

Remember: mic!

Geoff Coulson
Week 13 Lecture 1

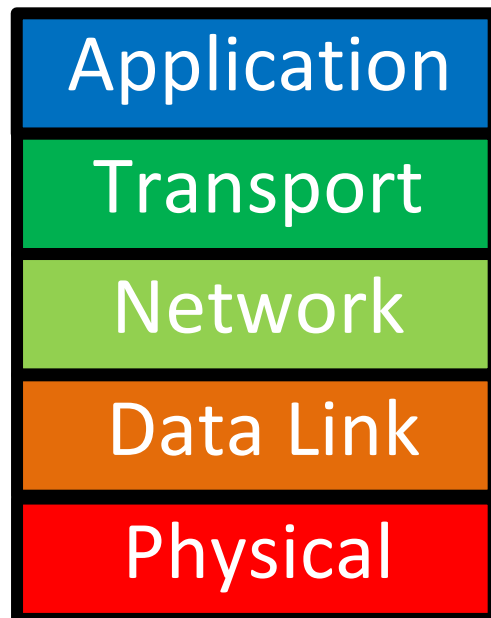
Network Applications

SCC. 203 – Computer Networks

- **Programming networked applications**
 - Interface between applications and the transport layer
 - Transport services available to applications
 - Two main transport protocols: TCP (reliable transport) and UDP (unreliable transport)
- **Some key network application architectures**
 - Client-Server
 - Peer-to-peer
 - Different types of structuring of peer-to-peer networks
 - Content distribution networks

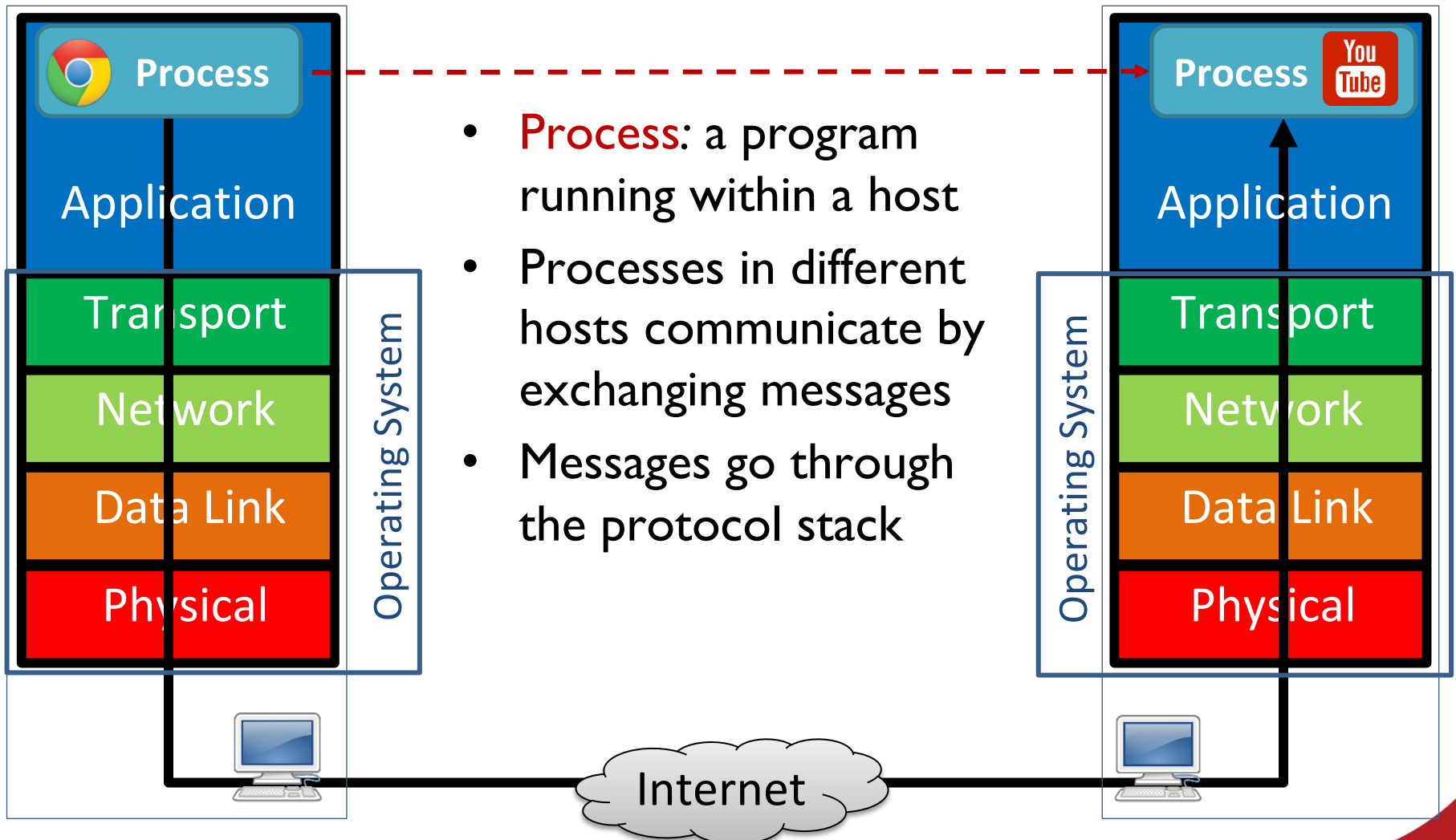
The TCP/IP Protocol Stack

- Internet functionality is split among different **layers**
- Header **encapsulation** used to achieve *separation of concerns* between each layer

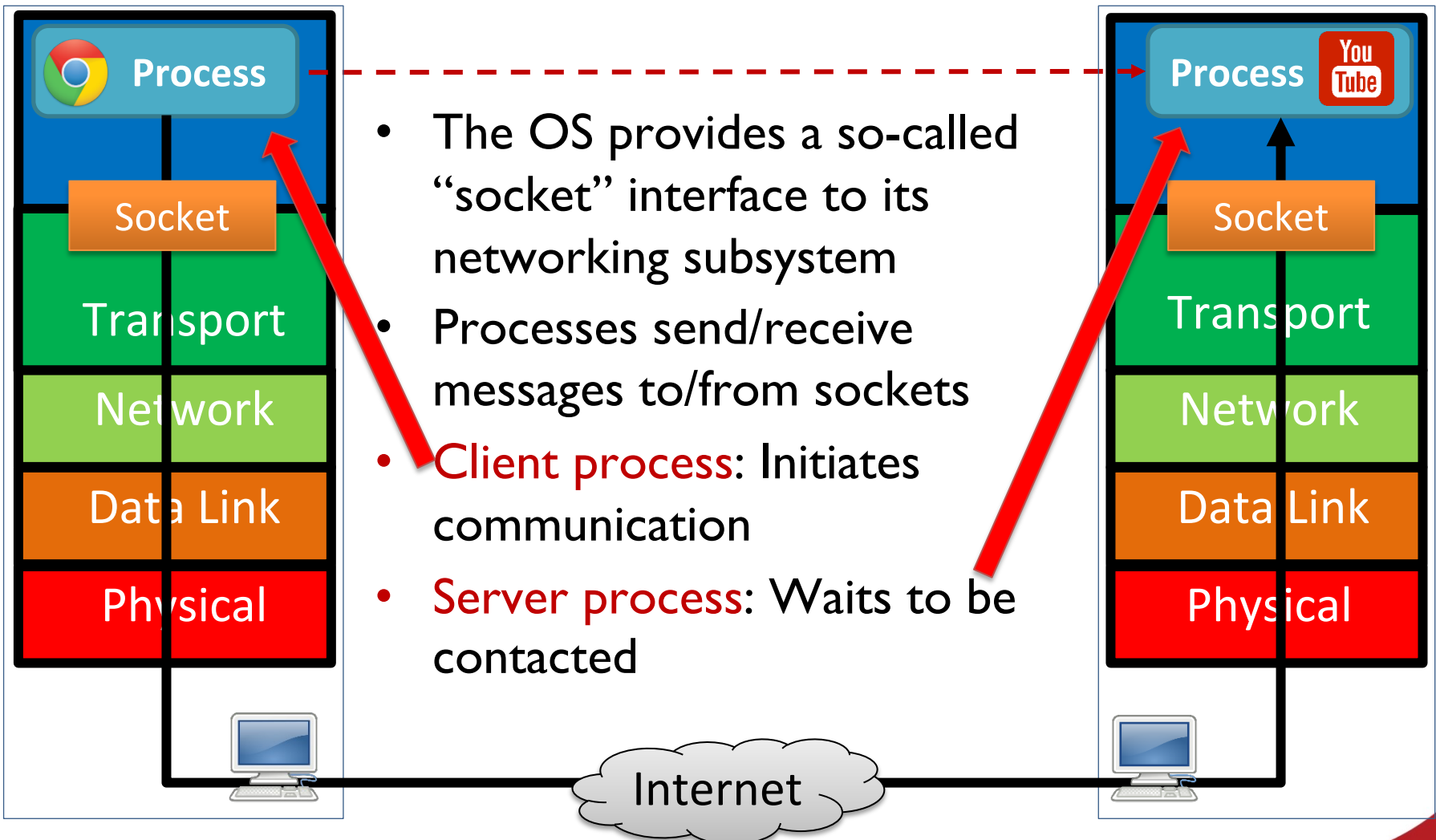


- Advantages
 - *Helps manage complexity through modularity*
- Disadvantages
 - *Performance overheads?*
 - *Redundant functionality?*

Communicating processes



Sockets: The interface between Processes and Transport Protocols



Identifying processes remotely

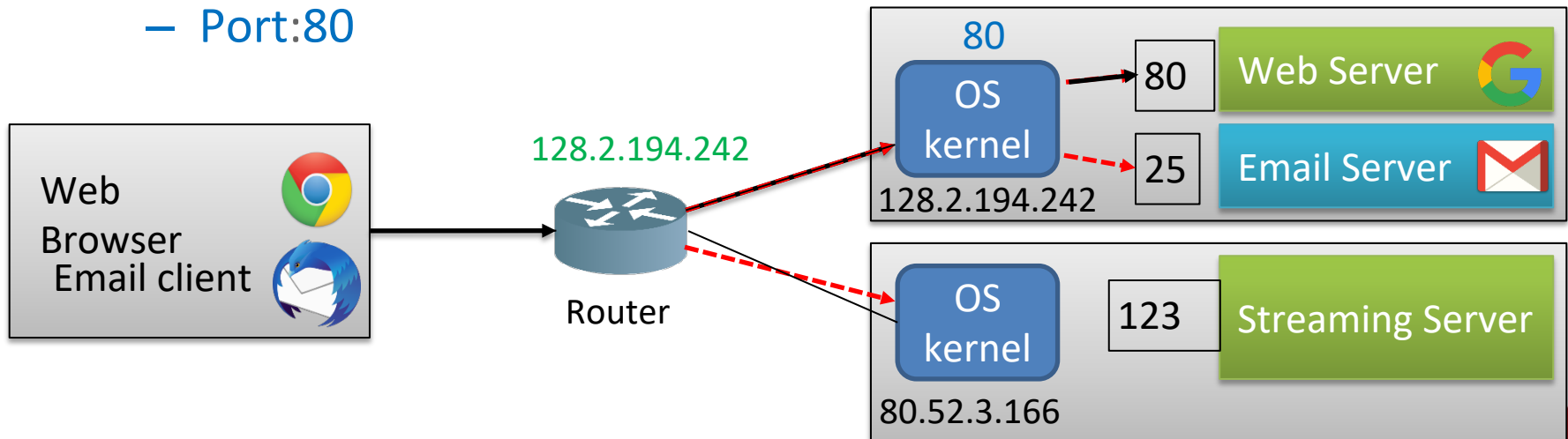
- A process is addressed over the network using two identifiers:
 - IP address
 - Network Layer identifier
 - 32 bits (IPv4) or 64 bits (IPv6)
 - Port number
 - Transport Layer identifier (16 bits)
- A “**5 tuple**” uniquely identifies traffic between hosts:
 - Two IP addresses, two port numbers, and the underlying transport protocol being used (e.g., TCP or UDP)
 - example: <206.62.226.35, 21, 198.69.10.2, 1500, TCP>
 - example: <206.62.226.35, 21, 198.69.10.2, 1499, UDP>

Why do we need both IP addresses and port numbers?

- Many processes may be running on the same host
- IP address is used by routers to forward messages to the correct host
- Then, the host's OS uses the port number to forward messages to the correct target process/ socket

Why do we need both IP addresses and port numbers?

- Many processes may be running on the same host
- IP address is used by routers to forward messages to the correct host
- Then, the host's OS uses the port number to forward messages to the correct target process/ socket
- Example: Request for a Web service
 - IP address: 128.2.194.242
 - Port:80



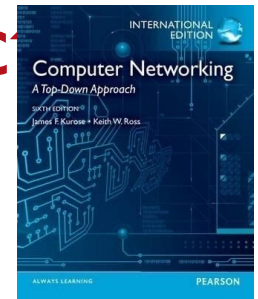
Knowing What Port Number To Use

- Popular applications have well-known ports
 - E.g., port 80 for Web and port 25 for e-mail
 - See <http://www.iana.org/assignments/port-numbers>
- Well-known ports vs. ephemeral ports
 - Typically, servers have a well-known port (e.g., port 80)
 - In range 0 - 1023 (requires root privileges to use)
 - Client picks an unused ephemeral (i.e., temporary) port
 - In range 1024 - 65535

UNIX's Socket API

- In UNIX, almost every input and output, physical and virtual, is made to *look like a file*
 - Files are represented by integer file descriptors
 - All input is like reading a file; all output is like writing a file
- So, a socket is like a file
 - E.g., 'standard' API calls ("system calls") are used - like `send()`, `recv()`, `close()`
 - plus some other socket-specific calls

Protocols from an application perspective



What do protocols have to do with this?

Application layer protocols

- ALP's are defined by
 - Types of messages employed (e.g., request, response, add-new-user, ...)
 - Message syntax and semantics
 - What fields messages have & how they are delineated
 - Meaning of information in fields
 - When and how processes, send & respond to messages
- Open protocols (e.g., HTTP, SMTP)
 - Defined in RFCs
 - Allow for interoperability
- Proprietary protocols (e.g., Skype, AppleTalk)
 - Tied to specific products

What might an application (or ALP) need from a transport service?

- **Data integrity**

- Some apps require 100% reliable data transfer (e.g. file transfer, web transactions)
- Other apps (e.g., audio) can tolerate some loss

- **Timing**

- Some apps (e.g., Internet telephony, interactive games) require low delay

- **Throughput**

- Some apps require a minimum level of throughput (e.g. multimedia)
- Other apps can make use of whatever throughput they get (“elastic apps”)

- **Security**

- Encryption, data integrity, ...

Transport service requirements of popular applications

Application	Data loss	Throughput	Time sensitive
File transfer	no loss	elastic	no
Email	no loss	elastic	no
Web documents	no loss	elastic	no
Real-time audio/video	loss-tolerant	audio: 5kbps-1Mbps video: 10kbps-5Mbps	yes, 100's msec
Stored audio/video	loss-tolerant	same as above	yes, few secs
Interactive games	loss-tolerant	few kbps up	yes, 100's msec
Text messaging	no loss	elastic	no

Internet transport protocols

TCP service:

- *Reliable transport* between sending and receiving process
 - All messages are delivered in the order they were sent
- *Flow control*: sender can't overwhelm receiver
 - + *Congestion control*: sender is throttled when network is overloaded
- *Connection-oriented*: initial setup is required between client and server processes
- *Does not provide*: timeliness, minimum throughput guarantee, security

UDP service:

- *Unreliable data transfer* between sending and receiving process
- *Does not provide*: reliability, in-order delivery, flow control, congestion control, timeliness, throughput guarantee, security, or connection setup

BUT timeliness *may* be better than TCP; and less overhead may be incurred for both hosts and the network

Internet apps: application, transport protocols

application	application layer protocol	underlying transport protocol
e-mail	SMTP [RFC 2821]	TCP
remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
file transfer	FTP [RFC 959]	TCP
streaming multimedia	HTTP (e.g., YouTube), RTP [RFC 1889]	TCP or UDP
Internet telephony	SIP, RTP, proprietary (e.g., Skype, Zoom)	TCP or UDP

Socket programming with UDP

- No “connection” between client & server
 - Sender explicitly attaches IP destination address and port # to each packet
 - Receiver extracts sender IP address and port number from received packet
 - Messages are called “datagrams”

Python example: UDP echo client

include Python's socket library	→	from socket import *
		serverName = 'hostname'
		serverPort = 12000
create UDP socket for server	→	clientSocket = socket(AF_INET, SOCK_DGRAM)
get user keyboard input	→	message = raw_input('Input text:')
Attach server name, port to message; send into socket	→	clientSocket .sendto(message.encode(), (serverName, serverPort))
read reply characters from socket into string	→	echoedMessage, serverAddress = clientSocket .recvfrom(2048)
print out received string and close socket	→	print echoedMessage.decode() clientSocket .close()

Python example: UDP echo server

```

from socket import *
serverPort = 12000

create UDP socket → serverSocket = socket(AF_INET, SOCK_DGRAM)

“bind” socket to local port
number 12000 → serverSocket.bind(("", serverPort))

print ("The server is ready to receive")

loop forever → while True:
    Read from UDP socket into
    message, also getting
    client's address (client IP
    and port) → message, clientAddress = serverSocket.recvfrom(1024)

    echo message back to this
    client → serverSocket.sendto(message.encode(),
                                clientAddress)
    
```

Socket programming with TCP

- Recall: TCP provides **reliable**, in-order byte-stream transfer (a “**pipe**”) between a client and a server
- TCP service is connection-oriented:
 - When a client application creates a socket, the client’s TCP instance establishes a connection to the server’s TCP instance
- When contacted by a client, a server TCP creates a new socket for server process to communicate with that specific client

Python example: TCP echo client

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect(serverName, serverPort)
sentence = raw_input('Input text:')
clientSocket.send(sentence.encode())
echoedSentence = clientSocket.recv(1024)
print ('From Server:', echoedSentence.decode())
clientSocket.close()
```

create TCP socket for
server, remote port 12000



No need to attach server
name, port



Python example: TCP echo server

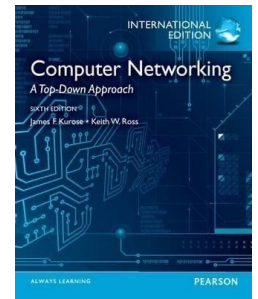
	from socket import *
	serverPort = 12000
create TCP welcoming socket	→ serverSocket = socket(AF_INET, SOCK_STREAM)
	serverSocket.bind(('',serverPort))
server begins listening for incoming TCP requests	→ serverSocket.listen()
	print 'The server is ready to receive'
loop forever	→ while True:
server waits on accept() for incoming requests, new socket created on return	→ connectionSocket, addr = serverSocket.accept()
	sentence = connectionSocket.recv(1024).decode()
read bytes from socket (but not address as in UDP)	
	connectionSocket.send(sentence, encode())
close connection to this client (doesn't close the "welcoming" socket)	→ connectionSocket.close()

What about security?

- TCP & UDP have no native encryption
 - e.g., clear-text passwords sent into a socket traverse Internet as clear-text
- SSL – Secure Socket Layer
 - Provides encrypted TCP connections
 - Also provides end-point authentication
- SSL is at application layer
 - applications use SSL libraries that “talk” to TCP
- (Now superseded by Transport Layer Security (TLS))

Application Architectures

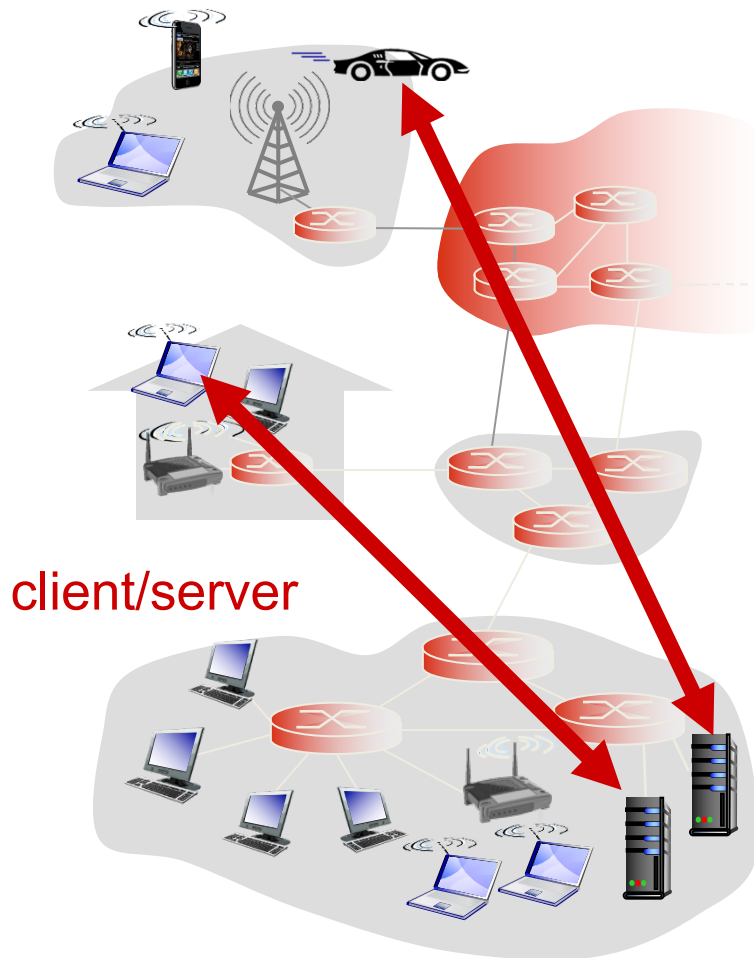
The basic communication concepts



Application architecture

- Main categories of application architecture
 - Client-Server
 - Peer-to-Peer (P2P)

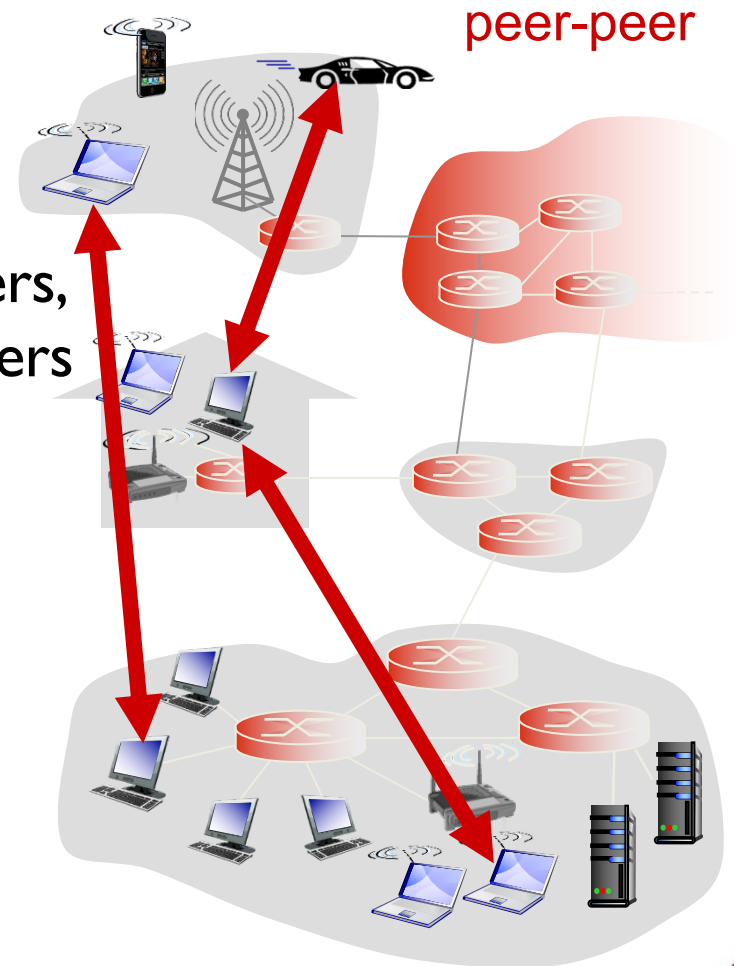
Client-Server architecture



- **Server:**
 - An always-on host
 - Has permanent IP address
 - (Can use data centres for scaling)
- **Clients:**
 - May be only intermittently connected
 - Private/ephemeral IP addresses
 - Do not communicate directly with each other

P2P architecture(s)

- No always-on server
- Arbitrary end-systems inter-communicate directly
- Peers request service from other peers, provide service in return to other peers
- *Self-scalability* – as well as making new service demands, new peers typically also bring new service capacity
- Peers are intermittently connected and IP addresses change routinely
 - Complex management!



P2P Networks: Napster



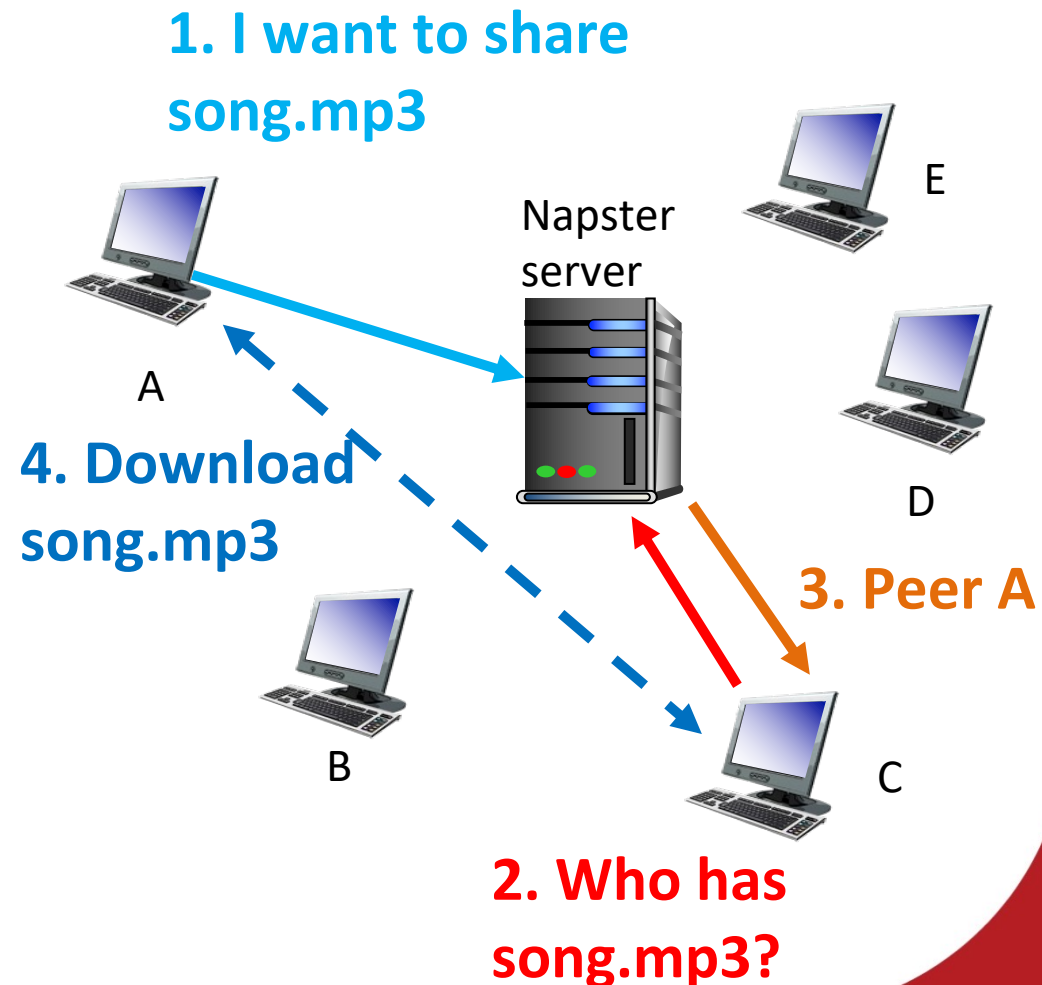
- Napster history: the rise
 - 1/99: Napster version 1
 - 5/99: company founded
 - 12/99: first lawsuits
 - 2000: 80 million users
- Napster history: the fall
 - Mid 2001: out of business due to lawsuits
 - Mid 2001: *dozens of decentralized P2P alternatives*
 - 2003: *emergence of pay services like Spotify/iTunes*



**Shawn Fanning,
Creator of Napster
Northeastern freshman**

How Napster worked

- Content location is centralised; but file transfer is decentralised
- Advantages:
 - Fast lookup time
- Disadvantages
 - Single point of failure for lookup
 - also: performance bottleneck
 - (Copyright infringement!)

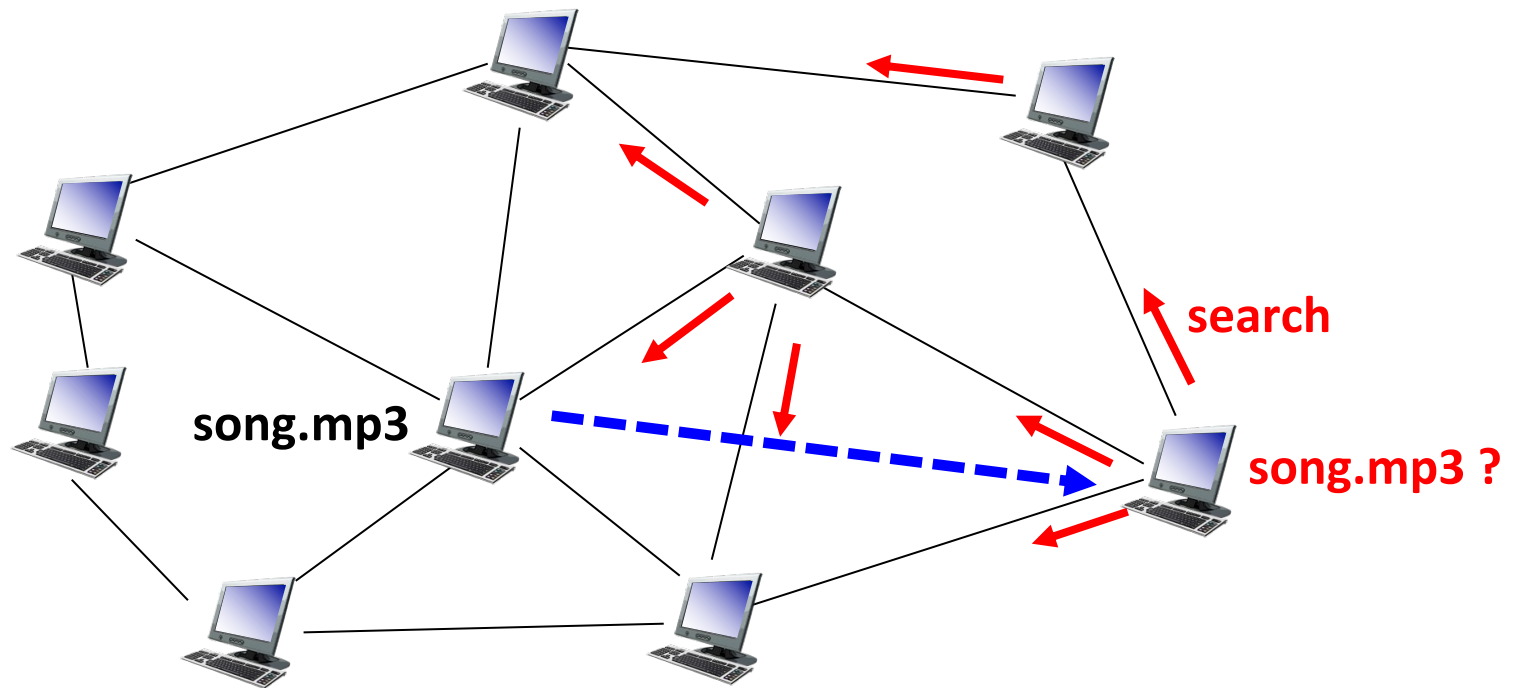




- Gnutella history
 - 2000: J. Frankel & T. Pepper released Gnutella
 - Soon after: many other clients (e.g., Morpheus, Limewire, Bearshare)
 - 2001: protocol enhancements, e.g., “supernodes/ultrapeers”
- Query flooding
 - **Join**: contact a few random(ish) nodes to become neighbours of
 - **Search**: I ask my neighbours, who ask their neighbours - if a node has it, it replies to me
 - **Fetch**: I get file directly from a node that claims to have it

No need for “publish” step

How Gnutella worked: Query flooding



- Advantages

- Fully decentralised - no bottleneck
- Distributed cost of searching

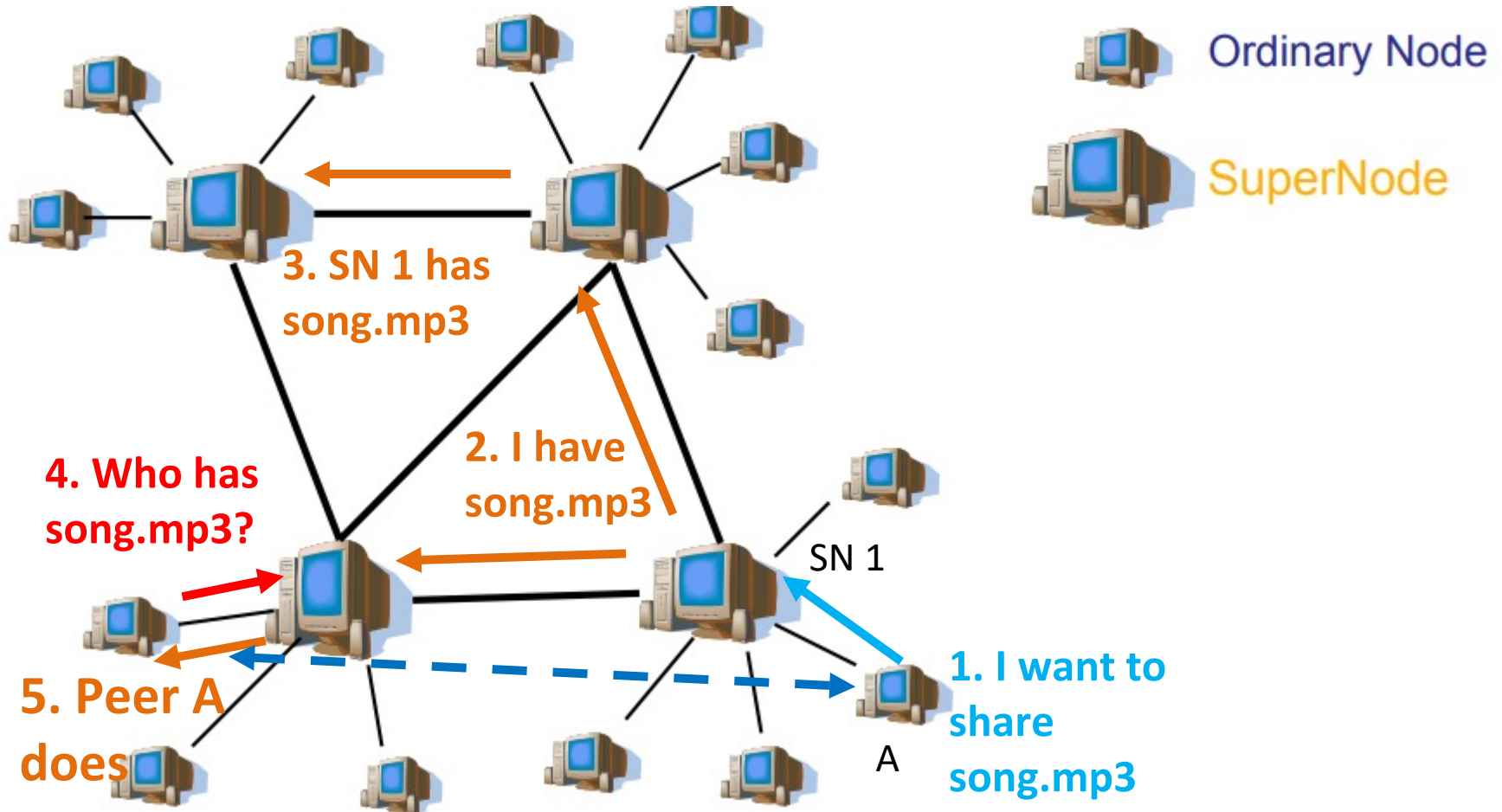
- Disadvantages

- Potentially long search time
- Prone to “free-loading”



- KaZaA history
 - 2001: created by Dutch company (Kazaa BV)
 - A single network called *FastTrack* was initially used by other clients as well
 - Eventually protocol changed so others could no longer use it
- Super-node hierarchy
 - **Join**: on startup, the client contacts a “super-node”
 - **Publish**: client sends a list of its available files to its super-node
 - **Search**: queries flooded among super-nodes
 - **Fetch**: get file directly from one or more peers

How KaZaa worked



P2P Networks: BitTorrent

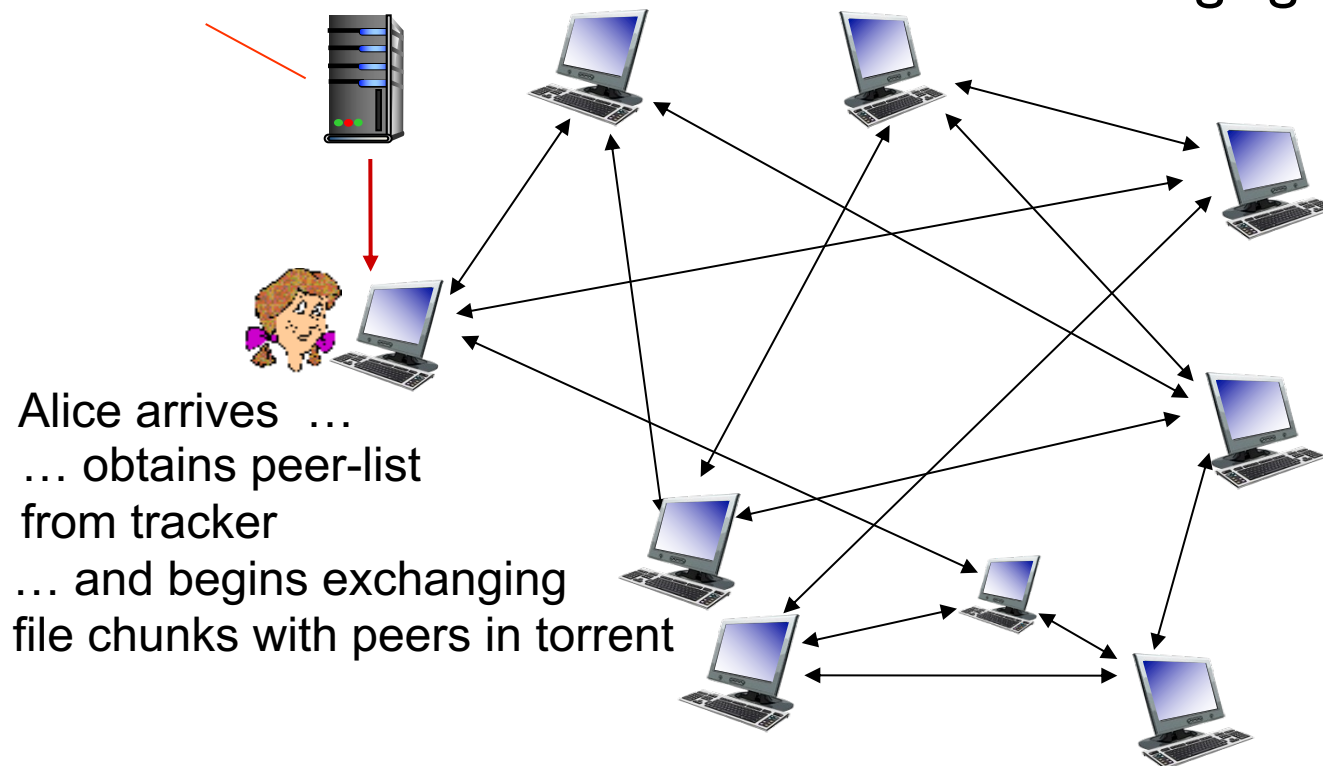


- BitTorrent history
 - 2002: debuted by B. Cohen
- Emphasis on efficient **fetching**, not searching
 - Distribute same file to many peers
 - Single publisher, many downloaders
- Preventing “free-loading”
 - Incentives for peers to contribute

BitTorrent Components

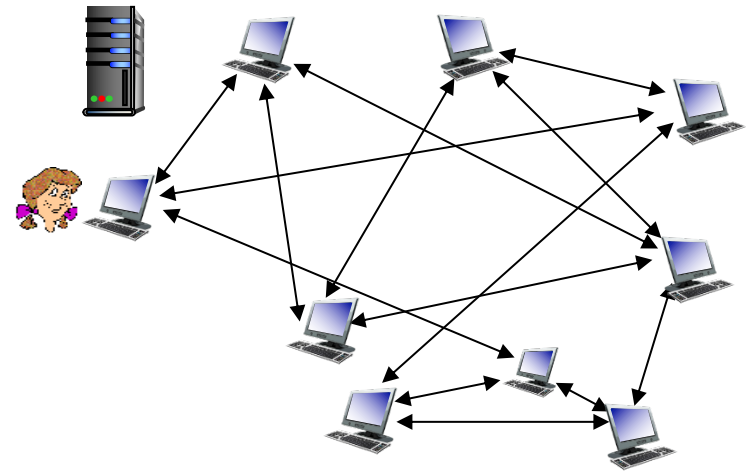
tracker: tracks peers participating in a torrent

torrent: a group of peers exchanging chunks of a file



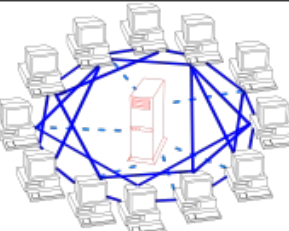
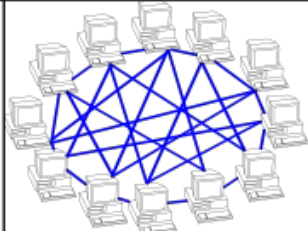
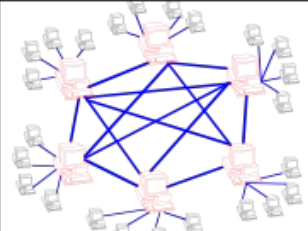
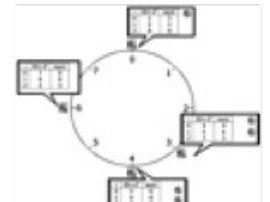
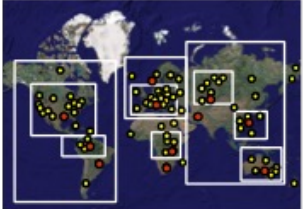
How BitTorrent works

- A new peer joining a torrent:
 - registers with tracker to get list of peers, connects to subset of peers (“neighbours”)
 - has no chunks initially, but will accumulate them over time from other peers



- 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

P2P Systems – Unstructured vs. Structured

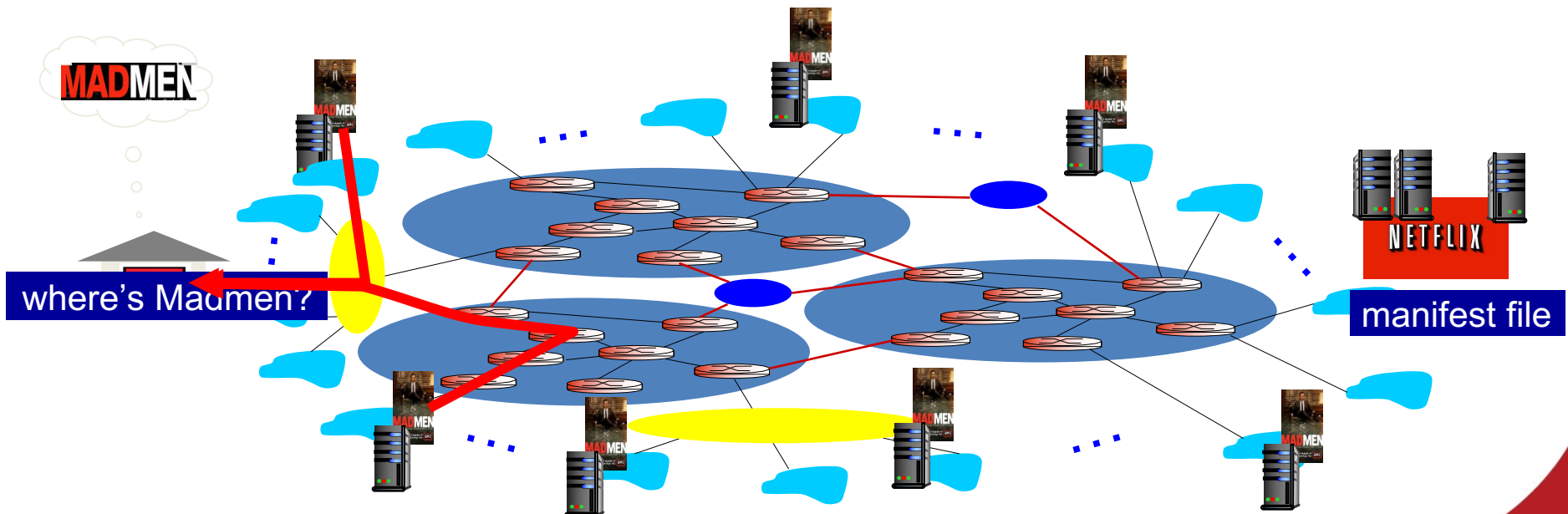
Unstructured P2P			Structured P2P	
Centralized P2P	Pure P2P	Hybrid P2P	DHT-Based	Hybrid P2P
<ol style="list-style-type: none"> 1. All features of Peer-to-Peer included 2. Central entity is necessary to provide the service 3. Central entity is some kind of index/group database <p>Examples:</p> <ul style="list-style-type: none"> ▪ Napster 	<ol style="list-style-type: none"> 1. All features of Peer-to-Peer included 2. Any terminal entity can be removed without loss of functionality 3. → no central entities <p>Examples:</p> <ul style="list-style-type: none"> ▪ Gnutella 0.4 ▪ Freenet 	<ol style="list-style-type: none"> 1. All features of Peer-to-Peer included 2. Any terminal entity can be removed without loss of functionality 3. → dynamic central entities <p>Examples:</p> <ul style="list-style-type: none"> ▪ Gnutella 0.6 ▪ Fasttrack ▪ eDonkey 	<ol style="list-style-type: none"> 1. All features of Peer-to-Peer included 2. Any terminal entity can be removed without loss of functionality 3. → No central entities 4. Connections in the overlay are “fixed” <p>Examples:</p> <ul style="list-style-type: none"> ▪ Chord ▪ CAN ▪ Kademlia 	<ol style="list-style-type: none"> 1. All features of Peer-to-Peer included 2. Peers are organized in a hierarchical manner 3. Any terminal entity can be removed without loss of functionality <p>Examples:</p> <ul style="list-style-type: none"> • RecNet • Globase.KOM
				

Content distribution (streaming)

- **Challenge:** how to stream content to hundreds of thousands of simultaneous users?
- **Option 1:** single, large “mega-server”
 - Single point of failure
 - Point of network congestion
 - Long path to distant clients
 - Multiple copies of video sent over outgoing link
- **Option 2:** store/serve multiple copies of videos at multiple geographically distributed sites (**CDN**)
 - Close to users
 - Resilient

Content Distribution Networks (CDNs)

- CDN: stores copies of given content at CDN nodes
- Subscriber requests content from CDN
 - is directed to nearby copy, retrieves content
 - may choose different copy if network is congested



Summary

-
- We've seen some basic principles of network applications...
 - Application Layer requirements
 - Application protocols overview
 - Transport Service Requirements
 - Basic socket programming
 - Socket Types
 - Socket API
 - Application architectures
 - Client/ Server vs. Peer-to-Peer
 - some Peer-to-Peer variants

Thanks for listening!
Any questions?

g.coulson@lancaster.ac.uk