CS 305 Lab Tutorial Lab 4
Advanced HTTP & Socket Programming

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

SUSTech
Southern University of Science and Technology

thanks to HHQ.Zhang, Q.Wang

Part A.

Advanced HTTP

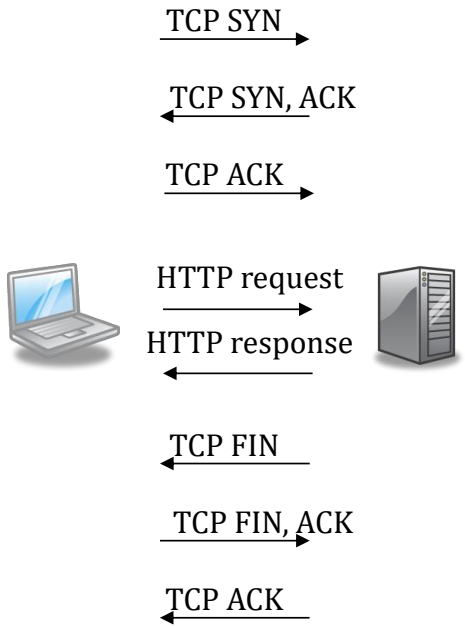
SUSTech
Southern University
of Science and Technology

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

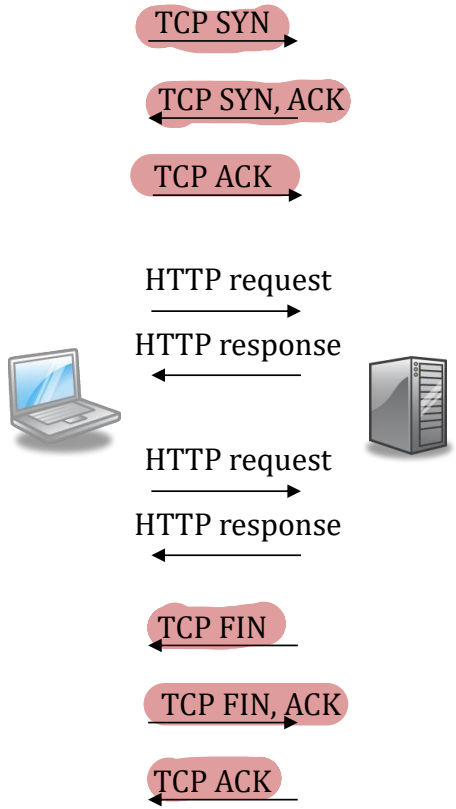
Problem in HTTP/1.0 connection

- 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

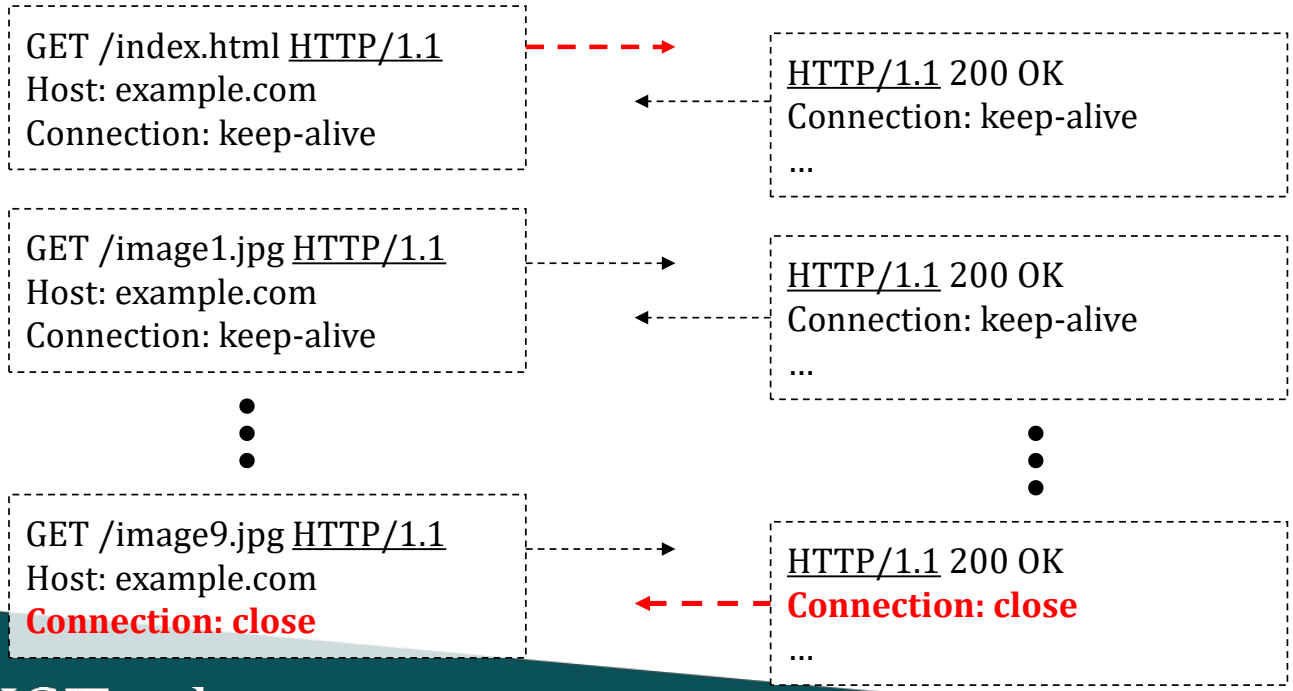


Connection:

- 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.

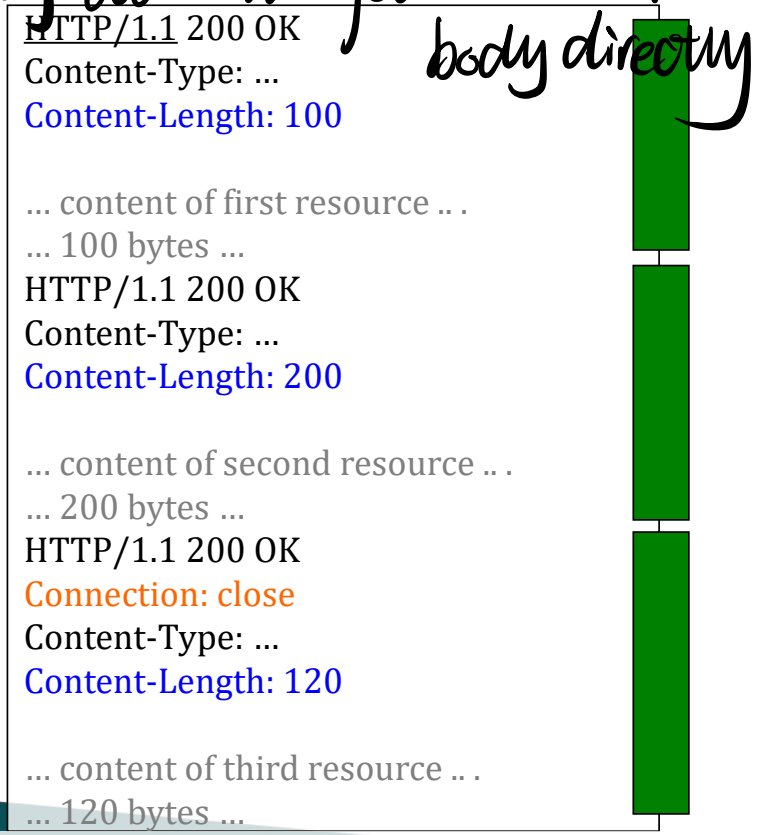


The server closes the connection after the last response.

Multiple messages in a connection

? response 消息 look like? why don't we get content from body directly

- 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.



Three responses transferred in a TCP stream

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

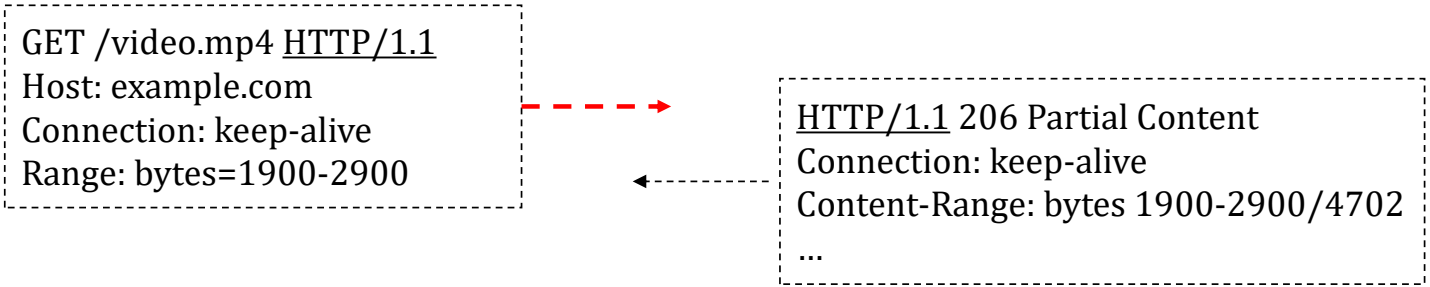
Length of a chunk in Hex

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).

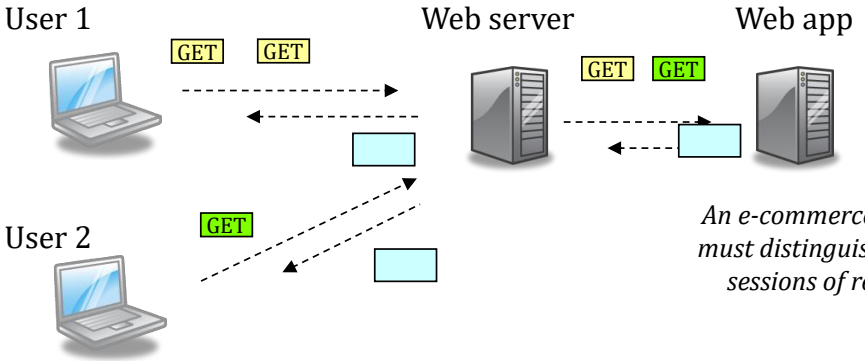
```
GET /wiki/Internet HTTP/1.1
Host: en.wikipedia.org
User-Agent: Mozilla/5.0 ... Firefox/3.5.3
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
```

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



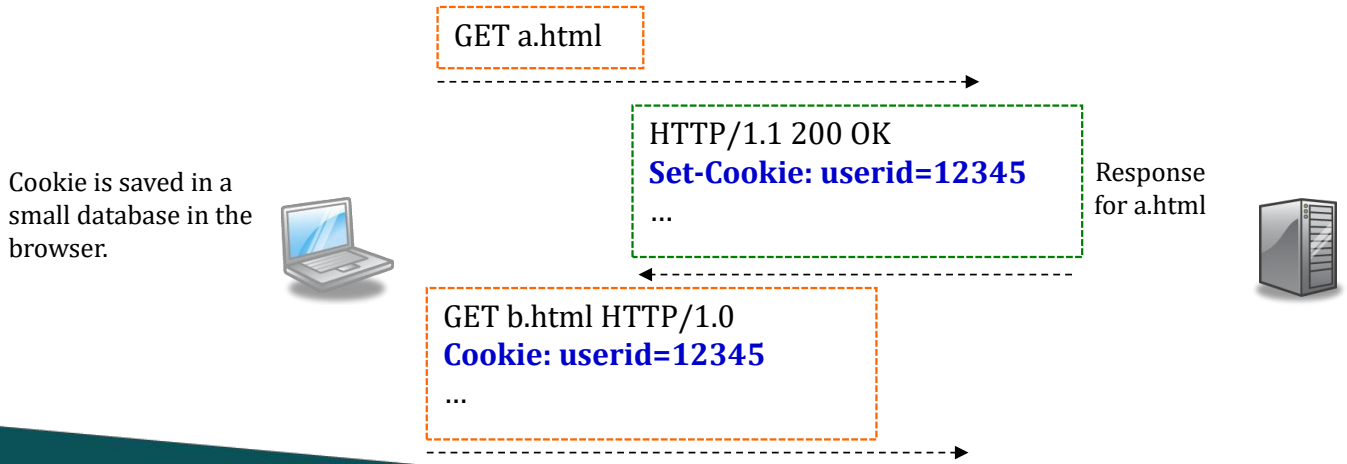
An e-commerce web app must distinguish the user sessions of requests.

Session Management

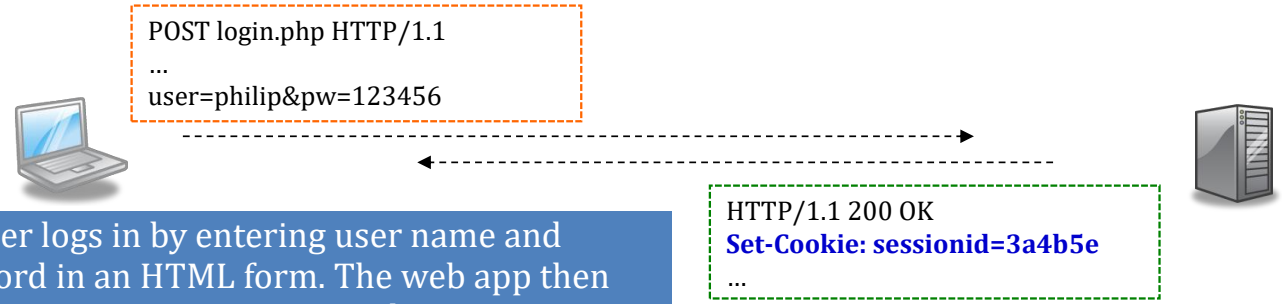
- A web app has to track a user's progress from one request to another
- Each request has to include some data to identify a user session
- Common approaches:
  - Cookies
  - Hidden form field (<input type="hidden">)
  - Query string

HTTP Cookies

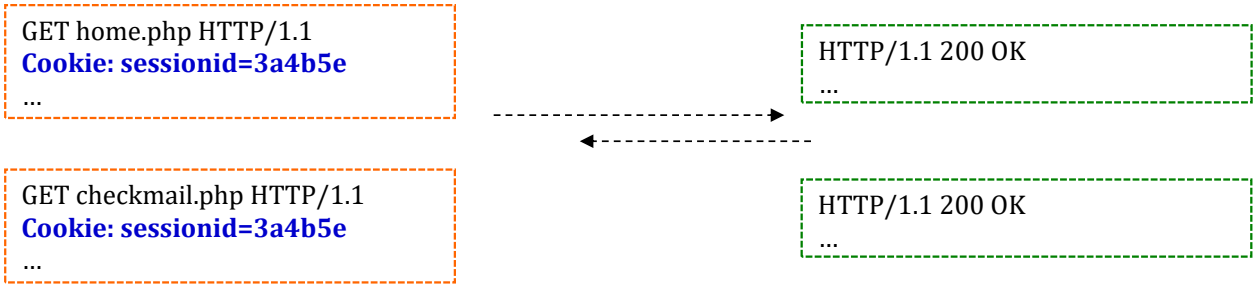
- Cookies are small pieces of data a web server asks a client to keep and send back in future requests.
  - Servers add header Set-Cookie: name=value in response
  - Clients add header Cookie: name=value in future requests



Example: Typical use of Session id



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.



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
...
```

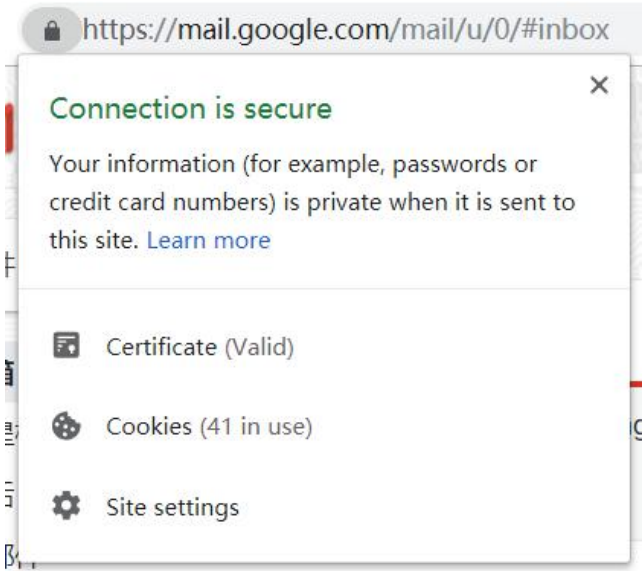
Encryption for HTTP

- **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

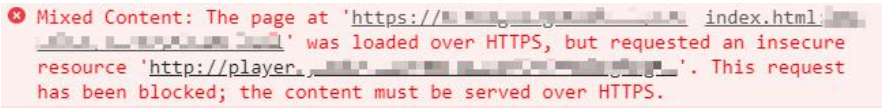
- Need to install/trust a digital certificate in the web server
- https – runs HTTP over a secured TCP connection
  - Use port 443
  - Usually TLS1.2
- A secure HTTP transaction
  - Attackers cannot read the request and response
  - Proxy (including cache servers) cannot read the messages either



cache cannot work anymore?

Partially secure web page

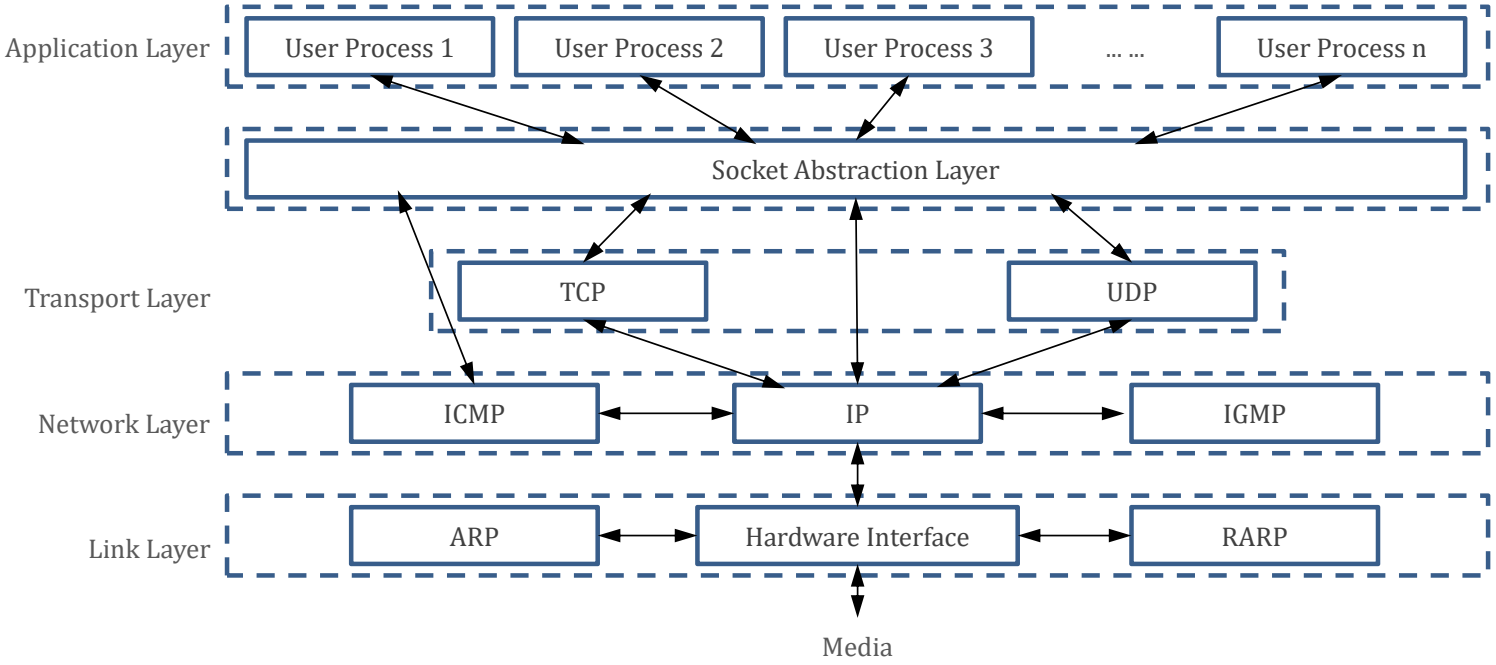
- A secure web page (https) that refers to unsecure resources (http)
  - e.g. the HTML page is using https, but the images inside are using http only
  - Unsecure resources may be modified and then added to the supposedly secure HTML page
    - Very serious if these are JavaScript files
  - HTTP requests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



Part B.

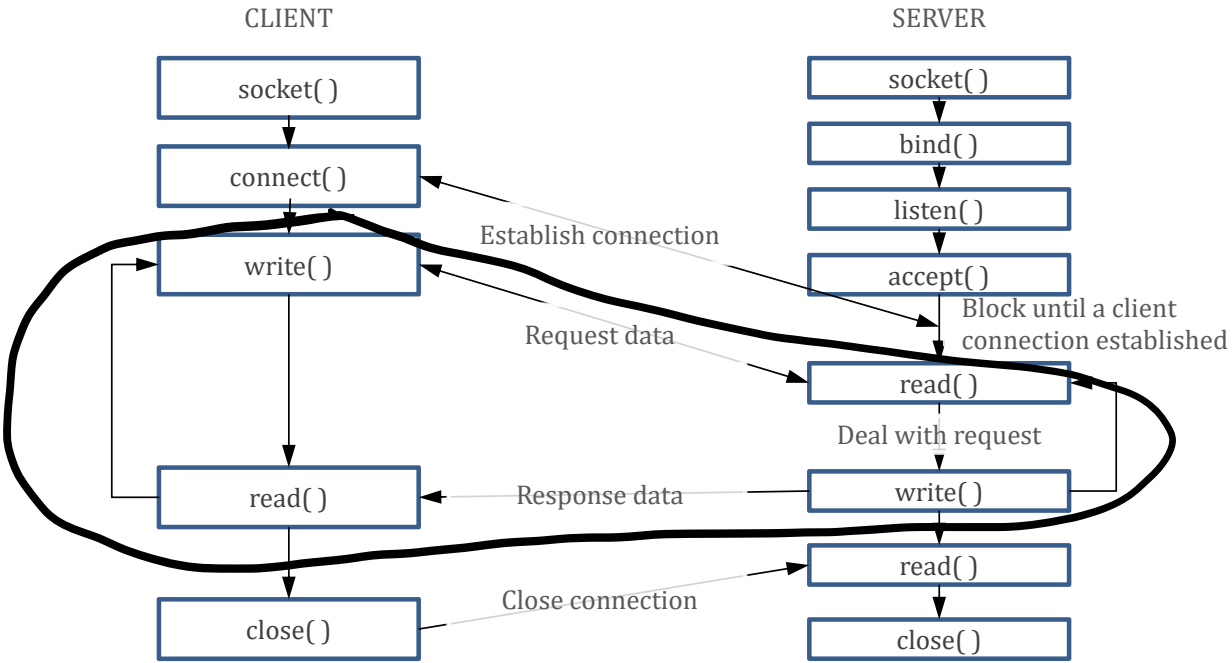
Socket Programming

Socket(1)


Application Layer
User Process 1
User Process 2
User Process 3
... ...
User Process n
Socket Abstraction Layer
Transport Layer
TCP
UDP
Network Layer
ICMP
IP
IGMP
Link Layer
ARP
Hardware Interface
RARP
Media

Socket(2)

SOCK_
STREAM



Example: Web Hello World Server

Server

```
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: Web Hello World Server

Example: Echo Server Multithreading

```
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
```

线程启动

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

```
light@DESKTOP-K4SPJVJ 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'
```

/

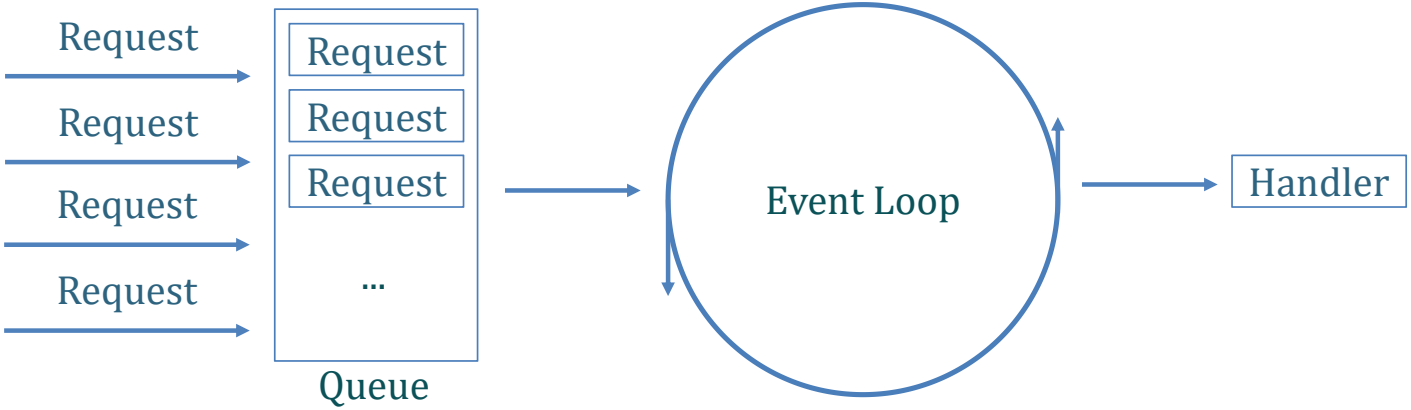
```
light@DESKTOP-K4SPJVJ MINGW64 /
$ telnet 127.0.0.1 5555
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
client 1
client 1
```

/

```
light@DESKTOP-K4SPJVJ 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
```

asyncio

- Handle requests with a single-threaded concurrently.



Event loop: 把一些异步函数注册到事件循环上，loop会循环执行这些函数，当执行到某个函数时，如果正在等待I/O返回，那么loop会暂停它去执行别的下次循环到这个函数会继续执行

Example: 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()
```

Example: 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()
```