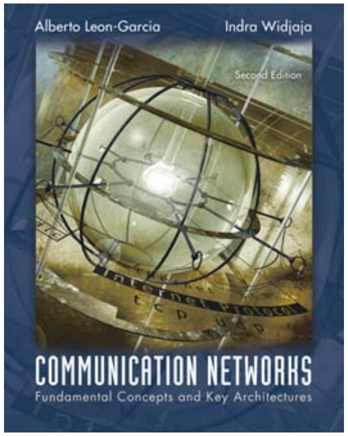
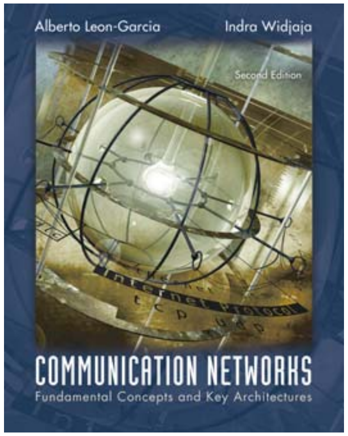
Chapter 2 Applications and Layered Architectures



Protocols, Services & Layering
OSI Reference Model
TCP/IP Architecture
How the Layers Work Together
Berkeley Sockets
Application Layer Protocols & Utilities

Chapter 2
Applications and Layered Architectures



Protocols, Services & Layering

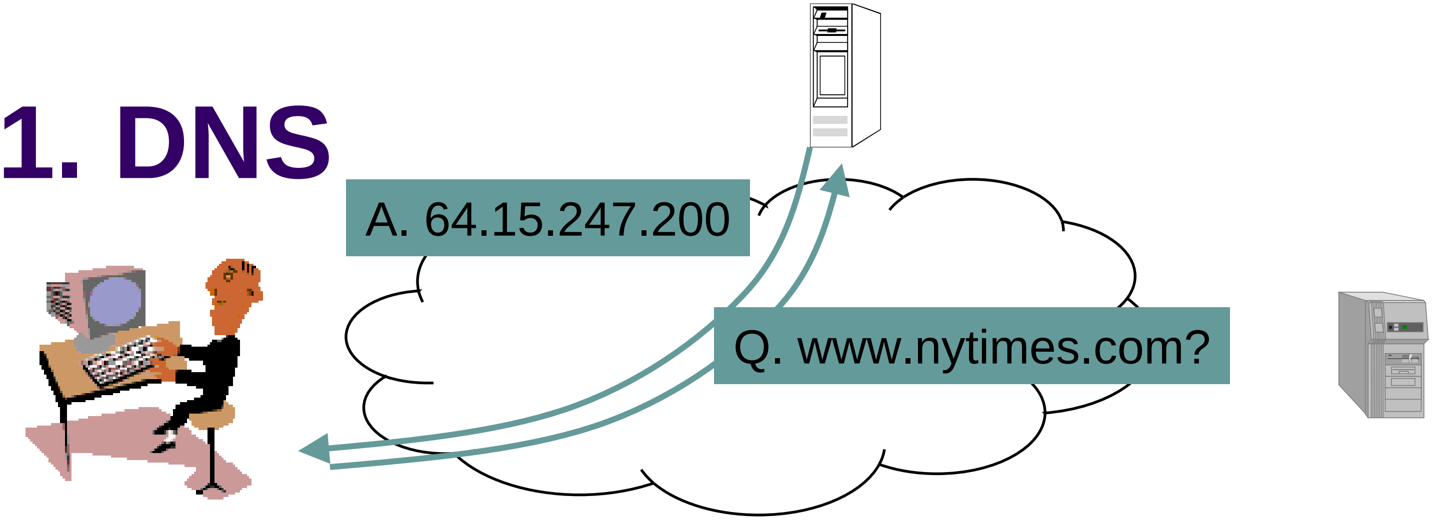
Layers, Services & Protocols

- The overall communications process between two or more machines connected across one or more networks is very complex
- ***Layering*** partitions related communications functions into groups that are manageable
- Each layer provides a ***service*** to the layer above
- Each layer operates according to a ***protocol***
- Let's use examples to show what we mean

Web Browsing Application

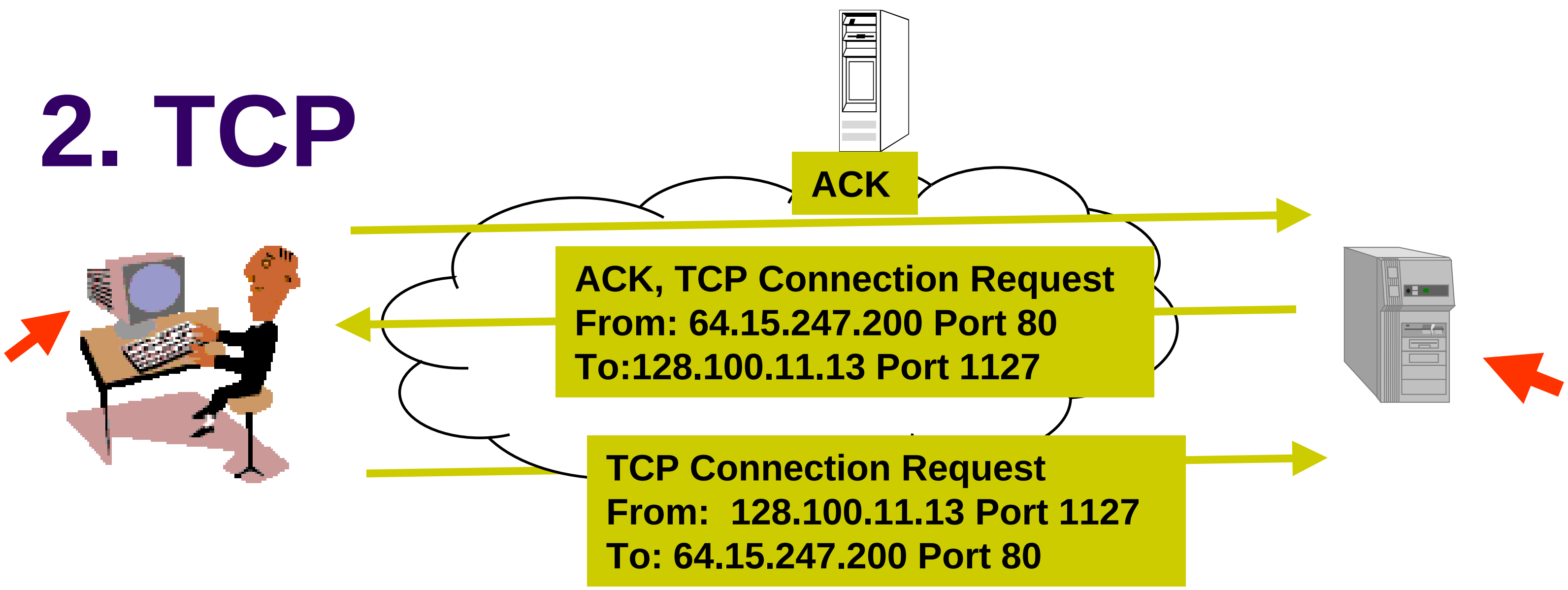
- World Wide Web allows users to access resources (i.e. documents) located in computers connected to the Internet
- Documents are prepared using HyperText Markup Language (HTML)
- A browser application program is used to access the web
- The browser displays HTML documents that include *links* to other documents
- Each link references a *Uniform Resource Locator* (URL) that gives the name of the machine and the location of the given document
- Let's see what happens when a user clicks on a link

1. DNS



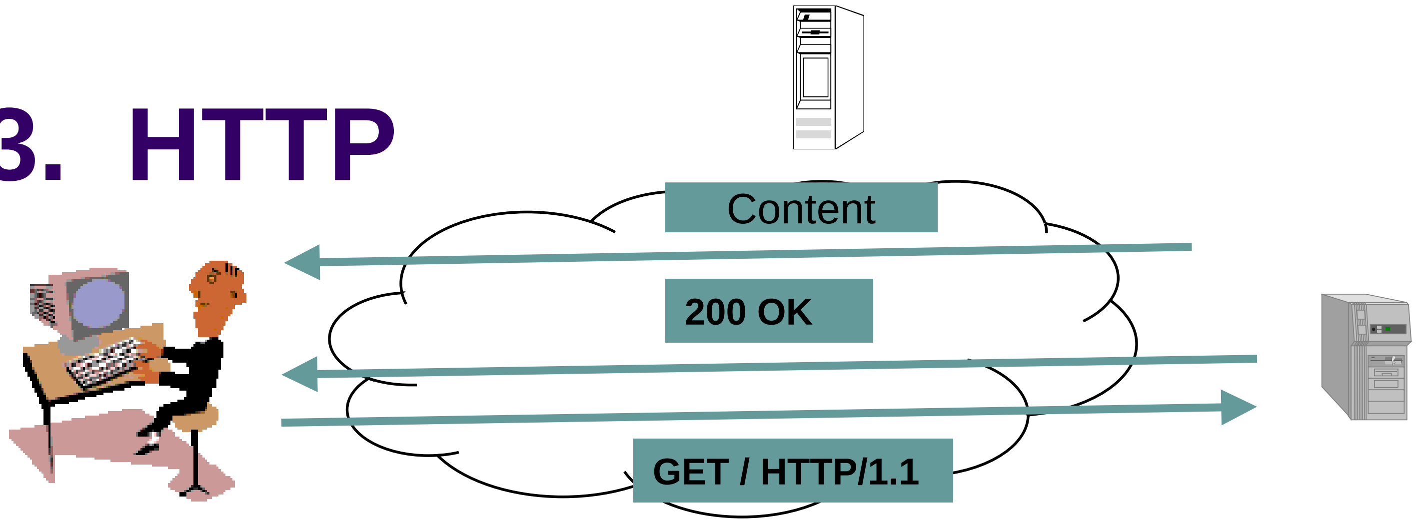
- User clicks on http://www.nytimes.com/
- URL contains Internet name of machine (www.nytimes.com), but not Internet address
- Internet needs Internet address to send information to a machine
- Browser software uses Domain Name System (DNS) protocol to send query for Internet address
- DNS system responds with Internet address

2. TCP



- Browser software uses HyperText Transfer Protocol (HTTP) to send request for document
- HTTP server waits for requests by listening to a well-known port number (80 for HTTP)
- HTTP client sends request messages through an “ephemeral port number,” e.g. 1127
- HTTP needs a Transmission Control Protocol (TCP) connection between the HTTP client and the HTTP server to transfer messages reliably

3. HTTP



- HTTP client sends its request message: “GET …”
- HTTP server sends a status response: “200 OK”
- HTTP server sends requested file
- Browser displays document

- Clicking a link sets off a chain of events across the Internet!
- Let’s see how protocols & layers come into play…

Protocols

- A *protocol* is a set of rules that governs how two or more communicating entities in a layer are to interact
- *Messages* that can be sent and received
- *Actions* that are to be taken when a certain event occurs, e.g. sending or receiving messages, expiry of timers
- **The purpose of a protocol is to provide a service to the layer above**

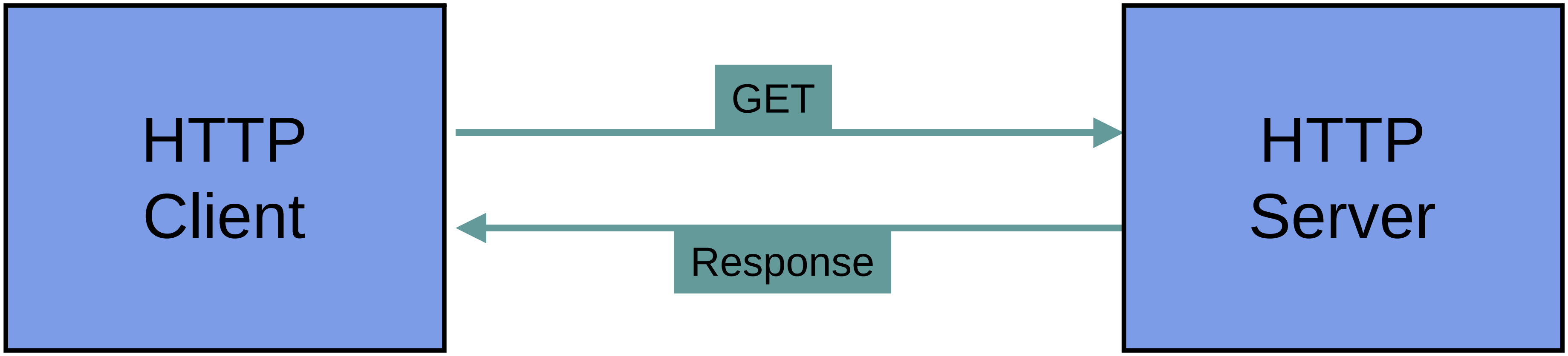
Layers

- A set of related communication functions that can be managed and grouped together
- Application Layer: communications functions that are used by application programs
  - HTTP, DNS, SMTP (email)
- Transport Layer: end-to-end communications between two processes in two machines
  - TCP, User Datagram Protocol (UDP)
- Network Layer: node-to-node communications between two machines
  - Internet Protocol (IP)

Example: HTTP

- HTTP is an application layer protocol
- Retrieves documents on behalf of a browser application program
- HTTP specifies fields in request messages and response messages
  - Request types; Response codes
  - Content type, options, cookies, …
- HTTP specifies actions to be taken upon receipt of certain messages

HTTP Protocol



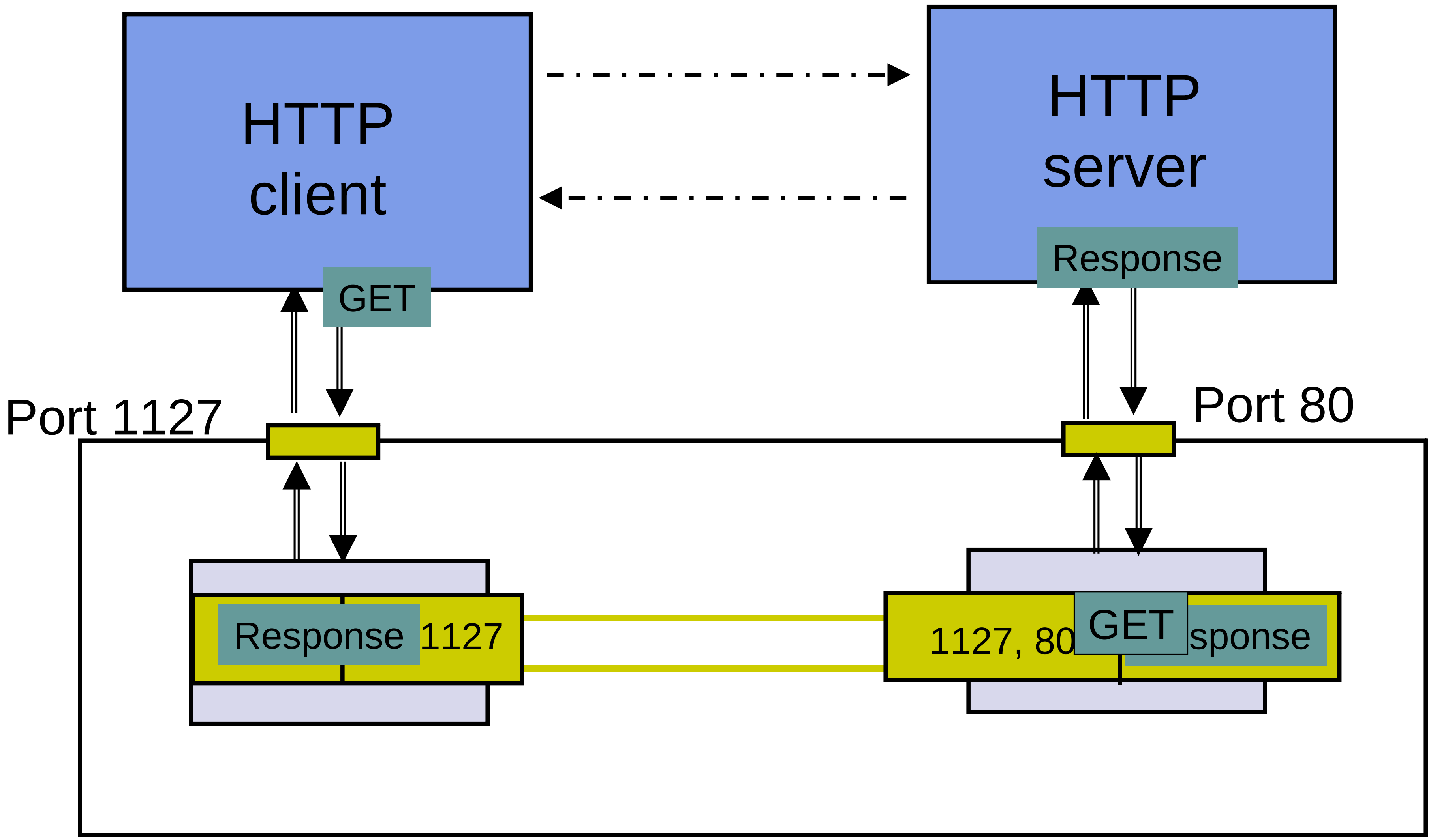
- HTTP assumes messages can be exchanged directly between HTTP client and HTTP server
- In fact, HTTP client and server are processes running in two different machines across the Internet
- HTTP uses the reliable stream transfer service provided by TCP

Example: TCP

- TCP is a transport layer protocol
- Provides *reliable byte stream service* between two processes in two computers across the Internet
- Sequence numbers keep track of the bytes that have been transmitted and received
- Error detection and retransmission used to recover from transmission errors and losses
- TCP is *connection-oriented*: the sender and receiver must first establish an association and set initial sequence numbers before data is transferred
- Connection ID is specified uniquely by

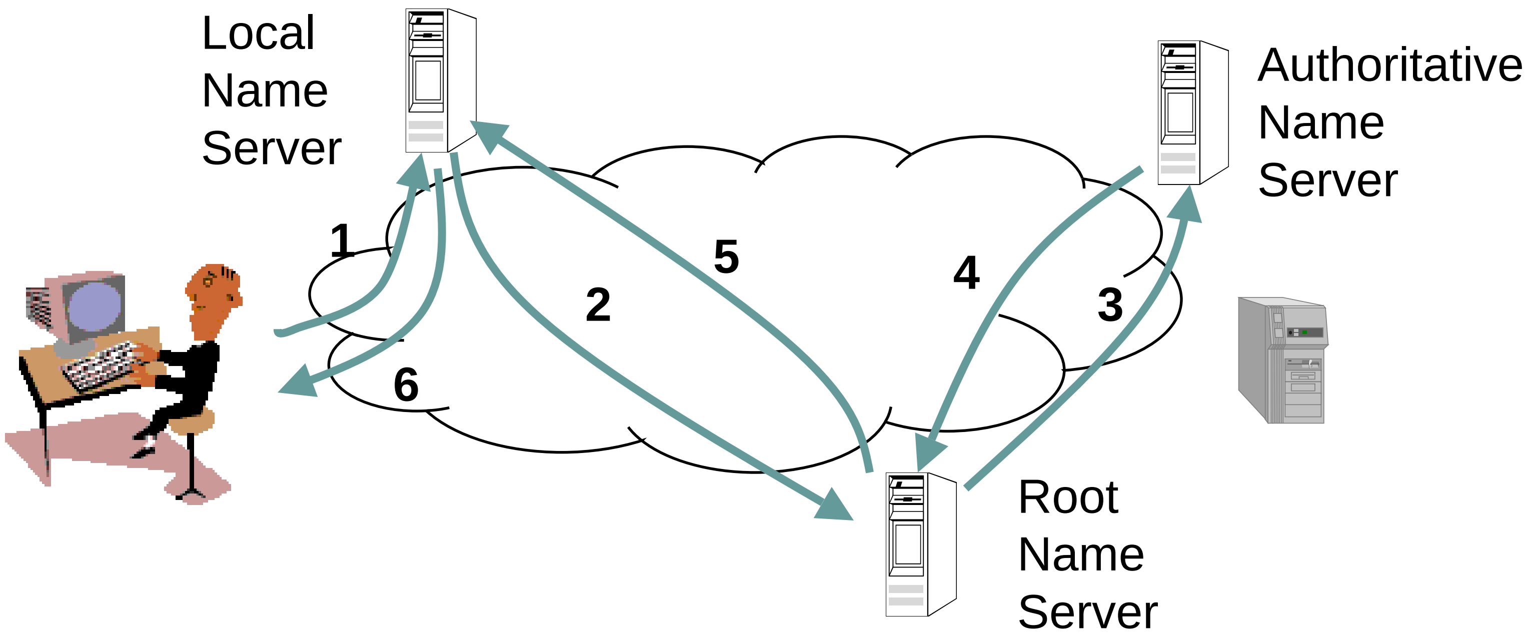
(send port #, send IP address, receive port #, receiver IP address)

HTTP uses service of TCP



Example: DNS Protocol

- DNS protocol is an application layer protocol
- DNS is a distributed database that resides in multiple machines in the Internet
- DNS protocol allows queries of different types
  - Name-to-address or Address-to-name
  - Mail exchange
- DNS usually involves short messages and so uses service provided by UDP
- Well-known port 53



- Local Name Server: resolve frequently-used names
  - University department, ISP
  - Contacts Root Name server if it cannot resolve query
- Root Name Servers: 13 globally
  - Resolves query or refers query to Authoritative Name Server
- Authoritative Name Server: last resort
  - Every machine must register its address with at least two authoritative name servers

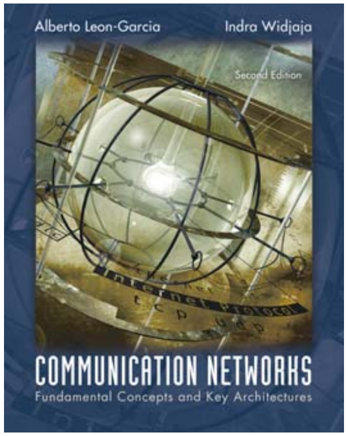
Example: UDP

- UDP is a transport layer protocol
- Provides *best-effort datagram service* between two processes in two computers across the Internet
- Port numbers distinguish various processes in the same machine
- UDP is *connectionless*
- Datagram is sent immediately
- Quick, simple, but not reliable

Summary

- Layers: related communications functions
  - Application Layer: HTTP, DNS
  - Transport Layer: TCP, UDP
  - Network Layer: IP
- Services: a protocol provides a communications service to the layer above
  - TCP provides connection-oriented reliable byte transfer service
  - UDP provides best-effort datagram service
- Each layer builds on services of lower layers
  - HTTP builds on top of TCP
  - DNS builds on top of UDP
  - TCP and UDP build on top of IP

Chapter 2
Applications and Layered Architectures



OSI Reference Model

Why Layering?

- Layering simplifies design, implementation, and testing by partitioning overall communications process into parts
- Protocol in each layer can be designed separately from those in other layers
- Protocol makes “calls” for services from layer below
- Layering provides flexibility for modifying and evolving protocols and services without having to change layers below
- Monolithic non-layered architectures are costly, inflexible, and soon obsolete

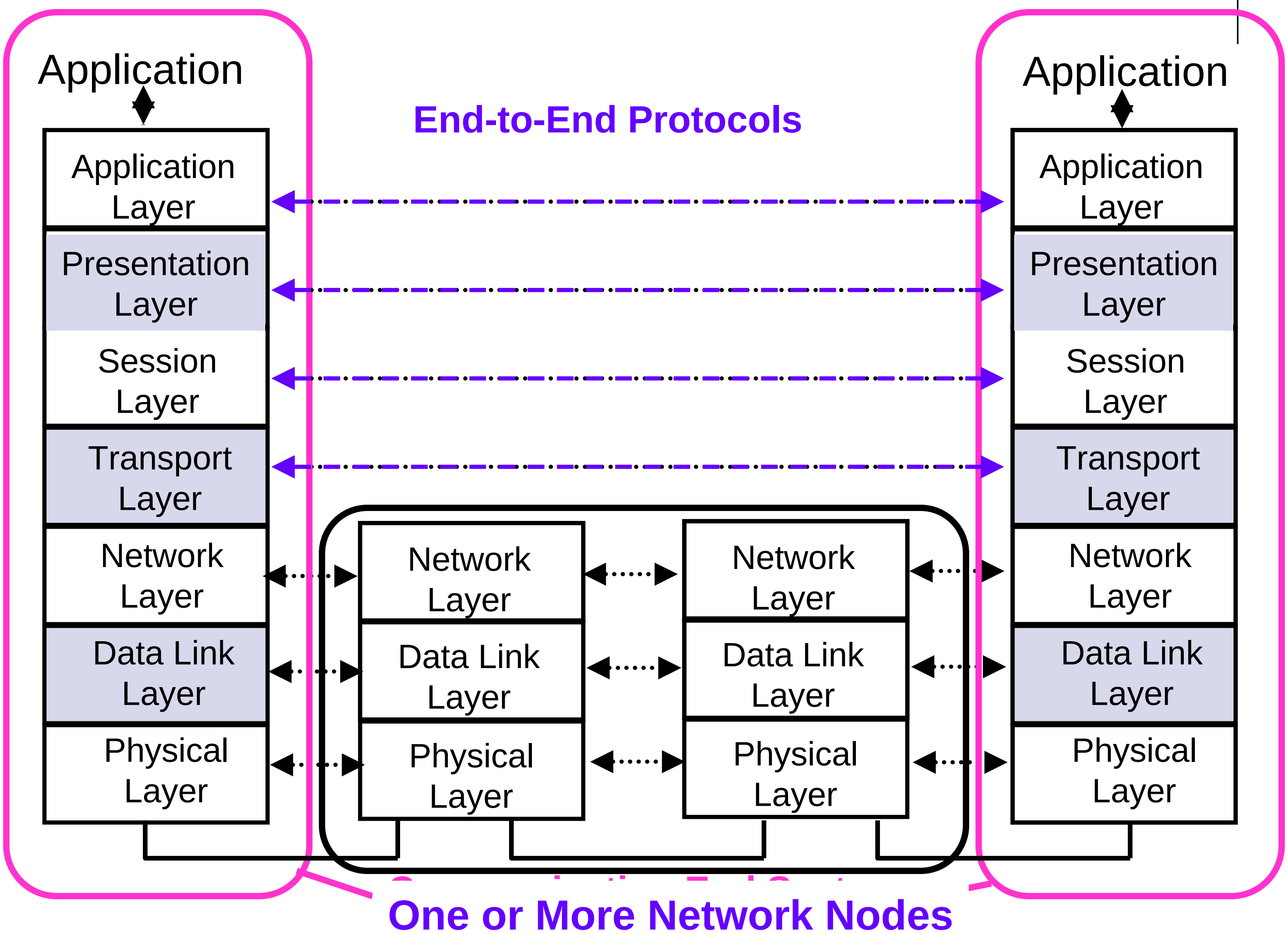
Open Systems Interconnection

- Network architecture:
  - Definition of all the layers
  - Design of protocols for every layer
- By the 1970s every computer vendor had developed its own proprietary layered network architecture
- Problem: computers from different vendors could not be networked together
- Open Systems Interconnection (OSI) was an international effort by the International Organization for Standardization (ISO) to enable multivendor computer interconnection

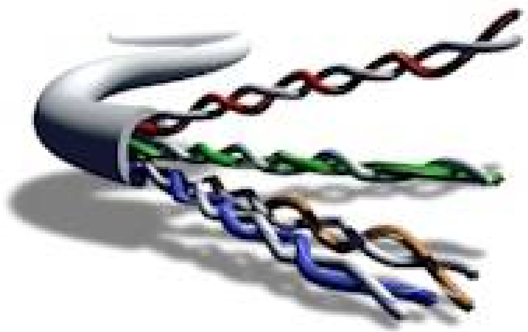
OSI Reference Model

- Describes a seven-layer abstract reference model for a network architecture
- Purpose of the reference model was to provide a framework for the development of protocols
- OSI also provided a unified view of layers, protocols, and services which is still in use in the development of new protocols
- Detailed standards were developed for each layer, but most of these are not in use
- TCP/IP protocols preempted deployment of OSI protocols

7-Layer OSI Reference Model



Physical Layer

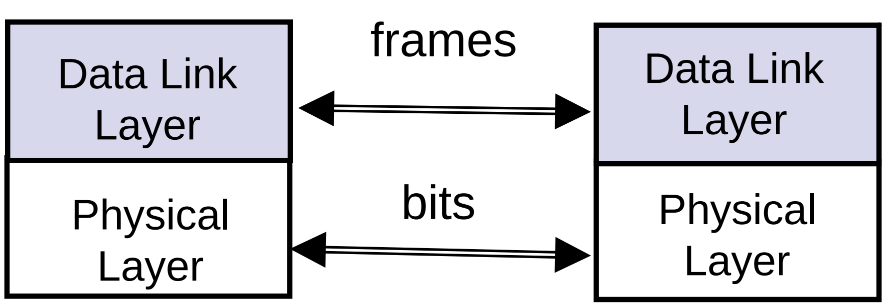


- Transfers bits across link
- Definition & specification of the physical aspects of a communications link
  - Mechanical: cable, plugs, pins...
  - Electrical/optical: modulation, signal strength, voltage levels, bit times, …
  - functional/procedural: how to activate, maintain, and deactivate physical links…

- Ethernet, DSL, cable modem, telephone modems…
- Twisted-pair cable, coaxial cable optical fiber, radio, infrared, …

Data Link Layer

- Transfers *frames* across *direct* connections
- Groups bits into frames
- Detection of bit errors; Retransmission of frames
- Activation, maintenance, & deactivation of data link connections
- Medium access control for local area networks
- Flow control

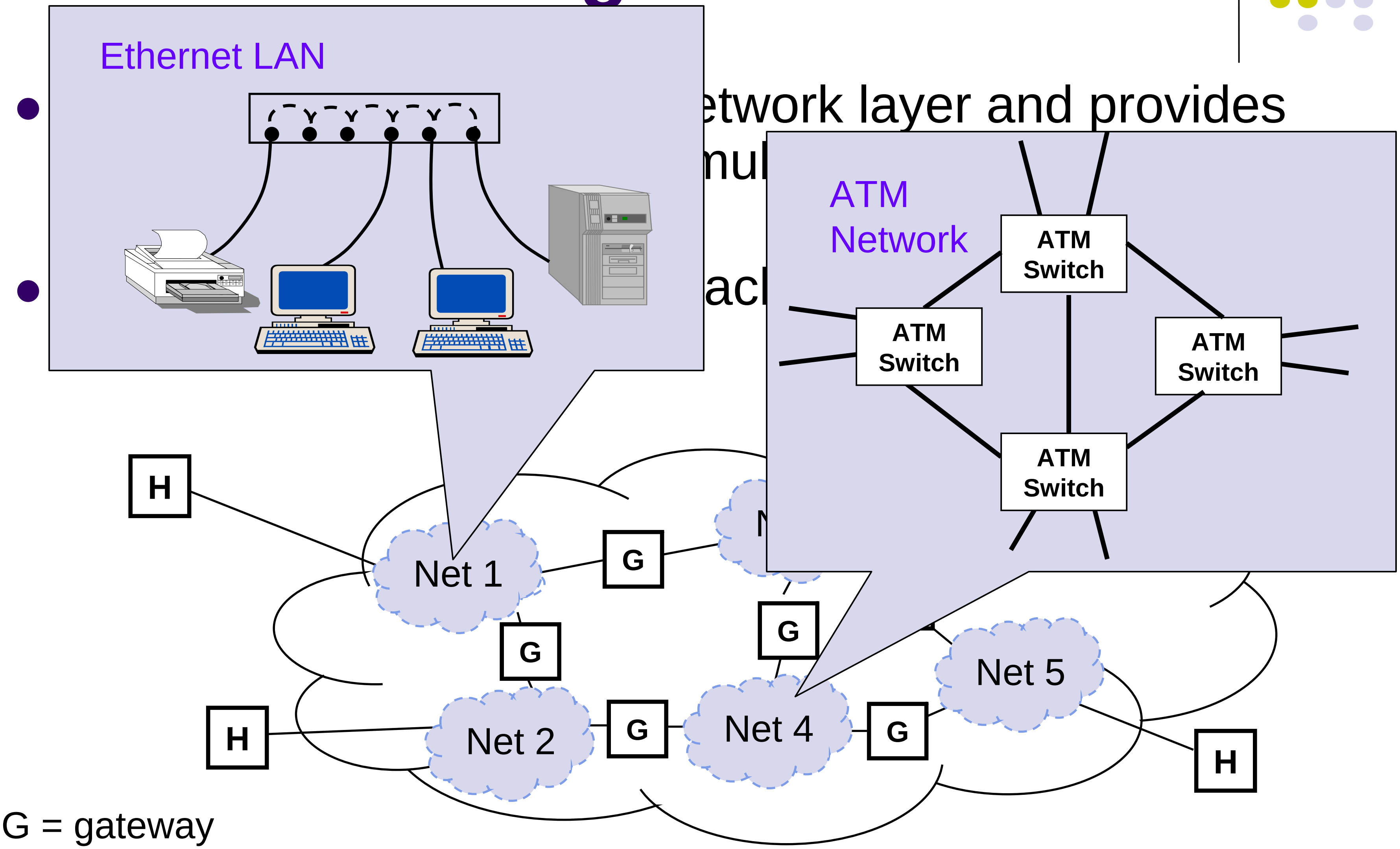


Network Layer

- Transfers *packets* across multiple links and/or multiple networks
- Addressing must scale to large networks
- Nodes *jointly* execute routing algorithm to determine paths across the network
- Forwarding transfers packet across a node
- Congestion control to deal with traffic surges
- Connection setup, maintenance, and teardown when connection-based

Internetworking

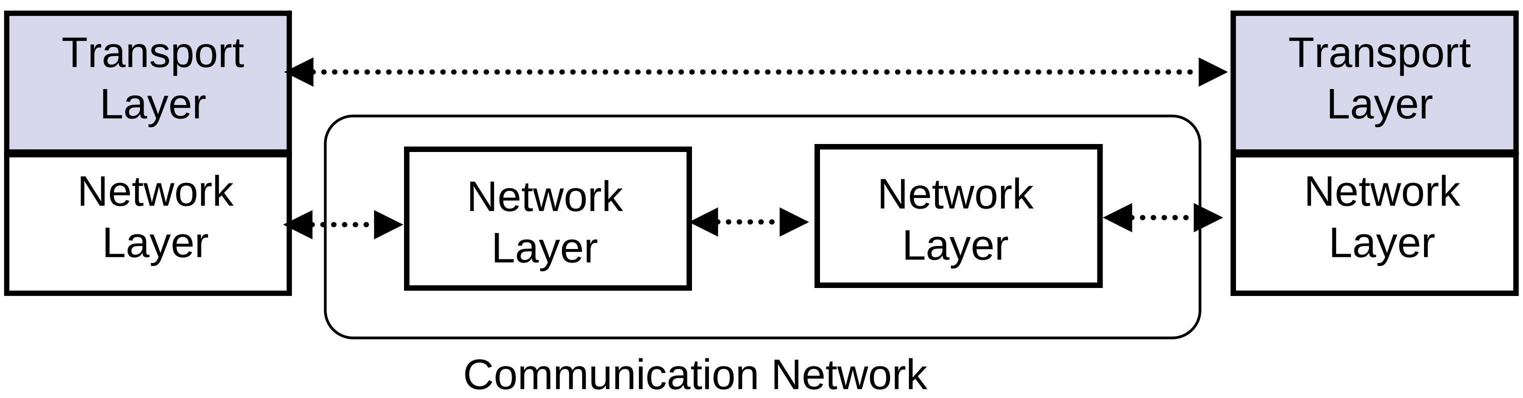
- ...etwork layer and provides ...mul...
- ...ac...



G = gateway
H = host

Transport Layer

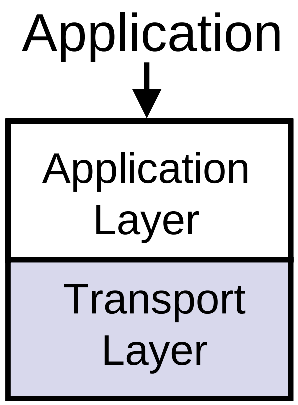
- Transfers data end-to-end from process in a machine to process in another machine
- Reliable stream transfer or quick-and-simple single-block transfer
- Port numbers enable multiplexing
- Message segmentation and reassembly
- Connection setup, maintenance, and release



Application & Upper Layers

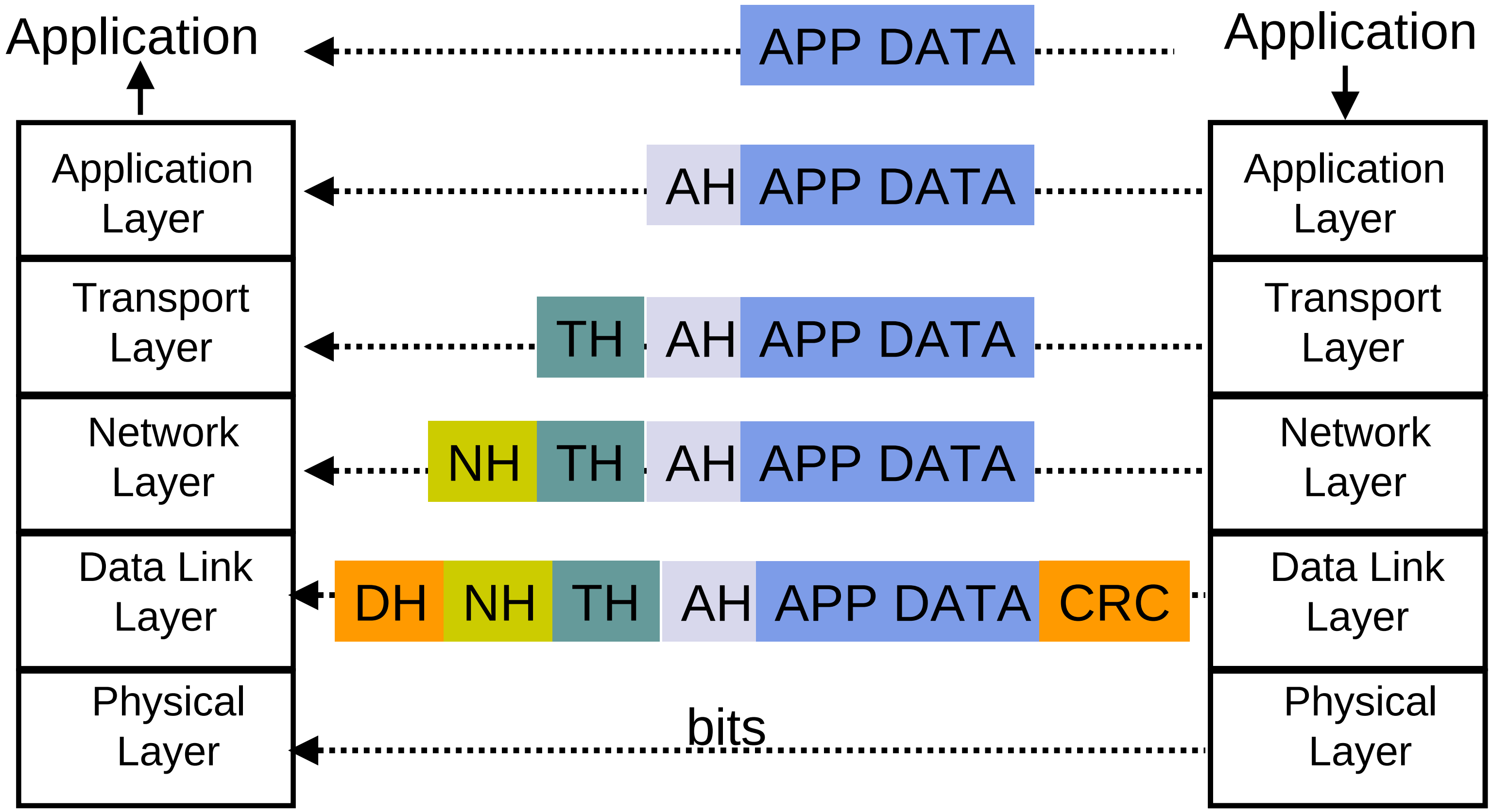
- Application Layer: Provides services that are frequently required by applications: DNS, web acess, file transfer, email…
- ~~Presentation Layer: machine-independent representation of data…~~
- ~~Session Layer: dialog management, recovery from errors, …~~

Incorporated into Application Layer



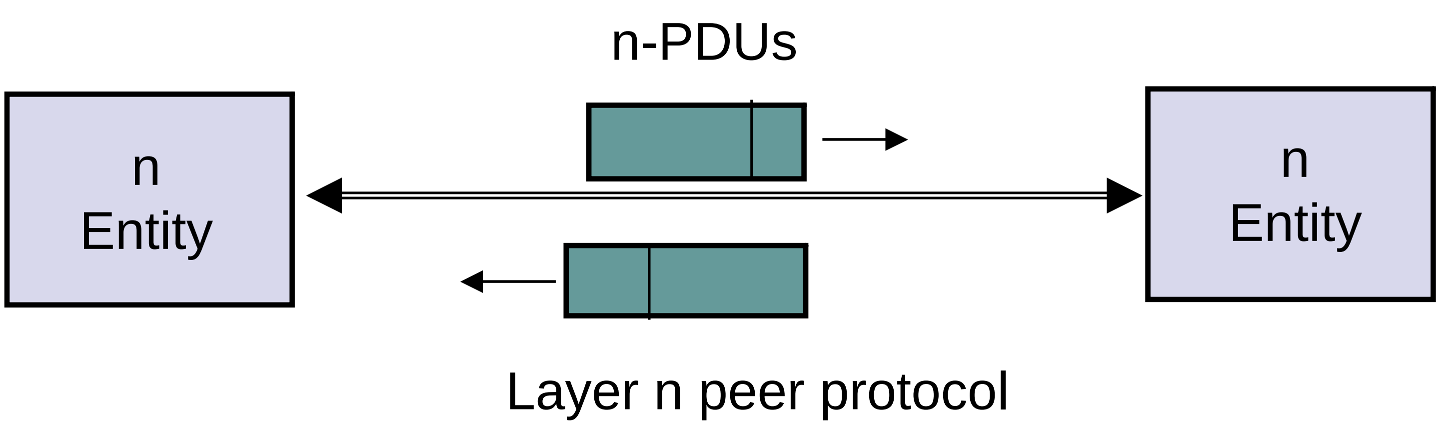
Headers & Trailers

- Each protocol uses a header that carries addresses, sequence numbers, flag bits, length indicators, etc…
- CRC check bits may be appended for error detection



OSI Unified View: Protocols

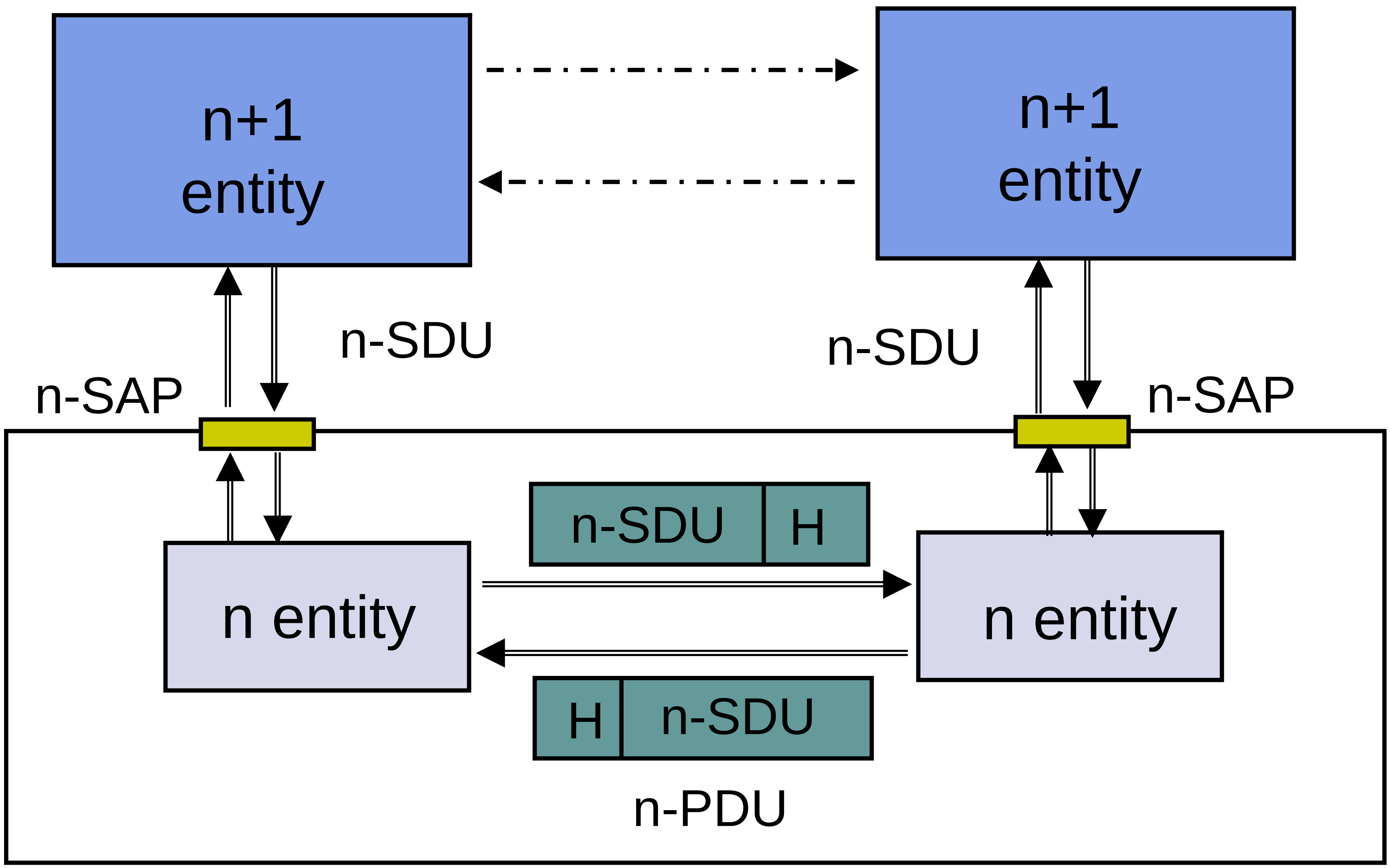
- Layer n in one machine interacts with layer n in another machine to provide a service to layer n +1
- The entities comprising the corresponding layers on different machines are called *peer processes.*
- The machines use a set of rules and conventions called the *layer-n protocol*.
- Layer-n peer processes communicate by exchanging *Protocol Data Units* (PDUs)



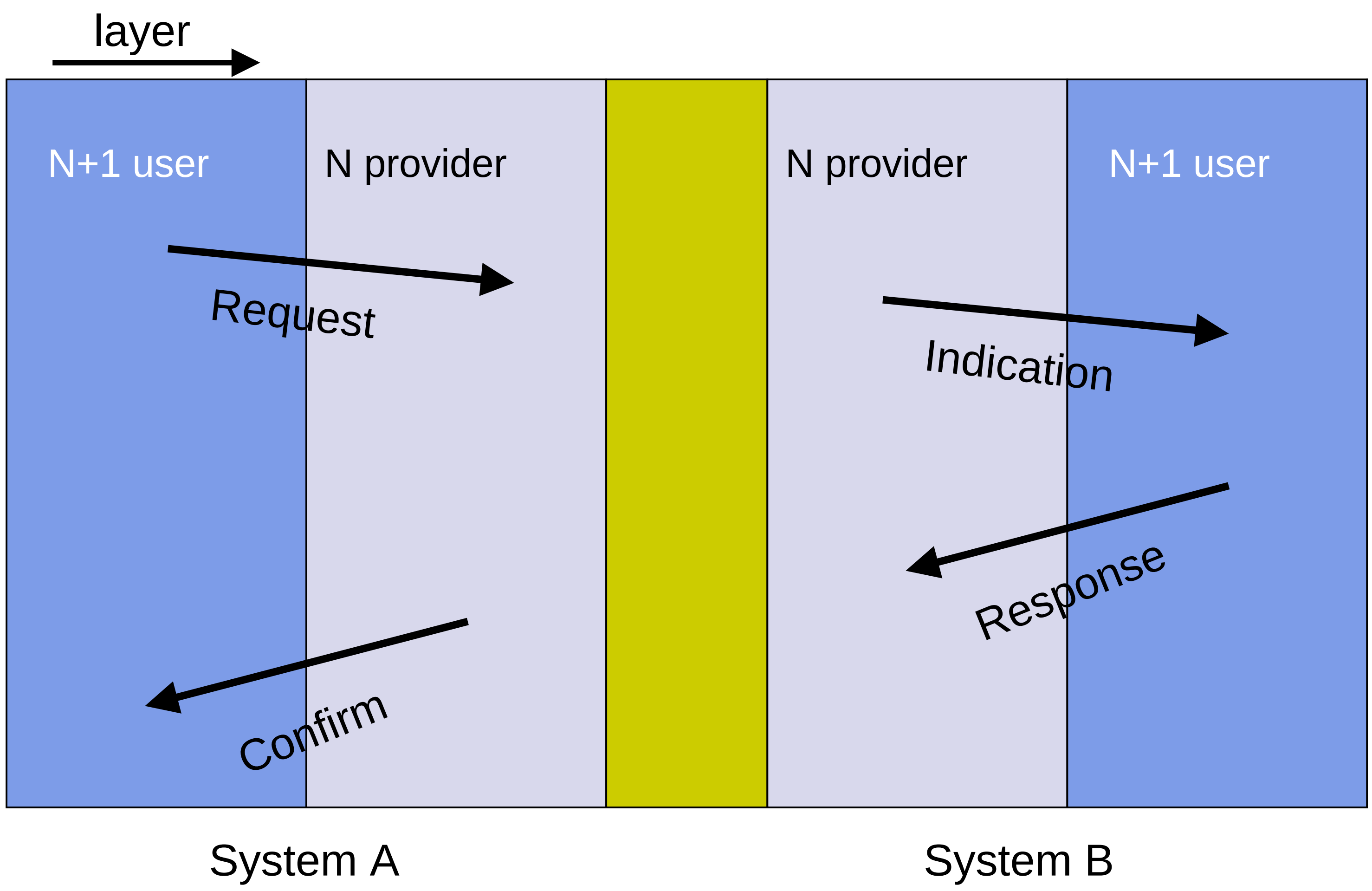
OSI Unified View: Services

- Communication between peer processes is virtual and actually indirect
- Layer n+1 transfers information by invoking the services provided by layer n
- Services are available at *Service Access Points (*SAP's)
- Each layer passes data & control information to the layer below it until the physical layer is reached and transfer occurs
- The data passed to the layer below is called a *Service Data Unit* (SDU)
- SDU's are *encapsulated* in PDU's

Layers, Services & Protocols



Interlayer Interaction

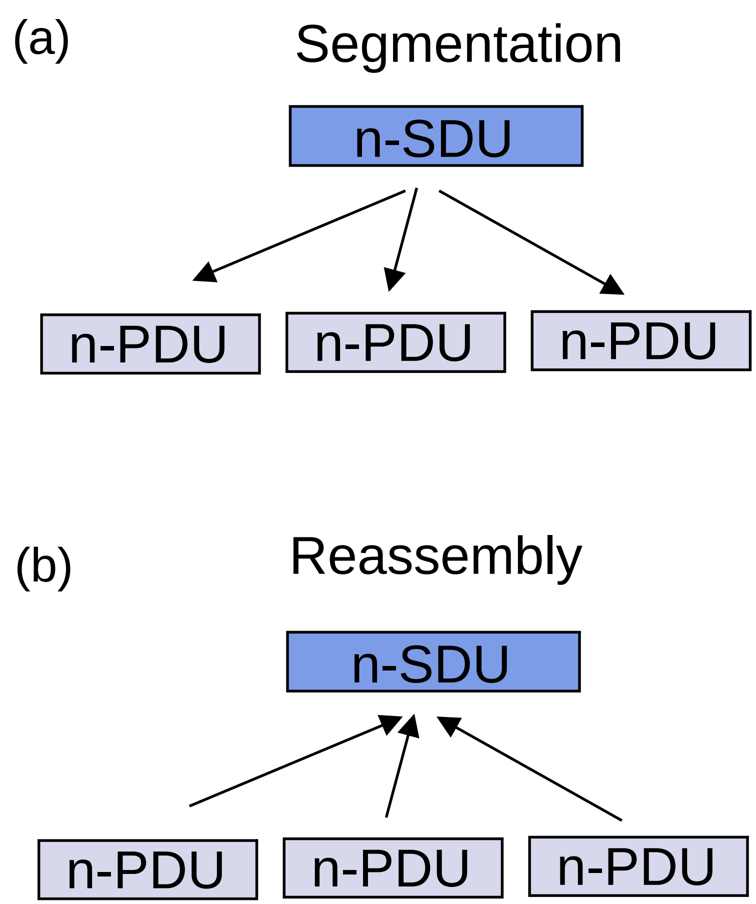


Connectionless & Connection-Oriented Services

- Connection-Oriented
  - Three-phases:
    1. Connection setup between two SAPs to initialize state information
    2. SDU transfer
    3. Connection release
  - E.g. TCP, ATM
- Connectionless
  - Immediate SDU transfer
  - No connection setup
  - E.g. UDP, IP
- Layered services need not be of same type
  - TCP operates over IP
  - IP operates over ATM

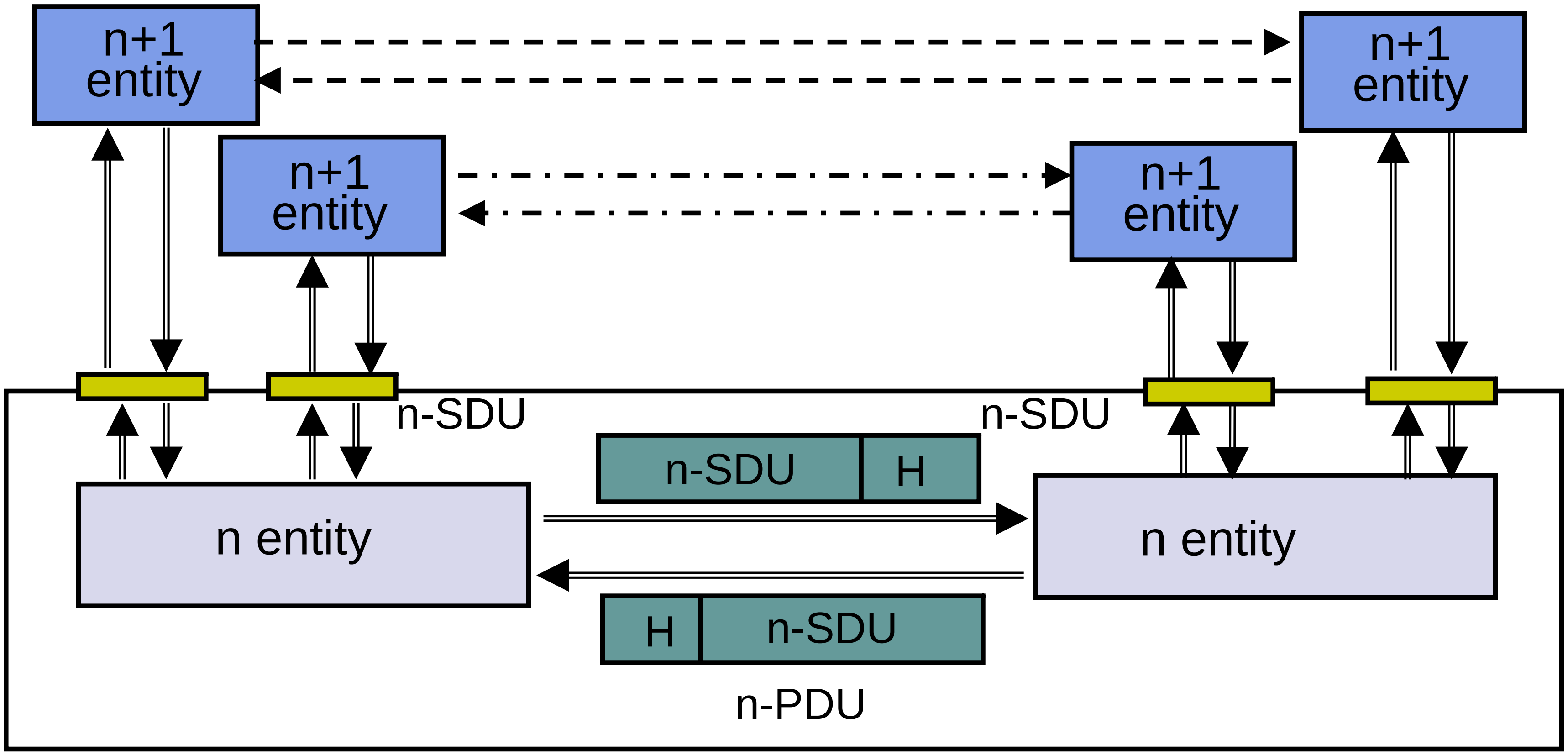
Segmentation & Reassembly

- A layer may impose a limit on the size of a data block that it can transfer for implementation or other reasons
- Thus a layer-n SDU may be too large to be handled as a single unit by layer-(n-1)
- Sender side: SDU is segmented into multiple PDUs
- Receiver side: SDU is reassembled from sequence of PDUs



Multiplexing

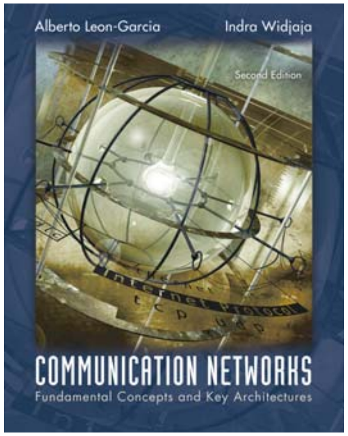
- Sharing of layer n service by *multiple* layer n+1 users
- Multiplexing tag or ID required in each PDU to determine which users an SDU belongs to



Summary

- Layers: related communications functions
  - Application Layer: HTTP, DNS
  - Transport Layer: TCP, UDP
  - Network Layer: IP
- Services: a protocol provides a communications service to the layer above
  - TCP provides connection-oriented reliable byte transfer service
  - UDP provides best-effort datagram service
- Each layer builds on services of lower layers
  - HTTP builds on top of TCP
  - DNS builds on top of UDP
  - TCP and UDP build on top of IP

Chapter 2
Applications and Layered Architectures

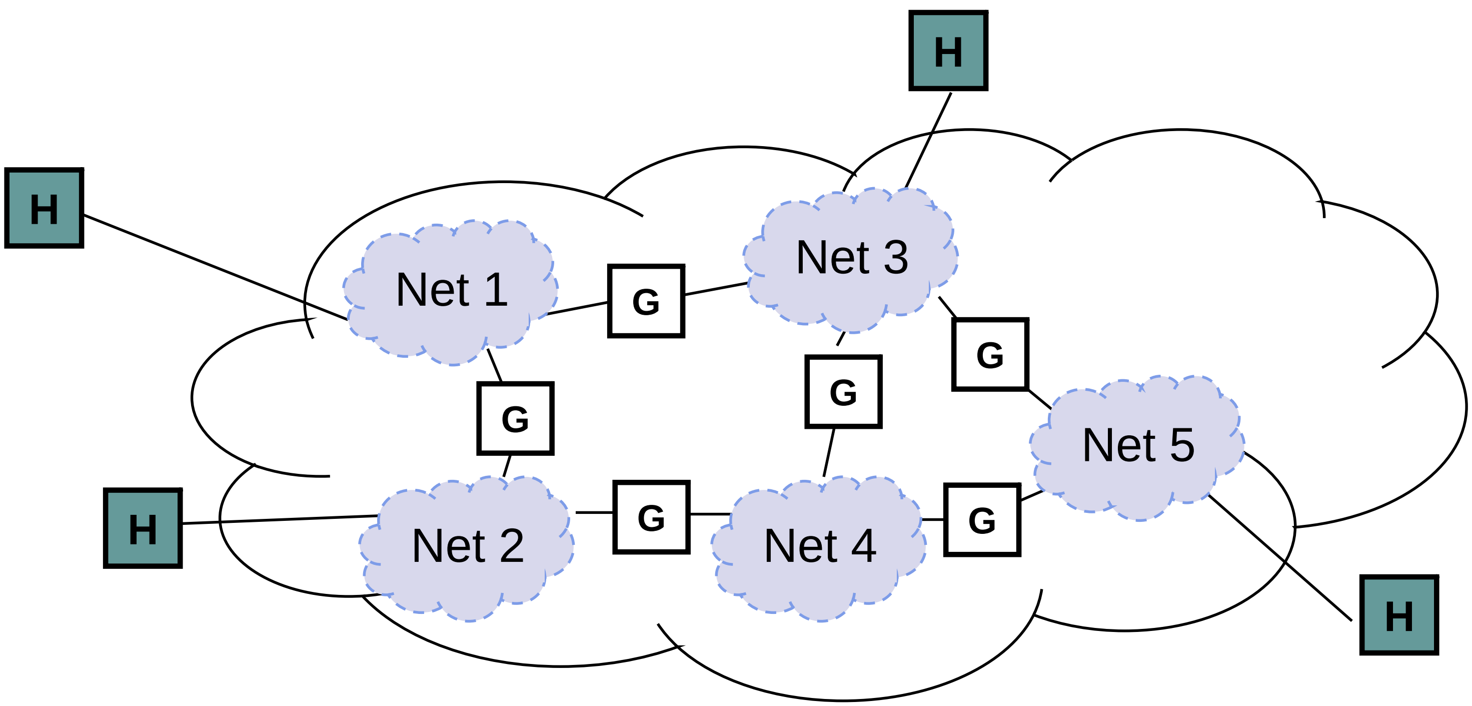


TCP/IP Architecture

How the Layers Work Together

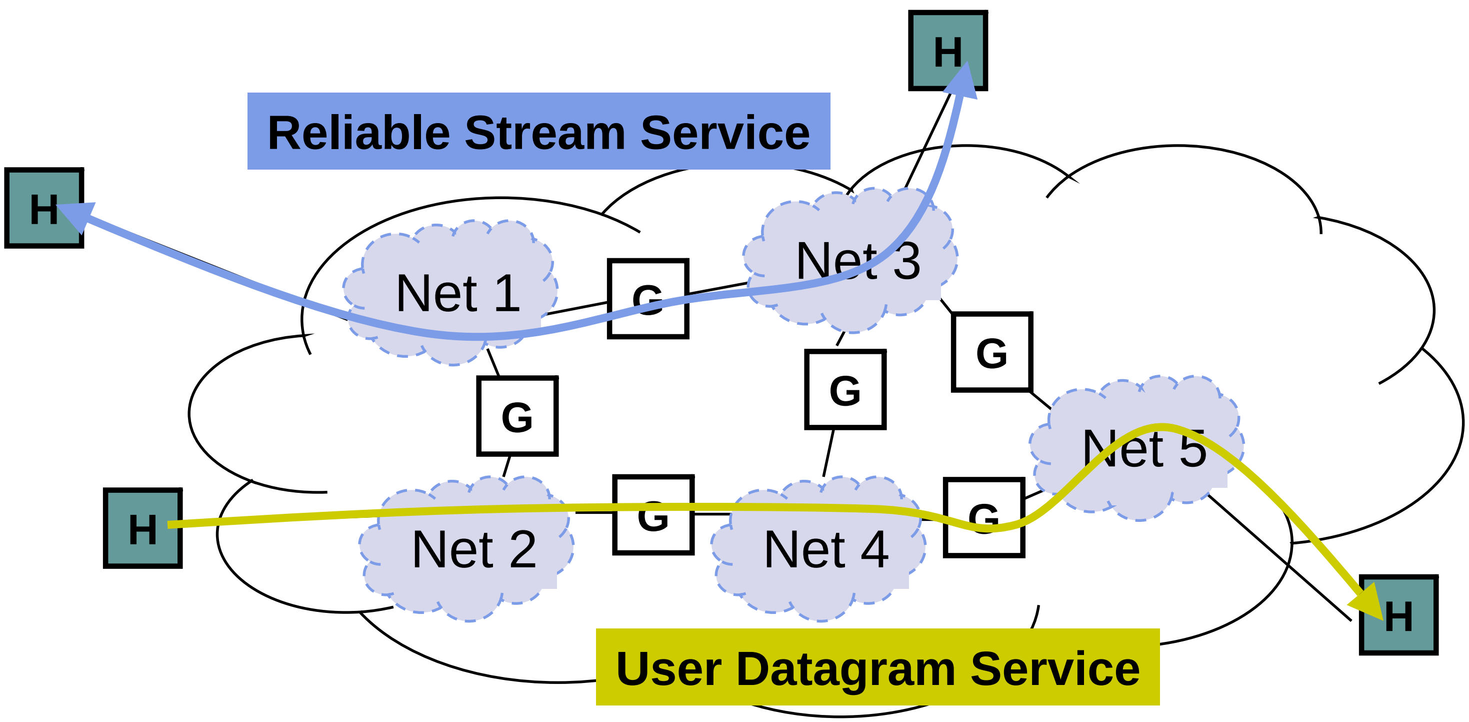
Why Internetworking?

- To build a “network of networks” or internet
  - operating over multiple, coexisting, different network technologies
  - providing ubiquitous connectivity through IP packet transfer
  - achieving huge economies of scale



Why Internetworking?

- To provide *universal communication services*
  - independent of underlying network technologies
  - providing common interface to user applications



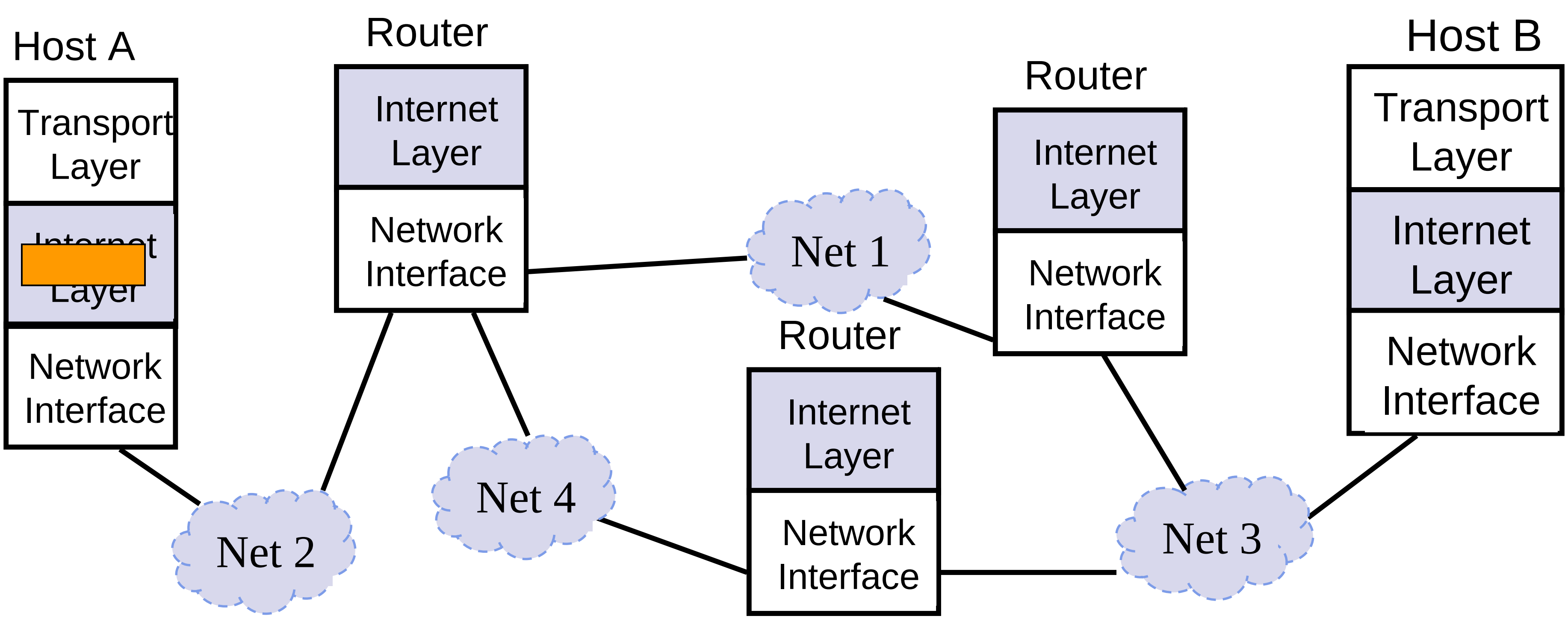
Why Internetworking?

- To provide *distributed applications*
  - Any application designed to operate based on Internet communication services immediately operates across the entire Internet
  - Rapid deployment of new applications
    - Email, WWW, Peer-to-peer
  - Applications independent of network technology
    - New networks can be introduced below
    - Old network technologies can be retired

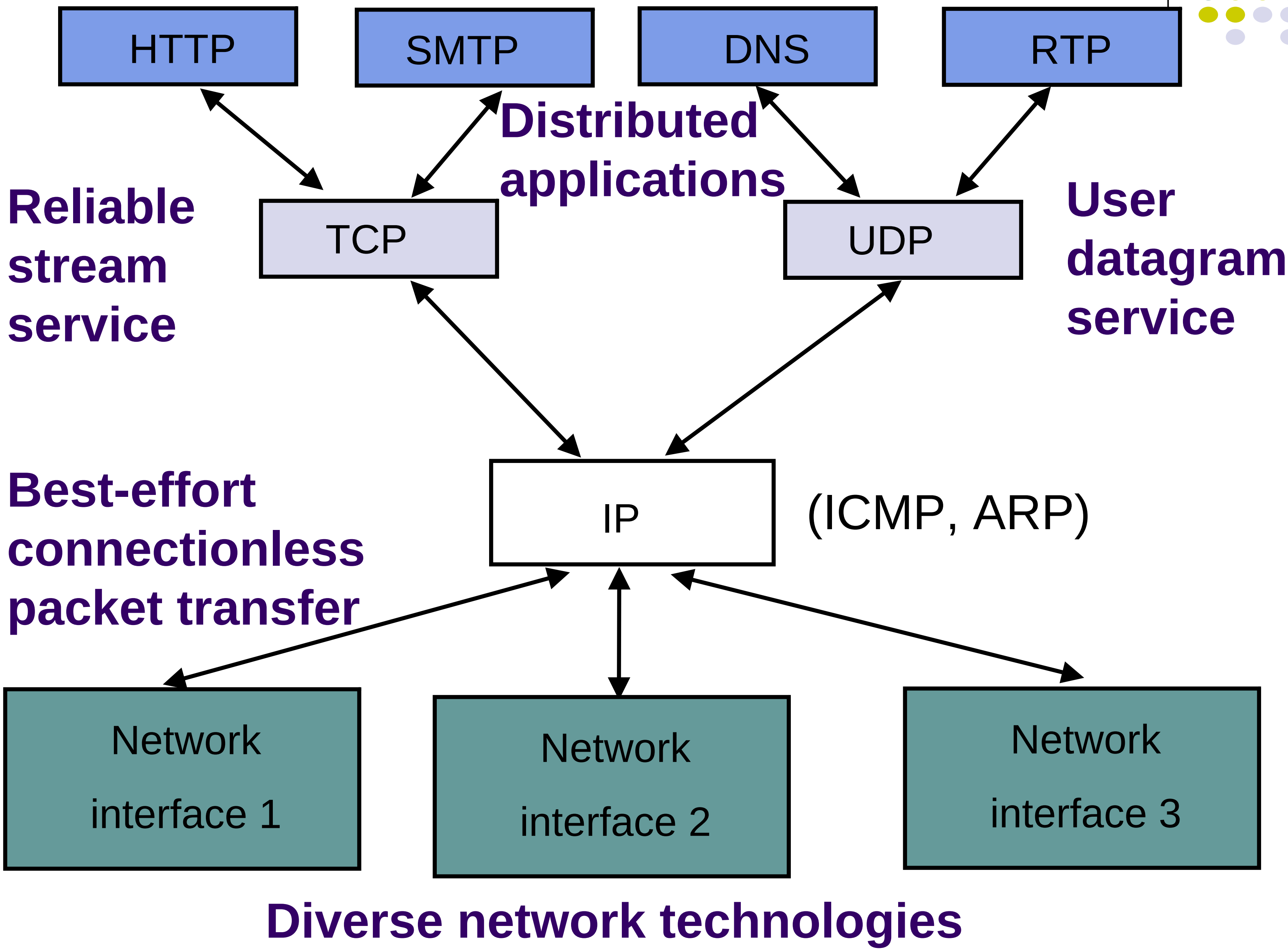
Internet Protocol Approach

- IP packets transfer information across Internet

  Host A IP → router→ router…→ router→ Host B IP
- IP layer in each router determines next hop (router)
- Network interfaces transfer IP packets across networks



TCP/IP Protocol Suite



Internet Names & Addresses

Internet Names

- Each host a a unique name
  - Independent of physical location
  - Facilitate memorization by humans
  - Domain Name
  - Organization under single administrative unit
- Host Name
  - Name given to host computer
- User Name
  - Name assigned to user

leongarcia@comm.utoronto.ca

Internet Addresses

- Each host has globally unique *logical* 32 bit IP address
- Separate address for each physical connection to a network
- Routing decision is done based on destination IP address
- IP address has two parts:
  - *netid* and *hostid*
  - *netid* unique
  - *netid* facilitates routing
- Dotted Decimal Notation:

  int1.int2.int3.int4

  (intj = jth octet)

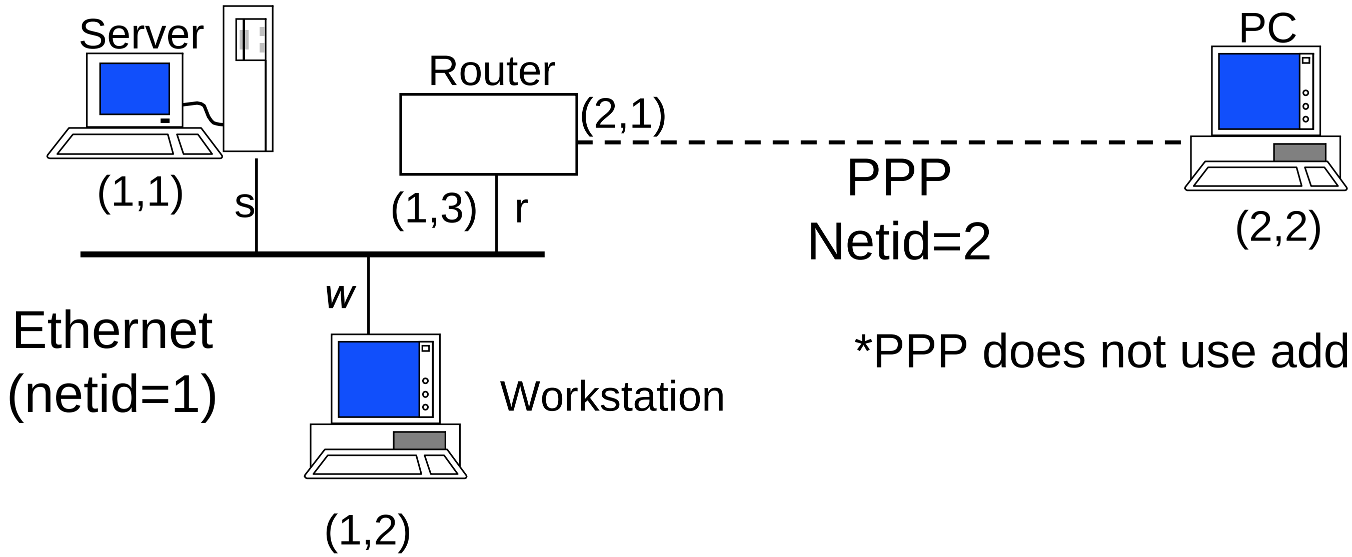
  128.100.10.13

DNS resolves IP name to IP address

Physical Addresses

- LANs (and other networks) assign physical addresses to the physical attachment to the network
- The network uses its own address to transfer packets or frames to the appropriate destination
- IP address needs to be resolved to physical address at each IP network interface
- Example: Ethernet uses 48-bit addresses
  - Each Ethernet network interface card (NIC) has globally unique Medium Access Control (MAC) or physical address
  - First 24 bits identify NIC manufacturer; second 24 bits are serial number
  - $\underbrace{00:90:27}_{\text{Intel}}:96:68:07$ 12 hex numbers

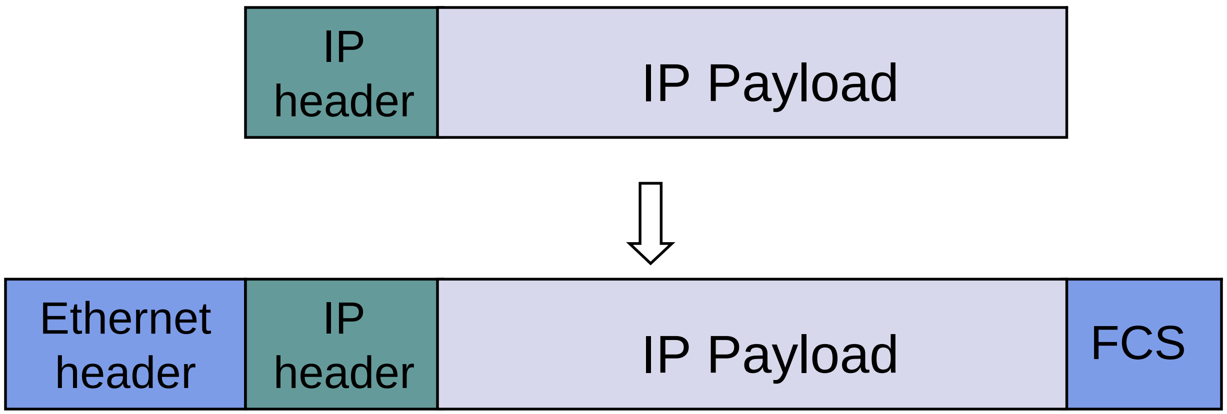
Example internet



*PPP does not use addresses

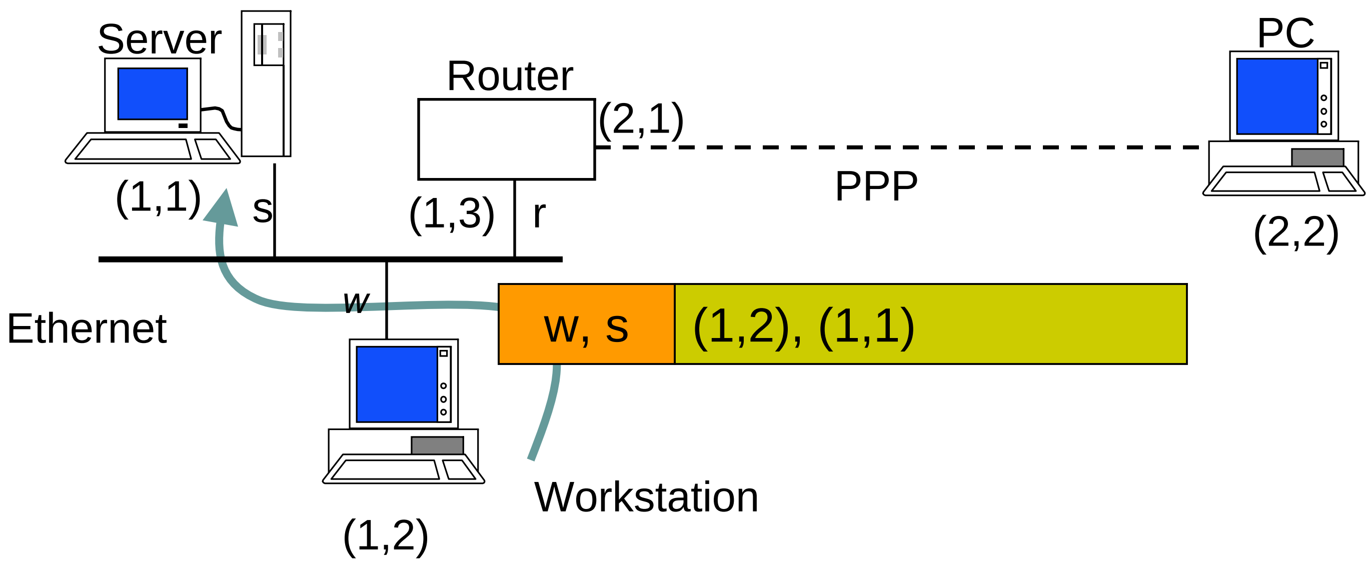
	netid	hostid	Physical address
server	1	1	s
workstation	1	2	w
router	1	3	r
router	2	1	-
PC	2	2	-

Encapsulation



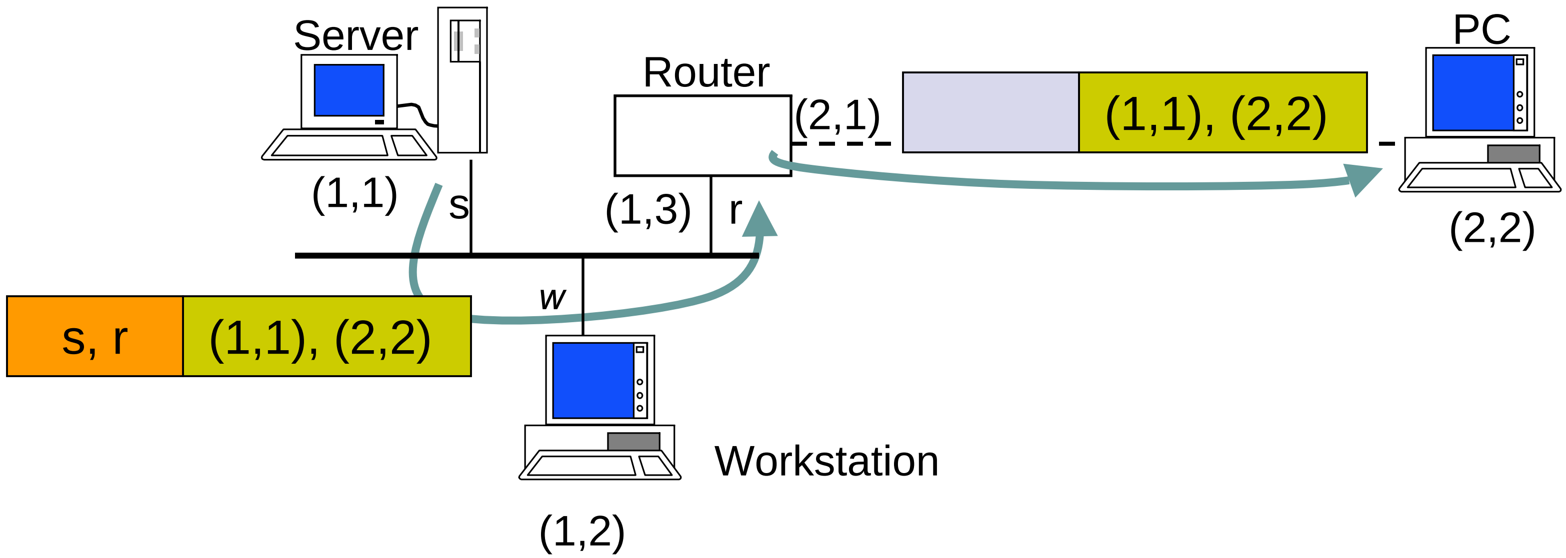
- Ethernet header contains:
  - source and destination physical addresses
  - network protocol type (e.g. IP)

IP packet from workstation to server



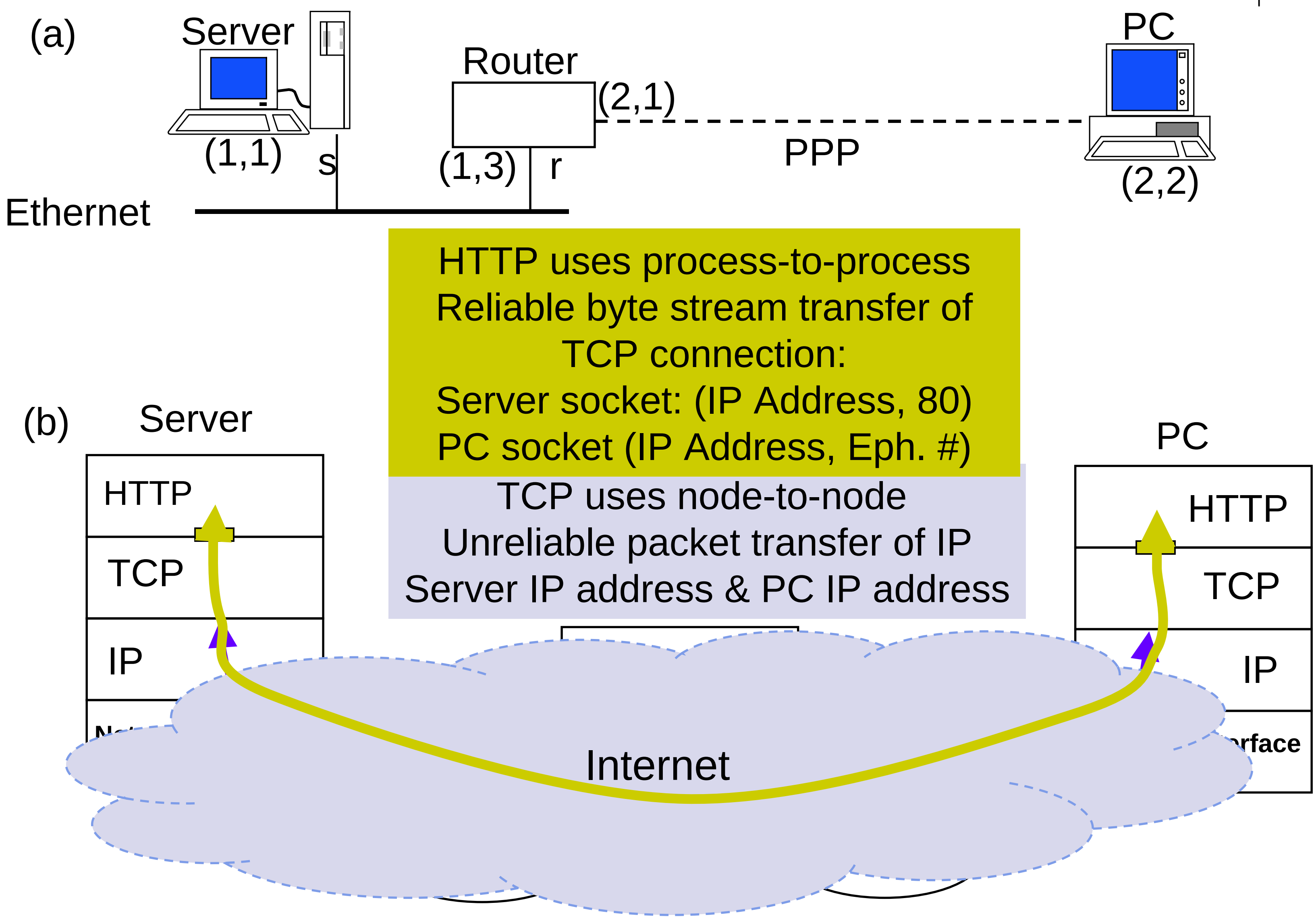
1. IP packet has (1,2) IP address for source and (1,1) IP address for destination
2. IP table at workstation indicates (1,1) connected to same network, so IP packet is encapsulated in Ethernet frame with addresses w and s
3. Ethernet frame is broadcast by workstation NIC and captured by server NIC
4. NIC examines protocol type field and then delivers packet to its IP layer

IP packet from server to PC



1. IP packet has (1,1) and (2,2) as IP source and destination addresses
2. IP table at server indicates packet should be sent to router, so IP packet is encapsulated in Ethernet frame with addresses s and r
3. Ethernet frame is broadcast by server NIC and captured by router NIC
4. NIC examines protocol type field and then delivers packet to its IP layer
5. IP layer examines IP packet destination address and determines IP packet should be routed to (2,2)
6. Router's table indicates (2,2) is directly connected via PPP link
7. IP packet is encapsulated in PPP frame and delivered to PC
8. PPP at PC examines protocol type field and delivers packet to PC IP layer

How the layers work together

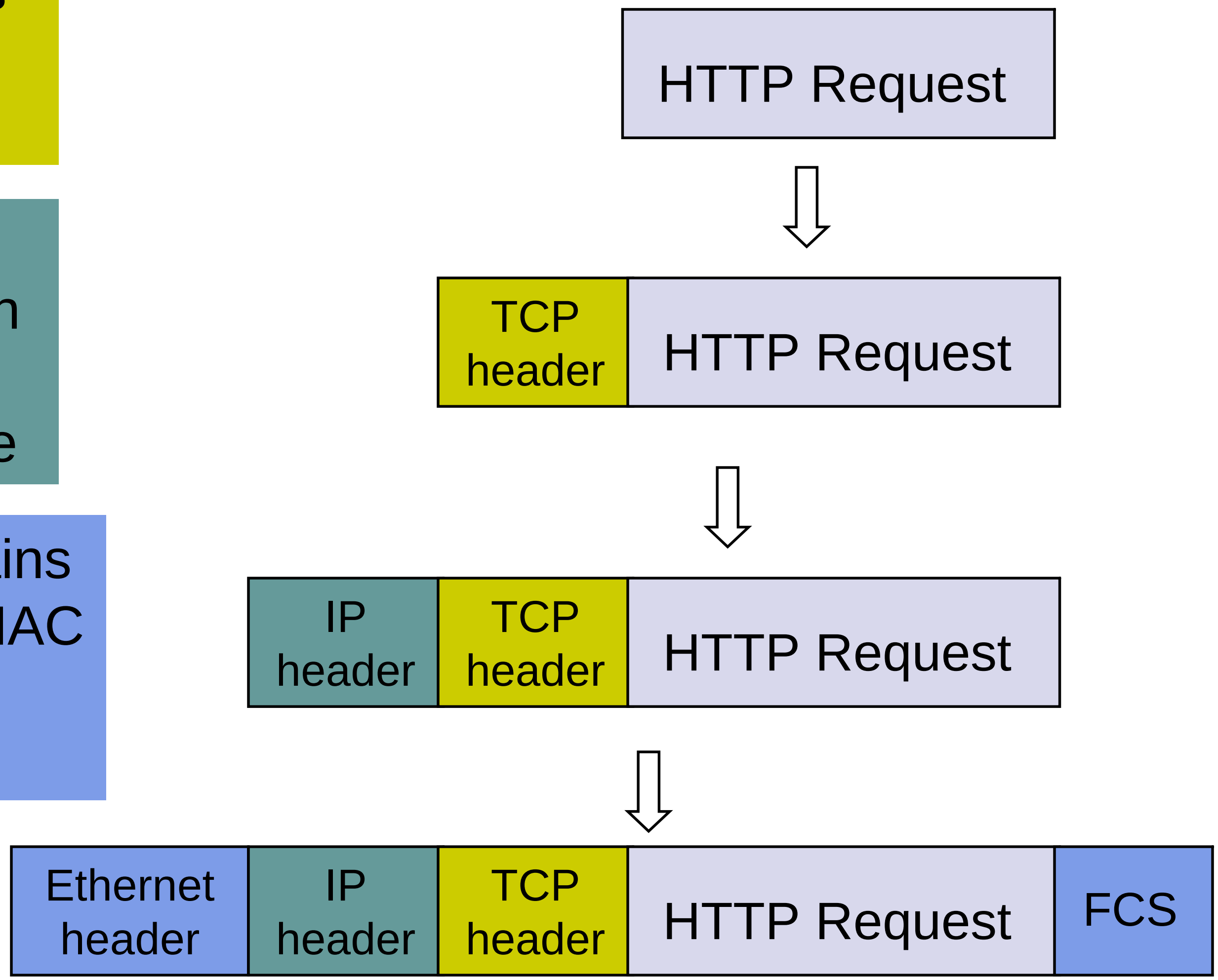


Encapsulation

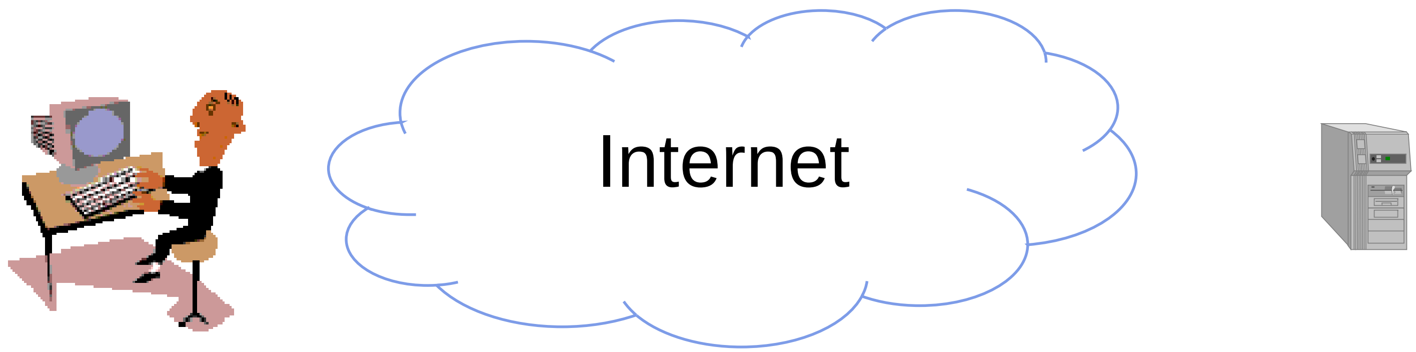
TCP Header contains source & destination port numbers

IP Header contains source and destination IP addresses; transport protocol type

Ethernet Header contains source & destination MAC addresses; network protocol type

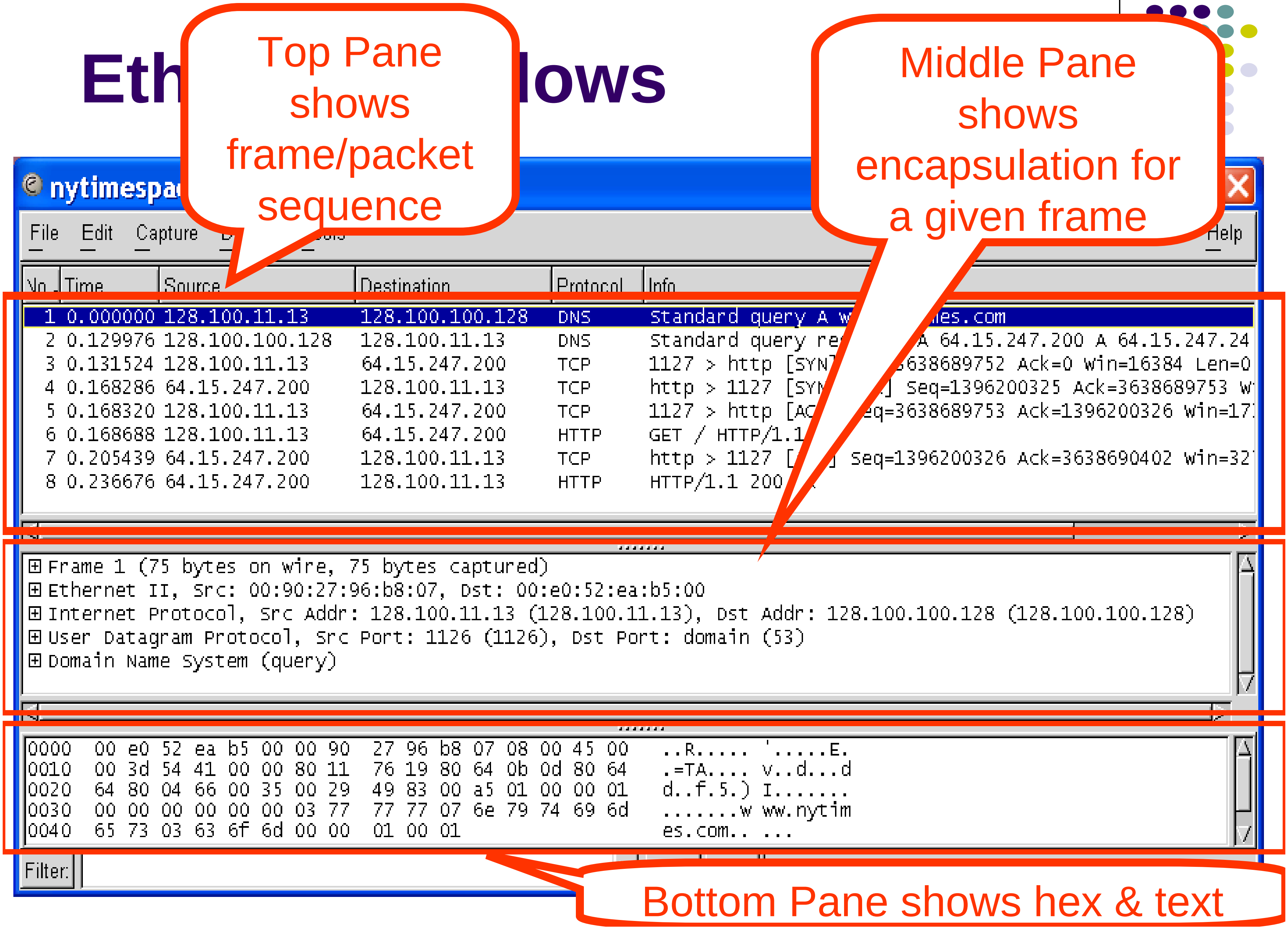


How the layers work together: Network Analyzer Example



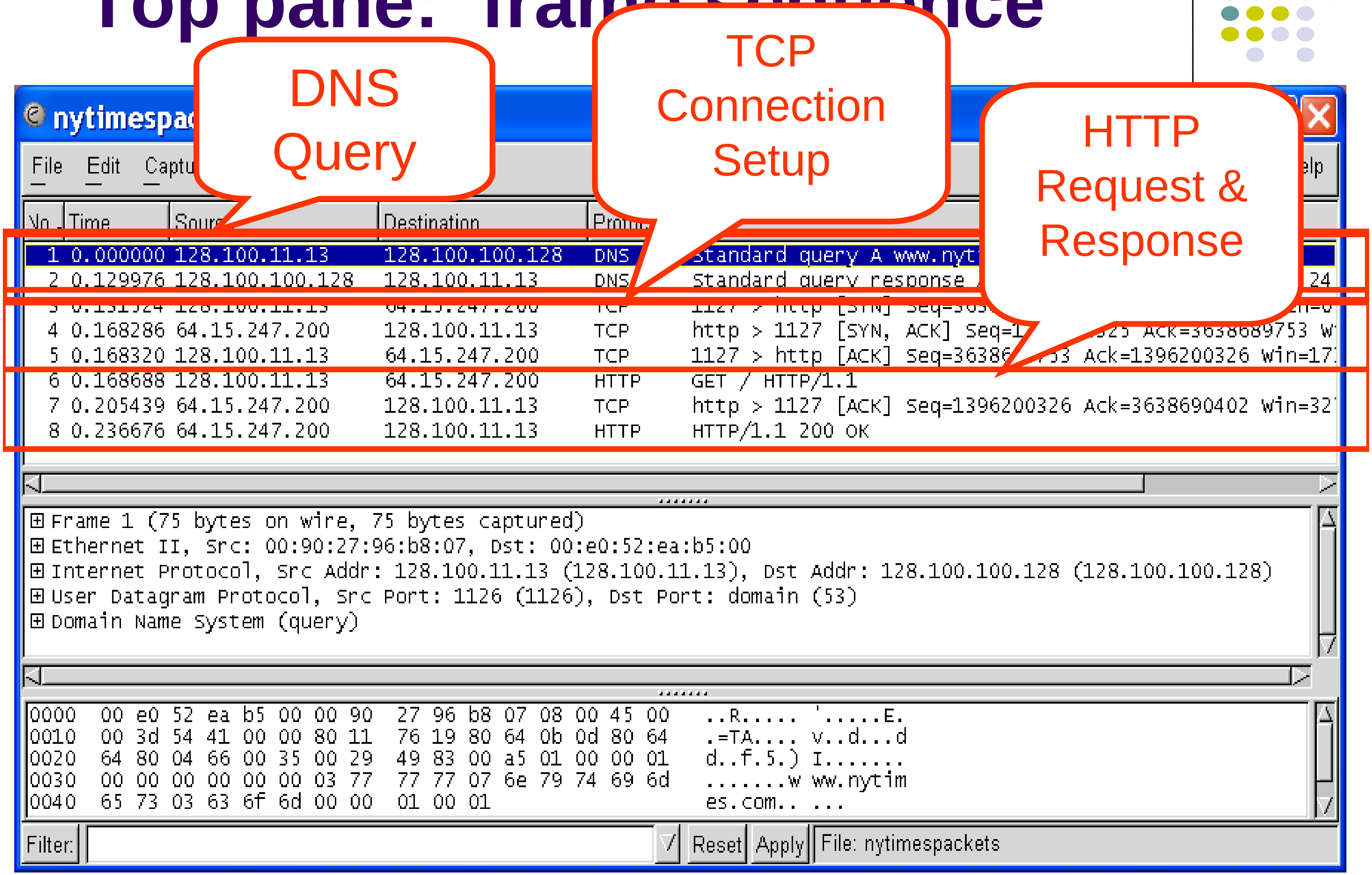
- User clicks on http://www.nytimes.com/
- *Ethereal* network analyzer captures all frames observed by its Ethernet NIC
- Sequence of frames and contents of frame can be examined in detail down to individual bytes

Eth lows

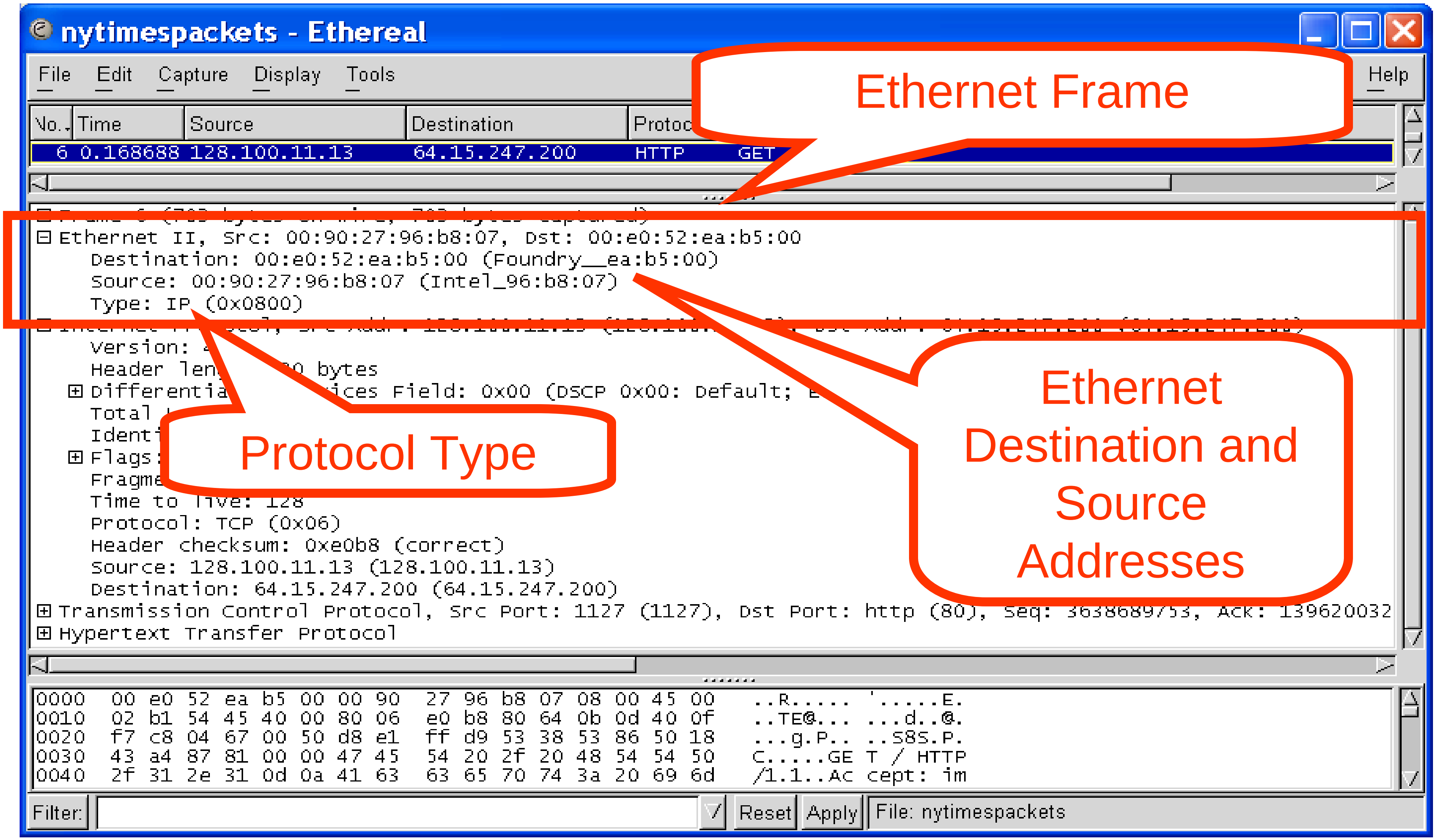


Top Pane shows frame/packet sequence
Middle Pane shows encapsulation for a given frame
Bottom Pane shows hex & text
nytimespa
File Edit Capture
Help
No. Time Source Destination Protocol Info
1 0.000000 128.100.11.13 128.100.100.128 DNS Standard query A w
es.com
2 0.129976 128.100.100.128 128.100.11.13 DNS Standard query re
A 64.15.247.200 A 64.15.247.24
3 0.131524 128.100.11.13 64.15.247.200 TCP 1127 > http [SYN]
Ack=0 Win=16384 Len=0
4 0.168286 64.15.247.200 128.100.11.13 TCP http > 1127 [SYN
Seq=1396200325 Ack=3638689753
5 0.168320 128.100.11.13 64.15.247.200 TCP 1127 > http [AC
eq=3638689753 Ack=1396200326
6 0.168688 128.100.11.13 64.15.247.200 HTTP GET / HTTP/1.1
7 0.205439 64.15.247.200 128.100.11.13 TCP http > 1127
Seq=1396200326 Ack=3638690402
8 0.236676 64.15.247.200 128.100.11.13 HTTP HTTP/1.1 200
Frame 1 (75 bytes on wire, 75 bytes captured)
Ethernet II, Src: 00:90:27:96:b8:07, Dst: 00:e0:52:ea:b5:00
Internet Protocol, Src Addr: 128.100.11.13 (128.100.11.13), Dst Addr: 128.100.100.128 (128.100.100.128)
User Datagram Protocol, Src Port: 1126 (1126), Dst Port: domain (53)
Domain Name System (query)
0000 00 e0 52 ea b5 00 00 90 27 96 b8 07 08 00 45 00 ..R..... '.....E.
0010 00 3d 54 41 00 00 80 11 76 19 80 64 0b 0d 80 64 .=TA.... v..d...d
0020 64 80 04 66 00 35 00 29 49 83 00 a5 01 00 00 01 d..f.5.) I.......
0030 00 00 00 00 00 00 03 77 77 77 07 6e 79 74 69 6d ....w ww.nytim
0040 65 73 03 63 6f 6d 00 00 01 00 01 es.com.. ...
Filter:

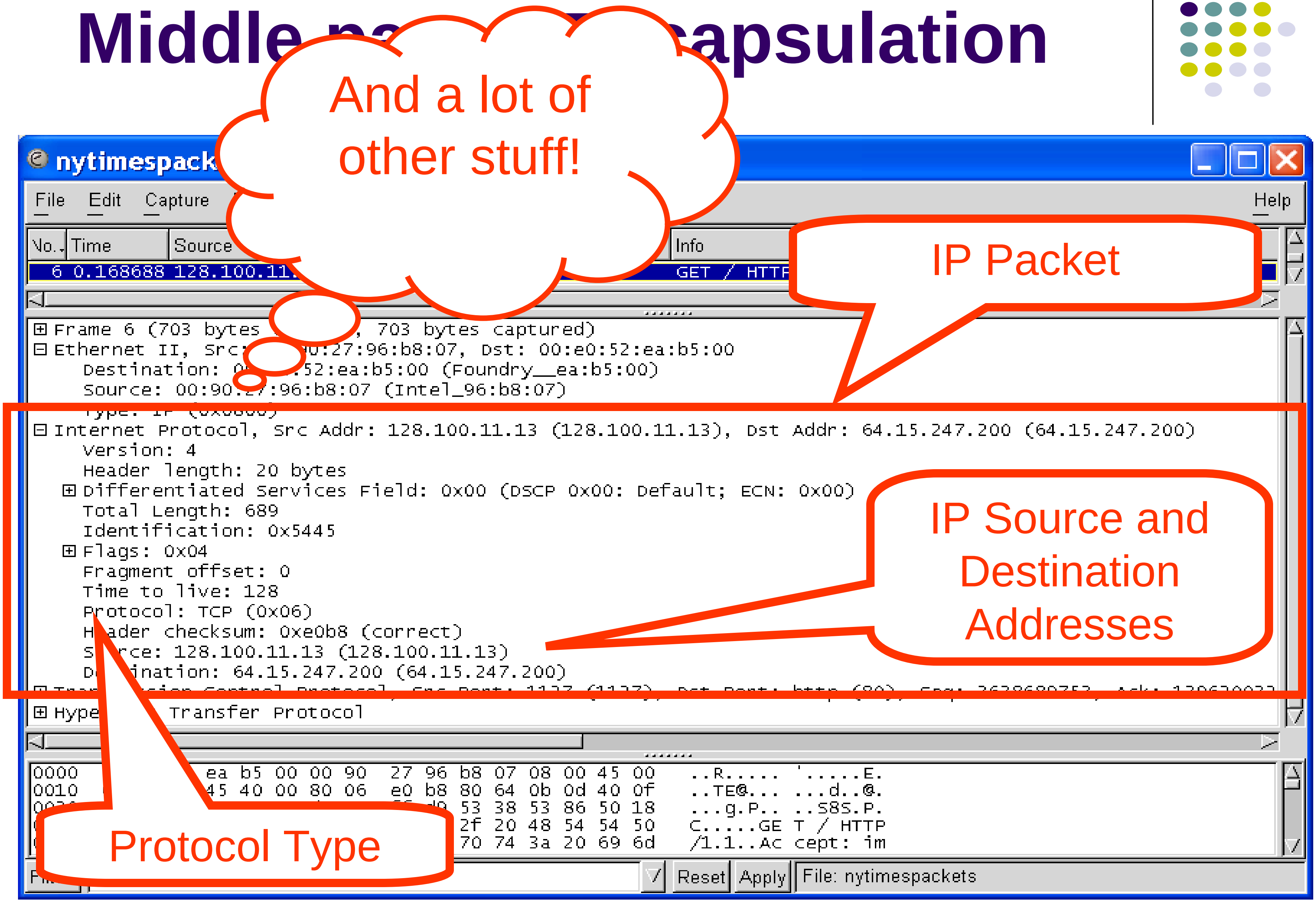
Top pane: frame sequence



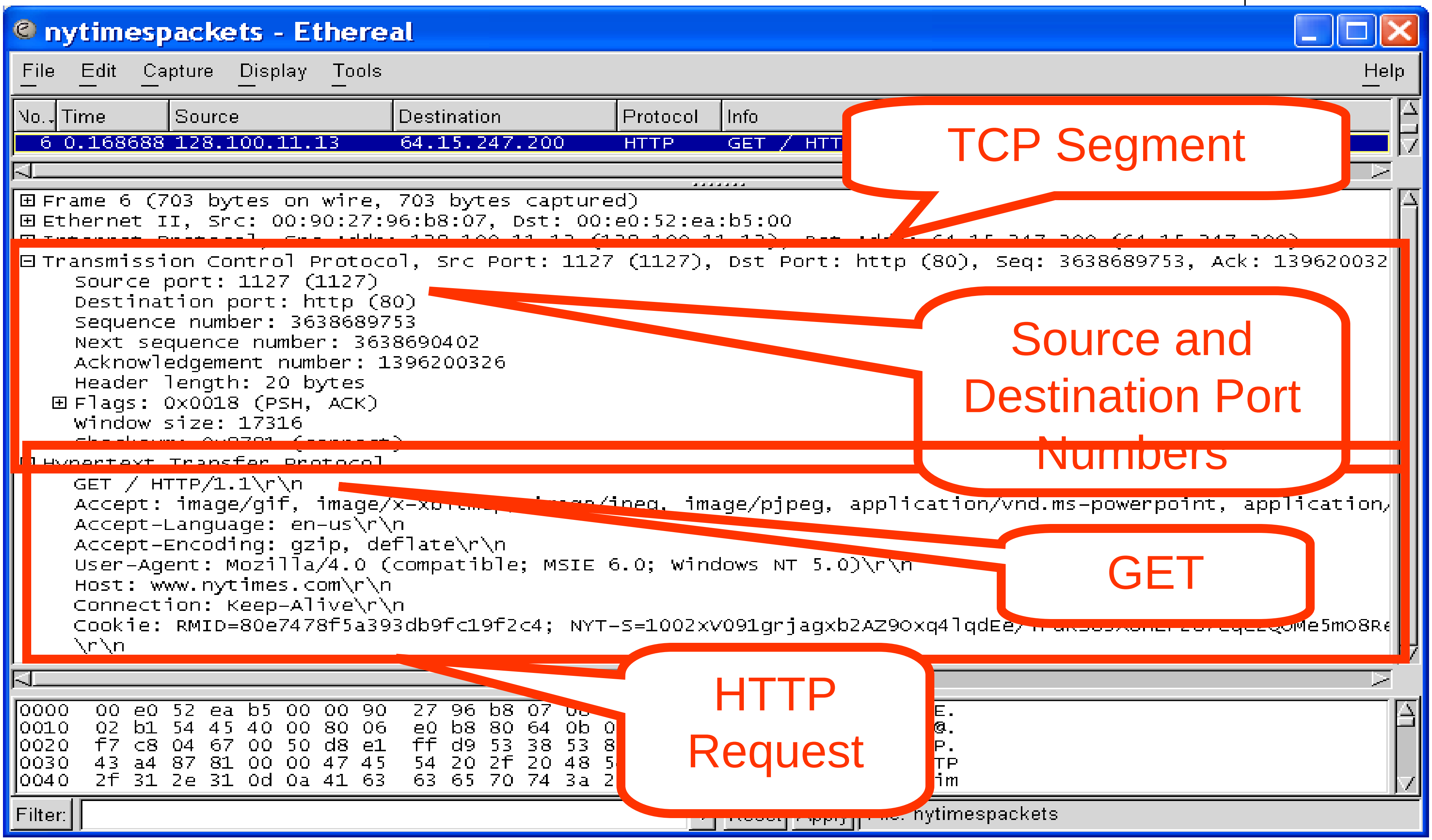
Middle pane: Encapsulation



Middle pa capsulation



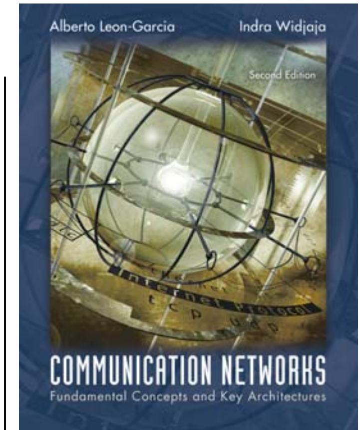
Middle pane: Encapsulation



Summary

- Encapsulation is key to layering
- IP provides for transfer of packets across diverse networks
- TCP and UDP provide universal communications services across the Internet
- Distributed applications that use TCP and UDP can operate over the entire Internet
- Internet names, IP addresses, port numbers, sockets, connections, physical addresses

Chapter 2
Applications and Layered Architectures

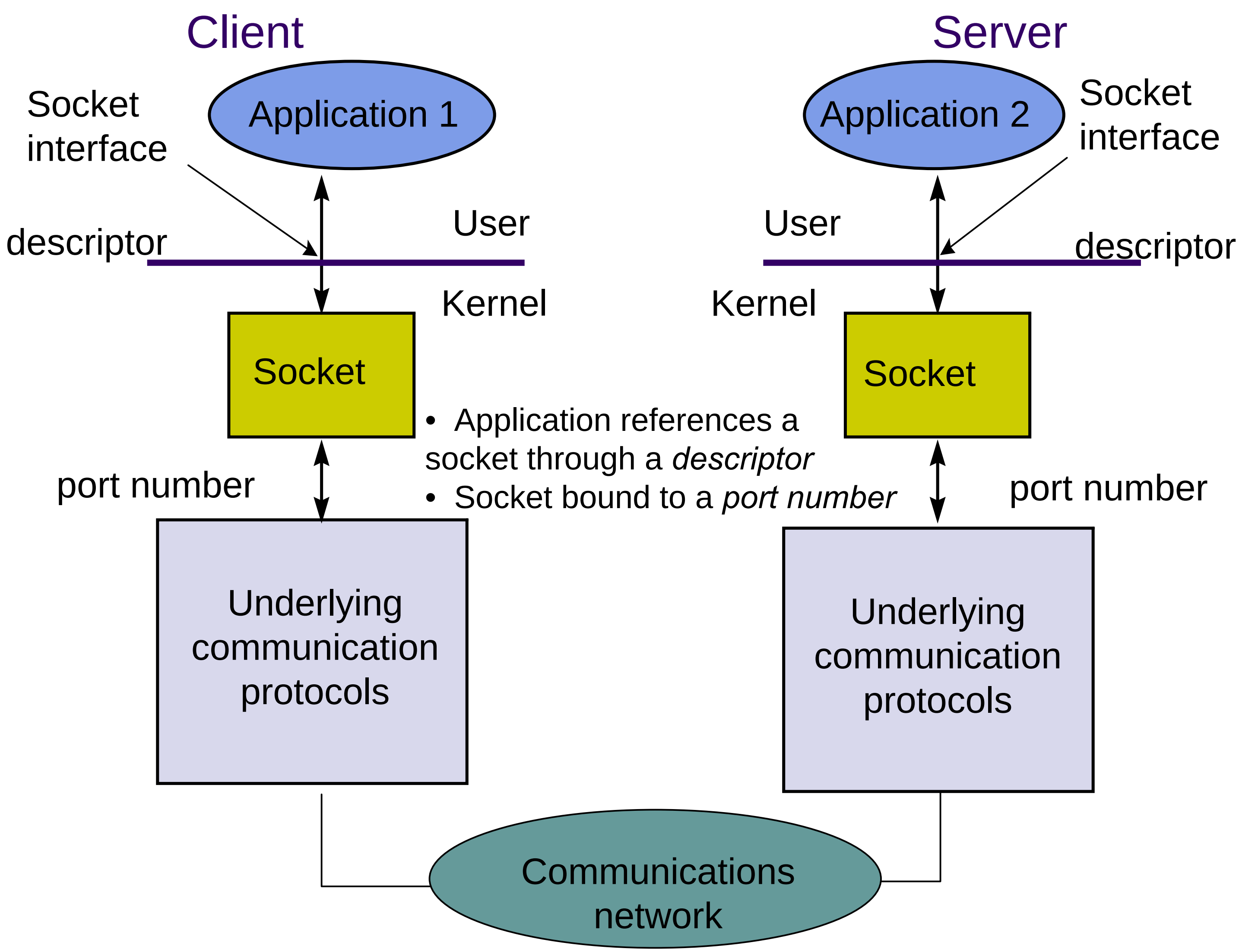


Sockets

Socket API

- API (Application Programming Interface)
  - Provides a standard set of functions that can be called by applications
- Berkeley UNIX Sockets API
  - Abstraction for applications to send & receive data
  - Applications create sockets that “plug into” network
  - Applications write/read to/from sockets
  - Implemented in the kernel
  - Facilitates development of network applications
  - Hides details of underlying protocols & mechanisms
- Also in Windows, Linux, and other OS’s

Communications through Socket Interface



Stream mode of service

Connection-oriented

- First, setup connection between two peer application processes
- Then, reliable bidirectional in-sequence transfer of *byte stream* (boundaries not preserved in transfer)
- Multiple write/read between peer processes
- Finally, connection release
- Uses TCP

- Connectionless
- Immediate transfer of one block of information (boundaries preserved)
- No setup overhead & delay
- Destination address with each block
- Send/receive to/from multiple peer processes
- Best-effort service only
  - Possible out-of-order
  - Possible loss
- Uses UDP

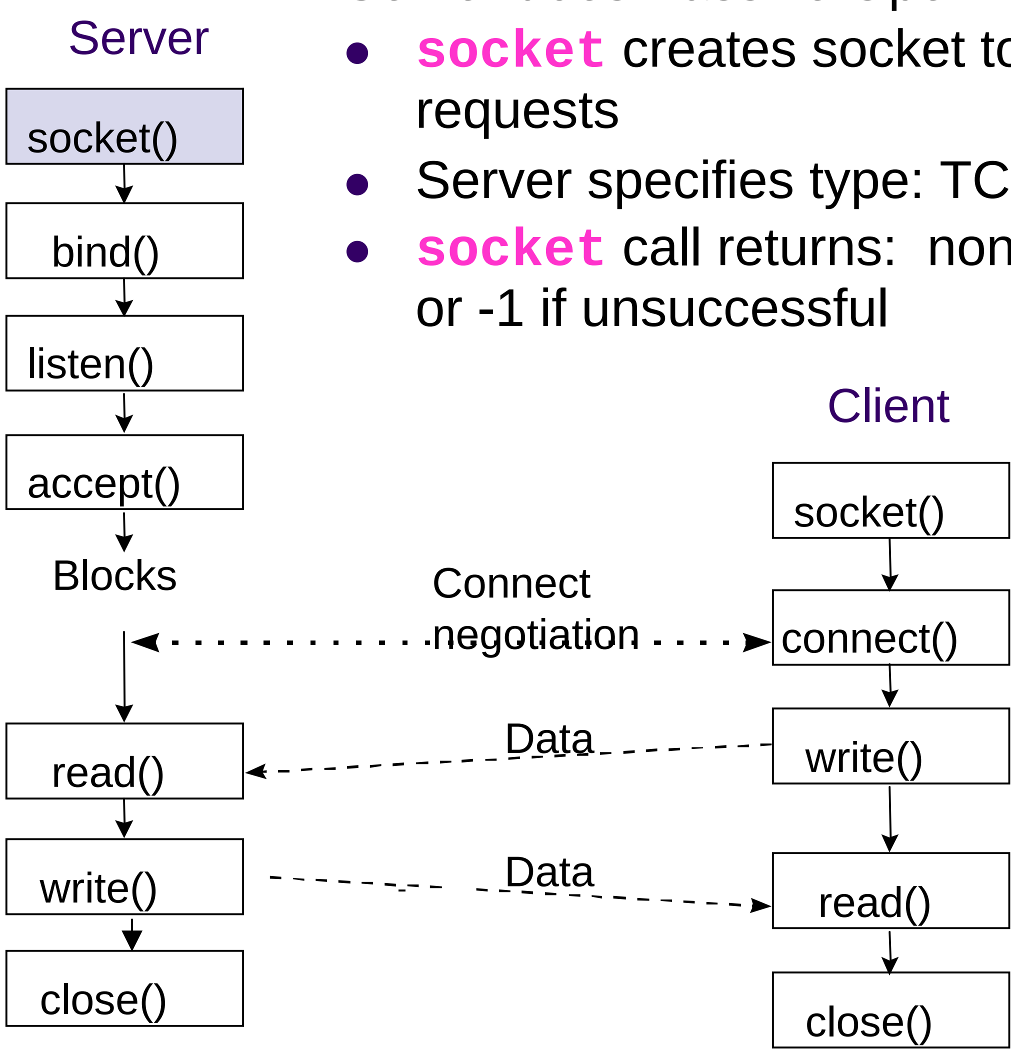
Client & Server Differences

- Server
  - Specifies well-known port # when creating socket
  - May have multiple IP addresses (net interfaces)
  - Waits passively for client requests
- Client
  - Assigned ephemeral port #
  - Initiates communications with server
  - Needs to know server's IP address & port #
    - DNS for URL & server well-known port #
  - Server learns client's address & port #

Socket Calls for Connection-Oriented Mode

Server does Passive Open

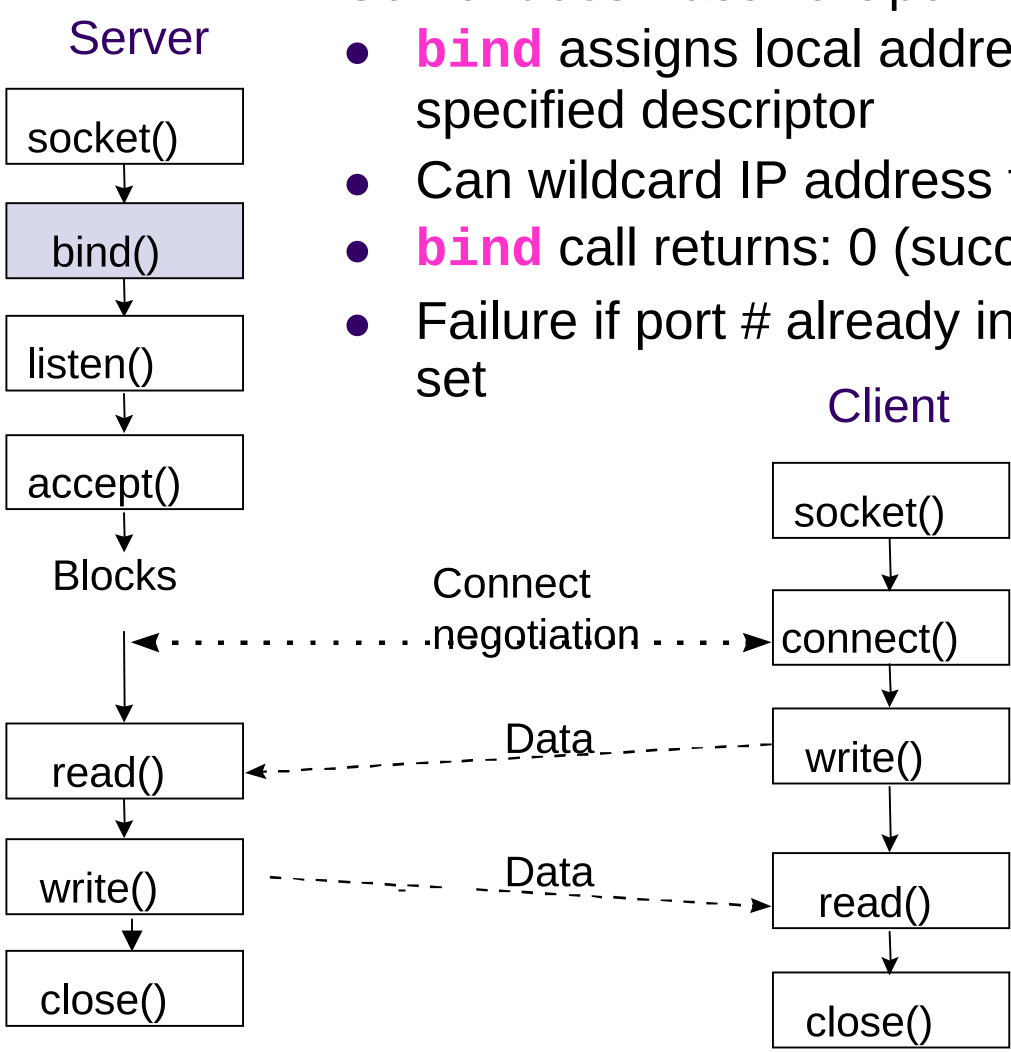
- **socket** creates socket to *listen* for connection requests
- Server specifies type: TCP (stream)
- **socket** call returns: non-negative integer *descriptor*; or -1 if unsuccessful



Socket Calls for Connection-Oriented Mode

Server does Passive Open

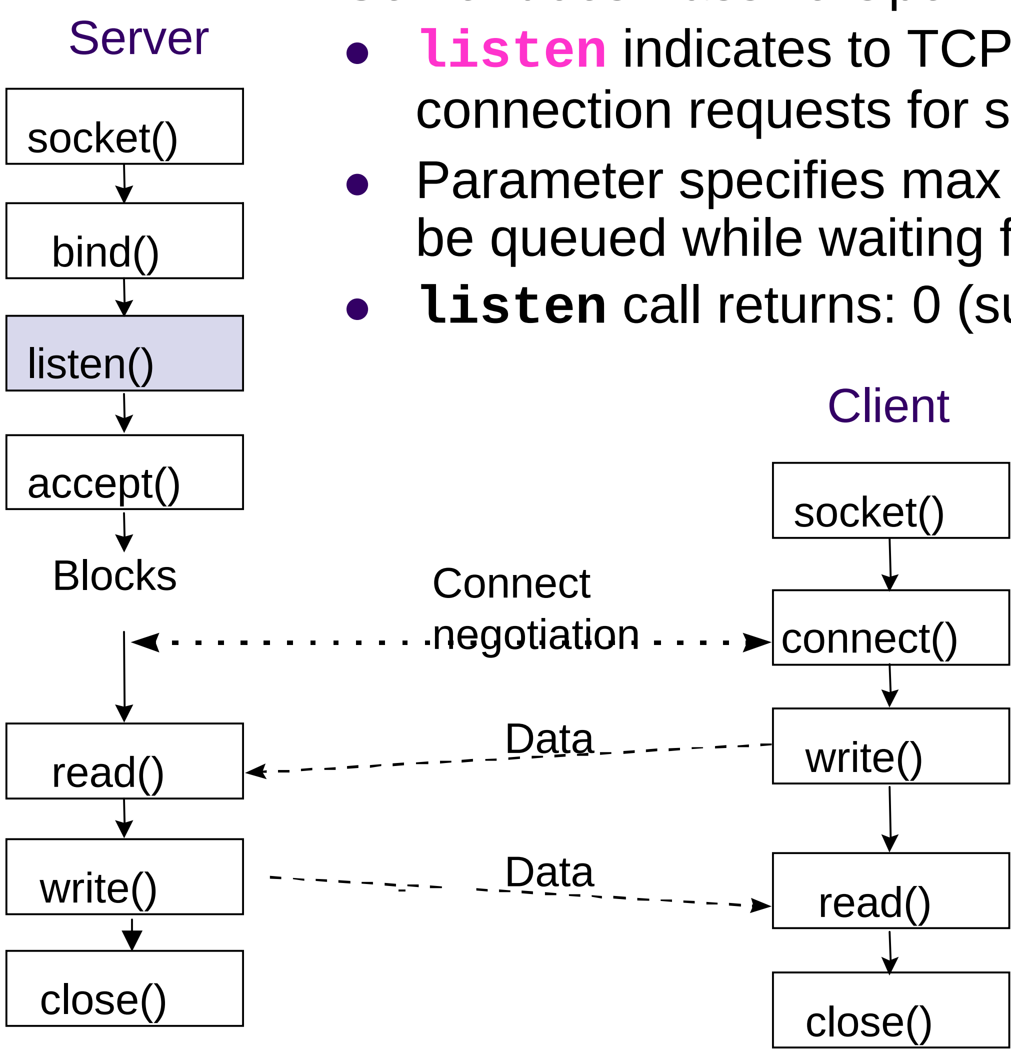
- **bind** assigns local address & port # to socket with specified descriptor
- Can wildcard IP address for multiple net interfaces
- **bind** call returns: 0 (success); or -1 (failure)
- Failure if port # already in use or if reuse option not set



Socket Calls for Connection-Oriented Mode

Server does Passive Open

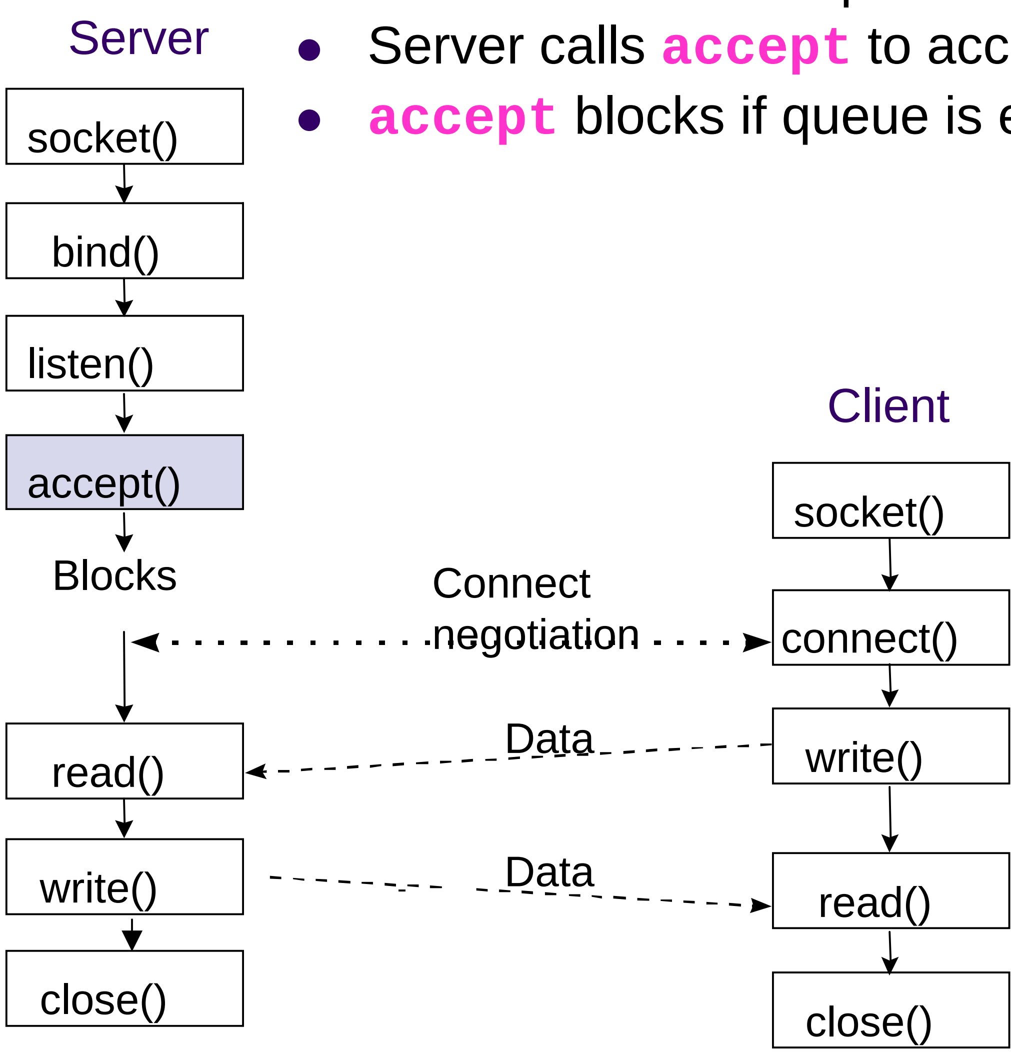
- `listen` indicates to TCP readiness to receive connection requests for socket with given descriptor
- Parameter specifies max number of requests that may be queued while waiting for server to accept them
- **`listen`** call returns: 0 (success); or -1 (failure)



Socket Calls for Connection-Oriented Mode

Server does Passive Open

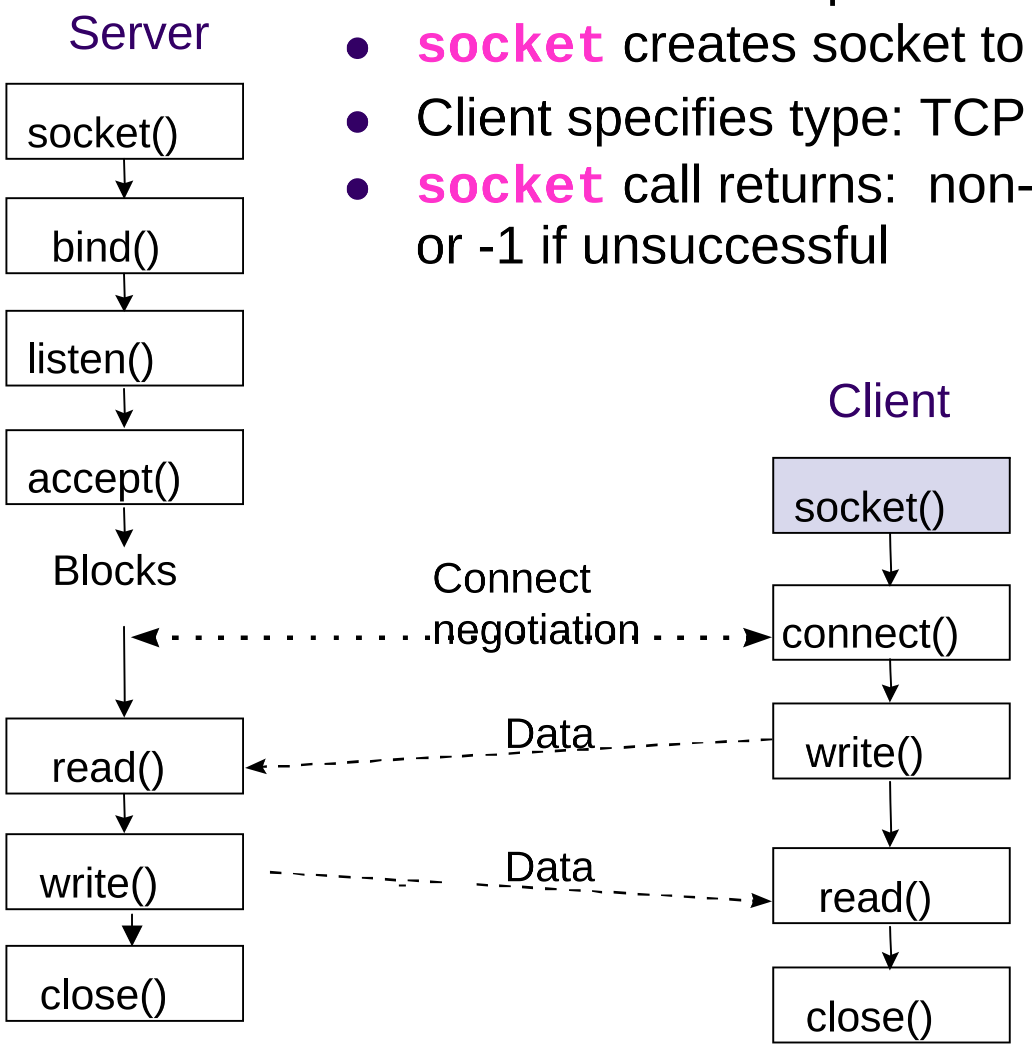
- Server calls **accept** to accept incoming requests
- **accept** blocks if queue is empty



Socket Calls for Connection-Oriented Mode

Client does Active Open

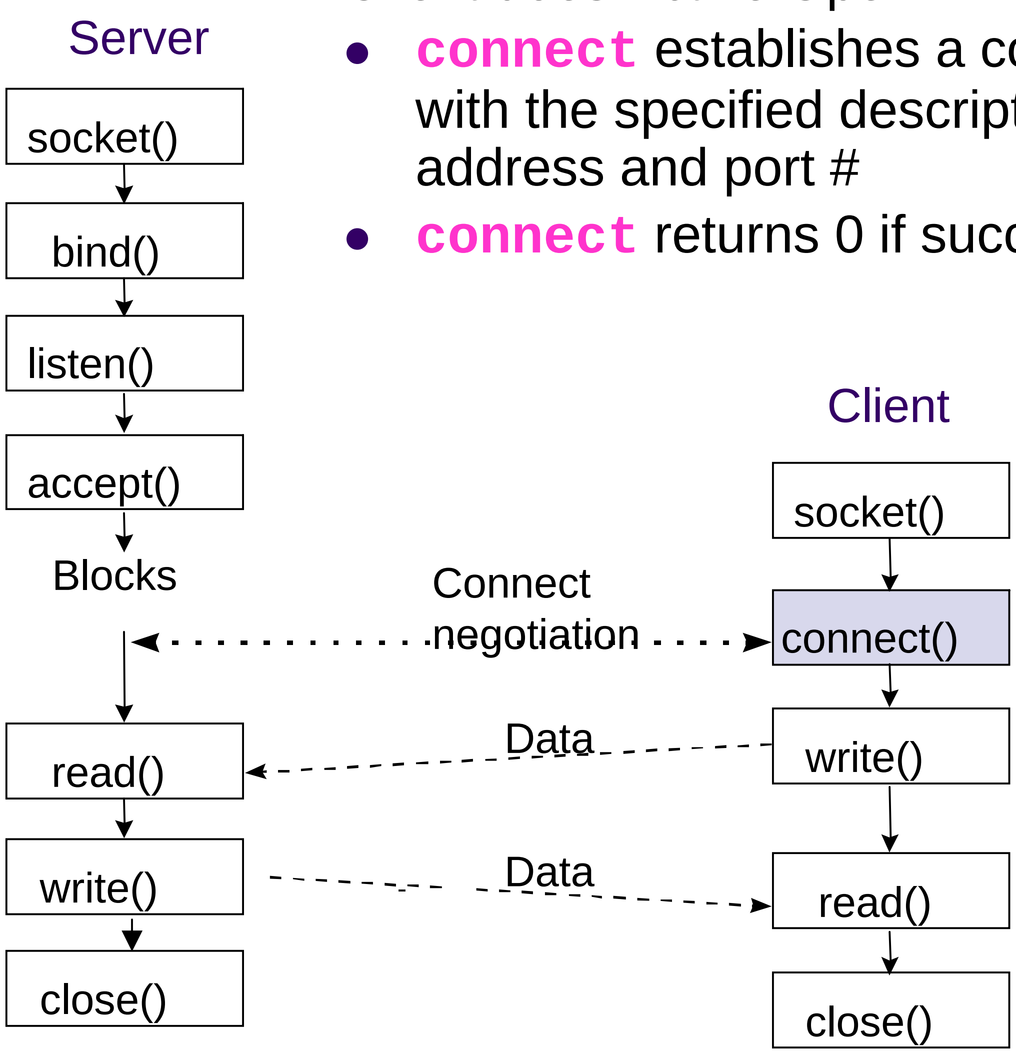
- **socket** creates socket to connect to server
- Client specifies type: TCP (stream)
- **socket** call returns: non-negative integer *descriptor*; or -1 if unsuccessful



Socket Calls for Connection-Oriented Mode

Client does Active Open

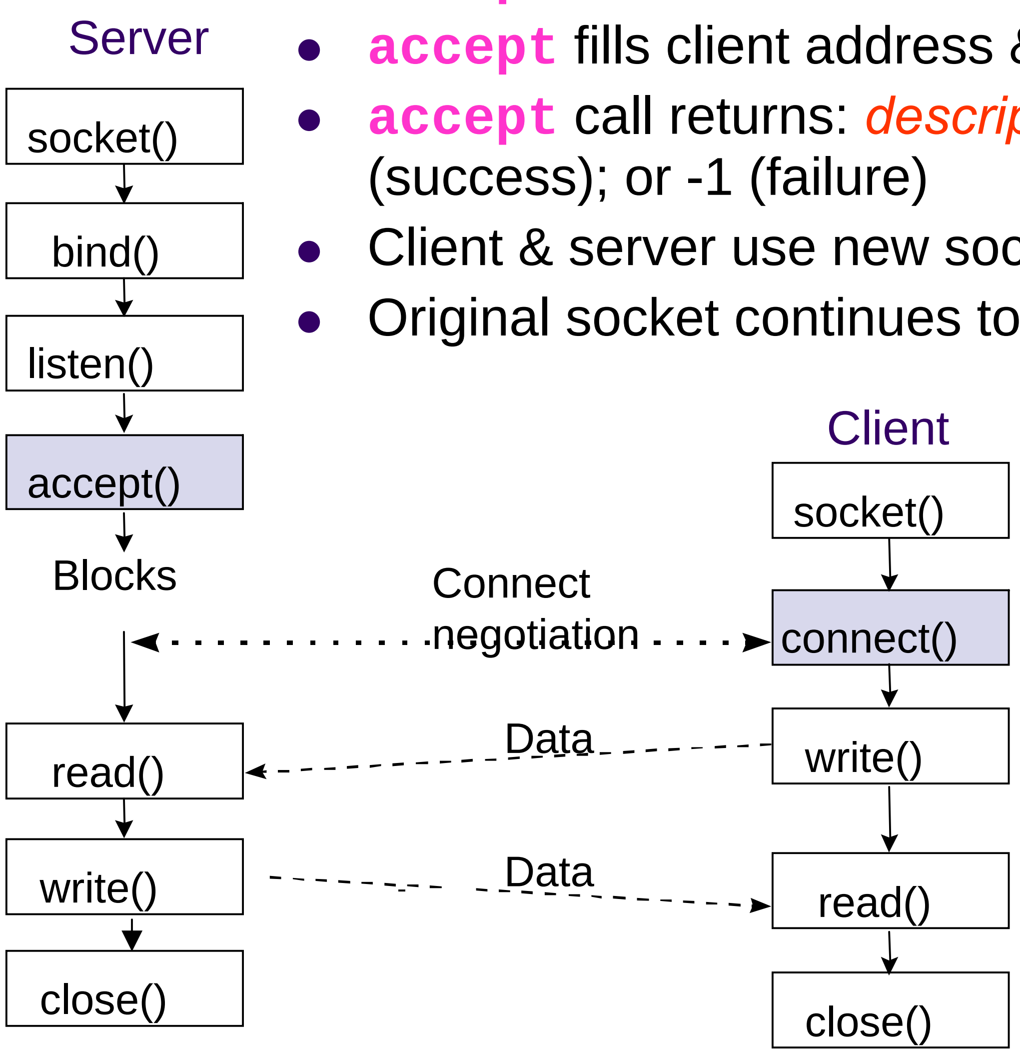
- **connect** establishes a connection on the local socket with the specified descriptor to the specified remote address and port #
- **connect** returns 0 if successful; -1 if unsuccessful



Note: **connect** initiates TCP three-way handshake

Socket Calls for Connection-Oriented Mode

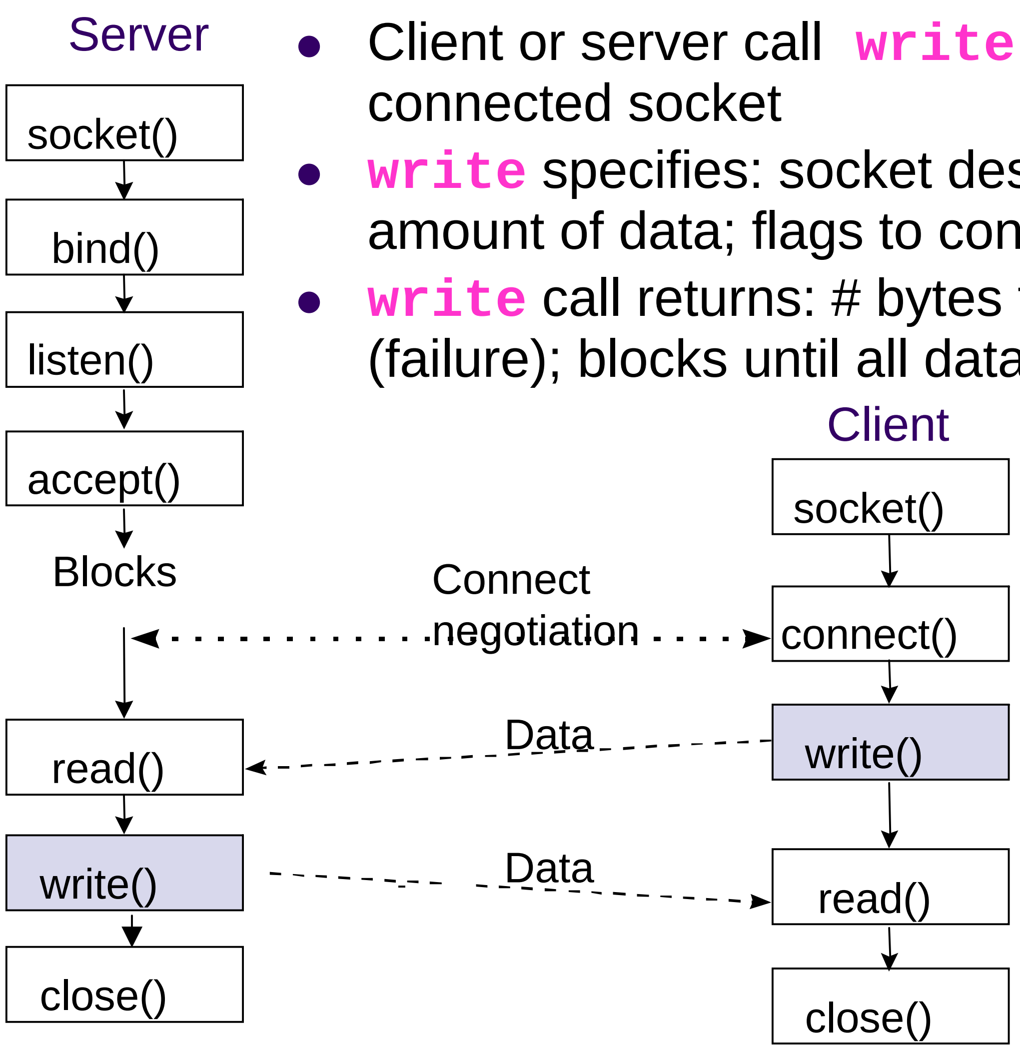
- `accept` wakes with incoming connection request
- `accept` fills client address & port # into address structure
- `accept` call returns: *descriptor of **new** connection socket* (success); or -1 (failure)
- Client & server use new socket for data transfer
- Original socket continues to listen for new requests



Socket Calls for Connection-Oriented Mode

Data Transfer

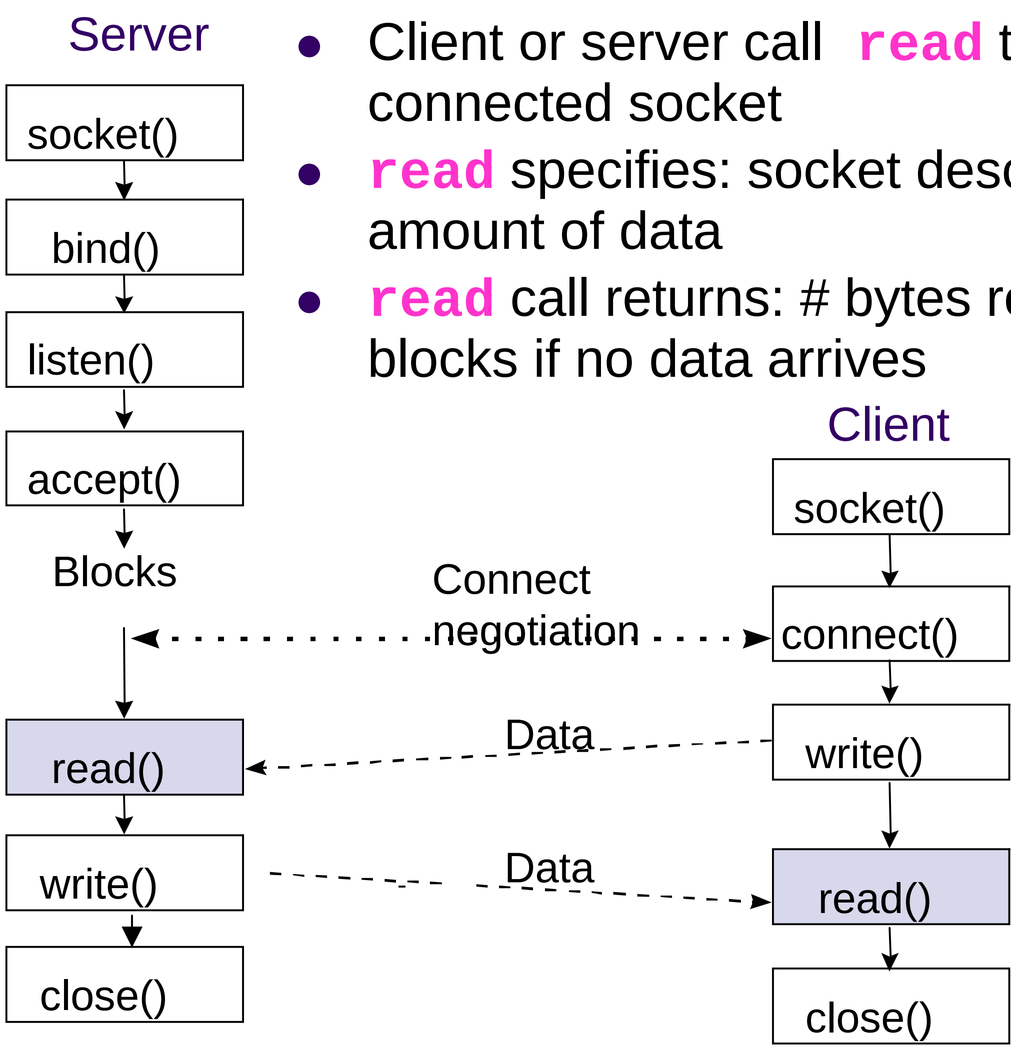
- Client or server call `write` to transmit data into a connected socket
- `write` specifies: socket descriptor; pointer to a buffer; amount of data; flags to control transmission behavior
- `write` call returns: # bytes transferred (success); or -1 (failure); blocks until all data transferred



Socket Calls for Connection-Oriented Mode

Data Transfer

- Client or server call **read** to receive data from a connected socket
- **read** specifies: socket descriptor; pointer to a buffer; amount of data
- **read** call returns: # bytes read (success); or -1 (failure); blocks if no data arrives

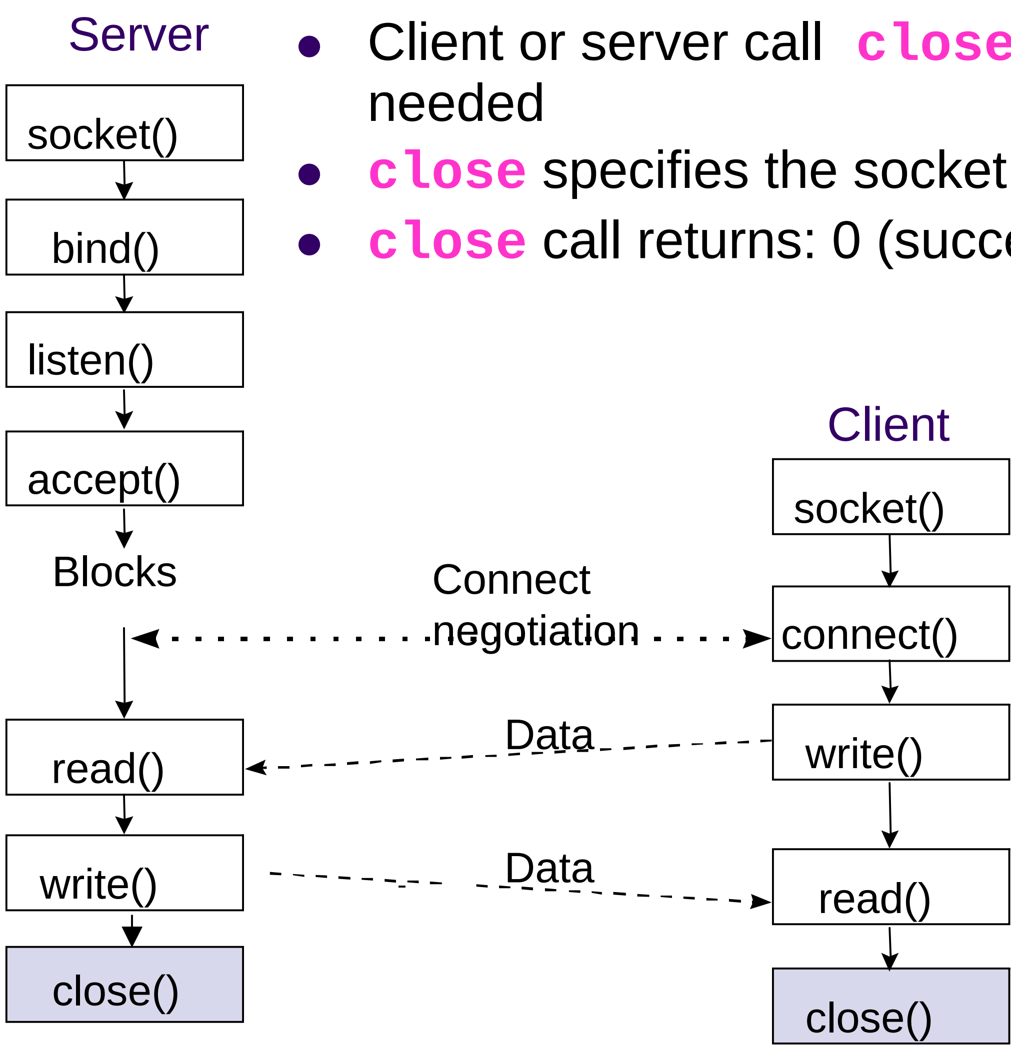


Note: **write** and **read** can be called multiple times to transfer byte streams in both directions

Socket Calls for Connection-Oriented Mode

Connection Termination

- Client or server call `close` when socket is no longer needed
- `close` specifies the socket descriptor
- `close` call returns: 0 (success); or -1 (failure)



Note: `close` initiates TCP graceful close sequence

Example: TCP Echo Server

```
/* A simple echo server using TCP */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVER_TCP_PORT                      3000
#define BUFLEN                               256

int main(int argc, char **argv)
{
        int     n, bytes_to_read;
        int     sd, new_sd, client_len, port;
        struct  sockaddr_in    server, client;
        char    *bp, buf[BUFLEN];

        switch(argc) {
        case 1:
                port = SERVER_TCP_PORT;
                break;
        case 2:
                port = atoi(argv[1]);
                break;
        default:
                fprintf(stderr, "Usage: %s [port]\n", argv[0]);
                exit(1);
        }

        /* Create a stream socket */
        if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
                fprintf(stderr, "Can't create a socket\n");
                exit(1);
        }

        /* Bind an address to the socket */
        bzero((char *)&server, sizeof(struct sockaddr_in));
        server.sin_family = AF_INET;
        server.sin_port = htons(port);
        server.sin_addr.s_addr = htonl(INADDR_ANY);
        if (bind(sd, (struct sockaddr *)&server,
        sizeof(server)) == -1) {
                fprintf(stderr, "Can't bind name to socket\n");
                exit(1);
        }

        /* queue up to 5 connect requests */
        listen(sd, 5);

        while (1) {
                client_len = sizeof(client);
                if ((new_sd = accept(sd, (struct sockaddr *)&client,
                &client_len)) == -1) {
                      fprintf(stderr, "Can't accept client\n");
                      exit(1);
                }

                bp = buf;
                bytes_to_read = BUFLEN;
                while ((n = read(new_sd, bp, bytes_to_read)) > 0) {
                      bp += n;
                      bytes_to_read -= n;
                }
                printf("Rec'd: %s\n", buf);

                write(new_sd, buf, BUFLEN);
                printf("Sent: %s\n", buf);
                close(new_sd);
        }
        close(sd);
        return(0);
}
```

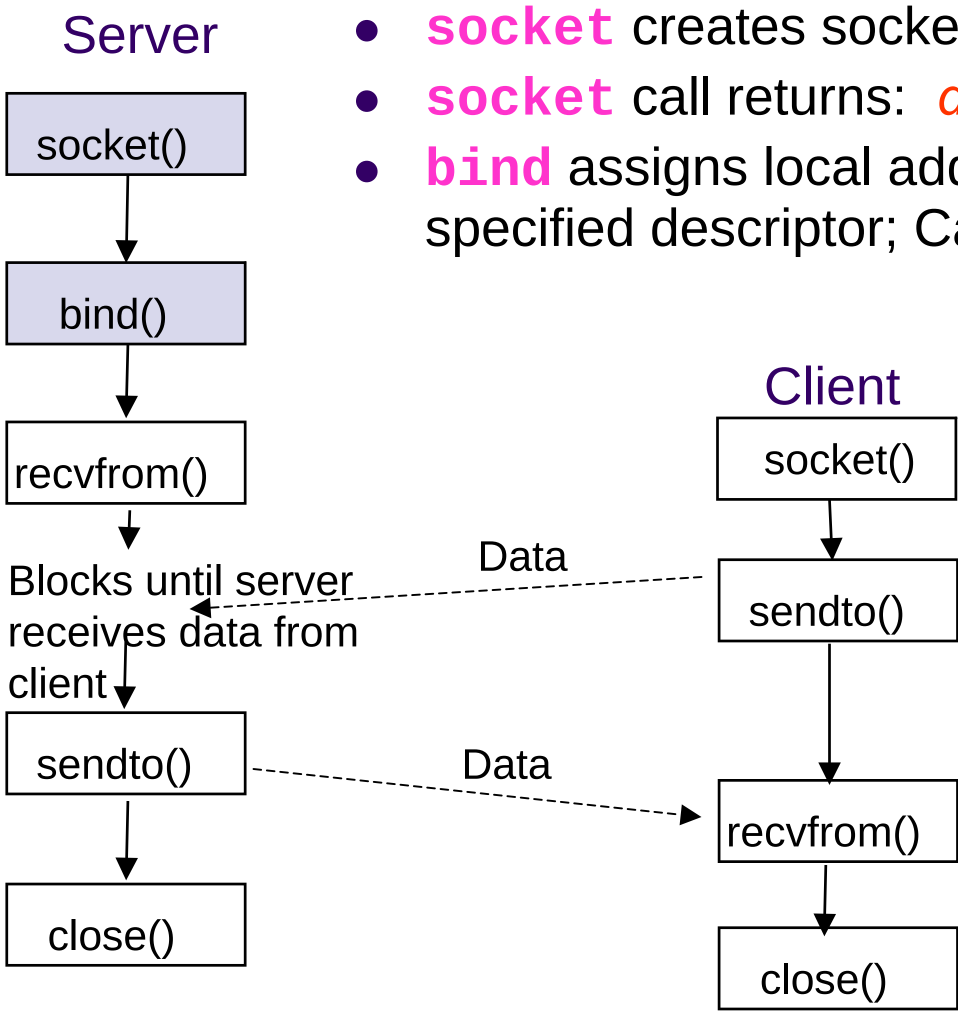
Example: TCP Echo Client

```
/* A simple TCP client */
#include <stdio.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVER_TCP_PORT                3000
#define BUFLEN                         256

int main(int argc, char **argv)
{
        int     n, bytes_to_read;
        int     sd, port;
        struct  hostent         *hp;
        struct  sockaddr_in     server;
        char    *host, *bp, rbuf[BUFLEN], sbuf[BUFLEN];

        switch(argc) {
        case 2:
                host = argv[1];
                port = SERVER_TCP_PORT;
                break;
        case 3:
                host = argv[1];
                port = atoi(argv[2]);
                break;
        default:
                fprintf(stderr, "Usage: %s host [port]\n", argv[0]);
                exit(1);
        }

        /* Create a stream socket */
        if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
                fprintf(stderr, "Can't create a socket\n");
                exit(1);
        }

        bzero((char *)&server, sizeof(struct sockaddr_in));
        server.sin_family = AF_INET;
        server.sin_port = htons(port);
        if ((hp = gethostbyname(host)) == NULL) {
                fprintf(stderr, "Can't get server's address\n");
                exit(1);
        }
        bcopy(hp->h_addr, (char *)&server.sin_addr, hp->h_length);

        /* Connecting to the server */
        if (connect(sd, (struct sockaddr *)&server,
        sizeof(server)) == -1) {
                fprintf(stderr, "Can't connect\n");
                exit(1);
        }
        printf("Connected: server's address is %s\n", hp->h_name);

        printf("Transmit:\n");
        gets(sbuf);
                write(sd, sbuf, BUFLEN);

        printf("Receive:\n");
        bp = rbuf;
        bytes_to_read = BUFLEN;
        while ((n = read(sd, bp, bytes_to_read)) > 0) {
                bp += n;
                bytes_to_read -= n;
        }
        printf("%s\n", rbuf);

        close(sd);
        return(0);
}
```

Socket Calls for Connection-Less Mode

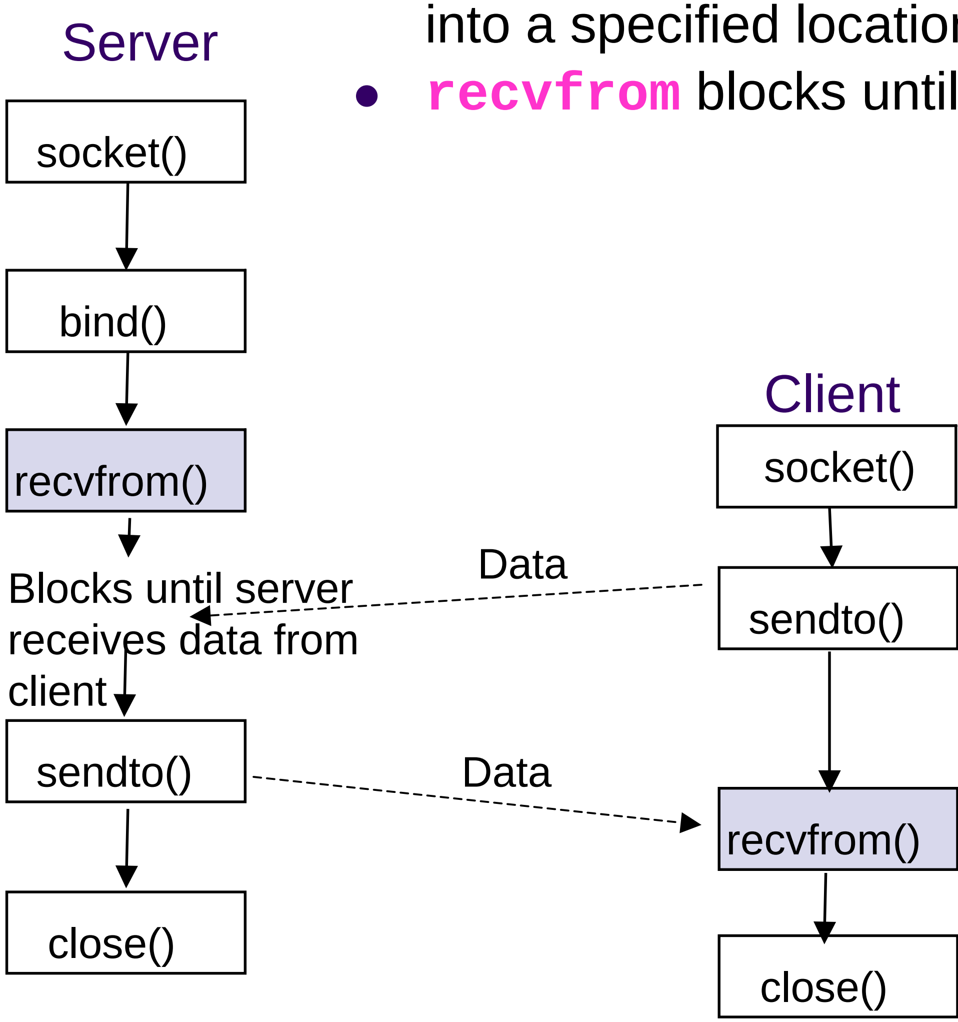
Server started

- **socket** creates socket of type UDP (datagram)
- **socket** call returns: *descriptor*; or -1 if unsuccessful
- **bind** assigns local address & port # to socket with specified descriptor; Can wildcard IP address



Socket Calls for Connection-Less Mode

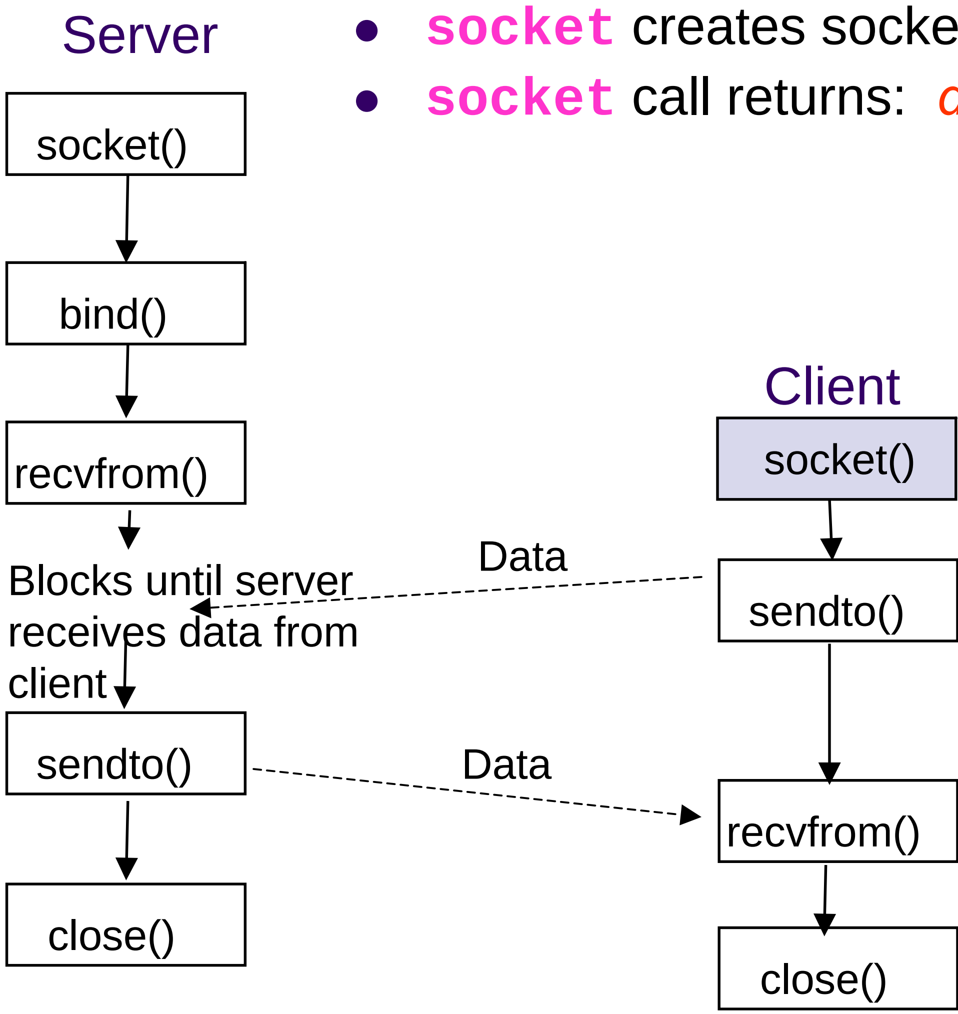
- `recvfrom` copies bytes received in specified socket into a specified location
- `recvfrom` blocks until data arrives



Socket Calls for Connection-Less Mode

Client started

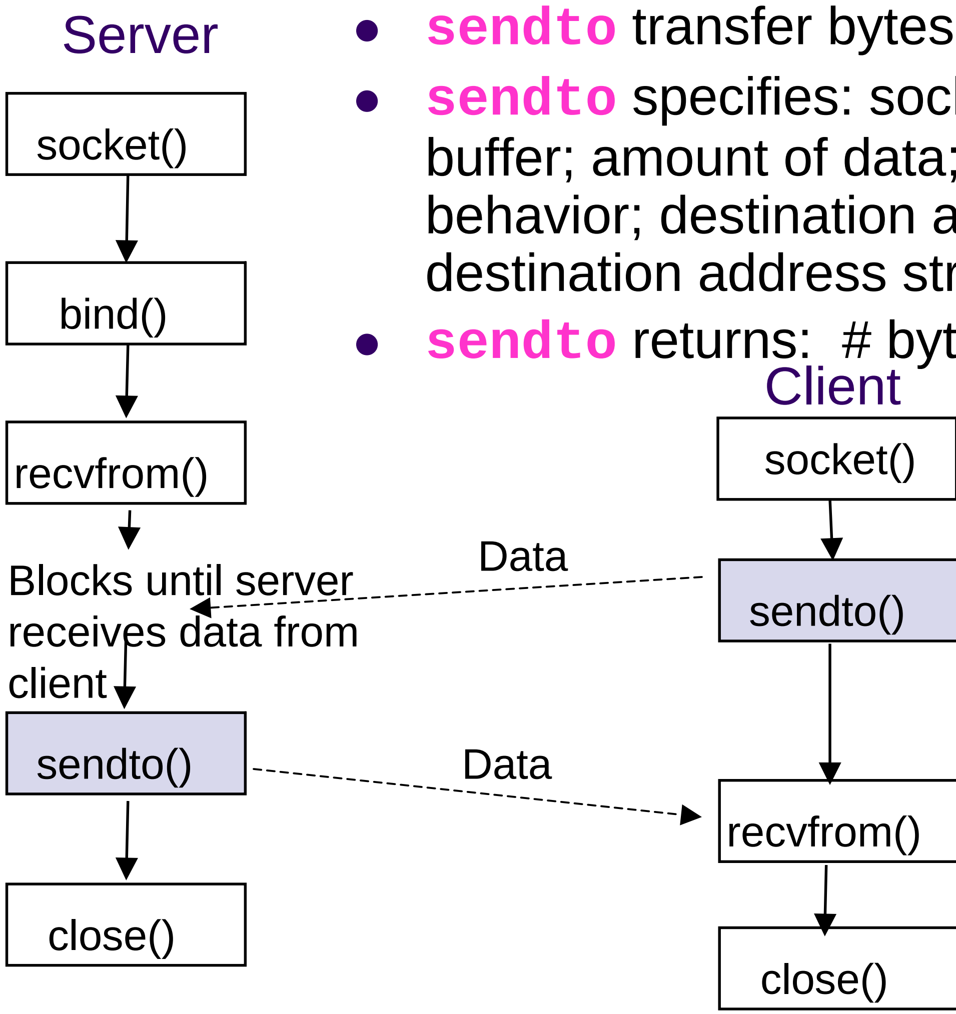
- `socket` creates socket of type UDP (datagram)
- `socket` call returns: *descriptor*; or -1 if unsuccessful



Socket Calls for Connection-Less Mode

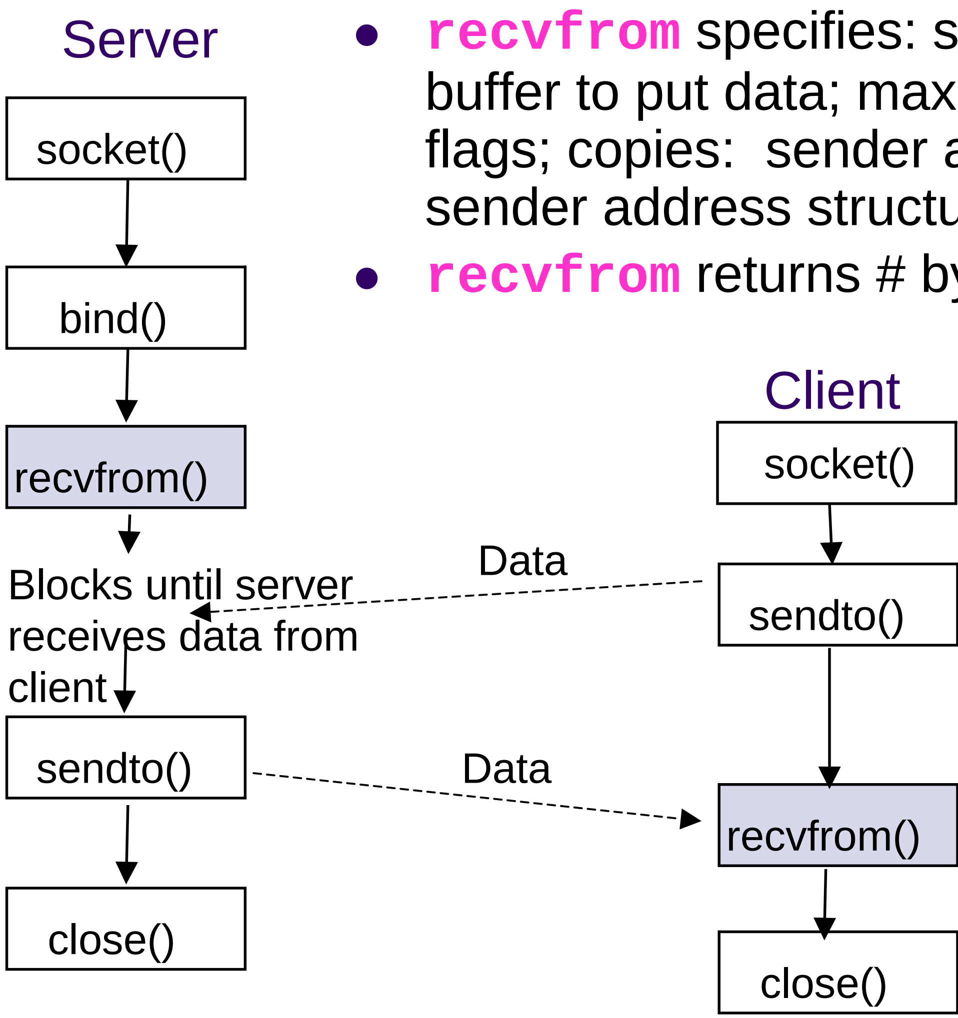
Client started

- **sendto** transfer bytes in buffer to specified socket
- **sendto** specifies: socket descriptor; pointer to a buffer; amount of data; flags to control transmission behavior; destination address & port #; length of destination address structure
- **sendto** returns: # bytes sent; or -1 if unsuccessful



Socket Calls for Connection-Less Mode

- **recvfrom** wakes when data arrives
- **recvfrom** specifies: socket descriptor; pointer to a buffer to put data; max # bytes to put in buffer; control flags; copies: sender address & port #; length of sender address structure
- **recvfrom** returns # bytes received or -1 (failure)

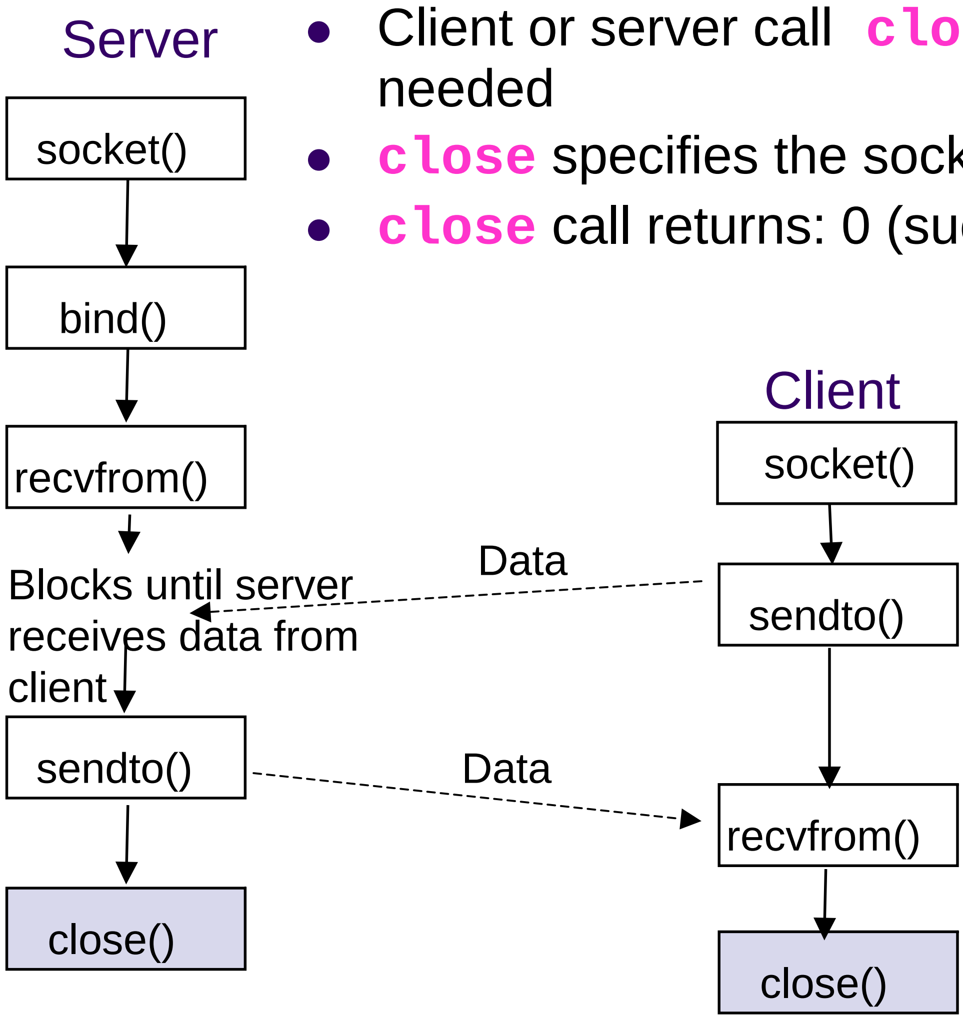


Note: **receivefrom** returns data from at most one **send**, i.e. from one datagram

Socket Calls for Connection-Less Mode

Socket Close

- Client or server call `close` when socket is no longer needed
- `close` specifies the socket descriptor
- `close` call returns: 0 (success); or -1 (failure)



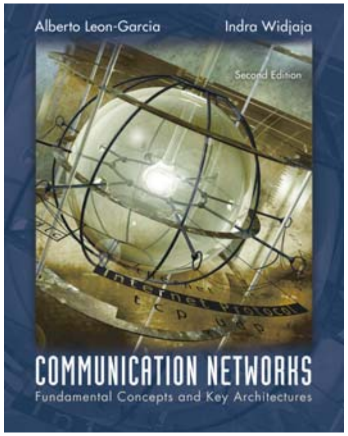
Example: UDP Echo Server

```
/* Echo server using UDP */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVER_UDP_PORT                         5000
#define MAXLEN                                  4096

int main(int argc, char **argv)
{
        int     sd, client_len, port, n;
        char    buf[MAXLEN];
        struct  sockaddr_in     server, client;

        switch(argc) {
        case 1:
                port = SERVER_UDP_PORT;
                break;
        case 2:
                port = atoi(argv[1]);
                break;
        default:
                fprintf(stderr, "Usage: %s [port]\n", argv[0]);
                exit(1);
        }

        /* Create a datagram socket */
        if ((sd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
                fprintf(stderr, "Can't create a socket\n");
                exit(1);
        }

        /* Bind an address to the socket */
        bzero((char *)&server, sizeof(server));
        server.sin_family = AF_INET;
        server.sin_port = htons(port);
        server.sin_addr.s_addr = htonl(INADDR_ANY);
        if (bind(sd, (struct sockaddr *)&server,
        sizeof(server)) == -1) {
                fprintf(stderr, "Can't bind name to socket\n");
                exit(1);
        }

        while (1) {
                client_len = sizeof(client);
                if ((n = recvfrom(sd, buf, MAXLEN, 0,
                (struct sockaddr *)&client, &client_len)) < 0) {
                      fprintf(stderr, "Can't receive datagram\n");
                      exit(1);
                }

                if (sendto(sd, buf, n, 0,
                (struct sockaddr *)&client, client_len) != n) {
                      fprintf(stderr, "Can't send datagram\n");
                      exit(1);
                }
        }
        close(sd);
        return(0);
}
```

Example: UDP Echo Client

```
#include <stdio.h>
#include <string.h>
#include <sys/time.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define SERVER_UDP_PORT         5000
#define MAXLEN                  4096
#define DEFLEN                  64

long delay(struct timeval t1, struct timeval t2)
{
        long d;
        d = (t2.tv_sec - t1.tv_sec) * 1000;
        d += ((t2.tv_usec - t1.tv_usec + 500) / 1000);
        return(d);
}
int main(int argc, char **argv)
{
        int     data_size = DEFLEN, port = SERVER_UDP_PORT;
        int     i, j, sd, server_len;
        char    *pname, *host, rbuf[MAXLEN], sbuf[MAXLEN];
        struct  hostent         *hp;
        struct  sockaddr_in     server;
        struct  timeval         start, end;
        unsigned long address;

        pname = argv[0];
        argc--;
        argv++;
        if (argc > 0 && (strcmp(*argv, "-s") == 0)) {
                if (--argc > 0 && (data_size = atoi(*++argv))) {
                        argc--;
                        argv++;
                }
                else {
                        fprintf(stderr,
                        "Usage: %s [-s data_size] host [port]\n", pname);
                        exit(1);
                }
        }
        if (argc > 0) {
                host = *argv;
                if (--argc > 0)
                        port = atoi(*++argv);
        }
        else {
                fprintf(stderr,
                "Usage: %s [-s data_size] host [port]\n", pname);
                exit(1);
        }

        if ((sd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
                fprintf(stderr, "Can't create a socket\n");
                exit(1);
        }
        bzero((char *)&server, sizeof(server));
        server.sin_family = AF_INET;
        server.sin_port = htons(port);
        if ((hp = gethostbyname(host)) == NULL) {
                fprintf(stderr, "Can't get server's IP address\n");
                exit(1);
        }
        bcopy(hp->h_addr, (char *) &server.sin_addr, hp->h_length);

        if (data_size > MAXLEN) {
                fprintf(stderr, "Data is too big\n");
                exit(1);
        }
        for (i = 0; i < data_size; i++) {
                j = (i < 26) ? i : i % 26;
                sbuf[i] = 'a' + j;
        }
        gettimeofday(&start, NULL); /* start delay measurement */
        server_len = sizeof(server);
        if (sendto(sd, sbuf, data_size, 0, (struct sockaddr *)
                &server, server_len) == -1) {
                fprintf(stderr, "sendto error\n");
                exit(1);
        }
        if (recvfrom(sd, rbuf, MAXLEN, 0, (struct sockaddr *)
                &server, &server_len) < 0) {
                fprintf(stderr, "recvfrom error\n");
                exit(1);
        }
        gettimeofday(&end, NULL); /* end delay measurement */
        if (strncmp(sbuf, rbuf, data_size) != 0)
                printf("Data is corrupted\n");
        close(sd);
        return(0);
}
```

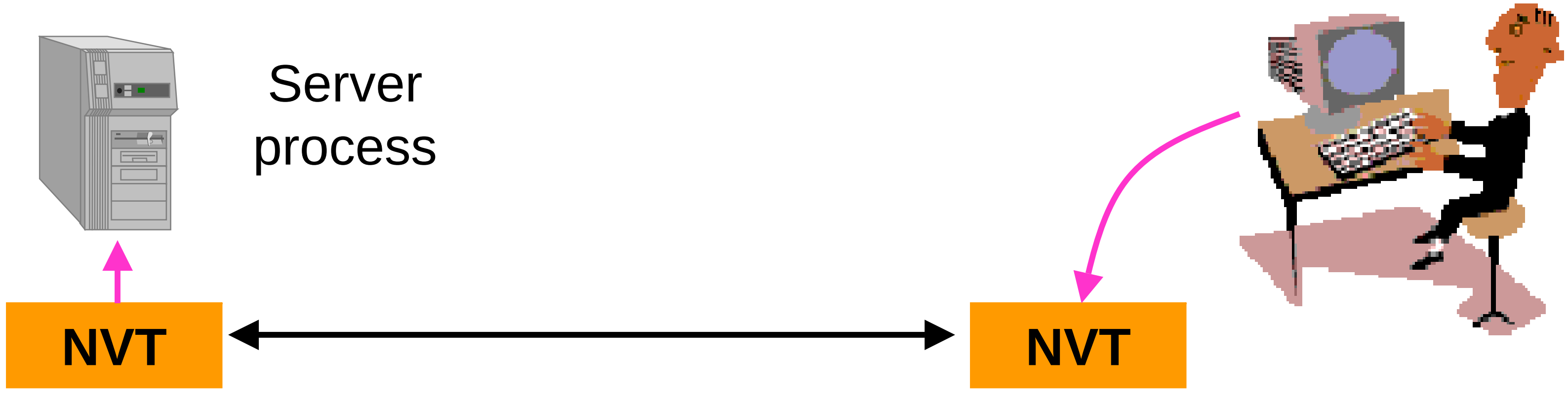
Chapter 2
Applications and Layered Architectures



Application Layer Protocols & IP Utilities

Telnet (RFC 854)

- Provides general bi-directional byte-oriented TCP-based communications facility (Network Virtual Terminal)
- Initiating machine treated as local to the remote host
- Used to connect to port # of other servers and to interact with them using command line



Network Virtual Terminal

- *Network Virtual Terminal*
- Lowest common denominator terminal
- Each machine maps characteristics to NVT
- Negotiate options for changes to the NVT
- Data input sent to server & echoed back
- Server control functions : interrupt, abort output, are-you-there, erase character, erase line
- Default requires login & password

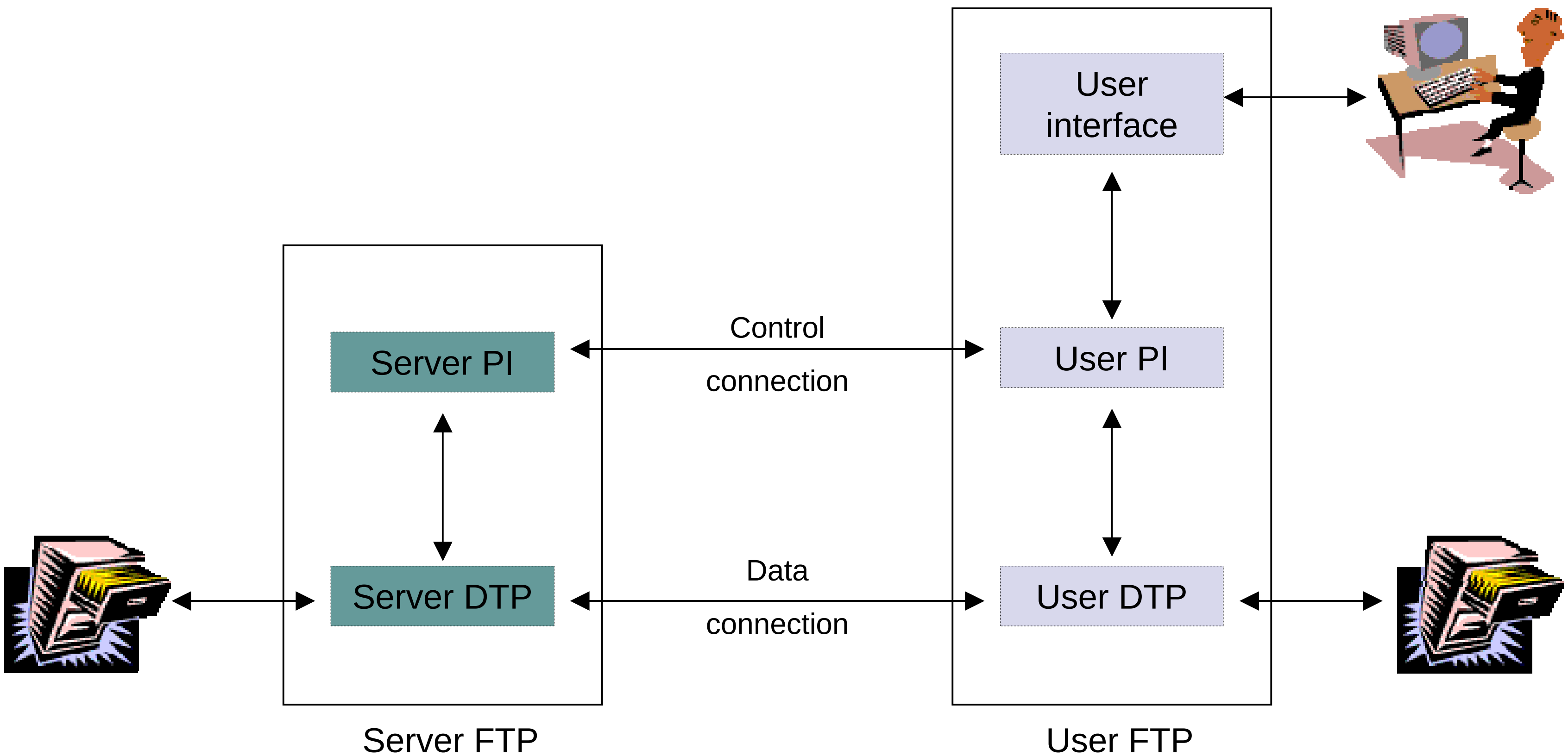
telnet

- A program that uses the Telnet protocol
- Establishes TCP socket
- Sends typed characters to server
- Prints whatever characters arrive
- Try it to retrieve a web page (HTTP) or to send an email (SMTP)

File Transfer Protocol (RFC 959)

- Provides for transfer of file from one machine to another machine
- Designed to hide variations in file storage
- FTP parameter commands specify file info
  - File Type: *ASCII*, EBCDIC, image, local.
  - Data Structure: *file*, record, or page
  - Transmission Mode: *stream*, block, compressed
- Other FTP commands
  - Access Control: USER, PASS, CWD, QUIT, …
  - Service: RETR, STOR, PWD, LIST, …

FTP File Transfer



PI = Protocol interface
DTP = Data transfer process

Two TCP Connections

Control connection

- Set up using Telnet protocol on well-known port 21
- FTP commands & replies between protocol interpreters
- PIs control the data transfer process
- User requests close of control connection; server performs the close

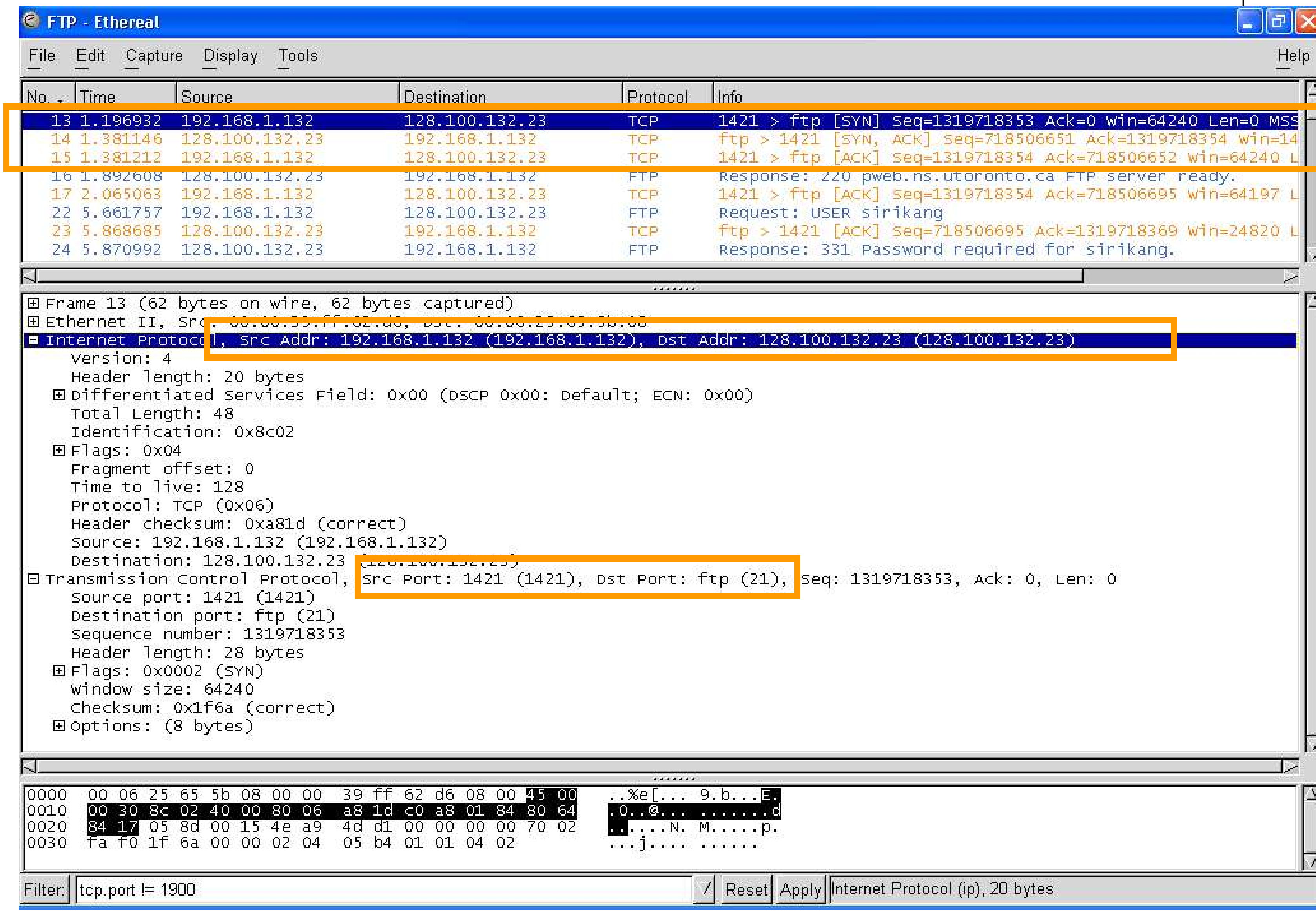
Data connection

- To perform file transfer, obtain lists of files, directories
- Each transfer requires new data connection
- Passive open by user PI with ephemeral port #
- Port # sent over control connection
- Active open by server using port 20

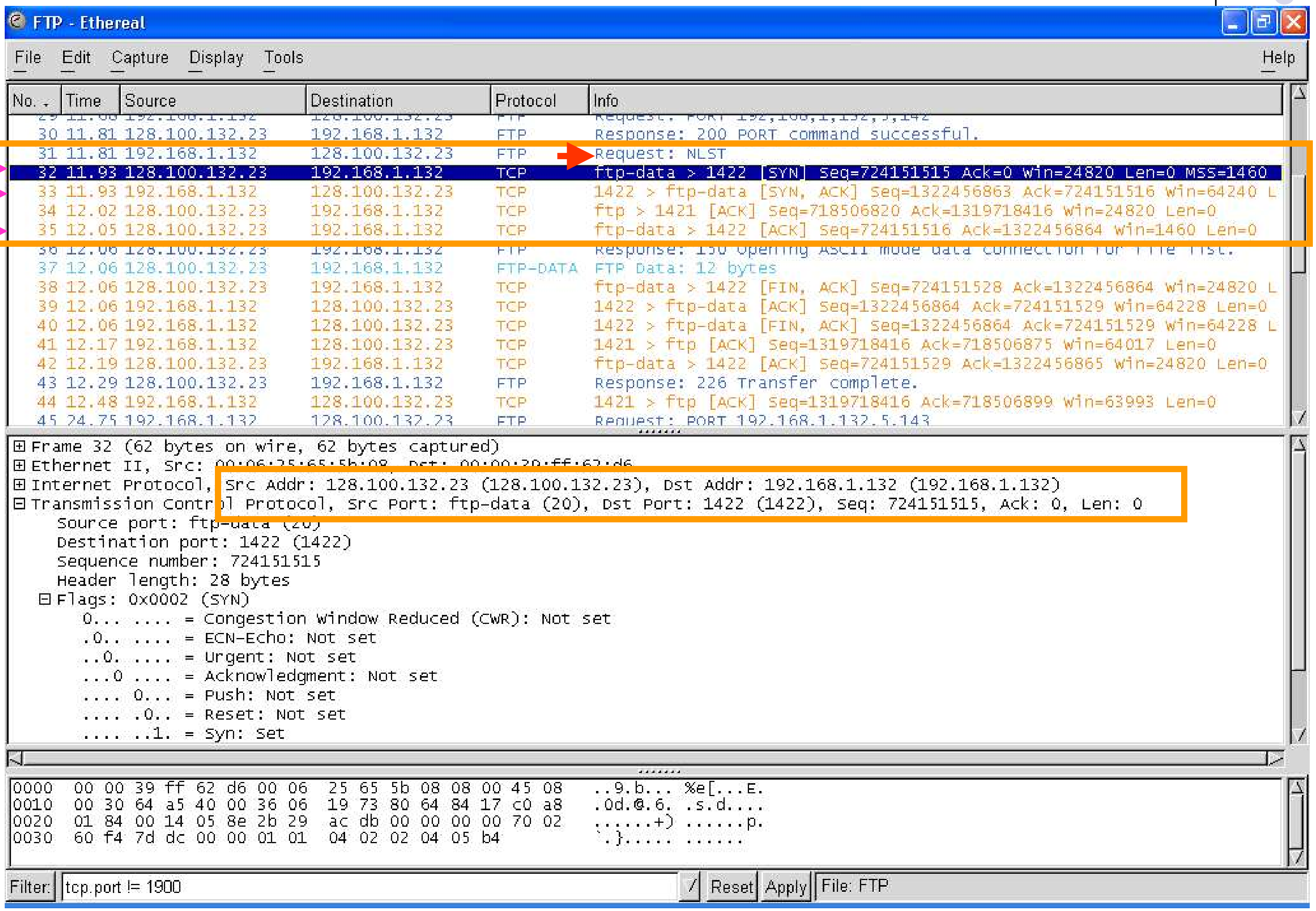
FTP Replies

Reply	Meaning
1yz	Positive preliminary reply (action has begun, but wait for another reply before sending a new command).
2yz	Positive completion reply (action completed successfully; new command may be sent).
3yz	Positive intermediary reply (command accepted, but action cannot be performed without additional information; user should send a command with the necessary information).
4yz	Transient negative completion reply (action currently cannot be performed; resend command later).
5zy	Permanent negative completion reply (action cannot be performed; do not resend it).
x0z	Syntax errors.
x1z	Information (replies to requests for status or help).
x2z	Connections (replies referring to the control and data connections).
x3z	Authentication and accounting (replies for the login process and accounting procedures).
x4z	Unspecified.
x5z	File system status.

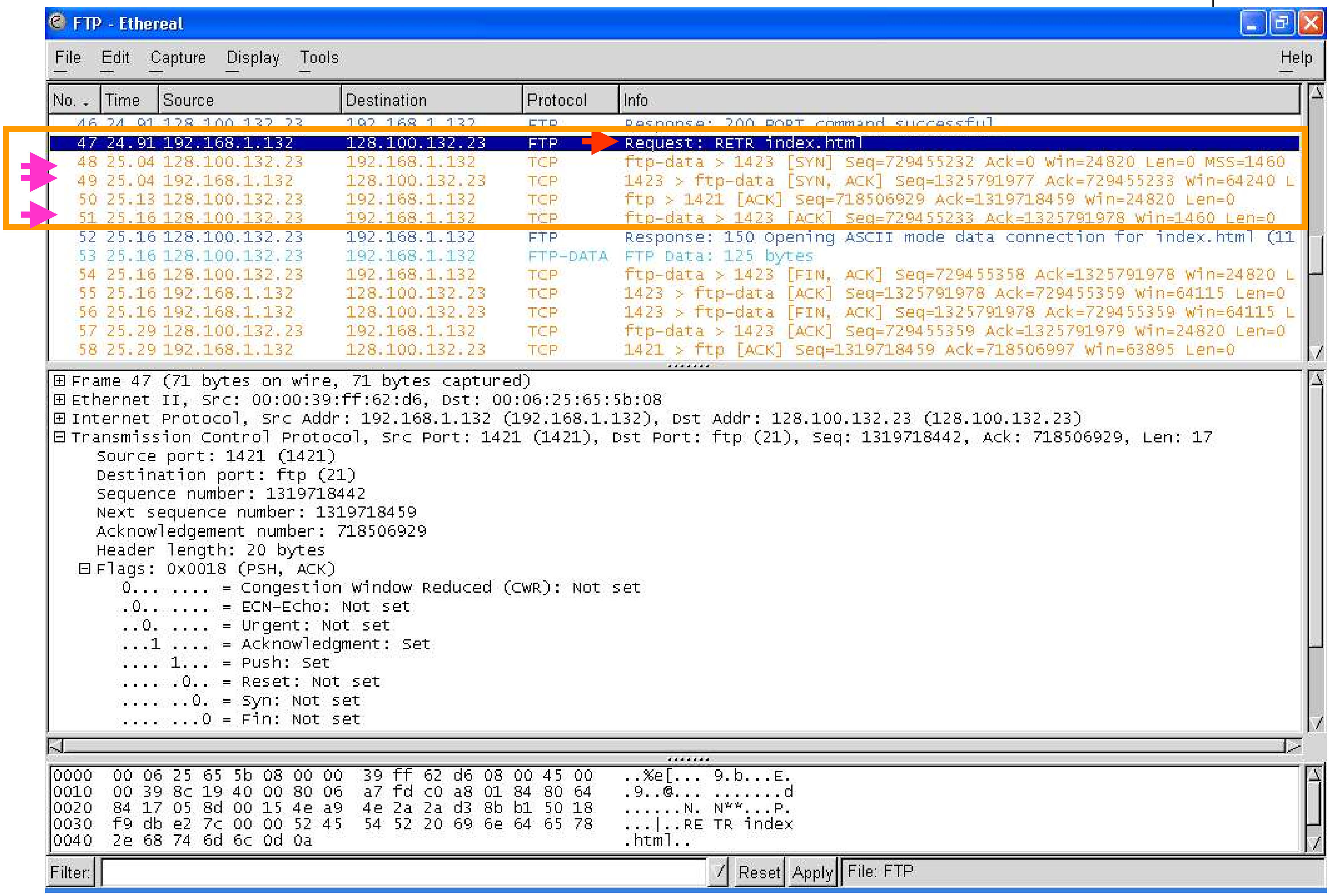
FTP Client (192.168.1.132: 1421) establishes Control Connection to FTP Server (128.100.132.23: 21)



User types *ls* to list files in directory (frame 31 on control) FTP Server (128.100.132.23: 20) establishes Data Connection to FTP Client (192.168.1.132: 1422)



User types *get index.html* to request file transfer in control connection (frame 47 request); File transfer on new data connection (port 1423, fr. 48, 49, 51)



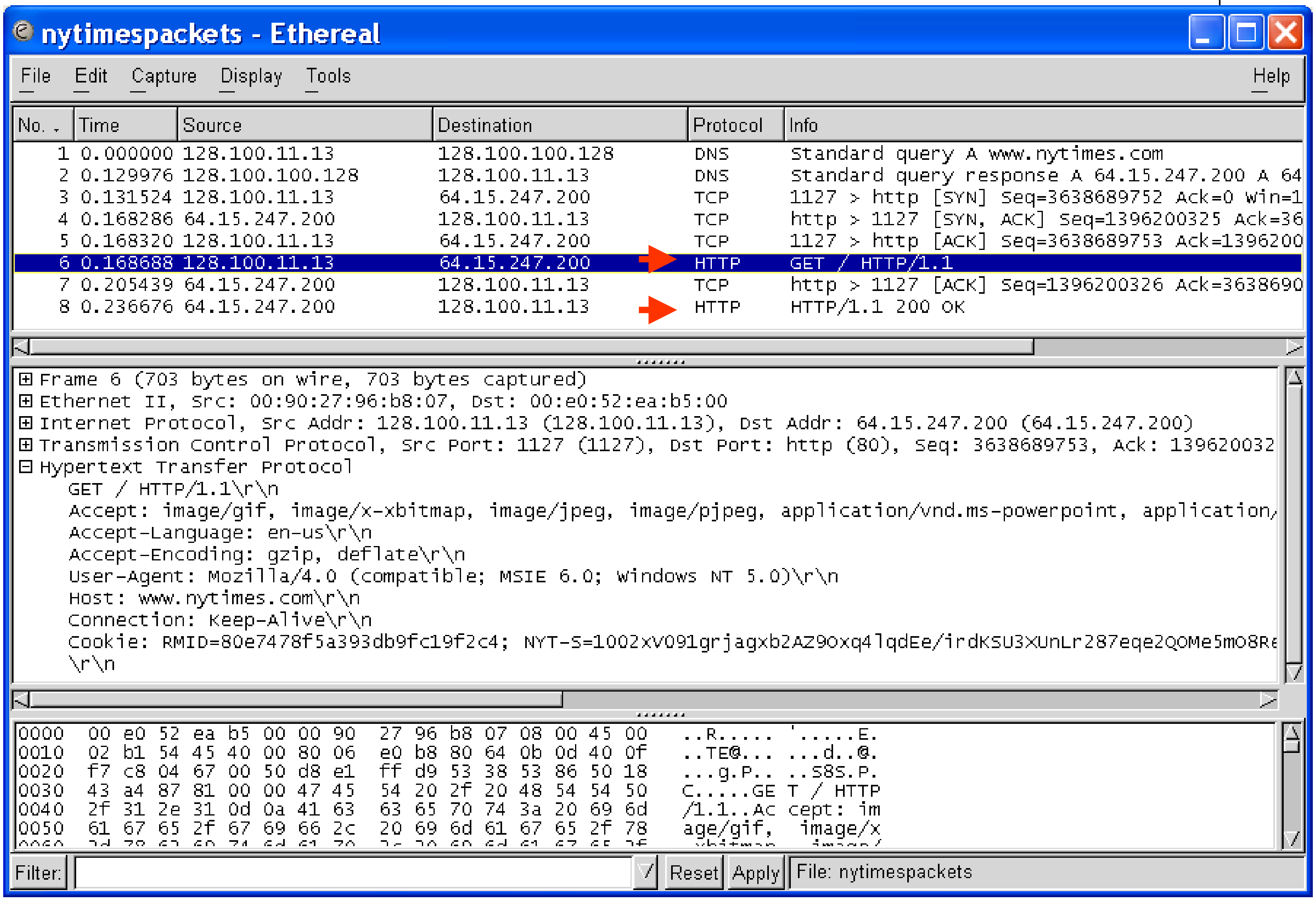
Hypertext Transfer Protocol

- RFC 1945 (HTTP 1.0), RFC 2616 (HTTP 1.1)
- HTTP provides communications between web browsers & web servers
- Web: framework for accessing documents & resources through the Internet
- Hypertext documents: text, graphics, images, hyperlinks
- Documents prepared using Hypertext Markup Language (HTML)

HTTP Protocol

- HTTP servers use well-known port 80
- Client request / Server reply
- Stateless: server does not keep any information about client
- HTTP 1.0 new TCP connection per request/reply (non-persistent)
- HTTP 1.1 persistent operation is default

HTTP Typical Exchange



HTTP Message Formats

- HTTP messages written in ASCII text
- Request Message Format
  - *Request Line (Each line ends with carriage return)*
    - *Method URL HTTP-Version \r\n*
    - Method specifies action to apply to object
    - URL specifies object
  - *Header Lines (Ea. line ends with carriage return)*
    - *Attribute Name: Attribute Value*
    - E.g. type of client, content, identity of requester, …
    - Last header line has extra carriage return)
  - *Entity Body (Content)*
    - Additional information to server

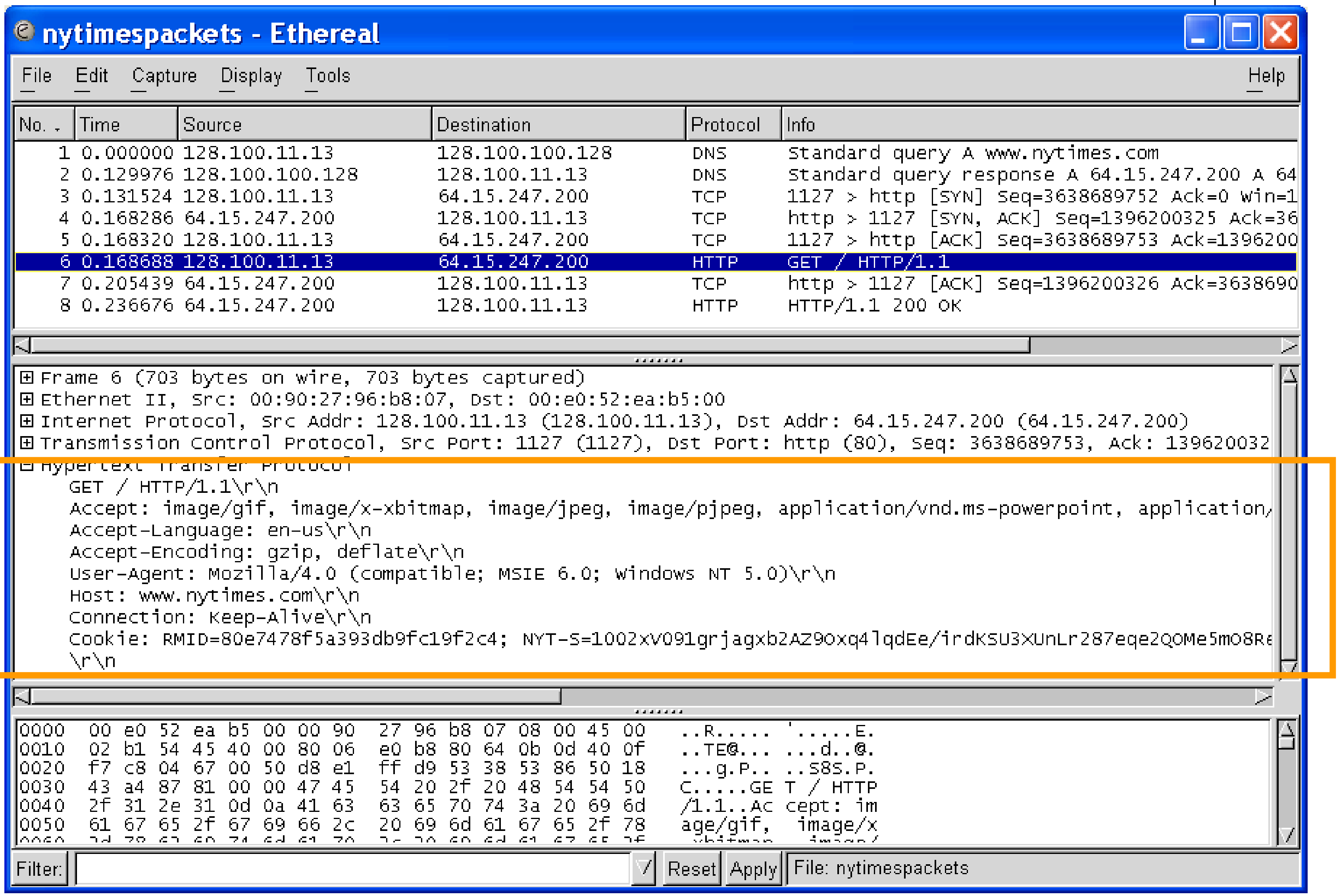
HTTP Request Methods

Request method	Meaning
GET	Retrieve information (object) identified by the URL.
HEAD	Retrieve meta-information about the object, but do not transfer the object; Can be used to find out if a document has changed.
POST	Send information to a URL (using the entity body) and retrieve result; used when a user fills out a form in a browser.
PUT	Store information in location named by URL
DELETE	Remove object identified by URL
TRACE	Trace HTTP forwarding through proxies, tunnels, etc.
OPTIONS	Used to determine the capabilities of the server, or characteristics of a named resource.

Universal Resource Locator

- Absolute URL
  - scheme://hostname[:port]/path
  - http://www.nytimes.com/
- Relative URL
  - /path
  - /

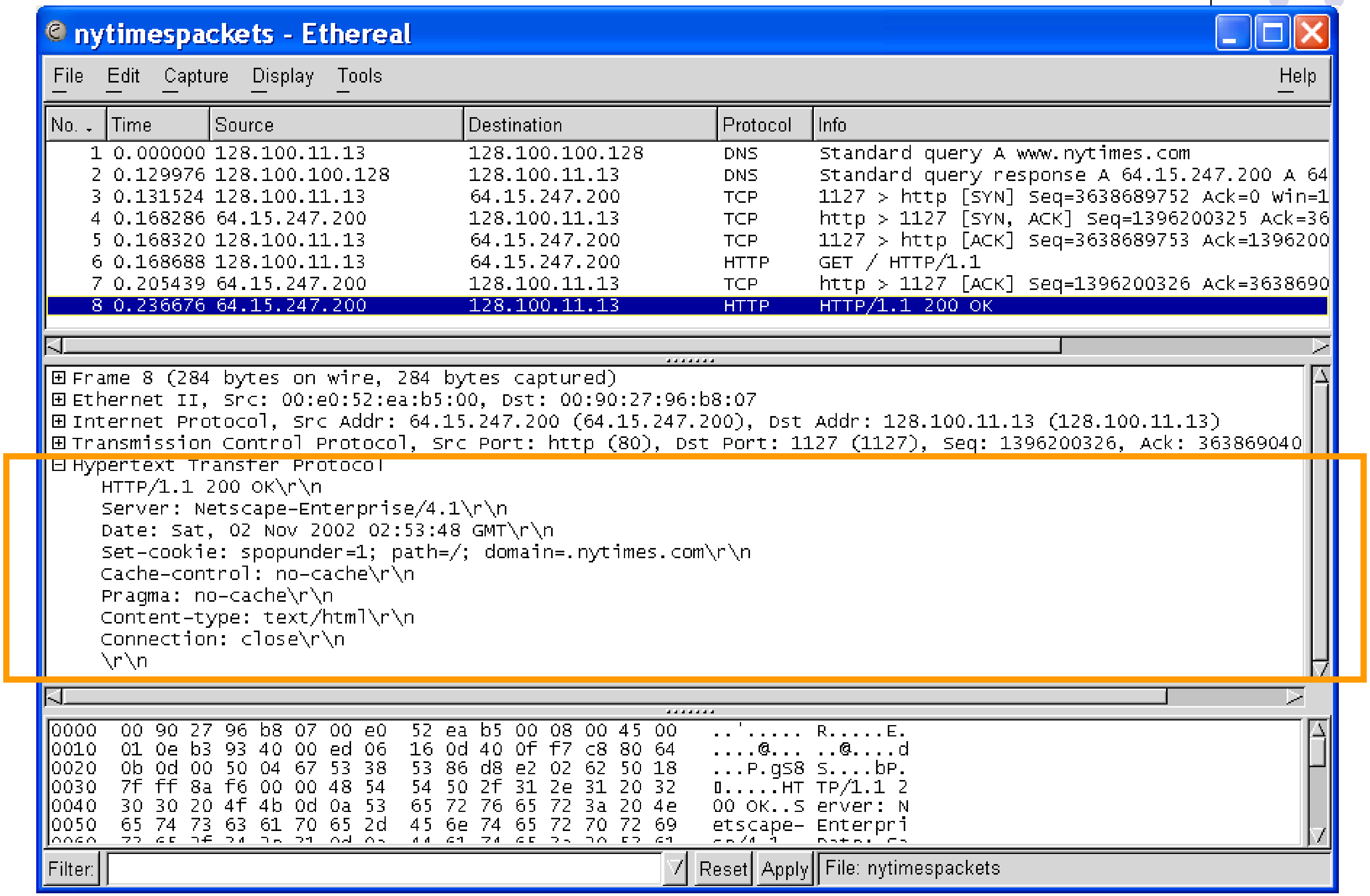
HTTP Request Message



HTTP Response Message

- Response Message Format
  - *Status Line*
    - *HTTP-Version Status-Code Message*
    - Status Code: 3-digit code indicating result
    - E.g. HTTP/1.0 200 OK
  - *Headers Section*
    - Information about object transferred to client
    - E.g. server type, content length, content type, …
  - *Content*
    - Object (document)

HTTP Response Message



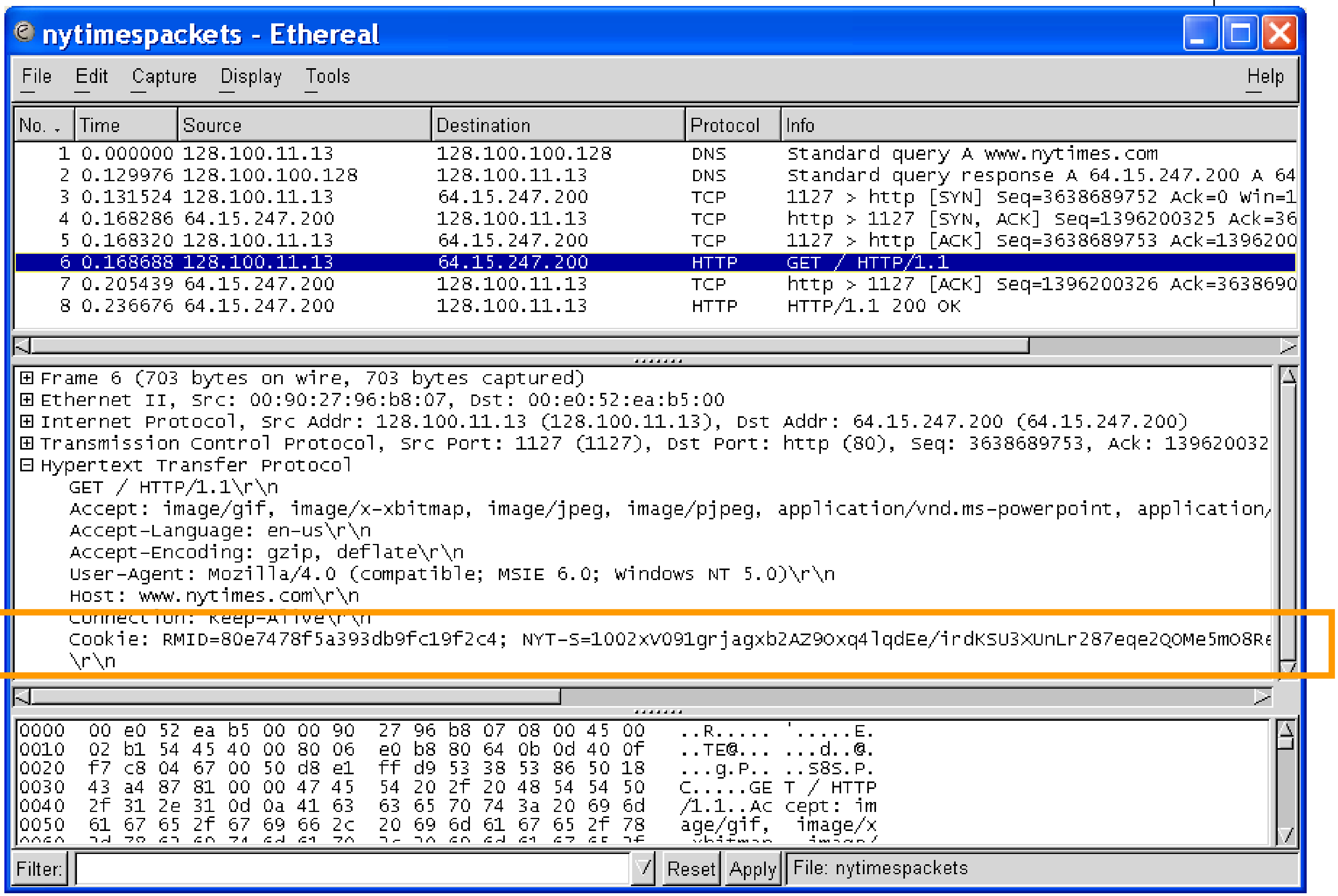
HTTP Proxy Server & Caching

- Web users generate large traffic volumes
- Traffic causes congestion & delay
- Can improve delay performance and reduce traffic in Internet by moving content to servers closer to the user
- Web proxy servers cache web information
  - Deployed by ISPs
  - Customer browsers configured to first access ISPs proxy servers
  - Proxy replies immediately when it has requested object or retrieves the object if it does not

Cookies and Web Sessions

- Cookies are data exchanged by clients & servers as header lines
- Since HTTP stateless, cookies can provide context for HTTP interaction
- *Set cookie* header line in reply message from server + unique ID number for client
- If client accepts cookie, cookie added to client’s cookie file (must include expiration date)
- Henceforth client requests include ID
- Server site can track client interactions, store these in a separate database, and access database to prepare appropriate responses

Cookie Header Line;
ID is 24 hexadecimal numeral



PING

- Application to determine if host is reachable
- Based on Internet Control Message Protocol
  - ICMP informs source host about errors encountered in IP packet processing by routers or by destination host
  - ICMP Echo message requests reply from destination host
- PING sends echo message & sequence #
- Determines reachability & round-trip delay
- Sometimes disabled for security reasons

PING from NAL host

```
Microsoft(R) Windows DOS
(c)Copyright Microsoft Corp 1990-2001.

C:\DOCUME~1\1>ping nal.toronto.edu

Pinging nal.toronto.edu [128.100.244.3] with 32 bytes of data:

Reply from 128.100.244.3: bytes=32 time=84ms TTL=240
Reply from 128.100.244.3: bytes=32 time=110ms TTL=240
Reply from 128.100.244.3: bytes=32 time=81ms TTL=240
Reply from 128.100.244.3: bytes=32 time=79ms TTL=240

Ping statistics for 128.100.244.3:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 79ms, Maximum = 110ms, Average = 88ms

C:\DOCUME~1\1>
```

Traceroute

- Find route from local host to a remote host
- Time-to-Live (TTL)
  - IP packets have TTL field that specifies maximum # hops traversed before packet discarded
  - Each router decrements TTL by 1
  - When TTL reaches 0 packet is discarded
- Traceroute
  - Send UDP to remote host with TTL=1
  - First router will reply ICMP Time Exceeded Msg
  - Send UDP to remote host with TTL=2, …
  - Each step reveals next router in path to remote host

Traceroute from home PC to university host

```
Tracing route to www.comm.utoronto.ca [128.100.11.60]
over a maximum of 30 hops:

  1     1 ms   <10 ms   <10 ms  192.168.2.1
  2     3 ms     3 ms     3 ms  10.202.128.1
  3     4 ms     3 ms     3 ms  gw04.ym.phub.net.cable.rogers.com [66.185.83.142]
  4     *        *        *     Request timed out.
  5    47 ms    59 ms    66 ms  gw01.bloor.phub.net.cable.rogers.com [66.185.80.230]
  6     3 ms     3 ms    38 ms  gw02.bloor.phub.net.cable.rogers.com [66.185.80.242]
  7     8 ms     3 ms     5 ms  gw01.wlfdle.phub.net.cable.rogers.com [66.185.80.2]
  8     8 ms     7 ms     7 ms  gw02.wlfdle.phub.net.cable.rogers.com [66.185.80.142]
  9     4 ms    10 ms     4 ms  gw01.front.phub.net.cable.rogers.com [66.185.81.18]
 10     6 ms     4 ms     5 ms  ra1sh-ge3-4.mt.bigpipeinc.com [66.244.223.237]
 11    16 ms    17 ms    13 ms  rx0sh-hydro-one-telecom.mt.bigpipeinc.com [66.244.223.246]
 12     7 ms    14 ms     8 ms  142.46.4.2
 13    10 ms     7 ms     6 ms  utorgw.onet.on.ca [206.248.221.6]
 14     7 ms     6 ms    11 ms  mcl-gateway.gw.utoronto.ca [128.100.96.101]
 15     7 ms     5 ms     8 ms  sf-gpb.gw.utoronto.ca [128.100.96.17]
 16     7 ms     7 ms    10 ms  bi15000.ece.utoronto.ca [128.100.96.236]
 17     7 ms     9 ms     9 ms  www.comm.utoronto.ca [128.100.11.60]

Trace complete.
```

Home Network

Rogers Cable ISP

Shaw Net

Hydro One

Ontario Net

University of Toronto

`ipconfig`

- Utility in Microsoft® Windows to display TCP/IP information about a host
- Many options
  - Simplest: IP address, subnet mask, default gateway for the host
  - Information about each IP interface of a host
    - DNS hostname, IP addresses of DNS servers, physical address of network card, IP address, …
  - Renew IP address from DHCP server

netstat

- Queries a host about TCP/IP network status
- Status of network drivers & their interface cards
  - #packets in, #packets out, errored packets, …
- State of routing table in host
- TCP/IP active server processes
- TCP active connections

netstat protocol statistics

```
IPv4 Statistics

  Packets Received                   = 71271
  Received Header Errors             = 0
  Received Address Errors            = 9
  Datagrams Forwarded                = 0
  Unknown Protocols Received         = 0
  Received Packets Discarded         = 0
  Received Packets Delivered         = 71271
  Output Requests                    = 70138
  Routing Discards                   = 0
  Discarded Output Packets           = 0
  Output Packet No Route             = 0
  Reassembly Required                = 0
  Reassembly Successful              = 0
  Reassembly Failures                = 0
  Datagrams Successfully Fragmented  = 0
  Datagrams Failing Fragmentation    = 0
  Fragments Created                  = 0

UDP Statistics for IPv4

  Datagrams Received     = 6810
  No Ports               = 15
  Receive Errors         = 0
  Datagrams Sent         = 6309
```

```
ICMPv4 Statistics

                            Received    Sent
  Messages                  10          6
  Errors                    0           0
  Destination Unreachable   8           1
  Time Exceeded             0           0
  Parameter Problems        0           0
  Source Quenches           0           0
  Redirects                 0           0
  Echos                     0           2
  Echo Replies              2           0
  Timestamps                0           0
  Timestamp Replies         0           0
  Address Masks             0           0
  Address Mask Replies      0           0

TCP Statistics for IPv4

  Active Opens                        = 798
  Passive Opens                       = 17
  Failed Connection Attempts          = 13
  Reset Connections                   = 467
  Current Connections                 = 0
  Segments Received                   = 64443
  Segments Sent                       = 63724
  Segments Retransmitted              = 80
```

`tcpdump` and Network Protocol Analyzers

- `tcpdump` program captures IP packets on a network interface (usually Ethernet NIC)
- Filtering used to select packets of interest
- Packets & higher-layer messages can be displayed and analyzed
- `tcpdump` basis for many network protocol analyzers for troubleshooting networks
- We use the open source Ethereal analyzer to generate examples
  - www.ethereal.com