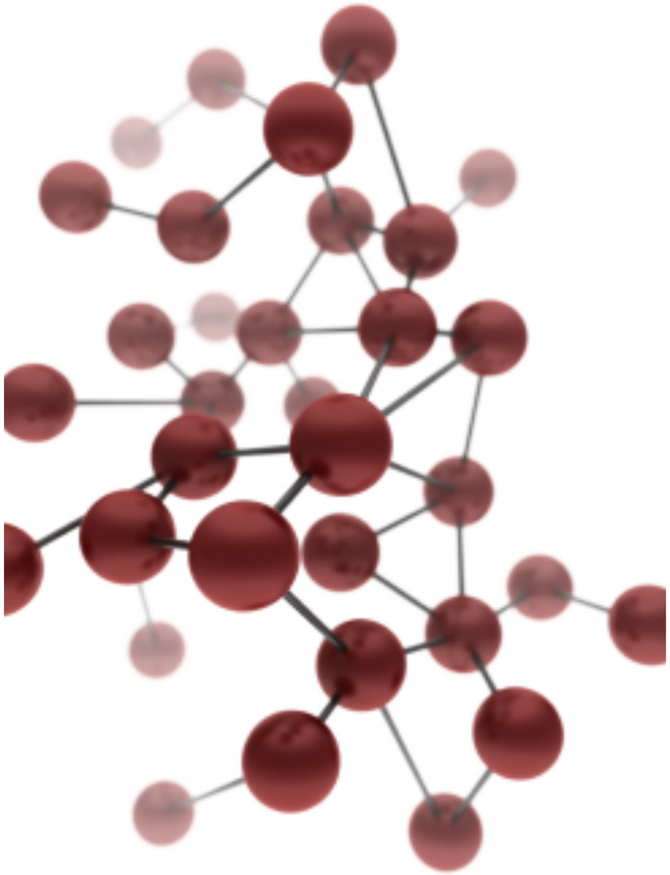


Lecture 2: Networking Concepts

Oct 3

Administrivia

- Project #1 out tomorrow
  - Will post on Piazza
  - Due 10/12 (Friday 11:59PM)

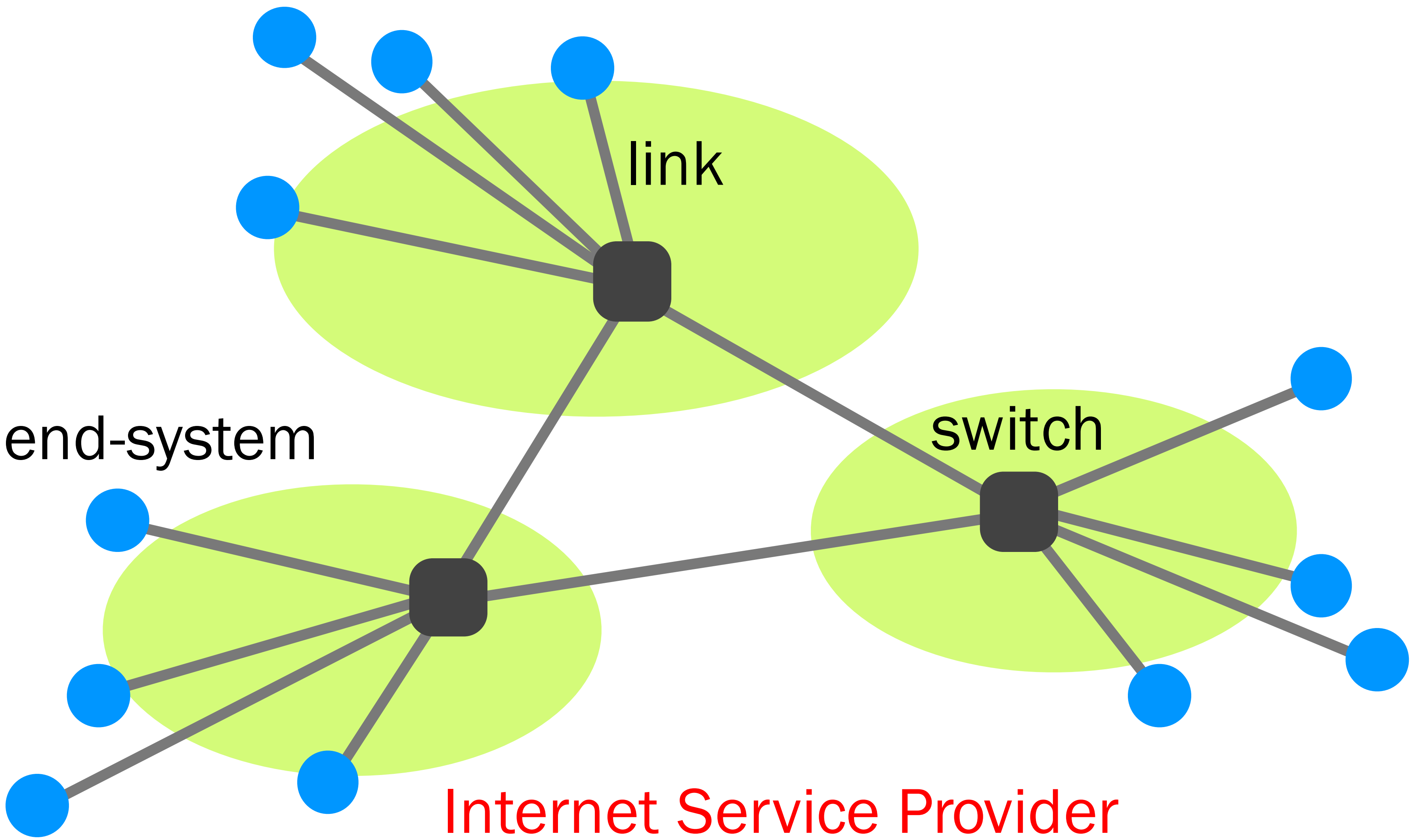
- Next M: Guest lecture
- Next W: No lecture, MORE time for project 1

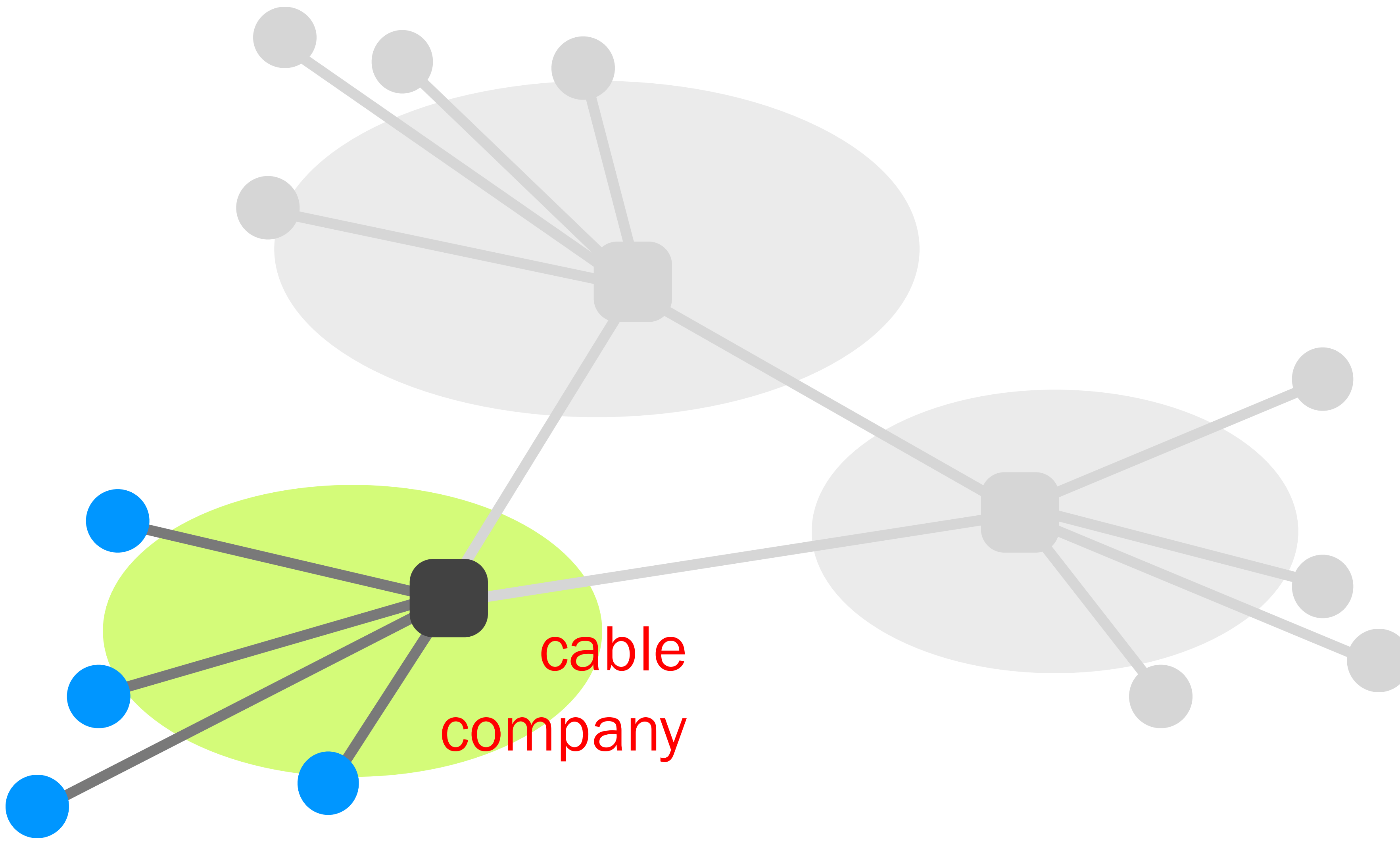
Recap: The Internet is…

- A federated system
- Of enormous scale
- Dynamic range
- Diversity
- Constantly evolving
- Asynchronous in operation
- Failure prone
- Constrained by what's practical to engineer
- Too complex for theoretical models
- "Working code" needn't mean much
- Performance benchmarks are too narrow

Today

- What is a network made of?
- How is it shared?
- Sockets
- How do we evaluate a network?

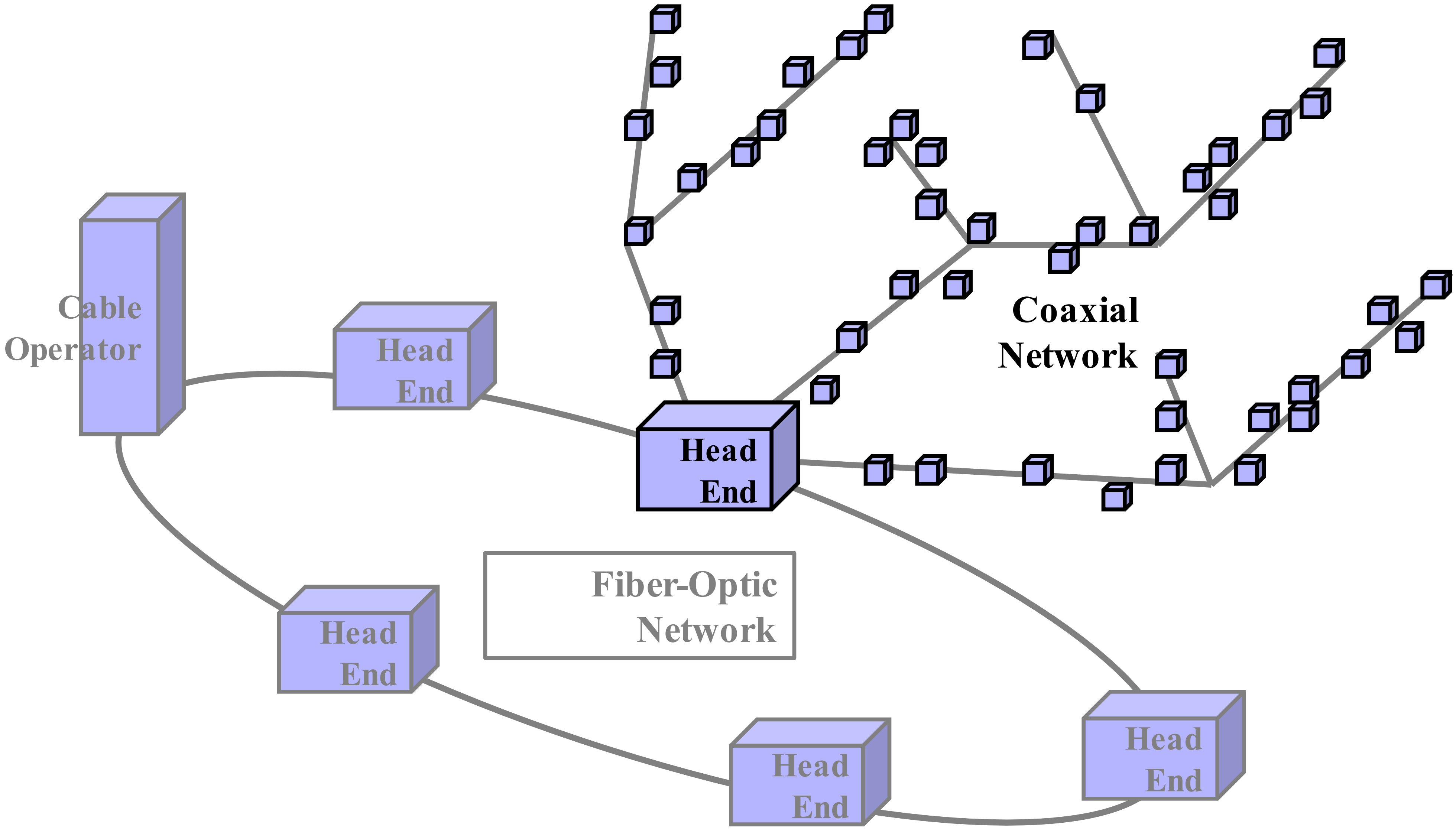

link
end-system
switch
Internet Service Provider

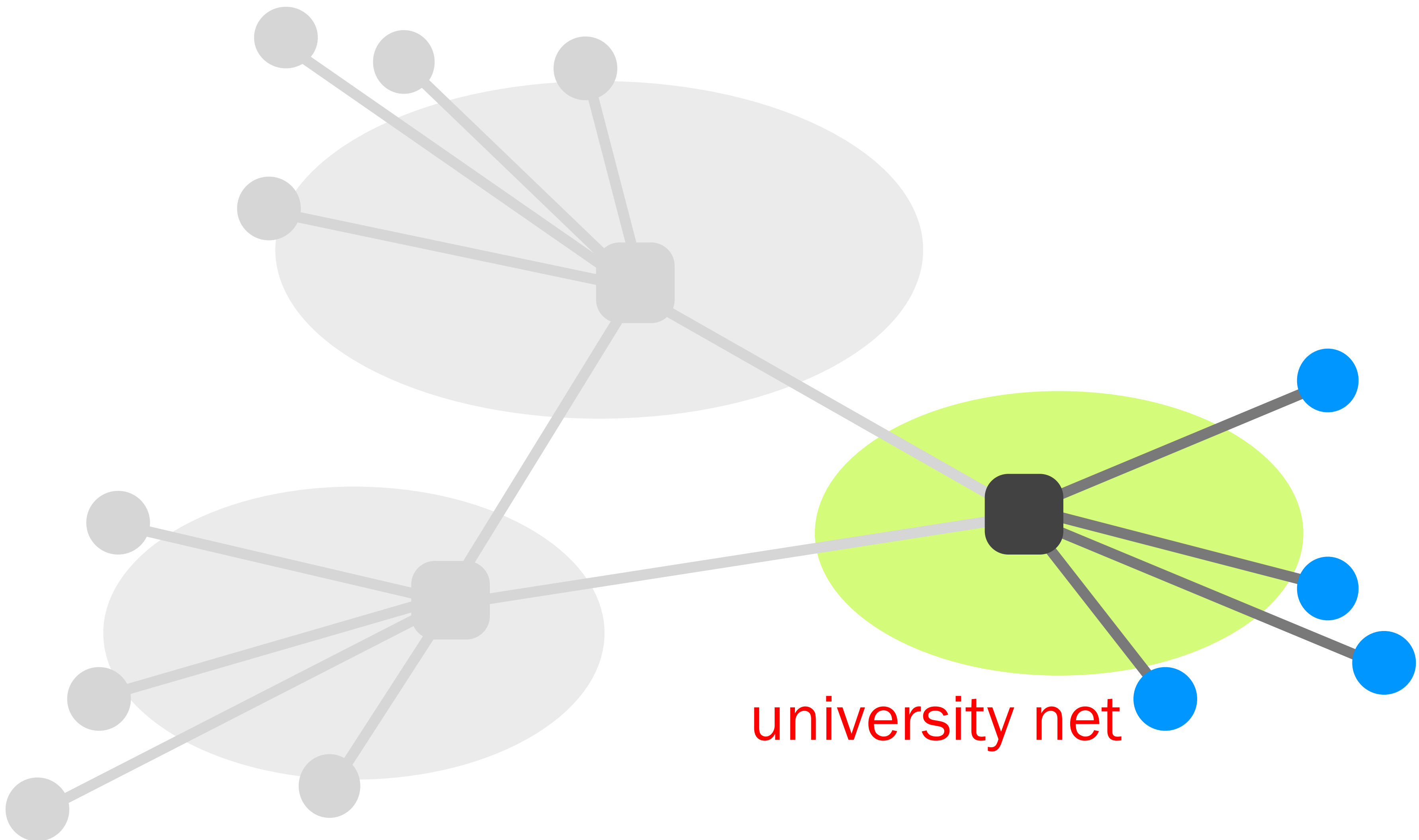

cable
company

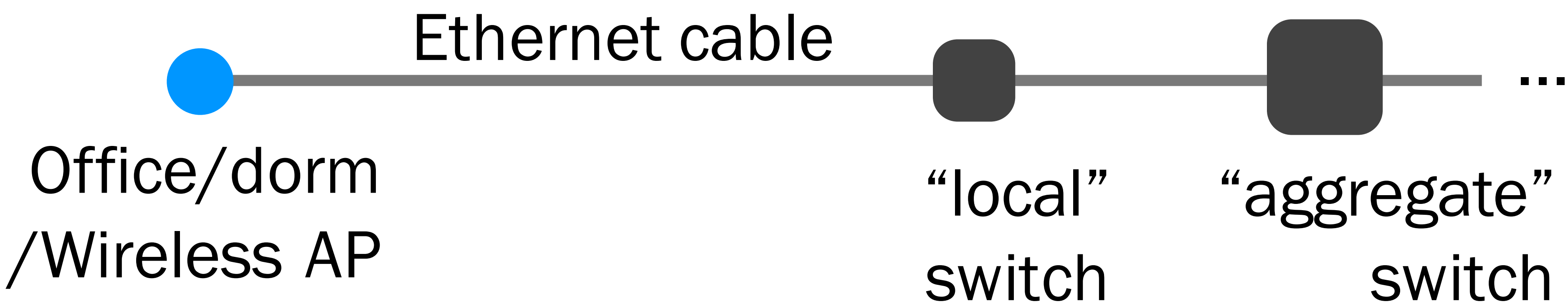
Cable

- Coaxial copper & fiber
- up to 42.8 Mbps downstream
- up to 30.7 Mbps upstream
- shared broadcast medium

Hybrid Fiber-Coax




university net

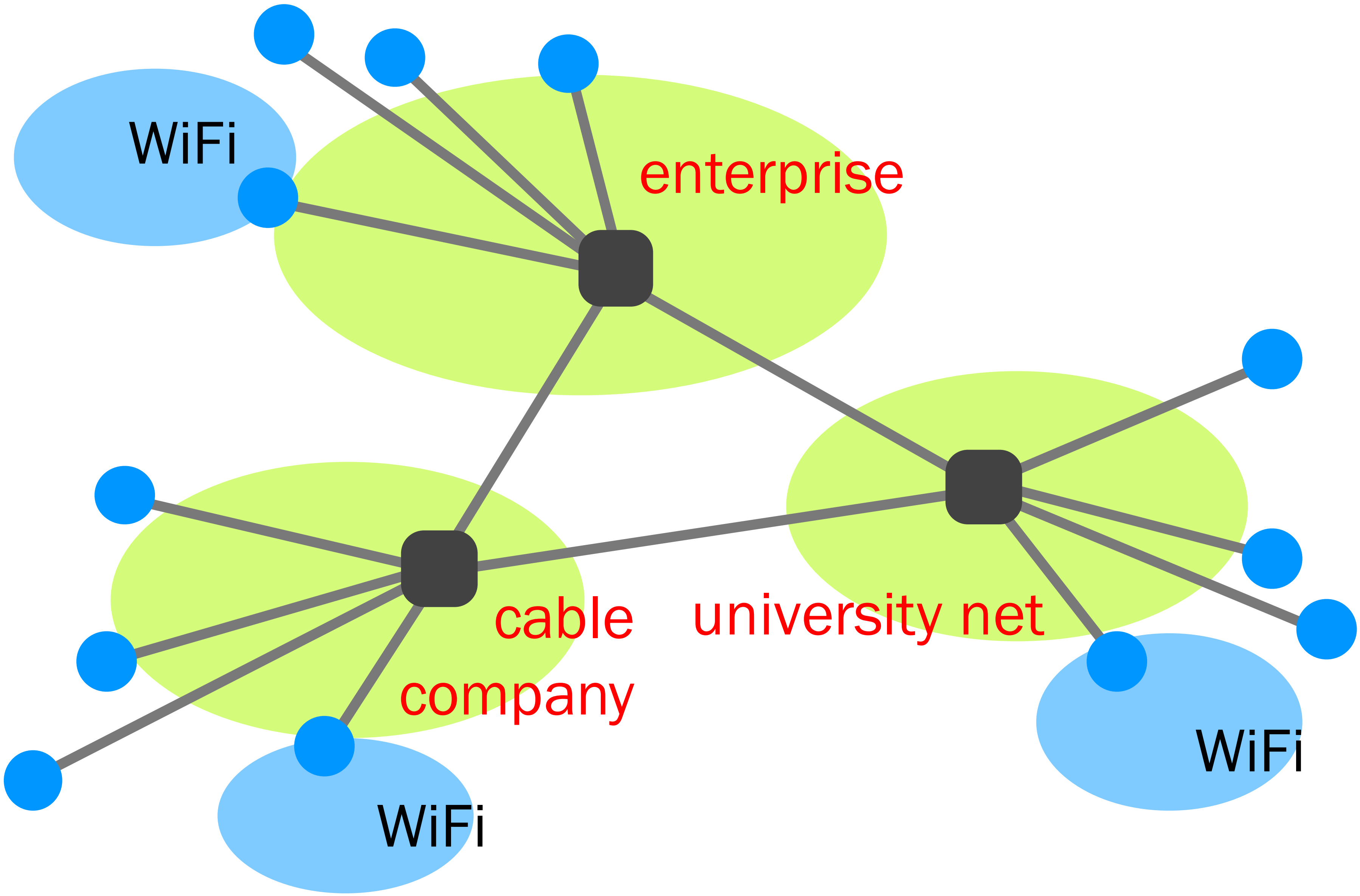

Ethernet cable
Office/dorm
/Wireless AP
“local”
switch
“aggregate”
switch
...

Ethernet

- Twisted pair copper
- 100 Mbps, 1 Gbps, 10 Gbps (each direction)

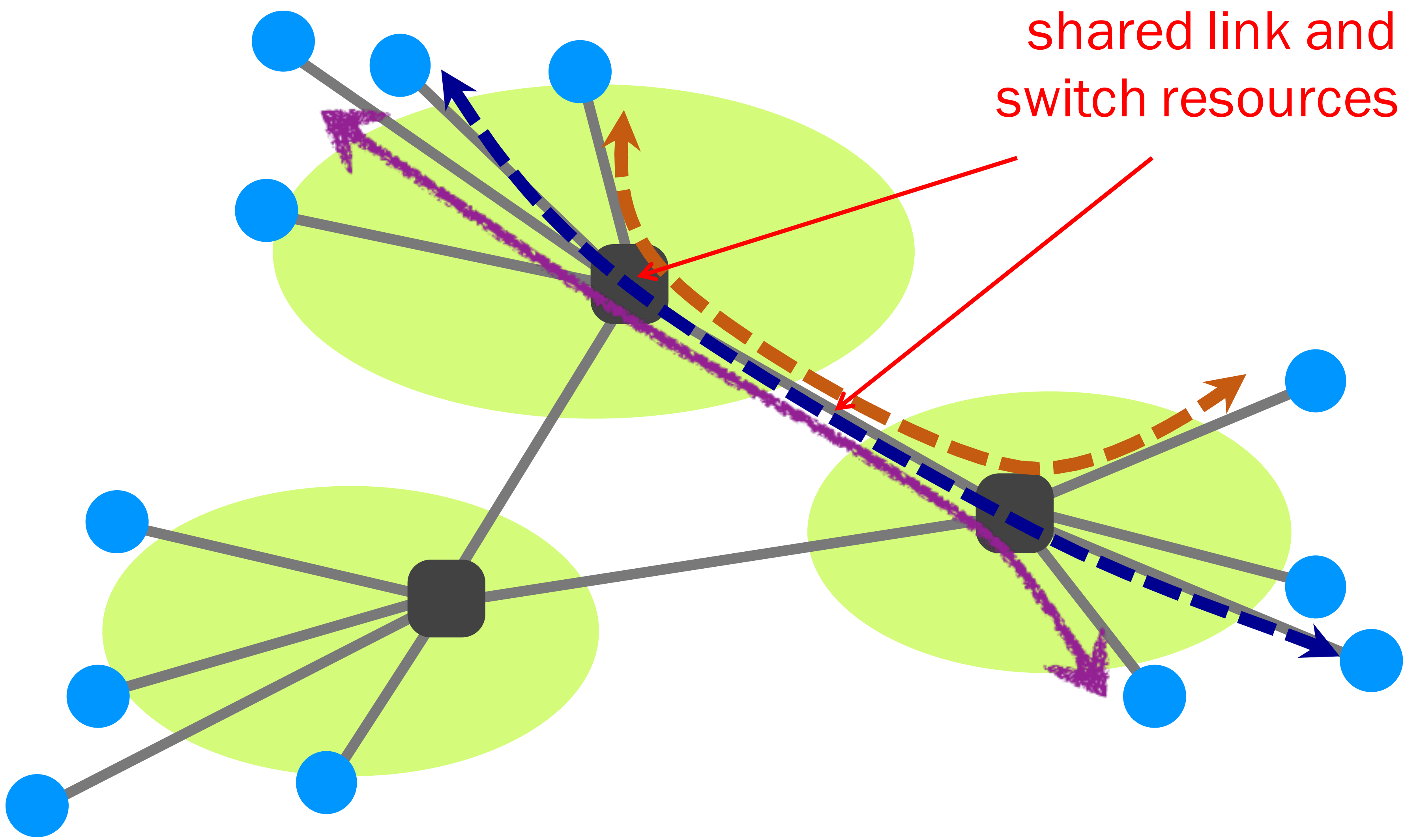
& more

- Cellular (smart phones)
- Satellite (remote areas)
- Fiber to the Home (home)
- Optical carrier (Internet backbone)

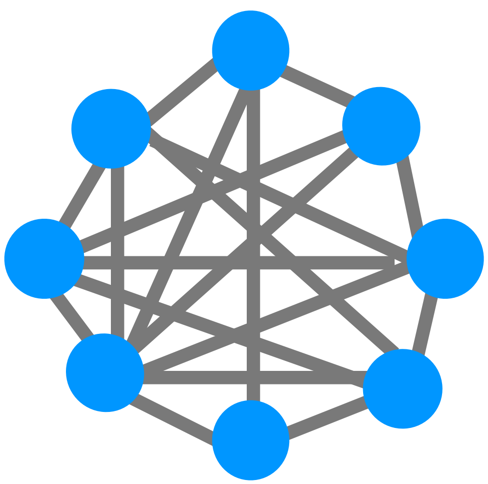

WiFi
enterprise
cable
company
university net
WiFi
WiFi

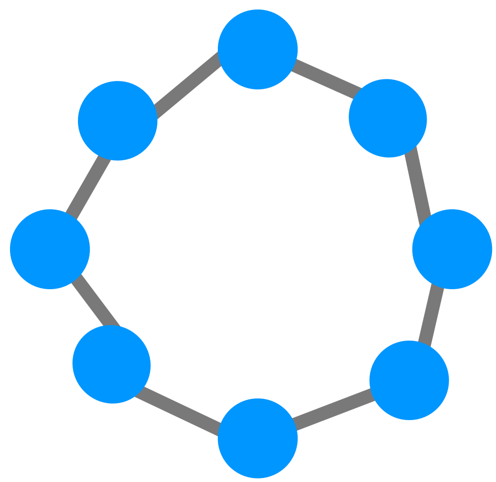
Today

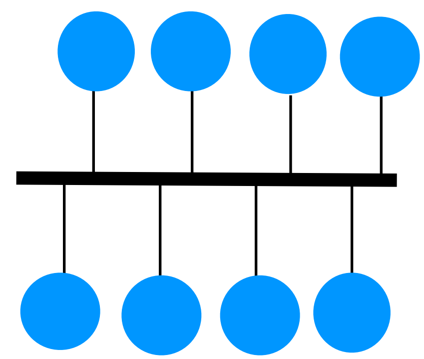
- What is a network made of?
- How is it shared?
- Sockets
- How do we evaluate a network?


shared link and
switch resources

How do we scale a network to many end-systems?



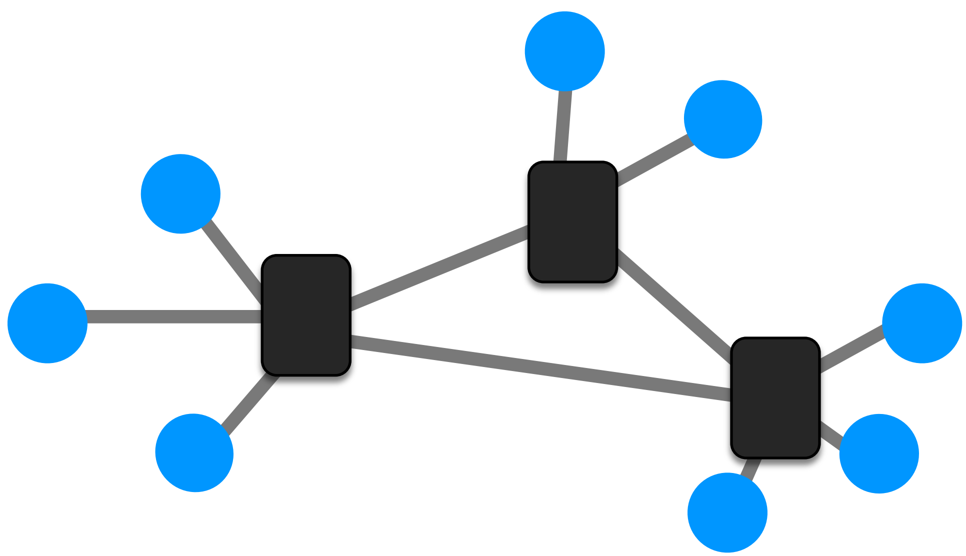




$O(n^2)$ links

per-node capacity scales as $1/n$

How do we scale a network to many end-systems?

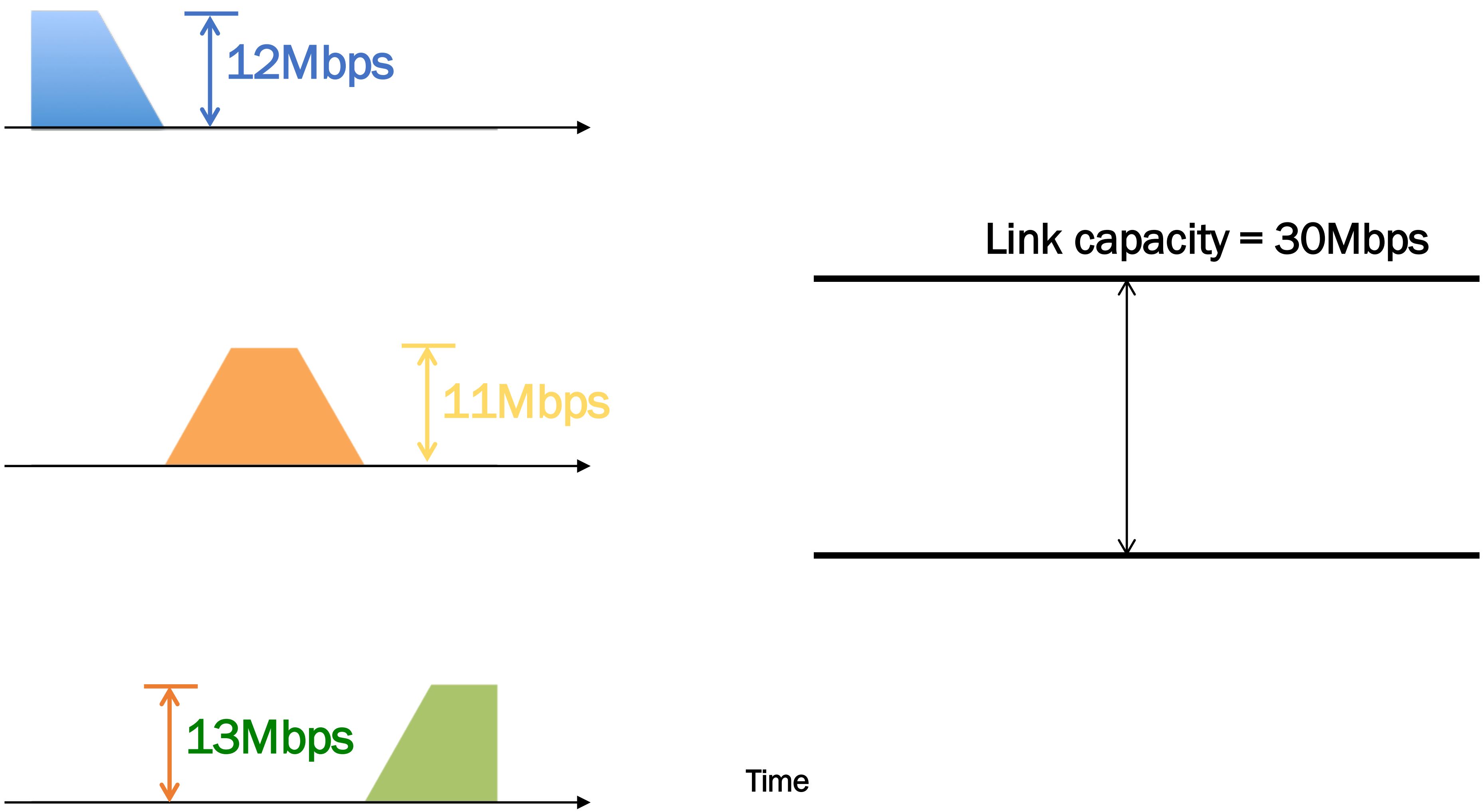


Switched networks enable efficient scaling!

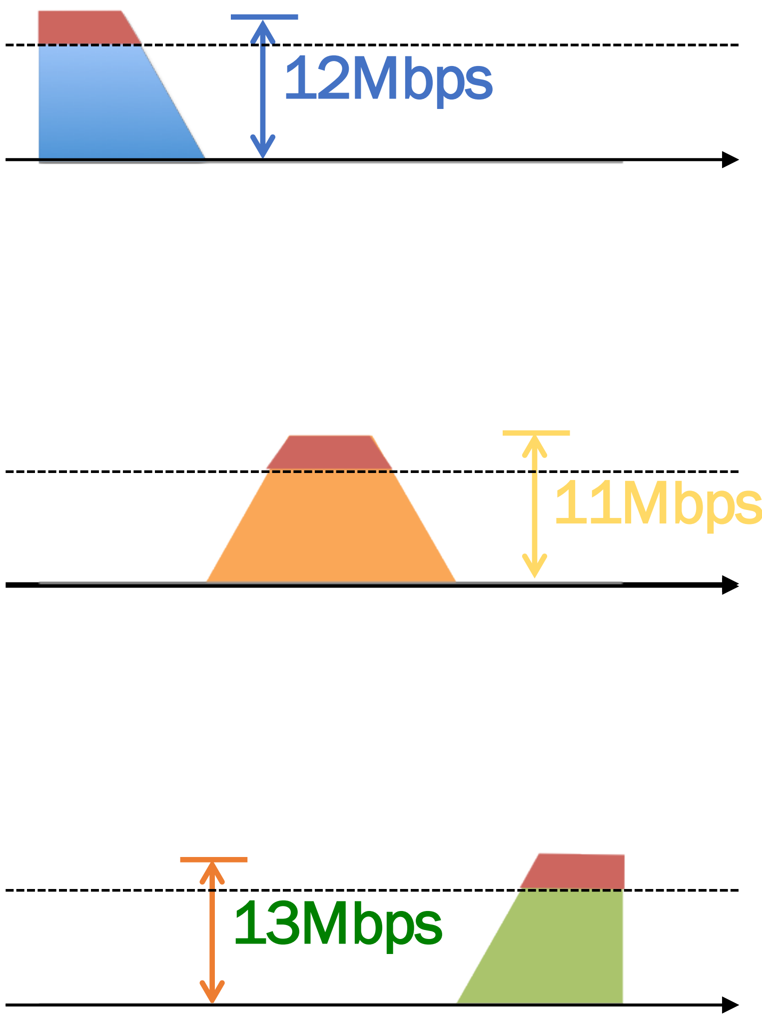
Two approaches to sharing

- Reservations
- On demand

Example: Three sources w/ "bursty" traffic

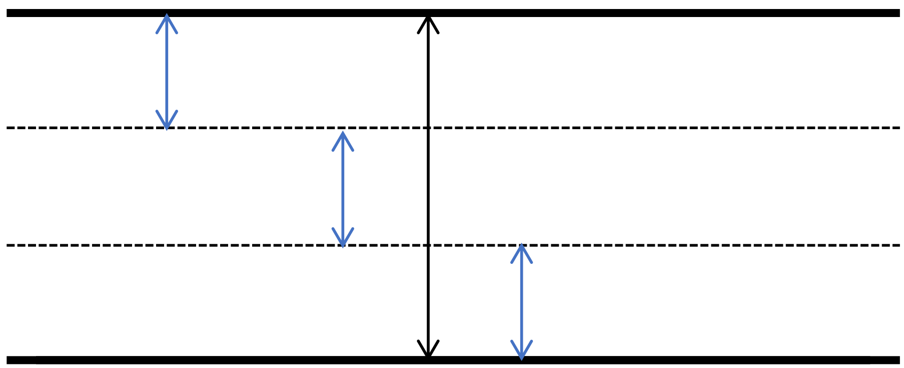


Intuition: reservations



Frequent overloading

Link capacity = 30Mbps

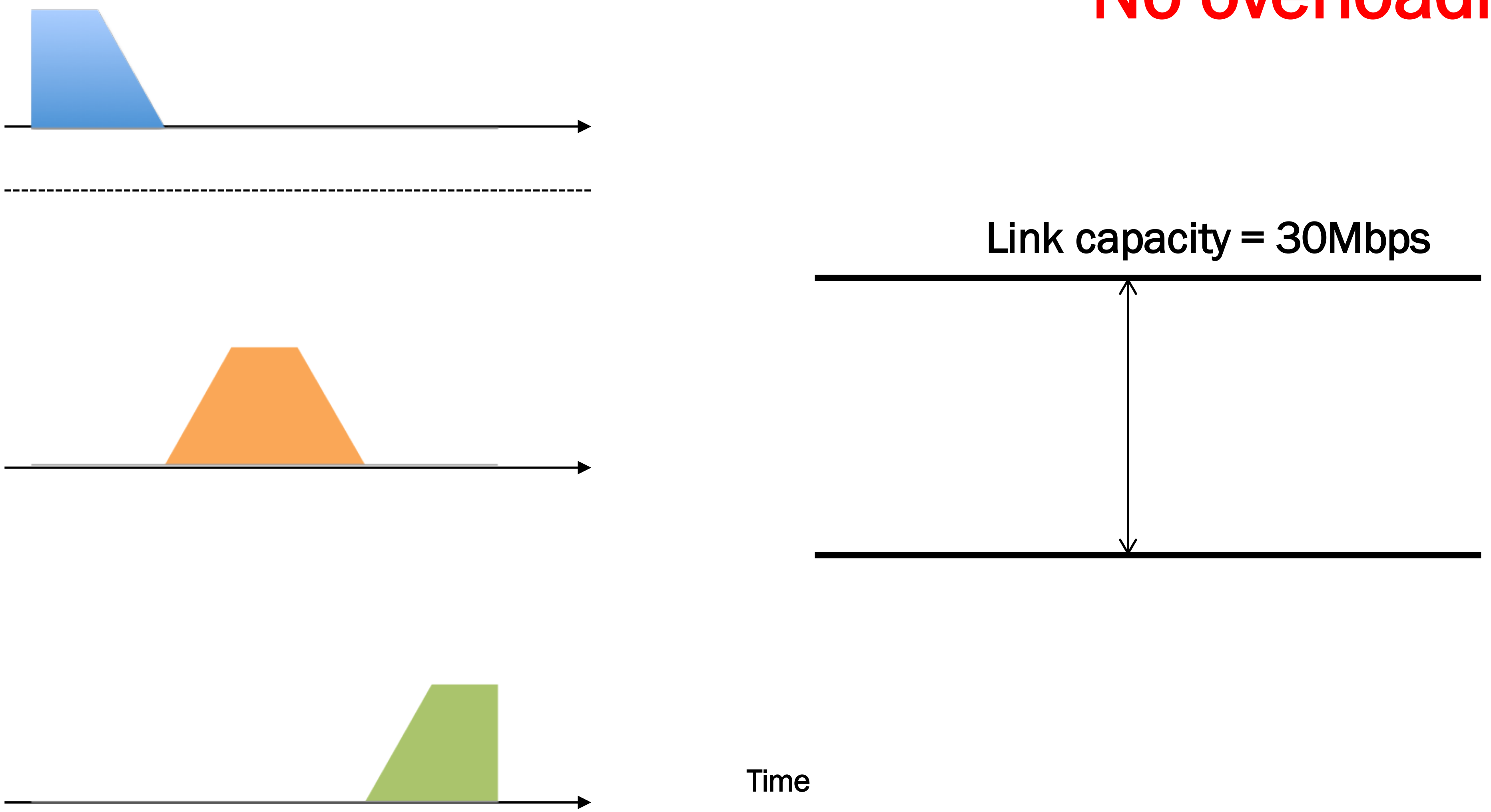


Each source gets 10Mbps

Time

Intuition: on demand

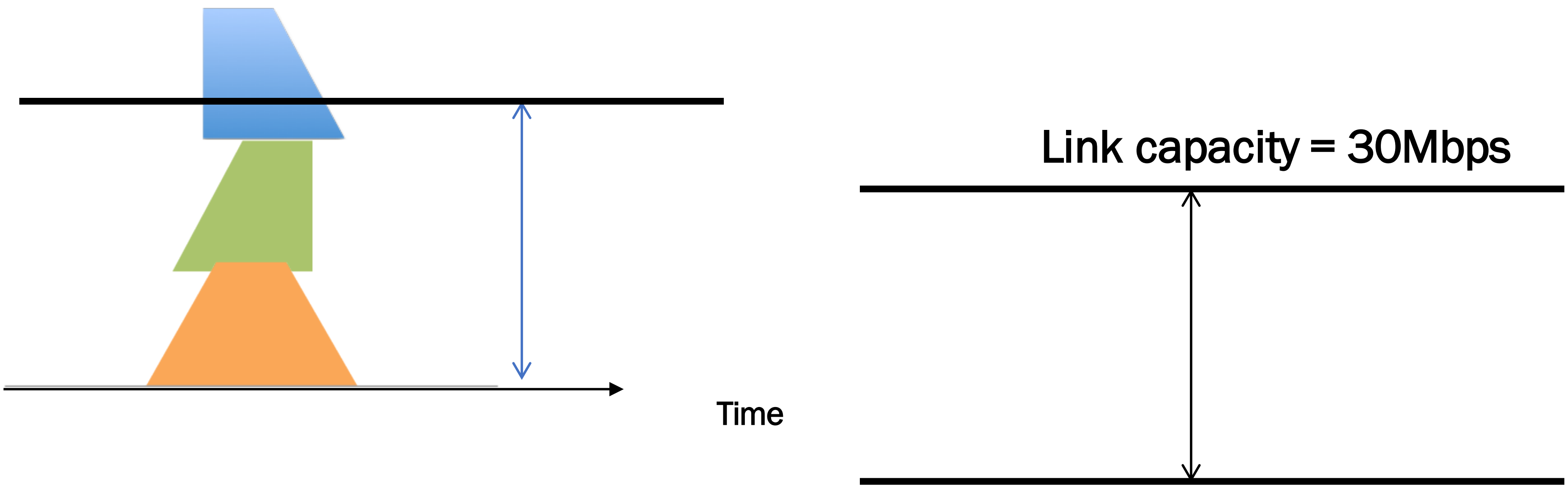
No overloading



Switching on-demand exploits *statistical multiplexing* better than reservations

- Sharing using the statistics of demand
- Good for bursty traffic (average << peak demand)
- Similar to insurance, with the same failure mode

Intuition: on demand



What do we do under overload?

Statistical multiplexing is a recurrent theme in computer science

- Phone network rather than dedicated lines
  - ancient history
- Packet switching rather than circuits
  - today's lecture
- Cloud computing
  - shared vs. dedicated machines

Two approaches to sharing

- Reservations
- On demand

How are these implemented?

Two approaches to sharing

- Reservations → circuit switching
- On demand → packet switching

How are these implemented?

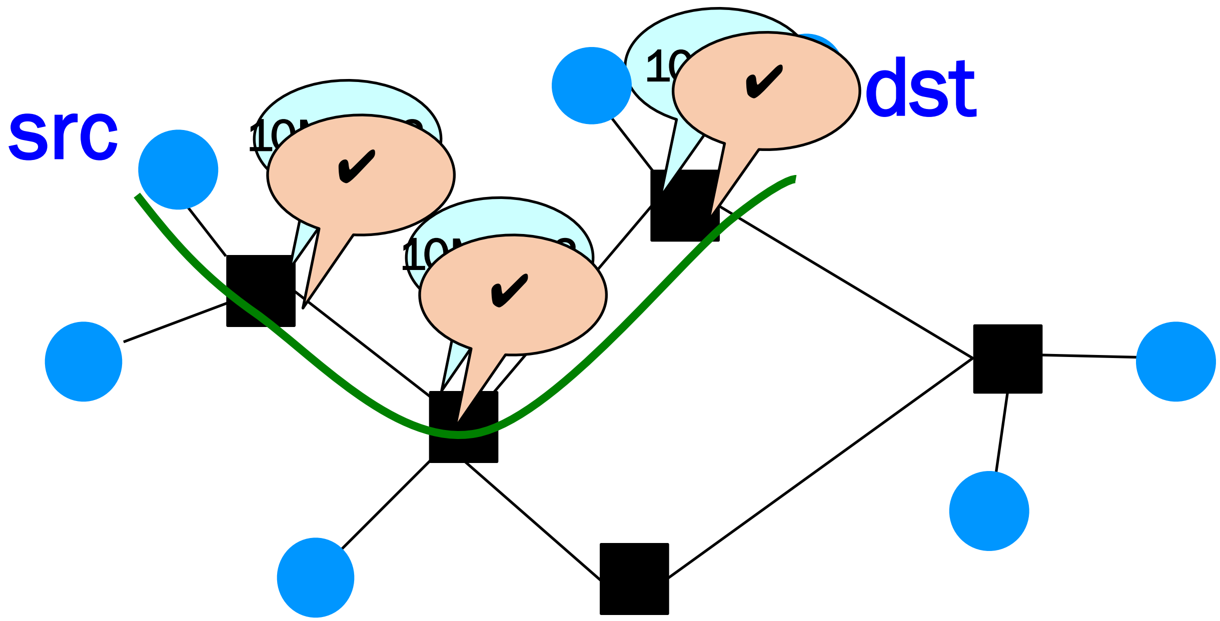
Two approaches to sharing

- Packet switching
  - packets treated on demand
  - admission control: <u>per packet</u>

- Circuit switching
  - resources reserved per active "connection"
  - admission control: <u>per connection</u>

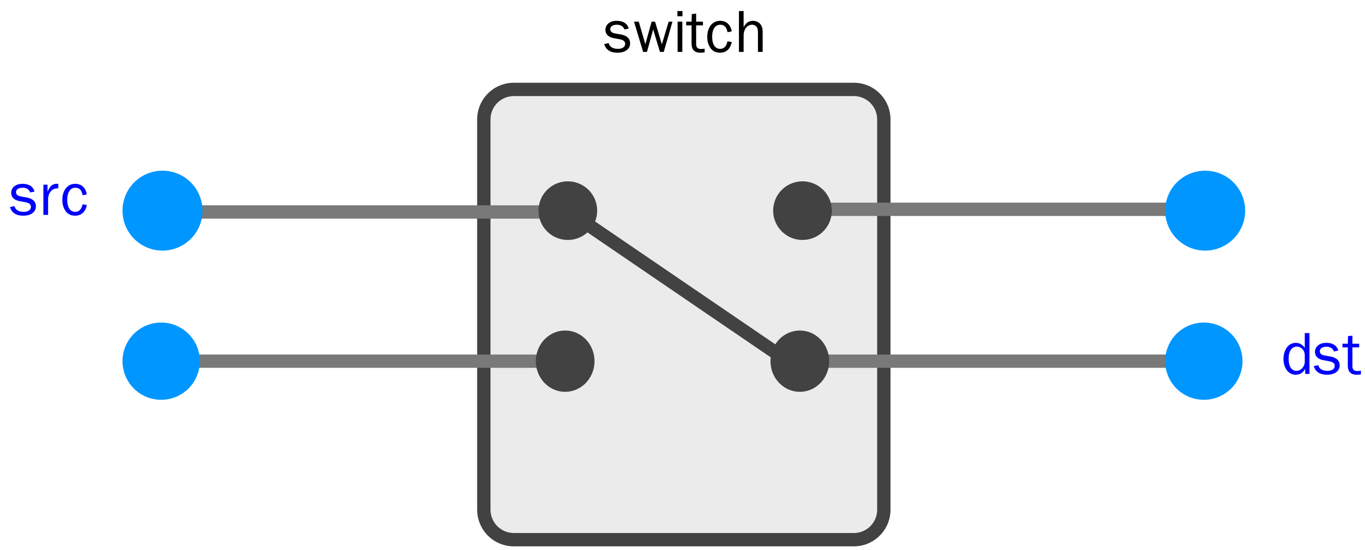
- A hybrid: virtual circuits
  - emulating circuit switching with packets (see text)

Circuit Switching



(1) src sends a reservation request to dst
(2) Switches "establish a circuit"
(3) src starts sending data
(4) src sends a "teardown circuit" message

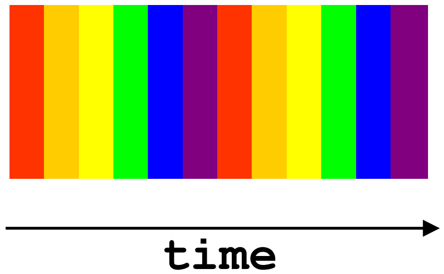
Circuit Switching



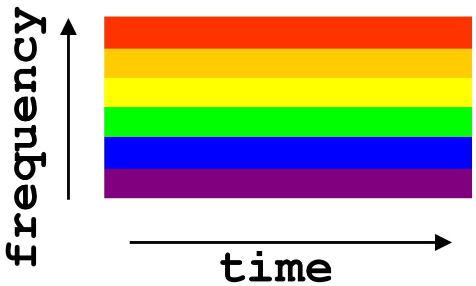
Reservation establishes a “circuit” within a switch

Many kinds of "circuits"

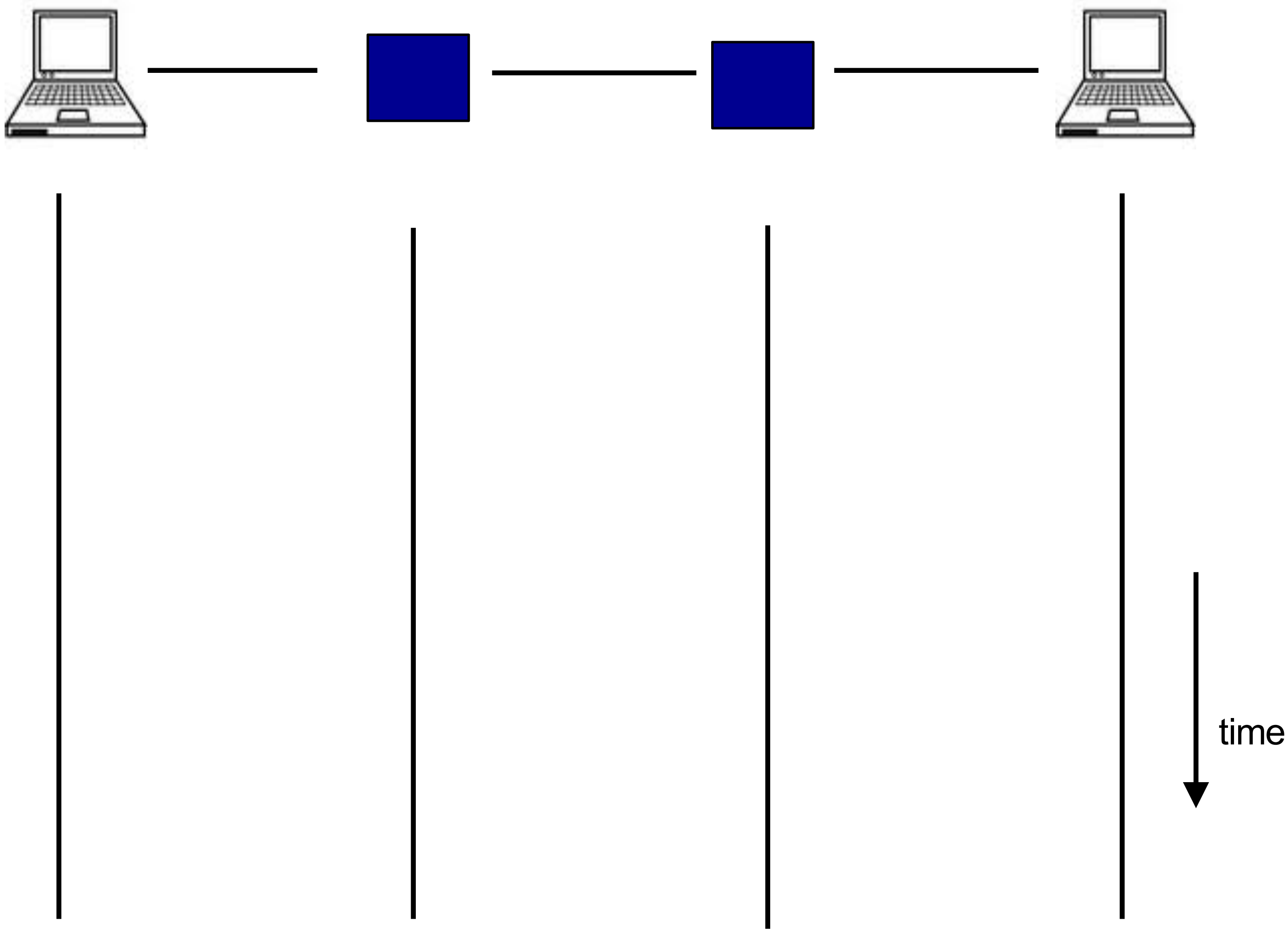
- Time division multiplexing
  - divide time in time slots
  - separate time slot per circuit



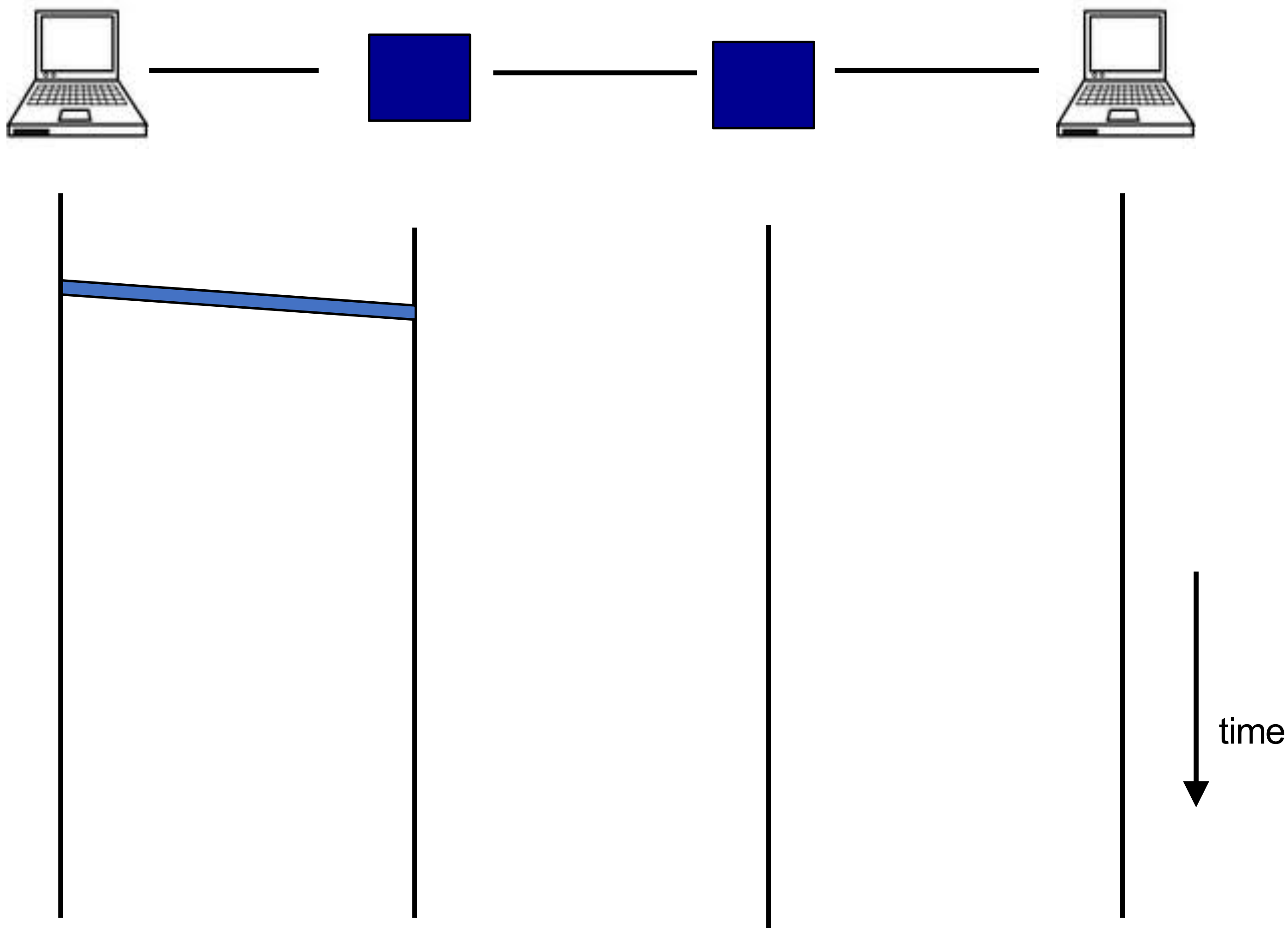
- Frequency division multiplexing
  - divide frequency spectrum in frequency bands
  - separate frequency band per circuit



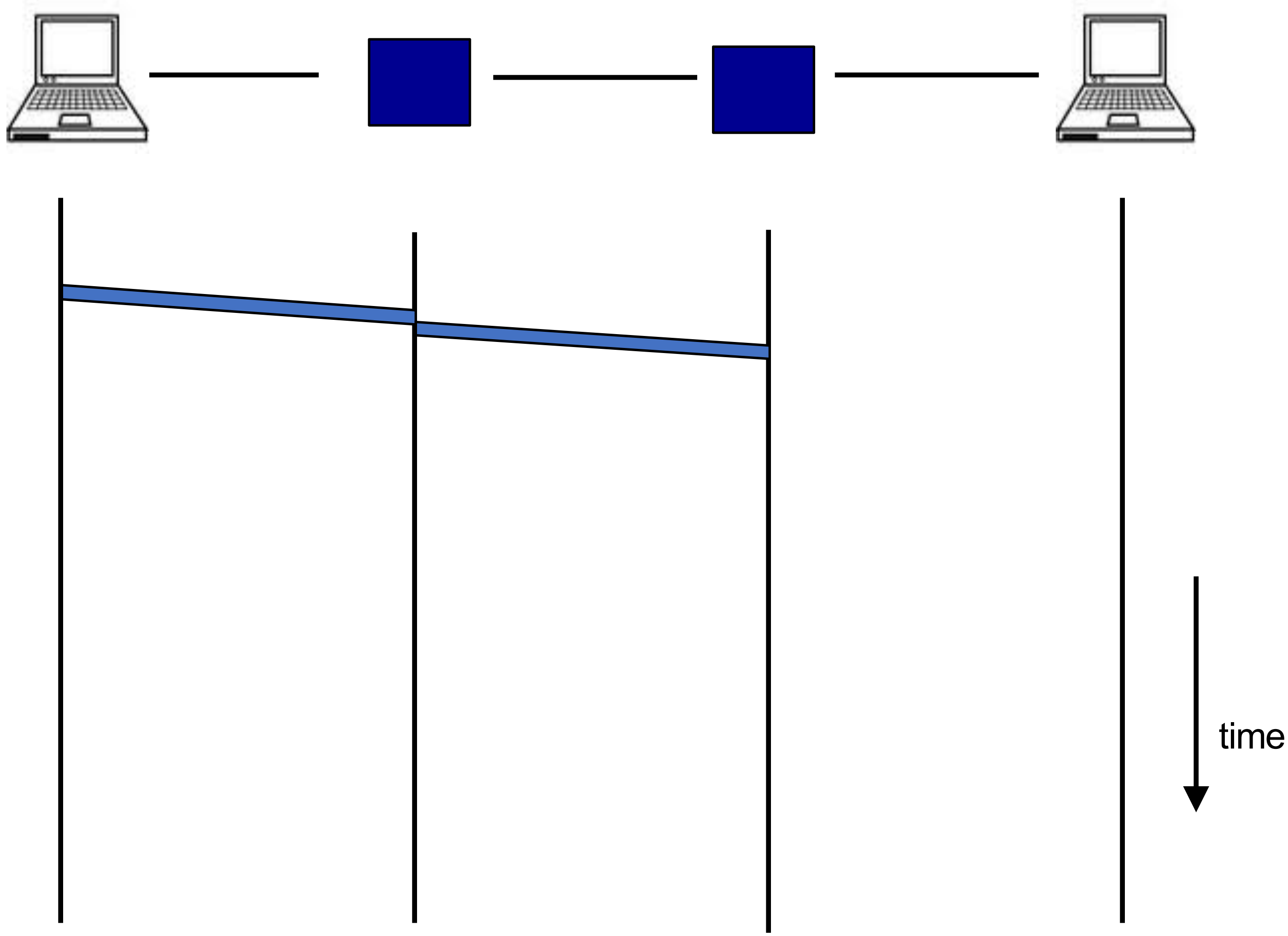
Timing in Circuit Switching



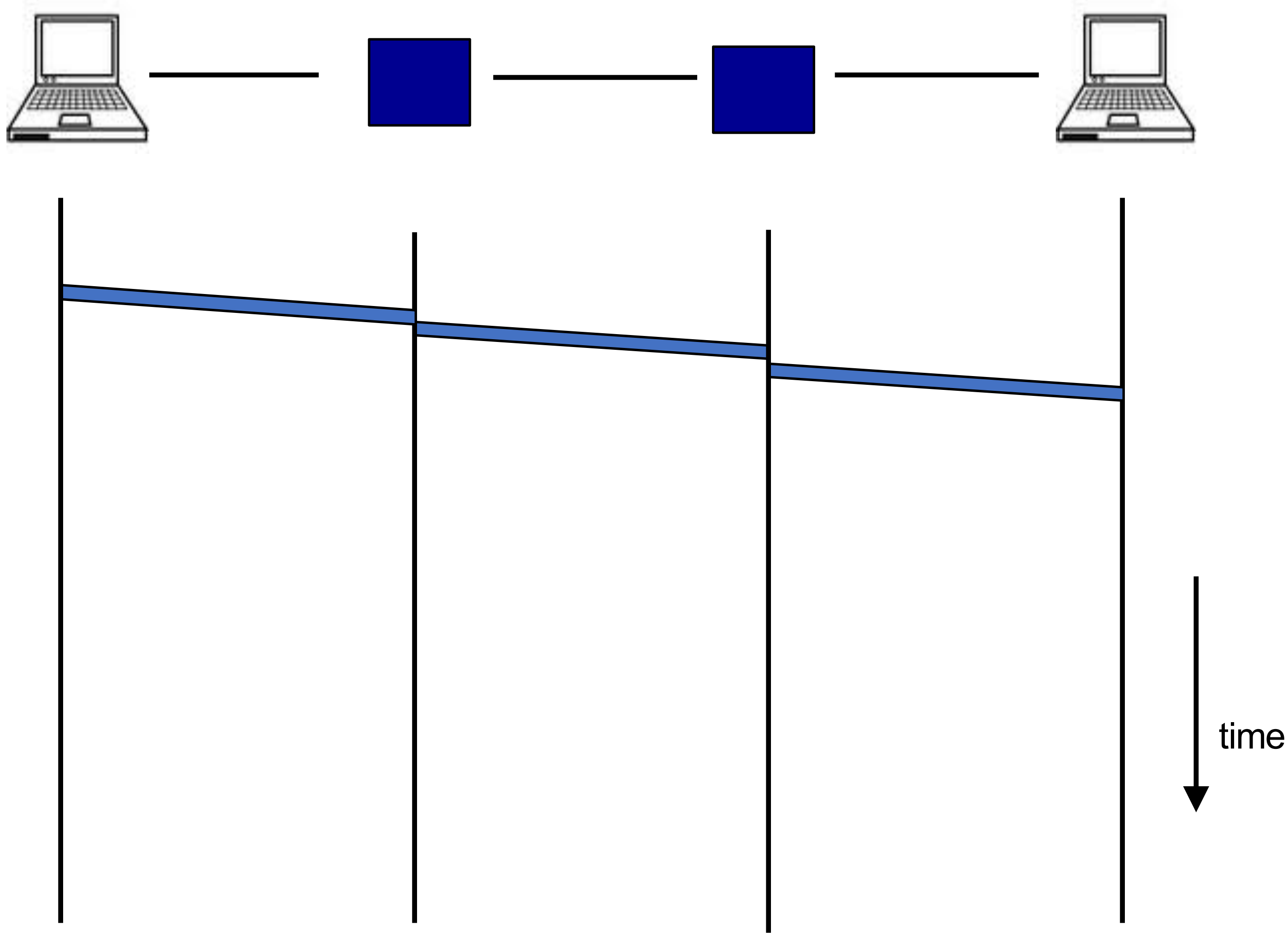
Timing in Circuit Switching



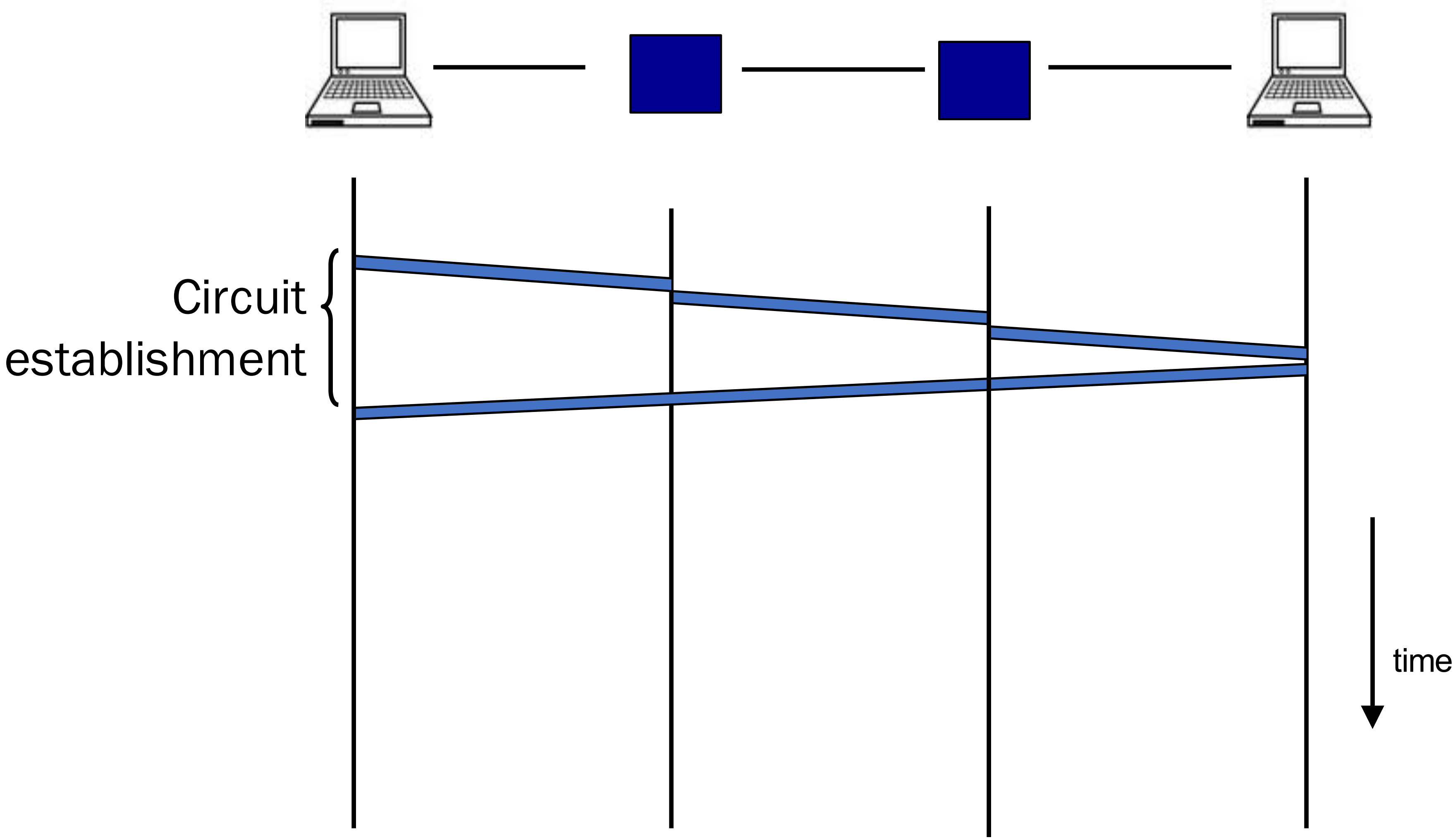
Timing in Circuit Switching



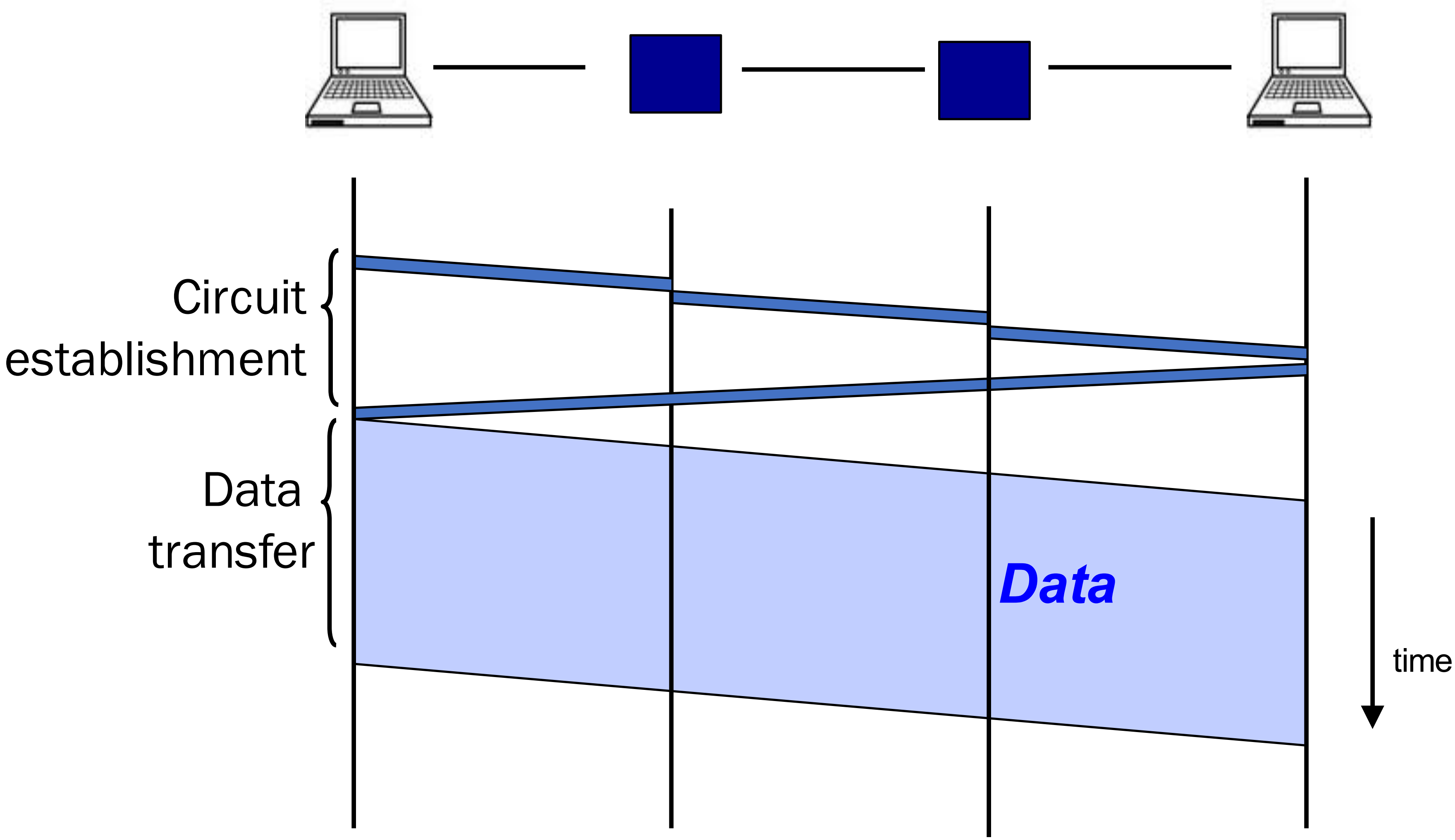
Timing in Circuit Switching



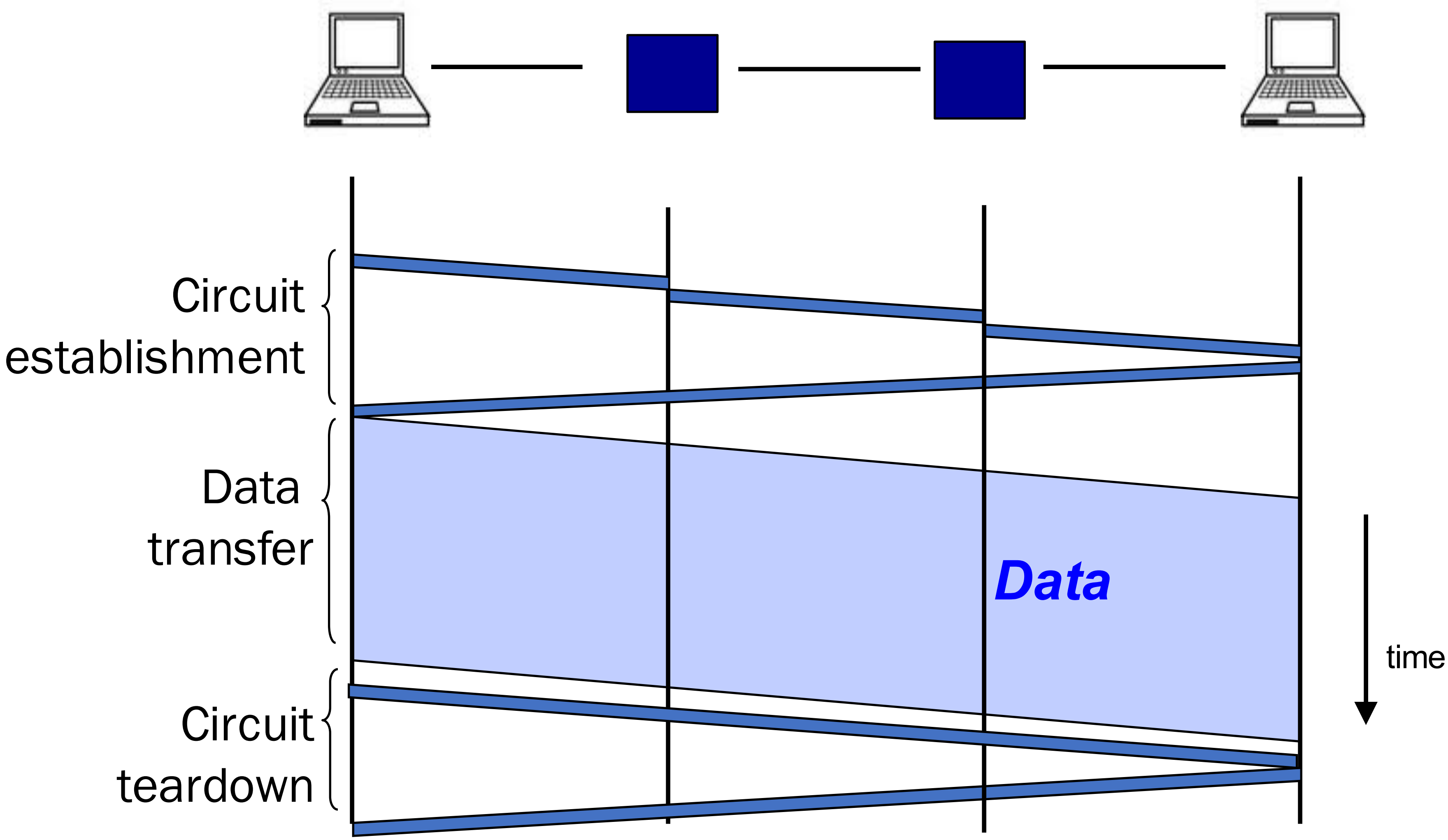
Timing in Circuit Switching



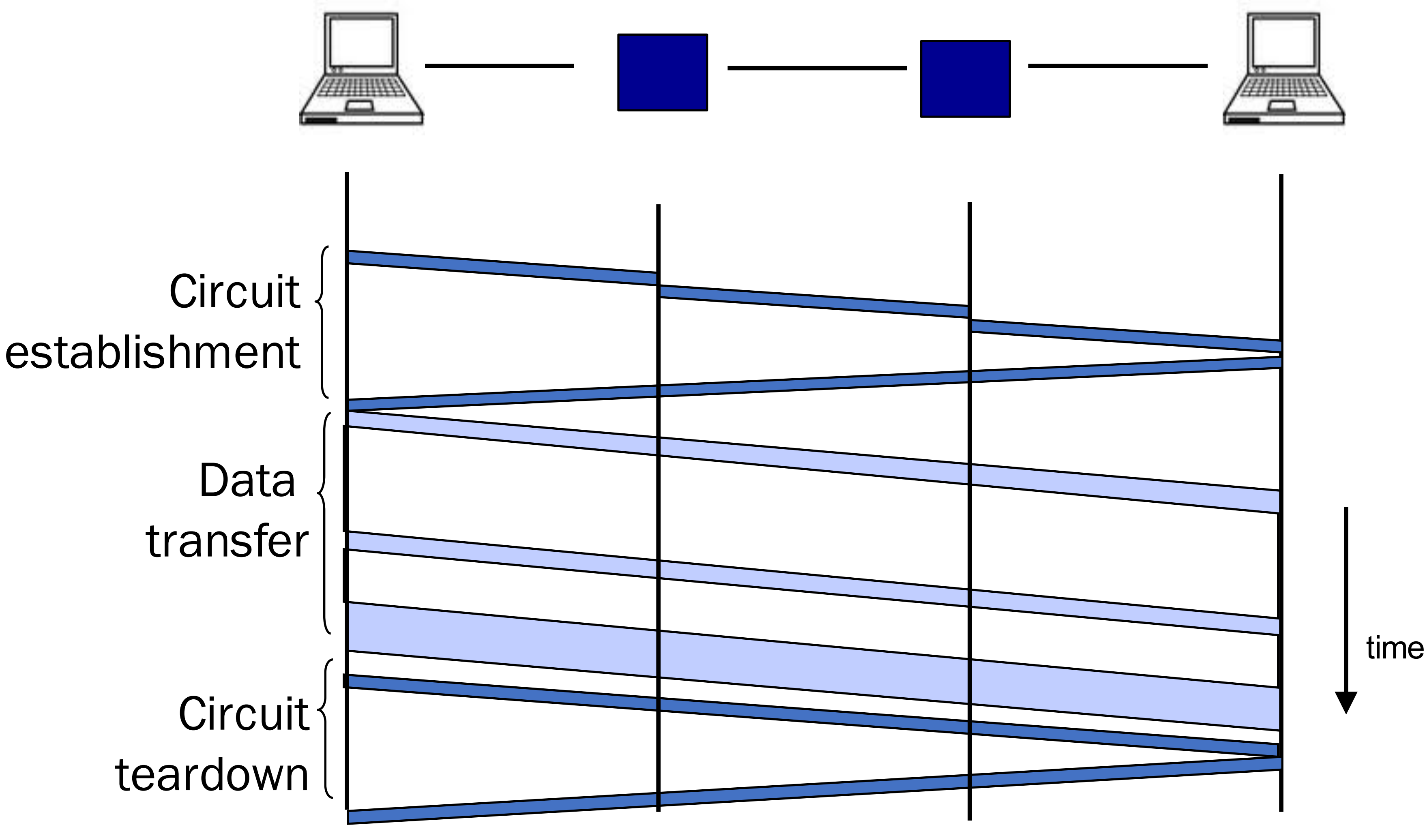
Timing in Circuit Switching



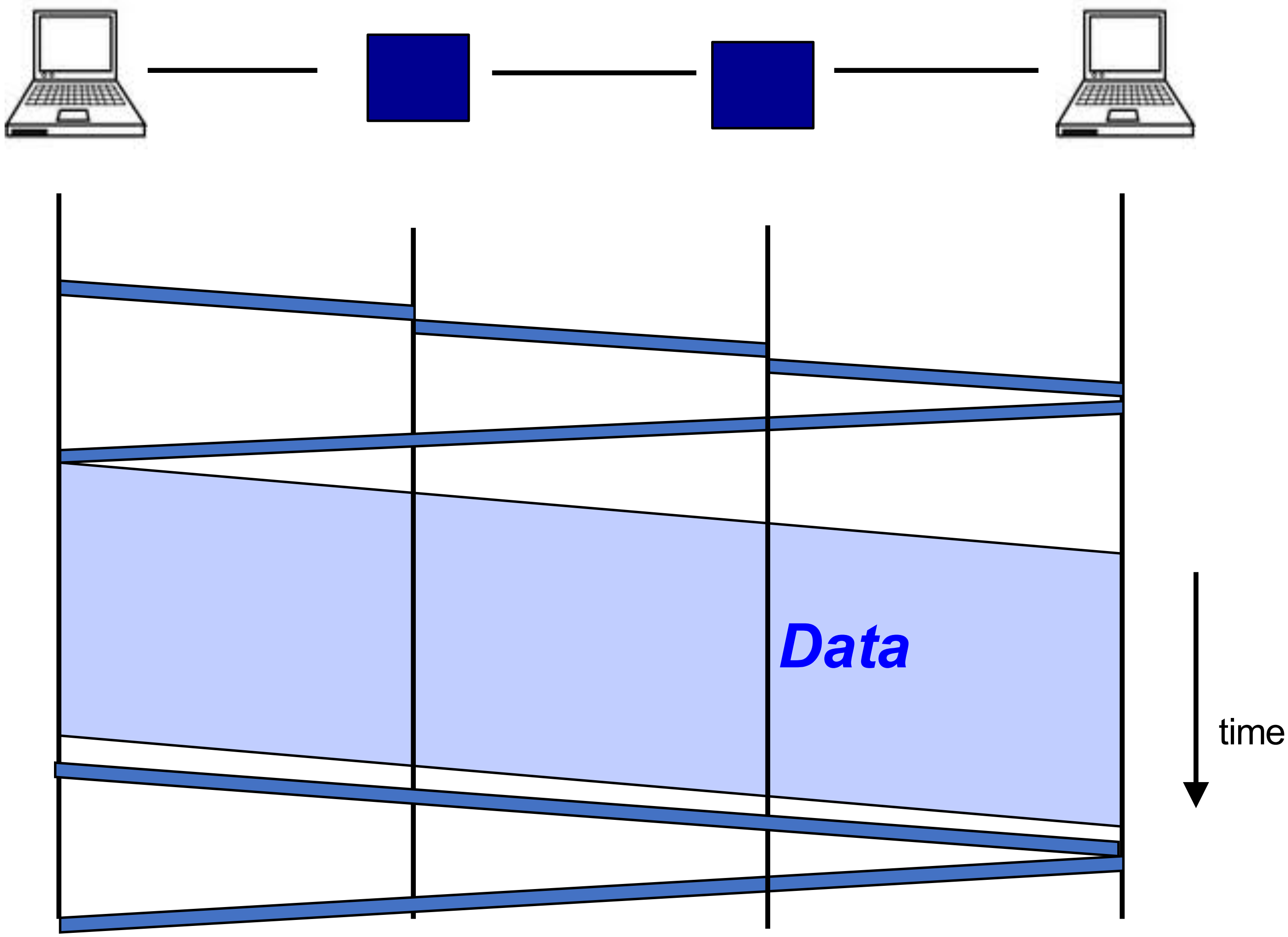
Timing in Circuit Switching



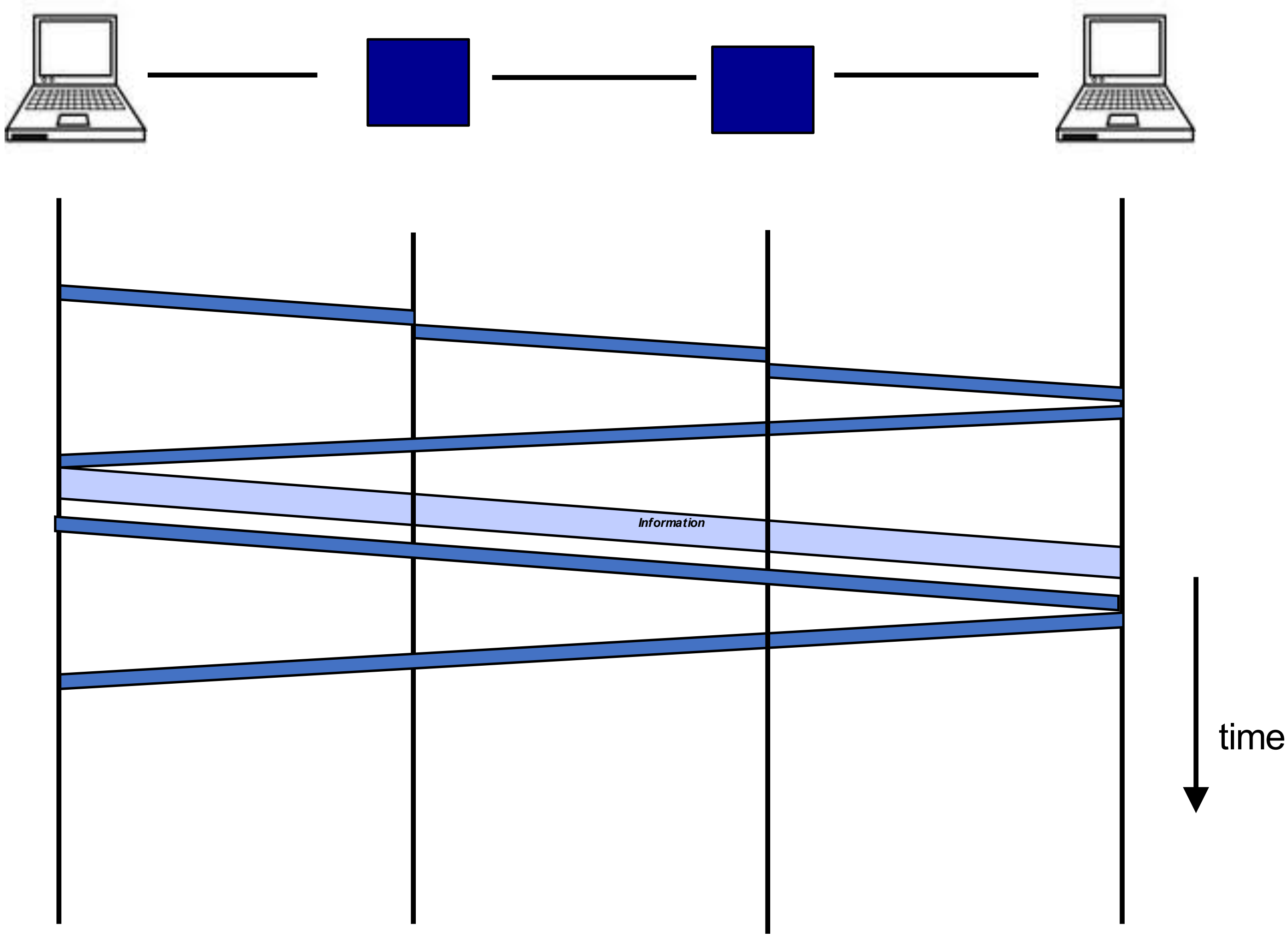
Timing in Circuit Switching



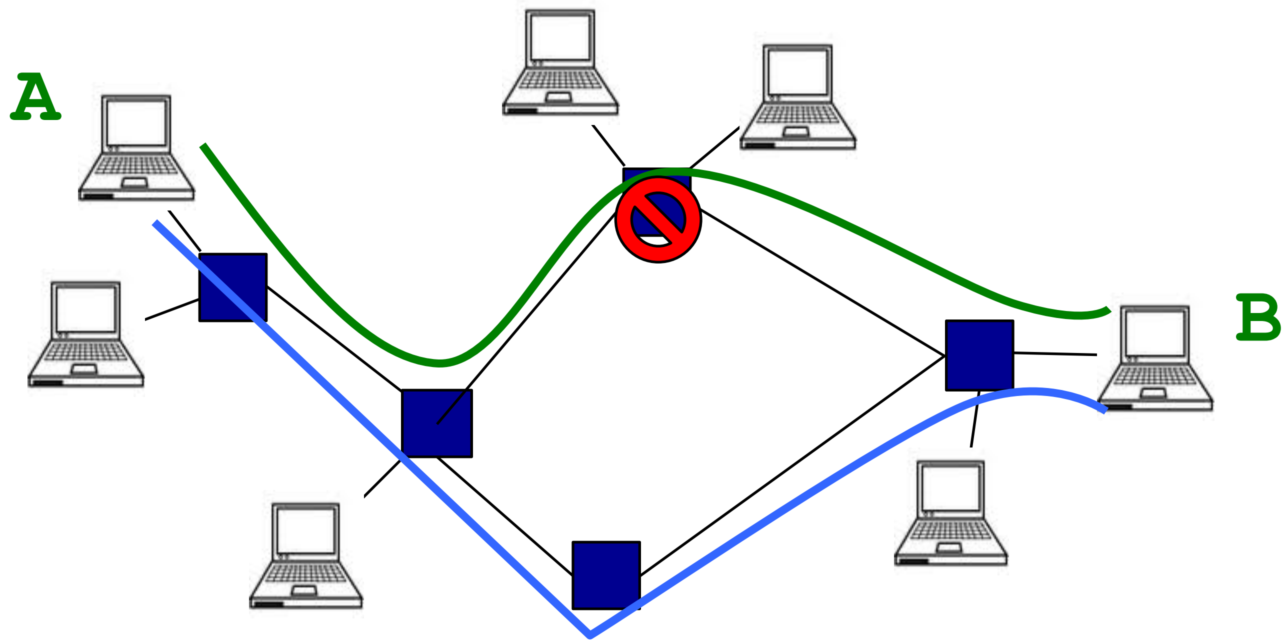
Timing in Circuit Switching



Timing in Circuit Switching



Circuit Switching

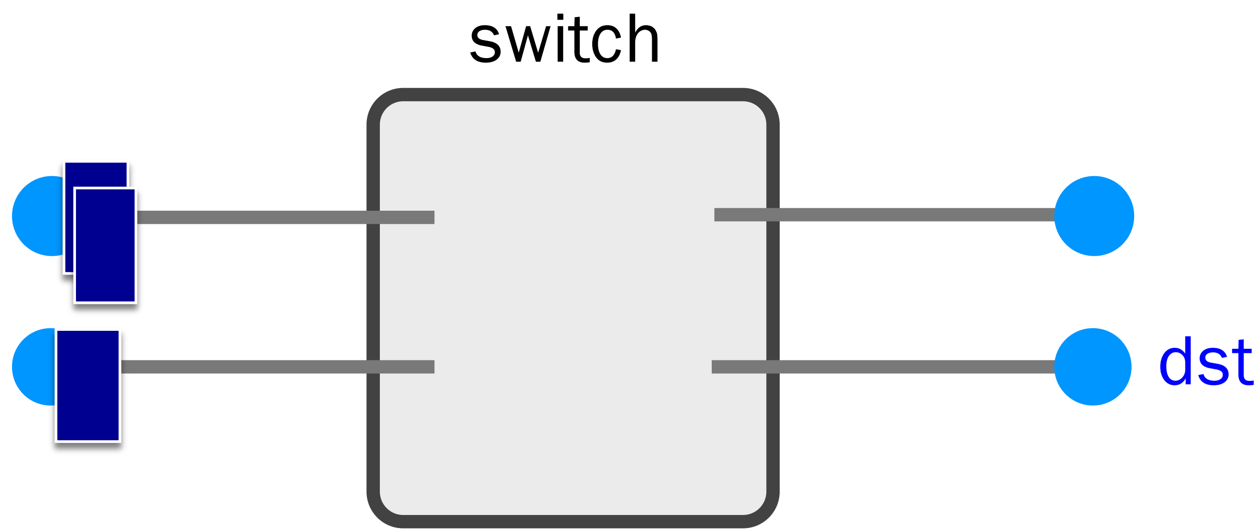


Circuit switching doesn’t “route around trouble”

Circuit Switching

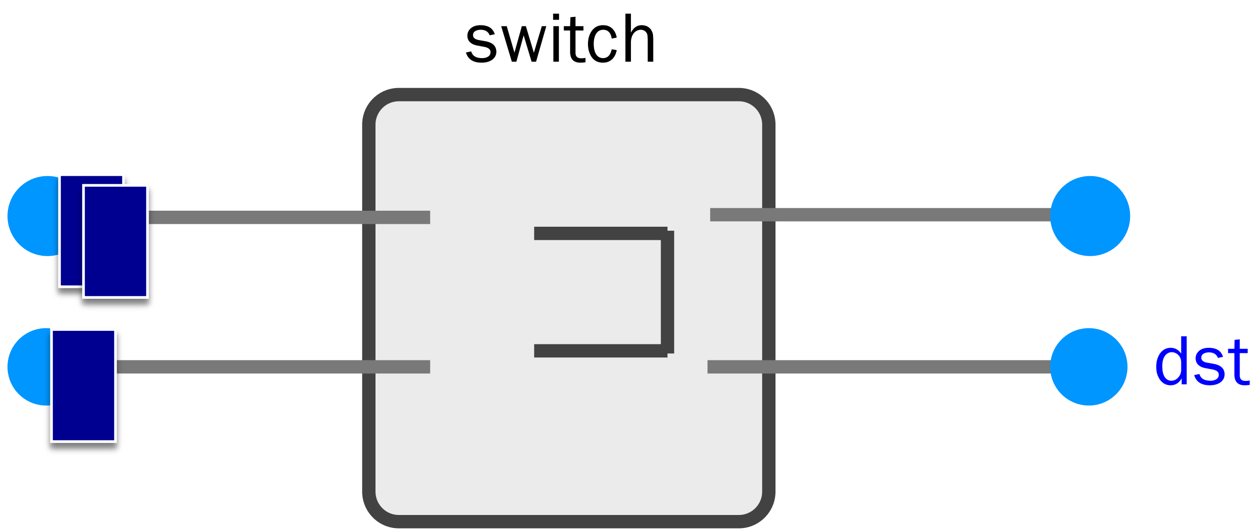
- Pros
  - predictable performance
  - simple/fast switching (once circuit established)
- Cons
  - complexity of circuit setup/teardown
  - inefficient when traffic is bursty
  - circuit setup adds delay
  - switch fails → its circuit(s) fails

Packet switching



Each packet contains destination (dst)
Each packet treated independently

Packet switching



Each packet contains destination (dst)
Each packet treated independently

With buffers to absorb transient overloads

Packet Switching

- Pros
  - efficient use of network resources
  - simpler to implement
  - robust: can "route around trouble"
- Cons
  - unpredictable performance
  - requires buffer management and congestion control

- *On-demand or reserve?*

Today

- What is a network made of?
- How is it shared?
    - **will assume packet switching from now on**

- Sockets
- How do we evaluate a network?

Time Travel to Sockets

Quick tutorial of Sockets

http://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html

Sockets

- Sockets
  - Basic API for programming network applications
  - Should have been covered (some) in CS 15400

Overview: Communication via Sockets

- Client
  1. Create a socket with the **socket()** system call
  2. Connect the socket to the address of the server using the **connect()** system call
  3. Send and receive data. The simplest is to use the **read()** and **write()** system calls.

- Server
  1. Create a socket with the **socket()** system call
  2. Bind the socket to an address using the **bind()** system call.
  3. Listen for connections with the **listen()** system call
  4. Accept a connection with the **accept()** system call (blocks until a client connects with the server)
  5. Send and receive data

Application Programming Interface

- Application interfaces to protocols can be
  - *Loosely specified*
  - *Specified precisely*
- An API is a set of function signatures
- It is an interface that may have multiple implementations
  - *System calls vs. libraries*
  - *Unix, Windows*
- An API can define dedicated abstractions/functions or reuse existing ones

Socket API

- API used to access the network

- A socket is an abstraction of a communication end-point
  - *Can be used to access different network protocols (not only TCP/IP)*

- A socket's characteristics are determined by calling specific functions

- A socket can be accessed as a file
  - *Some of the I/O API calls are similar (e.g., read, write, close)*

Creating a Socket

- **`int socket(int domain, int type, int protocol)`**
  - domain: **`PF_UNIX, PF_INET, PF_INET6`**
  - type: **`SOCK_STREAM, SOCK_DGRAM`**
  - protocol: normally 0, can be other values if there are multiple protocols available (**`IPPROTO_TCP, IPPROTO_UDP`**)
  - Return value: the file descriptor of the socket or -1 in case of errors

- UDP and TCP accessed using the same primitives but semantics are different

Protocol 0 = IP
UDP or TCP chosen according to type

Socket Domain & Type

- Two address domains:
    - **Unix** domain: when two processes sharing a common file system to communicate with each other
    - **Interne**t domain: when two processes running on any two hosts on the Internet communicate.
- Two socket types:
    - **Stream**: continuously stream of data (TCP)
    - **Datagram**: message as a block (UDP)

Socket Addresses

Really just a wrapper that provides format of data

```
struct sockaddr {
  u_short sa_family; /* family */
  char sa_data[14];  /* address data */
  };

struct sockaddr_in {     /* a TCP endpoint */
  u_int16_t sin_family; /* address family: AF_INET */
  u_int16_t sin_port;   /* port in network byte order */
  struct in_addr sin_addr; /* internet address */
};
```

Internet namespace, includes address and port

```
struct in_addr {
  u_int32_t s_addr; /* address in network byte order */
};
```

Headers defined In sys/socket.h

Socket Descriptors

- Similar to file descriptors
  - *Internal data structure specifies*
    - *Protocol used (or family – PF_INET)*
    - *Type of service (connectionless or connection-oriented)*
    - *IP addresses involved*
    - *Ports involved*

Essentially a "handle" on the socket, which can be used to do things like "accept."

- Per-process resource

Overview: Communication via Sockets

- Client
  1. Create a socket with the **socket()** system call
  2. Connect the socket to the address of the server using the **connect()** system call
  3. Send and receive data. The simplest is to use the **read()** and **write()** system calls.

- Server
  1. Create a socket with the **socket()** system call
  2. Bind the socket to an address using the **bind()** system call.
  3. Listen for connections with the **listen()** system call
  4. Accept a connection with the **accept()** system call (blocks until a client connects with the server)
  5. Send and receive data

Creating Connections

- Clients use **`connect()`** to open a connection to a specific IP address/port

- **`int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen)`**
  - **`sockfd`**: socket descriptor
  - **`serv_addr`**: destination address
  - **`addrlen`**: size of the structure
  - Return value: 0 in case of success, -1 in case of errors

Connecting with TCP and UDP

- TCP
  - ***`connect()`*** *starts the three-way handshake*
- UDP
  - *Nothing really happens, but the socket can only be used to send/receive datagrams to/from the specified address*

Overview: Communication via Sockets

- Client
  1. Create a socket with the **socket()** system call
  2. Connect the socket to the address of the server using the **connect()** system call
  3. Send and receive data. The simplest is to use the **read()** and **write()** system calls.

- Server
  1. Create a socket with the **socket()** system call
  2. Bind the socket to an address using the **bind()** system call.
  3. Listen for connections with the **listen()** system call
  4. Accept a connection with the **accept()** system call (blocks until a client connects with the server)
  5. Send and receive data

Binding Sockets to Addresses

- A socket can be bound to a specific IP address and port
- **`int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen)`**
  - ***`sockfd`***: *socket descriptor*
  - ***`my_addr`***: *socket address (usually of sockaddr_in type)*
  - ***`addrlen`***: *address structure length*
  - *Return value: 0 in case of success, -1 in case of error*
- Same semantics for TCP and UDP

Passive Connection-Oriented Sockets

- If a socket is used to receive TCP connections, it is necessary to bind it to an address and to specify that the socket is passive
- **`int listen(int s, int backlog)`**
  - **`s`**: socket descriptor
  - **`backlog`**: maximum length of the queue of pending connections (used to be the number of open/half-open connections, now only the open ones that are ready to be accepted are counted)
  - Return value: 0 in case of success, -1 in case of error

Why don't we bind for active sockets?

Accepting Connections

- **`int accept(int s, struct sockaddr *addr, socklen_t *addrlen)`**
  - **`s`**: socket descriptor
  - **`addr`**: structure that will be filled with the parameters of the **client** (maybe NULL)
  - **`addrlen`**: length of the structure in input, it is filled with the actual size of the data returned
  - Return value: a new socket file descriptor in case of success, -1 in case of errors

Accepting TCP Connections

- A call to **`accept()`** blocks the caller until a request is sent
- When a connection is made, **`accept()`** returns a new socket, associated with a specific client (virtual circuit)
- A virtual circuit is identified by:
  <**`srcIP, srcPort, dstIP, dstPort`**>
- There are no two identical virtual circuits at one time in the whole Internet

Sending Data On A Connected Socket

- **`int send(int s, const void *msg, size_t len, int flags)`**
  - s: connected socket
  - **`msg`**: message
  - **`len`**: size of message
  - flags: e.g., **`MSG_DONTWAIT`**
  - Return value: # of characters sent, or -1 in case of errors

Receiving Data On A Connected Socket

- **`int recv(int s, void *buf, size_t len, int flags)`**
  - s: connected socket
  - **`buf`**: buffer to store the message
  - **`len`**: max size of message in input, actual size of message in output
  - flags: e.g., **`MSG_PEEK`** allows one to peek at a message without removing it from the incoming data queue
  - Return value: the number of characters read, or -1 in case of errors

- This call will block the process if there is no data

Closing A Socket

- **`int close(int fd);`**
- **`fd`**: socket descriptor
- Return value: 0 in case of success, -1 in case of errors

- **`int shutdown(int s, int how)`**
- Can be used to close a socket partially
- s: connected socket
- how:
- **`SHUT_RD`**, further receptions are disallowed
- **`SHUT_WR`**, further transmissions are disallowed
- **`SHUT_RDWR`**, further receptions and transmissions are disallowed
- Return value: 0 in case of success, -1 in case of errors

Closing Semantics

- TCP
  - Close: FIN/ACK in both directions
  - Shutdown: FIN/ACK in one direction
- UDP
  - Close: Don't send anything. Just deallocate structure

Sending/Receiving Data on a Disconnected Socket (for UDP)

UDP does not require accept(). So it need source or destination address information in order to receive or send data.

- `int sendto(int s, const void *msg, size_t len, int flags, const struct sockaddr *to, socklen_t tolen)`

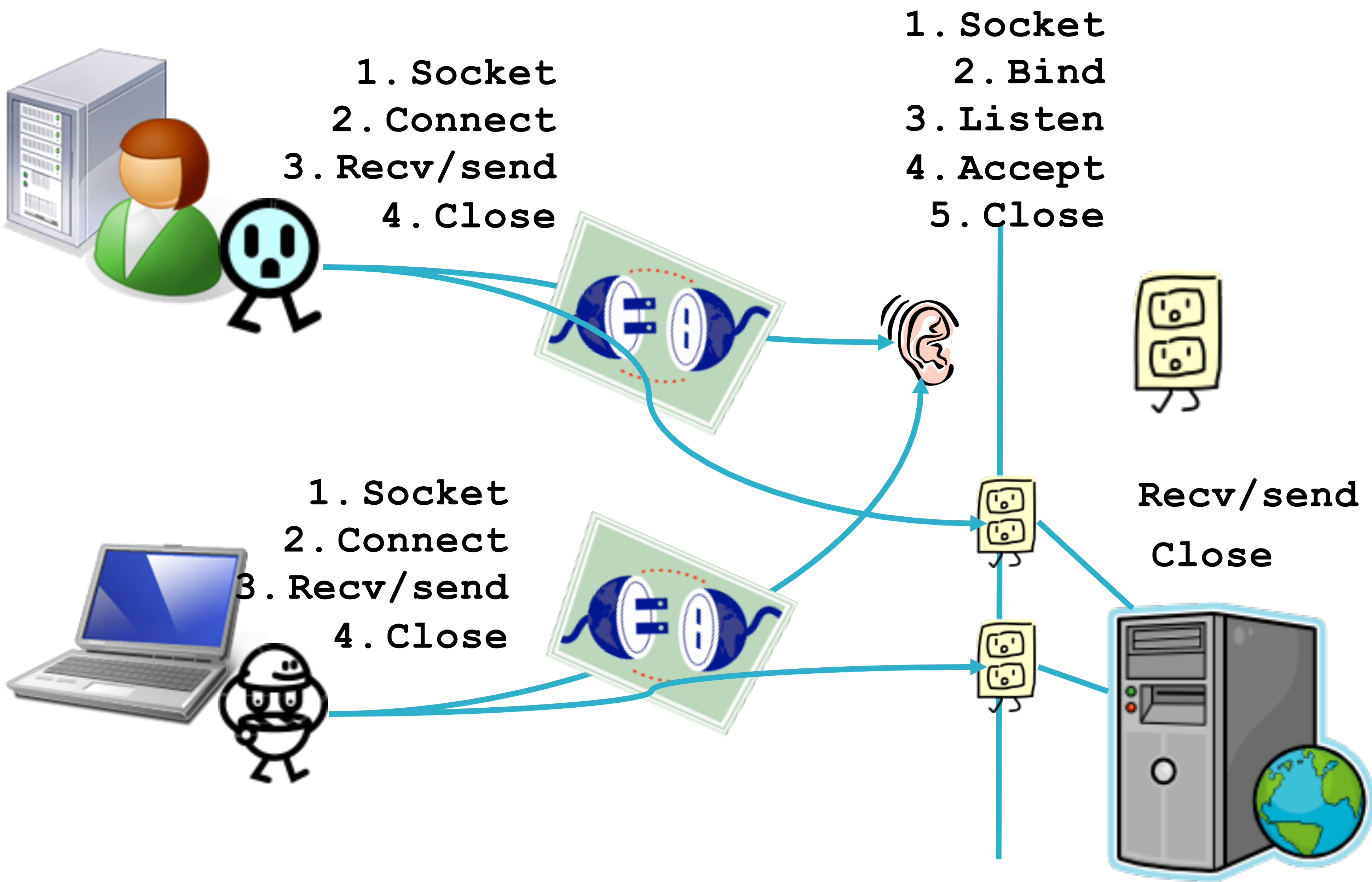
- `int recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen)`

Data Exchange in TCP and UDP

- TCP
  - recv/read will return a chunk of data
  - Not necessarily the data sent by means of a single send/write operation on the other side
  - THIS IS IMPORTANT! (semantics of a byte stream, not datagrams)

- UDP
  - recv/read will always return a datagram
  - If message size > buffer, fills buffer + discards rest

TCP Client and Server



http://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html

```
/* A simple server in the internet domain using TCP
   The port number is passed as an argument */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void error(char *msg)
{
    perror(msg);
    exit(1);
}

int main(int argc, char *argv[])
{
     int sockfd, newsockfd, portno, clilen;
     char buffer[256];
     struct sockaddr_in serv_addr, cli_addr;
     int n;
     if (argc < 2) {
         fprintf(stderr,"ERROR, no port provided\n");
         exit(1);
     }
     sockfd = socket(AF_INET, SOCK_STREAM, 0);
     if (sockfd < 0)
        error("ERROR opening socket");
     bzero((char *) &serv_addr, sizeof(serv_addr));
     portno = atoi(argv[1]);
     serv_addr.sin_family = AF_INET;
     serv_addr.sin_addr.s_addr = INADDR_ANY;
     serv_addr.sin_port = htons(portno);
     if (bind(sockfd, (struct sockaddr *) &serv_addr,
              sizeof(serv_addr)) < 0)
              error("ERROR on binding");
     listen(sockfd,5);
     clilen = sizeof(cli_addr);
     newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen)
     if (newsockfd < 0)
          error("ERROR on accept");
     bzero(buffer,256);
     n = read(newsockfd,buffer,255);
     if (n < 0) error("ERROR reading from socket");
     printf("Here is the message: %s\n",buffer);
     n = write(newsockfd,"I got your message",18);
     if (n < 0) error("ERROR writing to socket");
     return 0;
}
```

http://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void error(char *msg)
{
    perror(msg);
    exit(0);
}

int main(int argc, char *argv[])
{
    int sockfd, portno, n;

    struct sockaddr_in serv_addr;
    struct hostent *server;

    char buffer[256];
    if (argc < 3) {
       fprintf(stderr,"usage %s hostname port\n", argv[0]);
       exit(0);
    }
    portno = atoi(argv[2]);
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");
    server = gethostbyname(argv[1]);
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);
    }
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
         (char *)&serv_addr.sin_addr.s_addr,
         server->h_length);
    serv_addr.sin_port = htons(portno);
    if (connect(sockfd,(struct sockaddr *)&serv_addr,sizeof(serv_addr)) < 0)
        error("ERROR connecting");
    printf("Please enter the message: ");
    bzero(buffer,256);
    fgets(buffer,255,stdin);
    n = write(sockfd,buffer,strlen(buffer));
    if (n < 0)
         error("ERROR writing to socket");
    bzero(buffer,256);
    n = read(sockfd,buffer,255);
    if (n < 0)
         error("ERROR reading from socket");
    printf("%s\n",buffer);
    return 0;
}
```