SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Lec3 – The Application Layer

50.012 Networks

Jit Biswas

Cohort 1: TT7&8 (1.409-10)

Cohort 2: TT24&25 (2.503-4)

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Lec3 – The Application Layer

50.012 Networks

Jit Biswas

Cohort 1: TT7&8 (1.409-10)

Cohort 2: TT24&25 (2.503-4)

Introduction

- Todays lecture: The application layer
  - Application layer addressing/ URLs
  - Sockets
  - FTP as an example application layer protocol
  - HTTP as an important application layer protocol
  - P2P Applications

The Application Layer

Overview of the application layer

- Contains protocols that are specific to applications
- These protocols can rely on lower layers to provide routing, error correction, etc.
- Examples application layer protocols:
  - FTP and SMTP
  - HTTP traffic with HTML and image content
  - FTP traffic with binary data
  - NTP traffic to synchronize the time
  - Skype traffic for audio/video chat
- Addressing on application layer: URLs/URIs
- Socket: interface between the application and transport layer!

Application layer – some other protocols

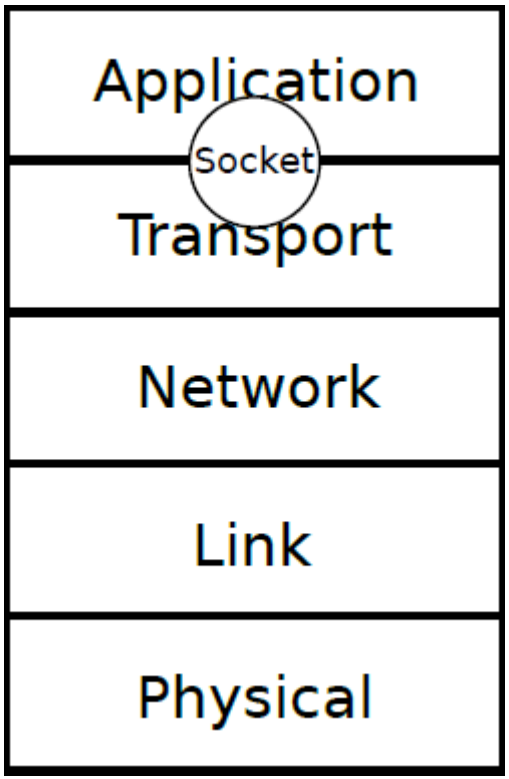
- Border Gateway Protocol (BGP)
- Domain Name Service (DNS)
- Dynamic Host Control Protocol (DHCP)
- Telnet (Teletype Network)
- File Transfer Protocol
- Internet Mail Access Protocol (IMAP)
- Simple Mail Transfer Protocol (SMTP)
- Secure Shell (SSH)
- Transport Layer Security (TLS)

Application layer addressing

- Servers can be addressed using domain names
- How can we address resources on a certain system?
- Must be globally unique name, may come with protocol identifier (allows client to use correct application to handle URL content)
- This is what uniform resource locators (URLs) provide:
  - protocol://domain.name/resource/path
  - `https://edimension.sutd.edu.sg/webapps/login/`
  - Globally unique, resolves to different IPs from in/outside SUTD
- Custom protocol IDs are possible, if the client can handle them
  - example: Apple's itms:// protocolID for the iTunes store

Sockets

- From CS perspective: object that you can write, read, or listen on
- Create the socket with lower layer information (e.g., transport protocol, port number, IP address)
- Receiver will run application with similar socket
- Data pushed into sender socket will magically appear at receiver socket



Example app: TCP server

Python TCPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while 1:
     connectionSocket, addr = serverSocket.accept()

     sentence = connectionSocket.recv(1024)
     capitalizedSentence = sentence.upper()
     connectionSocket.send(capitalizedSentence)
     connectionSocket.close()
```

Example app: TCP client

Python TCPClient

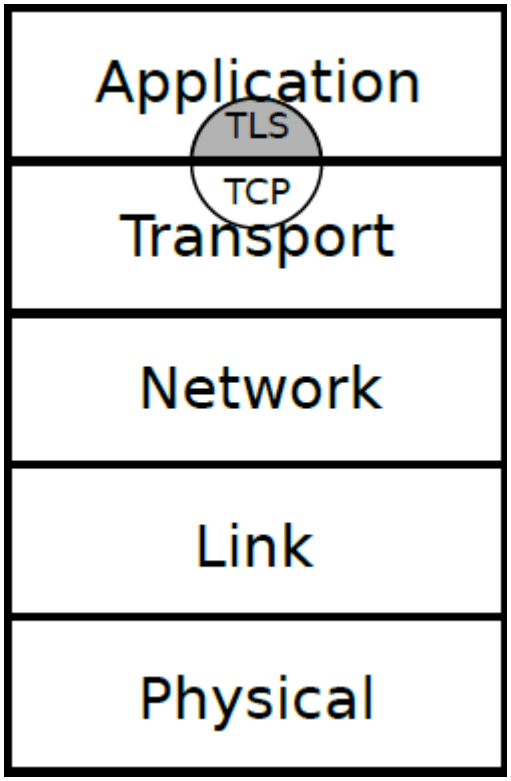
```
from socket import *
serverName = 'localhost'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(1024)
print 'From Server:', modifiedSentence
clientSocket.close()
```

“Services” provided to the lower layers

- "Nice to have" properties for message transport
  - Message integrity checks
  - Good throughput
  - Low transmission delays
  - Security
- This is what is available:
  - TCP (on transport layer) provides integrity and flow control
  - UDP (on transport layer) does not provide any of the properties
- Add "security" through TLS on application layer (over TCP)
- No guarantees on low transmission delays or good throughput

Transport Layer Security (TLS)

- We will discuss TLS in detail in 50.020
- TLS is somewhat "in between" application and transport layer
- It relies on TCP, but provides services to applications
- TLS can be configured in different ways, able to provide
  - Message authentication, confidentiality, integrity
- TLS is often used to improve security of application layer protocols
  - e.g. HTTP over TLS = HTTPS



Application Layer Protocols

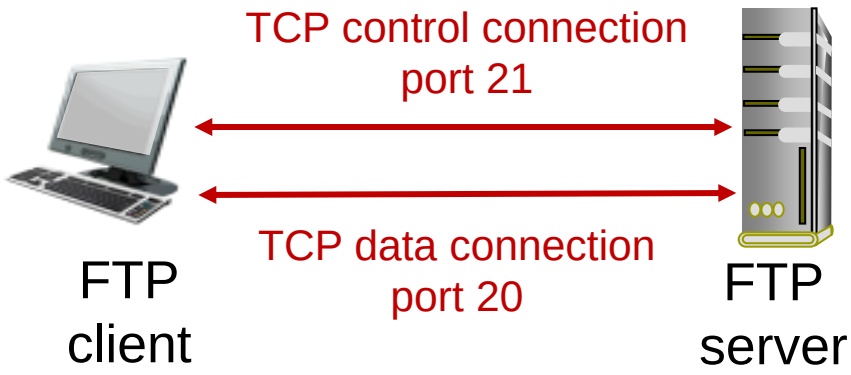
What are protocols?

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

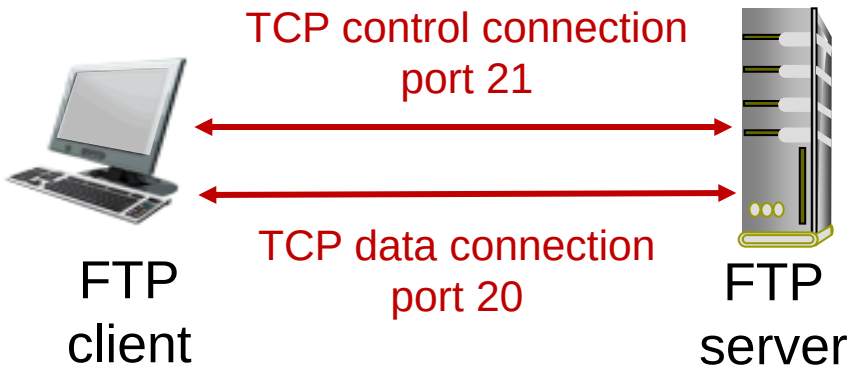
- In general: "language" spoken between machines
- Protocols generally define four aspects:
  - Types of messages exchanged (e.g., request, response)
  - Message syntax (e.g. format)
  - Message semantics (how to interpret)
  - Rules for processing of messages (how to react)
- Open protocols have these specified publicly, but proprietary protocols also exist (e.g. Skype)
- If protocols do not encrypt the messages, it is often possible to reverse-engineer unspecified protocols

FTP and HTTP

FTP: separate control, data connections

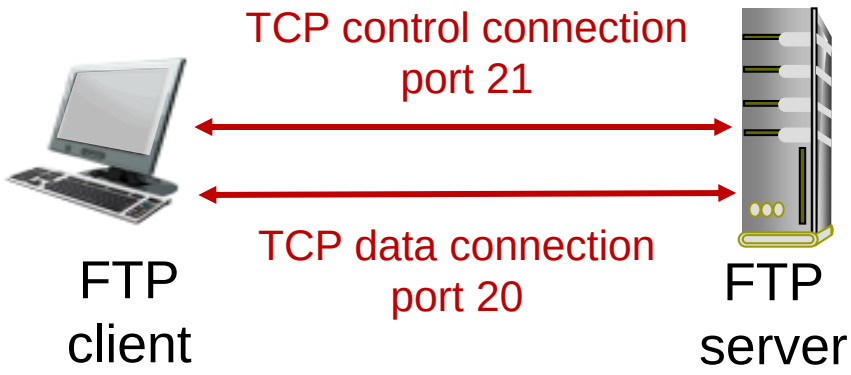


FTP: separate control, data connections



- FTP client contacts FTP server at port 21, specifying TCP as transport protocol
- Client obtains authorization over control connection
- Client browses remote directory by sending commands over control connection.
- When server receives a command for a file transfer, the server opens a TCP data connection to client
- After transferring one file, server closes connection.

FTP: separate control, data connections

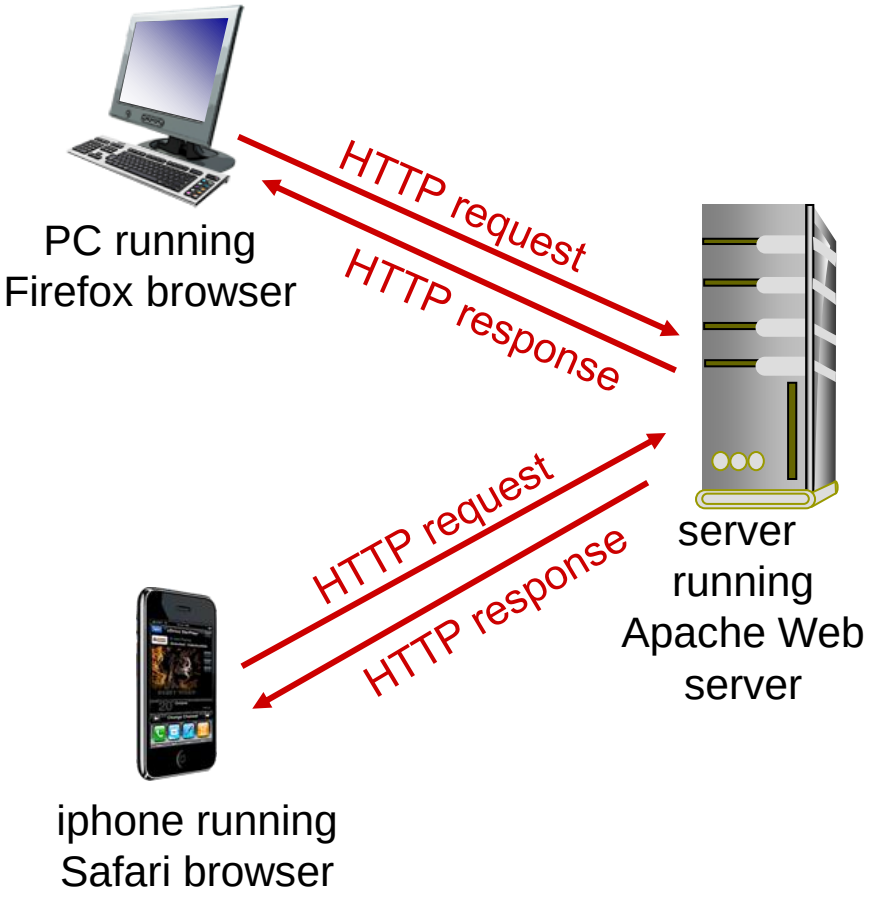


- Server opens a second TCP data connection to transfer another file.
- Control connection: “out of band”
- FTP server maintains “state”: current directory, earlier authentication

HTTP overview

HTTP: hypertext transfer protocol

- Web's application layer protocol
- client/server model
  - *client:* browser that requests, receives, (using HTTP protocol) and "displays" Web objects
  - *server:* Web server sends (using HTTP protocol) objects in response to requests



HTTP overview (continued)

uses TCP:

- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- TCP connection closed

HTTP is "stateless"

- server maintains no information about past client requests

aside

protocols that maintain "state" are complex!

- past history (state) must be maintained
- if server/client crashes, their views of "state" may be inconsistent, must be reconciled

HTTP connections

non-persistent HTTP

- at most one object sent over TCP connection
  - connection then closed
- downloading multiple objects required multiple connections

persistent HTTP

- multiple objects can be sent over single TCP connection between client, server

HTTP - details

- HTTP is a protocol that allows you to request files (e.g. html files, images)
  - Did you know that you can send data as well, or delete data?
- Types of messages (HTTP "verbs"): GET, PUT, POST, DELETE, HEAD, OPTIONS, CONNECT, . . .
- Message syntax: HTTP GET is ASCII-based, POST can also use binary data
  - Message start with verb, then the relative address of resources, then protocol version
- Most common message types: GET and POST
- Semantics (interpretation) and rules for processing also defined by RFC

HTTP GET

- HTTP GET is the most common verb: the client requests the server to send a resources identifies by an URL
- Example:

```
GET /demo.asp?name1=value1&name2=value2 HTTP/1.1
host: www.sutd.edu.sg
```

- GET requests are defined to have no side-effect on the server
- Query strings can be sent with request

HTTP POST

- HTTP POST is used to submit data, e.g., forms: the client sends the server some content
- Data can be binary or string, no length limit
- Repeating a POST message will re-post the data (possibly bad)
- Example:

```
POST /demo.asp HTTP/1.1
Host: www.sutd.edu.sg
name1=value1&name2=value2
```

- GET requests are defined to have no side-effect on the server
- Query strings can be sent with request /demo.asp?name1=value1&name2=value2

HTTP responses

- Status line: first line of the HTTP response, includes:
  - A numeric status code
  - Sentence with reason
- It is up to the clients to interpret the response codes

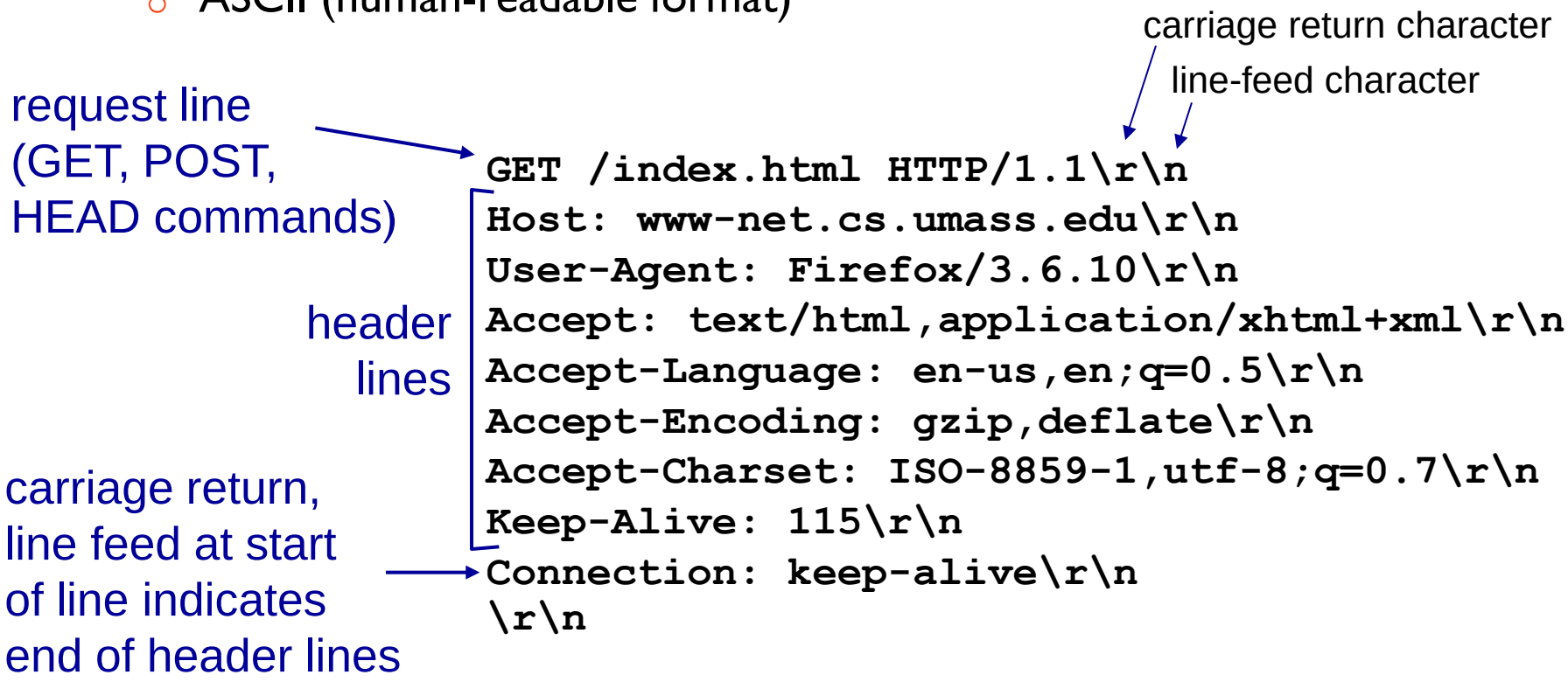
HTTP responses

- Status line: first line of the HTTP response, includes:
  - A numeric status code
  - Sentence with reason
- It is up to the clients to interpret the response codes

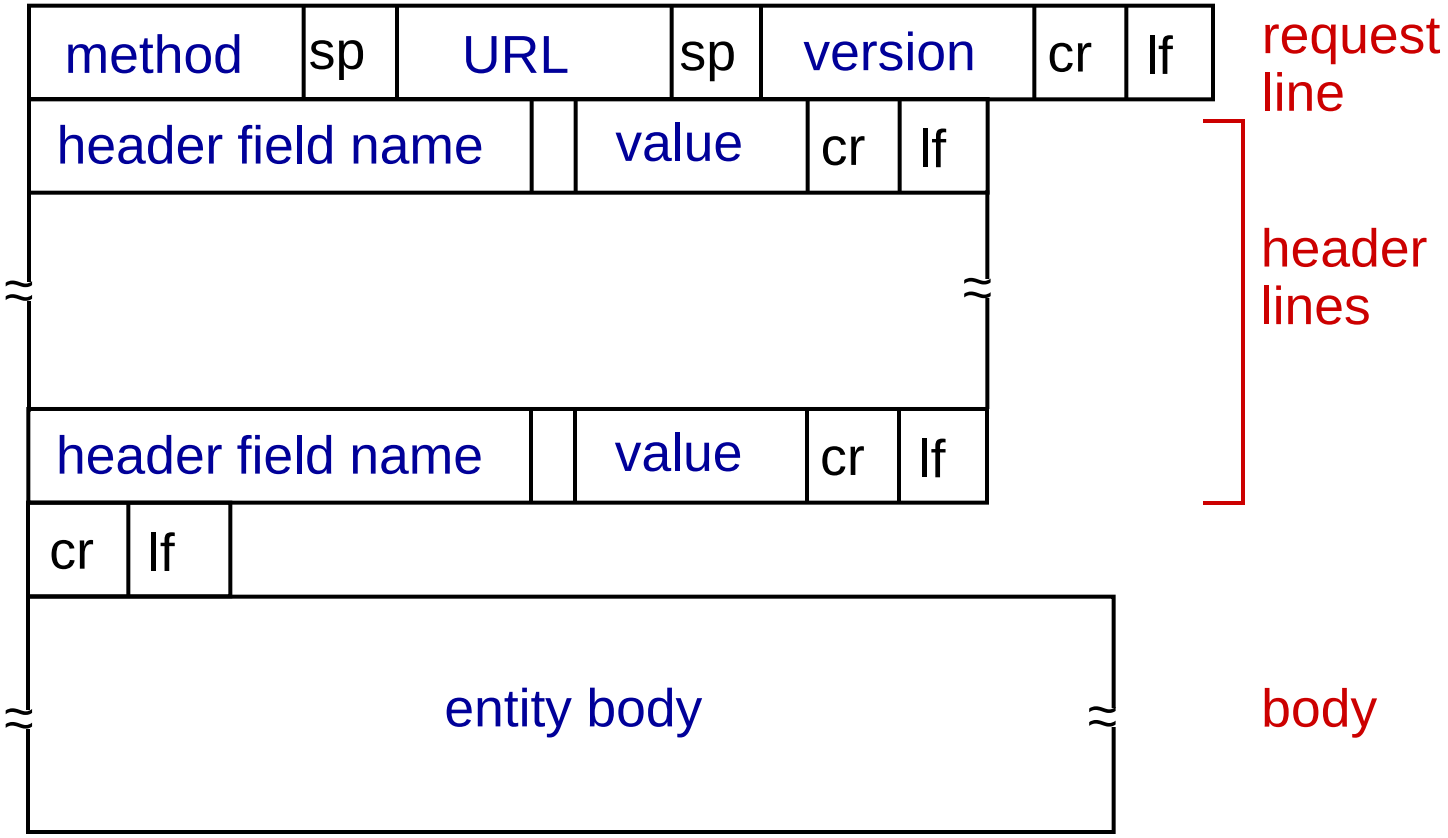
Common status codes
- 200 OK
- 201 Created (e.g. after successful POST)
- 301 Moved permanently
- 404 Not Found
- 500 Server error

HTTP request message

- two types of HTTP messages: *request, response*
- HTTP request message:
  - ASCII (human-readable format)



HTTP request message: general format



HTTP HTML response example

```
HTTP/1.1 200 OK
Server: Apache
Content-Encoding: gzip
Content-Type: text/html; charset=utf-8
Last-Modified: Mon, 02 Sep 2016 02:20:45 GMT
Expires: Mon, 02 Sep 2016 02:21:17 GMT
Cache-Control: public, max-age=32
Content-Length: 23943
Accept-Ranges: bytes
Date: Mon, 19 Sep 2017 23:59:00 GMT
Age: 15
Connection: keep-alive
Vary: User-Agent,Accept-Encoding,X-Export-Format
```

status line (protocol status code status phrase)

header lines

+ all the HTML files following

Cookies

- As HTTP servers are stateless, how are sequential requests correlated?
- First connection to server creates a cookie value, subsequent communication from user includes the cookie value
- Cookies are stored in the browser

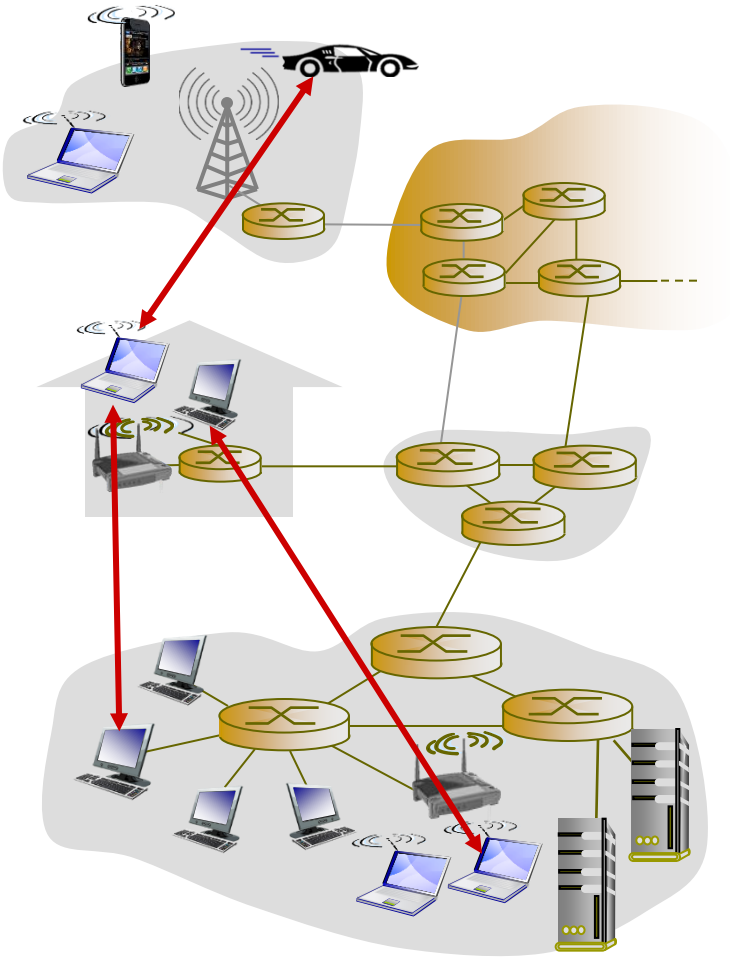
An example of P2P Applications - BitTorrent

Pure P2P architecture

- *no* always-on server
- arbitrary end systems directly communicate
- peers are intermittently connected and change IP addresses

examples:

- file distribution (BitTorrent)
- Streaming (KanKan)
- VoIP (Skype)

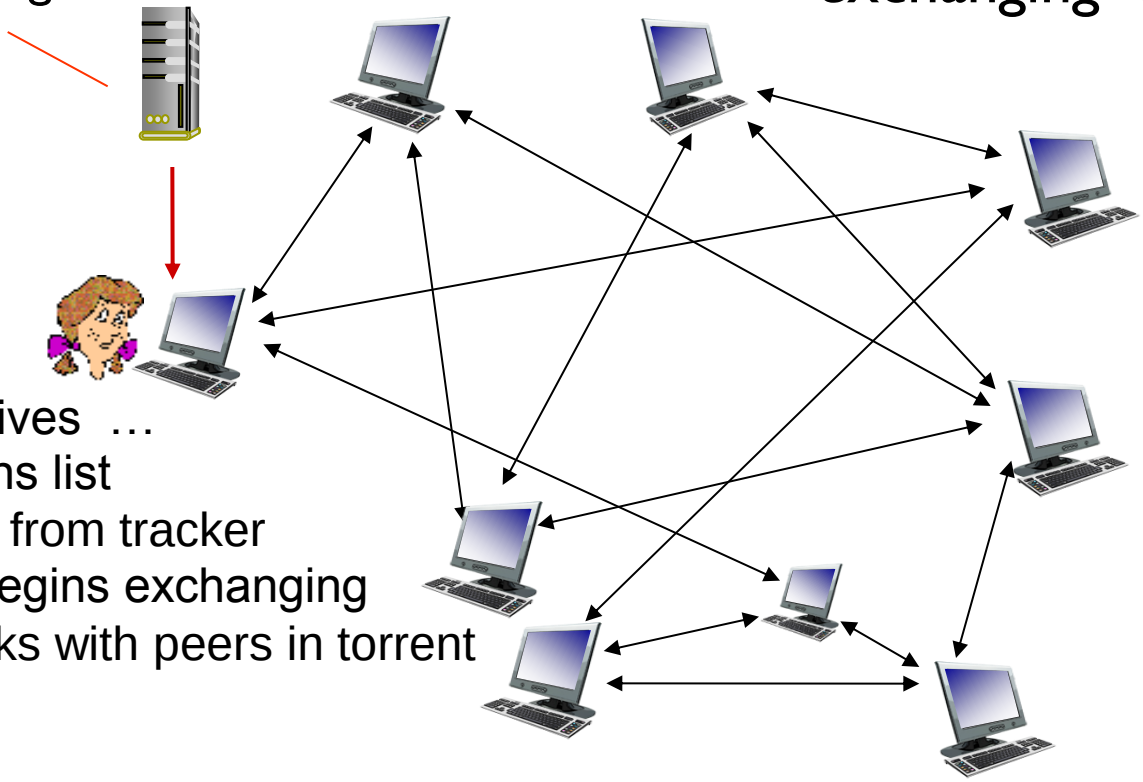


P2P file distribution: BitTorrent

- file divided into 256Kb chunks
- peers in torrent send/receive file chunks

tracker: tracks peers participating in torrent

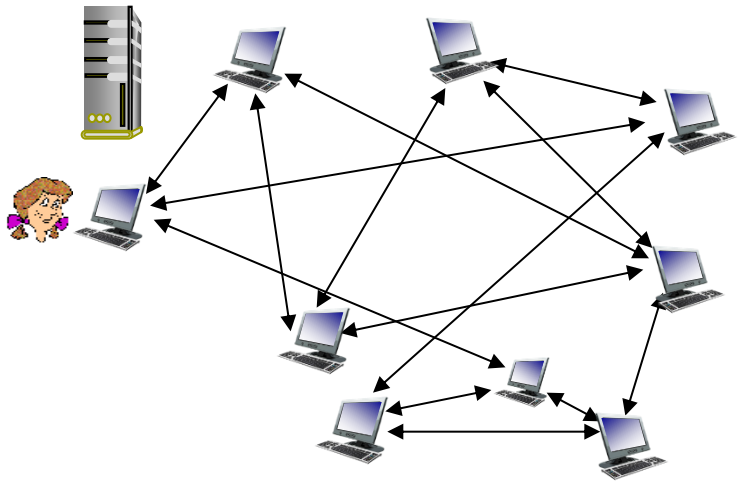
torrent: group of peers exchanging chunks of a file



Alice arrives …
… obtains list
of peers from tracker
… and begins exchanging
file chunks with peers in torrent

P2P file distribution: BitTorrent

- peer joining torrent:
  - has no chunks, but will accumulate them over time from other peers
  - registers with tracker to get list of peers, connects to subset of peers ("neighbors")



- while downloading, peer uploads chunks to other peers
- peer may change peers with whom it exchanges chunks
- *churn:* peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain in torrent

BitTorrent: requesting, sending file chunks

requesting chunks:

- at any given time, different peers have different subsets of file chunks
- periodically, Alice asks each peer for list of chunks that they have
- Alice requests missing chunks from peers, rarest first

sending chunks: tit-for-tat

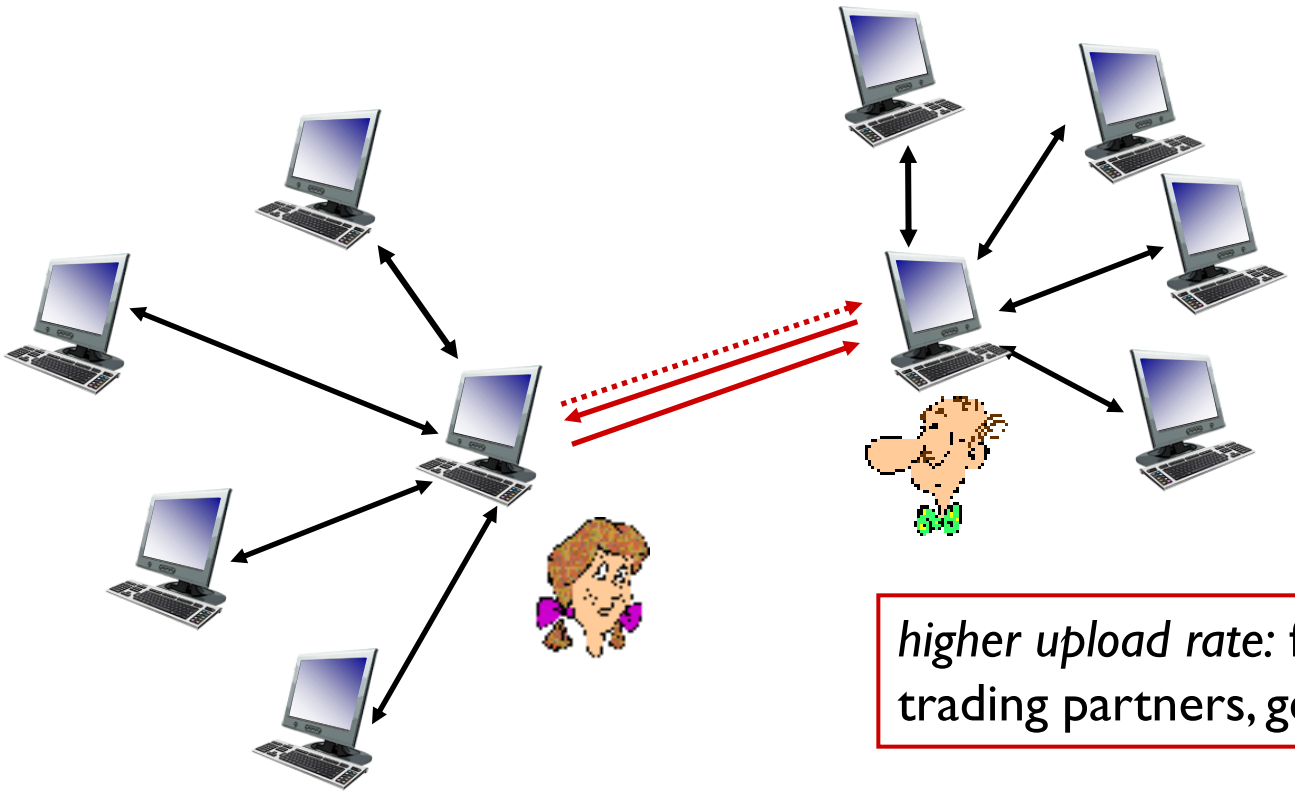
- Alice sends chunks to those four peers currently sending her chunks *at highest rate*
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every 10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - "optimistically unchoke" this peer
  - newly chosen peer may join top 4

BitTorrent: tit-for-tat

(1) Alice "optimistically unchokes" Bob

(2) Alice becomes one of Bob's top-four providers; Bob reciprocates

(3) Bob becomes one of Alice's top-four providers



higher upload rate: find better trading partners, get file faster !

Conclusion

- We discussed the application layer in more detail
- What is the interface to lower layers (Sockets)
- How addressing is done (URLs)
- Application layer protocols
- FTP and SMTP as useful application layer protocols
- HTTP as a greatly used application layer protocol
- BitTorrent as an example of P2P applications