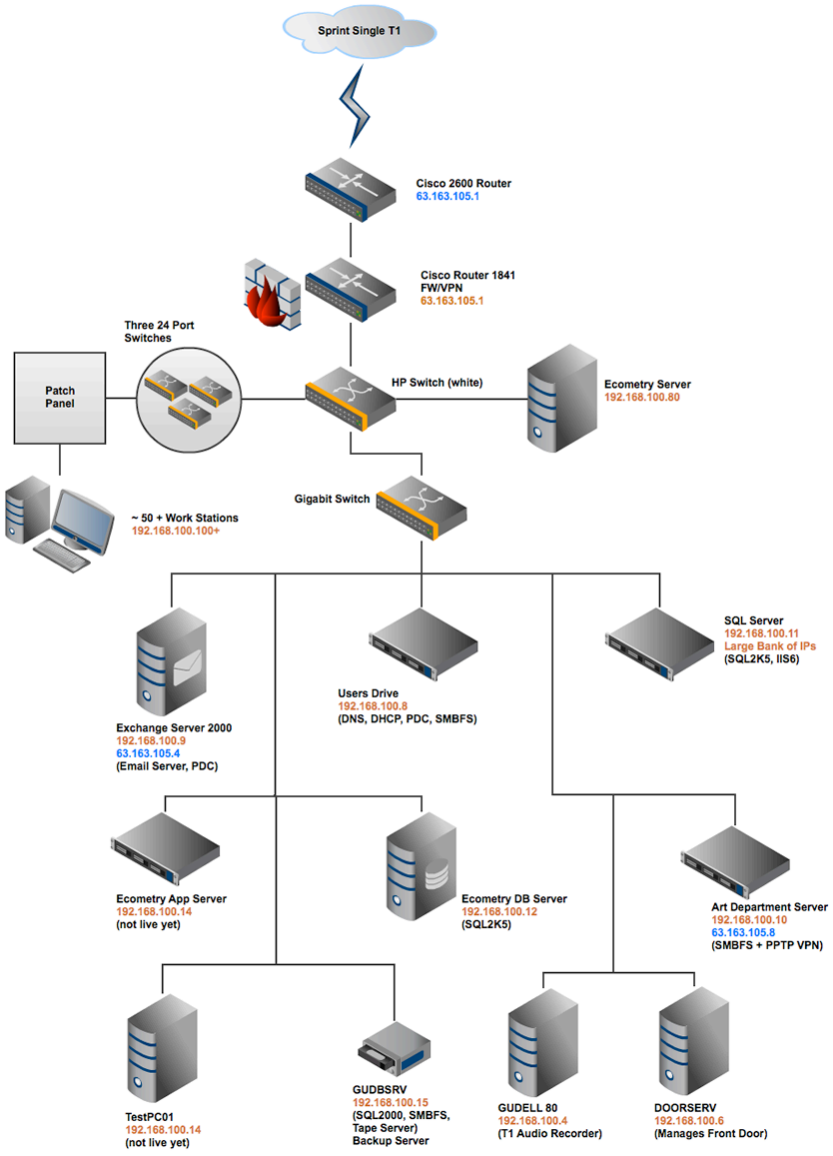
CMPSC 311- Introduction to Systems Programming
Module: Network Programming

Professor Patrick McDaniel
Fall 2013

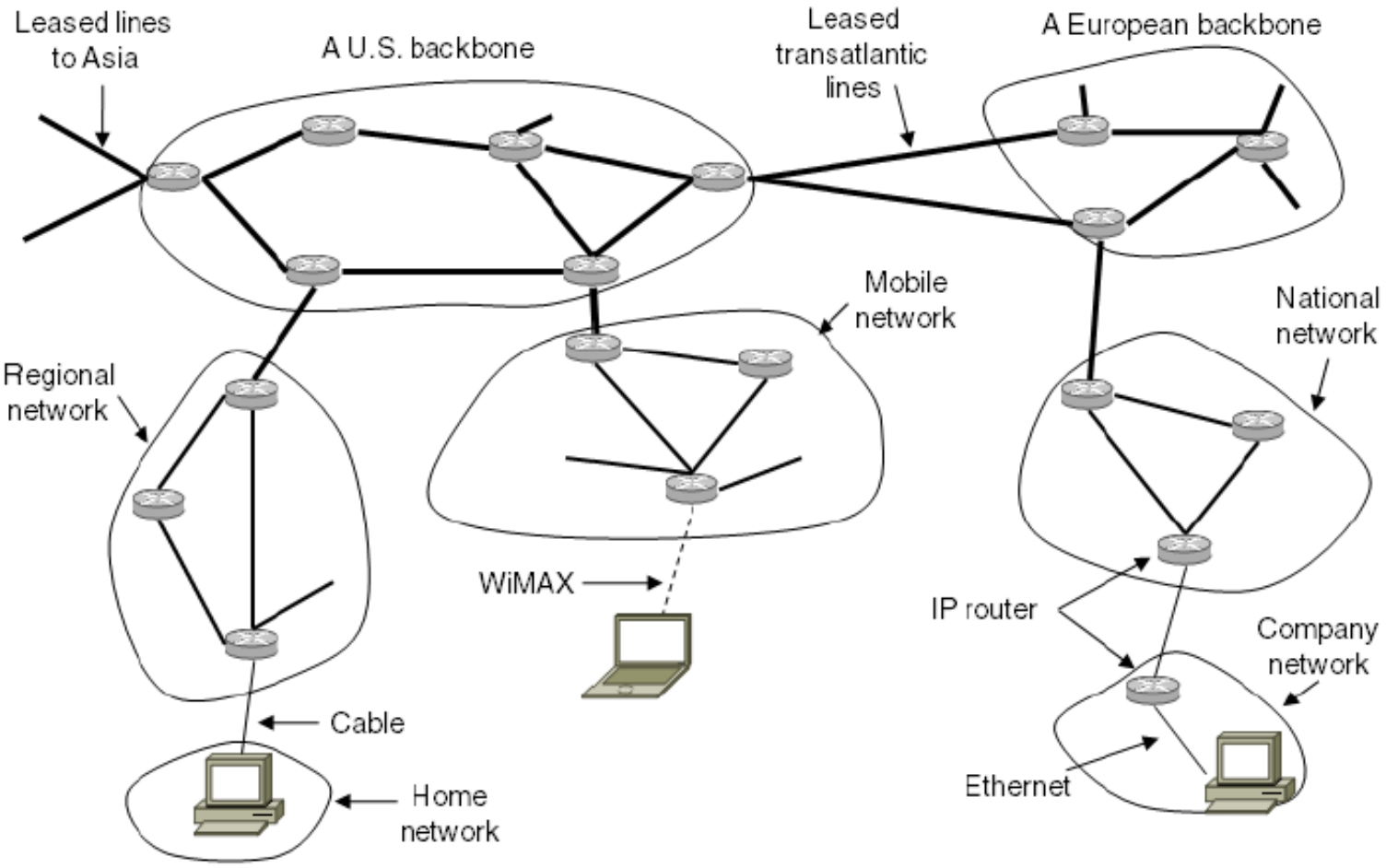
What is a network?

- A network is a collection of computing devices that share a transmission media
  - Traditional wired networks (ethernet)
  - High-speed backbone (fibre)
  - Wireless (radio)
  - Microwave
  - Infrared
- The way the network is organized is called the *network topology*



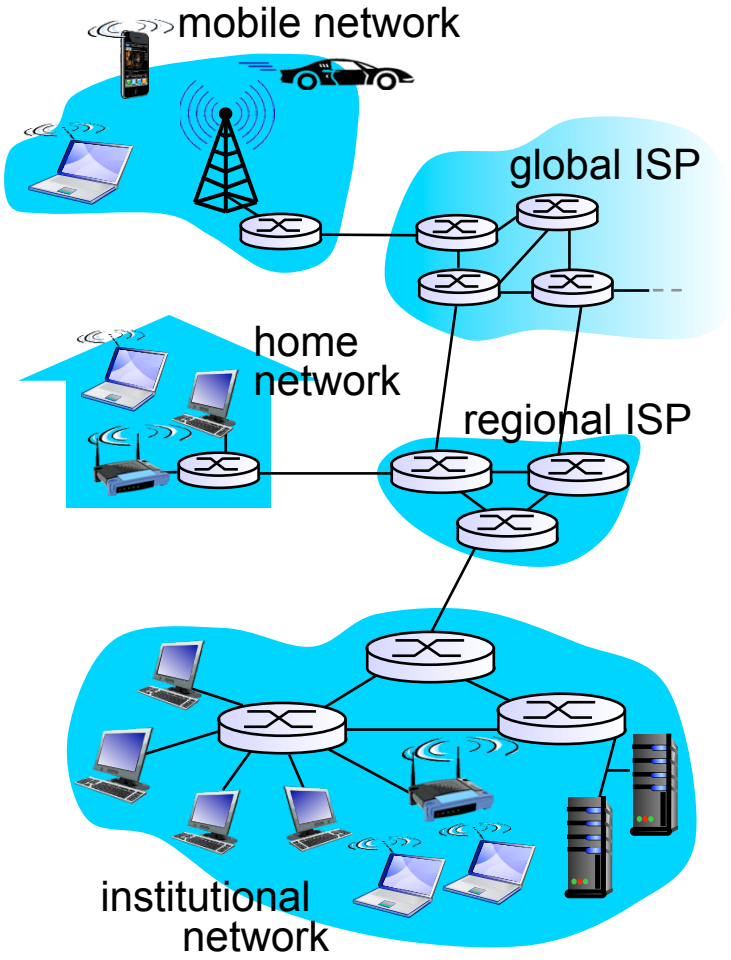
The Internet

The Internet is an interconnected collection of many networks.

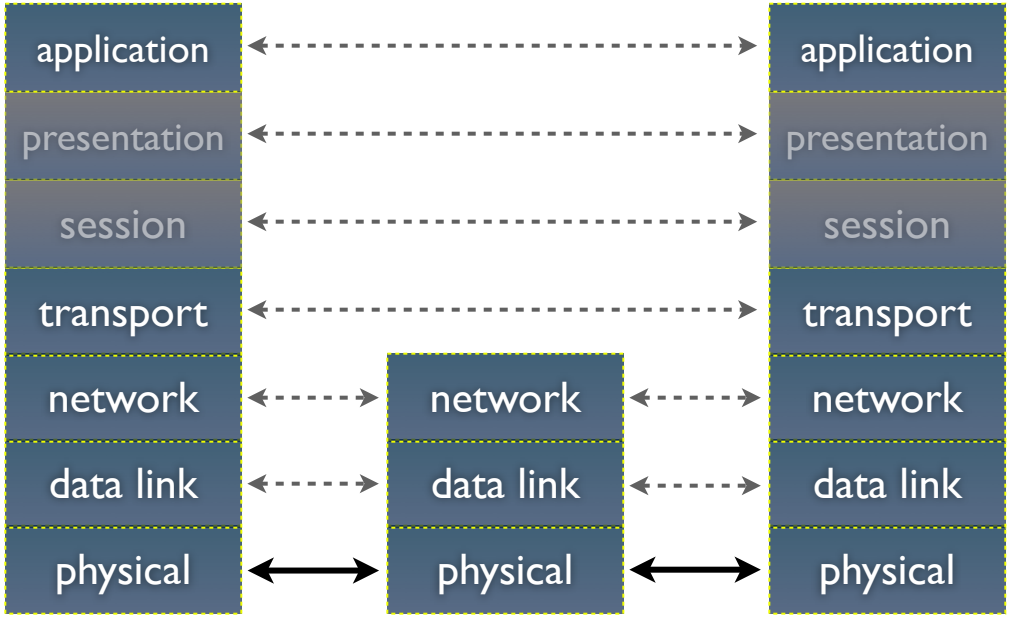


A Closer Look at Network Structure:

- network *edge*:
  - hosts: clients and servers
  - servers often in data centers
- *access networks, physical media*: wired, wireless communication links
- network *core*:
  - interconnected routers
  - network of networks



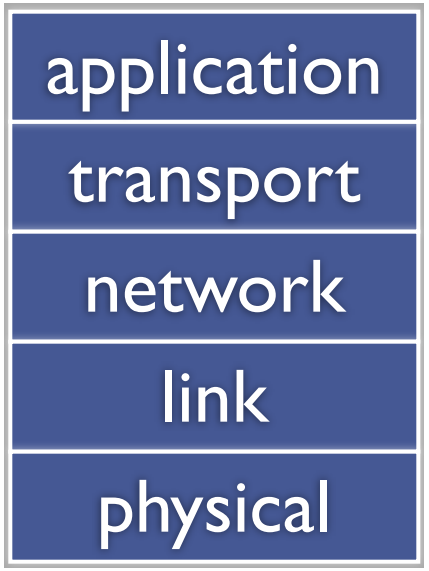
OSI Layer Model



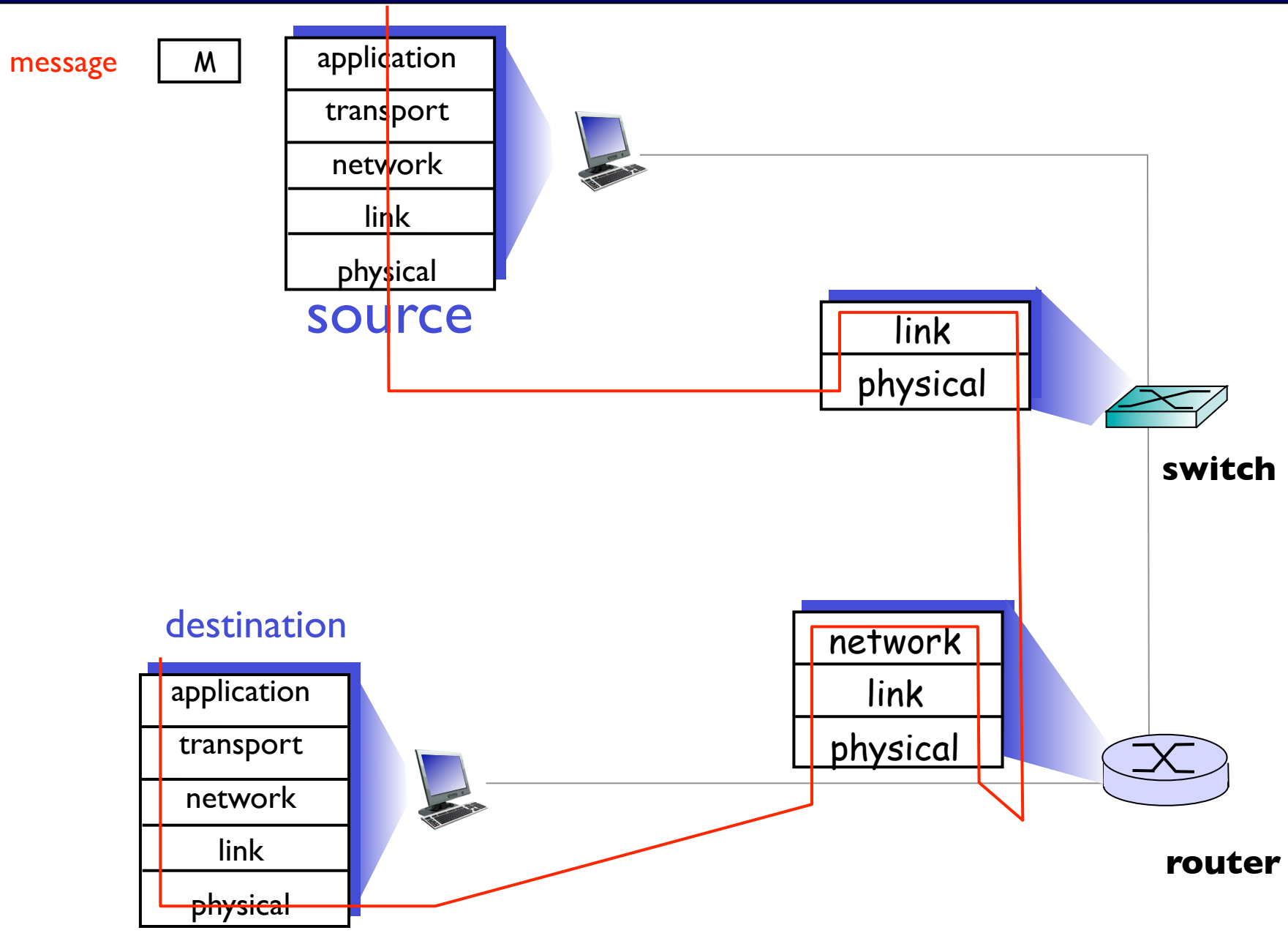
- Application protocols
  - the format and meaning of messages between application entities
  - e.g., HTTP is an application level protocol that dictates how web browsers and web servers communicate
    - HTTP is implemented on top of TCP streams

Internet protocol stack

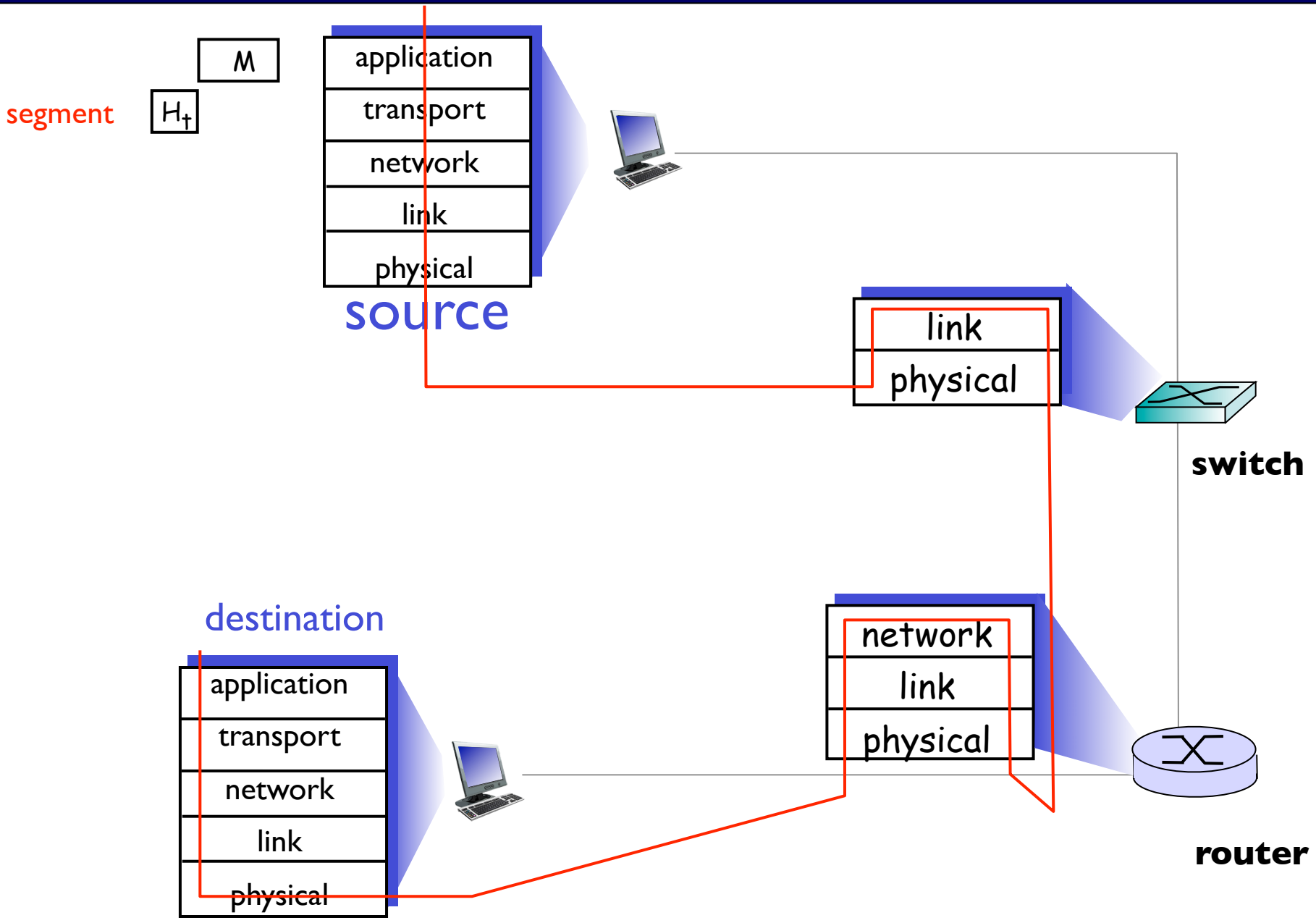
- application: supporting network applications
  - FTP, SMTP, HTTP
- transport: process-process data transfer
  - TCP, UDP
- network: routing of datagrams from source to destination
  - IP, routing protocols
- link: data transfer between neighboring network elements
  - PPP, Ethernet
- physical: bits "on the wire"



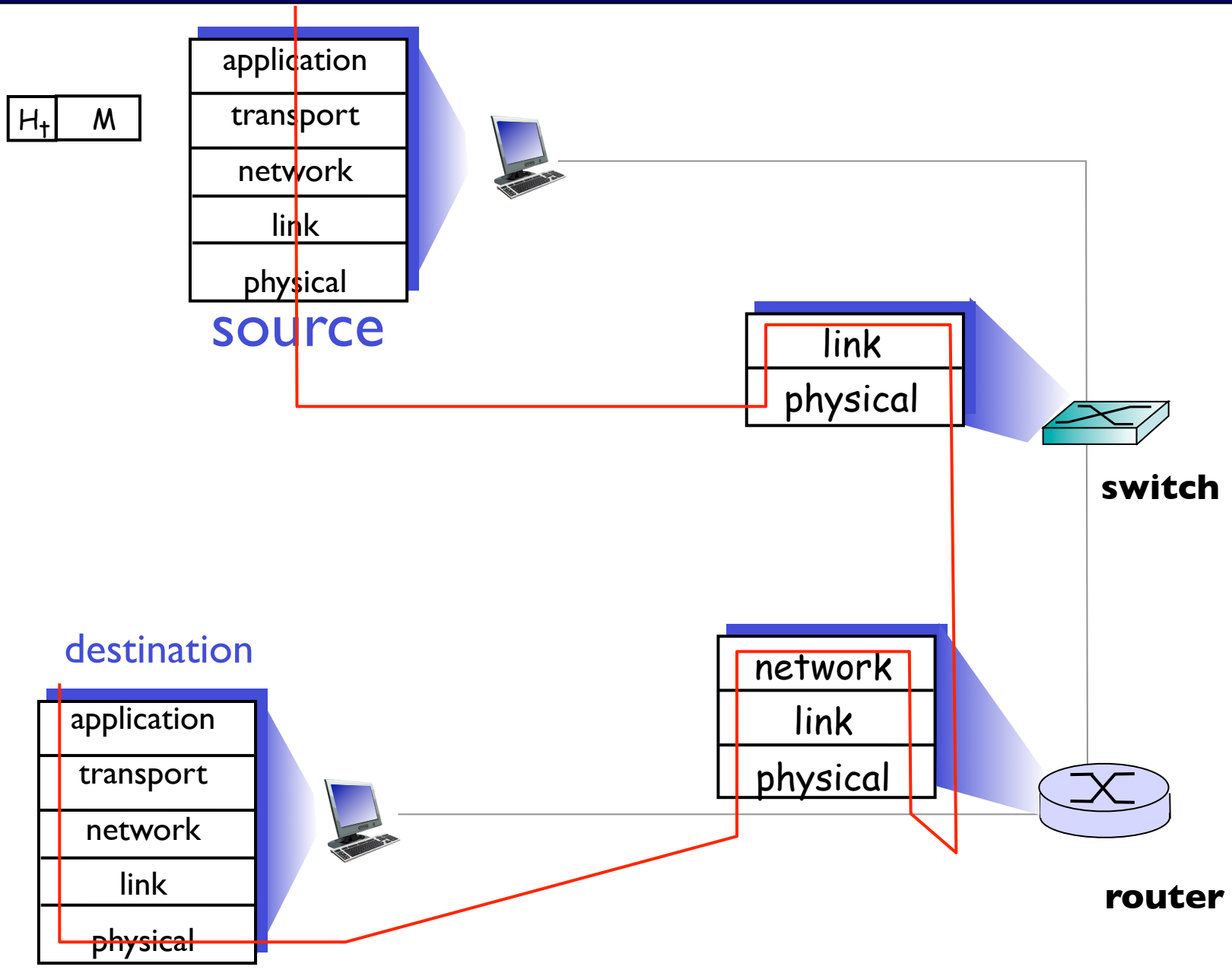
Encapsulation



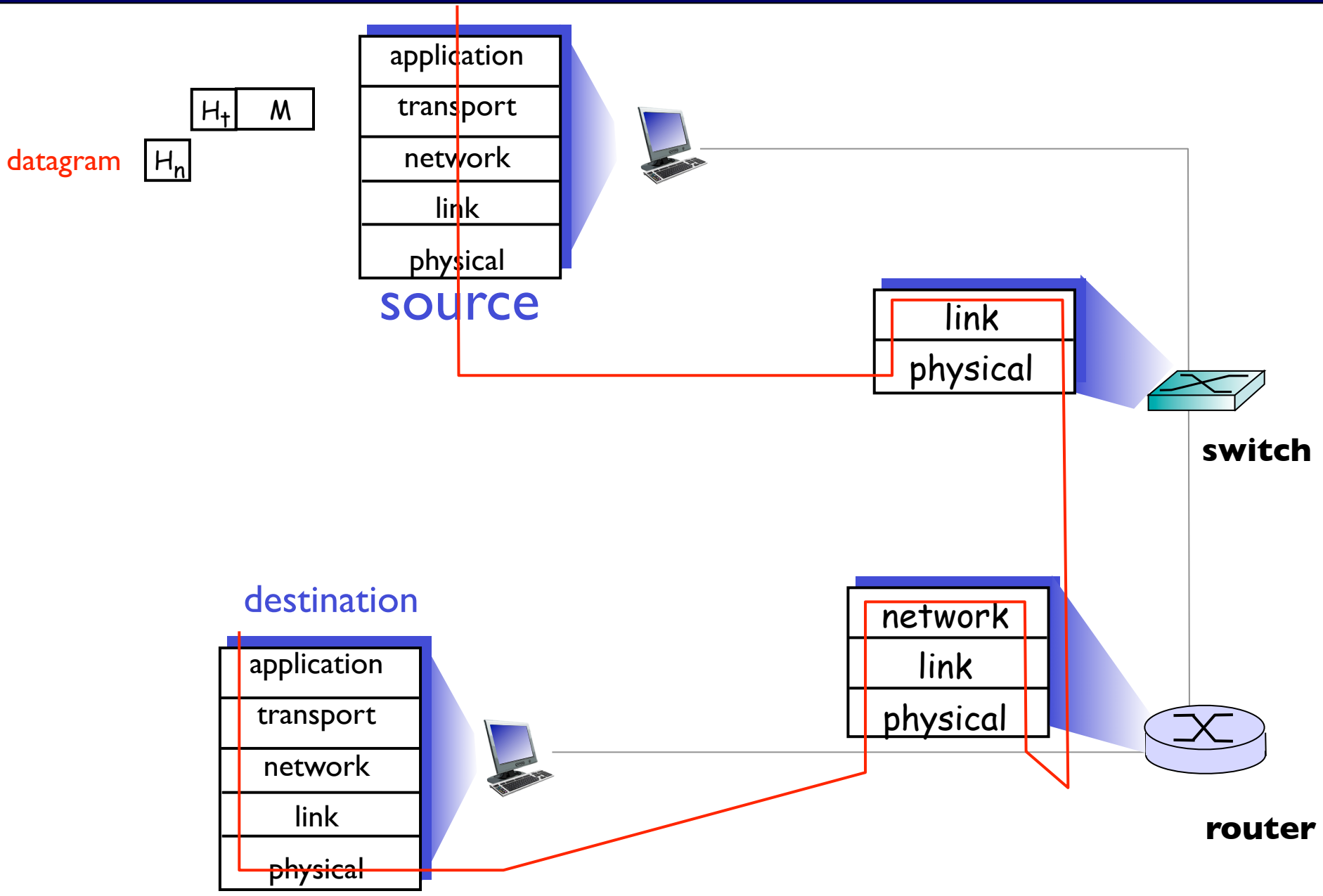
Encapsulation



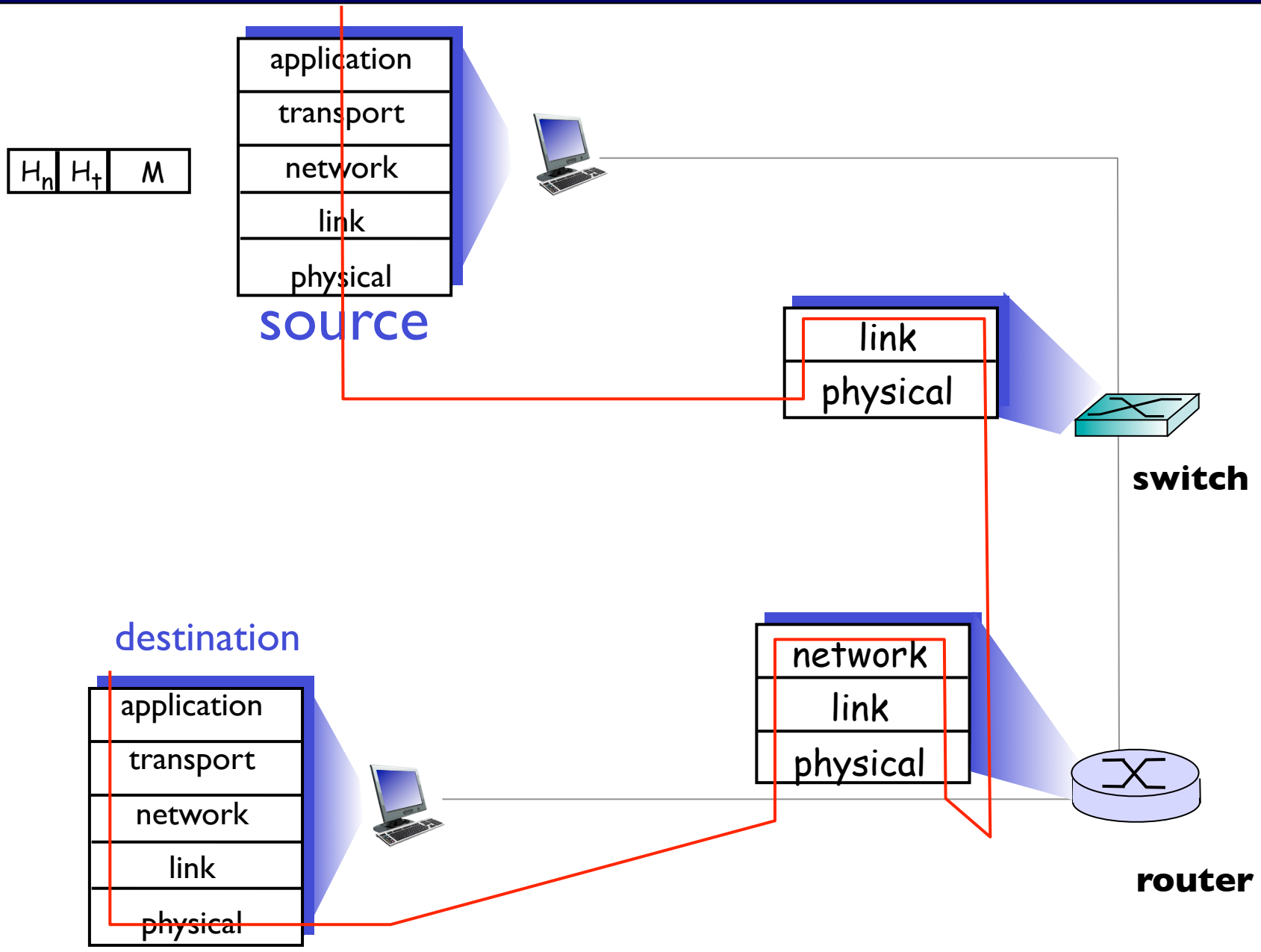
Encapsulation



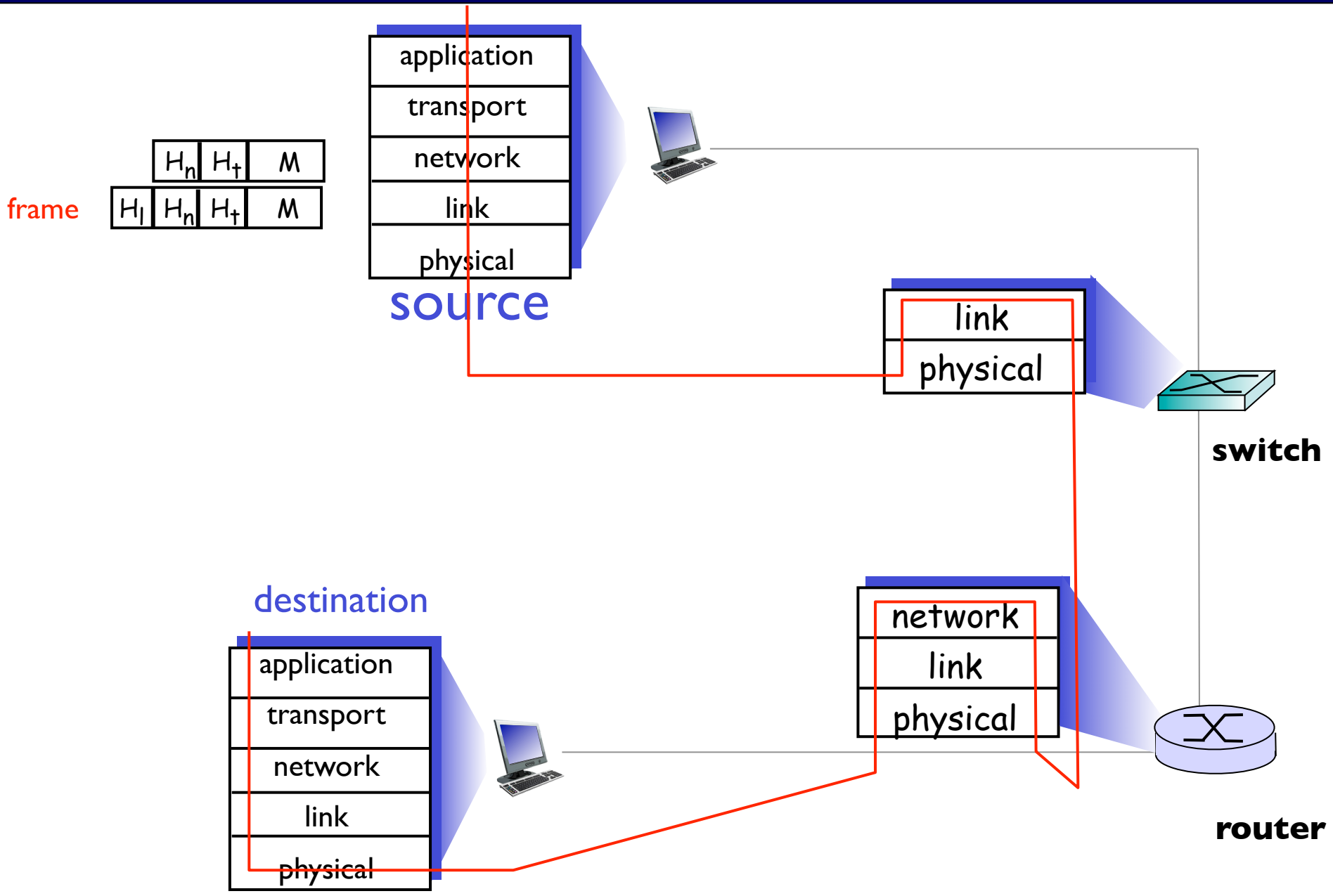
Encapsulation



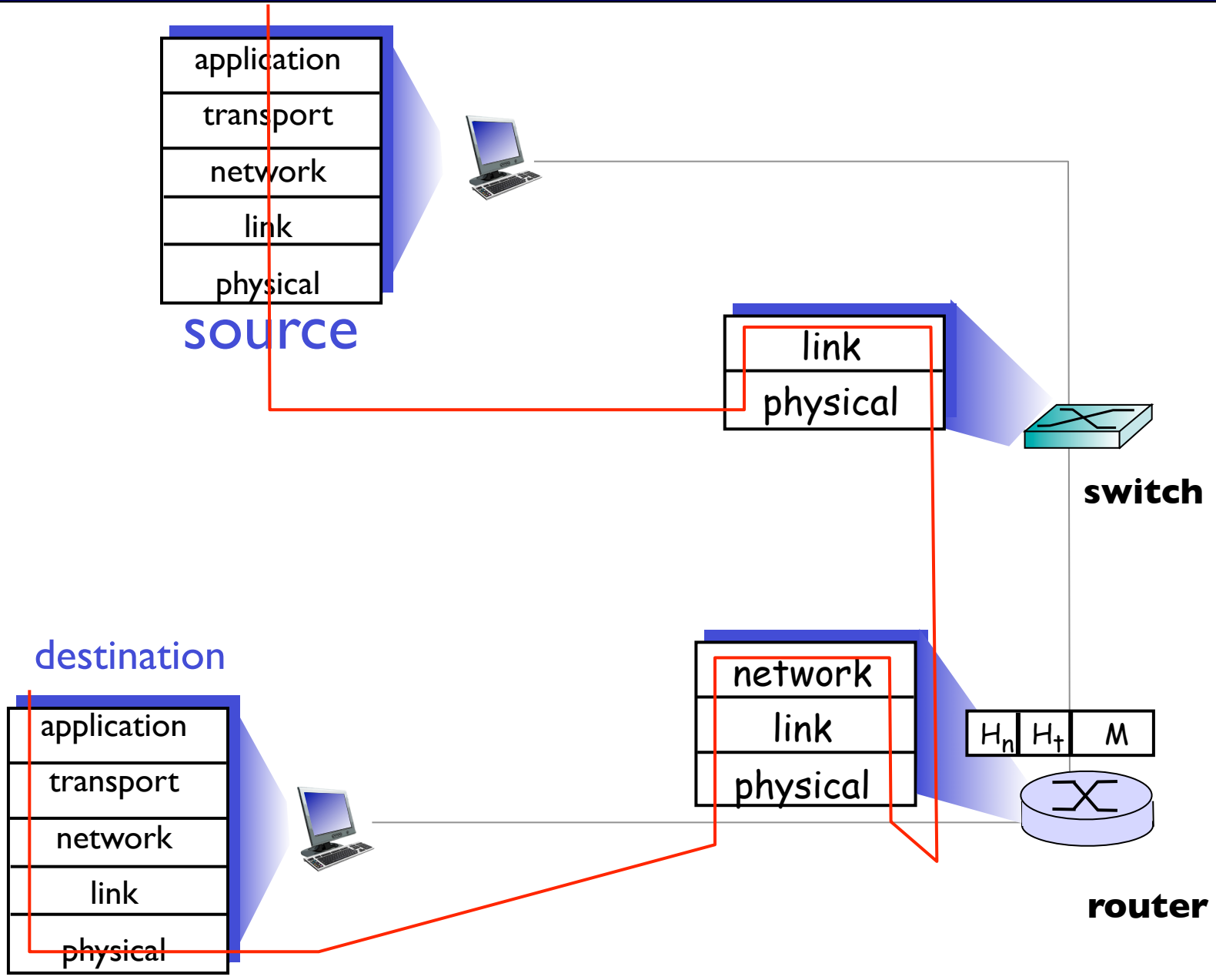
Encapsulation



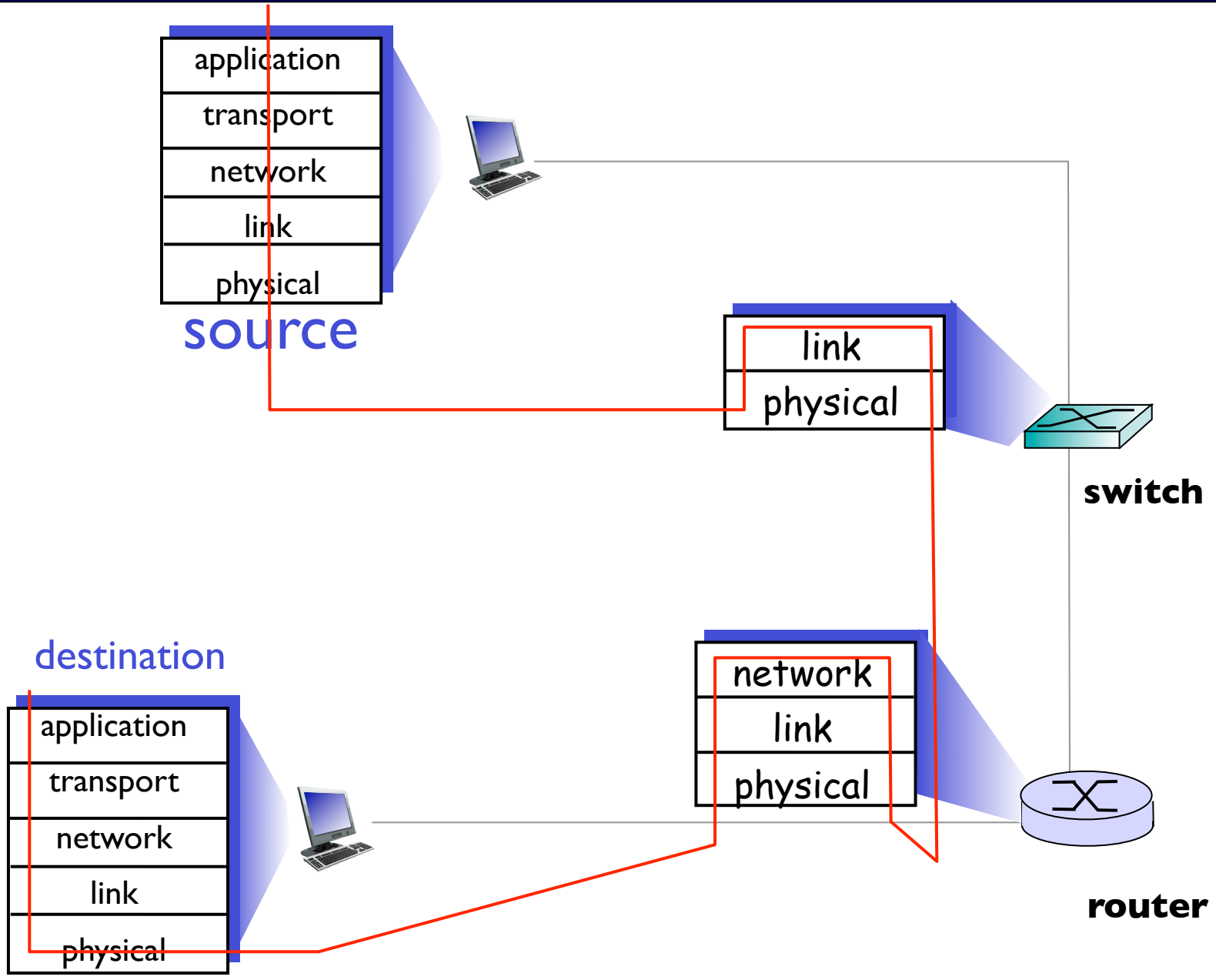
Encapsulation



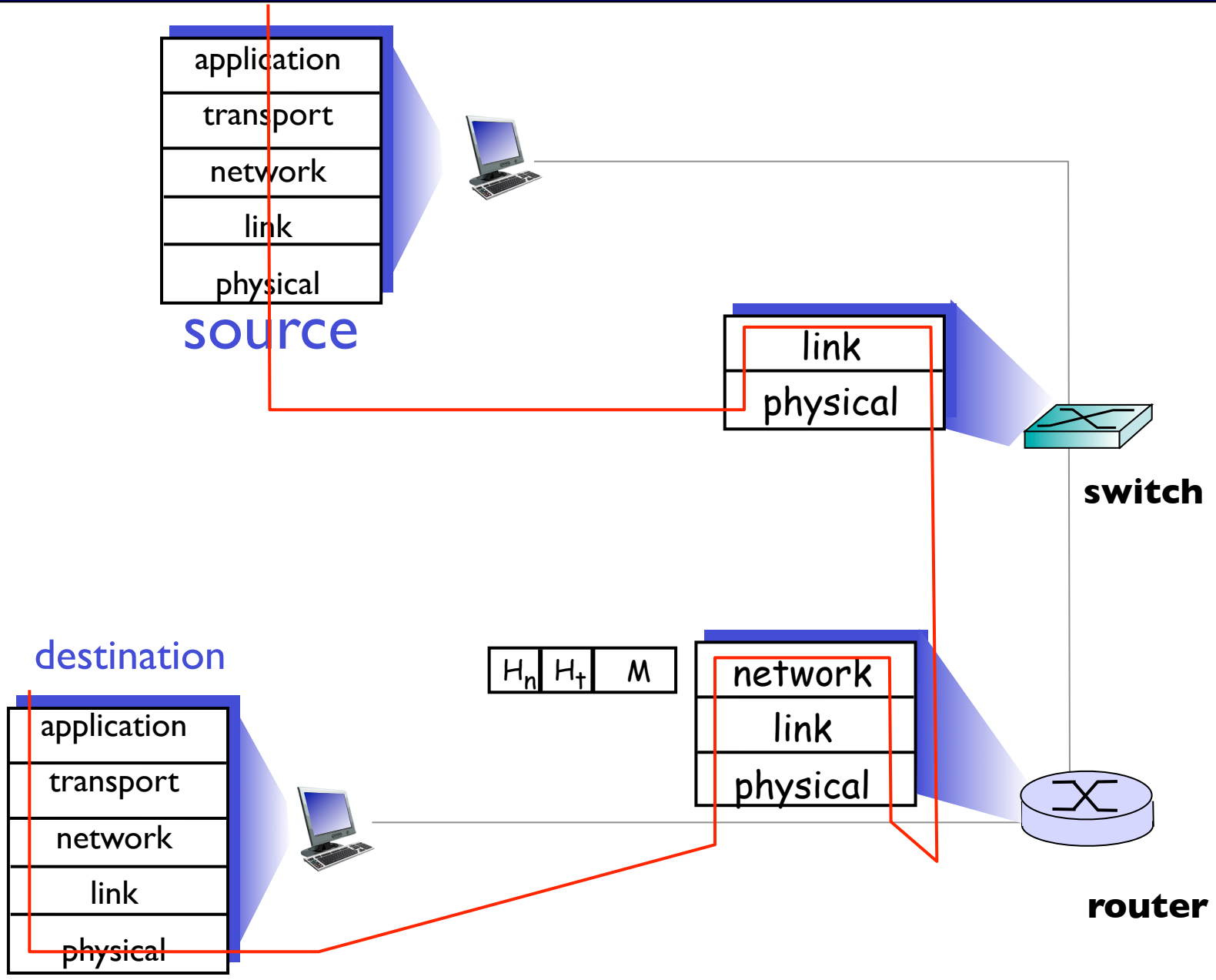
Encapsulation



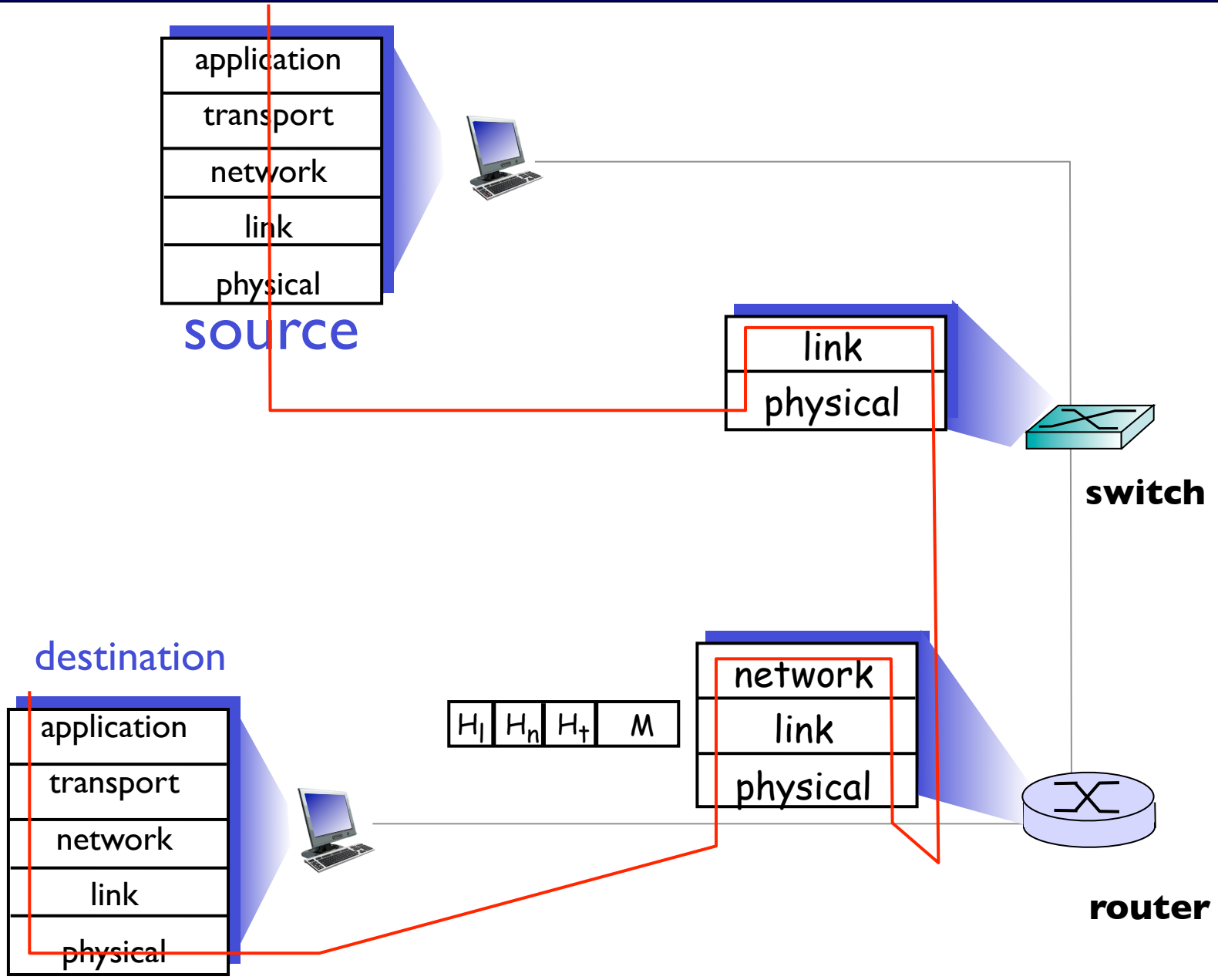
Encapsulation



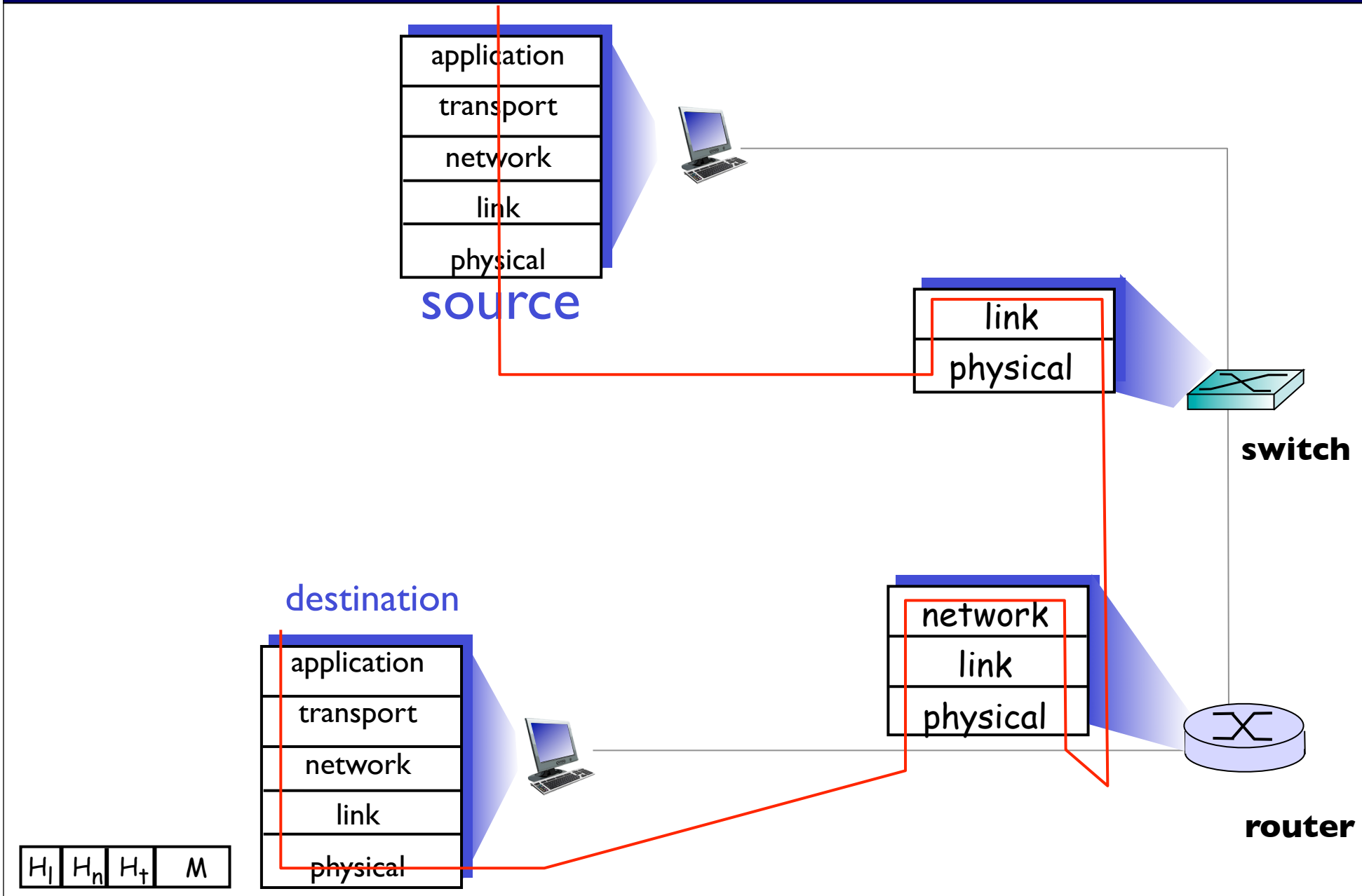
Encapsulation



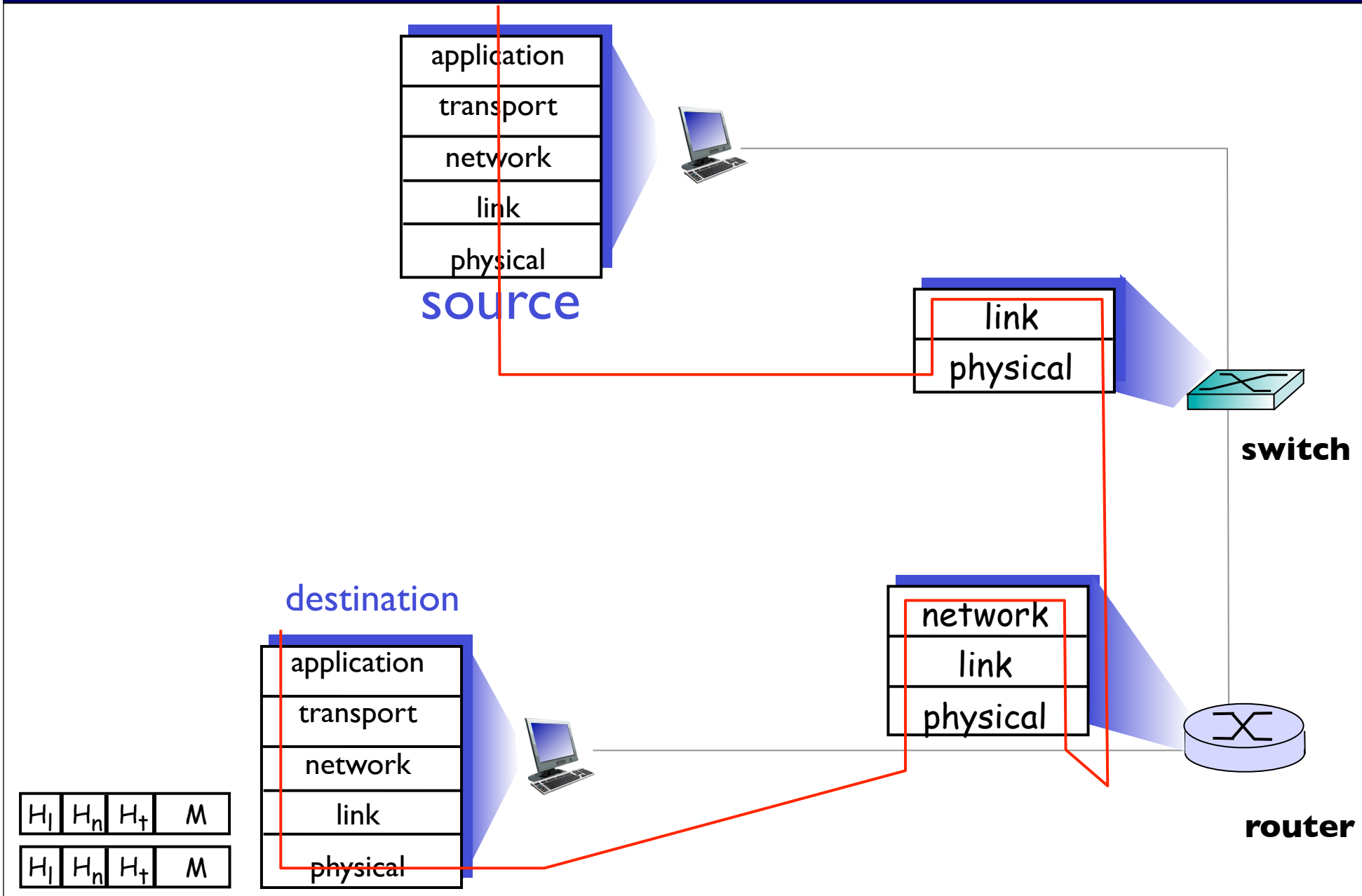
Encapsulation



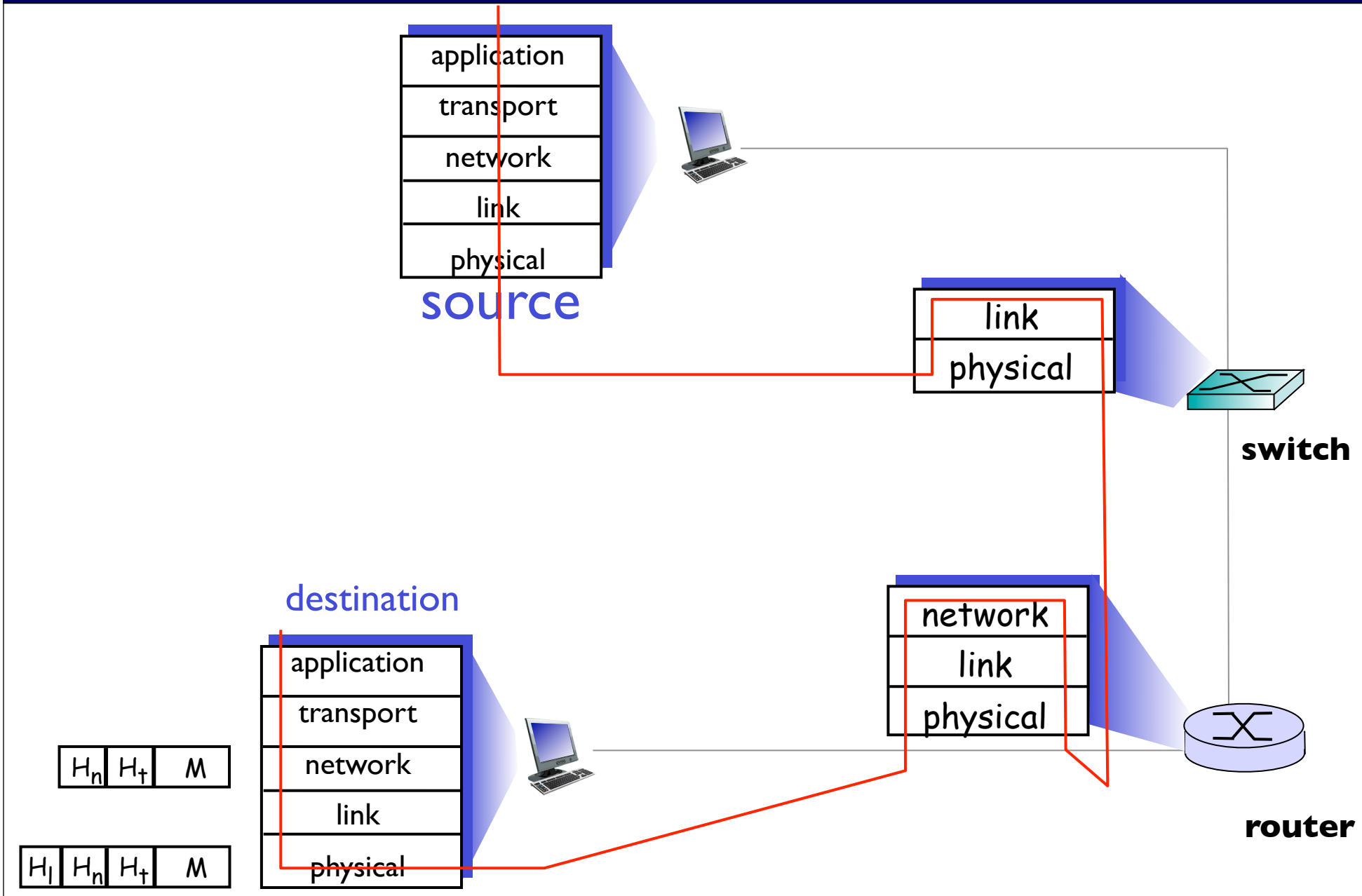
Encapsulation



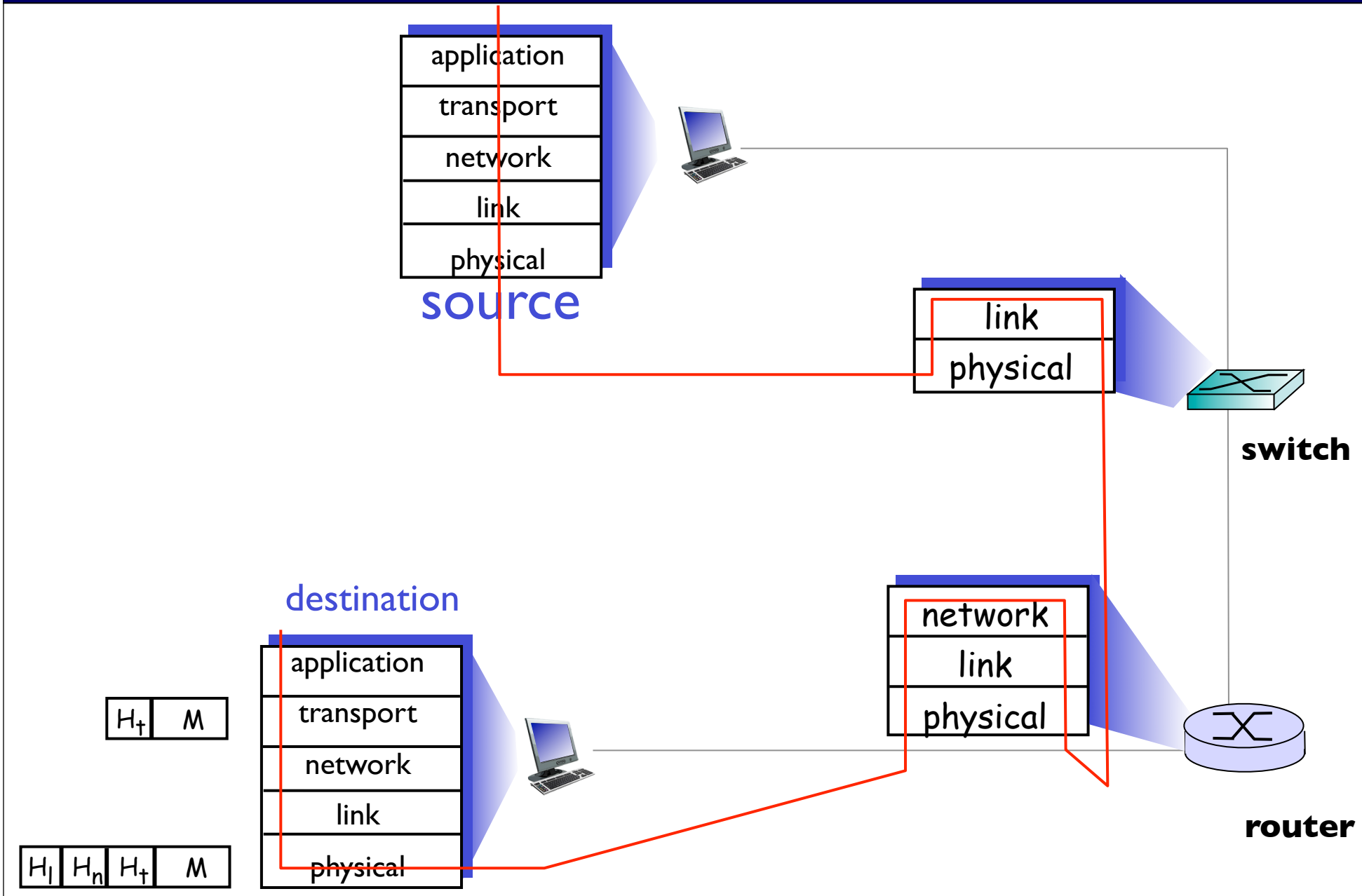
Encapsulation



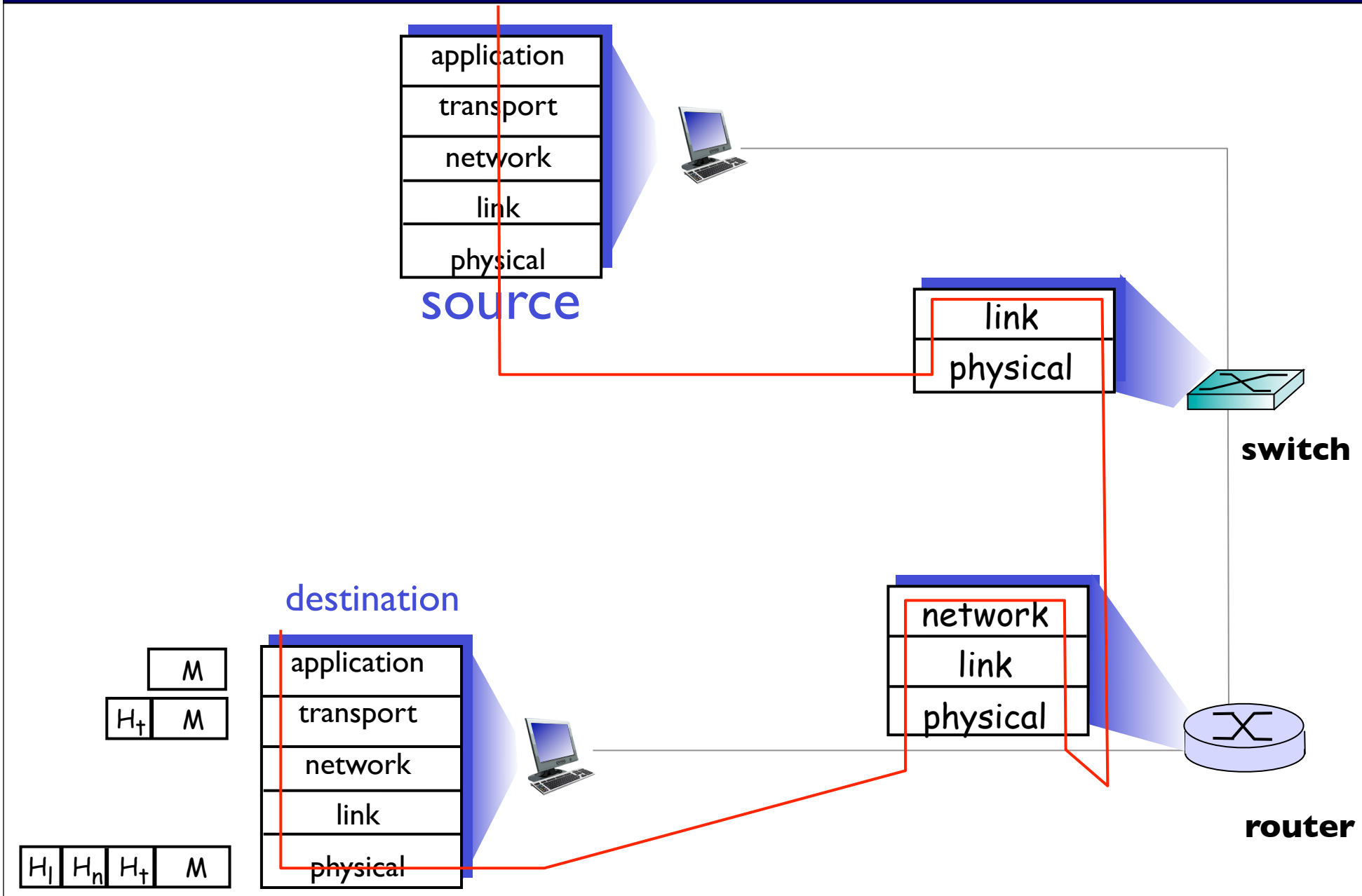
Encapsulation



Encapsulation



Encapsulation



The E2E Argument

PENNSTATE 1855

- *Idea*: most systems require end-to-end communication service, but low-level features have costs (performance) incurred by all users ... thus ...
- It is important that the features provided at a low level remain very simple ... yielding ...

Smart endpoints ... dumb minimal network

- *Consequence*: the network is simple and not very receptive to new (often complicated) services being added into them. Thus, systems must implement their own functions over the largely dumb (but fast an reliable) network

Network vs. Web

PENNSTATE 1855

- The network is a service ...
  - A conduit for data to be passed between systems.
  - Layers services (generally) to allow flexibility.
  - Highly scalable.
  - This is a public channel.

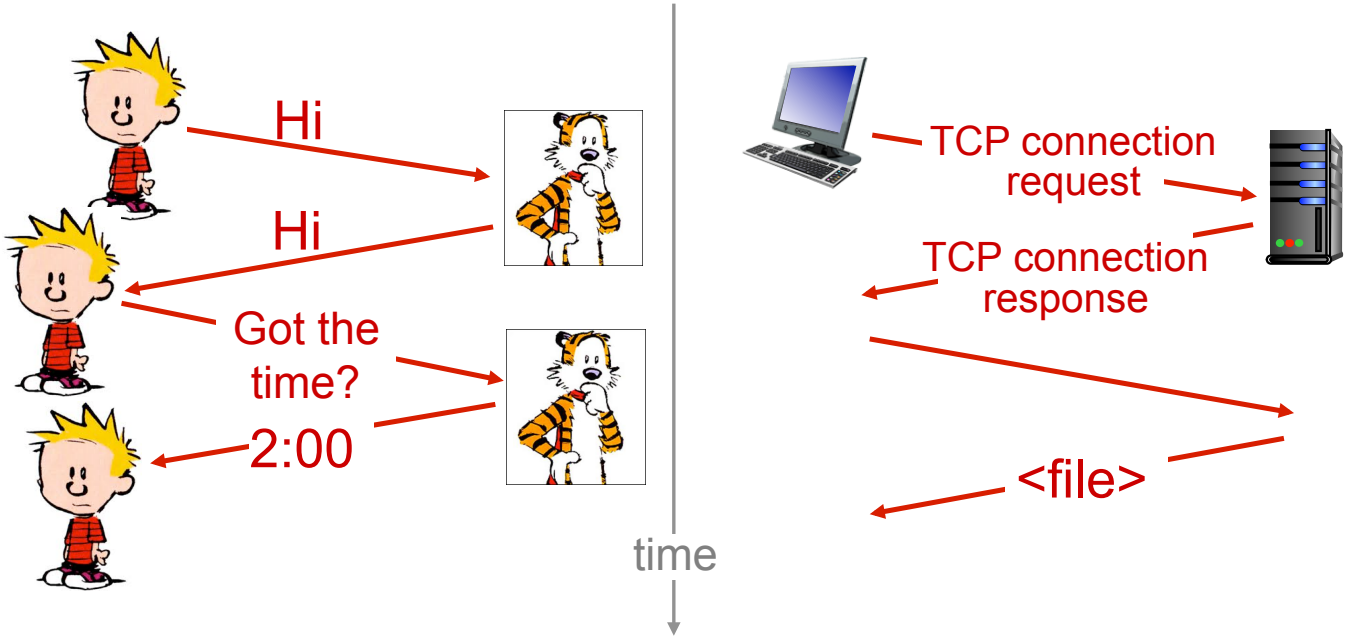
- The Web is an application
  - This is an application for viewing/manipulating content.
  - The services are unbounded by services, e.g., Java.
  - This can either be public (as in CNN's website), or private (as in enterprise internal HR websites).

Networks Systems

- Conceptually, think about network programming as two or more programs on the same or different computers talking to each other
  - The send messages back and forth
  - The "flow" of messages and the meaning of the message content is called the *network protocol* or just protocol

What's a Protocol?

- Example: A human protocol and a computer protocol:



Socket Programming

PENNSTATE 1855

- Almost all meaningful careers in programming involve at least some level of network programming.
- Most of them involve sockets programming
  - Berkeley sockets originated in 4.2 BSD Unix circa 1983
    - it is the standard API for network programming
    - available on most OSs
  - POSIX socket API
    - a slight updating of the Berkeley sockets API
    - a few functions were deprecated or replaced
    - better support for multi-threading was added

Recall file descriptors

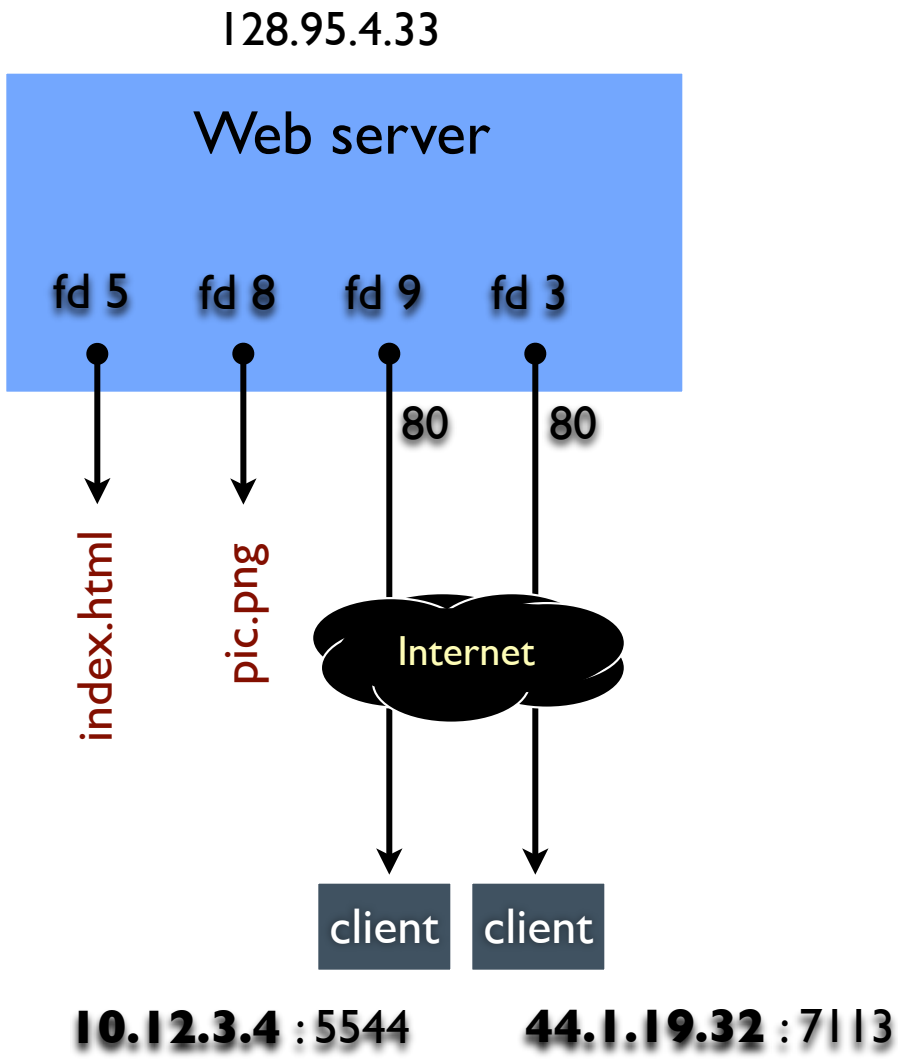
PENNSTATE 1855

- Remember open, read, write, and close?
  - POSIX system calls interacting with files
  - recall `open()` returns a *file descriptor*
    - an integer that represents an open file
    - inside the OS, it's an index into a table that keeps track of any state associated with your interactions, such as the file position
    - you pass the file descriptor into read, write, and close

Networks and sockets

- UNIX makes all I/O look like file I/O
  - the good news is that you can use `read()` and `write()` to interact with remote computers over a network!
  - just like with files....
    - A program can have multiple network channels open at once
    - you need to pass `read()` and `write()` a *file descriptor* to let the OS know which network channel you want to write to or read from
  - The file descriptor used for network communications is a *socket*

Pictorially



file descriptor	type	connected to?
0	pipe	stdin (console)
1	pipe	stdout (console)
2	pipe	stderr (console)
3	TCP socket	local: 128.95.4.33:80 remote: 44.1.19.32:7113
5	file	index.html
8	file	pic.png
9	TCP socket	local: 128.95.4.33:80 remote: 102.12.3.4:5544

OS's descriptor table

Types of sockets

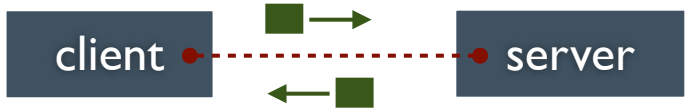
PENNSTATE 1855

- Stream sockets
  - for connection-oriented, point-to-point, reliable bytestreams
    - uses TCP, SCTP, or other stream transports
- Datagram sockets
  - for connection-less, one-to-many, unreliable packets
    - uses UDP or other packet transports
- Raw sockets
  - for layer-3 communication (raw IP packet manipulation)

Stream sockets

- Typically used for client / server communications
  - but also for other architectures, like peer-to-peer
- Client
  - an application that establishes a connection to a server
- Server
  - an application that receives connections from clients

1. establish connection

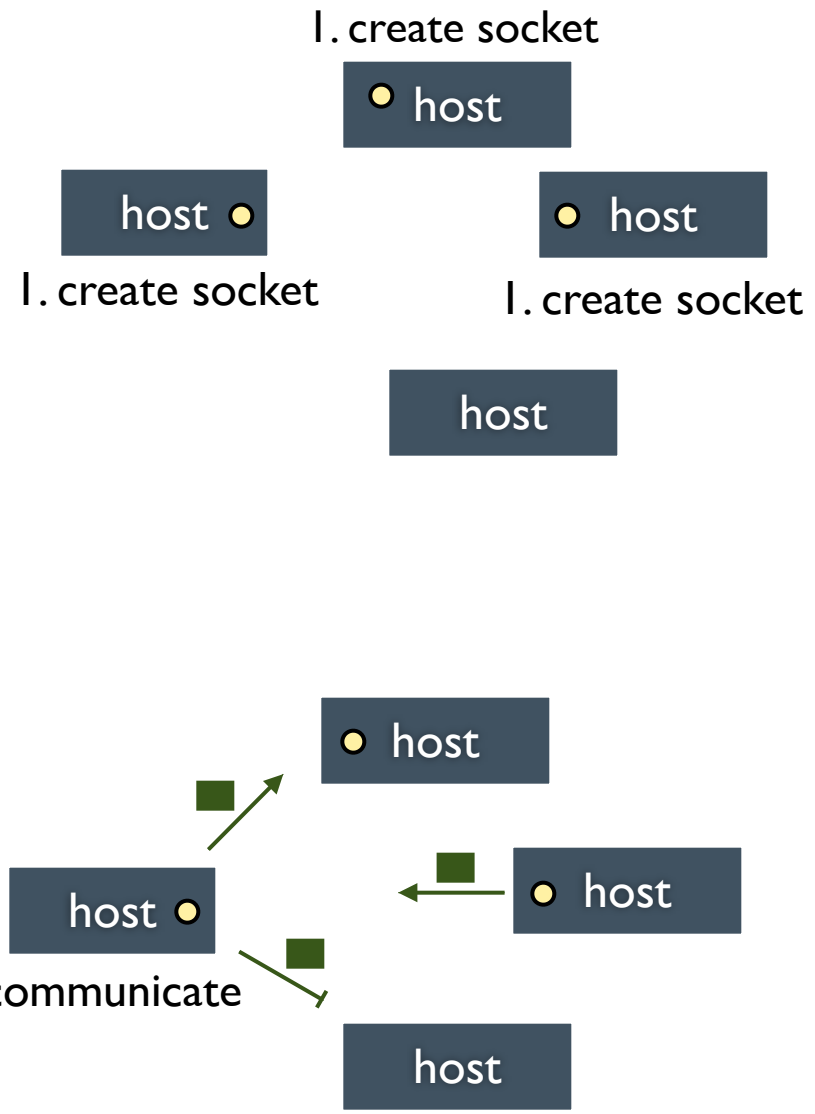


2. communicate

3. close connection

Datagram sockets

- Used less frequently than stream sockets
  - they provide no flow control, ordering, or reliability
- Often used as a building block
  - streaming media applications
  - sometimes, DNS lookups



Note: this is also called "*connectionless*" communication

Network communication

- Every communication/connection is defined by:
  1. A transport protocol type (TCP, UDP)
  2. An IP address (e.g., 192.168.27.4)
  3. A port (80/http)

Network addresses

- Every device on the Internet needs to have an address
  - Needs to be unique to make sure that it can be reached unambiguously
- For IPv4, an IP address is a 4-byte tuple
  - e.g., 128.95.4.1 (80:5f:04:01 in hex)
- For IPv6, an IP address is a 16-byte tuple
  - e.g., 2d01:0db8:f188:0000:0000:0000:0000:1f33
    - 2d01:0db8:f188::1f33 in shorthand

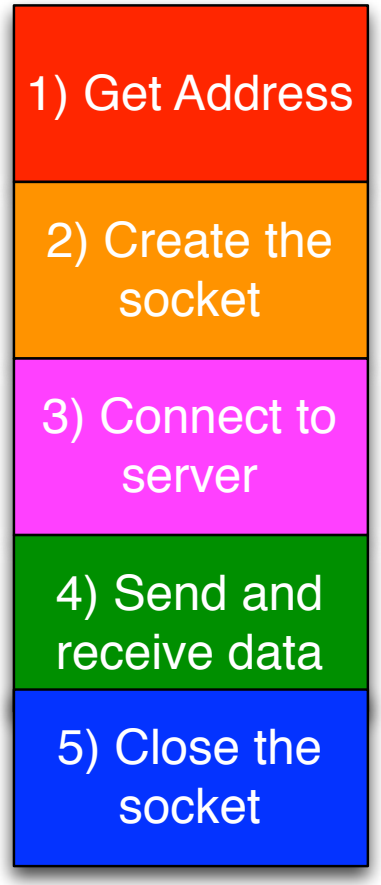
Network Ports

PENNSTATE 1855

- Every computer has a numbered set of locations that represent the available "services" can be reached at
  - Ports are numbered 0 to 65535
    - 0 to 1023 are called "*well known*" or "*reserved*" ports, where you need special privileges to receive on these ports
  - Each transport (UDP/TCP) has its own list of ports
- Interesting port numbers
  - 20/21 - file transfer protocol (file passing)
  - 22 - secure shell (remote access)
  - 25 - Simple mail transfer protocol (email)
  - 53 - domain name service (internet naming)
  - 80 - HTTP (web)

Programming a client

- We'll start by looking at the API from the point of view of a client connecting to a server over TCP
  - there are five steps:
    1. figure out the address/port to connect to
    2. create a socket
    3. connect the socket to the remote server
    4. `read()` and `write()` data using the socket
    5. close the socket



inet_aton()

1) Get Address

- The `inet_aton()` converts a IPv4 address into the UNIX structure used for processing:

```
int inet_aton(const char *addr, struct in_addr *inp);
```

- Where,
  - `addr` is a string containing the address to use
  - `inp` is a pointer to the structure containing the UNIX internal representation of an address
    - used in later system calls

- `inet_aton()` returns 0 if failure!

Putting it to use ...

1) Get Address

```c
#include <stdlib.h>
#include <arpa/inet.h>

int main(int argc, char **argv) {
  struct sockaddr_in v4, sa; // IPv4
  struct sockaddr_in6 sa6; // IPv6

  // IPv4 string to sockaddr_in. (both ways)
  inet_aton("192.0.1.1", &(v4.sin_addr));
  inet_pton(AF_INET, "192.0.2.1", &(sa.sin_addr));

  // IPv6 string to sockaddr_in6.
  inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr));

  return( 0 );
}
```

Getting back to strings?

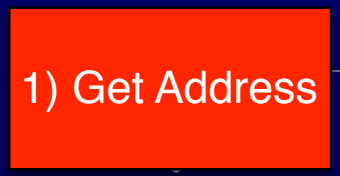
1) Get Address

- The `inet_ntoa()` converts a IPv4 address into the UNIX structure used for processing:

```
char *inet_ntoa(struct in_addr in);
```

```
struct sockaddr_in caddr, sa; // IPv4
struct sockaddr_in6 sa6; // IPv6
char astring[INET6_ADDRSTRLEN];  // IPv6

// Start by converting
inet_aton( "192.168.8.9", &caddr.sin_addr);
inet_pton(AF_INET, "192.0.2.1", &(sa.sin_addr));
inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr));

// Return to ASCII strings
inet_ntop(AF_INET6, &(sa6.sin6_addr), astring, INET6_ADDRSTRLEN);
printf( "IPv4 : %s\n", inet_ntoa(caddr.sin_addr) );
printf( "IPv6 : %s\n", astring );
```

Domain Name Service (DNS)



- People tend to use DNS names, not IP addresses
  - the sockets API lets you convert between the two
  - it's a complicated process, though:
    - a given DNS name can have many IP addresses
    - many different DNS names can map to the same IP address
      - an IP address will map onto at most one DNS names, and maybe none
    - a DNS lookup may require interacting with many DNS servers

Note: The "dig" Linux program is used to check DNS entries.

Domain Name Service (DNS)

1) Get Address

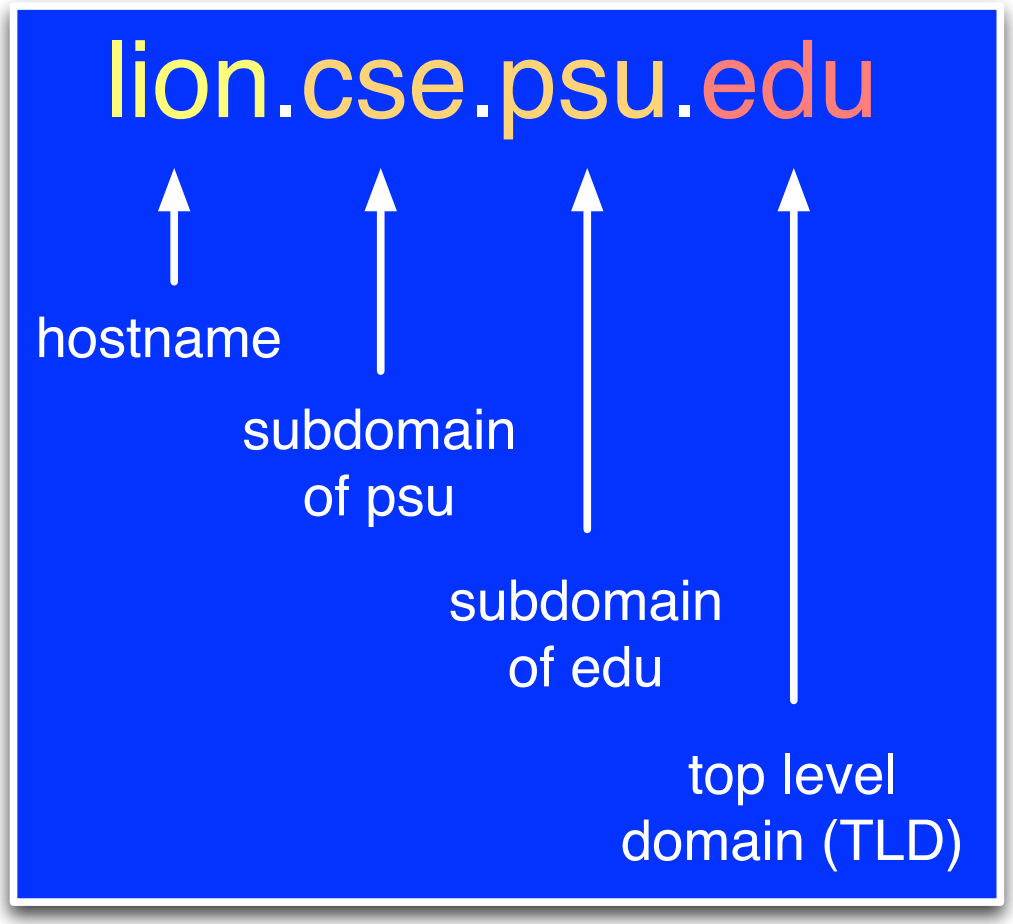
```
$ dig lion.cse.psu.edu

; <<>> DiG 9.9.2-P1 <<>> lion.cse.psu.edu
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 53447
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; MBZ: 0005 , udp: 4000
;; QUESTION SECTION:
;lion.cse.psu.edu.              IN      A

;; ANSWER SECTION:
lion.cse.psu.edu.       5       IN      A       130.203.22.184

;; Query time: 38 msec
;; SERVER: 127.0.1.1#53(127.0.1.1)
;; WHEN: Tue Nov 12 14:02:11 2013
;; MSG SIZE  rcvd: 61
```

Note: The "dig" Linux program is used to check DNS entries.

The FQDN

1) Get Address

- Every system that is supported by DNS has a unique *fully qualified domain name*



DNS hierarchy

1) Get Address

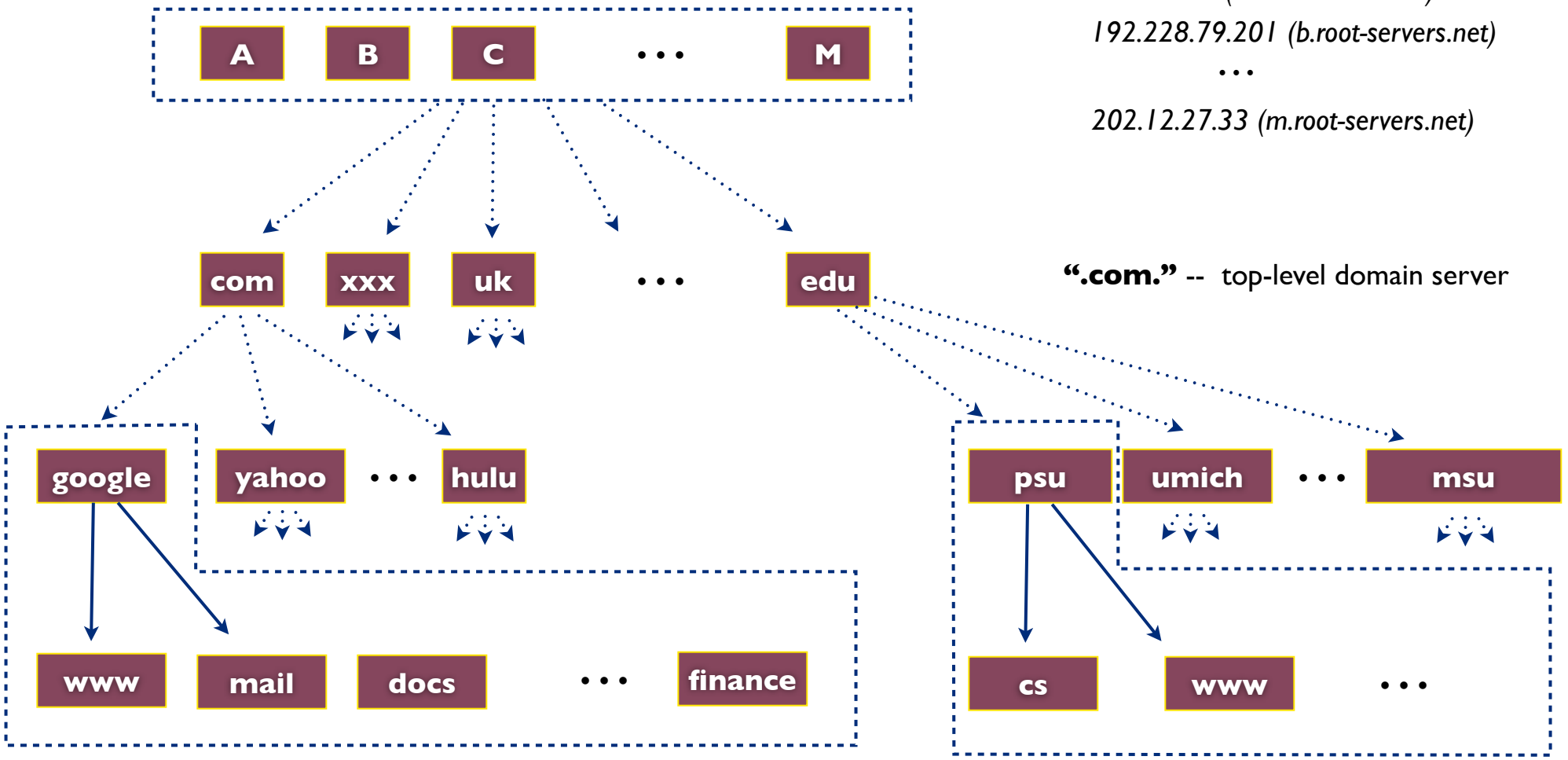
"." -- root name servers

198.41.0.4 (a.root-servers.net)

192.228.79.201 (b.root-servers.net)

...

202.12.27.33 (m.root-servers.net)



".com." -- top-level domain server

Resolving DNS names

1) Get Address

- The POSIX way is to use `getaddrinfo()`
- a pretty complicated system call; the basic idea...
  - set up a "hints" structure with constraints you want respected
    - e.g., IPv6, IPv4, or either
  - indicate which host and port you want resolved
    - host: a string representation; DNS name or IP address
  - returns a list of results packet in an "`addrinfo`" struct
  - free the `addrinfo` structure using `freeaddrinfo()`

DNS resolution (the easy way)

1) Get Address

- The `gethostbyname()` uses DNS to look up a name and return the host information

```
struct hostent *gethostbyname(const char *name);
```

-

```
char hn = "lion.cse.psu.edu";
struct hostent * hstinfo;
struct in_addr **addr_list;
if ( (hstinfo = gethostbyname(hn)) == NULL ) {
  return( -1 );
}

addr_list = (struct in_addr **)hstinfo->h_addr_list;
printf( "DNS lookup [%s] address [%s]\n", hstinfo->h_name,
        inet_ntoa(*addr_list[0]) );
```

```
$ ./network
DNS lookup [lion.cse.psu.edu] address [130.203.22.184]
$
```

Creating a socket

2) Create the socket

- The `socket()` function creates a file handle for use in network communication:

```
int socket(int domain, int type, int protocol);
```

- Where,
  - `domain` is the communication domain
    - `AF_INET` (IPv4), `AF_INET6` (IPv6)
  - `type` is the communication semantics (stream vs. datagram)
    - SOCK_STREAM is stream (using TCP by default)
    - SOCK_DGRAM is datagram (using UDP by default)
  - `protocol` selects a protocol from available (not used often)

Note: creating a socket doesn't connect to anything

Creating a socket

2) Create the socket

- The `socket()` function creates a file handle for use in network communication:

```
int socket(int domain, int type, int protocol);
```

-

```
// Create the socket
int socket_fd;
socket_fd = socket(PF_INET, SOCK_STREAM, 0);
if (socket_fd == -1) {
    printf( "Error on socket creation [%s]\n", strerror(errno) );
    return( -1 );
}
```

  - SOCK_STREAM is stream (using TCP by default)
  - SOCK_DGRAM is datagram (using UDP by default)
- `protocol` selects a protocol from available (not used often)

Note: creating a socket doesn't connect to anything

Specifying an address

3) Connect to server

- The next step is to create an address to connect to by specifying the address and port in the proper form.
  - protocol family (`addr.sin_family`)
  - port (`addr.sin_port`)
  - IP address (`addr.sin_addr`)

```
// Variables
char *ip = "127.0.0.1";
unsigned short port = 16453;
struct sockaddr_in caddr;

// Setup the address information
caddr.sin_family = AF_INET;
caddr.sin_port = htons(port);
if ( inet_aton(ip, &caddr.sin_addr) == 0 ) {
  return( -1 );
}
```

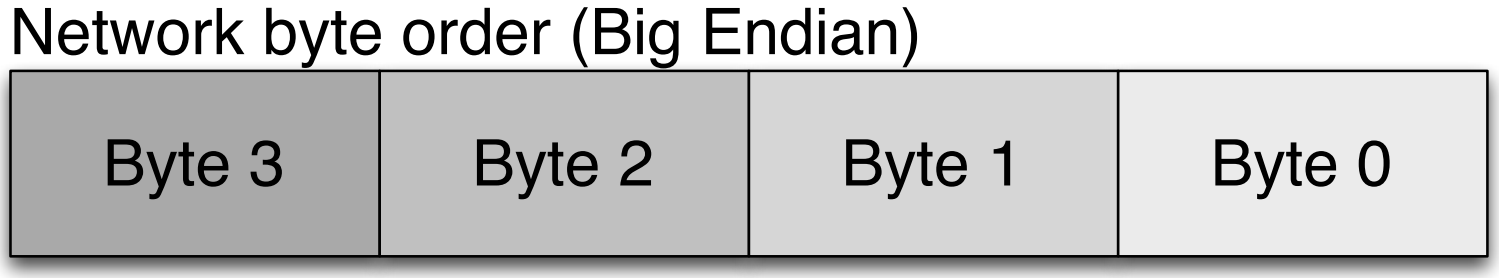
Network byte order

3) Connect to server

- When sending data over a network you need to convert your integers to be in *network byte order*, and back to *host byte order* upon receive:

```
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

- Where each of these functions receives a NBO or HBO 32 or 16 byte and converts it to the other.



connect()

3) Connect to server

- The `connect()` system call connects the socket file descriptor to the specified address

```
int connect(int sockfd, const struct sockaddr *addr,
                                socklen_t addrlen);
```

- Where,
  - `sockfd` is the socket (file handle) obtained previously
  - `addr` - is the address structure
  - `addlen` - is the size of the address structure
  - Returns 0 if successfully connected, -1 if not

```
if ( connect(sock, (const struct sockaddr *)&caddr,
             sizeof(struct sockaddr)) == -1 ) {
  return( -1 );
}
```

Reading and Writing

4) Send and receive data

- Primitive reading and writing mechanisms that only process only blocks of opaque data:

```
ssize_t write(int fd, const void *buf, size_t count);
    ssize_t read(int fd, void *buf, size_t count);
```

- Where `fd` is the file descriptor, `buf` is an array of bytes to write from or read into, and `count` is the number of bytes to read or write
- In both read() and write(), the value returned is the number of bytes read and written.
  - Be sure to always check the result
- On reads, you are responsible for supplying a buffer that is large enough to put the output into.

close()

5) Close the socket

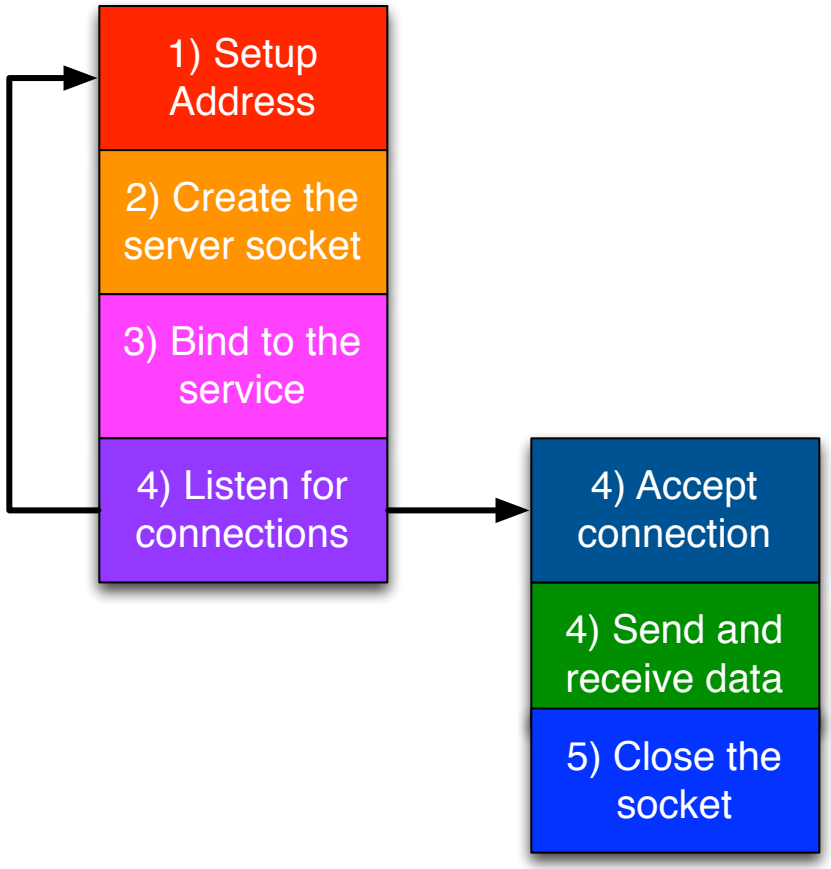
- `close()` closes the connection and deletes the associated entry in the operating system's internal structures

```
close( socket_fd );
socket_fd = -1;
```

Note: set handles to –1 to avoid use after close.

Programming a server

- Now we'll look at the API from the point of view of a server who receives connections from clients
  - there are seven steps:
    1. figure out the port to "bind" to
    2. create a socket
    3. bind the service
    4. begin listening for connections
    5. receive connection
    6. `read()` and `write()` data
    7. close the socket



Setting up a server address

1) Setup Address

- All you need to do is specify the service port you will use for the connection:

```
struct sockaddr_in saddr;
saddr.sin_family = AF_INET;
saddr.sin_port = htons(16453);
saddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

- However, you don't specify an IP address because you are receiving connections at the local host.
  - Instead you use the special "any" address

`htonl(INADDR_ANY)`

Next: creating the socket is as with the client

2) Create the server socket

Binding the service

3) Bind to the service

- The `bind()` system call associates a socket with the server connection (e.g., taking control of HTTP)

```
int bind(int sockfd, const struct sockaddr *addr,
                                socklen_t addrlen
```

- Where,
  - `sockfd` is the socket (file handle) obtained previously
  - `addr` - is the address structure
  - `addlen` - is the size of the address structure
  - Returns 0 if successfully connected, -1 if not

```
if ( bind(sock, (const struct sockaddr *)&saddr,
          sizeof(struct sockaddr)) == -1 ) {
  return( -1 );
}
```

Listening for connections

4) Listen for connections

- The `listen()` system call tells the OS to start receiving connections on behalf of the process

```
int listen(int sockfd, int backlog);
```

- Where,
  - `sockfd` is the socket (file handle) obtained previously
  - `backlog` - is the number of connections to queue
    - A program may process connections as fast as it wants, and the OS will hold the client in a waiting state until you are ready
    - Beware of waiting too long (timeout)

```
if ( listen(sock, 5) == -1 ) {
  return( -1 );
}
```

Waiting for data/connections

4) Listen for connections

- `select()` waits for data to process:

```
int select(int nfds,
           fd_set *read_fds,
           fd_set *write_fds,
           fd_set *error_fds,
           struct timeval *timeout);
```

- Waits (up to timeout) events of the following types:
  - readable events on (`read_fds`)
  - writeable events on (`write_fds`)
  - error events on (`error_fds`)

Note: processing select is complex, see man page ...

Accepting connections

4) Listen for connections

- The `listen()` system call tells the OS to start receiving connections on behalf of the process

```
int listen(int sockfd, int backlog);
```

- Where,
  - `sockfd` is the socket (file handle) obtained previously
  - `backlog` - is the number of connections to queue
    - A program may process connections as fast as it wants, and the OS will hold the client in a waiting state until you are ready
    - Beware of waiting too long (timeout)

```
if ( listen(sock, 5) == -1 ) {
  return( -1 );
}
```

Accepting connections

4) Accept connection

- The `accept()` system call receives the connection from the client:

```
int connect(int sockfd, const struct sockaddr *addr,
                                socklen_t addrlen);
```

- Where,
  - `sockfd` is the socket (file handle) obtained previously
  - `addr` - is the address structure for client (filled in)
  - `addlen` - is the size of the address structure
  - Returns the new socket handle or -1 if failure

Accepting connections

4) Accept connection

- The `accept()` system call receives the connection from the client:

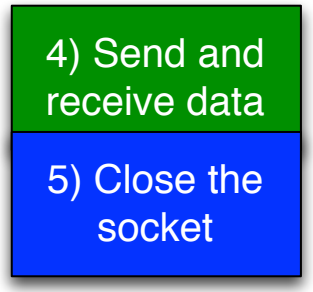
```
int connect(int sockfd, const struct sockaddr *addr,
                                    socklen_t addrlen);
```

- Where,
  - `sockfd` is the socket (file handle) obtained previously
  - `addr` - is the address structure for client (filled in)

```
inet_len = sizeof(caddr);
if ( (client = accept(server, (struct sockaddr *)&caddr,
       &inet_len )) == -1 ) {
  printf( "Error on client accept [%s]\n", strerror(errno) );
  close(server);
  return( -1 );
}
printf( "Server new client connection [%s/%d]",
        inet_ntoa(caddr.sin_addr), caddr.sin_port );
```

The rest ...

- From the server perspective, receiving and sending on the newly received socket is the same as if it were a client
  - read() and write() for sending and receiving
  - close() for closing the socket



Putting it all together (client)

Putting it all together (client)

```c
int client_operation( void ) {

    int socket_fd;
    uint32_t value;
    struct sockaddr_in caddr;
    char *ip = "127.0.0.1";

    caddr.sin_family = AF_INET;
    caddr.sin_port = htons(16453);
    if ( inet_aton(ip, &caddr.sin_addr) == 0 ) {
        return( -1 );
    }

    socket_fd = socket(PF_INET, SOCK_STREAM, 0);
    if (socket_fd == -1) {
        printf( "Error on socket creation [%s]\n", strerror(errno) );
        return( -1 );
    }

    if ( connect(socket_fd, (const struct sockaddr *)&caddr, sizeof(struct sockaddr)) == -1 ) {
        printf( "Error on socket connect [%s]\n", strerror(errno) );
        return( -1 );
    }

    value = htonl( 1 );
    if ( write( socket_fd, &value, sizeof(value)) != sizeof(value) ) {
        printf( "Error writing network data [%s]\n", strerror(errno) );
        return( -1 );
    }
    printf( "Sent a value of [%d]\n", ntohl(value) );

    if ( read( socket_fd, &value, sizeof(value)) != sizeof(value) ) {
        printf( "Error reading network data [%s]\n", strerror(errno) );
        return( -1 );
    }
    value = ntohl(value);
    printf( "Receivd a value of [%d]\n", value );

    close(socket_fd); // Close the socket
    return( 0 );
}
```

Putting it all together (server)

Putting it all together (server)

PENNSTATE 1855

```
int server_operation( void ) {

    int server, client;
    uint32_t value, inet_len;
    struct sockaddr_in saddr, caddr;

    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(16453);
    saddr.sin_addr.s_addr = htonl(INADDR_ANY);

    server = socket(PF_INET, SOCK_STREAM, 0);
    if (server == -1) {
        printf( "Error on socket creation [%s]\n", strerror(errno) );
        return( -1 );
    }

    if ( bind(server, (struct sockaddr *)&saddr, sizeof(struct sockaddr)) == -1 ) {
        printf( "Error on socket bind [%s]\n", strerror(errno) );
        return( -1 );
    }

    if ( listen( server, 5 ) == -1 ) {
        printf( "Error on socket listen [%s]\n", strerror(errno) );
        return( -1 );
    }
```

... Together (server, part 2)

PENNSTATE 1855

... Together (server, part 2)

```
    while ( 1 ) {

        inet_len = sizeof(caddr);
        if ( (client = accept( server, (struct sockaddr *)&caddr, &inet_len )) == -1 ) {
            printf( "Error on client accept [%s]\n", strerror(errno) );
            close(server);
            return( -1 );
        }
        printf( "Server new client connection [%s/%d]", inet_ntoa(caddr.sin_addr), caddr.sin_port );

        if ( read( client, &value, sizeof(value)) != sizeof(value) ) {
            printf( "Error writing network data [%s]\n", strerror(errno) );
            close(server);
            return( -1 );
        }
        value = ntohl(value);
        printf( "Receivd a value of [%d]\n", value );

        value++;
        if ( write( client, &value, sizeof(value)) != sizeof(value) ) {
            printf( "Error writing network data [%s]\n", strerror(errno) );
            close(server);
            return( -1 );
        }
        printf( "Sent a value of [%d]\n", value );

        close(client); // Close the socket
    }

    return( 0 );
}
```