CS 3251- Computer Networking 1: P2P and Sockets

Professor Patrick Traynor

Lecture 06

9/5/2013

Announcements

- Assignment 1 is due 9/12/2013 (next Thursday)
  - Where are we?
  - What sorts of problems are we having?

Recap

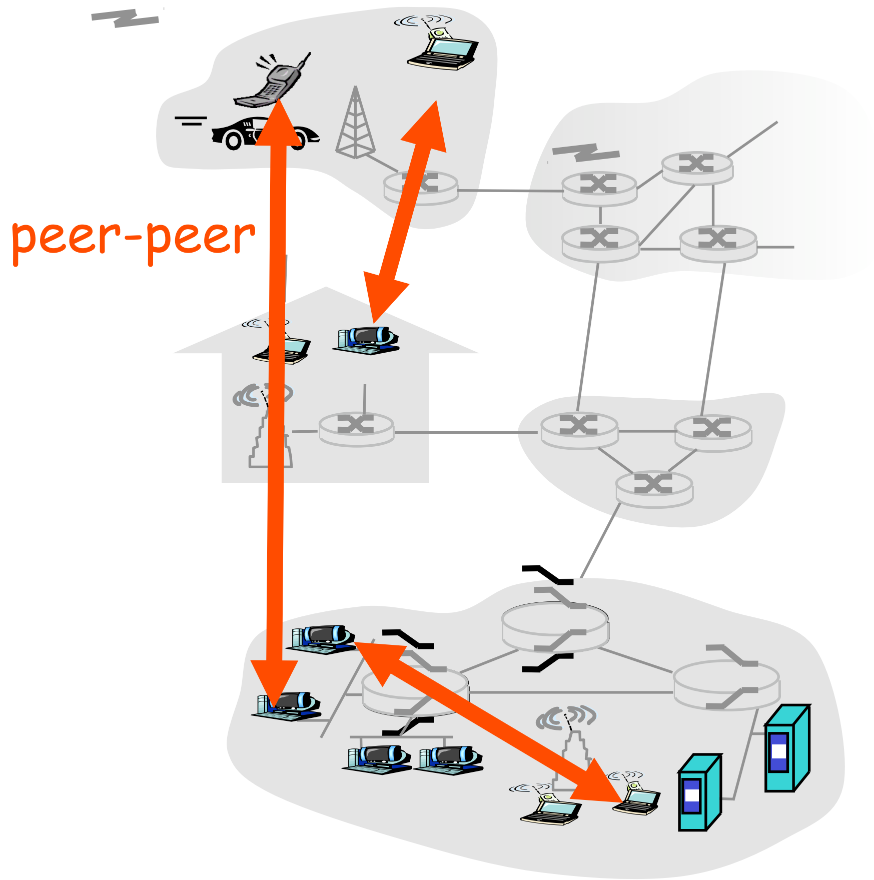
- SMTP is the language that mail servers use to exchange messages.
  - SMTP is push-based... why?
  - You can run SMTP from a telnet window. Anything interesting here?
- DNS translates between names and IP addresses.
  - Returns NS, MX, CNAME and A records.
  - Never designed to be secure - and we're beginning to pay for that.

Chapter 2: Application Layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
- 2.5 DNS
- 2.6 P2P Applications

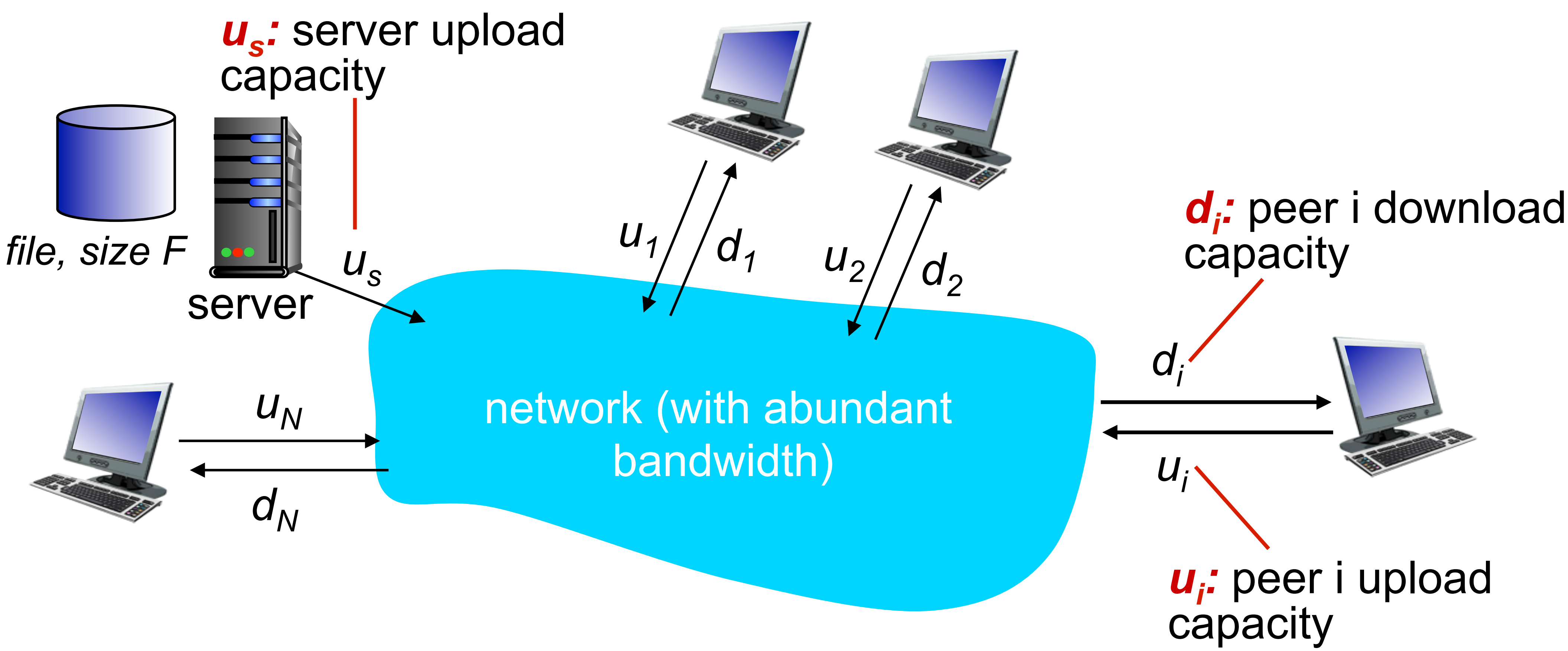
Pure P2P Architecture

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



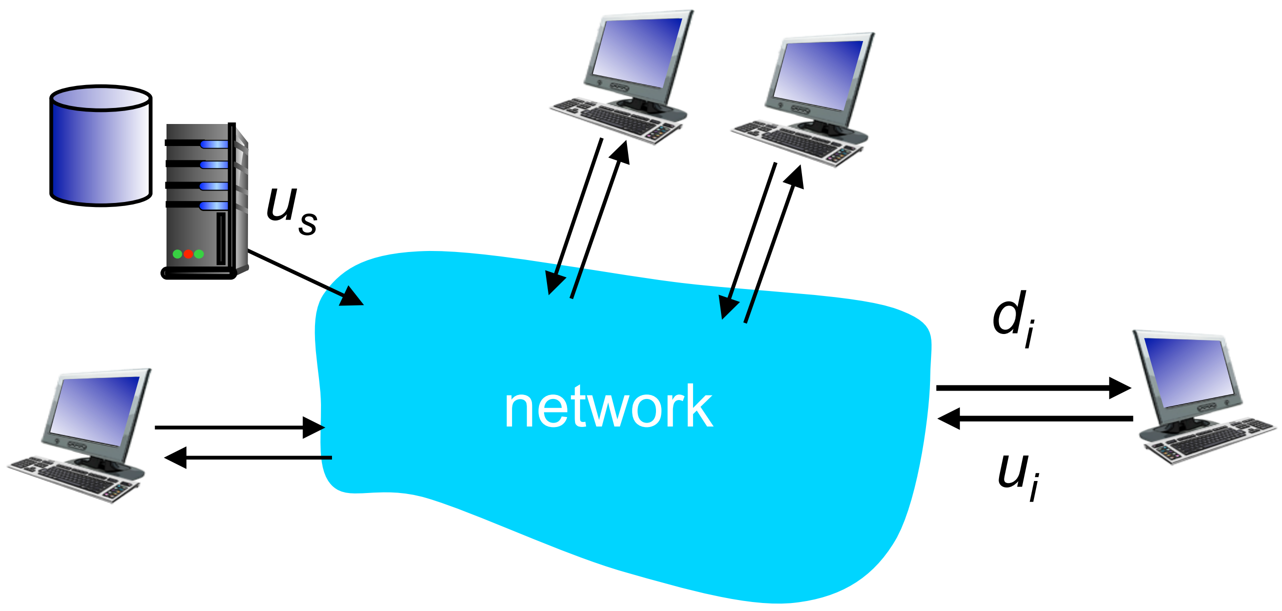
File Distribution: Server-Client vs P2P

Question: How much time does it take to distribute a file from one server to N peers?



File Distribution Time: Server-Client

- *server transmission:* must sequentially send (upload) N file copies:
  - time to send one copy: F/u_s
  - time to send N copies: NF/u_s
- *client*: each client must download file copy
  - d_{min} = min client download rate
  - min client download time: F/d_{min}

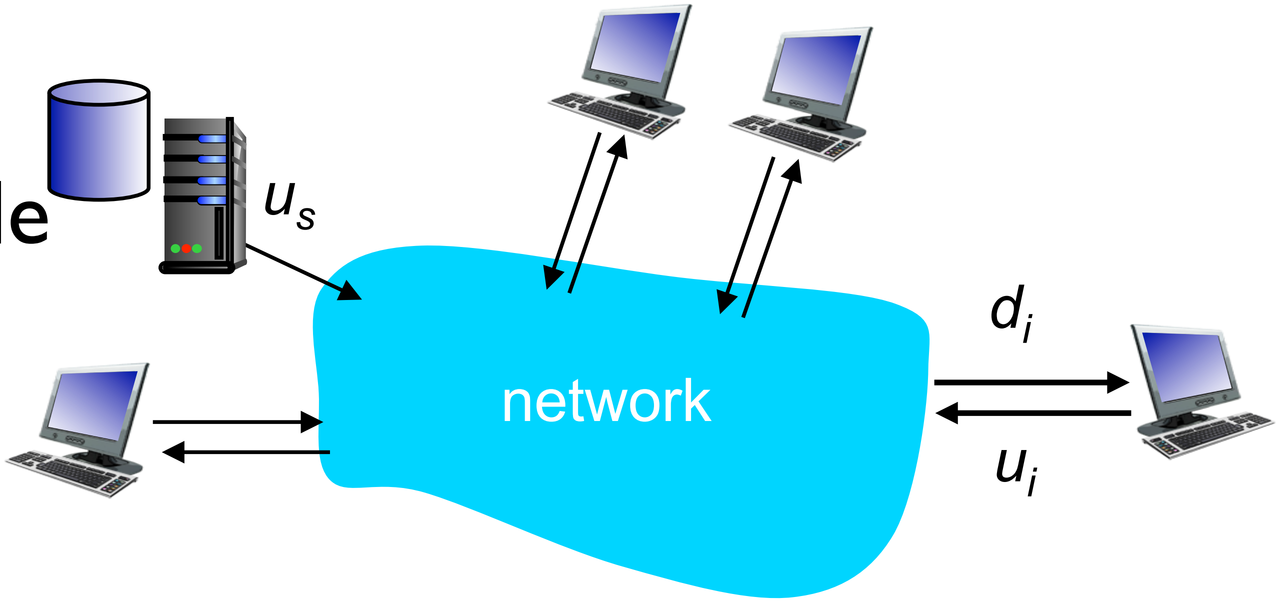


increases linearly in N (for large N)

time to distribute F to N clients using client-server approach $D_{c\text{-}s} > max\{NF/u_s, F/d_{min}\}$

File Distribution Time: P2P

- *server transmission:* must upload at least one copy:
  - time to send one copy: F/u_s
- *client:* each client must download file copy
  - min client download time: F/d_{min}
- *clients:* NF bits must be downloaded (aggregate)
  - fastest possible upload rate: $u_s + \Sigma u_i$



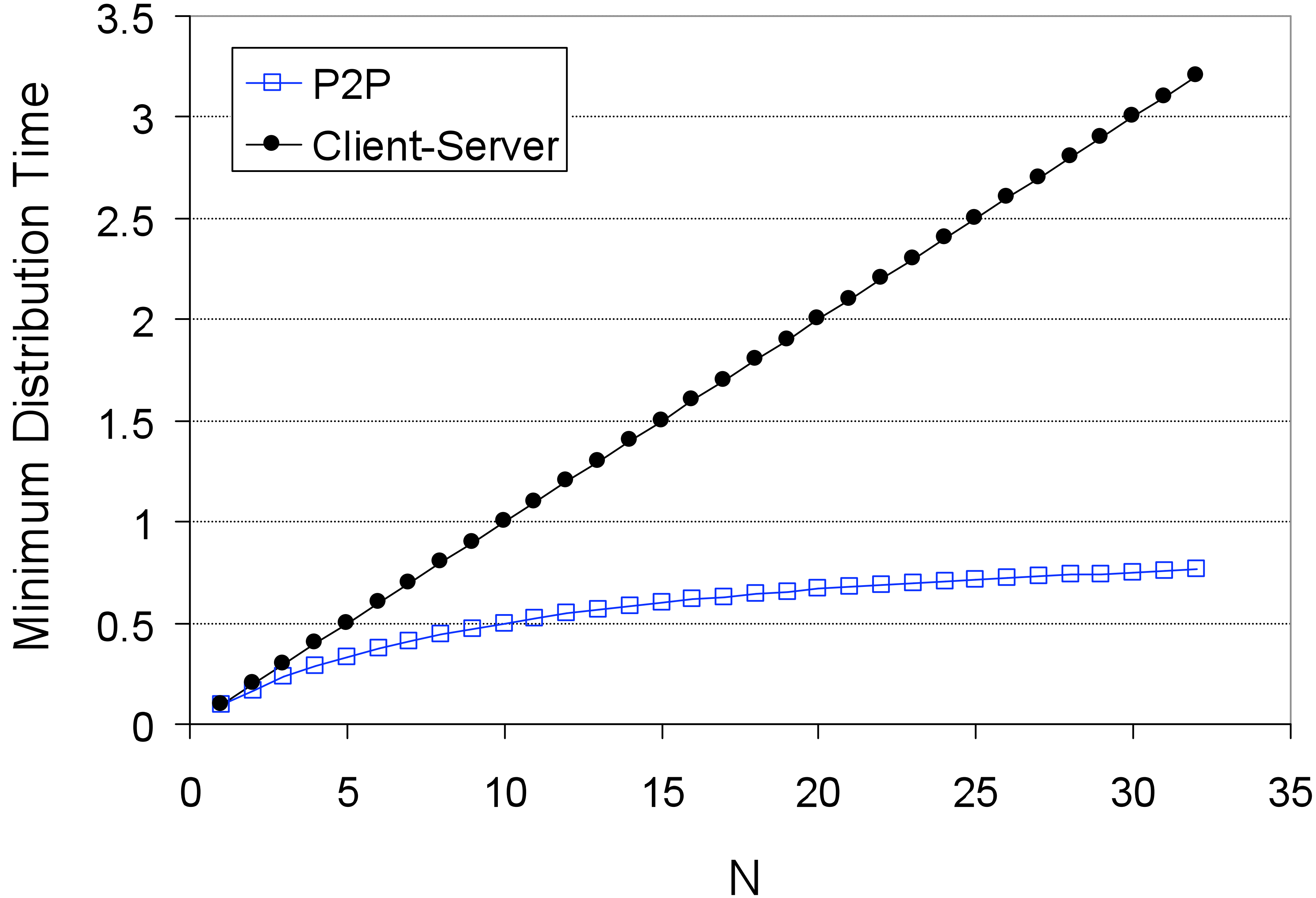
increases linearly in N (for large N)

same here!

$$\boxed{d_{P2P} = \max\{ F/u_s, F/d_{min}, NF/(u_s + \Sigma u_i) \}}$$

Server-Client vs P2P: Example

Client upload rate = u, F/u = 1 hour, us = 10u, dmin ≥ us

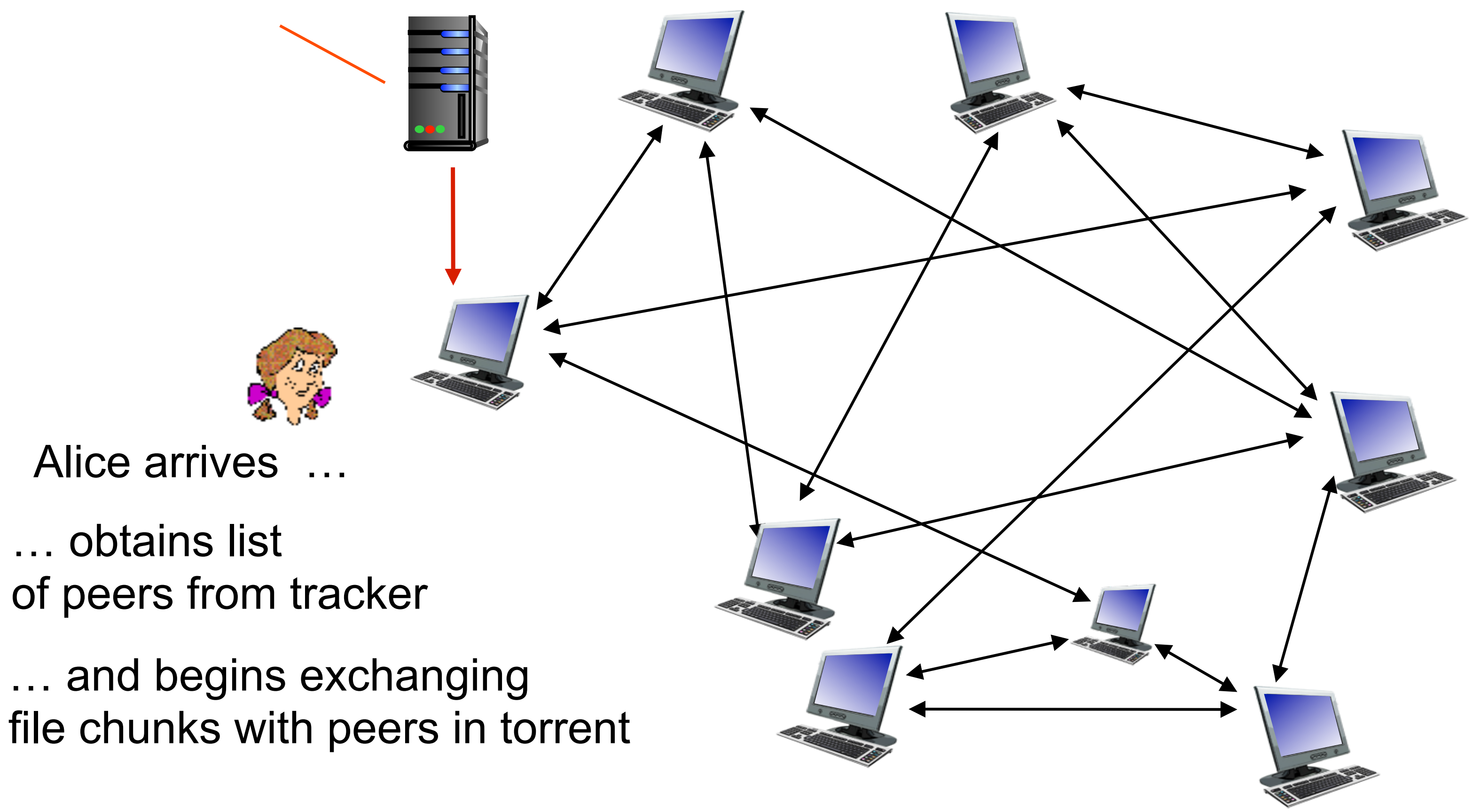


File Distribution: Bit Torrent

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

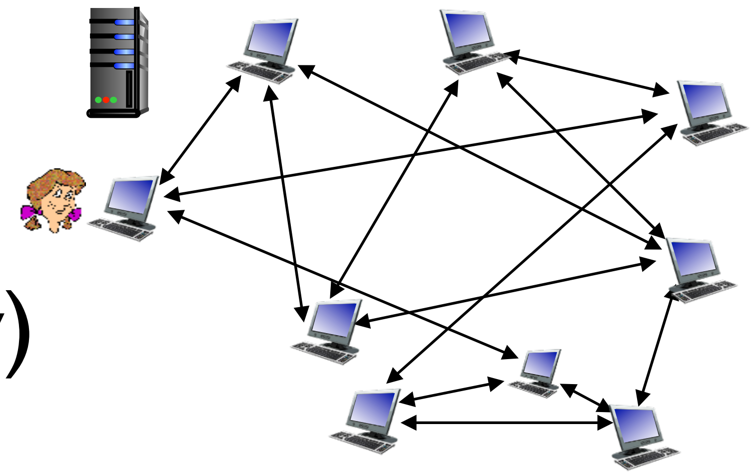
tracker: tracks peers participating in torrent

torrent: group of peers exchanging chunks of a file



BitTorrent (continued)

- Peer joining torrent...
  - ...has no chunks, but will accumulate them over time
  - ...registers with tracker to get list of peers, connects to subset of peers ("neighbors")
- While downloading, peer uploads chunks to other peers.
- *Churn*: Peers may come and go.
- Once peer has entire file, it may (selfishly) leave or (altruistically) remain



BitTorrent (even more)

Requesting Chunks

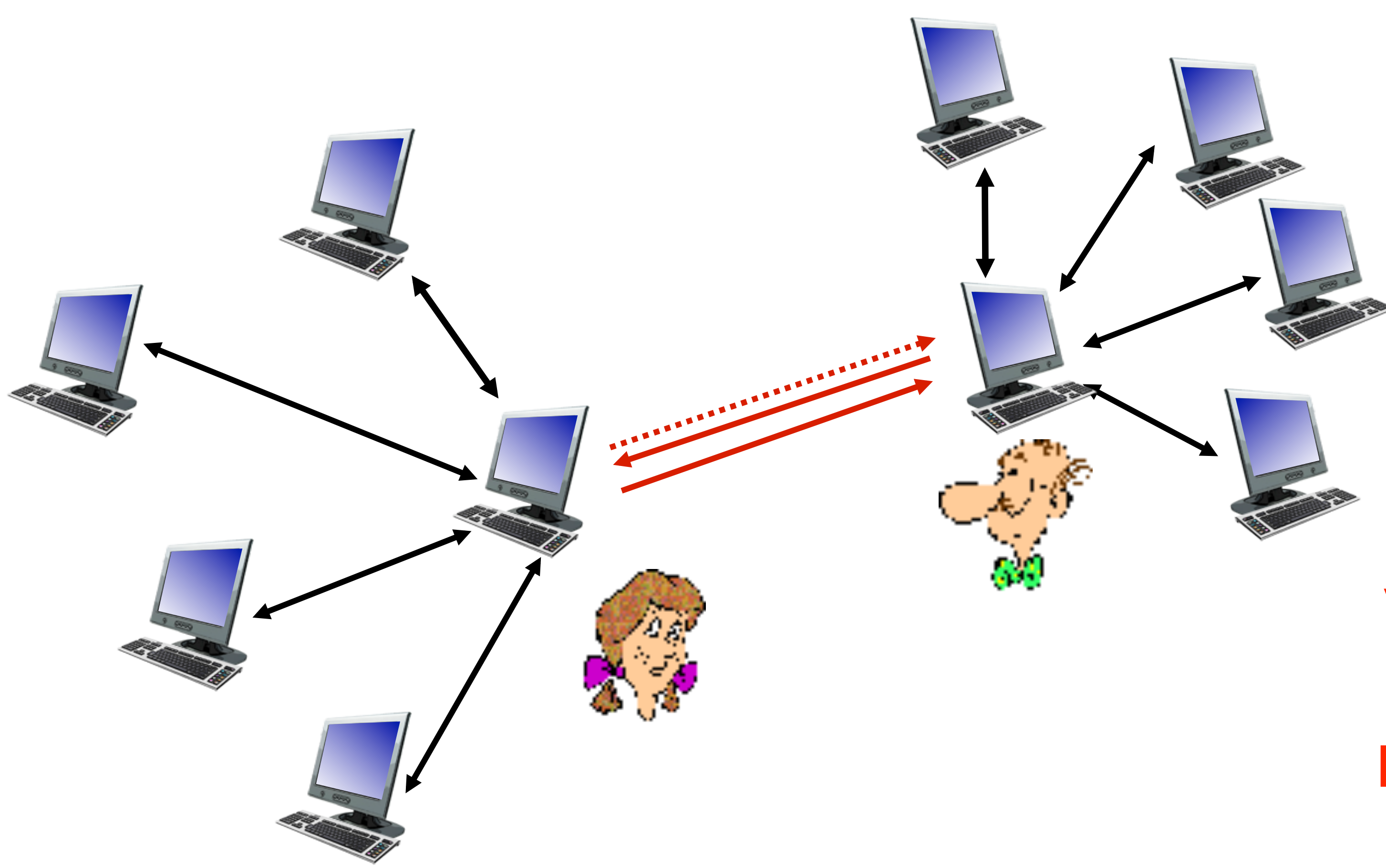
- at any given time, different peers have different subsets of file chunks
- periodically, a peer (Alice) asks each neighbor for list of chunks that they have.
- Alice sends requests for her missing chunks
  - rarest first

Sending Chunks

- Alice sends chunks to four neighbors currently sending her chunks at the highest rate
  - re-evaluate top 4 every 10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - newly chosen peer may join top 4
  - "optimistically unchoke"

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

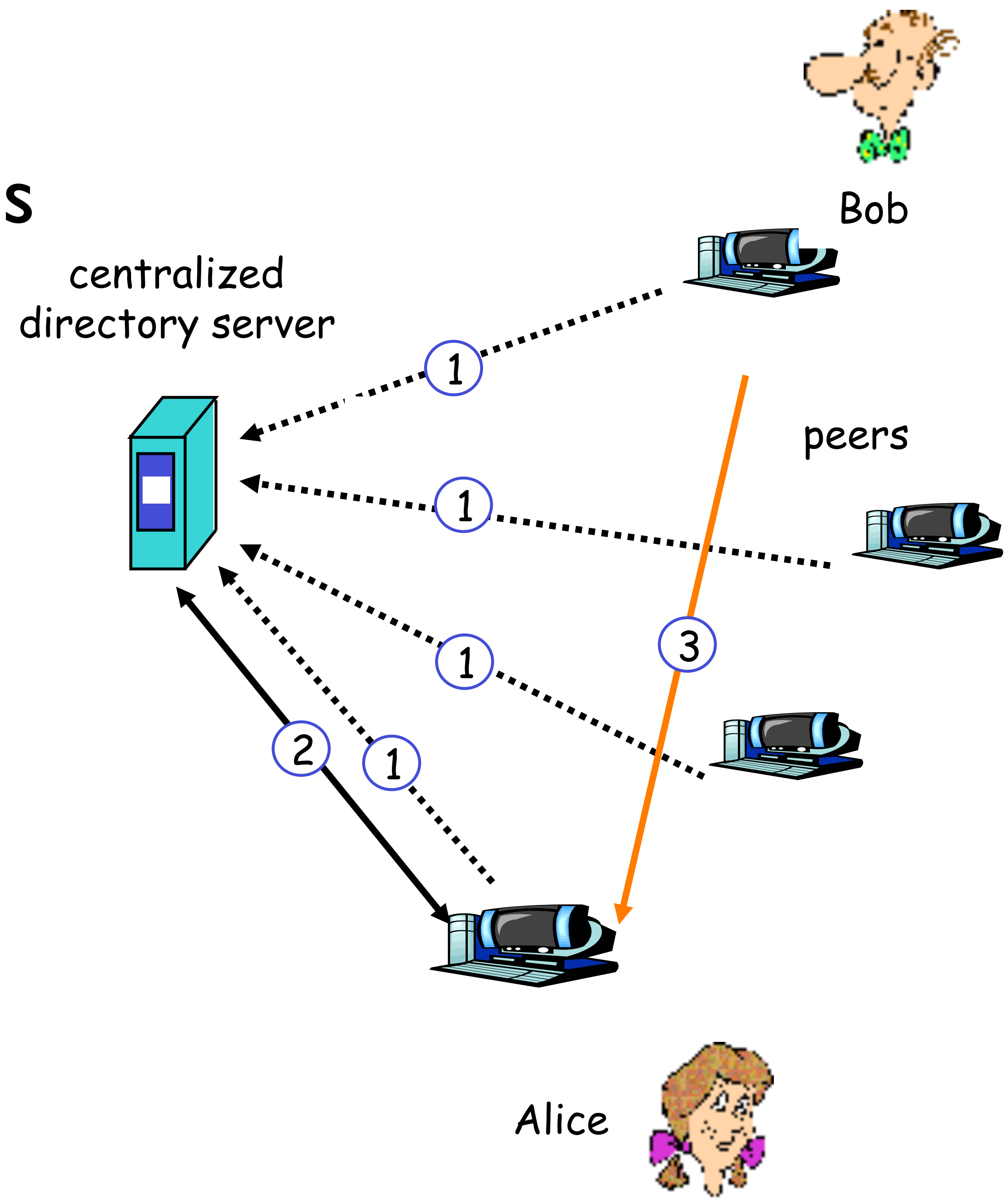


With higher upload rate, can find better trading partners & get file faster!

P2P: Finding Information - Centralized Index

- original "Napster" design

1. When peer connects, it informs central server:
    - IP address
    - content
2. Alice queries for "Hey Jude"
3. Alice requests file from Bob



P2P: Problems with Centralized Directory

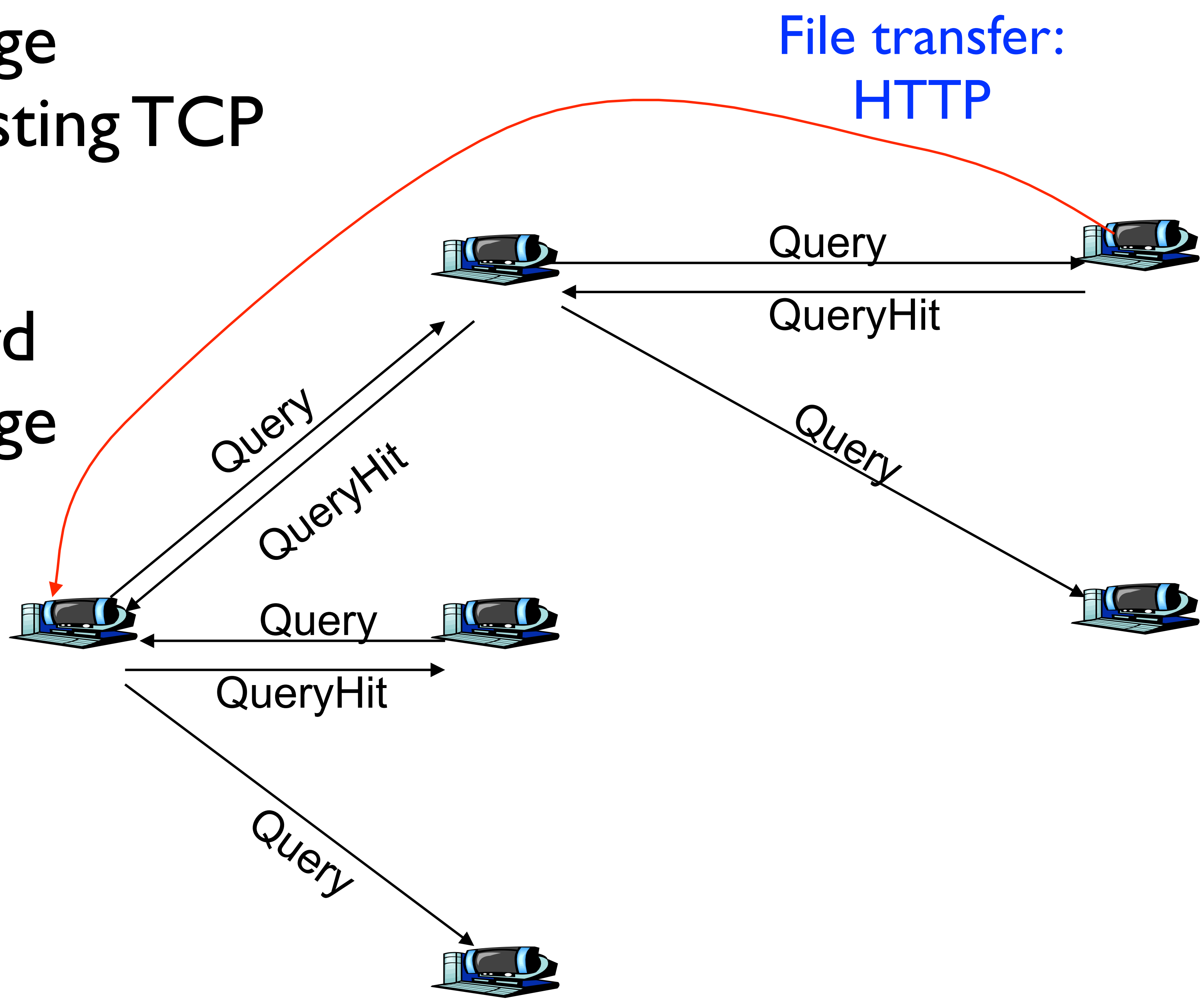
- Single point of failure
- Performance bottleneck
- Copyright infringement: "target" of lawsuit is obvious

file transfer is decentralized, but locating content is highly centralized

P2P: Finding Information - Query Flooding

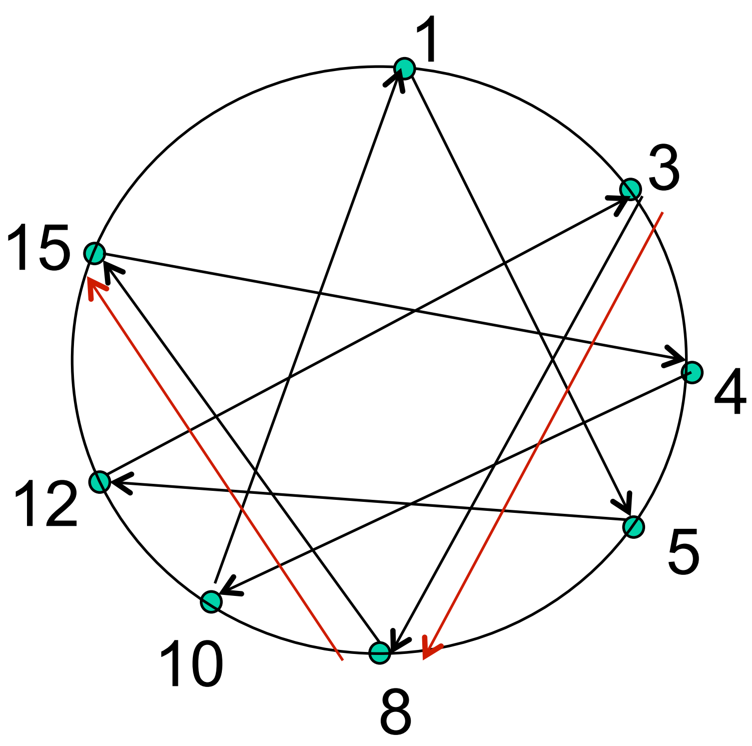
- Query message sent over existing TCP connections
- peers forward Query message
- QueryHit sent over reverse path

Scalability: limited scope flooding



Distributed Hash Tables

- DHT: a *distributed P2P database*
- database has (key, value) pairs; examples:
  - key: ss number; value: human name
  - key: movie title; value: IP address
- Distribute the (key, value) pairs over the (millions of peers)
- a peer queries DHT with key
  - DHT returns values that match the key
- peers can also insert (key, value) pairs



Chapter 2: Summary

Most importantly: We learned about protocols

- Communications architectures
  - Client/Server
  - P2P
  - Hybrid
- Stateless vs Stateful
- Reliable vs Unreliable transfer
- Complexity at the network edge
- Message Formats:
  - headers vs. data

Socket Programming

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

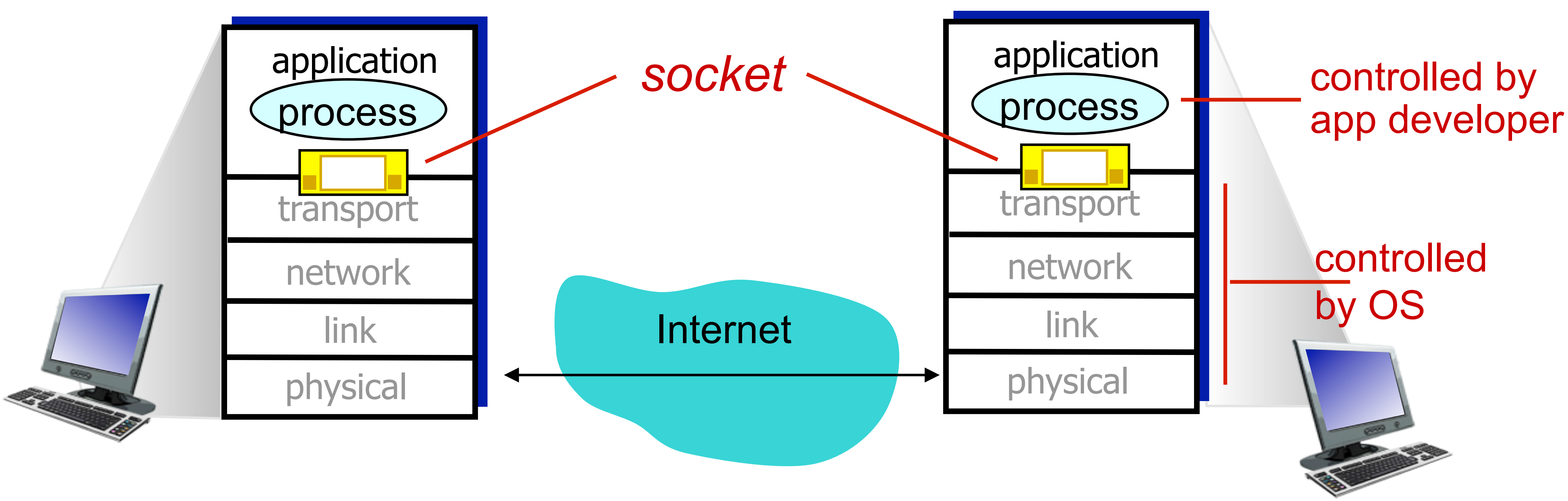
- Socket API
  - Introduced ins BSD4.1 Unix 1981
  - Explicitly created, used & released by apps
  - client/server paradigm
  - two types of transport service via socket API:
    - unreliable datagram (UDP)
    - reliable, byte stream-oriented (TCP)

socket

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

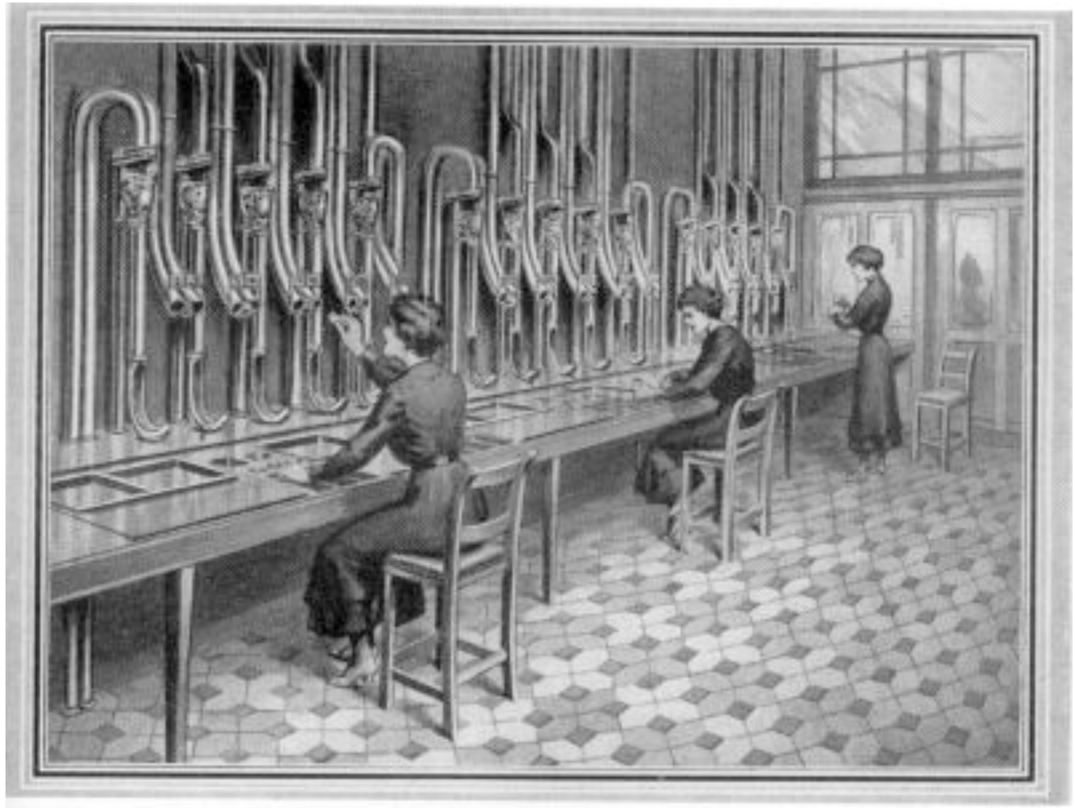
Socket-Programming Using TCP

- Goal: learn how to build client/server applications that communicate using sockets
- Socket: a door between application process and end-end-transport protocol (UDP or TCP)



Just Like the Bank Tube

- You do not need to be aware of all of the operations that occur in the tube, but you can potentially impact transport.
  - Selecting a solid capsule ensures reliable delivery of contents.
  - A less solid capsule might send your spare change flying...



Socket Basics

- Most API functions return -1 on failure
- Creating a new socket
  - `int socket(int addressFamily, int type, int protocol)`
  - *addressFamily:* AF_INET (used to be PF_INET)
  - *type:* SOCK_STREAM, SOCK_DGRAM
  - *protocol:* IPPROTO_TCP, IPPROTO_UDP
- Closing a socket
  - `int close(int socket)`

Specifying Addresses

- API uses the generic `struct sockaddr`

```
struct sockaddr
{
  unsigned short sa_family; /* Address family (e.g., AF_INET) */
  char sa_data[14];         /* Family-specific address info */
};
```

- AF_INET has a specific “instance”

```
struct in_addr
{
  unsigned long s_addr;      /* Internet address (32 bits) */
};

struct sockaddr_in
{
  unsigned short sin_family; /* Internet Protocol (AF_INET) */
  unsigned short sin_port;   /* Address port (16 bits) */
  struct in_addr sin_addr;   /* Internet address (32 bits) */
  char sin_zero[8];          /* Not used */
};
```

- PF_XXX and AF_XXX historically interchangeable

An Example

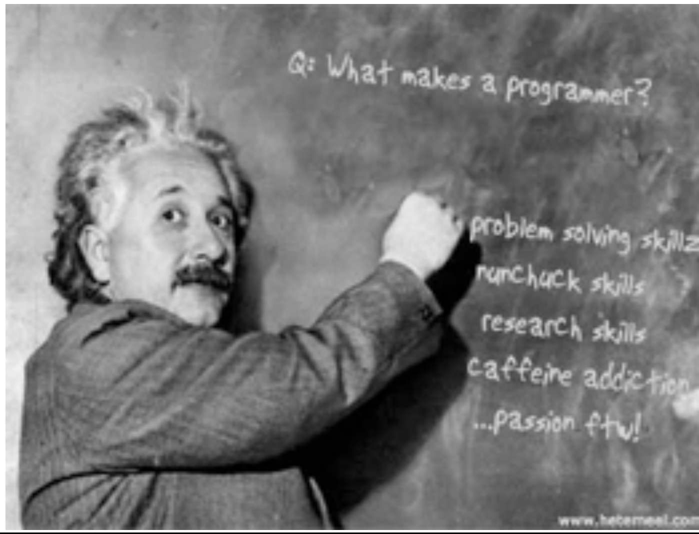
- Steps
  - Clear the memory structure!
  - Assign the address family
  - Assign the IP address (here we derive it from a string)
  - Assign the port (Note htons())

```
char servIP = "10.0.0.1";
unsigned short servPort = 25;
struct sockaddr_in servAddr;

memset(&servAddr, 0, sizeof(servAddr));
servAddr.sin_family      = AF_INET;
servAddr.sin_addr.s_addr = inet_addr(servIP);
servAddr.sin_port        = htons(servPort);
```

Socket Programming with TCP

- Client must contact server.
  - Server process must be running.
  - Server must have created socket (door) the welcomes client's contact
- Client contacts server by:
  - creating client 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 (more later)



TCP Basics

- In TCP, you must first create a connection
  - **int** connect(**int** socket, **struct sockaddr** **foreignAddress*, **unsigned int** *addressLength*)
- Then, you can send and receive
  - Returns number of bytes sent or received (-1 on error)
  - **int** send(**int** *socket*, **const void** **msg*, **unsigned int** *msgLength*, **int** *flags*)
  - **int** recv(**int** *socket*, **void** **rcvBuffer*, **unsigned int** *bufferLength*, **int** *flags*)
- An Example:

```
int sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
connect(sock, (struct sockaddr*) &servAddr, sizeof(servAddr));
...
num_bytes = send(sock, dataString, dataLen, 0);
...
num_bytes = recv(sock, buf, bufSize-1, 0);
buf[num_bytes] = '\0';
```

Don't forget error checking!

A Word About Boundaries

- TCP provides *flow control*
  - Why is flow control important?
- This means the data provided to send() might be broken into parts
  - The same goes for recv()
  - Moral: do not assume correspondence between send() and recv()
    - Commonly place recv() inside a while loop (until returned bytes is 0)
    - Also, sender and receiver may have different buffer sizes

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 <u>unreliable</u> transfer of groups of bytes ("datagrams") between client and server

A simple client

```
int sock;
struct sockaddr_in serv_addr;
char *msg;
int msglen;
int rbytes;
...
/* Create a new socket */
if ((sock = socket(PF_INET,  SOCK_STREAM, IPPROTO_TCP) < 0)) {
    fatal_error("socket() failed");
}
...
/* Construct the server address */
memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family          = AF_INET;
serv_addr.sin_addr.s_addr     = inet_addr("10.0.0.1");
serv_addr.sin_port            = htons(25);
...
/* Connect to server */
if (connect(sock, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
    fatal_error("connect() failed");
}
...
/* Send a message */
msglen = strlen(msg);
if (send(sock, msg, msglen, 0) != msglen) {
    fatal_error("send() sent unexpected number of bytes");
}
...
/* Wait for reply */
...
```

Client chooses random port

- Here, the client establishes a TCP connection to a server with a known IP address and port
- How does the server know the address and port of the client?
- When will the connection occur?

TCP Server

- TCP servers perform for steps
  - Create a new socket (`socket()`)
  - Assign a port number to the socket (`bind()`)
  - Inform the system to listen for connections (`listen()`)
  - Repeatedly accept and process connections
    `accept(), recv(),send(),close()`

Binding to a port

- Servers run on *known* ports (e.g., 80 for HTTP)
- Use the bind() function to bind to a port
  - `int bind(int socket, struct sockaddr *localAddress, unsigned int addressLength)`
- What IP address should be used?
  - `INADDR_ANY` (any incoming interface)
- Example:

```
memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family          = AF_INET;
serv_addr.sin_addr.s_addr     = htonl(INADDR_ANY);
serv_addr.sin_port            = htons(80);
...
if (bind(sock, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
    fatal_error("bind() failed");
}
```

Listening

- Once the socket is bound to a port, the server can listen for new connections.
- We inform the system of our intentions with `listen()`
  - `int listen(int socket, int queueLimit)`
- *queueLimit* specifies an upperbound on the number of incoming connections
- Example:

```
#define MAXPENDING 10       /* Maximum number of incoming connections */
...
if (listen(sock, MAXPENDING) < 0) {
    fatal_error("listen() failed");
}
```

Processing Connections

- New client connections are accepted with `accept()`
  - `int accept(int socket, struct sockaddr *clientAddress, unsigned int *addressLength)`
- The call *blocks* until a new connection occurs, returning a socket descriptor or -1 on error
- Example:

```
int clnt_sock;
struct sockaddr_in clnt_addr;
int clnt_len;
...
clnt_len = sizeof(clnt_addr);
if ( (clnt_sock = accept(sock, (struct sockaddr *) &clnt_addr, &clnt_len)) < 0) {
    fatal_error("accept() failed");
}
...
do {
    /* recieve data */
    rbytes = recv(clnt_sock, buf, buflen, );
    ...
} while (...);

/* send response */

/* close the client socket */
close(clnt_sock);
```

Putting things together

```c
#define MAXPENDING 10       /* Maximum number of incoming connections */
...
int sock;
struct sockaddr_in serv_addr;
int clnt_sock;
struct sockaddr_in clnt_addr;
int clnt_len;
...
/* Create a new socket */
if ((sock = socket(PF_INET,  SOCK_STREAM, IPPROTO_TCP) < 0)) {
    fatal_error("socket() failed");
}
...
/* Construct the server address */
memset(&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family        = AF_INET;
serv_addr.sin_addr.s_addr   = htonl(INADDR_ANY);
serv_addr.sin_port          = htons(80);
...
/* Bind the connection */
if (bind(sock, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
    fatal_error("bind() failed");
}
...
/* Listen for new connections */
if (listen(sock, MAXPENDING) < 0) {
    fatal_error("listen() failed");
}
...
/* Wait for clients to connect */
while (1) {
    clnt_len = sizeof(clnt_addr);
    if ( (clnt_sock = accept(sock, (struct sockaddr *) &clnt_addr, &clnt_len)) < 0) {
        fatal_error("accept() failed");
    }
    ...
    handle_client(clnt_sock);
}
```

- Server uses an infinite loop to continually wait for connections
- How do you handle multiple simultaneous connections?

Simultaneous Connections

- There is more than one way to handle multiple simultaneous connections
  - multiple processes (`fork()` and `exec()`)
  - multiple threads (pthreads)
- What about listening on multiple ports?
  - multiplexing (`select()`)
- Take a look at `select()`
  - See TCP/IP Sockets in C (or your favorite reference) for more information on processes and threads

Constructing Messages

- Until now, we have only discussed sending character strings
- How do we send more complex data structures?
  - We convert data structures to and from byte arrays
    - serialization, deserialization
    - marshalling, unmarshalling
    - deflating, inflating
- Remember, we must be cognizant of Endianess
  - Always use network format (big endian)
    - htonl(), htons(), ntohl(), ntohs()

Encoding data

- There are multiple ways of encoding data
  - Convert numbers to strings representing each digit
  - send the bytes directly
- When does the receiver stop receiving?
  - We can use a delimiter (similar to '\0' in char arrays)
  - We can establish predefined data formats
  - What if data is an arbitrary length?
    - Data framing: use a header of a predefined size.
      - The header has fixed sized fields and specifies size of data

Example: Framing a Message

```c
struct mesg {
    short len;
    char  *data;
};

/* On the sending side */
struct mesg message;
char *data;
int data_sz;
int msglen;

/* Assign data */
message.len = strlen(some_string);
message.data = strdup(some_string);

/* Create data buffer */
data_sz = sizeof(len) + len;
if ((data = malloc(data_sz)) == NULL) {
    fatal_error("malloc() failed");
}

/* convert to network format, saving length for use */
msglen = message.len;
message.len = htons(message.len);

/* Pack data */
memcpy(data, &(message.len), sizeof(message.len));
memcpy(data+sizeof(message.len), message.data, msglen);

/* Send the data */
if (send(sock, data, data_sz, 0) != data_sz) {
    fatal_error("send() failed");
}

...
```

```c
/* on the receiving side */
int recv_data(int sock, char *buf, int sz, int flags)
{
    int totb = 0; /* total bytes received */
    int retb;     /* temporary received bytes */
    do {
        if ((retb = recv(sock, &buf[totb], sz-totb, flags)) < 0) {
            return -1;
        }
        /* increment totb */
        totb += retb;
    } while ( totb < sz );

    return totb;
}
int main(int argc, char *argv[])
{
    short len;
    char *msg;
    ...
    /* receive the header */
    recv_data(sock, &len, sizeof(len), 0);

    /* convert to machine format */
    len = ntohs(len);

    /* allocate space */
    if ((msg = malloc(len+1)) == NULL) {
        fatal_error("malloc() failed");
    }

    /* receive the string */
    recv_data(sock, msg, len, 0);
    msg[len] = '\0';
    printf("Received string: [%s]\n", msg);
    ...
}
```

More on Addresses

- Retrieving addresses
  - inet_addr() returns -1 on error, however, this is the same as the address 255.255.255.255
    - Instead, you can use `inet_aton("10.0.0.1", &(serv_addr.sin_addr));`
  - What about DNS? - `gethostbyname()`

```
unsigned long resolve_name(char *name)
{
    struct hostent *host;          /* Structure containing host information */

    if ((host = gethostbyname(name)) == NULL) {
        fprintf(stderr, "gethostbyname() failed\n");
        exit(1);
    }

    return *((unsigned long *) host->h_addr_list[0]);
}
```

Socket Options

- Default options work for most cases, however, occasionally, an application will set specific options on a socket
  - See your favorite reference for a list of options
  - getsockopt(), setsockopt()
- In particular, the following may be useful in Project 2

```
int on = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
```

  - Used on a server to allow an IP/port address to be reused, otherwise, bind() may fail in certain situations

Next Time

- Transport Layer
  - Let's finally talk about the details behind TCP and UDP.
- Project 1
  - Remember that this due no later than 5pm on Thursday.
  - If you haven't started already, *you need to do this now...*