Chapter 2

Application layer

Liping Shen 申丽萍

lpshen@sjtu.edu.cn

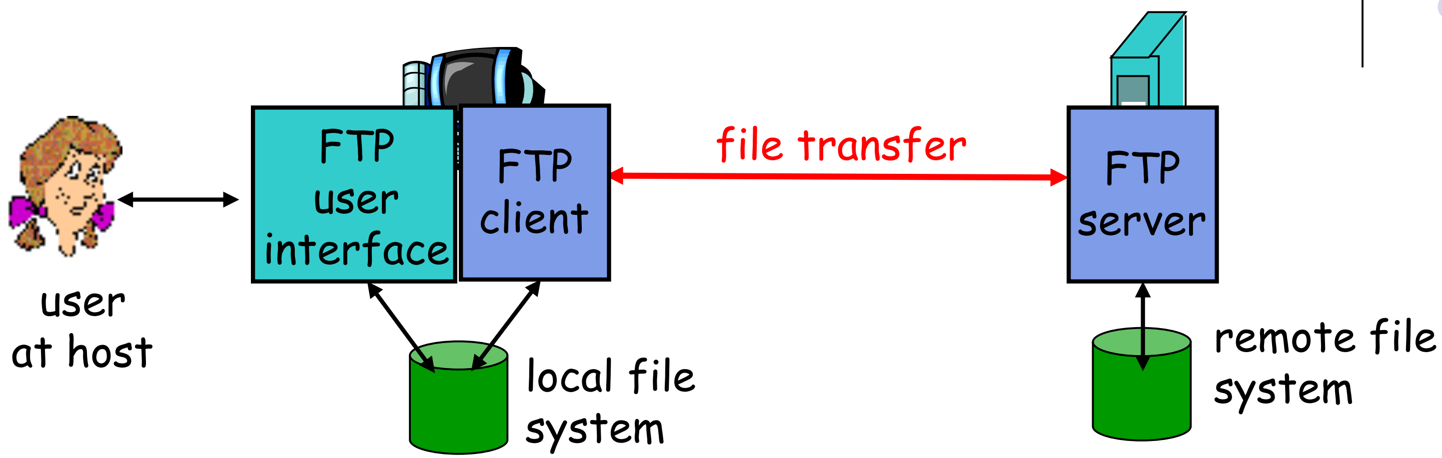
http://jpkc.seiee.sjtu.edu.cn/jsjwl

These slides are based upon the exceptional slides provided by Kurose and Ross

Chapter 2: Roadmap

- Principles of network applications
- DNS
- Web and HTTP
- FTP
- Electronic Mail
  - SMTP, POP3, IMAP
- P2P applications
- Socket programming with TCP
- Socket programming with UDP

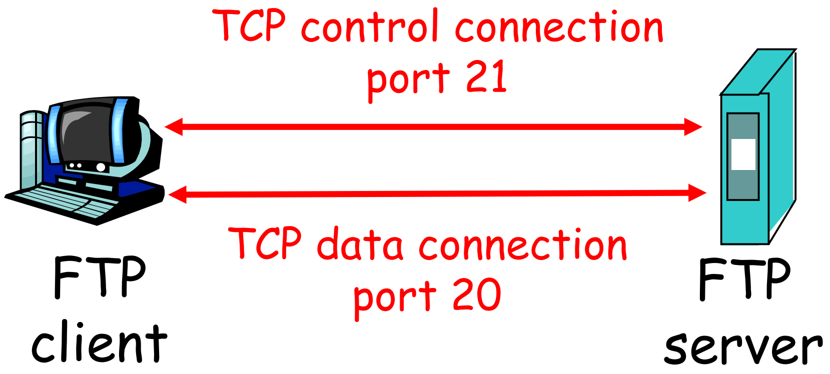
FTP: the file transfer protocol (RFC 959)



- transfer file to/from remote host
- client/server model
  - *client:* side that initiates transfer (either to/from remote)
  - *server:* remote host

FTP: separate control, data connections

- TCP control connection:
  - authorization
  - sending commands
  - "out of band", persistent
- TCP data connection:
  - one connection for one file
  - connection closed after transferring one file.
  - active vs. passive
  - non-persistent
- FTP server maintains "state": current directory, earlier authentication



FTP commands, responses

Sample commands:

- sent as ASCII text over control channel
- **`USER`** ***`username`***
- **`PASS`** ***`password`***
- **`LIST`** return list of file in current directory
- **`RETR filename`** retrieves (gets) file
- **`STOR filename`** stores (puts) file onto remote host

Sample return codes

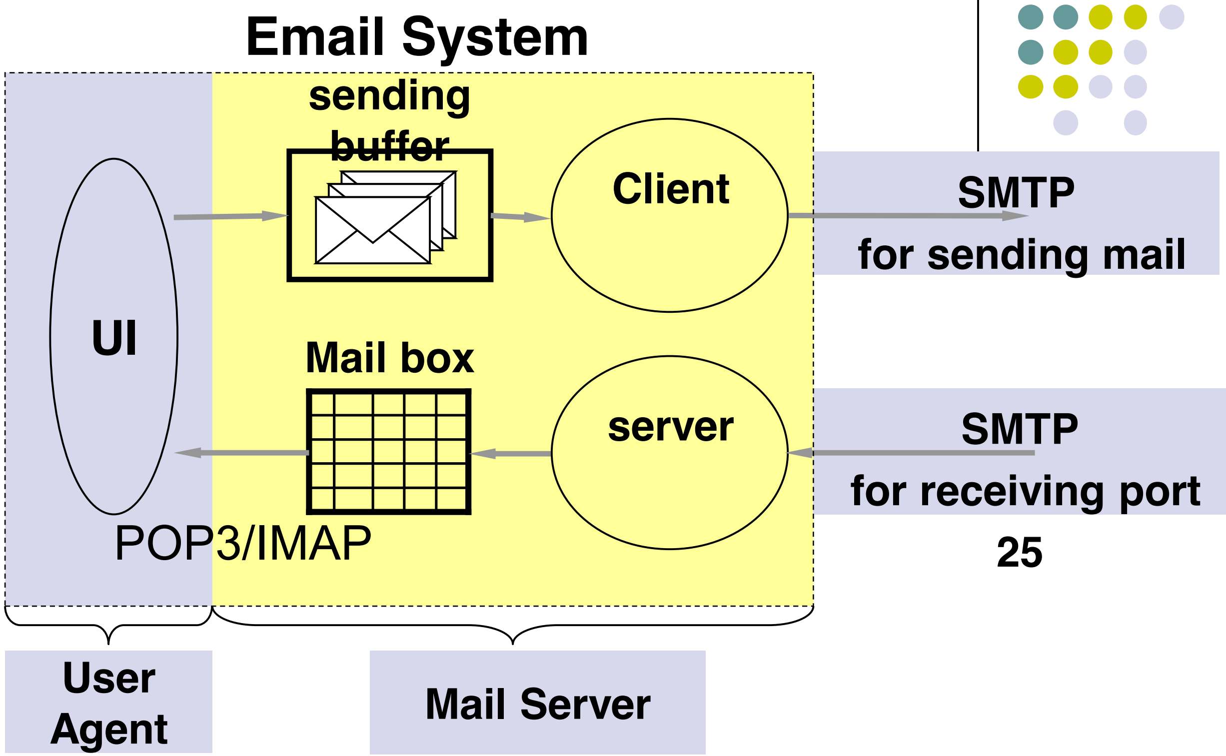
- status code and phrase (as in HTTP)
- **`331 Username OK, password required`**
- **`125 data connection already open; transfer starting`**
- **`425 Can't open data connection`**
- **`452 Error writing file`**

Telnet

Chapter 2: Roadmap

- Principles of network applications
- DNS
- Web and HTTP
- FTP
- Electronic Mail
    - SMTP, POP3, IMAP
- P2P applications
- Socket programming with TCP
- Socket programming with UDP

Electronic Mail



Four components:

- user agents
- mail servers
- transfer protocol: SMTP
- access protocols: POP3, IMAP

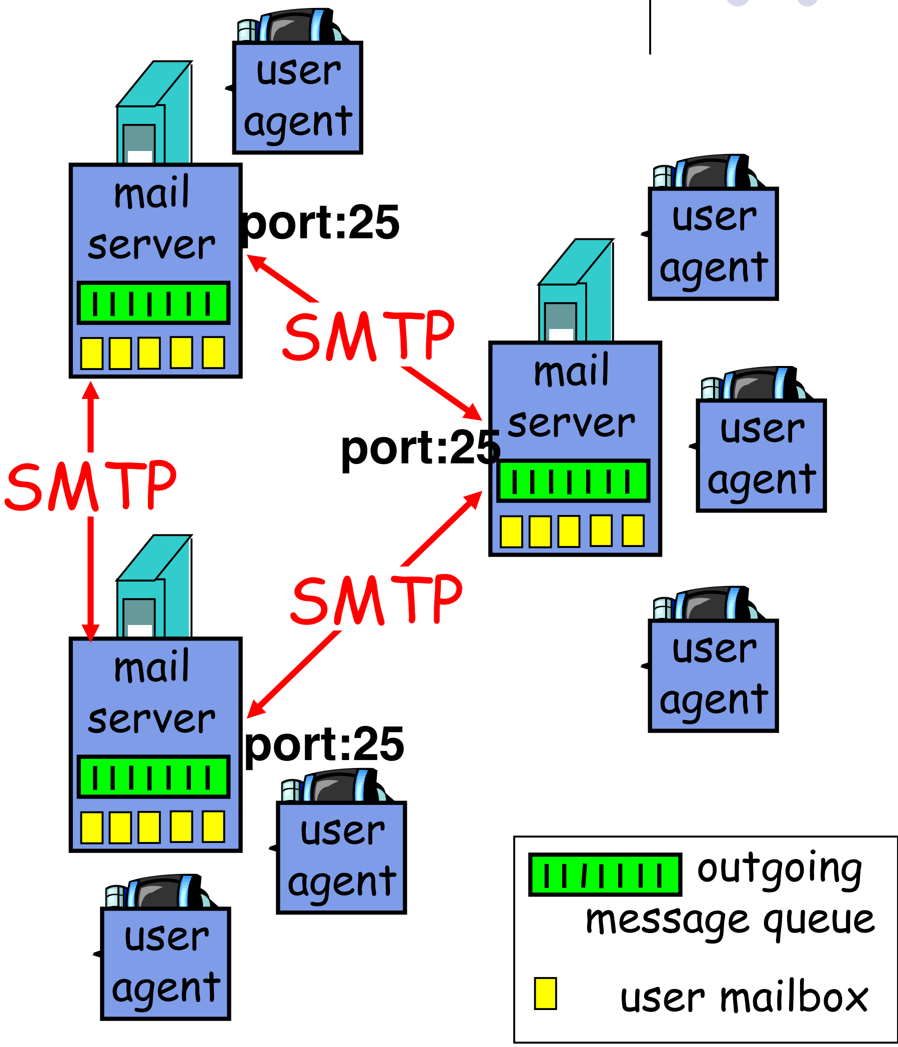
User Agent

- a.k.a. "mail reader"
- composing, editing, reading mail messages
- e.g. Outlook, foxmail, Eudora
- outgoing, incoming messages stored on server

Electronic Mail: mail servers

Mail Servers

- mailbox contains incoming messages for user
- message queue of outgoing (to be sent) mail messages
- SMTP protocol between mail servers to send email messages
  - duplex TCP connection at port 25
  - client: sending mail server
  - server: receiving mail server

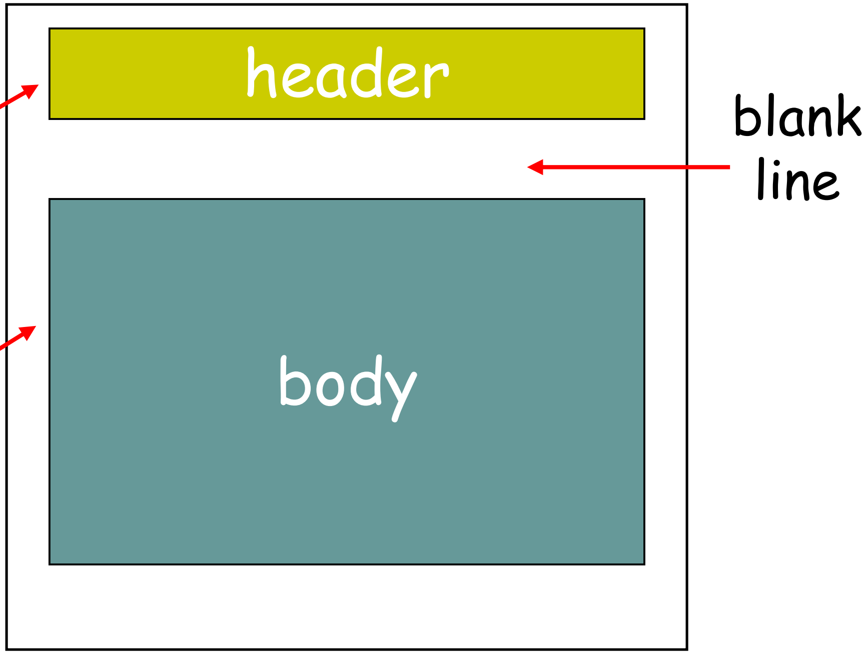


Mail message format (RFC822)

SMTP: protocol for exchanging email msgs

RFC 822: standard for text message format:

- header lines, e.g.,
  - To:/CC:/BCC:
  - From:/Sender:/Reply-To:
  - Subject:/Keywords:
  - X-**: user defined
- body
  - the “message”, ASCII characters only



MIME [RFC 1341]
Multipurpose Internet Mail Extensions

- Problems with RFC822 ASCII-based email system:
  - Languages with accents (French, German).
  - Languages in non-Latin alphabets (Hebrew, Russian).
  - Languages without alphabets (Chinese, Japanese).
  - Messages not containing text at all (audio or images).
- MIME: to continue to use the RFC 822 format, but to add structure to the message body and define encoding rules for non-ASCII messages.

MIME Message Headers

- MIME defines five new message headers.
- Binary data should be sent encoded in base64 or quoted-printable form.

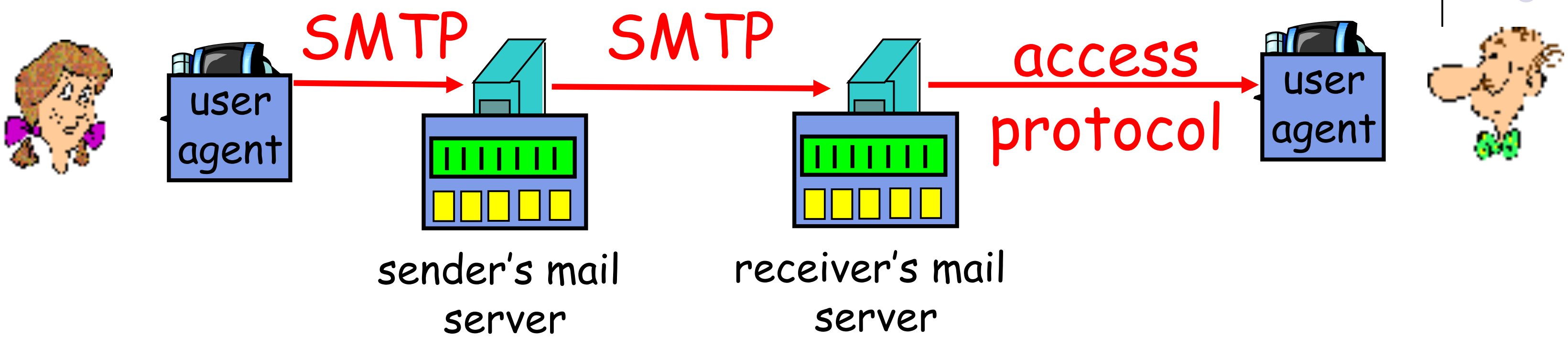
Header	Meaning
MIME-Version:	Identifies the MIME version
Content-Description:	Human-readable string telling what is in the message
Content-Id:	Unique identifier
Content-Transfer-Encoding:	How the body is wrapped for transmission
Content-Type:	Type and format of the content

MIME Content Types

Type	Subtype	Description
Text	Plain	Unformatted text
	Enriched	Text including simple formatting commands
Image	Gif	Still picture in GIF format
	Jpeg	Still picture in JPEG format
Audio	Basic	Audible sound
Video	Mpeg	Movie in MPEG format
Application	Octet-stream	An uninterpreted byte sequence
	Postscript	A printable document in PostScript
Message	Rfc822	A MIME RFC 822 message
	Partial	Message has been split for transmission
	External-body	Message itself must be fetched over the net
Multipart	Mixed	Independent parts in the specified order
	Alternative	Same message in different formats
	Parallel	Parts must be viewed simultaneously
	Digest	Each part is a complete RFC 822 message

The initial MIME types and subtypes defined in RFC 2045.

Mail access protocols

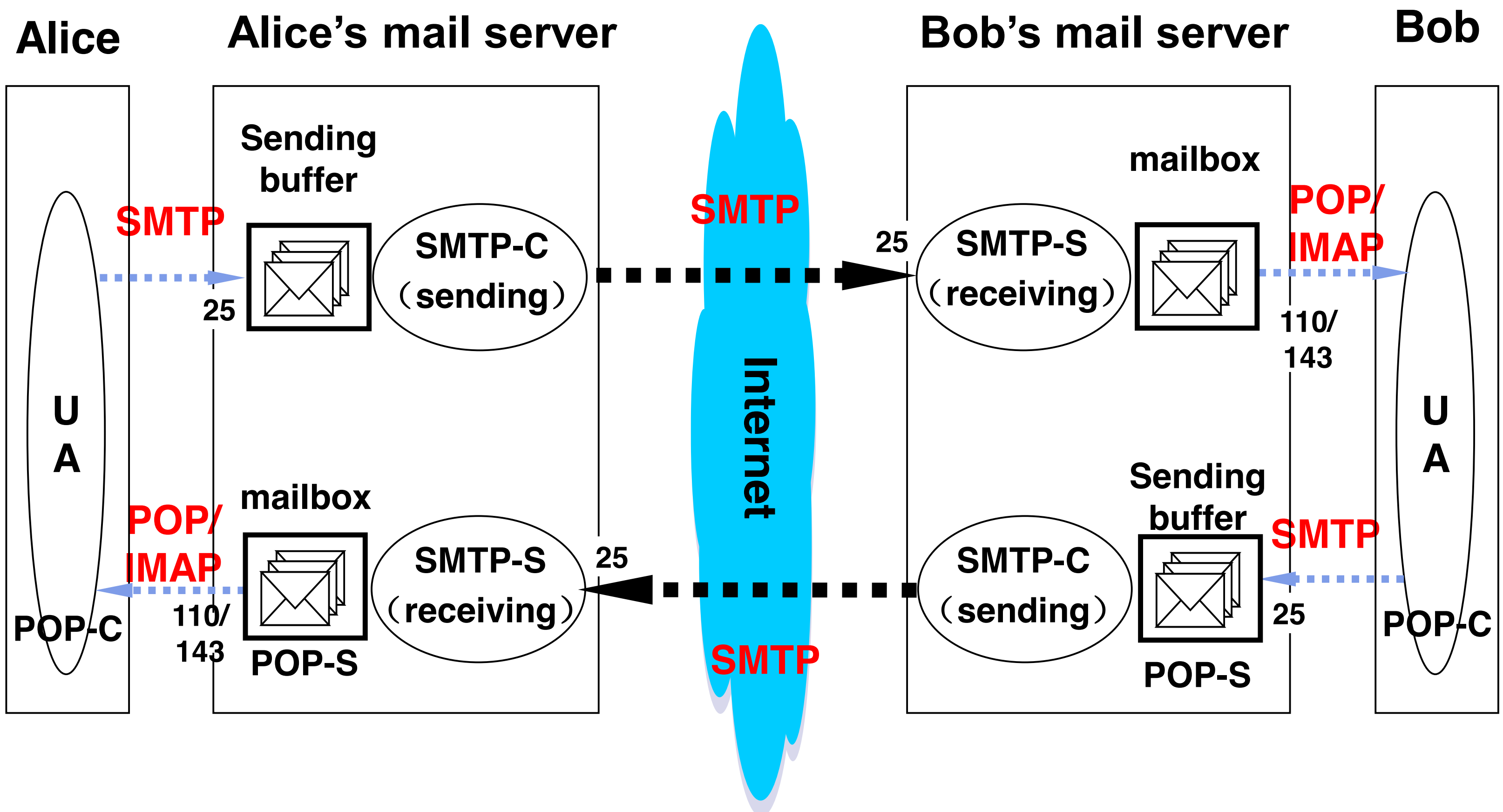


- SMTP: delivery/storage to receiver's server
- Mail access protocol: retrieval from server
  - POP: Post Office Protocol [RFC 1939]
    - download and delete
    - is stateless across sessions
  - IMAP: Internet Mail Access Protocol [RFC 1730]
    - Keep all messages at server
    - keeps user state across sessions
  - HTTP: gmail, livemail,163mail etc [RFC1945].

POP3 and IMAP

Feature	POP3	IMAP
Where is protocol defined?	RFC 1939	RFC 2060
Which TCP port is used?	110	143
Where is e-mail stored?	User's PC	Server
Where is e-mail read?	Off-line	On-line
Connect time required?	Little	Much
Use of server resources?	Minimal	Extensive
Multiple mailboxes?	No	Yes
Who backs up mailboxes?	User	ISP
Good for mobile users?	No	Yes
User control over downloading?	Little	Great
Partial message downloads?	No	Yes
Are disk quotas a problem?	No	Could be in time
Simple to implement?	Yes	No
Widespread support?	Yes	Growing

Whole Message Transfer Process

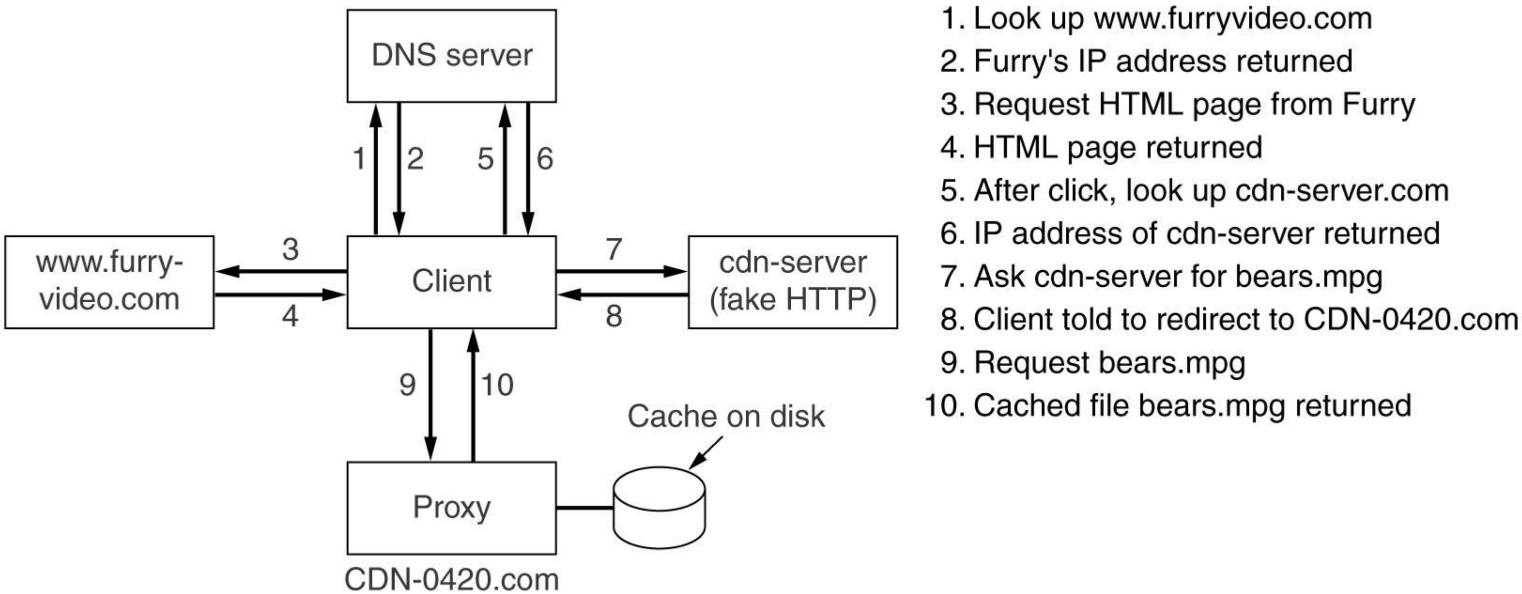


Chapter 2: Roadmap

- Principles of network applications
- DNS
- Web and HTTP
- FTP
- Electronic Mail
  - SMTP, POP3, IMAP
- P2P applications
- Socket programming with TCP
- Socket programming with UDP

Content Delivery Networks

- Replicate Web pages on a bunch of servers.
- Efficient distribution of popular content
- Faster delivery for clients



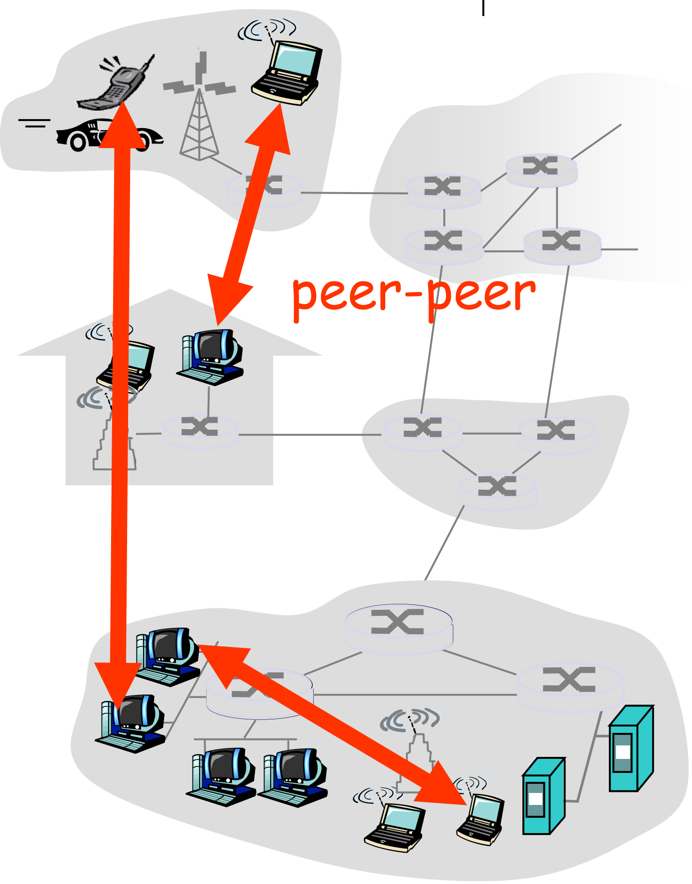
Steps in looking up *www.furry-video.com* which is a page containing references to replicated web pages (identified as *http://cdn-server.com/...*)

File Sharing and Distribution

- Delivery with client/server CDNs:
  - Efficient, scales up for popular content
  - Reliable, managed for good service
- … but some disadvantages too:
  - Need for dedicated infrastructure
  - Centralized control/oversight
  - Expensive
- P2P(Peer-to-Peer)
  - Goal is delivery without dedicated infrastructure or centralized control
  - Still efficient at scale, and reliable
  - Key idea is to have participants (or peers) help themselves

P2P (Peer-to-Peer)

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



Three topics:

- P2P architecture
- Distributed Hash Table
- Case Study: BitTorrent,Skype

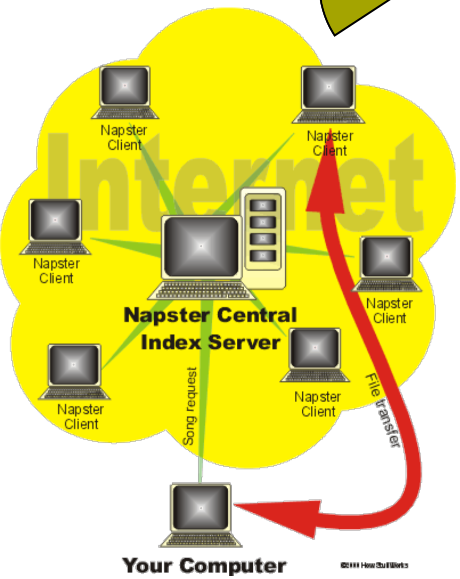
P2P Challenges

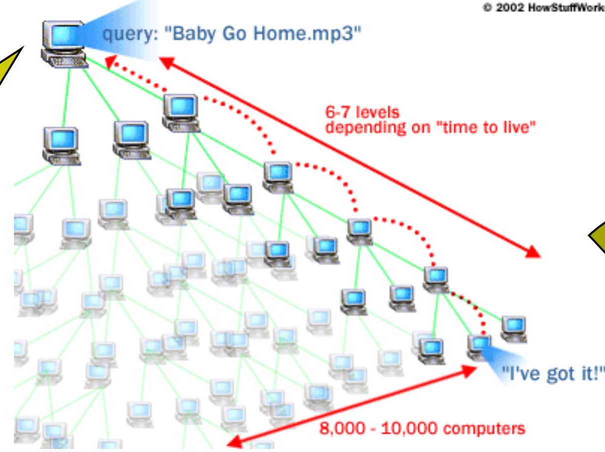
- No servers on which to rely
  - Communication must be peer-to-peer and self-organizing, not client-server
- Limited capabilities
  - How can one peer deliver content to all other peers?
- Decentralized indexing
  - How will peers find content,find each other?
- Participation incentives
  - Why will peers help each other?

P2P Architecture

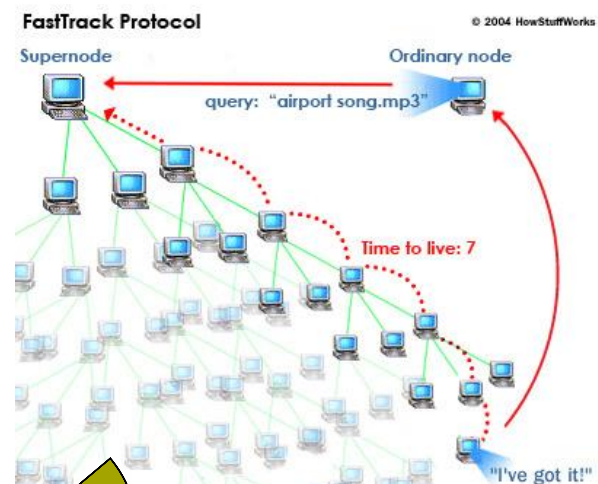
- Centralized:
  - Napster ’99 gone
- Decentralized:
  - Gnutella ‘01
- Hierarchical :
  - KaZaA ’03 -> Skype
- Distributed index:
  - DHT-based
- Semi-centralized
  - BitTorrent ’01 onwards (popular)

Napster

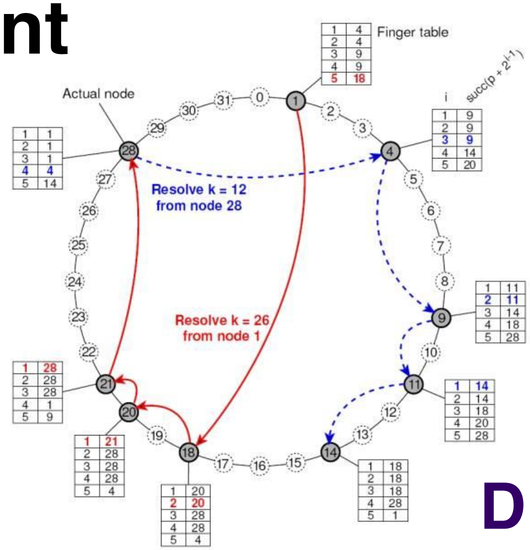




Gnutella



KaZaA



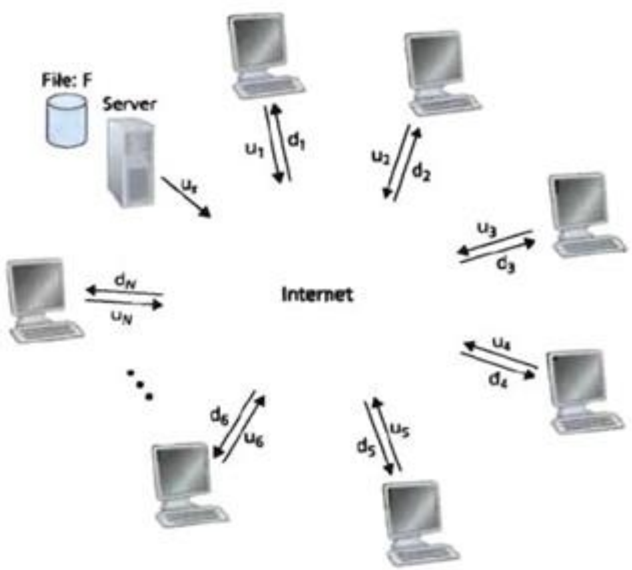
DHT

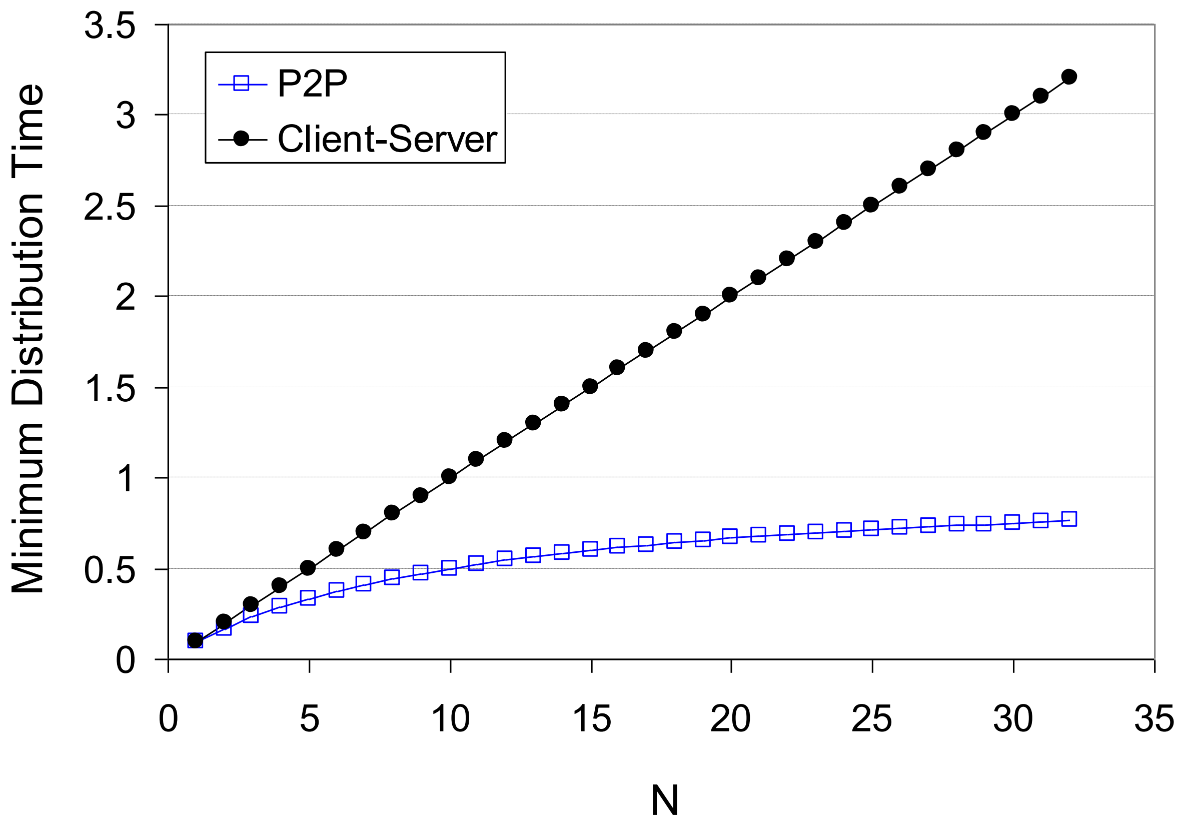
Skype, BitTorrent

2017-03-09

Scalability of P2P: example

a simple quantitative model for distributing a file to a fixed set of peers



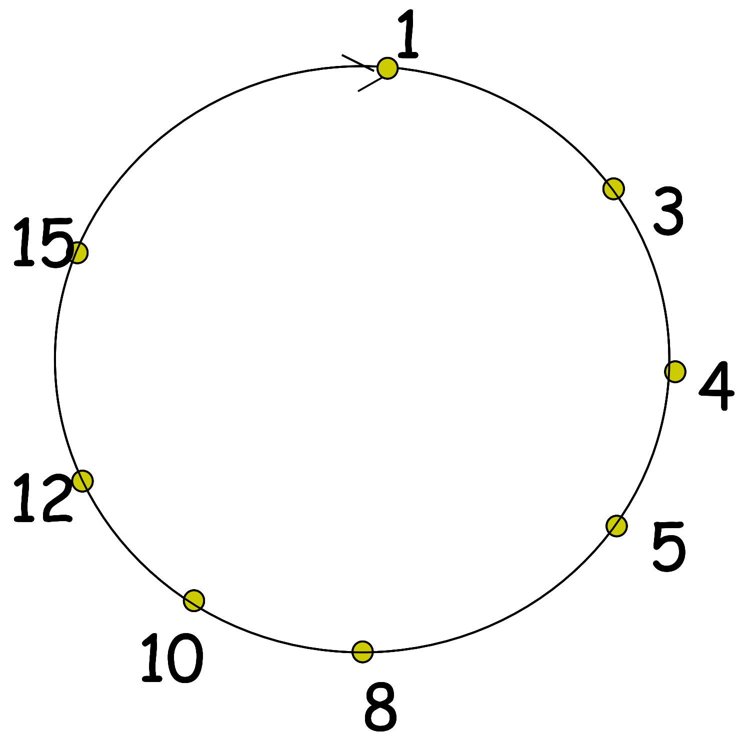


Client upload rate = u, F/u = 1 hour, u_s = 10u, $d_{min} \geq u_s$

Distributed Hash Table (DHT)

- Hash function assigns each node *and* key an n-bit *identifier* using a base hash function such as SHA-1 within the same range (2^n)
  - ID(node) = hash(IP, Port)
  - ID(key) = hash(resource name)
- The distributed index Database has (key, value) pairs;
  - key: content keywords;
  - value: IP address(es) of the host(s) with the content

DHT Overlay Network – Circular DHT



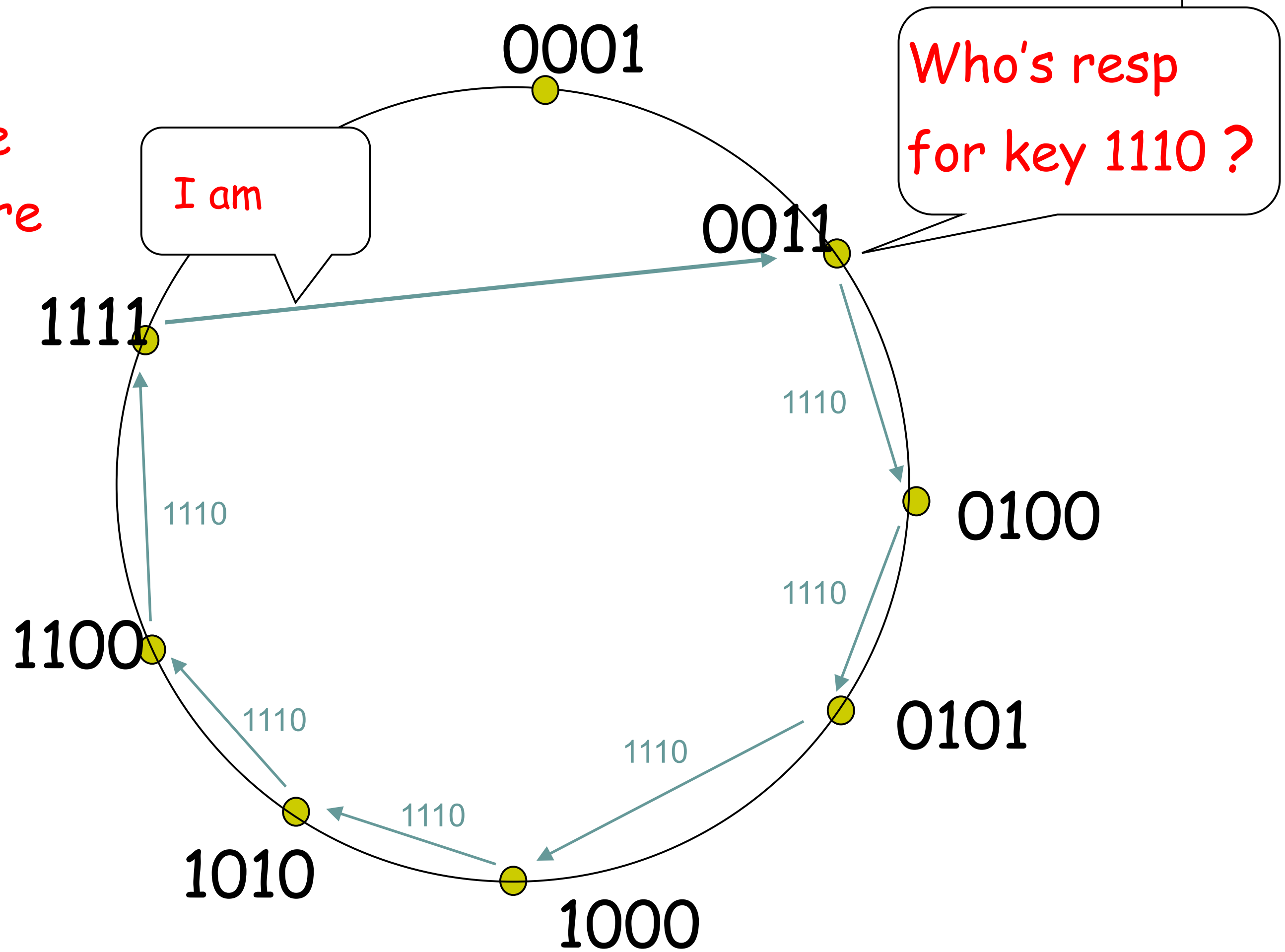
- Assign integer identifier to each peer:
  - ID(peer) = hash(IP, Port)
- Each peer *only* aware of immediate successor (actual nodes) clockwise

Key/index Location

- How to store (key, value) pairs in peers?
- Rule: store (key, value) pair to the peer that has the closest ID.
- Convention: closest is the immediate successor of the key.
- Example: peer IDs (1,3,4,5,8,10,12,14)
  - key = 13, then successor peer = 14
  - key = 15, then successor peer = 1

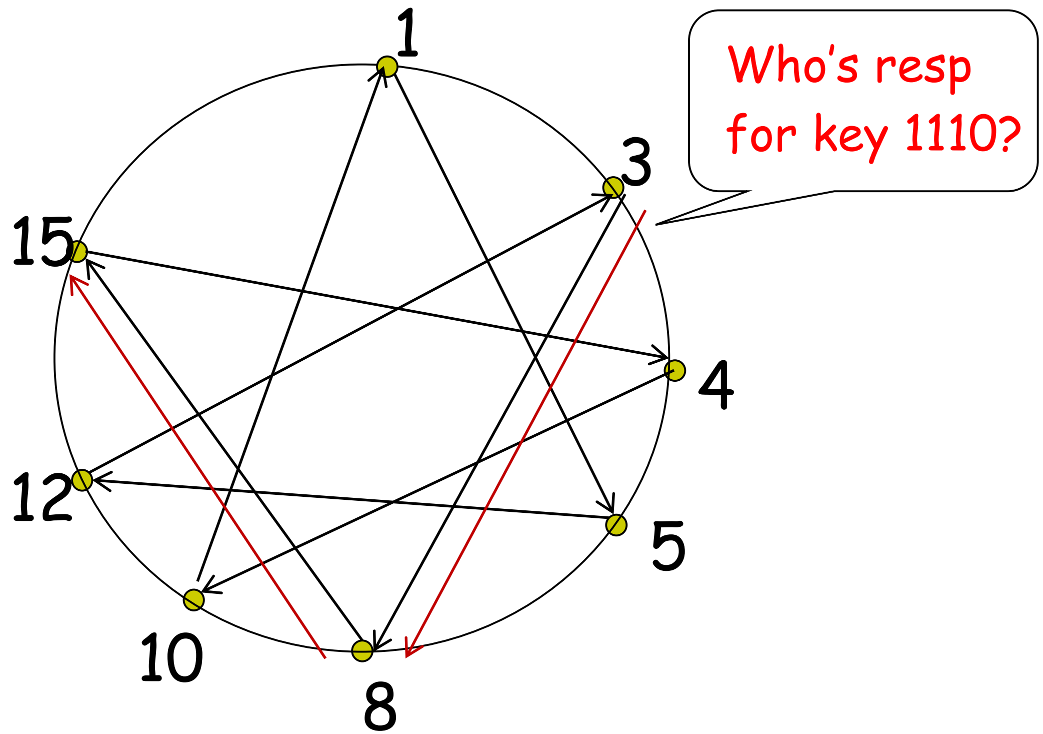
Key Lookup

O(N) messages on avg to resolve query, when there are N peers



Define closest as closest successor

Key Lookup with Shortcuts



- Each peer keeps track of IP addresses of successor and some short cuts.
- Example: reduced from 6 to 2 messages.
- Possible to design shortcuts so O(log N) neighbors, O(log N) messages in query

Acceleration of Lookups

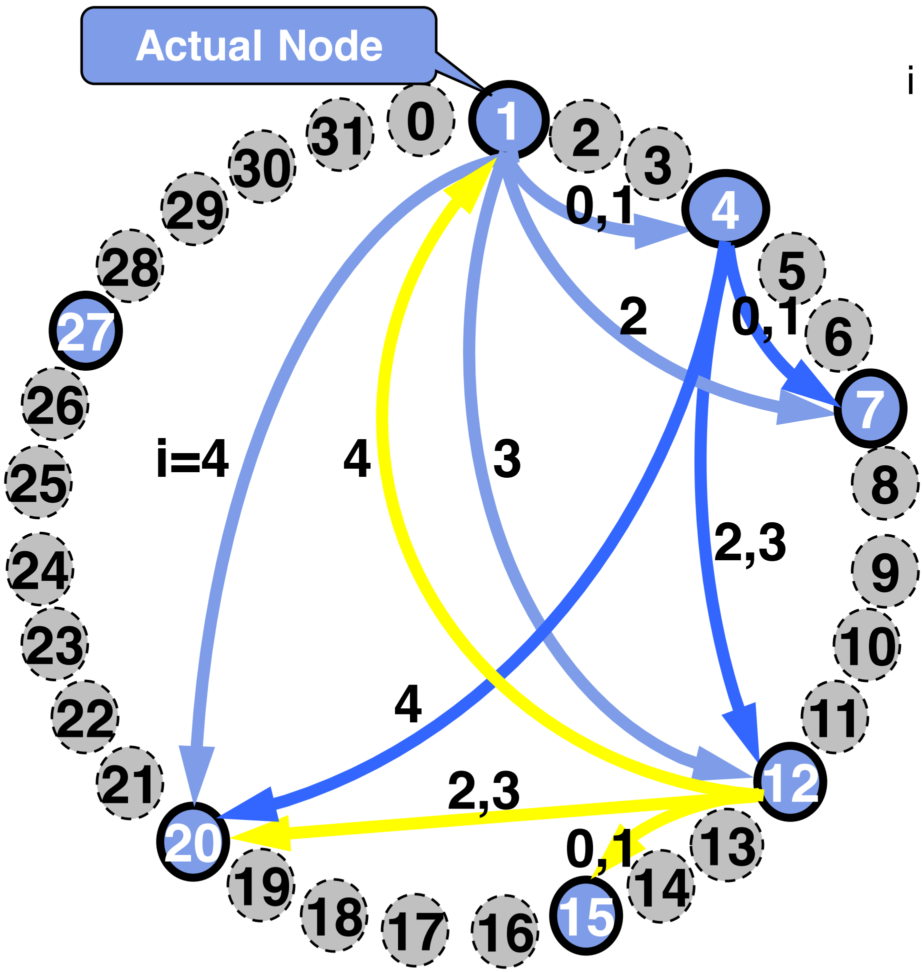
- Each node maintains a routing table with (at most) *m* entries (where $N=2^m$) called the **finger table**
- i^{th} entry in the node k's finger table :

  Start [i] = [k + 2^i] mod (2^m) [i=0…m-1], Successor (start [i])

- Lookups take O($log(N)$) hops:

  1. At Node k, if k<key <successor (k), then the node holding information about key is successor (k) and the search terminates.
  2. the finger table is searched to find the entry whose start field is the closest predecessor of key.
  3. A request is then sent directly to the S node in that finger table entry to ask it to continue the search from step 1.

DHT Finger Table Example



i		
i = 0	2	4
1	3	4
2	5	7
3	9	12
4	17	20

Noe 1

5	7
6	7
8	12
12	12
20	20

Node 4

8	12
9	12
11	12
15	15
23	27

Node 7

13	15
14	15
16	20
20	20
28	1

Node 12

i		
i = 0	16	20
1	17	20
2	19	20
3	23	27
4	31	1

Node 15

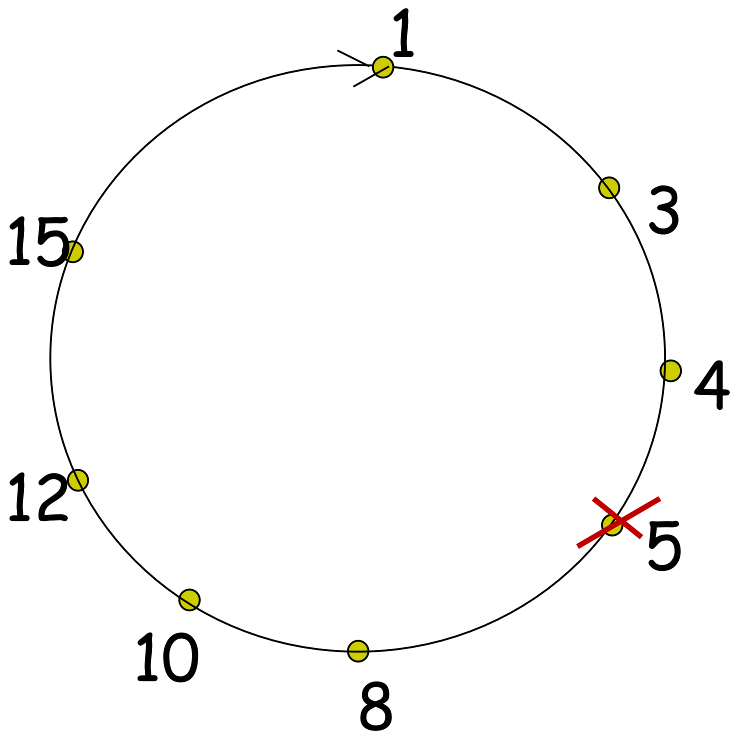
21	27
22	27
24	27
28	1
4	4

Node 20

28	1
29	1
31	1
3	4
11	12

Node 27

Peer leaves/joins

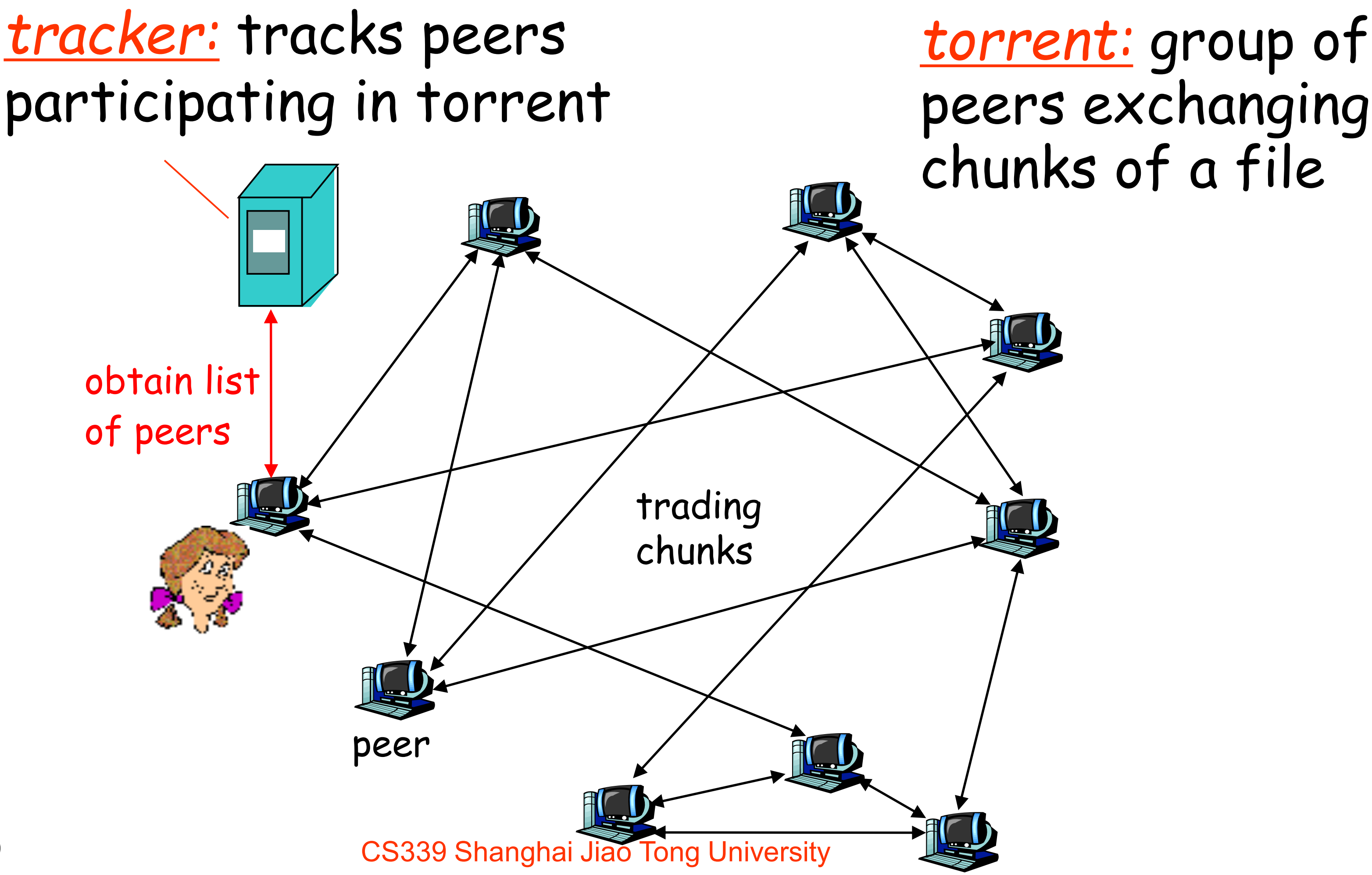


- To handle peer churn, require each peer to know the IP address of its two successors.
- Each peer periodically pings its two successors to see if they are still alive.

- Peer 5 abruptly leaves
- Peer 4 detects; makes 8 its immediate successor; asks 8 who its immediate successor is; makes 8's immediate successor its second successor.
- What if peer 6 wants to join?

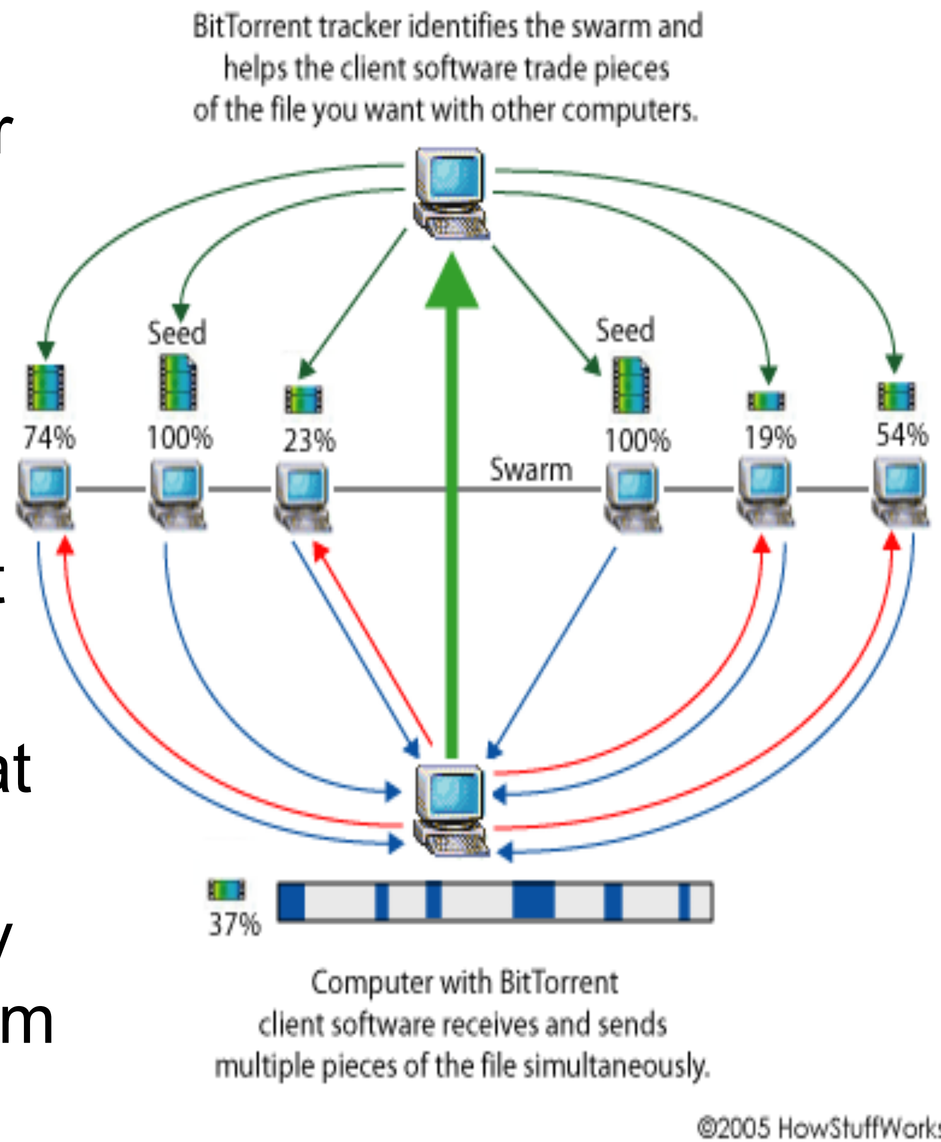
P2P Case study: BitTorrent

- P2P file distribution



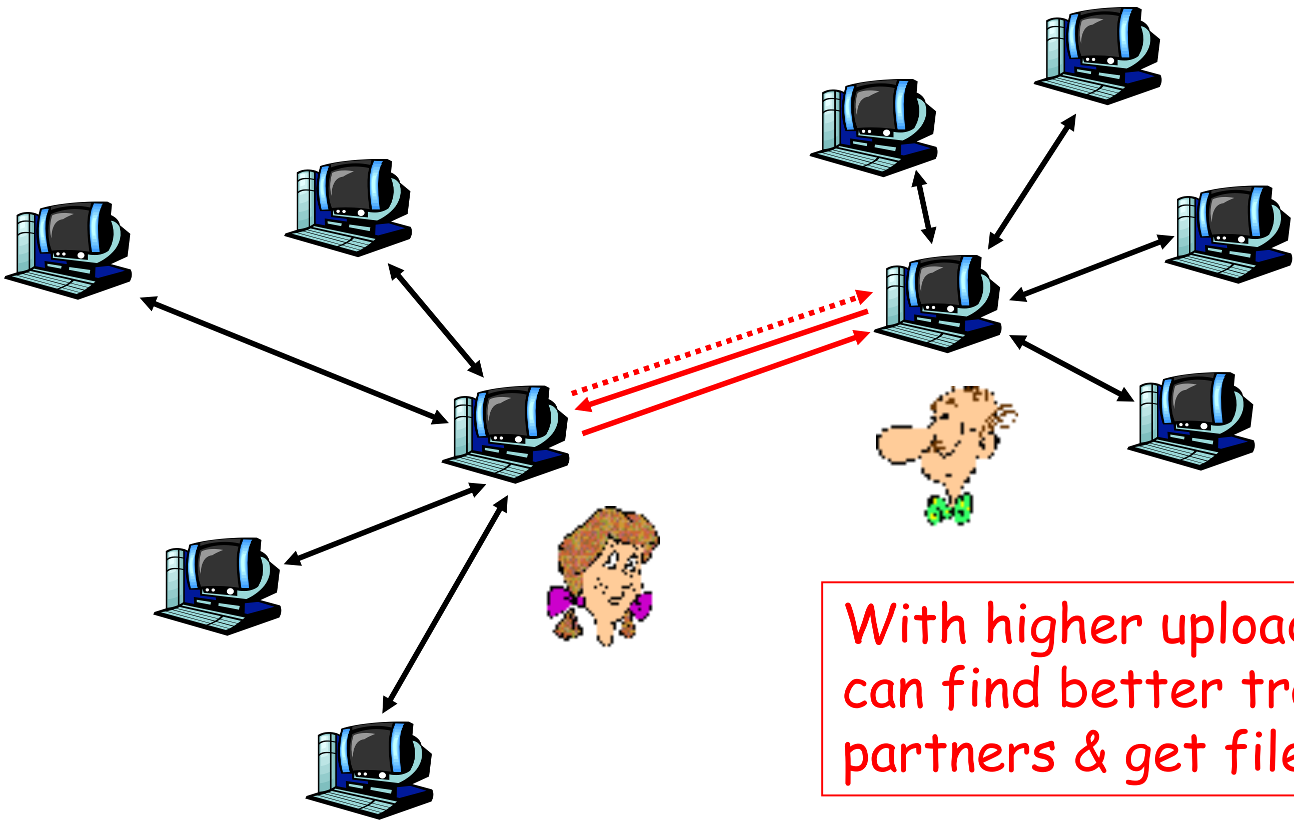
BitTorrent Protocol

- Start with click on a link for the file you want
- Contact tracker to join and get list of peers (with at least seed peer)
- Trade pieces with different peers.
- **tit-for-tat:** favor peers that upload to you rapidly; “choke” peers that don’t by slowing your upload to them



BitTorrent: Tit-for-tat

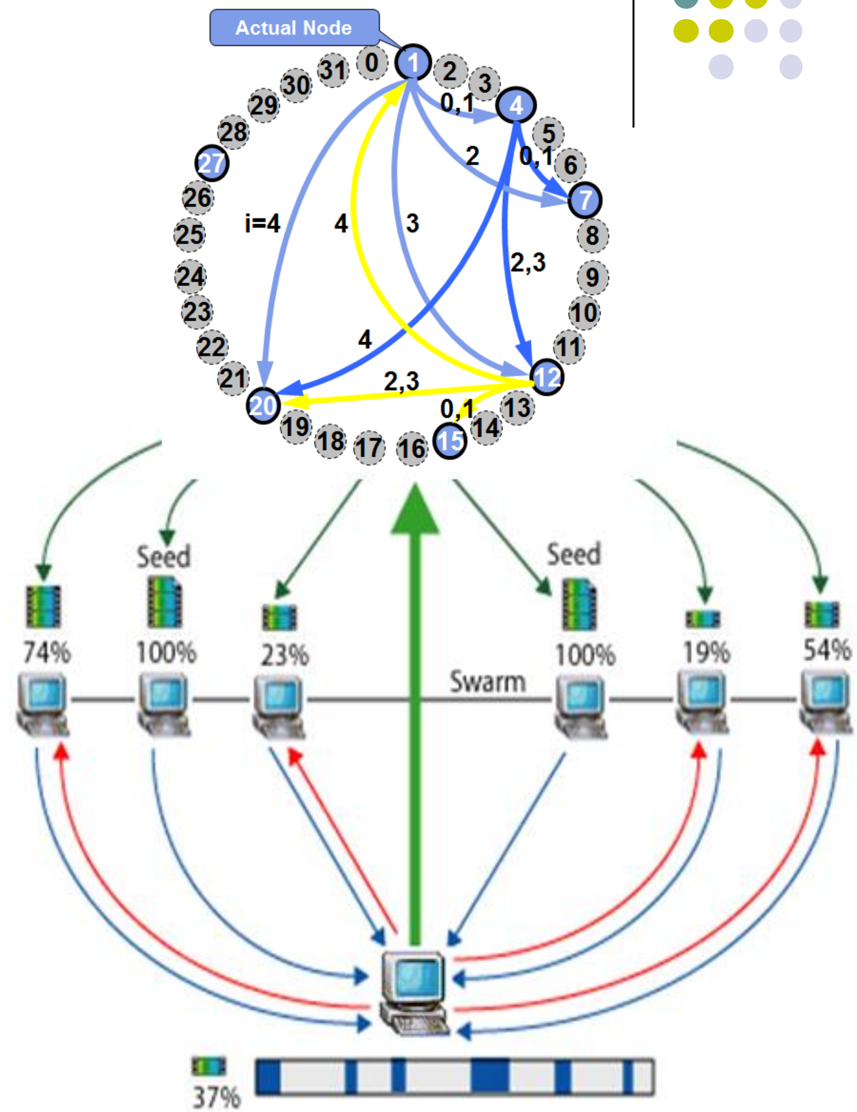
(1) Alice "optimistically unchokes" Bob
(2) Alice becomes one of Bob's **top-four** providers; Bob reciprocates
(3) Bob becomes one of Alice's top-four providers



With higher upload rate, can find better trading partners & get file faster!

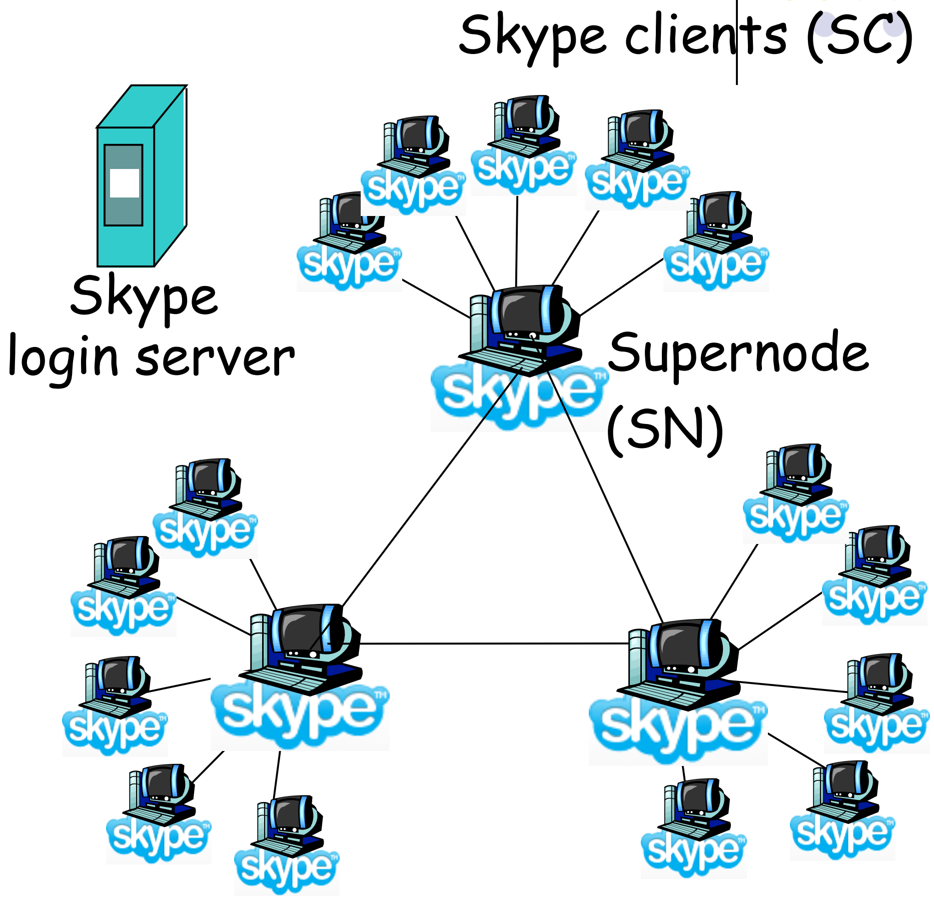
Trackerless BitTorrent

- Start with click on a link for the file you want
- Query DHT index, return all the nodes with the file (trunks)
- Trade pieces with different peers.
- Choking unhelpful peers encourages participation
- Add index with (file key, ip) to the DHT



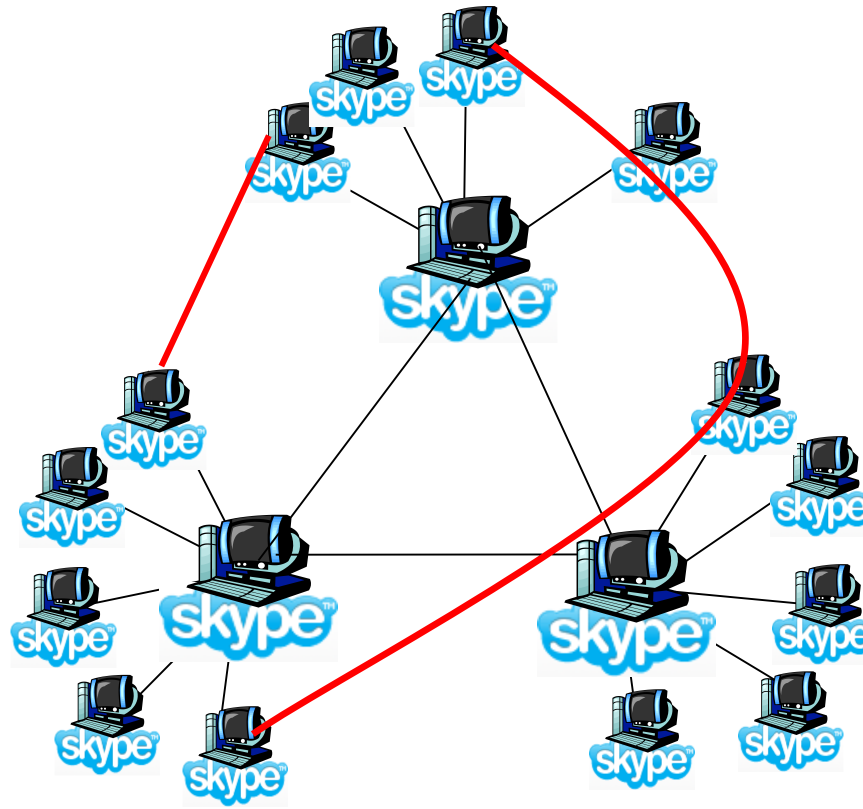
P2P Case study: Skype

- inherently P2P: pairs of users communicate.
- proprietary application-layer protocol
- hierarchical overlay with SuperNodes
- Index maps usernames to IP addresses; distributed over SNs



Skype: Peers as relays

- Problem when both Alice and Bob are behind “NATs”.
  - NAT prevents an outside peer from initiating a call to insider peer
- Solution:
  - Alice and Bob connect with their SNs.
  - Relay is chosen. Each peer initiates session with relay.
  - Peers can now communicate through NATs via relay



P2P Outlook

- Alternative to CDN-style client/server content distribution
  - With potential advantages
- P2P and DHT technologies finding more widespread use over time
  - E.g., part of skype, Amazon
  - Expect hybrid systems in the future

Chapter 2: Roadmap

- Principles of network applications
- Web and HTTP
- FTP
- Electronic Mail
  - SMTP, POP3, IMAP
- DNS
- P2P applications
- **Socket programming with TCP**
- Socket programming with UDP

Socket programming

Goal: learn how to build client/server application that communicate using sockets

Socket API

- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- client/server paradigm
- two types of transport service via socket API:
  - TCP
  - UDP

socket

a *application-created*, *OS-controlled* interface (a "door") into which application process can both send and receive messages to/from another application process

Socket programming basics

- Server must be running before client can send anything to it. not dormant
- Server must have a socket (door) through which it receives and sends segments
- Similarly client needs a socket
- Socket is locally identified with a port number
- Client needs to know server IP address and socket port number.

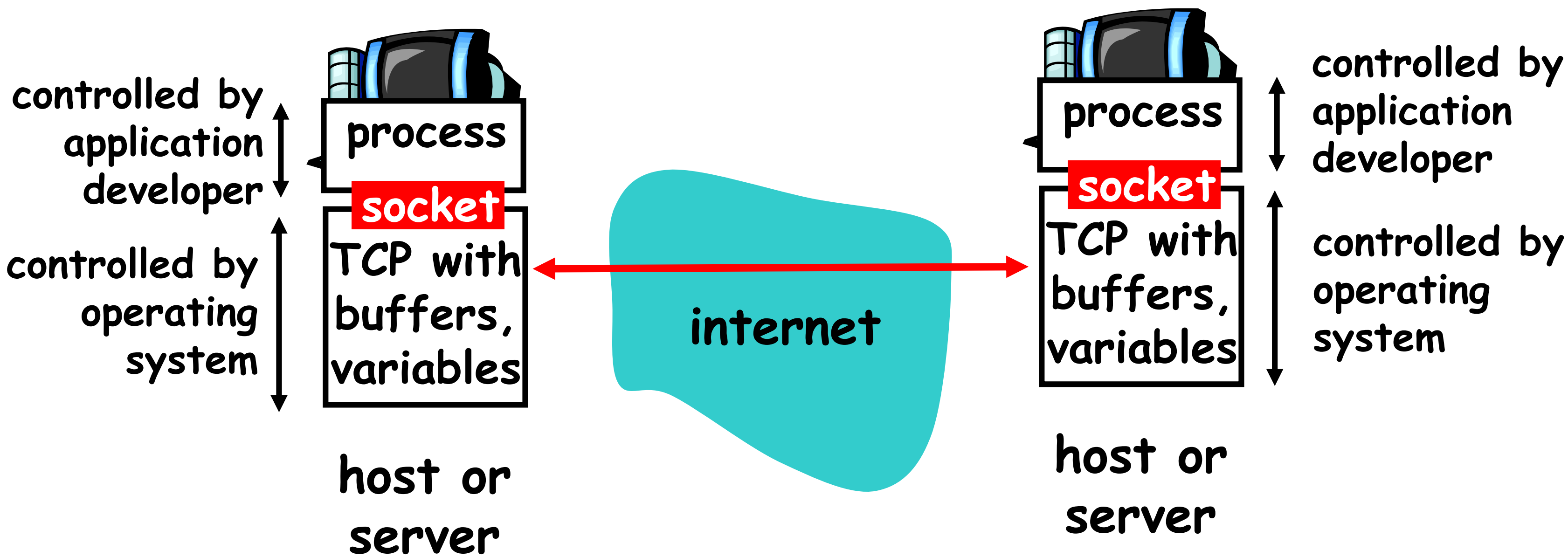
Socket API Primitives

Primitive	Meaning
SOCKET	Create a new communication endpoint
BIND	Associate a local address (port) with a socket
LISTEN	Announce willingness to accept connections
ACCEPT	Passively establish an incoming connection
CONNECT	Actively attempt to establish a connection
SEND(TO)	Send some data over the socket
RECEIVE(FROM)	Receive some data over the socket
CLOSE	Release the socket

Socket-programming using TCP

Socket: a door between application process and end-end-transport protocol (UCP or TCP)

TCP service: reliable transfer of **bytes** from one process to another



Socket programming *with TCP*

Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

Client contacts server by:

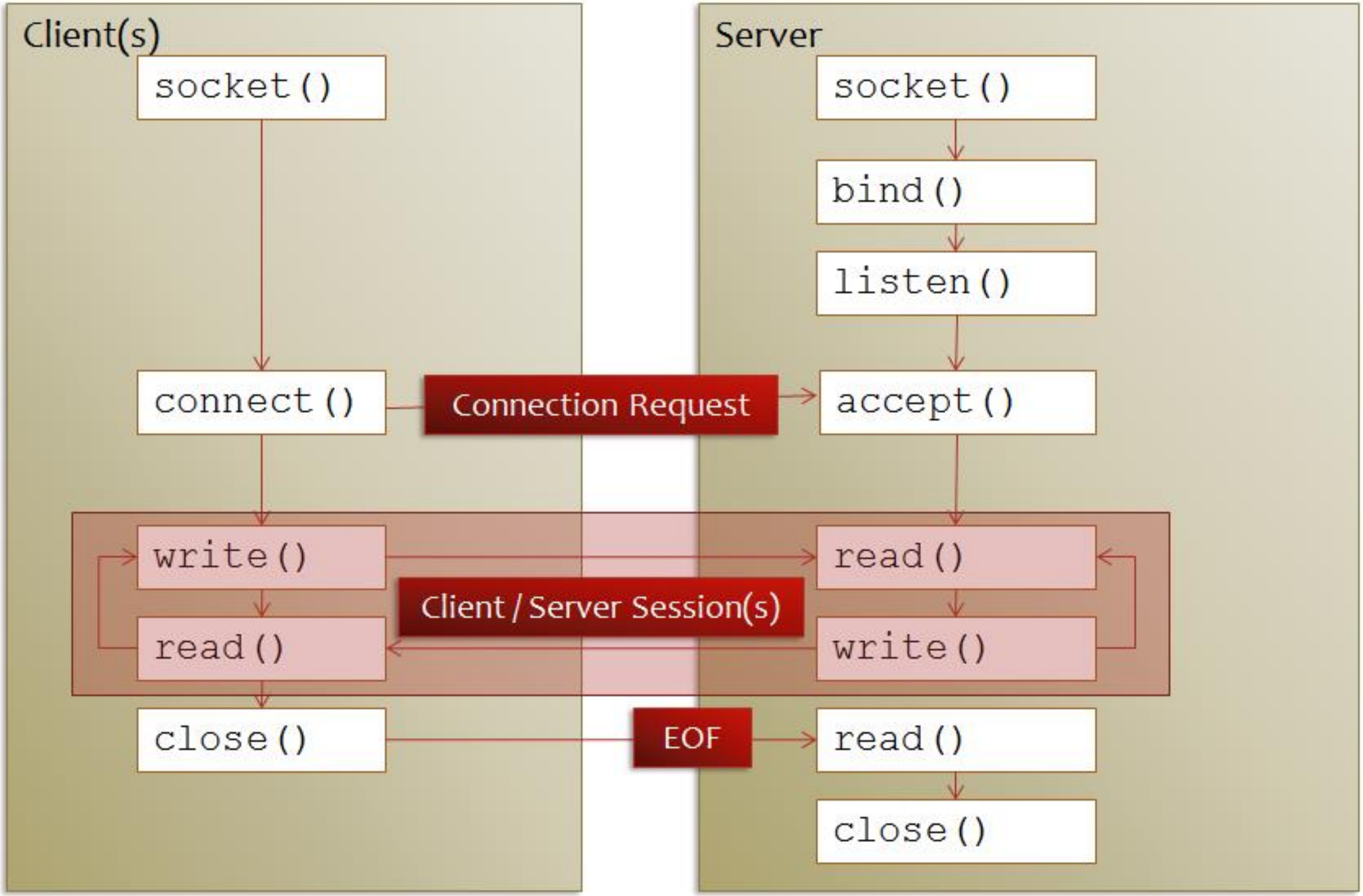
- creating client-local TCP socket
- specifying IP address, port number of server process
- When client creates socket: client TCP establishes connection to server TCP

- When contacted by client, server TCP creates new socket for server process to communicate with client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients

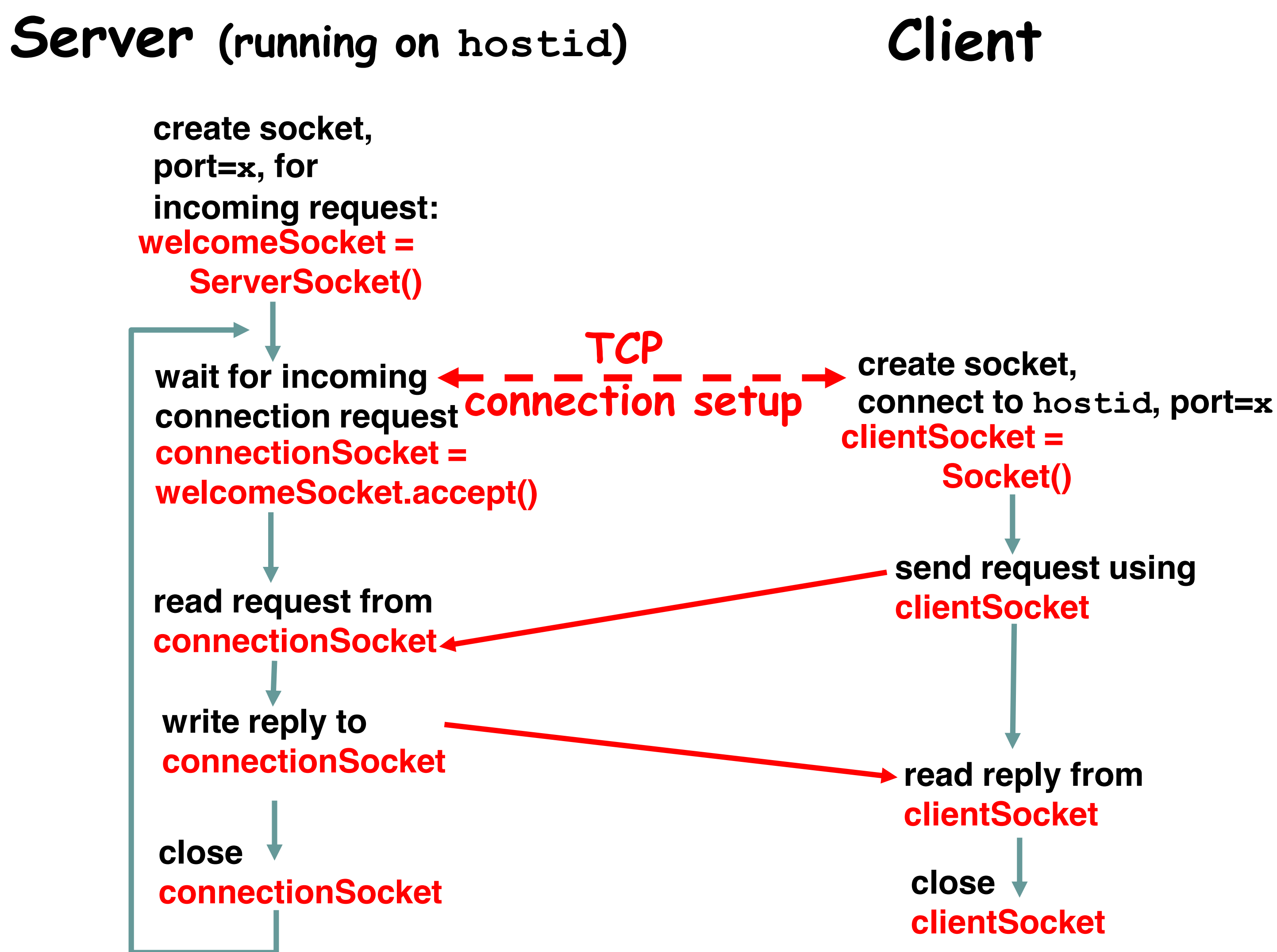
application viewpoint

TCP provides reliable, in-order transfer of bytes ("pipe") between client and server

Client/server C socket interaction: TCP



Client/server java socket interaction: TCP



Socket programming with TCP

Example client-server app:

1) client reads line from standard input (**inFromUser** stream) , sends to server via socket (**outToServer** stream)

2) server reads line from socket

3) server converts line to uppercase, sends back to client

4) client reads, prints modified line from socket (**inFromServer** stream)

Example: Java client (TCP)

```
import java.io.*;
import java.net.*;
class TCPClient {

    public static void main(String argv[]) throws Exception
    {
        String sentence;
        String modifiedSentence;

        BufferedReader inFromUser =
          new BufferedReader(new InputStreamReader(System.in)

        String hostname=argv[0];
        Socket clientSocket = new Socket(hostname, 6789);

        DataOutputStream outToServer =
          new DataOutputStream(clientSocket.getOutputStream())
```

Create input stream

Create client socket, connect to server

Create output stream attached to socket

Example: Java client (TCP), cont.

Create input stream attached to socket

```
BufferedReader inFromServer =
  new BufferedReader(new
  InputStreamReader(clientSocket.getInputStream()));
```

```
sentence = inFromUser.readLine();
```

Send line to server

```
outToServer.writeBytes(sentence + '\n');
```

Read line from server

```
modifiedSentence = inFromServer.readLine();

System.out.println("FROM SERVER: " + modifiedSentence);

clientSocket.close();

    }
}
```

Example: Java server (TCP)

```
import java.io.*;
import java.net.*;

class TCPServer {

  public static void main(String argv[]) throws Exception
   {
     String clientSentence;
     String capitalizedSentence;

     ServerSocket welcomeSocket = new ServerSocket(6789);

     while(true) {

         Socket connectionSocket = welcomeSocket.accept();

        BufferedReader inFromClient =
          new BufferedReader(new
          InputStreamReader(connectionSocket.getInputStream()));
```

Create welcoming socket at port 6789

Wait, on welcoming socket for contact by client

Create input stream, attached to socket

Example: Java server (TCP), cont

Create output stream, attached to socket

```
DataOutputStream  outToClient =
  new DataOutputStream(connectionSocket.getOutputStream());
```

Read in line from socket

```
clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '\n';
```

Write out line to socket

```
outToClient.writeBytes(capitalizedSentence);
      }
    }
  }
```

End of while loop, loop back and wait for another client connection

TCP observations & questions

- Server has two types of sockets:
  - ServerSocket and Socket
- When client knocks on serverSocket's "door," server creates connectionSocket and completes TCP conx.
- Dest IP and port are not explicitly attached to segment.
- Can multiple clients use the server?

Chapter 2: Roadmap

- Principles of network applications
- Web and HTTP
- FTP
- Electronic Mail
  - SMTP, POP3, IMAP
- DNS
- P2P applications
- Socket programming with TCP
- **Socket programming with UDP**

Socket programming *with UDP*

UDP: no "connection" between client and server

- no handshaking
- sender explicitly attaches IP address and port of destination to each packet
- server must extract IP address, port of sender from received packet

UDP: transmitted data may be received out of order, or lost

application viewpoint

UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server

Note: **the official terminology for a UDP packet is "datagram". In this class, we instead use "UDP segment".**

Client/server socket interaction: UDP

Server (running on `hostid`)

create socket,
port= x.
serverSocket =
DatagramSocket()

read datagram from
serverSocket

write reply to
serverSocket
specifying
client address,
port number

Client

create socket,
clientSocket =
DatagramSocket()

Create datagram with server IP and
port=x; send datagram via
clientSocket

read datagram from
clientSocket

close
clientSocket

Example: Java client (UDP)

```
import java.io.*;
import java.net.*;

class UDPClient {
    public static void main(String args[]) throws Exception
    {

      BufferedReader inFromUser =
        new BufferedReader(new InputStreamReader(System.in));

      DatagramSocket clientSocket = new DatagramSocket();

      String hostname=argv[0];
      InetAddress IPAddress = InetAddress.getByName("hostname");

      byte[] sendData = new byte[1024];
      byte[] receiveData = new byte[1024];

      String sentence = inFromUser.readLine();

      sendData = sentence.getBytes();
```

Create input stream

Create client socket

Translate hostname to IP address using DNS

Example: Java client (UDP), cont.

Create datagram with data-to-send, length, IP addr, port

```
DatagramPacket sendPacket =
  new DatagramPacket(sendData, sendData.length, IPAddress, 9876
```

Send datagram to server

```
clientSocket.send(sendPacket);
```

```
DatagramPacket receivePacket =
  new DatagramPacket(receiveData, receiveData.length);
```

Read datagram from server

```
clientSocket.receive(receivePacket);

String modifiedSentence =
   new String(receivePacket.getData());

System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
}

}
```

Example: Java server (UDP)

```
import java.io.*;
import java.net.*;

class UDPServer {
  public static void main(String args[]) throws Exception
   {

     DatagramSocket serverSocket = new DatagramSocket(9876);

     byte[] receiveData = new byte[1024];
     byte[] sendData  = new byte[1024];

     while(true)
       {

         DatagramPacket receivePacket =
            new DatagramPacket(receiveData, receiveData.length);
          serverSocket.receive(receivePacket);
```

Create datagram socket at port 9876

Create space for received datagram

Receive datagram

Example: Java server (UDP), cont

Get IP addr port #, of sender

Create datagram to send to client

Write out datagram to socket

```
String sentence = new String(receivePacket.getData());

InetAddress IPAddress = receivePacket.getAddress();

int port = receivePacket.getPort();

String capitalizedSentence = sentence.toUpperCase();

sendData = capitalizedSentence.getBytes();

DatagramPacket sendPacket =
   new DatagramPacket(sendData, sendData.length, IPAddress,
                      port);

serverSocket.send(sendPacket);
      }
    }
  }
```

End of while loop, loop back and wait for another datagram

UDP observations & questions

- Both client server use DatagramSocket
- Dest IP and port are explicitly attached to segment.
- Can the client send a segment to server without knowing the server's IP address and/or port number?
- Can multiple clients use the server?

The struct sockaddr

- The generic:

```
struct sockaddr {
    u_short sa_family;
    char sa_data[14];
};
```

  - sa_family
    - specifies which address family is being used
    - determines how the remaining 14 bytes are used

- The Internet-specific:

```
struct sockaddr_in {
    short sin_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
};
```

  - sin_family = AF_INET
  - sin_port: port # (0-65535)
  - sin_addr: IP-address
  - sin_zero: unused

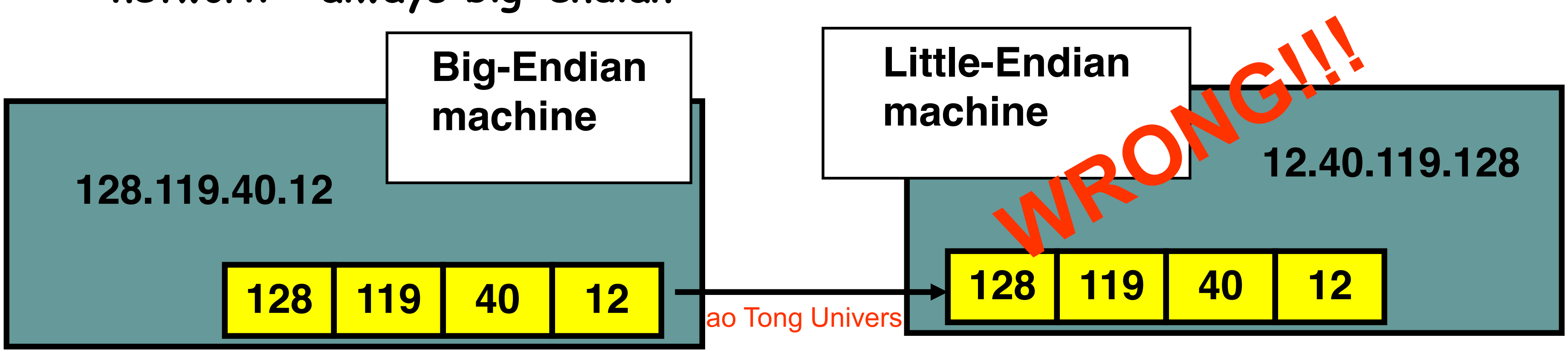
```
struct in_addr {
  u_long s_addr;
};
```

Address and port byte-ordering

- Address and port are stored as integers

```
u_short sin_port; (16 bit)
in_addr sin_addr; (32 bit)
```

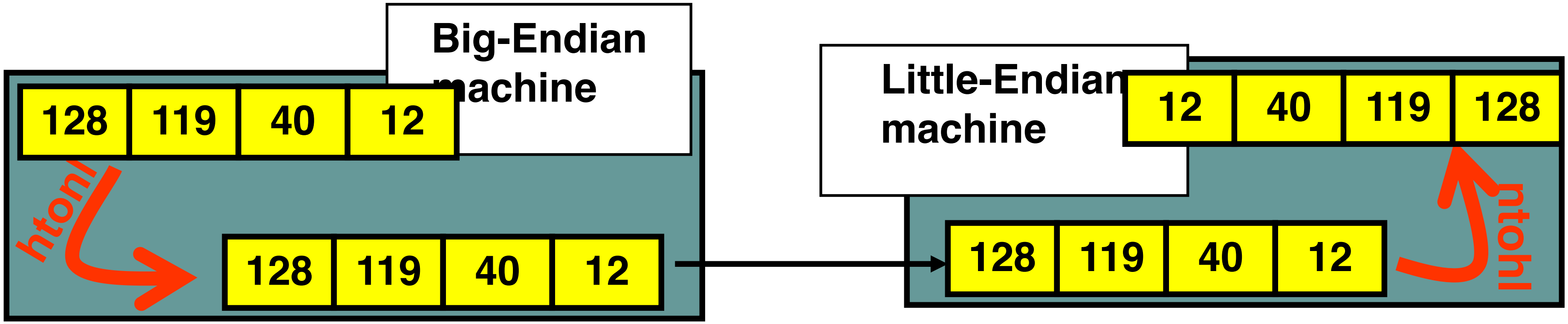
- **Problem:**
  - different machines / OS's use different word orderings
    - little-endian: lower bytes first
    - big-endian: higher bytes first
  - these machines may communicate with one another over the network
- **Solution:** Network Byte-Ordering: the byte ordering used by the network – always big-endian



UNIX's byte-ordering funcs

- u_long htonl(u_long x);
- u_short htons(u_short x);
- u_long ntohl(u_long x);
- u_short ntohs(u_short x);

- On big-endian machines, these routines do nothing
- On little-endian machines, they reverse the byte order



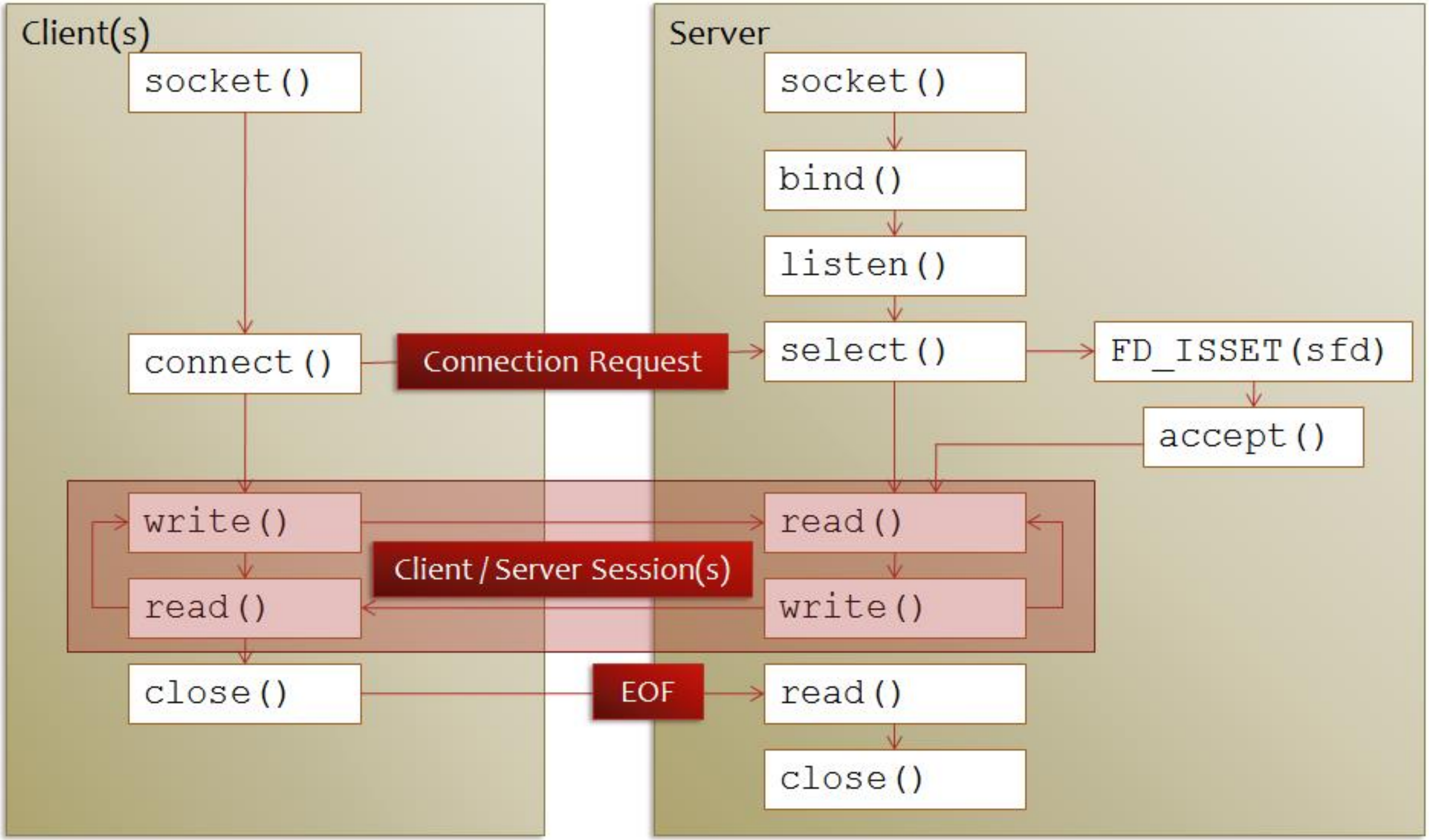
- Same code would have worked regardless of endian-ness of the two machines

Dealing with blocking calls

- Many of the functions we saw block until a certain event
  - accept: until a connection comes in
  - connect: until the connection is established
  - recv, recvfrom: until a packet (of data) is received
  - send, sendto: until data is pushed into socket's buffer
    - Q: why not until received?
- For simple programs, blocking is convenient
- What about more complex programs?
  - multiple connections
  - simultaneous sends and receives
  - simultaneously doing non-networking processing

How did we add concurrency?

- Processes
  - Uses fork()
  - Easy to understand
  - A lot to consider about causing complexity
- Threads
  - Natural concurrency (new thread per connection)
  - Easier to understand
  - Complexity is increased (possible race conditions)
- Use non-blocking I/O
  - Uses select()
  - Explicit control flow (no race conditions!)
  - Explicit control flow more complicated though


Client(s)
socket()
connect()
Connection Request
write()
read()
Client / Server Session(s)
close()
EOF
Server
socket()
bind()
listen()
select()
FD_ISSET(sfd)
accept()
read()
write()
read()
close()

Other Socket API Functions

Action	BSD
Conversion from text address to packed address	inet_aton
Conversion from packed address to text address	inet_ntoa
Forward lookup for host name/service	gethostbyname, gethostbyaddr, getservbyname, getservbyport
Reverse lookup for host name/service	gethostbyaddr, getservbyport

Chapter 2: Summary

our study of network apps now complete!

- application architectures
  - client-server
  - P2P
  - hybrid
- application service requirements:
  - reliability, bandwidth, delay
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- **specific protocols:**
  - **HTTP**
  - **FTP**
  - **SMTP, POP, IMAP**
  - **DNS**
  - **P2P: BitTorrent, Skype**
- **socket programming**

Chapter 2: Summary

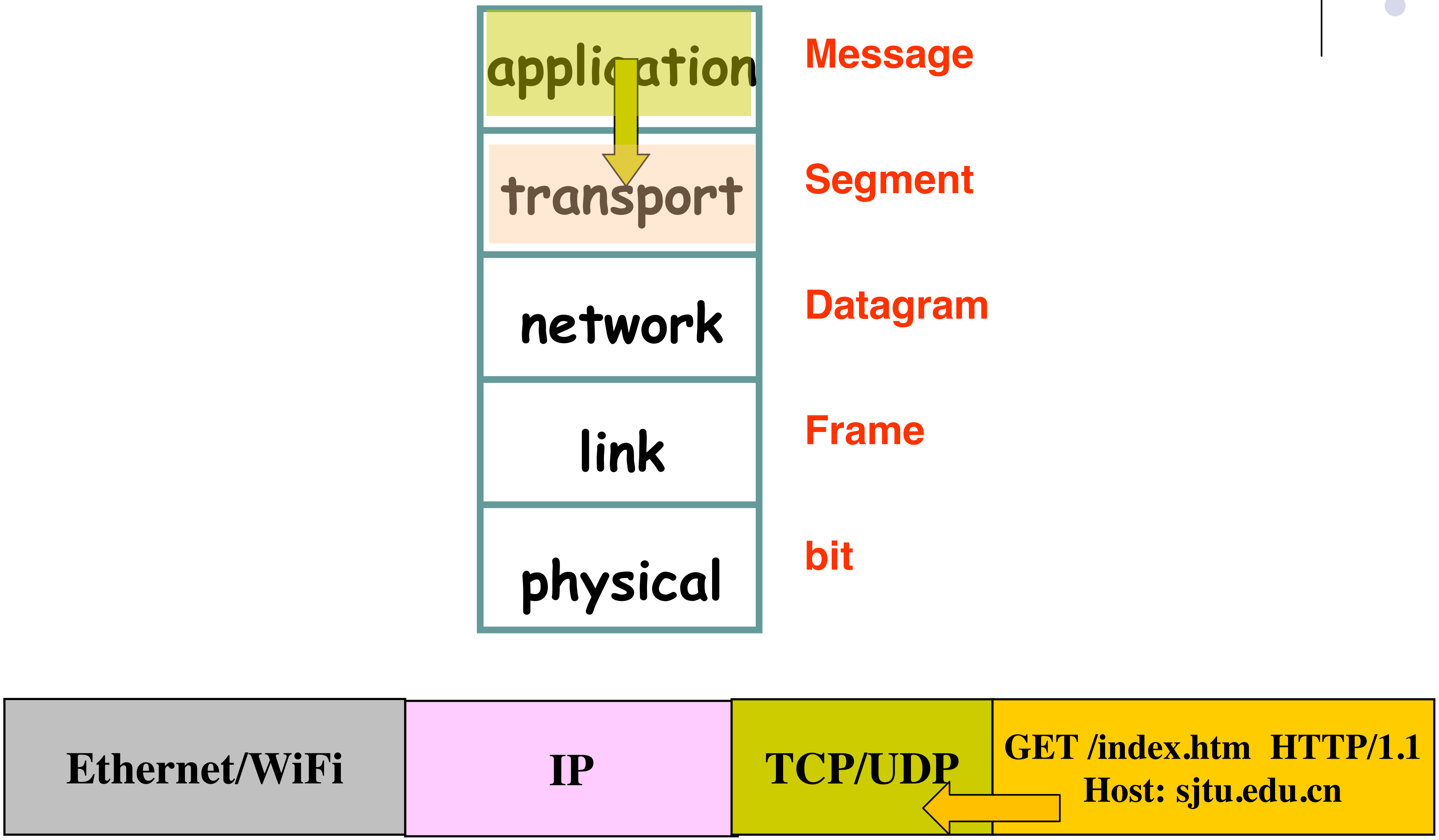
<u>Most importantly:</u> learned about *protocols*

- typical request/reply message exchange:
  - client requests info or service
  - server responds with data, status code
- message formats:
  - headers: fields giving info about data
  - data: info being communicated

Important themes:

- **control vs. data msgs**
  - **in-band, out-of-band**
- **centralized vs. decentralized**
- **stateless vs. stateful**
- **reliable vs. unreliable msg transfer**
- **"complexity at network edge"**

Chapter 2: summary



Assignment2

- Watch MOOC video online, reference to chapter2 of textbook and ppts, use WireShark to observe the packets about protocols of DNS, HTTP, FTP, SMTP, BitTorrent. Answer following questions for each of DNS, HTTP, FTP, SMTP, BitTorrent:
  - <u>Is it reliable or not? connection oriented or not? use TCP or UDP?</u>
  - What transport service does the app need?
  - Why is there a UDP?
  - <u>What is the interaction model of the protocol: Client/Server or P2P, and how?</u>
  - <u>What is the message format and semantics?</u>
- next class:
  - Discussion in groups
  - Finish the quiz each group
  - Group presentation