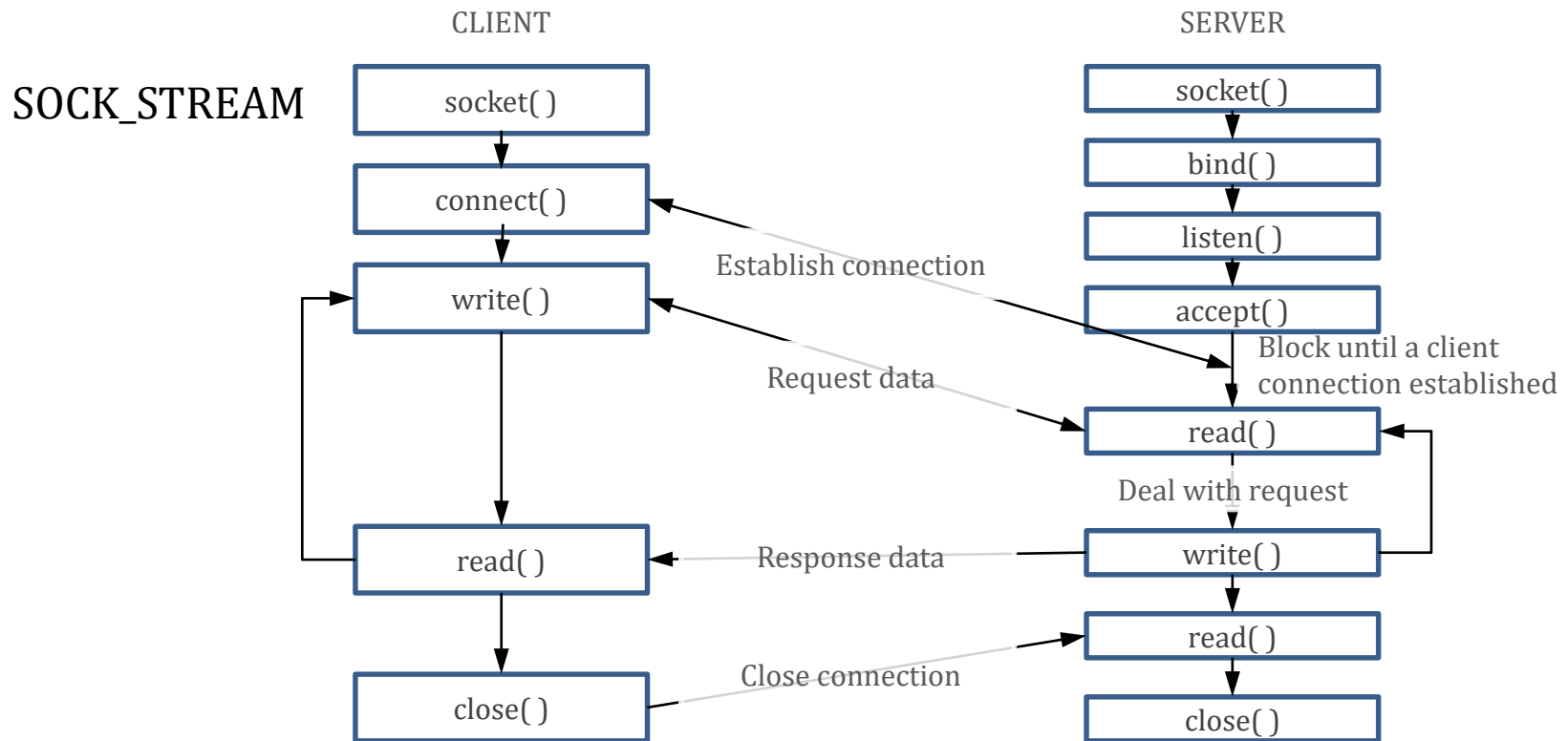
Part B.

# Socket Programming

# Socket (1)



# Socket (2)



# Socket Example 1: Echo Server (1)

```
import socket

def echo():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.bind(('127.0.0.1', 5555))
    sock.listen(10)
    sock.settimeout(0.5)

    while True:
        try:
            conn, address = sock.accept()
            while True:
                data = conn.recv(2048)
                if data and data != b'exit':
                    conn.send(data)
                    print(data)
                else:
                    conn.close()
                    break
        except socket.timeout:
            continue

if __name__ == "__main__":
    try:
        echo()
    except KeyboardInterrupt:
        pass
```

# Socket Example 1: Echo Server (2)

- 1. Run the server echo on Linux(or macOS) and Windows separately, is there any difference?
- 2. How to exit the loop?
- 3. Is there anyway to improve the server to make it work the same on different OS?

# Socket Example 1: Echo Server (3)



```
/c/Users/light/PycharmProjects/CS305-2

light@DESKTOP-K4SPJVV MINGW64 /c/Users/light/PycharmProjects/CS305-2
$ python echo.py
b'test\r\n'
b'CS305 is Awsome.\r\n'

/

light@DESKTOP-K4SPJVV MINGW64 /
$ telnet 127.0.0.1 5555
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
test
test
CS305 is Awsome.
CS305 is Awsome.
exit
Connection closed by foreign host.

light@DESKTOP-K4SPJVV MINGW64 /
$
```

# Example 2: Mimic a Simple Web Server (1)

```
def web():
```

```
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
    sock.bind(('127.0.0.1', 8080))
```

```
    sock.listen(10)
```

```
    while True:
```

```
        conn, address = sock.accept()
```

```
        data = conn.recv(2048).decode().split('\r\n')
```

```
        print(data[0].split(' '))
```

```
        res = err404
```

```
        if data[0].split(' ')[1] == '/':
```

```
            res = hello
```

```
        for line in res :
```

```
            conn.send(line)
```

```
        conn.close()
```

```
if __name__ == "__main__":
```

```
    try:
```

```
        web()
```

```
    except KeyboardInterrupt:
```

```
        pass
```

```
import socket
```

```
hello = [b'HTTP/1.0 200 OK\r\n',
```

```
        b'Connection: close'
```

```
        b'Content-Type:text/html; charset=utf-8\r\n',
```

```
        b'\r\n',
```

```
        b'<html><body>Hello World!<body></html>\r\n',
```

```
        b'\r\n']
```

```
err404 = [b'HTTP/1.0 404 Not Found\r\n',
```

```
        b'Connection: close'
```

```
        b'Content-Type:text/html; charset=utf-8\r\n',
```

```
        b'\r\n',
```

```
        b'<html><body>404 Not Found<body></html>\r\n',
```

```
        b'\r\n']
```



# Example 2: Mimic a Simple Web Server (2)



The image shows two terminal windows. The top window is titled `/c/Users/light/PycharmProjects/CS305-2` and shows the execution of a Python script `web_hello.py`. The script receives two GET requests: one for `/` and one for `/not-exist`. The bottom window is titled `/` and shows the output of `curl` commands. The first `curl` command to `127.0.0.1:8080` returns `<html><body>Hello World!</body></html>`. The second `curl` command to `127.0.0.1:8080/not-exist` returns `<html><body>404 Not Found</body></html>`.

```
/c/Users/light/PycharmProjects/CS305-2
light@DESKTOP-K4SPJVV MINGW64 /c/Users/light/PycharmProjects/CS305-2
$ python web_hello.py
['GET', '/', 'HTTP/1.1']
['GET', '/not-exist', 'HTTP/1.1']

/
light@DESKTOP-K4SPJVV MINGW64 /
$ curl 127.0.0.1:8080
<html><body>Hello World!</body></html>

light@DESKTOP-K4SPJVV MINGW64 /
$ curl 127.0.0.1:8080/not-exist
<html><body>404 Not Found</body></html>
```

# Example 3: Echo Server Multithreading (1)

```
import socket, threading

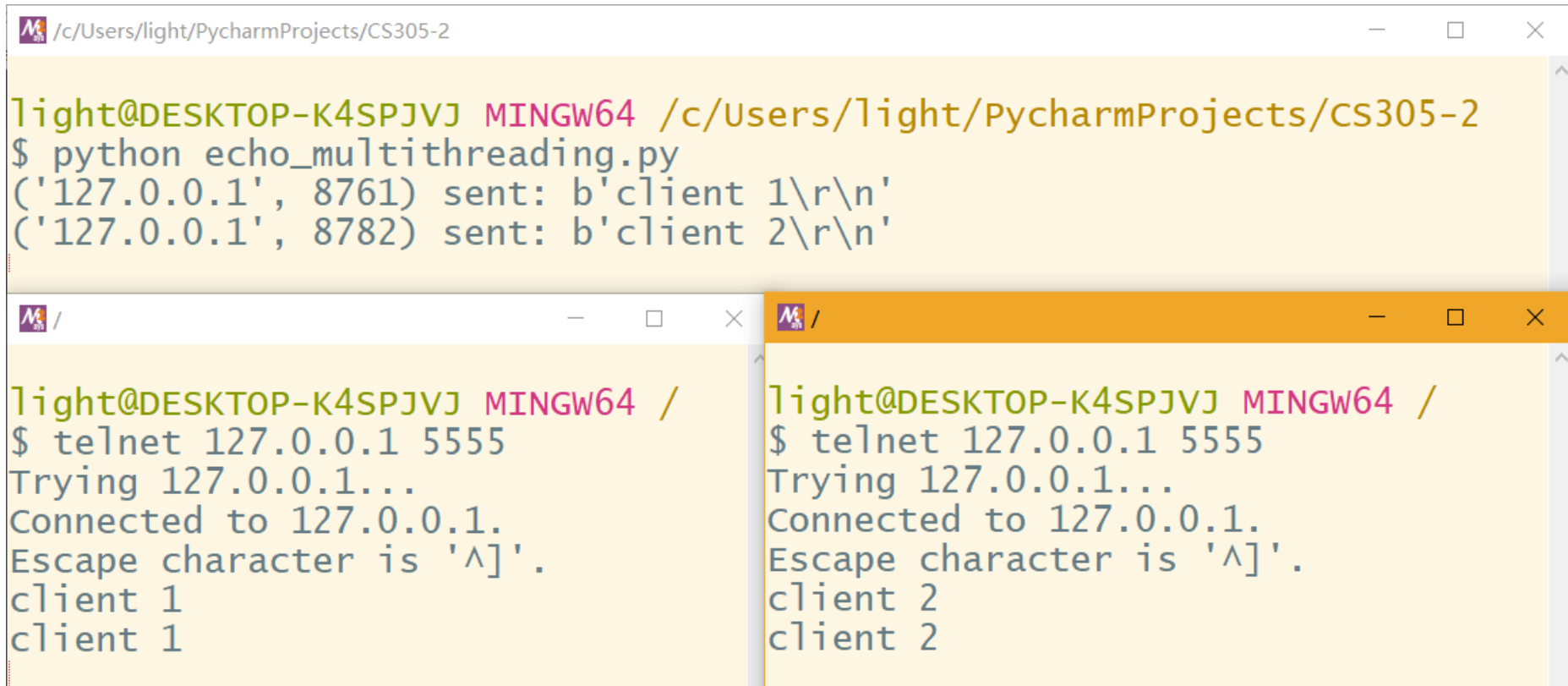
class Echo(threading.Thread):
    def __init__(self, conn, address):
        threading.Thread.__init__(self)
        self.conn = conn
        self.address = address

    def run(self):
        while True:
            data = self.conn.recv(2048)
            if data and data != b'exit\r\n':
                self.conn.send(data)
                print('{} sent: {}'.format(self.address, data))
            else:
                self.conn.close()
                return

def echo():
    sock = socket.socket(socket.AF_INET,
                          socket.SOCK_STREAM)
    sock.bind(('127.0.0.1', 5555))
    sock.listen(10)
    while True:
        conn, address = sock.accept()
        Echo(conn, address).start()

if __name__ == "__main__":
    try:
        echo()
    except KeyboardInterrupt:
        pass
```

# Example 3: Echo Server Multithreading (2)



The image shows three terminal windows from a Windows command prompt. The top window shows the execution of a Python script named `echo_multithreading.py`. The script sends two messages to `127.0.0.1` on ports `8761` and `8782`, both containing the text `client 1` and `client 2` respectively. The bottom-left window shows a telnet client connecting to `127.0.0.1` on port `5555`, receiving the message `client 1`. The bottom-right window shows another telnet client connecting to `127.0.0.1` on port `5555`, receiving the message `client 2`.

```
light@DESKTOP-K4SPJVV MINGW64 /c/Users/light/PycharmProjects/CS305-2
$ python echo_multithreading.py
('127.0.0.1', 8761) sent: b'client 1\r\n'
('127.0.0.1', 8782) sent: b'client 2\r\n'
```

```
light@DESKTOP-K4SPJVV MINGW64 /
$ telnet 127.0.0.1 5555
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
client 1
client 1
```

```
light@DESKTOP-K4SPJVV MINGW64 /
$ telnet 127.0.0.1 5555
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
client 2
client 2
```

# Example 4: asyncio Web Hello

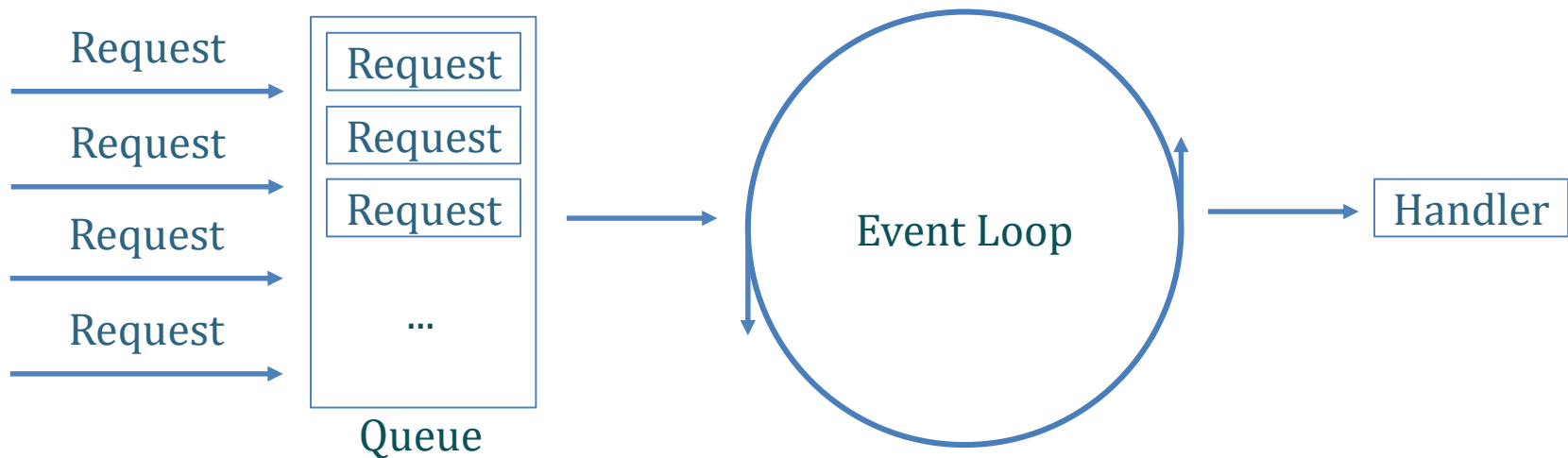
```
import asyncio
```

```
async def dispatch(reader, writer):  
    while True:  
        data = await reader.readline()  
        message = data.decode().split(' '  
        print(data)  
        if data == b'\r\n':  
            break  
        writer.writelines([  
            b'HTTP/1.0 200 OK\r\n',  
            b'Content-Type:text/html; charset=utf-8\r\n',  
            b'Connection: close\r\n',  
            b'\r\n',  
            b'<html><body>Hello World!<body></html>\r\n',  
            b'\r\n'  
        ])  
        await writer.drain()  
        writer.close()
```

```
if __name__ == '__main__':  
    loop = asyncio.get_event_loop()  
    coro = asyncio.start_server(dispatch, '127.0.0.1', 8080, loop=loop)  
    server = loop.run_until_complete(coro)  
  
    # Serve requests until Ctrl+C is pressed  
    print('Serving on {}'.format(server.sockets[0].getsockname()))  
    try:  
        loop.run_forever()  
    except KeyboardInterrupt:  
        pass  
  
    # Close the server  
    server.close()  
    loop.run_until_complete(server.wait_closed())  
    loop.close()
```

# asyncio

- Handle requests with a single-threaded concurrently.



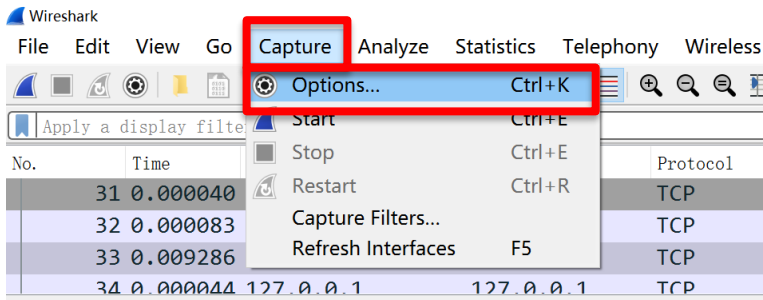
# Practice 1

- We have illustrated how to mimic a simple web server and how to handle multiple requests concurrently in examples. Please improve example 2 to handle more circumstances that tagged different status codes (at least 6 circumstances showing in next page).
- Use Wireshark to capture and analyze the packets, list the source IP, source port number, destination IP, destination port number and response's status code of each session.

# Common status codes

Status code	Meaning
200 OK	The server successfully carried out the action that the client requested. For GET request, the response body contains a representation of the requested resource.
204 No content	The server successfully carried out the action, but declined to return any representation. In Ajax, this usually means the browser should not refresh the user interface.
400 Bad request	Generic client-side error. Probably a request format error.
404 Not found	The server cannot find the resource at the requested URL.
500 Internal server error	Generic server-side error. Probably a server-side program run-time error.
503 Service unavailable	The web server is not available, probably because of overloading or maintenance.

# Tips: capture loopback traffic



No.	Time	Source	Destination	Protocol
16	0.000241	127.0.0.1	127.0.0.1	TCP
17	0.000022	127.0.0.1	127.0.0.1	TCP
18	0.000023	127.0.0.1	127.0.0.1	TCP
19	0.000012	127.0.0.1	127.0.0.1	TCP

