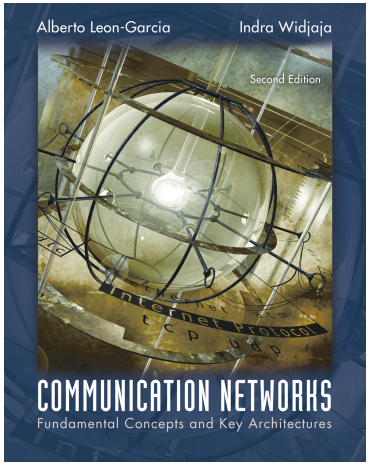
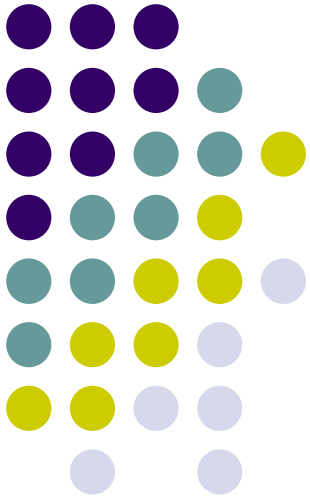
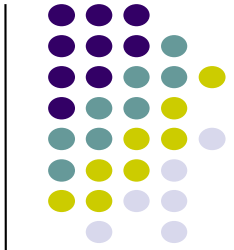
Chapter 2
Applications and Layered Architectures



Protocols, Services & Layering

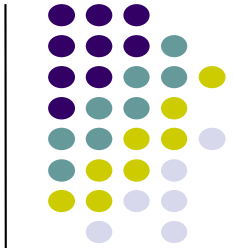


Layers, Services & Protocols



- 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***

Layer

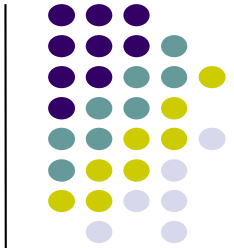


- A set of related communication functions that can be managed and grouped together

Examples of Layers

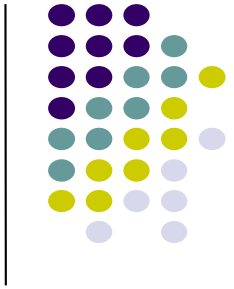
- Application Layer: communications functions that are used by application programs
  - HTTP (Browsing), DNS (URL to IP address), SMTP (email)
- Transport Layer: end-to-end communications *between two processes* in two machines
  - TCP Reliable Stream Service, UDP Datagram Service
- Network Layer: communications *between machines*
  - Internet Protocol (IP) best effort service

Protocols



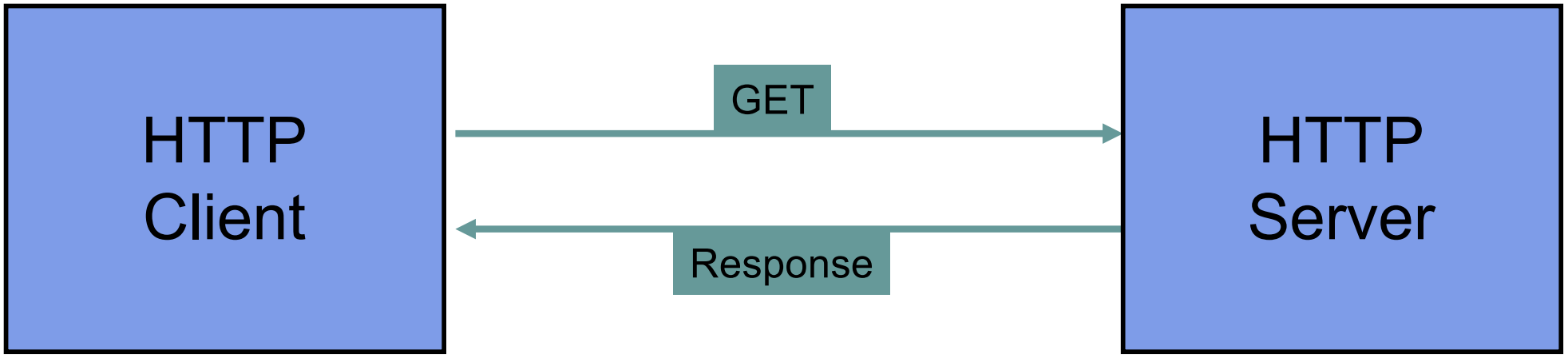
- A *protocol* is a set of rules that governs how two or more communicating entities *in a layer* are to interact
- *Messages* that can be sent and received
- *Actions* that are to be taken when a certain event occurs, e.g. sending or receiving messages, expiry of timers
- **The purpose of a protocol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

Example: HTTP



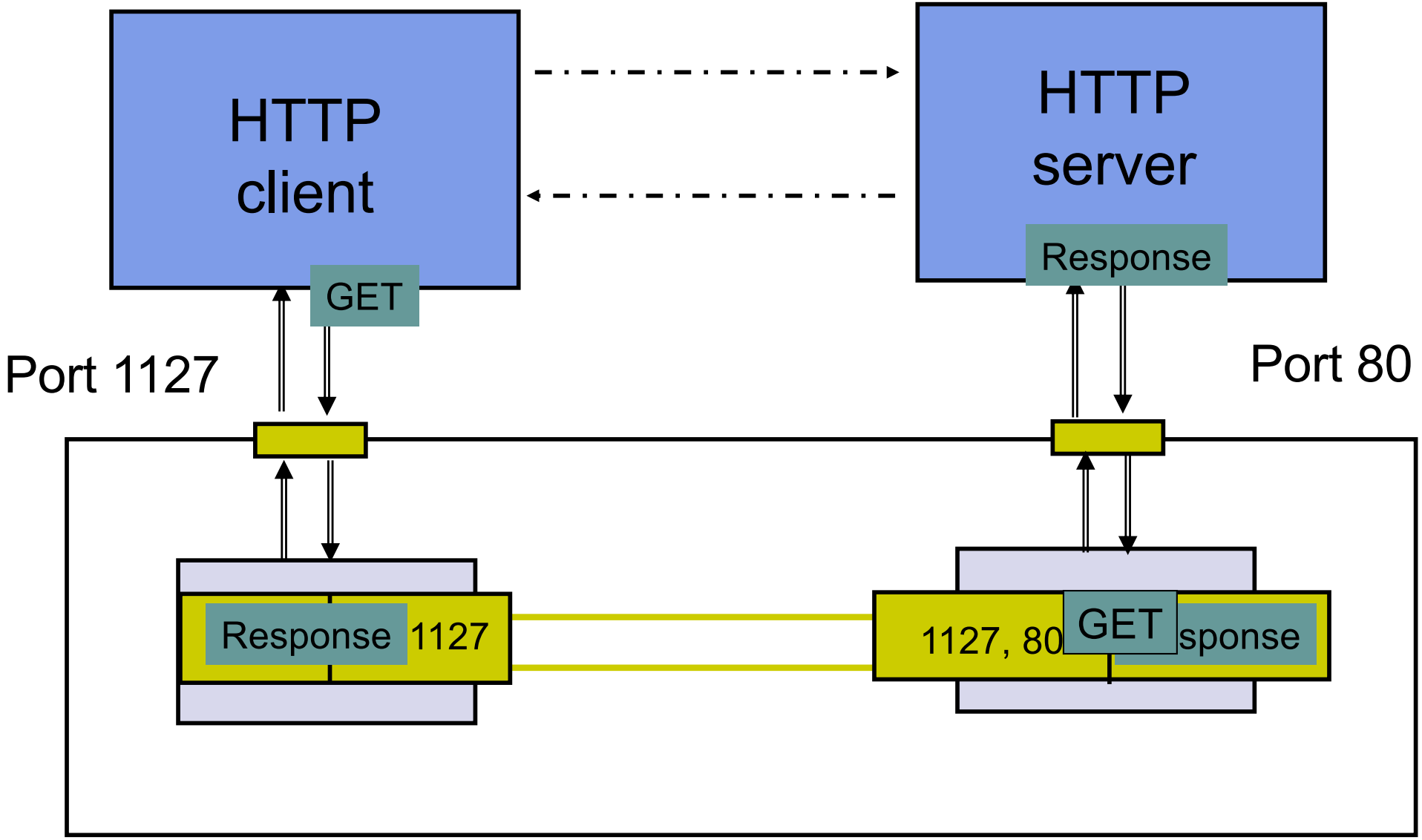
- HTTP is an application layer protocol
- Retrieves documents on behalf of a browser application program
- HTTP specifies fields in request messages and response messages
  - Request types; Response codes
  - Content type, options, cookies, …
- HTTP specifies actions to be taken upon receipt of certain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

HTTP Protocol

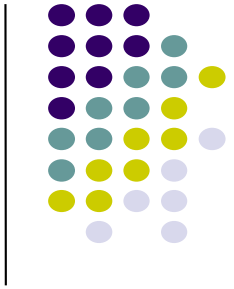


- HTTP provides service to a browsing application
- HTTP messages exchanged between HTTP client and HTTP server
- HTTP client and server are processes running in two different machines across the Internet
- HTTP uses the reliable stream transfer service provided by TCP (Transmission Control Protocol)

HTTP uses service of TCP



Example: TCP

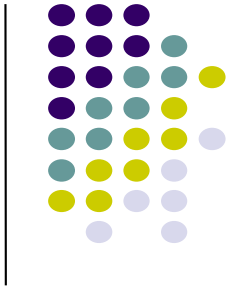


- TCP is a transport layer protocol
- Provides *reliable byte stream service between two processes* in two computers across the Internet
- Sequence numbers keep track of the bytes that have been transmitted and received
- Error detection and retransmission used to recover from transmission errors, losses, duplications
- 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

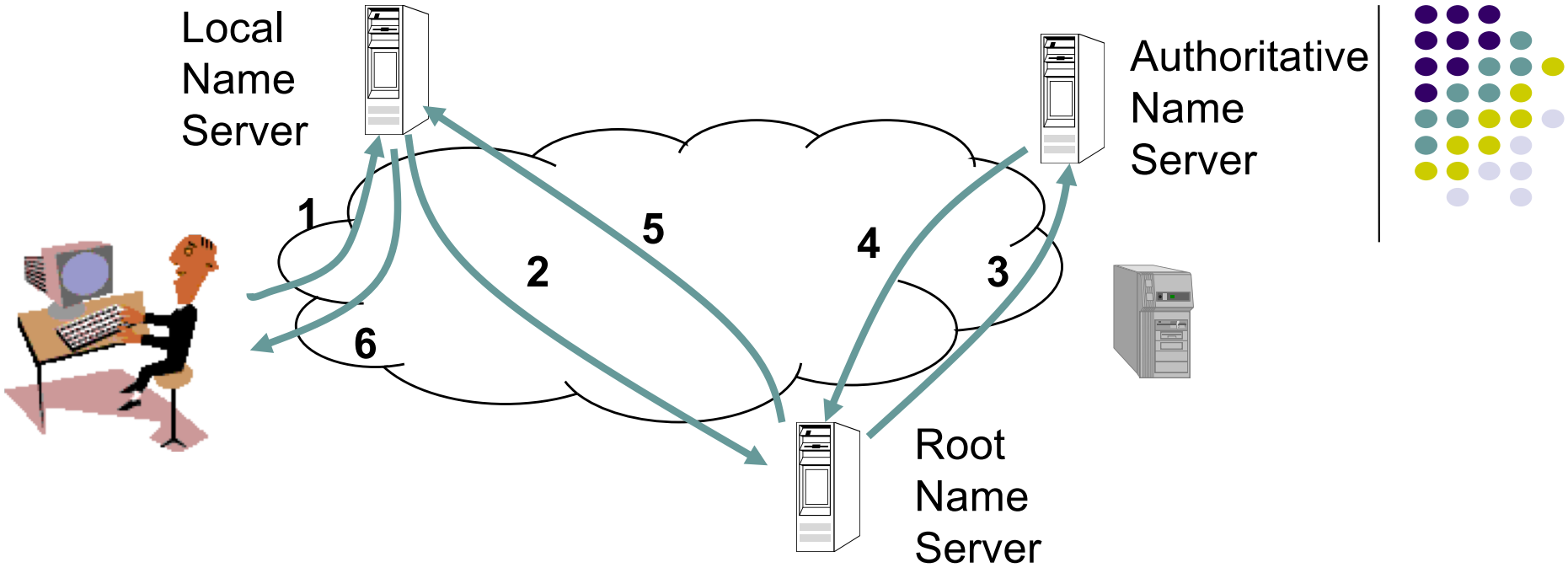
(send port #, send IP address, receive port #, receiver IP address)

- HTTP server has well-known port # 80

Example: DNS Protocol

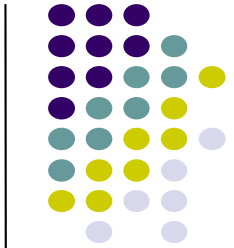


- DNS (Domain Name System) is an application layer protocol
- DNS is a distributed database that resides in multiple machines in the Internet
- DNS protocol allows queries of different types
  - Name-to-address or Address-to-name
  - Mail exchange
- DNS usually involves short messages and so uses service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



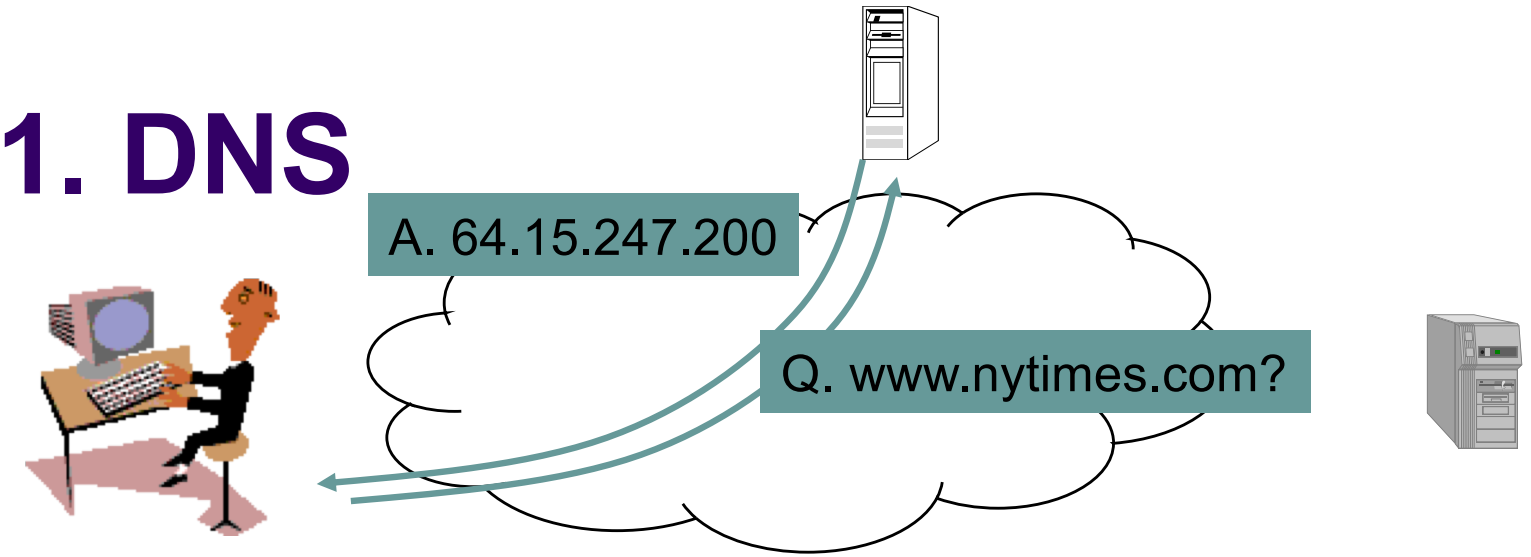
- Local Name Server: resolve frequently-used names
  - University department, ISP
  - Contacts Root Name server if it cannot resolve query
- Root Name Servers: 13 globally (actually 100's)
  - Resolves query or refers query to Authoritative Name Server
- Authoritative Name Server: last resort
  - Every machine must register its address with at least two authoritative name servers

Web Browsing Application



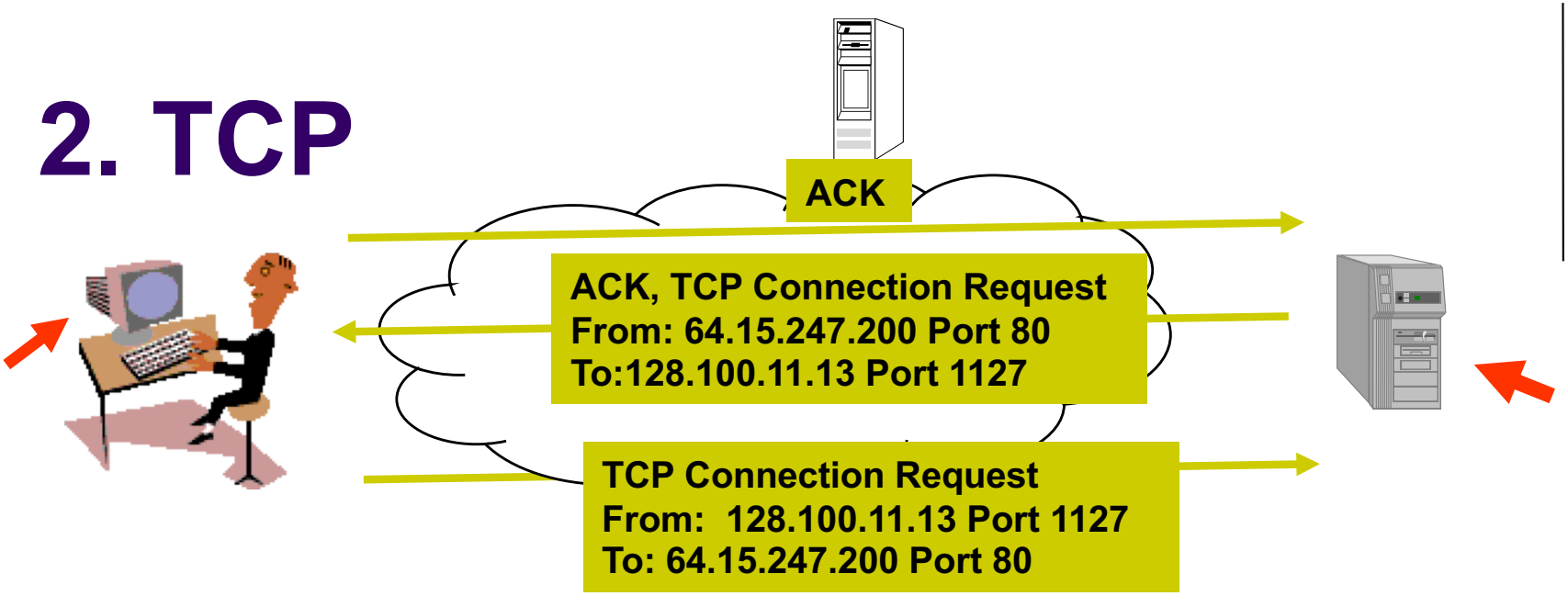
- World Wide Web allows users to access resources (i.e. documents) located in computers connected to the Internet
- Documents prepared using HyperText Markup Language (HTML)
- A browser application program is used to access the web
- The browser displays HTML documents that include *links* to other documents
- 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



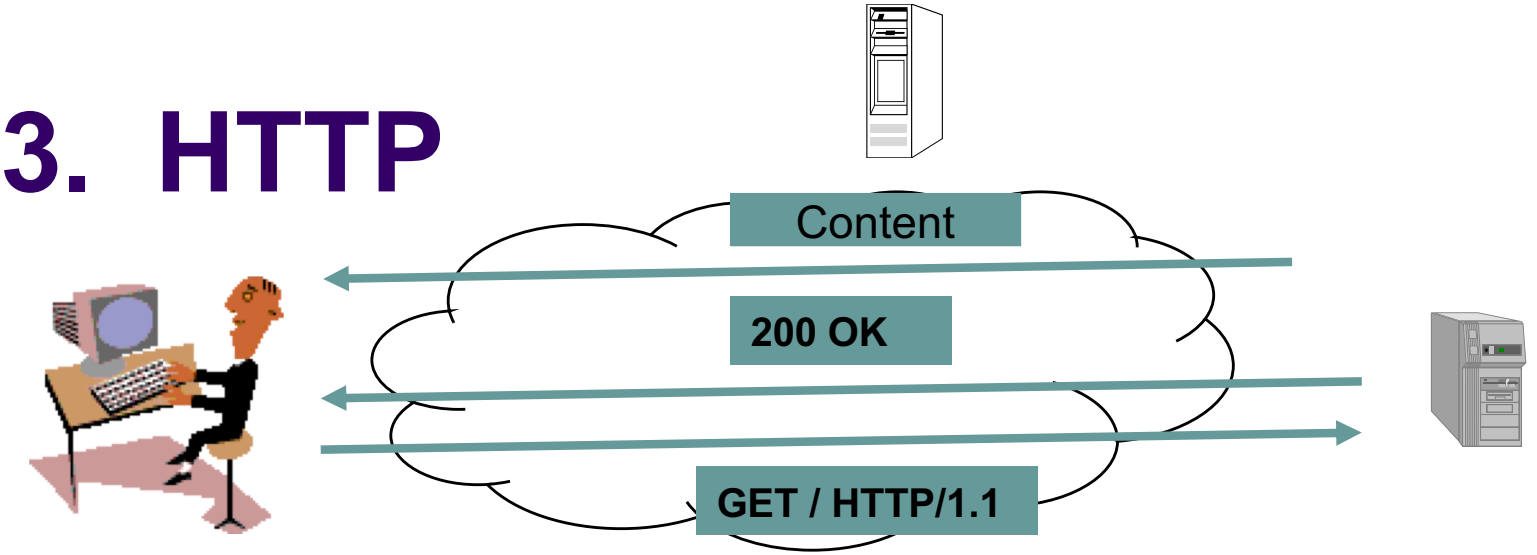
- User clicks on http://www.nytimes.com/
- URL contains Internet name of machine (www.nytimes.com), but not Internet address
- Internet needs Internet address to send packets to a machine
- Browser software uses DNS protocol to send query for Internet address
- DNS system responds with Internet address

2. TCP



- Browser uses 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 TCP connection between the HTTP client and HTTP server to transfer messages reliably

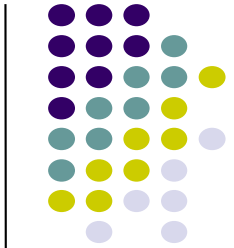
3. HTTP



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

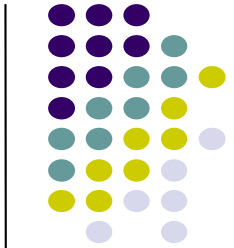
- Clicking a link sets off a chain of events across the Internet!
- DNS, TCP, HTTP, Browser display

Example: UDP



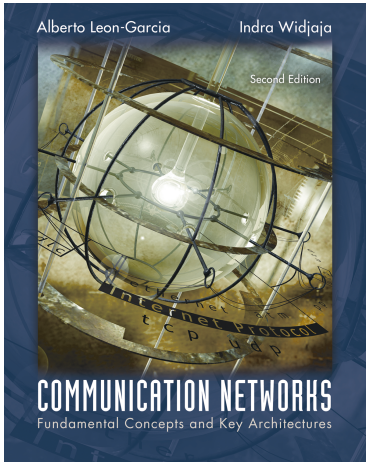
- UDP is a transport layer protocol
- Provides *best-effort datagram service* between two processes in two computers across the Internet
- Port numbers distinguish various processes in the same machine
- UDP is *connectionless*
- Datagram is sent immediately
- Quick, simple, but not reliable
- Used in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

Summary

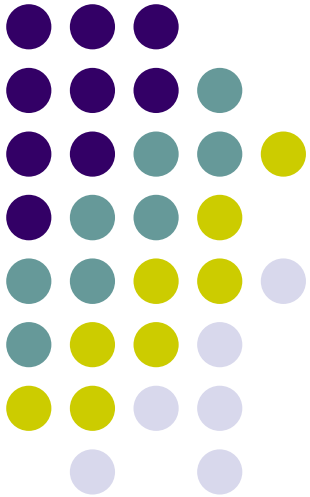


- Layers: related communications functions
  - Application Layer: HTTP, DNS
  - Transport Layer: TCP, UDP
  - Network Layer: IP
- Services: a protocol provides a communications service to the layer above
  - TCP provides connection-oriented reliable stream service
  - UDP provides connectionless datagram 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

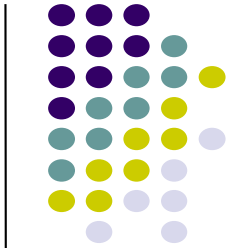
Chapter 2
Applications and Layered Architectures



OSI Reference Model

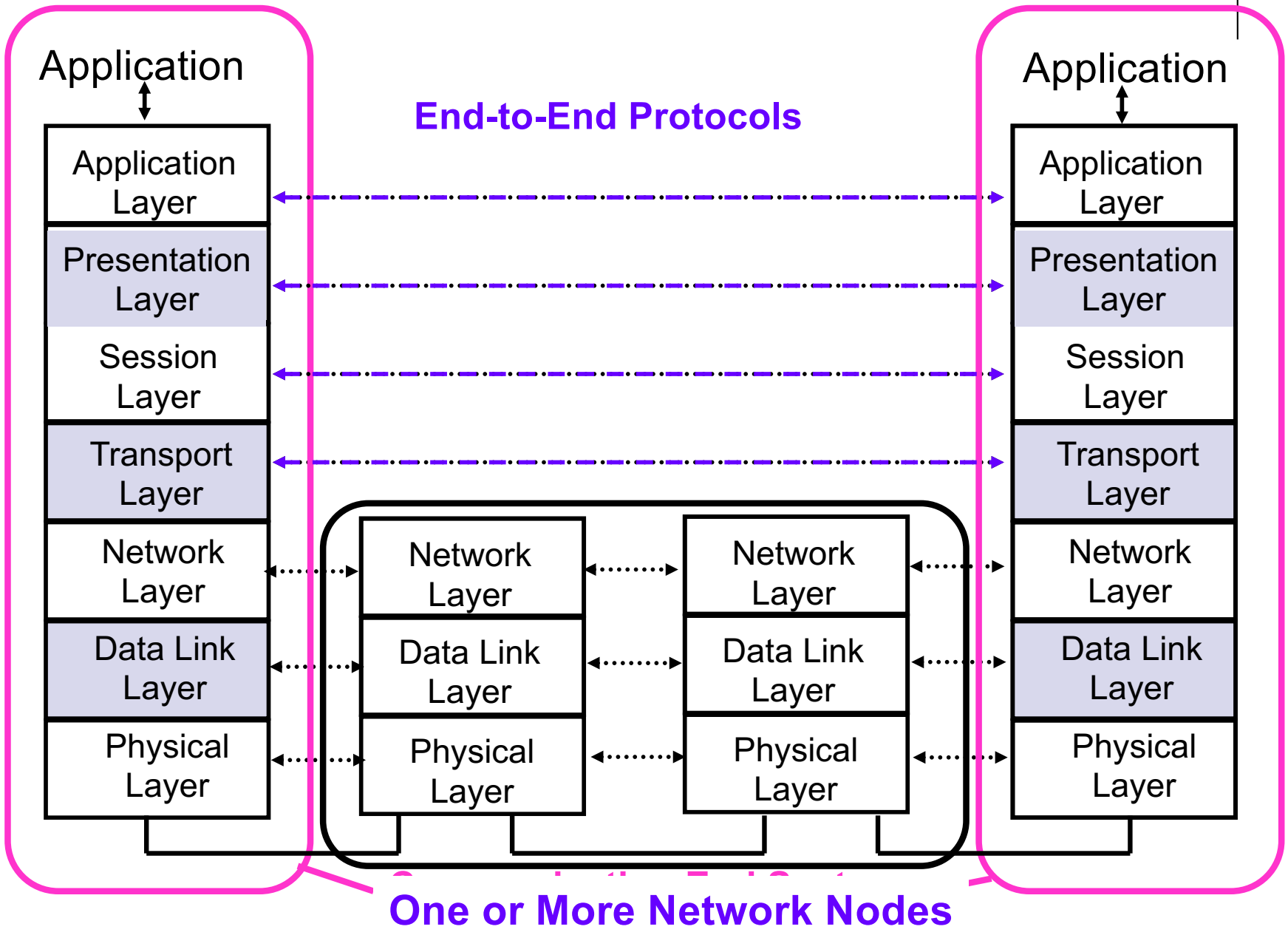


OSI Reference Model



- A seven-layer abstract reference model for a network architecture
- Provides a framework for the development of protocols
- Unified view of layers, protocols, and services which is used 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

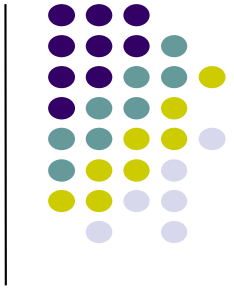
7-Layer OSI Reference Model



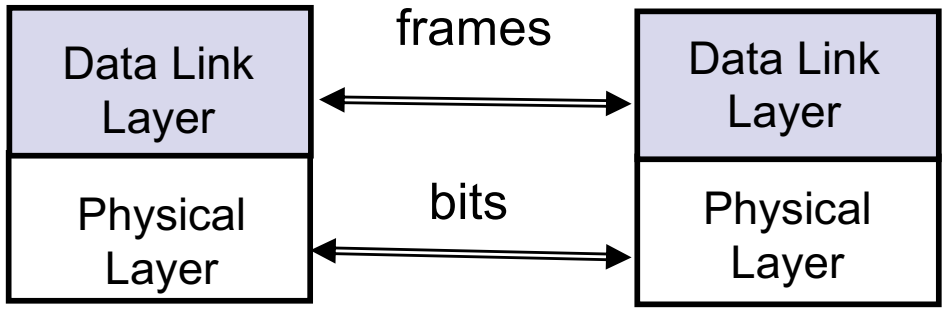
Physical Layer

- 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, cell phone,…
- Twisted-pair cable, coaxial cable optical fiber, radio, infrared, …

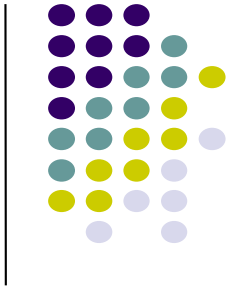
Data Link Layer



- Transfers ***frames*** between *directly connected* machines
- Groups bits into frames
- Detection of bit errors; Retransmission of frames
- Activation, maintenance, & deactivation of data link connections
- Access control for local area networks (WIFI, Ethernet)
- Flow control between machines of different speeds



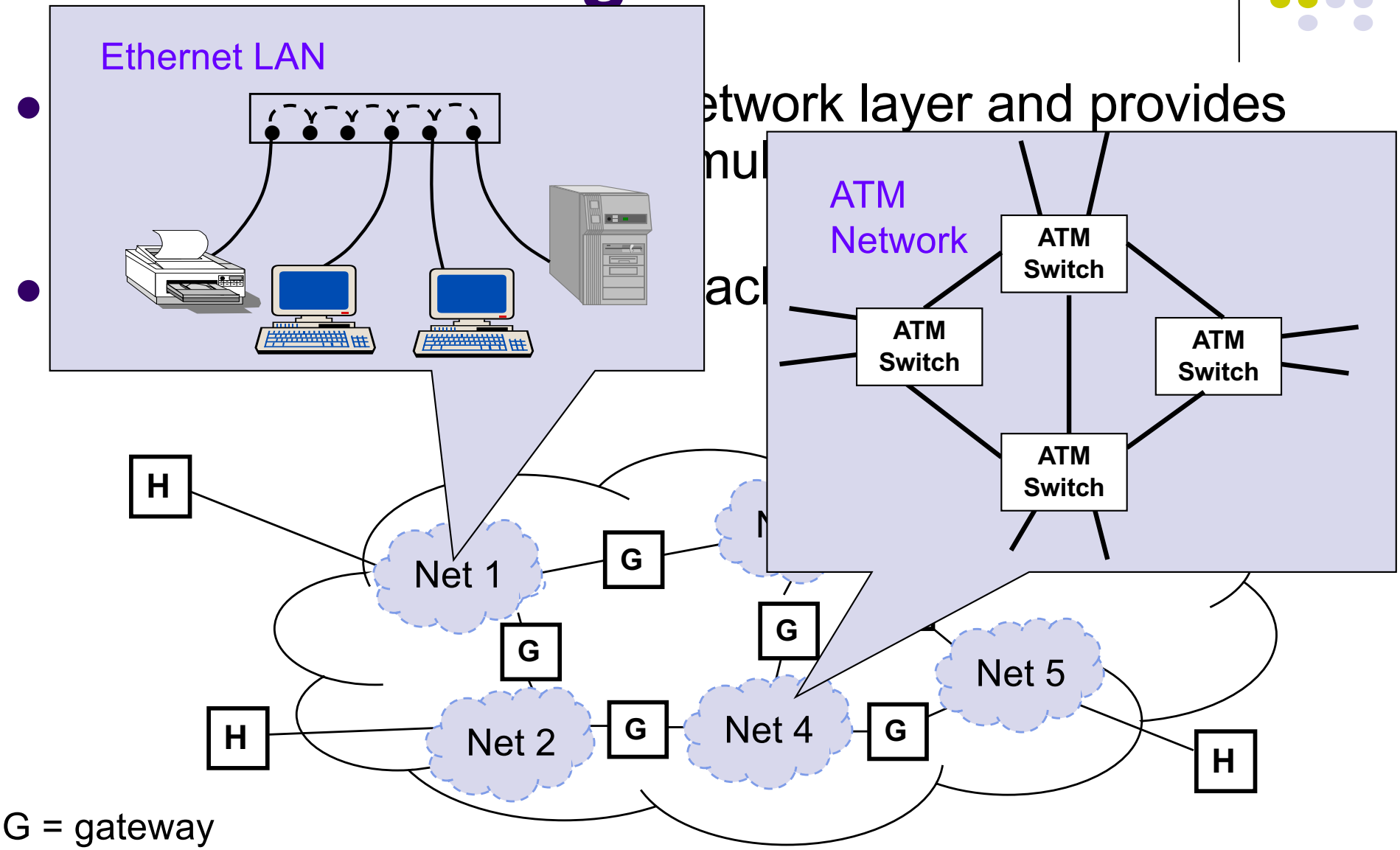
Network Layer



- 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 along hops in path
- Congestion control to deal with traffic surges
- Connection setup, maintenance, and teardown when connection-based

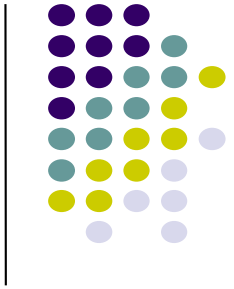
Internetworking

- etwork layer and provides
  mul
- ac

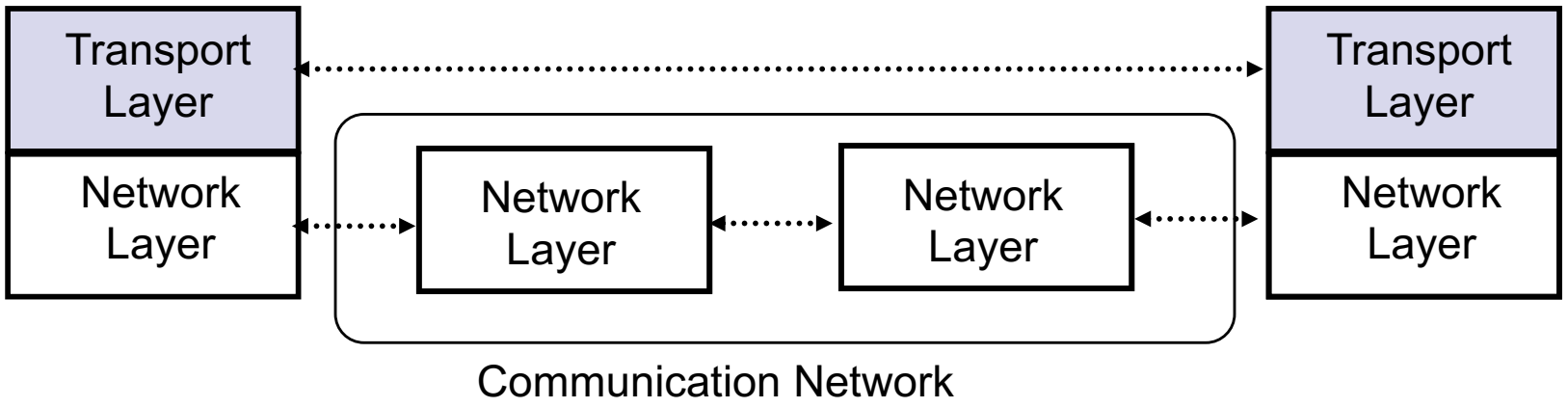


G = gateway
H = host

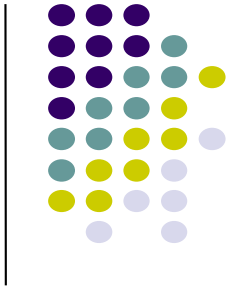
Transport Layer



- 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

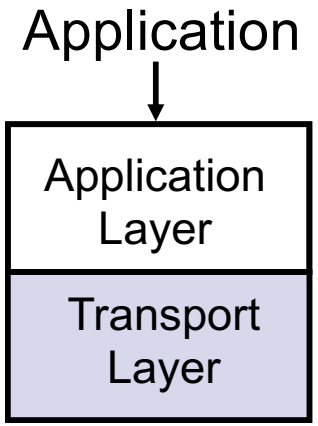


Application & Upper Layers



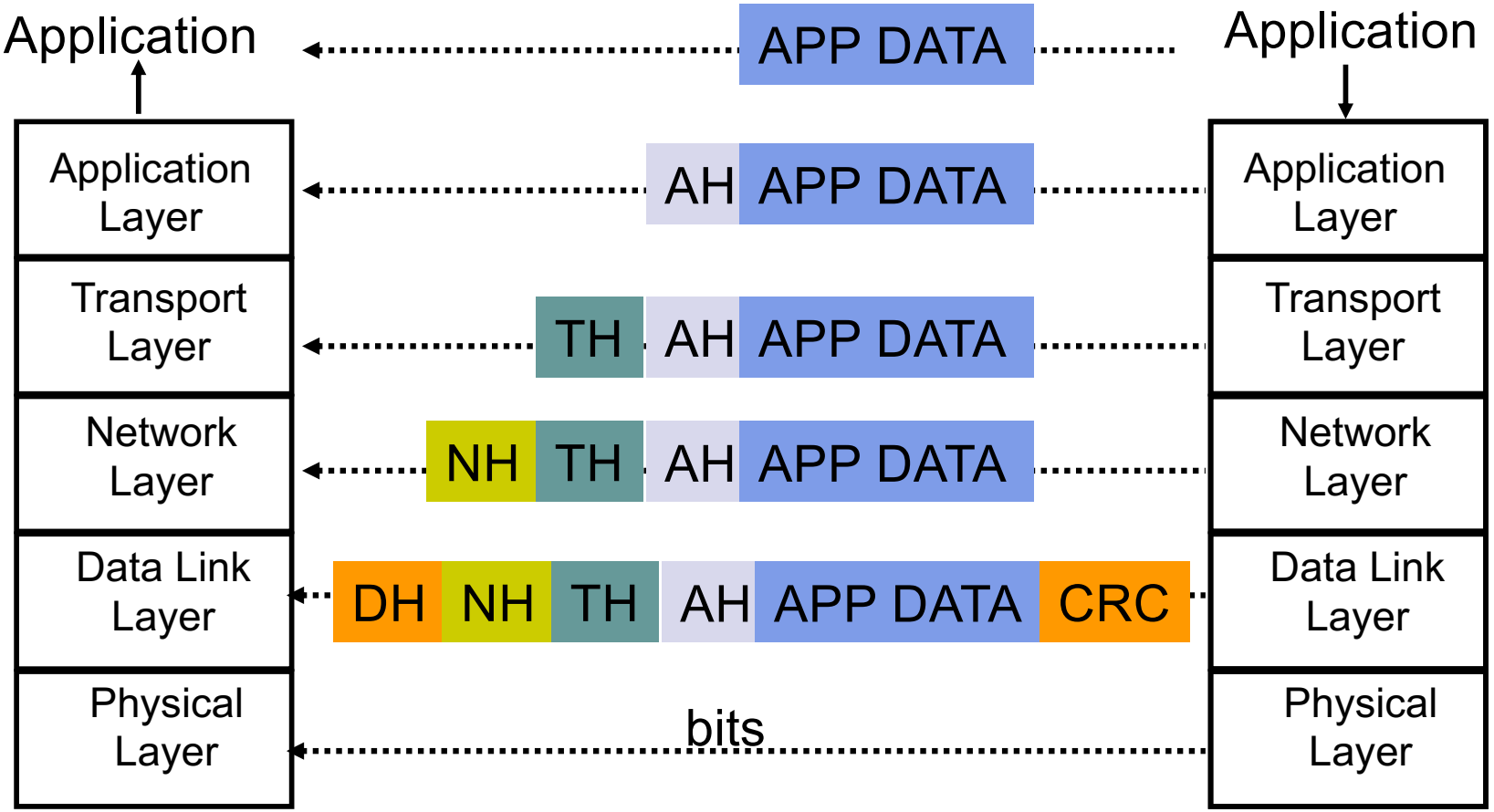
- Application Layer: Provides services that are frequently required by applications: DNS, web acess, file transfer, email…
- Presentation Layer: machine-independent representation of data…
- Session Layer: dialog management, recovery from errors, …

Incorporated into Application Layer

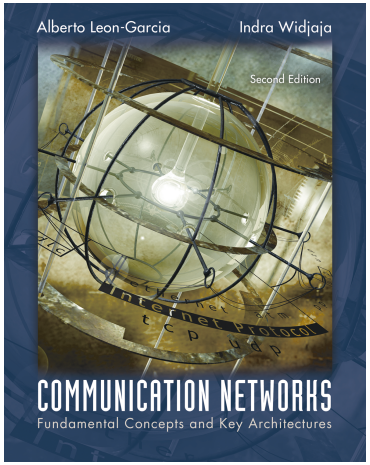


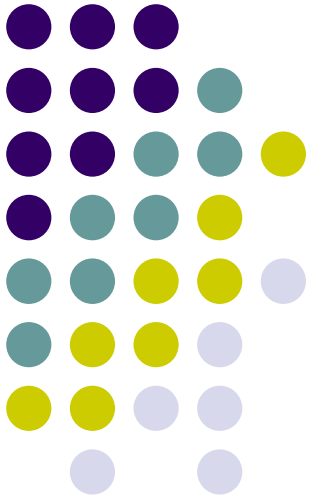
Headers & Trailers

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



Chapter 2
Applications and Layered Architectures



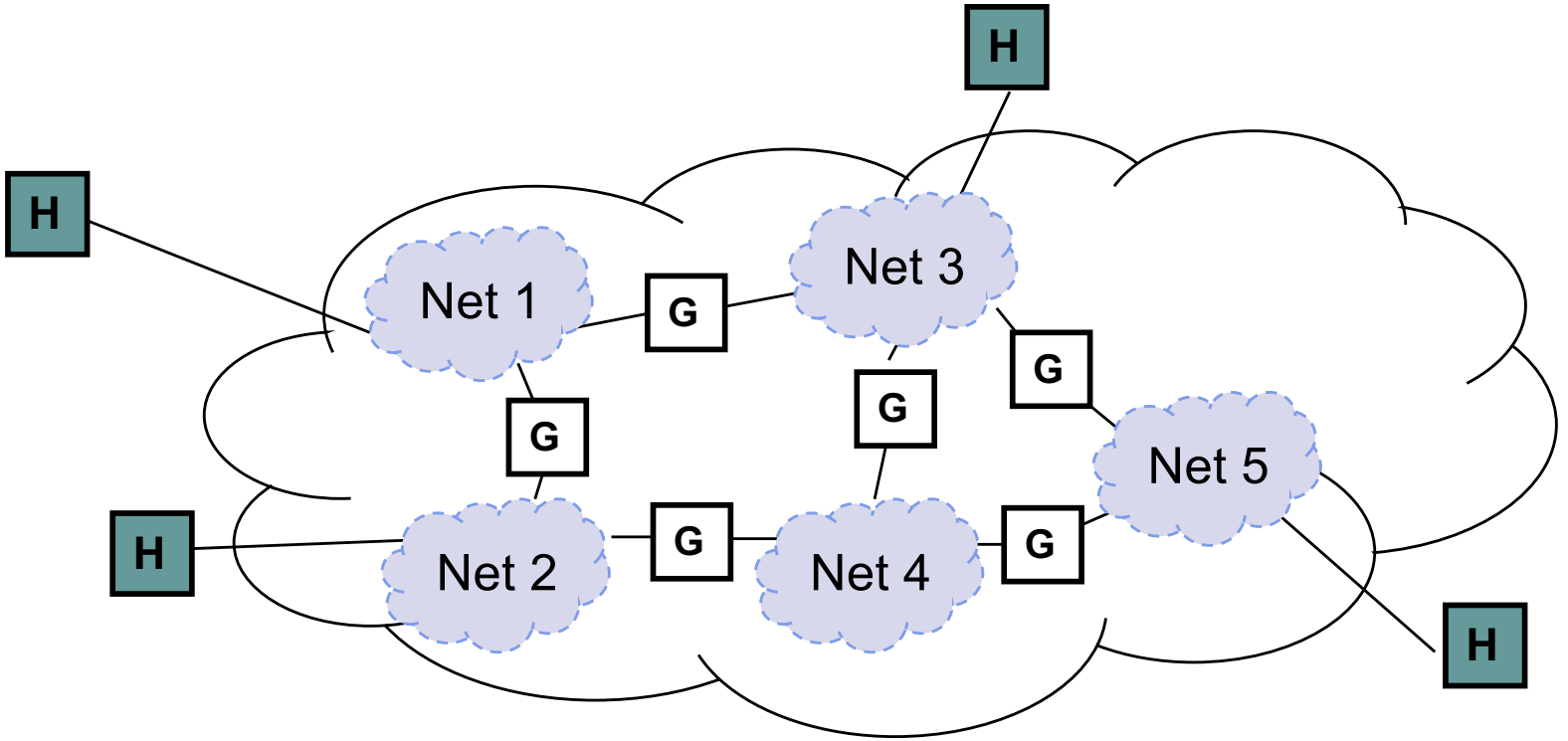


TCP/IP Architecture

How the Layers Work Together

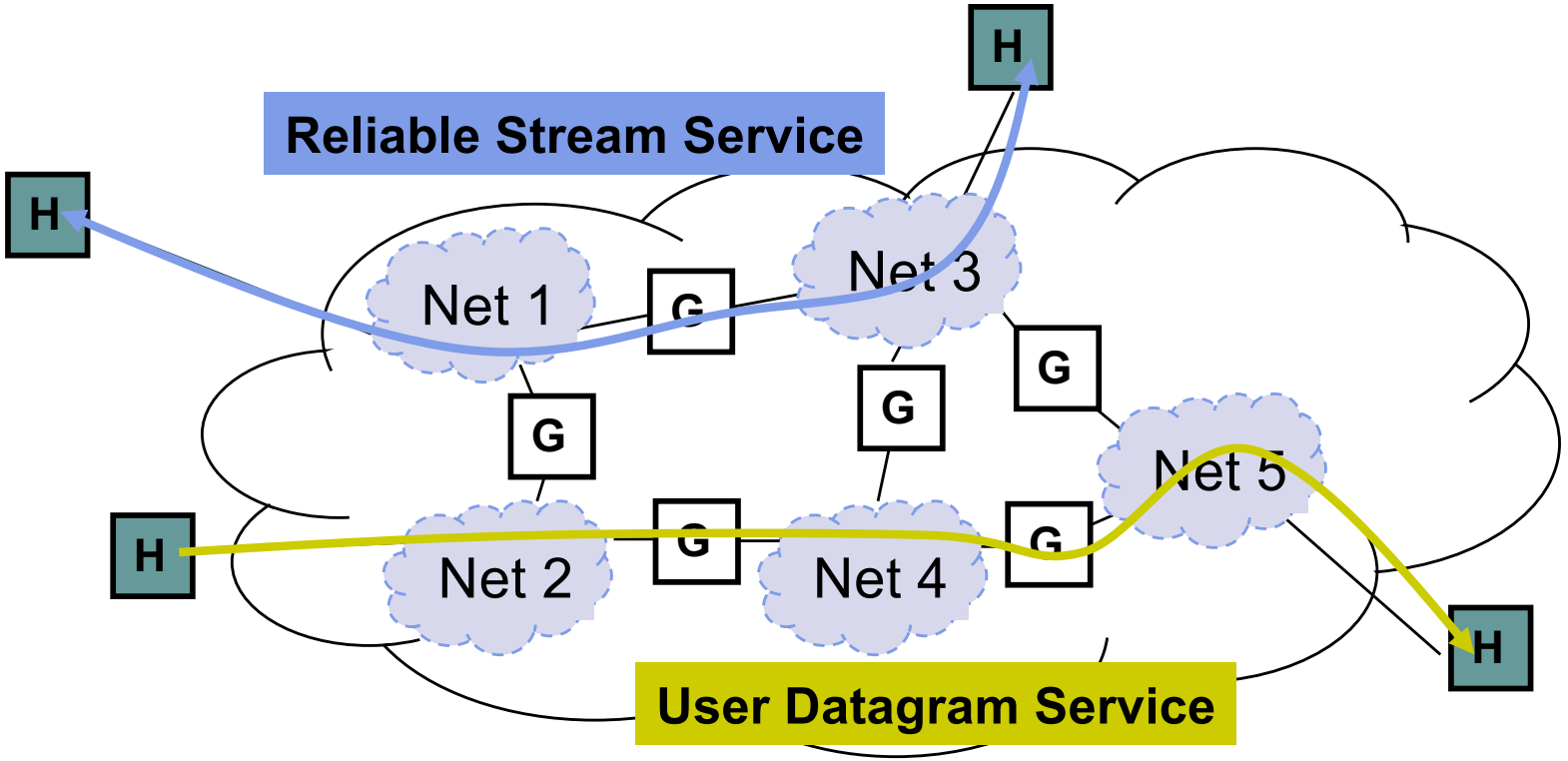
Why Internetworking?

- 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

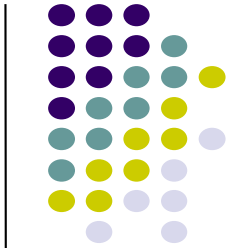


Why Internetworking?

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



Why Internetworking?

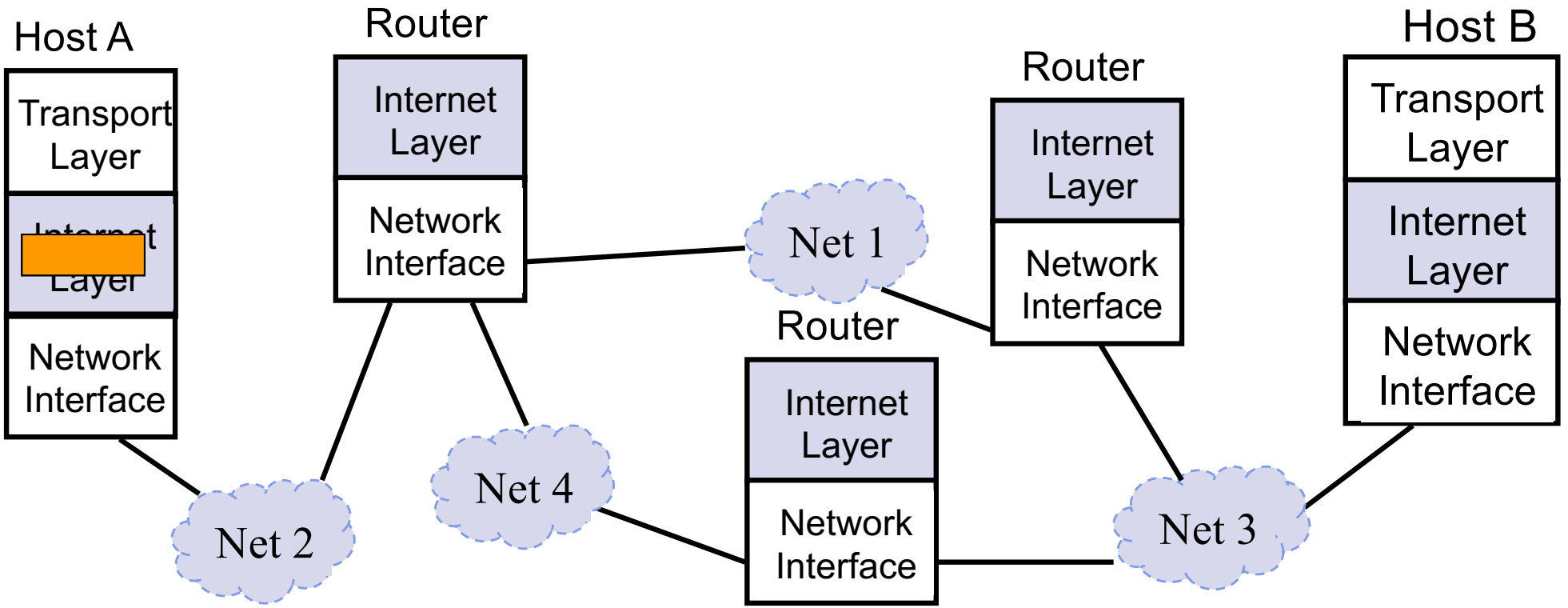


- To provide *distributed applications*
  - Any application designed to operate based on Internet communication services immediately operates across the entire Internet
  - Rapid deployment of new applications
    - Email, WWW, Peer-to-peer
  - Applications independent of network technology
    - New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

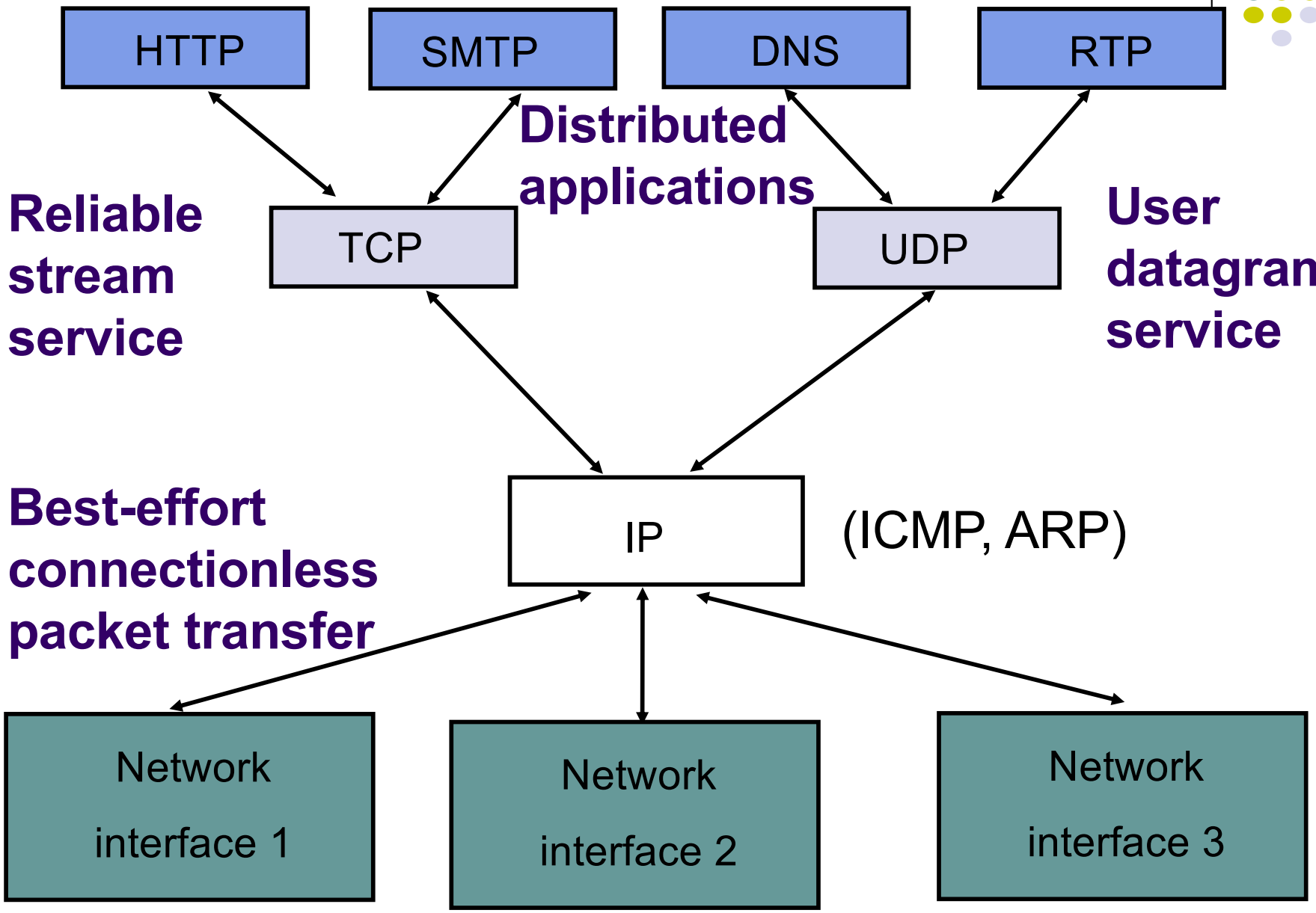
Internet Protocol Approach

- IP packets transfer information across Internet

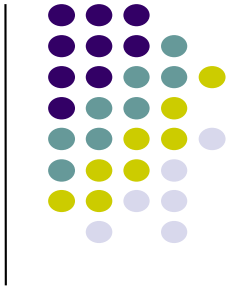
  Host A IP → router→ router…→ router→ Host B IP
- IP layer in each router determines next hop (router)
- Network interfaces transfer IP packets across networks



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:

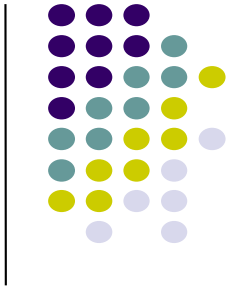
  int1.int2.int3.int4

  (intj = jth octet)

  128.100.10.13

DNS resolves IP name to IP address

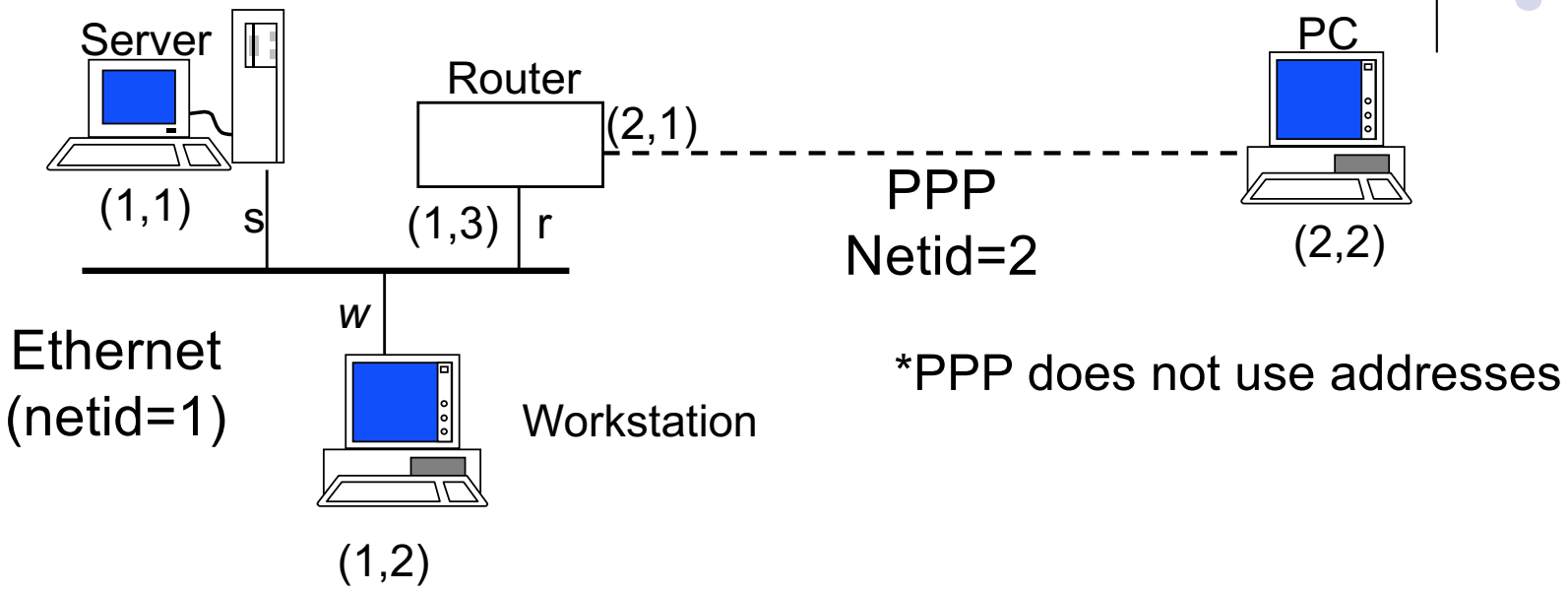
Physical Addresses



- 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
  - 00:90:27:96:68:07 12 hex numbers

    Intel

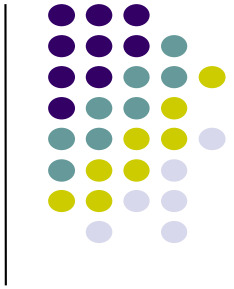
Example internet

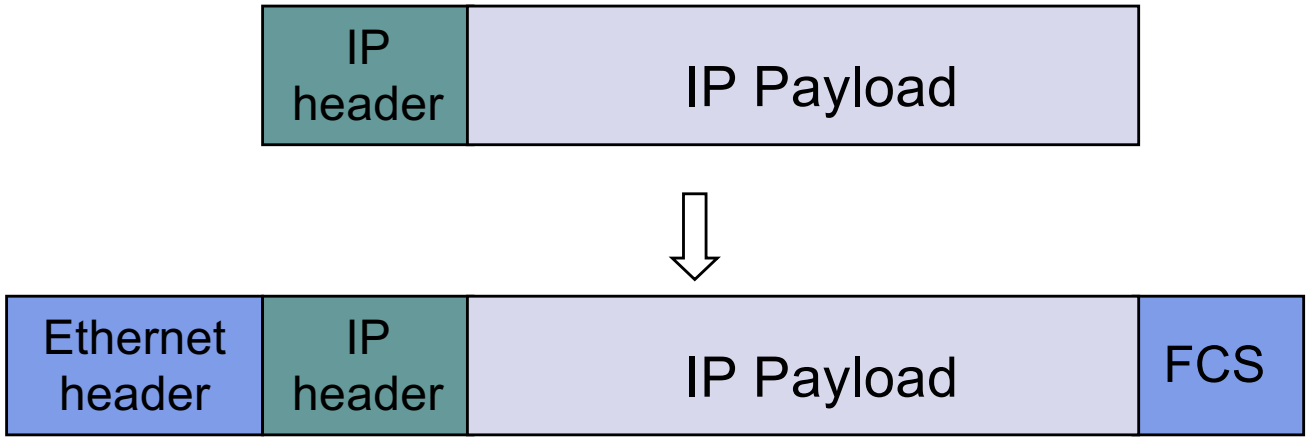


*PPP does not use addresses

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

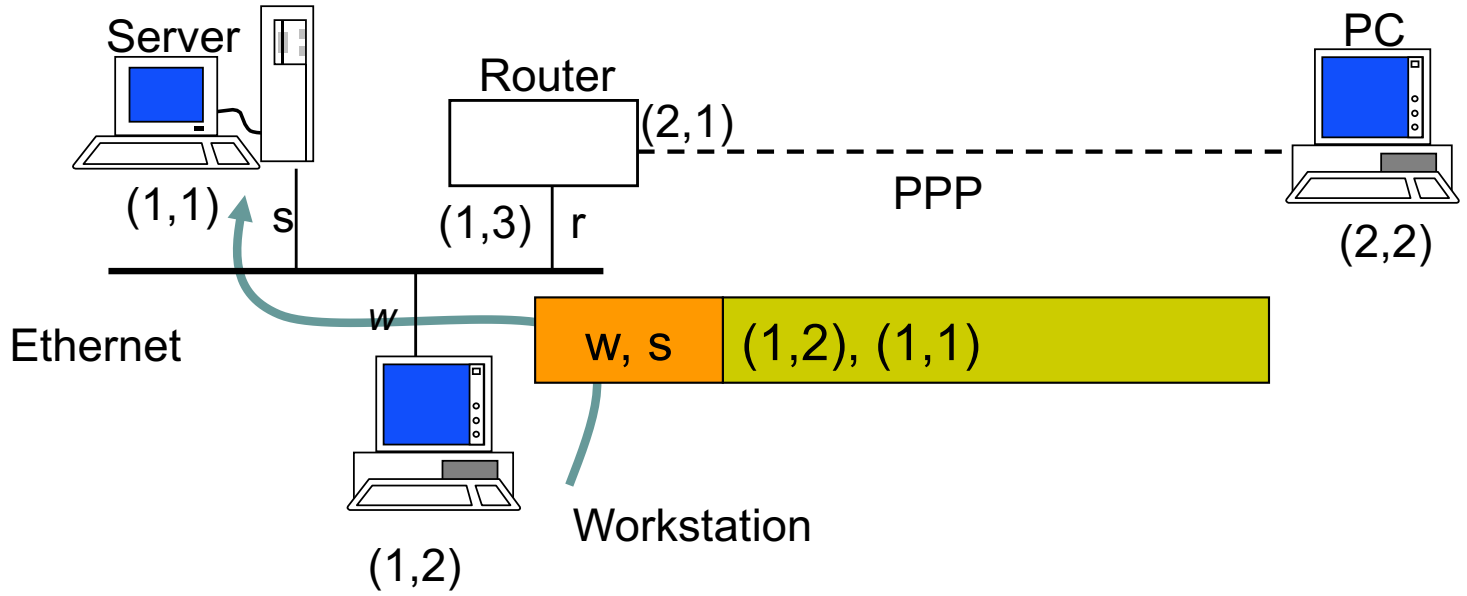
Encapsulation





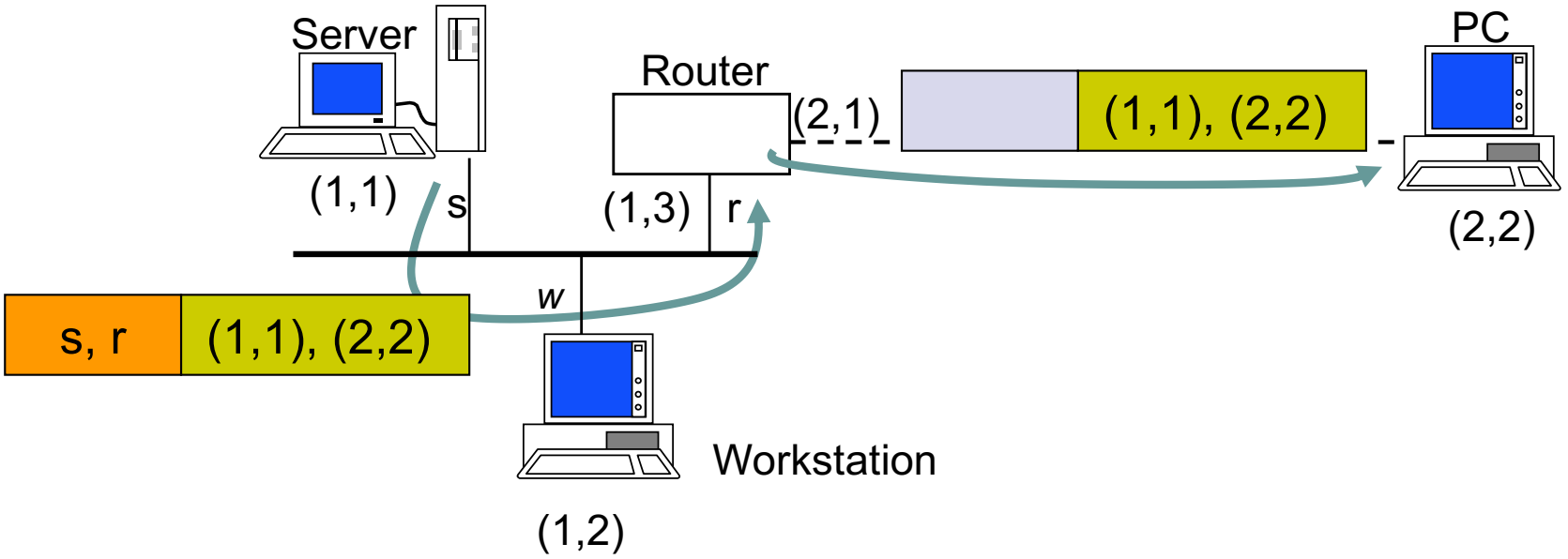
- Ethernet header contains:
  - source and destination physical addresses
  - network protocol type (e.g. IP)

IP packet from workstation to server



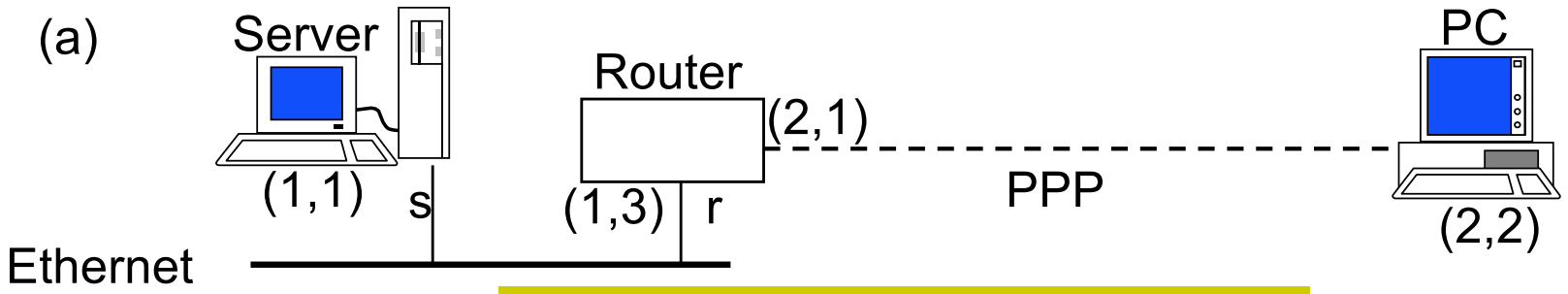
1. IP packet has (1,2) IP address for source and (1,1) IP address for destination
2. IP table at workstation indicates (1,1) connected to same network, so IP packet is encapsulated in Ethernet frame with addresses w and s
3. Ethernet frame is broadcast by workstation NIC and captured by server NIC
4.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

IP packet from server to PC



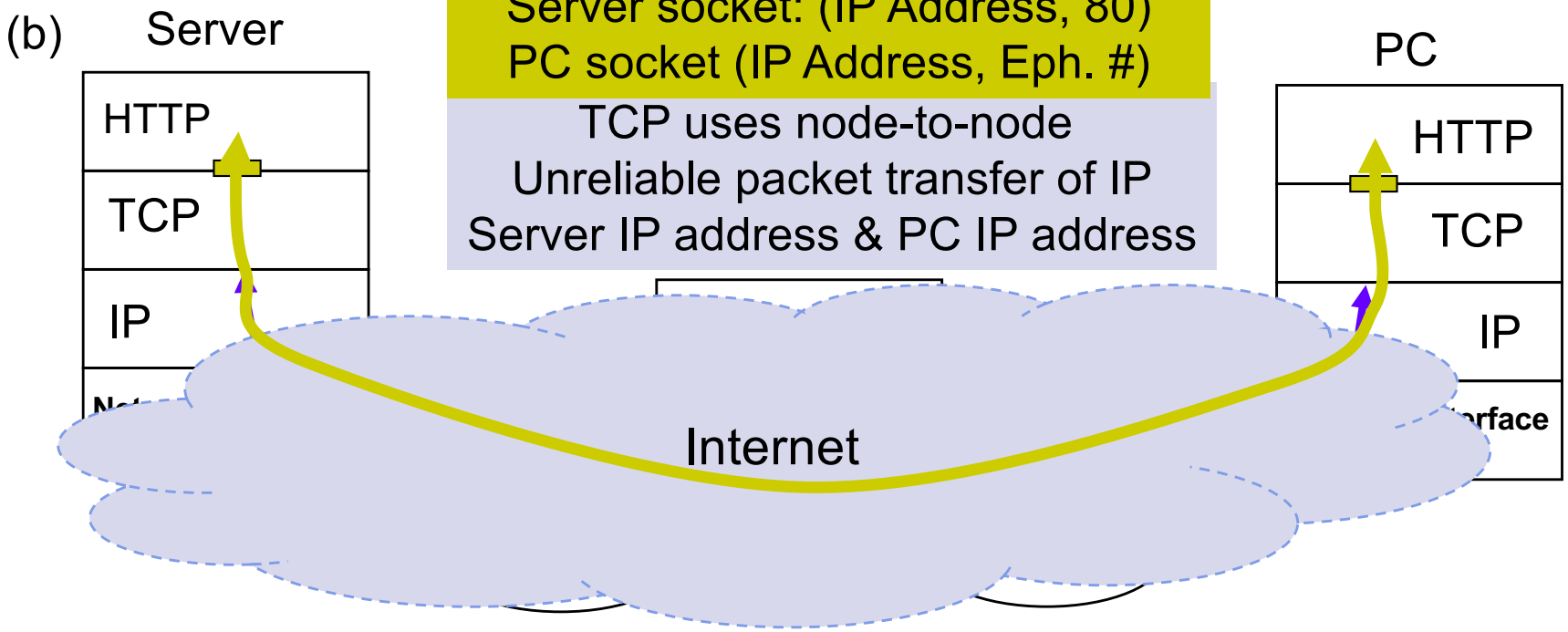
1. IP packet has (1,1) and (2,2) as IP source and destination addresses
2. IP table at server indicates packet should be sent to router, so IP packet is encapsulated in Ethernet frame with addresses s and r
3. Ethernet frame is broadcast by server NIC and captured by router NIC
4. NIC examines protocol 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



HTTP uses process-to-process
Reliable byte stream transfer of
TCP connection:
Server socket: (IP Address, 80)
PC socket (IP Address, Eph. #)

TCP uses node-to-node
Unreliable packet transfer of IP
Server IP address & PC IP address

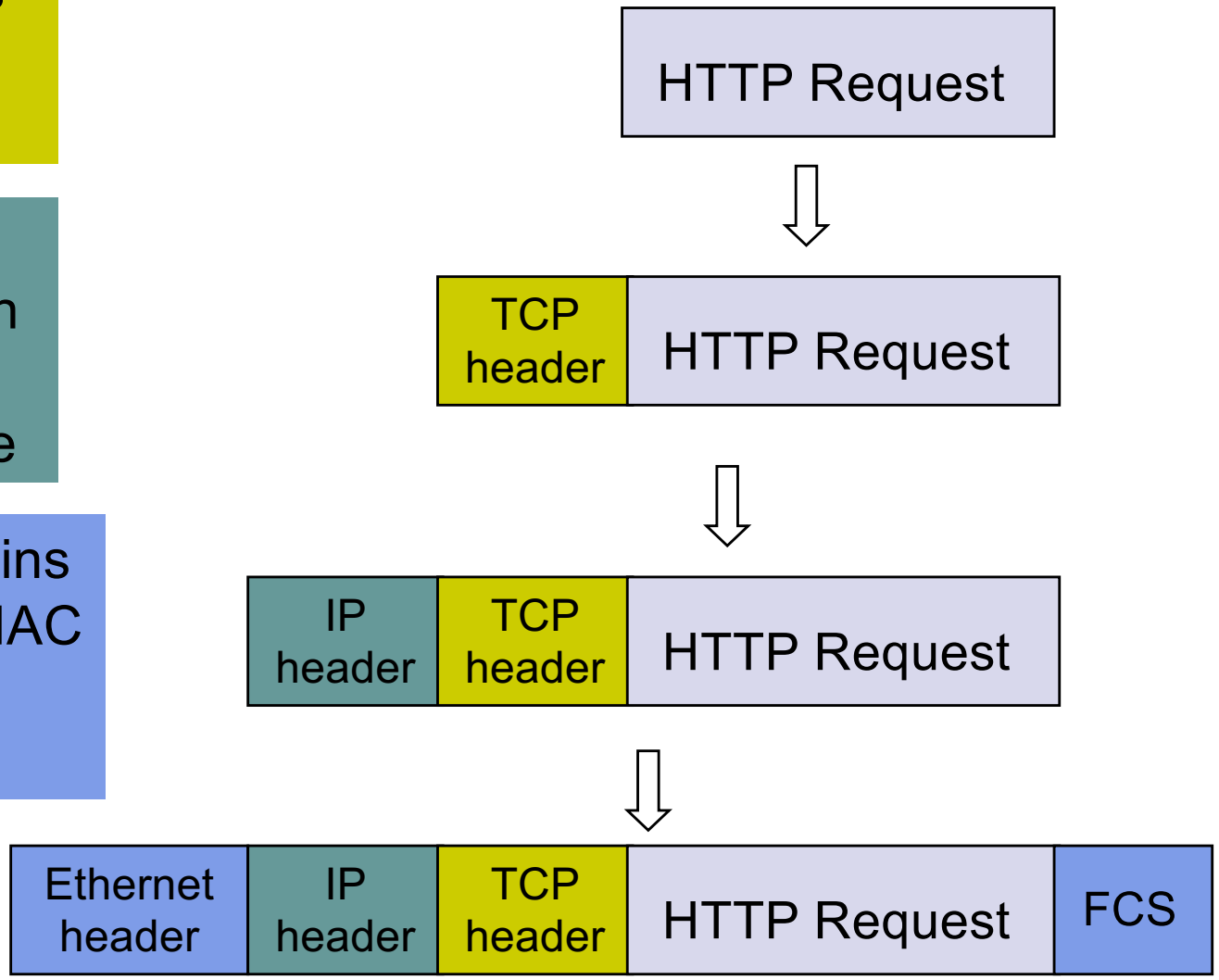


Encapsulation

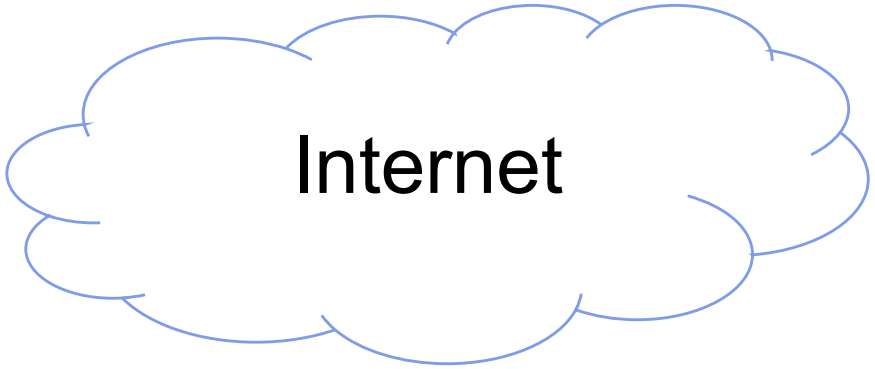
TCP Header contains source & destination port numbers

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

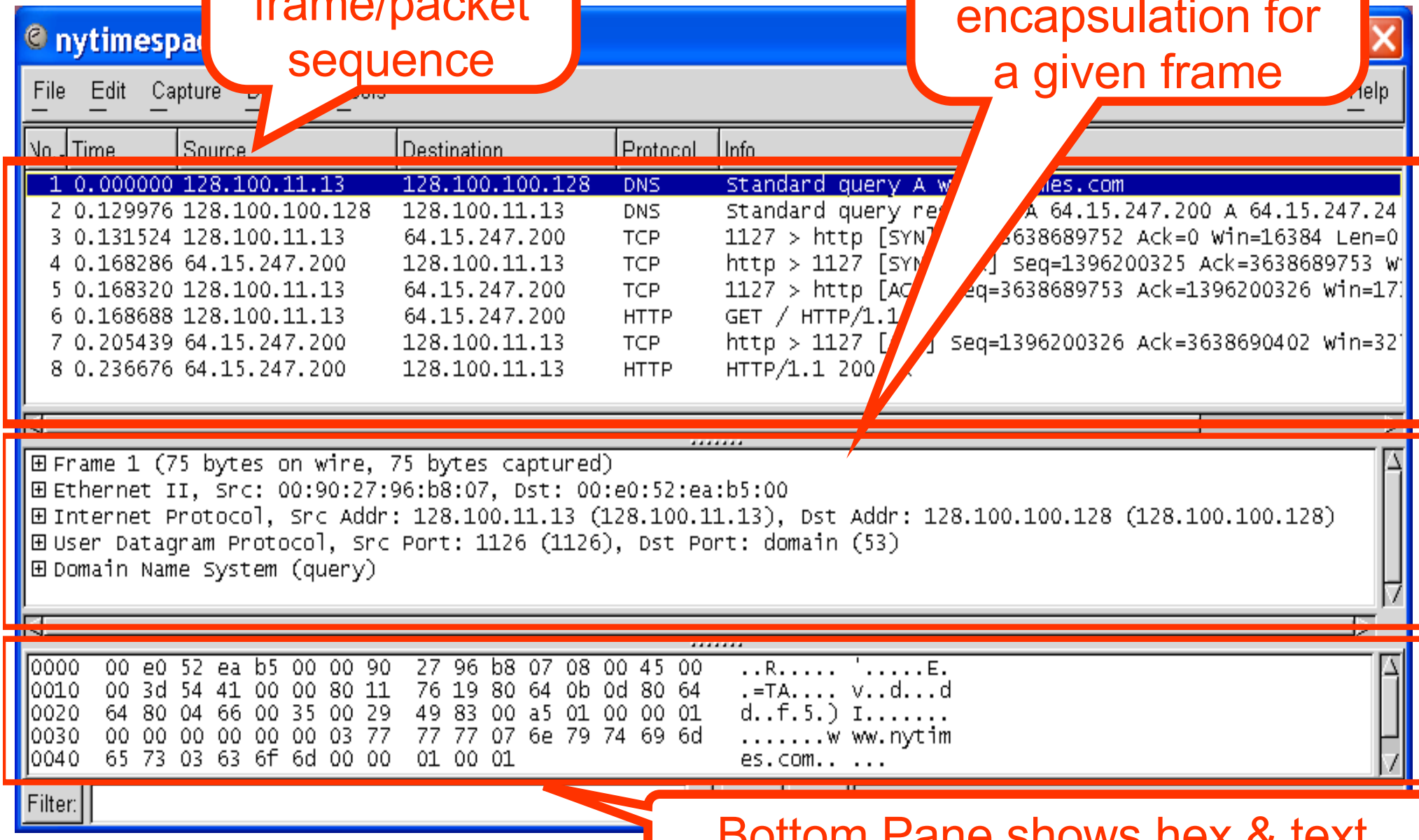


- User clicks on http://www.nytimes.com/
- *Ethereal* network analyzer captures all frames observed by its Ethernet NIC
- Sequence of frames and contents of frame can be examined in detail down to individual bytes

Eth lows

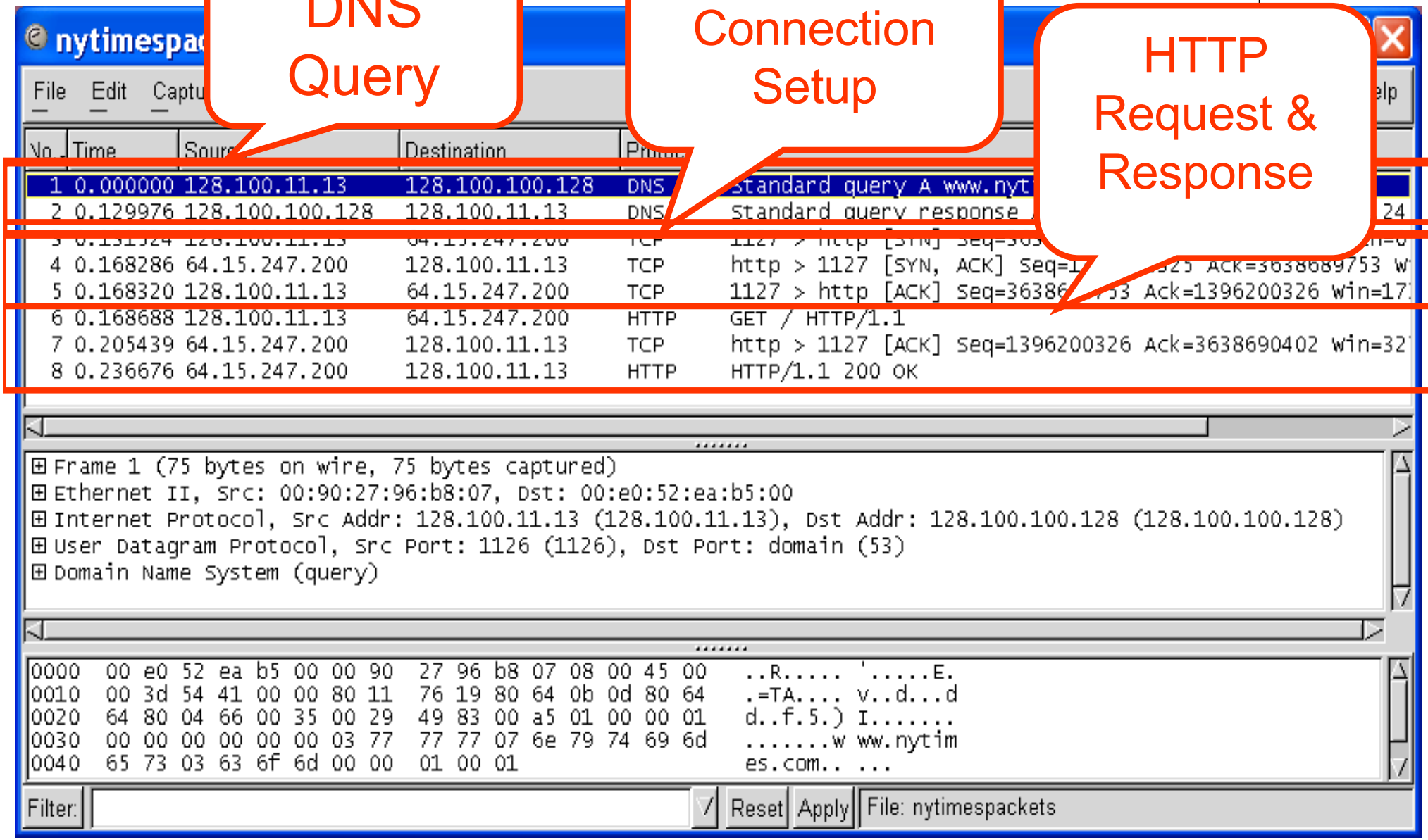
Top Pane shows frame/packet sequence

Middle Pane shows encapsulation for a given frame

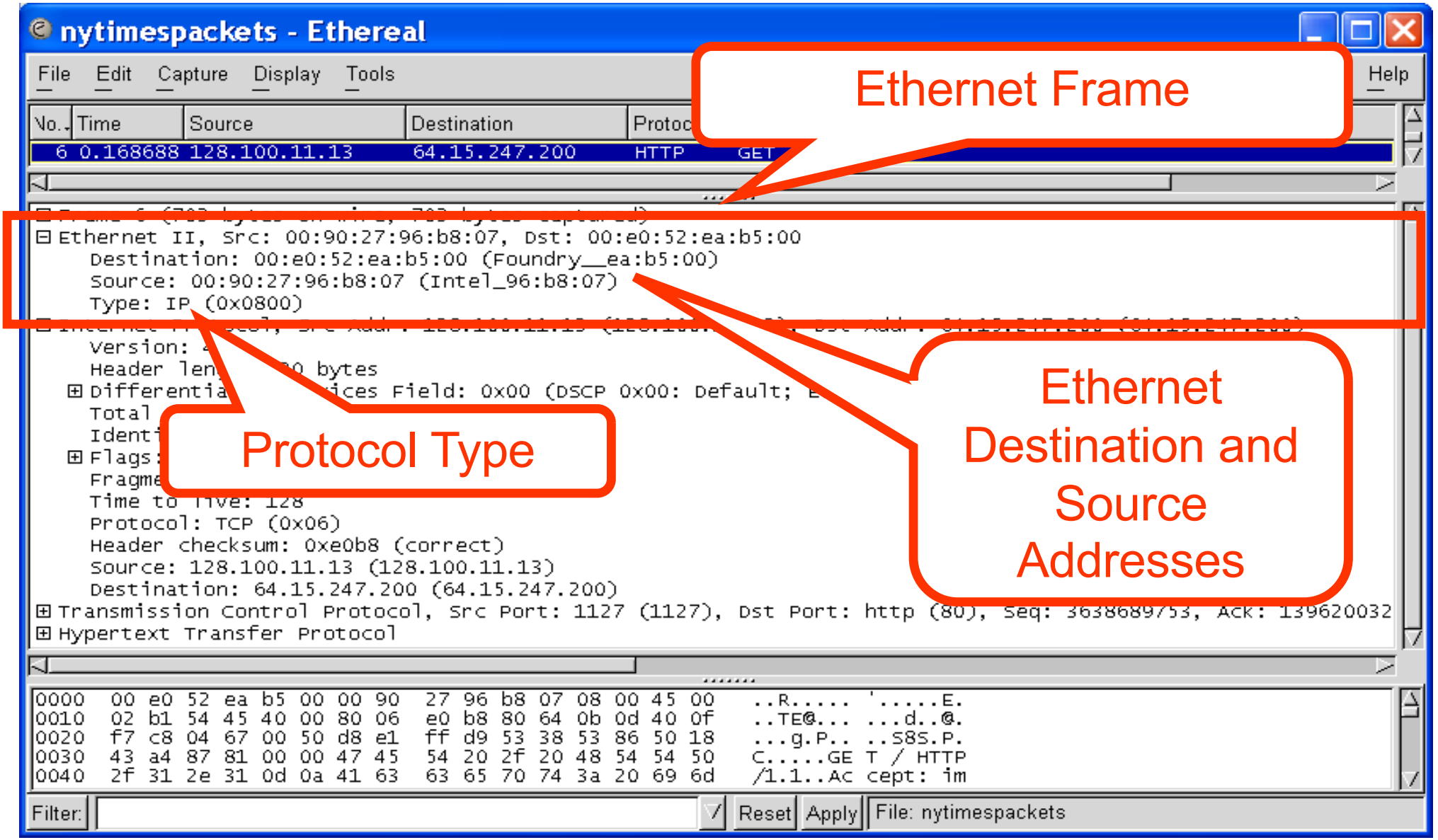


Bottom Pane shows hex & text

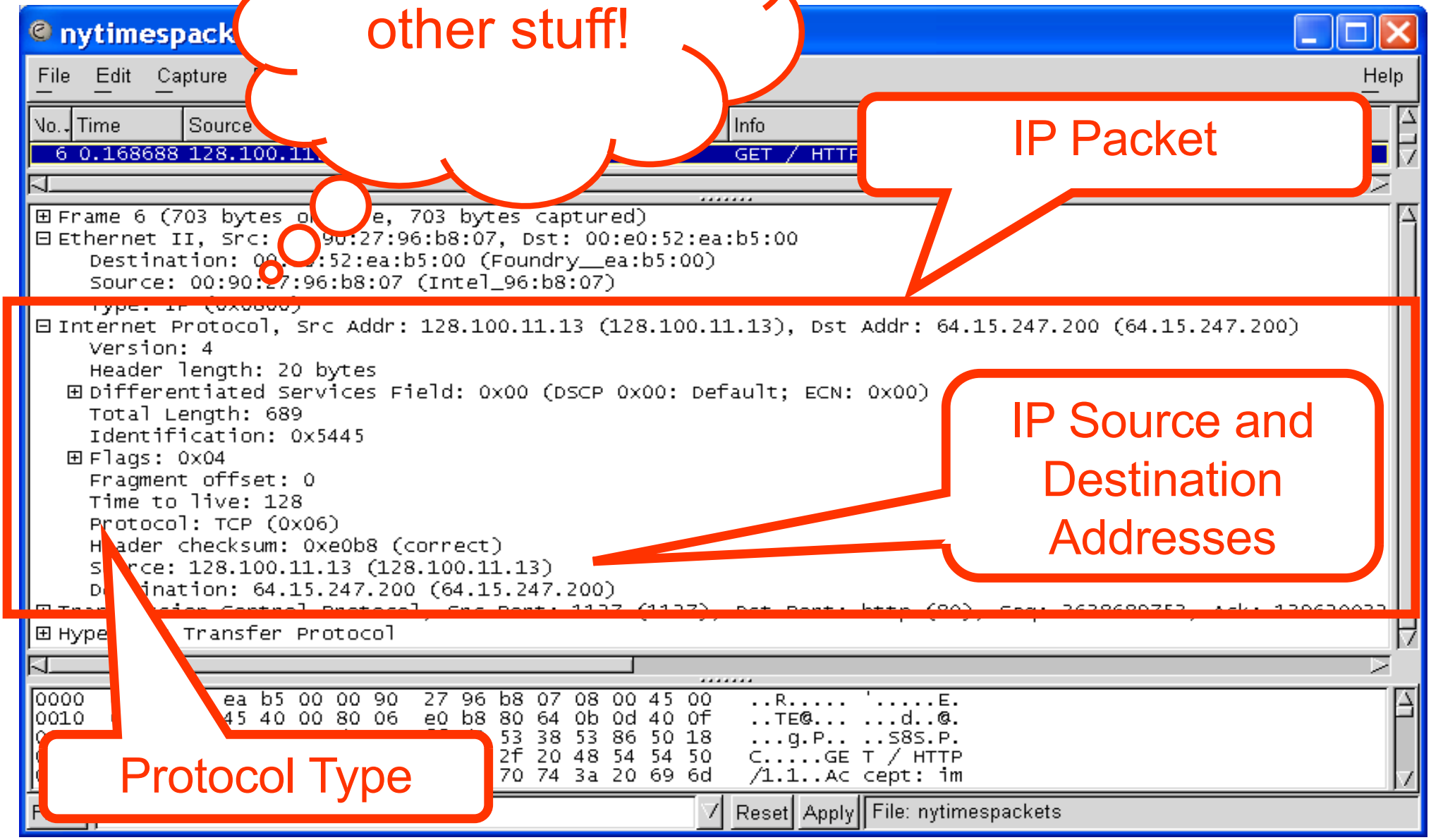
Top pane: frame sequence



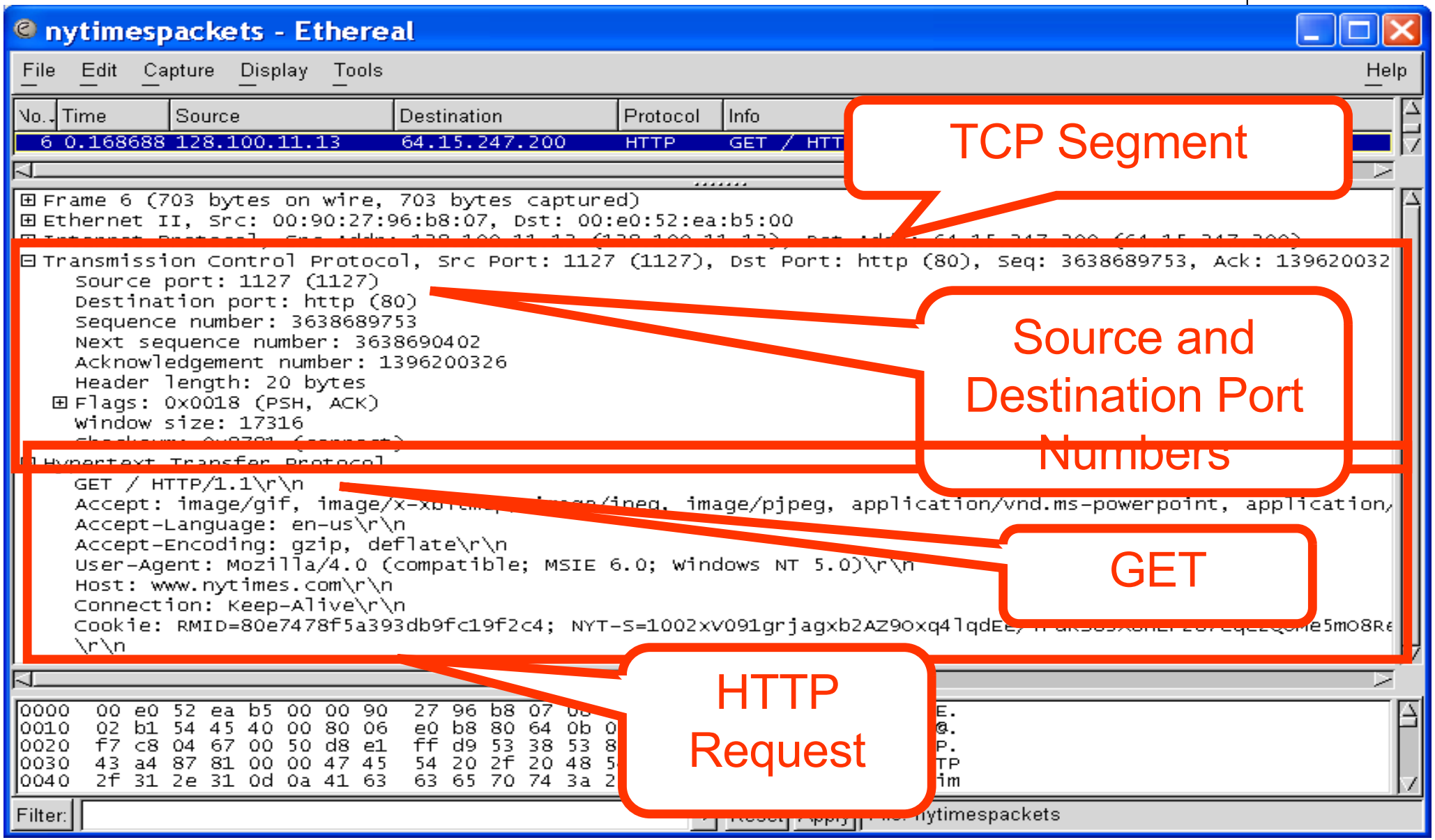
Middle pane: Encapsulation



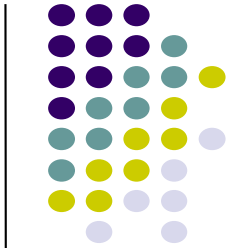
Middle ... capsulation



Middle pane: Encapsulation

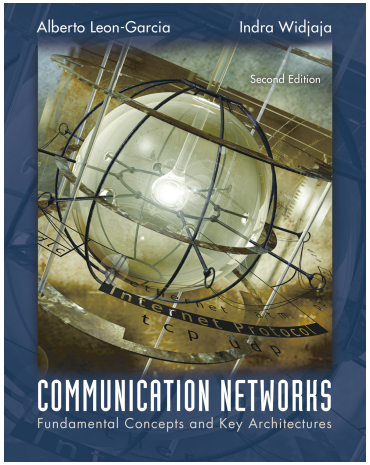


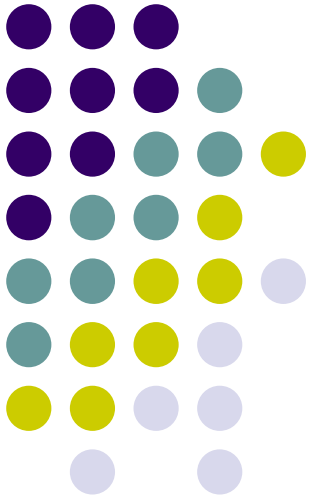
Summary



- 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



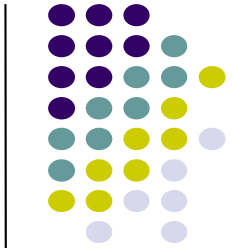


Application Layer Protocols

HTTP

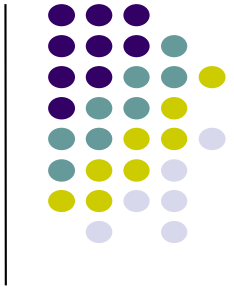
RTP & SIP

HTTP Protocol



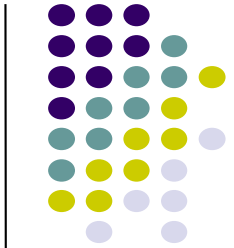
- HTTP servers use well-known port 80
- Client request / Server reply
- Stateless: server does not keep any information about client
- HTTP 1.0 new TCP connection per request/reply (non-persistent)
- HTTP 1.1 persistent operation is default

HTTP Message Formats



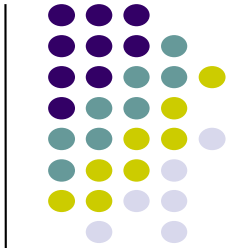
- HTTP messages written in ASCII text
- Request Message Format
  - *Request Line (Each line ends with carriage return)*
    - *Method URL HTTP-Version \r\n*
    - Method specifies action to apply to object
    - URL specifies object
  - *Header Lines (Ea. line ends with carriage return)*
    - *Attribute 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



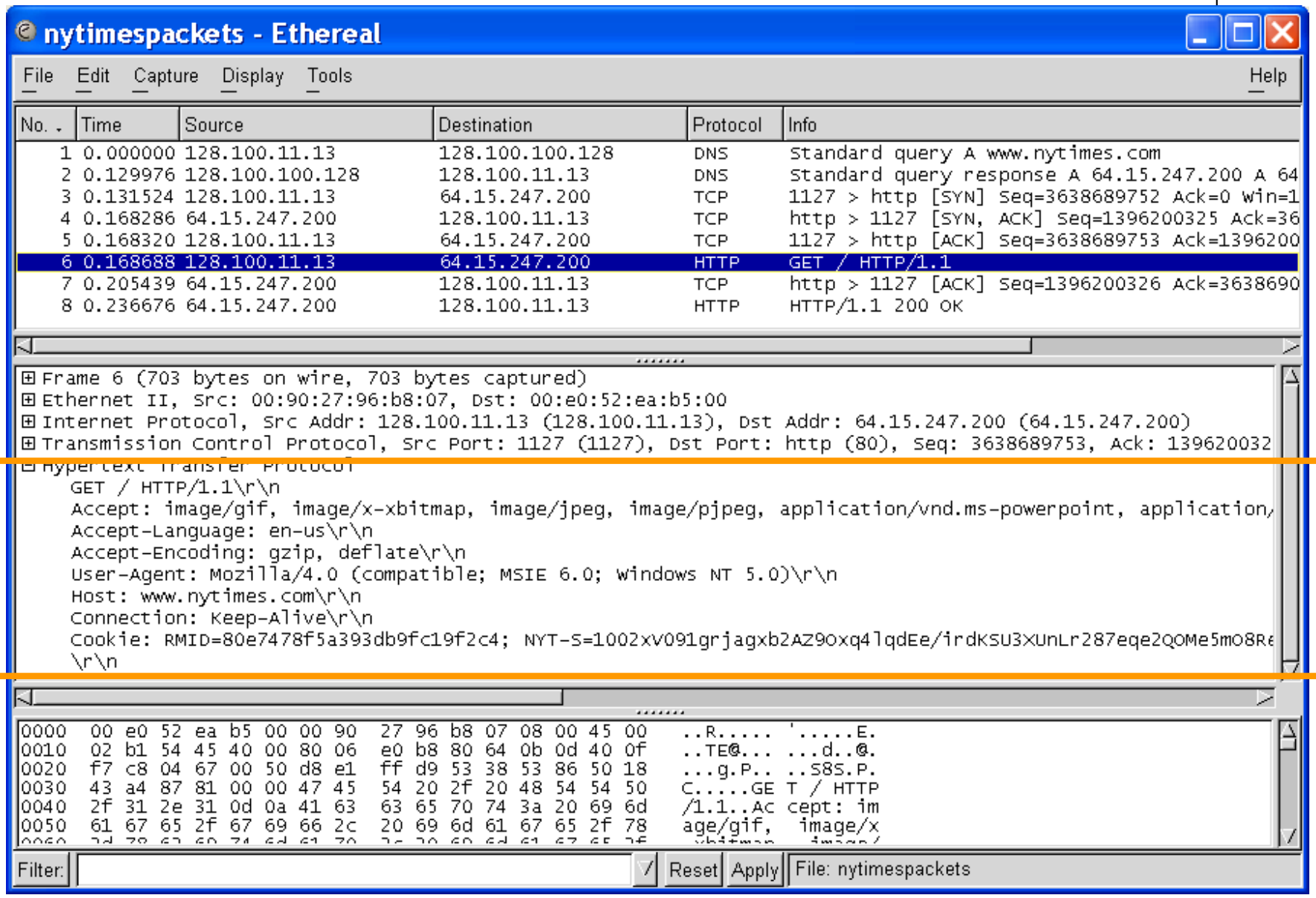
Request method	Meaning
GET	Retrieve information (object) identified by the URL.
HEAD	Retrieve meta-information about the object, but do not transfer the object; Can be used to find out if a document has changed.
POST	Send information to a URL (using the entity body) and retrieve result; used when a user 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

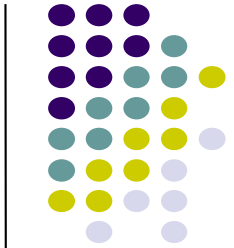


- Absolute URL
  - scheme://hostname[:port]/path
  - http://www.nytimes.com/
- Relative URL
  - /path
  - /

HTTP Request Message

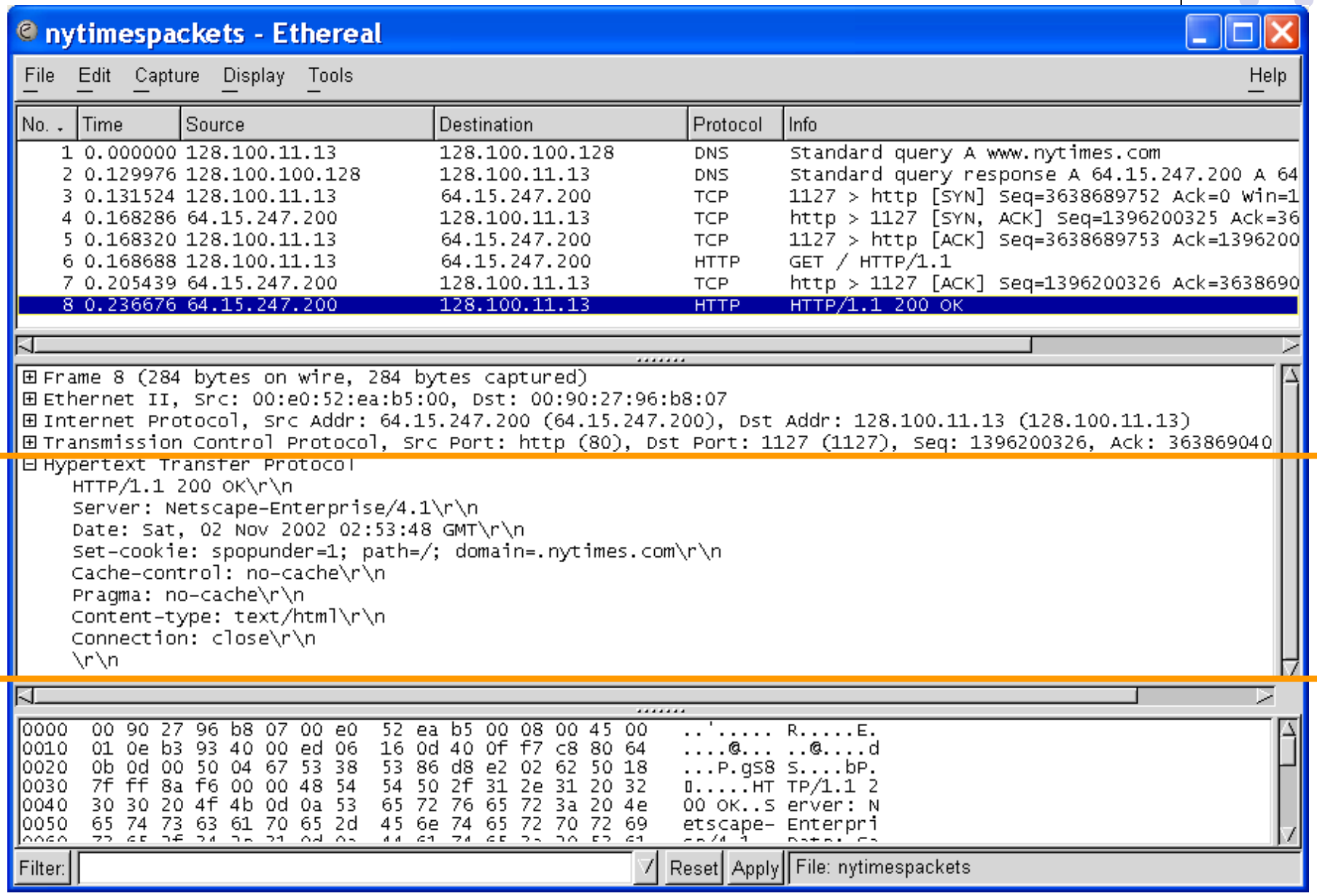


HTTP Response Message

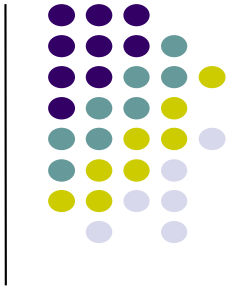


- Response Message Format
  - *Status Line*
    - *HTTP-Version Status-Code Message*
    - Status Code: 3-digit code indicating result
    - E.g. HTTP/1.0 200 OK
  - *Headers Section*
    - Information about object transferred to client
    - E.g. server type, content length, content type, …
  - *Content*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

HTTP Response Message

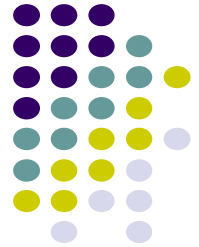


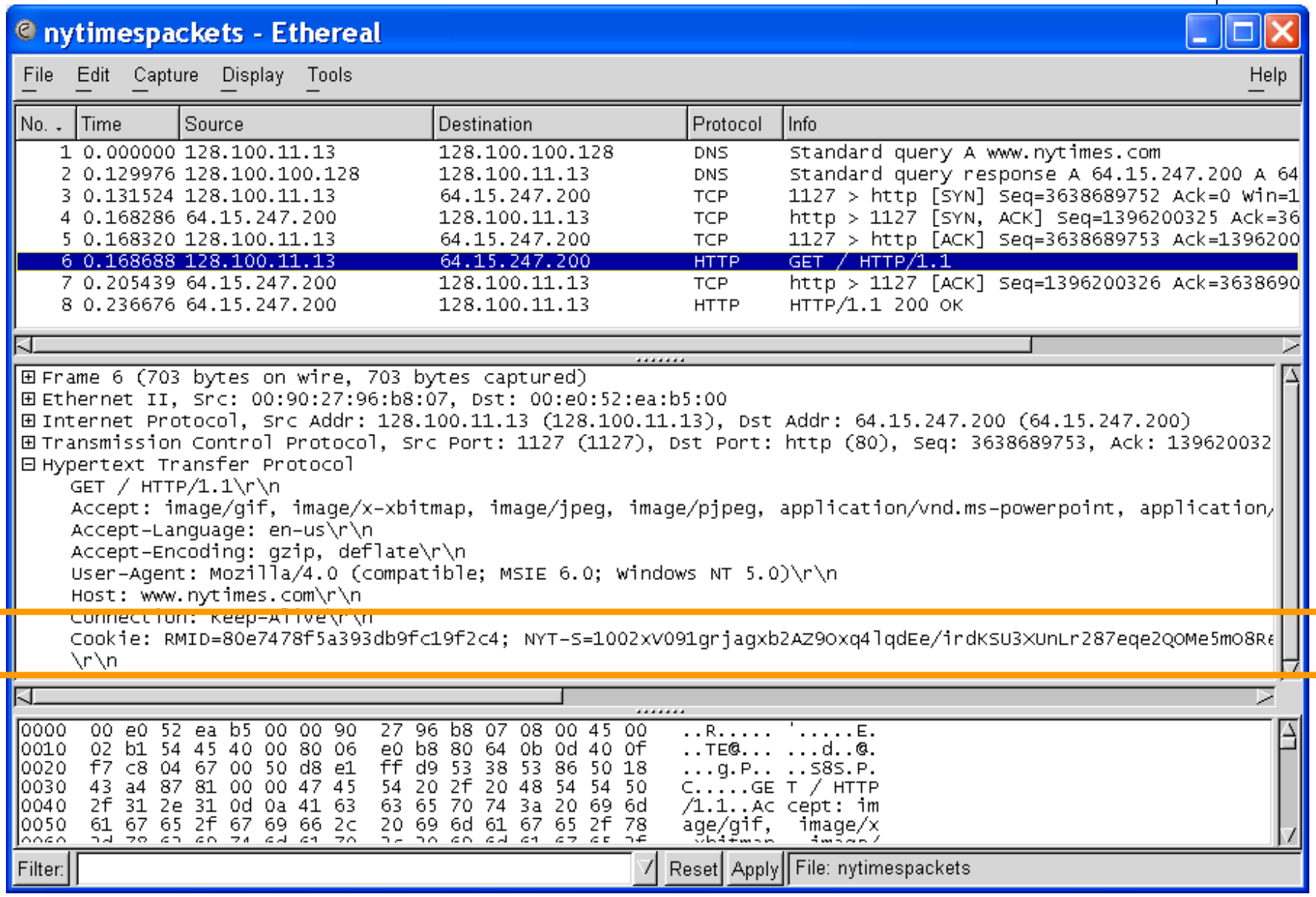
Cookies and Web Sessions



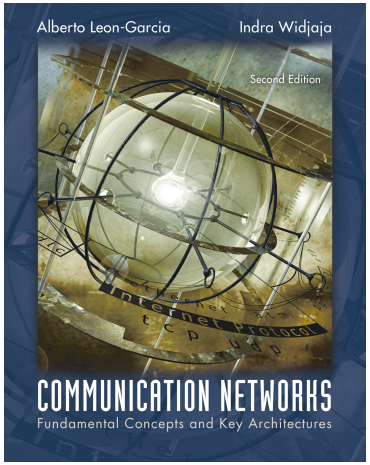
- Cookies are data exchanged by clients & servers as header lines
- Since HTTP stateless, cookies can provide context for HTTP interaction
- *Set cookie* header line in reply message from server + unique ID number for client
- If client accepts cookie, cookie added to client's cookie file (must include 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

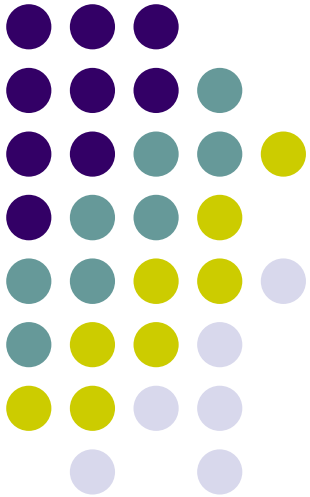
Cookie Header Line;
ID is 24 hexadecimal numeral





Voice Over IP





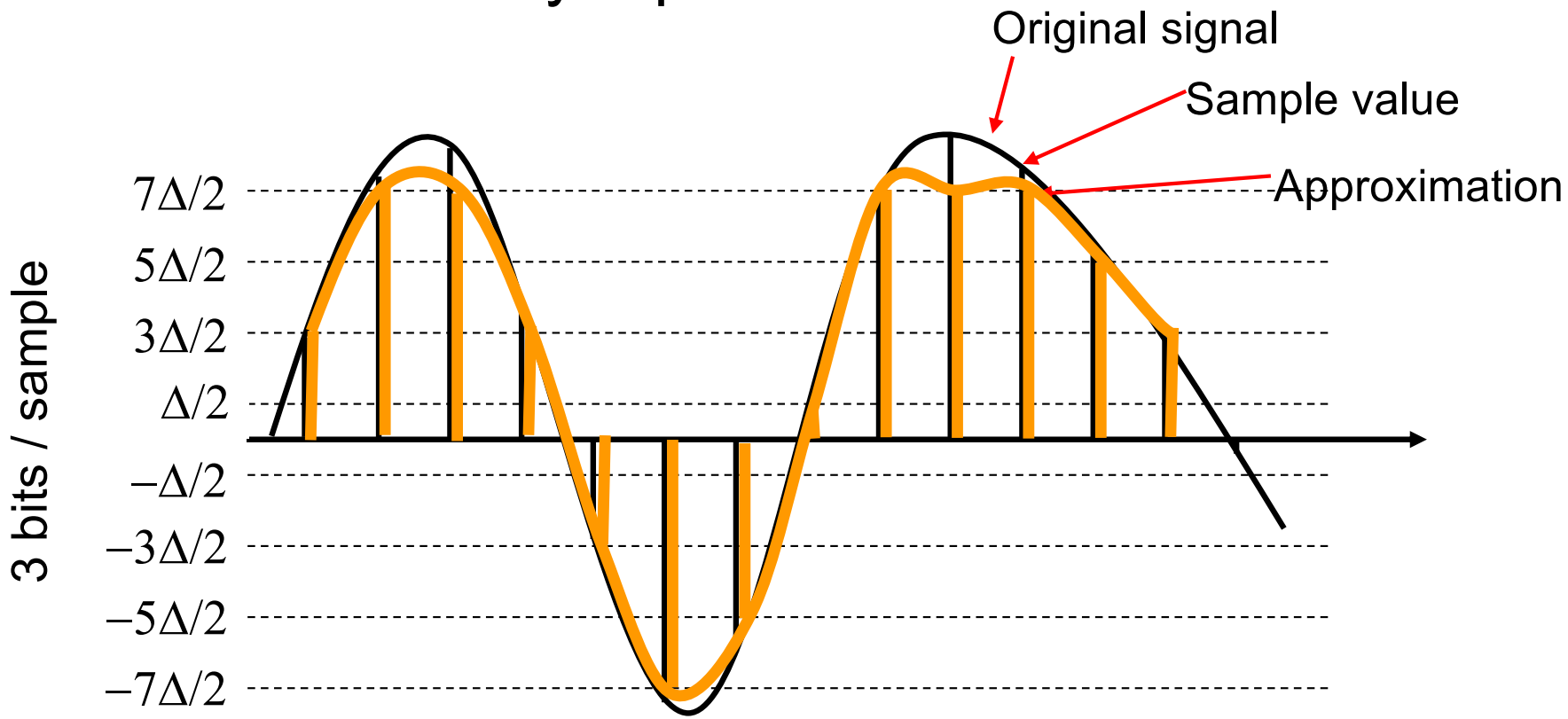
Digitization

Timing Recovery

RTP over UDP

Digitization of Analog Signal

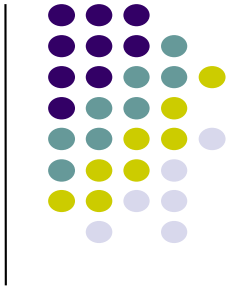
- Sample analog signal in time and amplitude
- Find closest binary representation



R_s = Bit rate = # bits/sample x # samples/second

= n bits/sample 2 W samples/sec

Example: Voice & Audio



Telephone voice

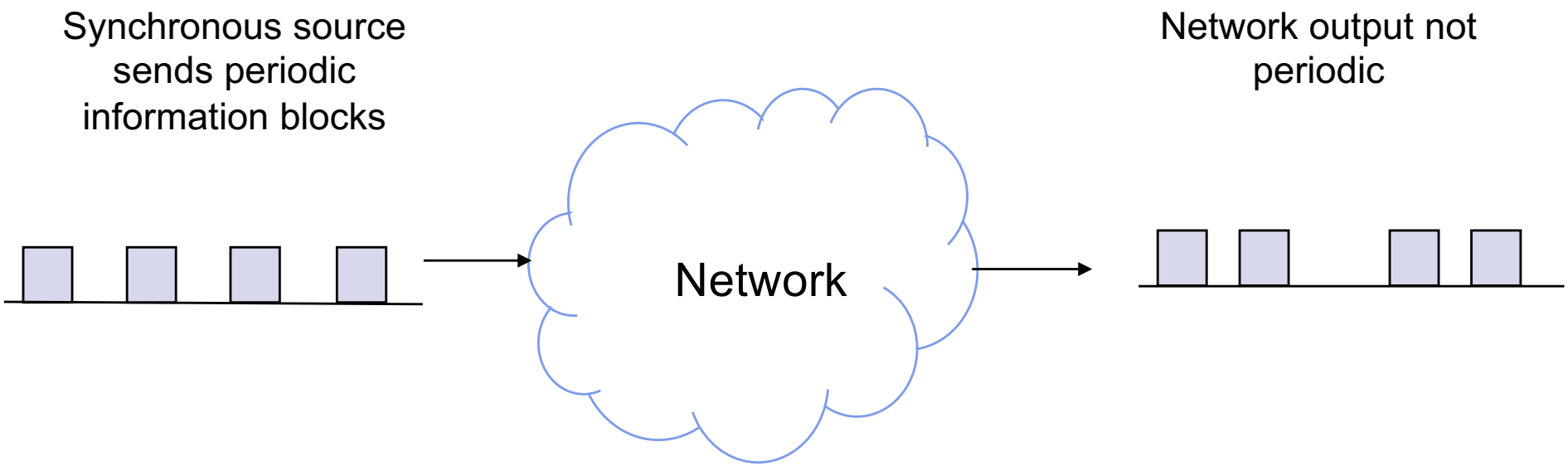
- W_s = 4 kHz → 8000 samples/sec
- 8 bits/sample
- R_s=8 x 8000 = 64 kbps

- Cellular phones use more powerful compression algorithms: 8-12 kbps

CD Audio

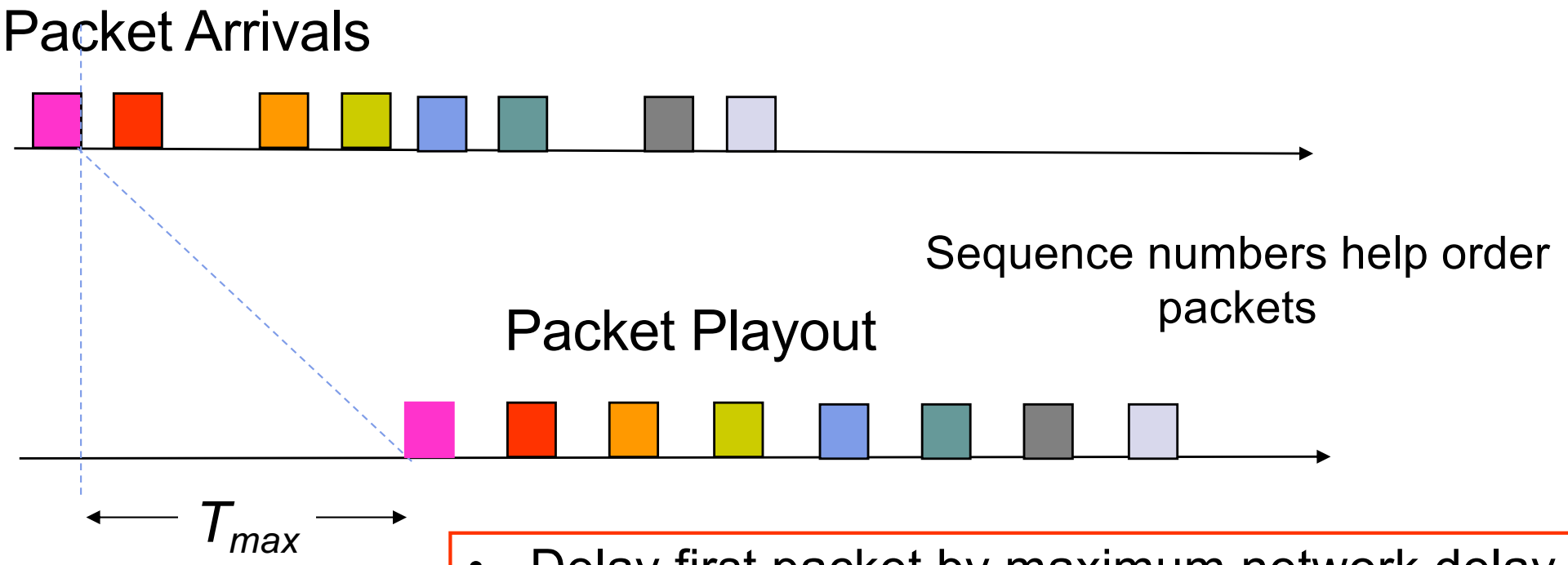
- W_s = 22 kHertz → 44000 samples/sec
- 16 bits/sample
- R_s=16 x 44000= 704 kbps per audio channel
- MP3 uses more powerful compression algorithms: 50 kbps per audio channel

Timing Recovery for Synchronous Services



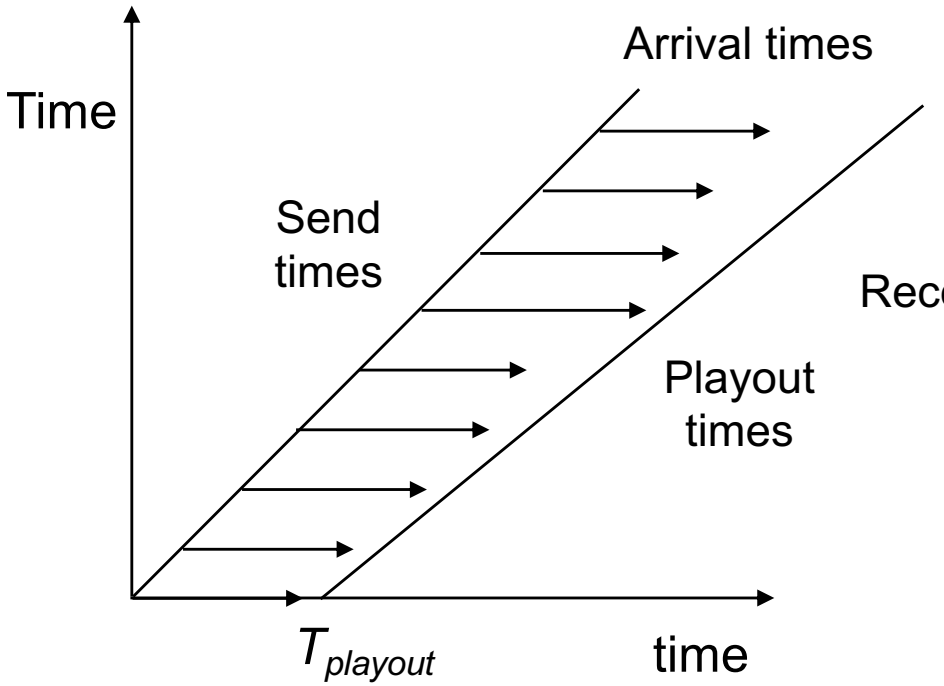
- Voice, audio, or video generate a synchronous information stream
- Information carried by equally-spaced fixed-length packets
- Packet switching introduces random delays
- Timing recovery is needed to re-establish the synchronous nature of the stream

Introduce Playout Buffer

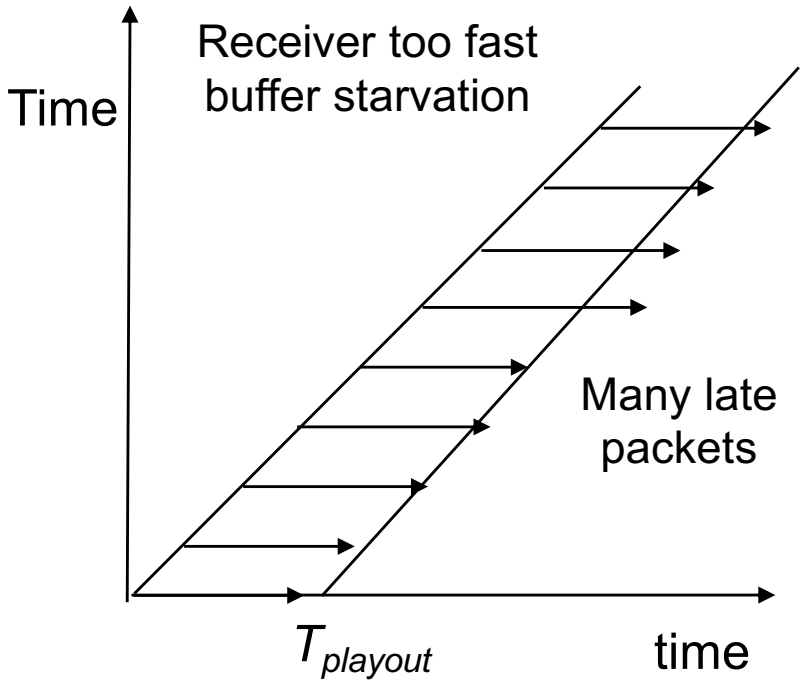


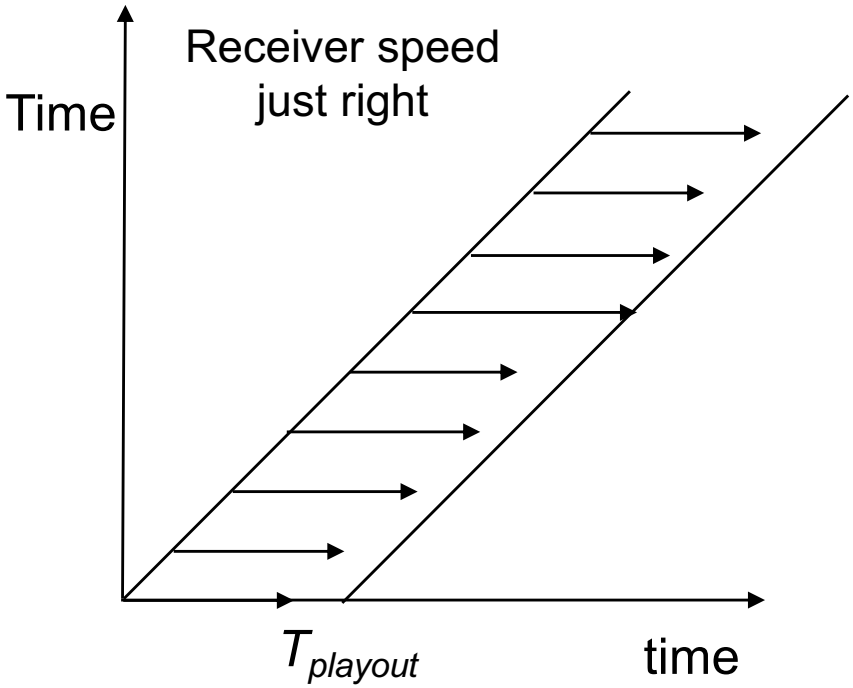
- Delay first packet by maximum network delay
- All other packets arrive with less delay
- Playout packet uniformly thereafter

Playout clock must be synchronized to transmitter clock



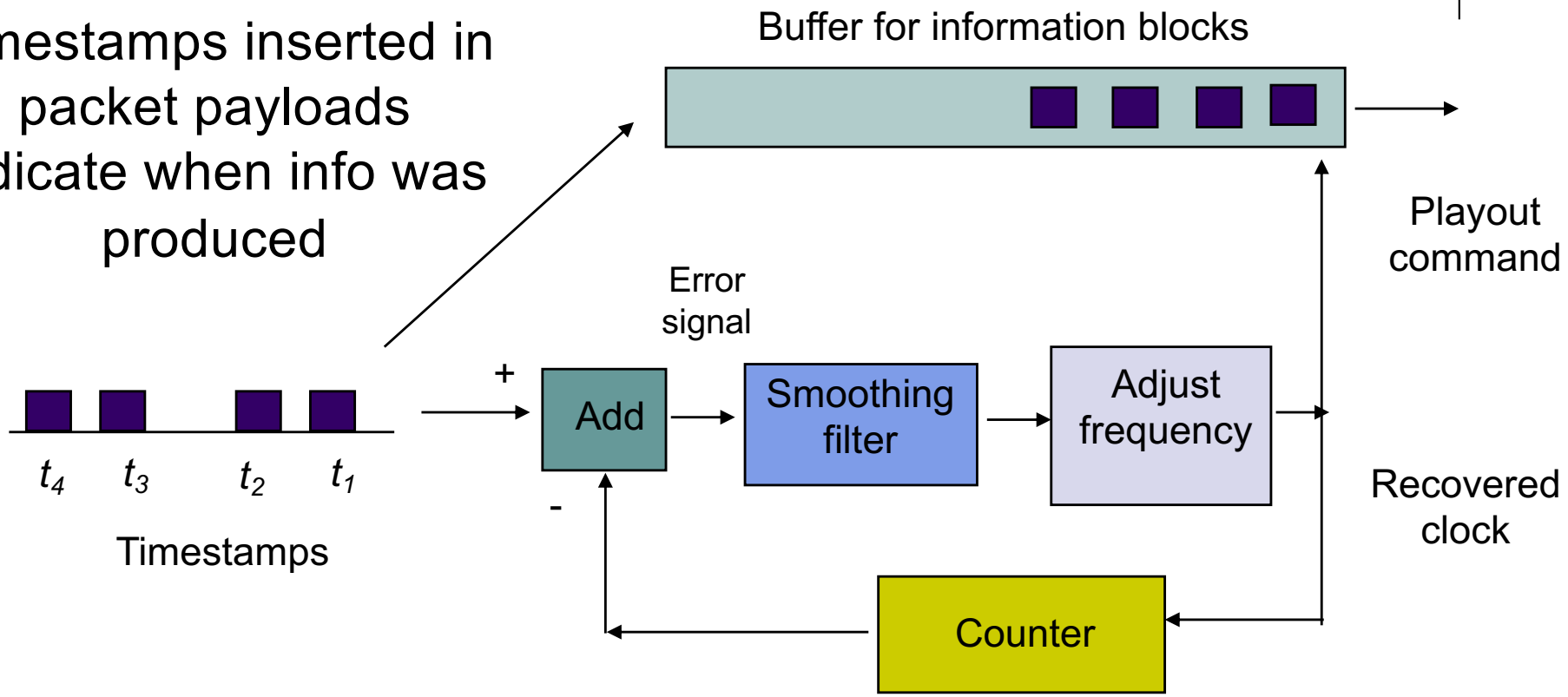
Receiver too slow; buffer fills and overflows





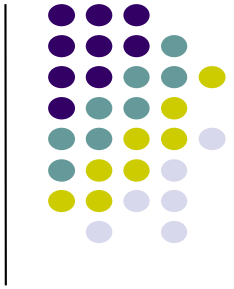
Clock Recovery

Timestamps inserted in packet payloads indicate when info was produced



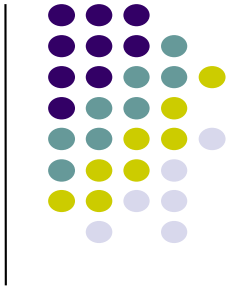
- Counter attempts to replicate transmitter clock
- Frequency of counter is adjusted according to arriving timestamps
- Jitter introduced by network causes fluctuations in buffer & in local clock

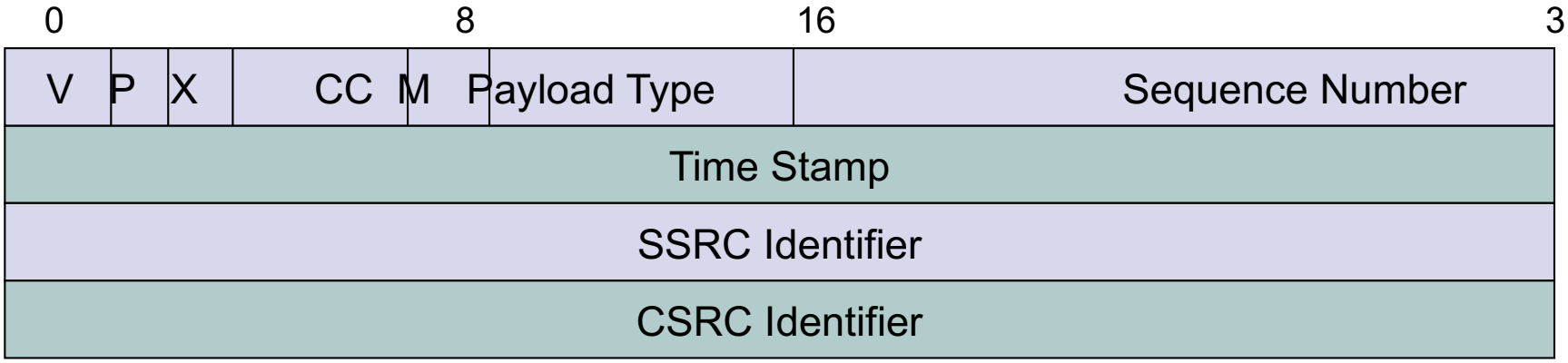
Real-Time Protocol



- RTP designed to support real-time applications such as voice, audio, video
- RTP provides means to carry:
  - Type of information source
  - Sequence numbers
  - Timestamps
- Timing recovery done by higher layer protocol
  - MPEG for video, MP3 for audio
- RTP runs over UDP

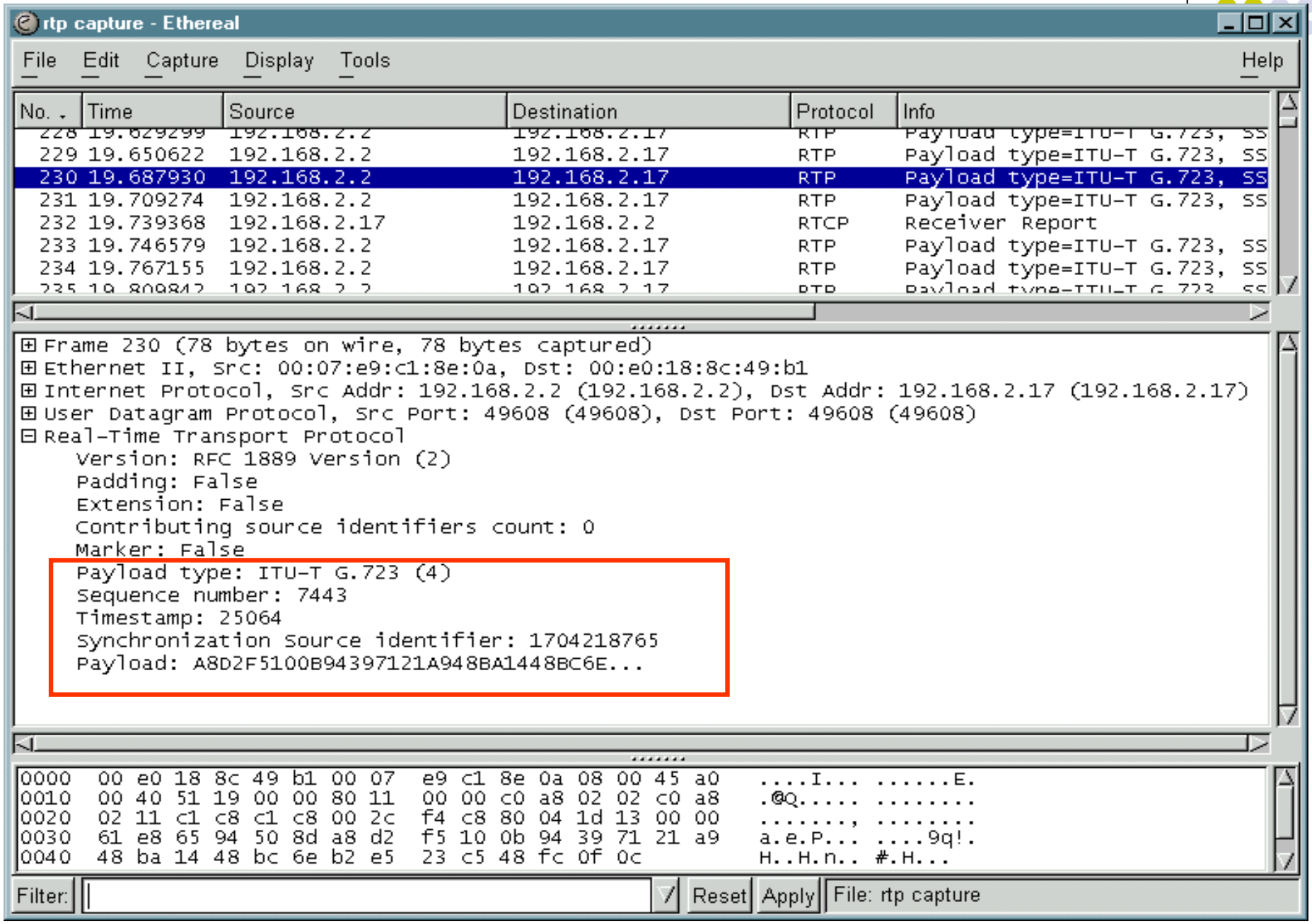
RTP Packet Format



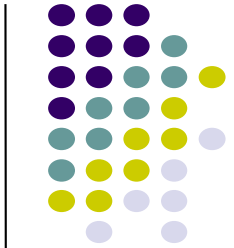


- Payload Type: e.g. PCM, MPEG2, …
- Sequence Number: detect packet loss
- Timestamp: sampling instant of first byte
- Synchronization Source: ID for synch source
- CSRC List: contributing sources to payload

RTP Packet

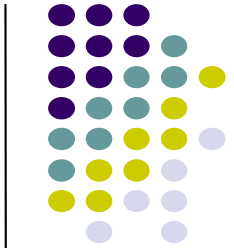

rtp capture - Ethereal
File Edit Capture Display Tools Help
No. Time Source Destination Protocol Info
229 19.650622 192.168.2.2 192.168.2.17 RTP Payload type=ITU-T G.723, SS
230 19.687930 192.168.2.2 192.168.2.17 RTP Payload type=ITU-T G.723, SS
231 19.709274 192.168.2.2 192.168.2.17 RTP Payload type=ITU-T G.723, SS
232 19.739368 192.168.2.17 192.168.2.2 RTCP Receiver Report
233 19.746579 192.168.2.2 192.168.2.17 RTP Payload type=ITU-T G.723, SS
234 19.767155 192.168.2.2 192.168.2.17 RTP Payload type=ITU-T G.723, SS
Frame 230 (78 bytes on wire, 78 bytes captured)
Ethernet II, Src: 00:07:e9:c1:8e:0a, Dst: 00:e0:18:8c:49:b1
Internet Protocol, Src Addr: 192.168.2.2 (192.168.2.2), Dst Addr: 192.168.2.17 (192.168.2.17)
User Datagram Protocol, Src Port: 49608 (49608), Dst Port: 49608 (49608)
Real-Time Transport Protocol
Version: RFC 1889 Version (2)
Padding: False
Extension: False
Contributing source identifiers count: 0
Marker: False
Payload type: ITU-T G.723 (4)
Sequence number: 7443
Timestamp: 25064
Synchronization Source identifier: 1704218765
Payload: A8D2F5100B94397121A948BA1448BC6E...
0000 00 e0 18 8c 49 b1 00 07 e9 c1 8e 0a 08 00 45 a0 ....I... ......E.
0010 00 40 51 19 00 00 80 11 00 00 c0 a8 02 02 c0 a8 .@Q..... ....
0020 02 11 c1 c8 c1 c8 00 2c f4 c8 80 04 1d 13 00 00 ....
0030 61 e8 65 94 50 8d a8 d2 f5 10 0b 94 39 71 21 a9 a.e.P... ....9q!.
0040 48 ba 14 48 bc 6e b2 e5 23 c5 48 fc 0f 0c H..H.n.. #.H...
Filter: Reset Apply File: rtp capture

RTP Control Protocol (RTCP)

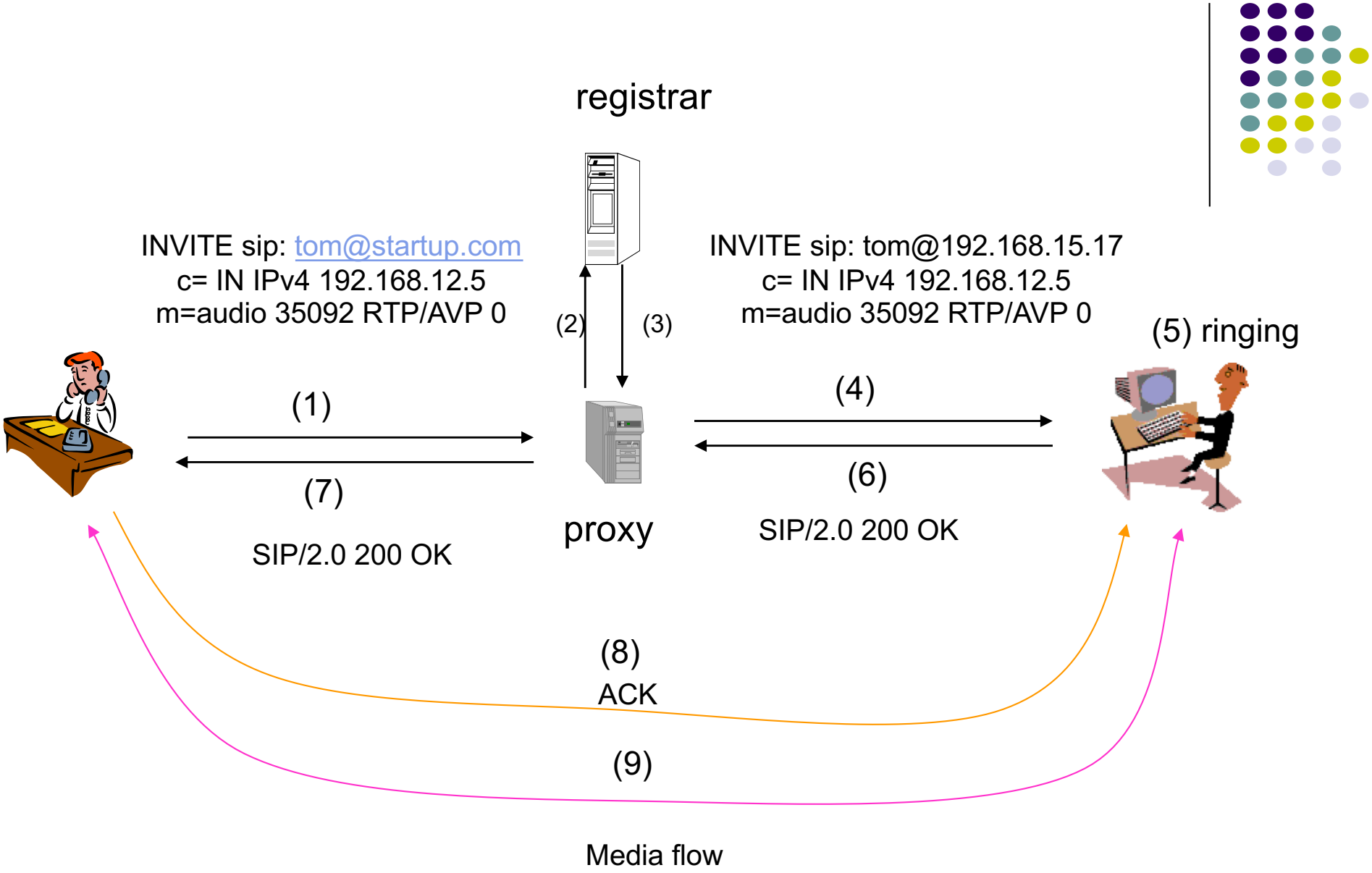


- RTP companion protocol
- Monitors quality of service at receivers
- Conveys monitored info to senders
- Canonical Name CNAME for each participant
- RTCP Packets
  - Sender Report Packet
  - Receiver Report Packet
  - Source Description (SDES)
  - BYE: end of participation by sender
  - APP: application specific functions

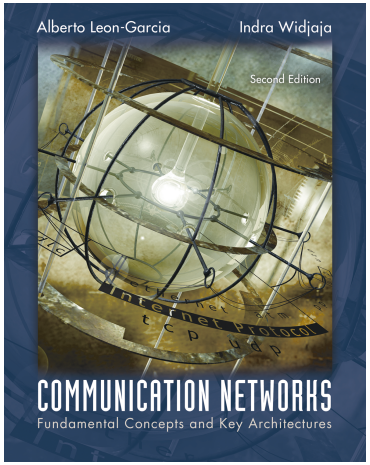
Session Initiation Protocol



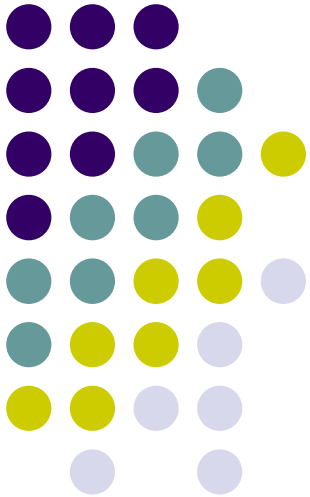
- Session: association involving exchange of data between Internet end systems
  - Internet telephone call; multimedia videoconference; instant messaging; event notification
- Session Initiation Protocol
  - Setting up, maintaining, terminating session
  - People & media devices
  - Multicast or mesh of unicast connections
  - Support for user mobility
  - Over UDP or TCP


registrar
INVITE sip: tom@startup.com
c= IN IPv4 192.168.12.5
m=audio 35092 RTP/AVP 0
(2)
(3)
INVITE sip: tom@192.168.15.17
c= IN IPv4 192.168.12.5
m=audio 35092 RTP/AVP 0
(5) ringing
(1)
(4)
(7)
(6)
proxy
SIP/2.0 200 OK
SIP/2.0 200 OK
(8)
ACK
(9)
Media flow

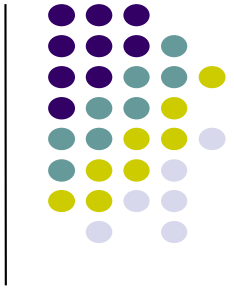
Chapter 2
Applications and Layered Architectures



Sockets

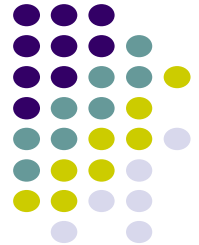


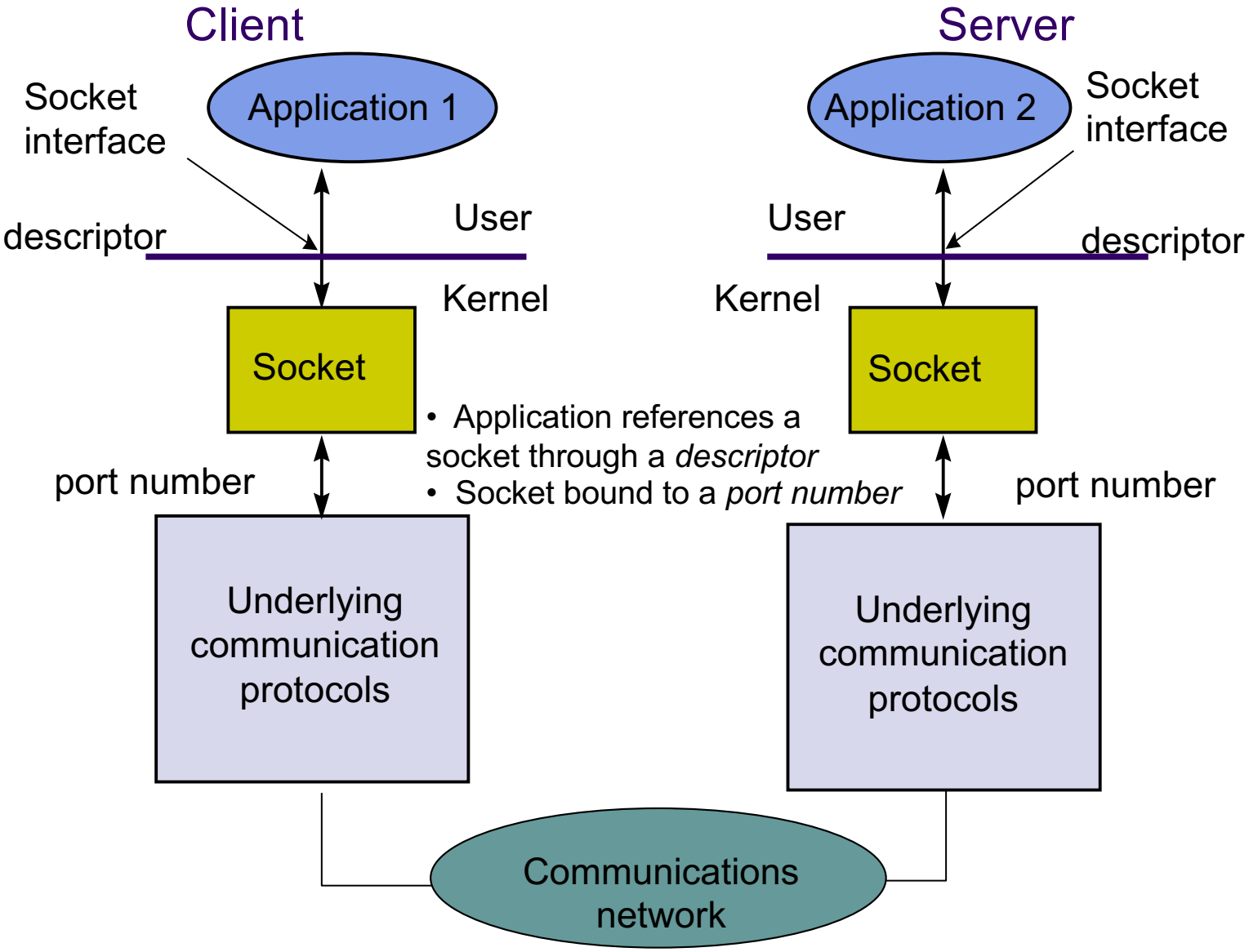
Socket API



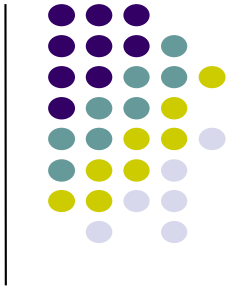
- API (Application Programming Interface)
  - Provides a standard set of functions that can be called by applications
- Berkeley UNIX Sockets API
  - Abstraction for applications to send & receive data
  - Applications create sockets that “plug into” network
  - Applications write/read to/from sockets
  - 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

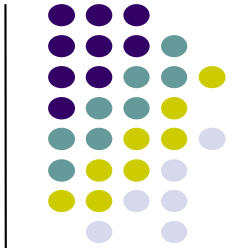


Connection-oriented

- 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

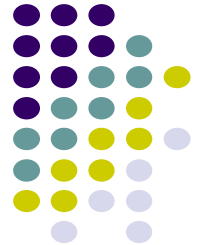
- Connectionless
- Immediate transfer of one block of information (boundaries preserved)
- No setup overhead & delay
- Destination address with each block
- Send/receive to/from multiple peer processes
- Best-effort service only
  - Possible out-of-order
  - Possible loss
- Uses UDP

Client & Server Differences



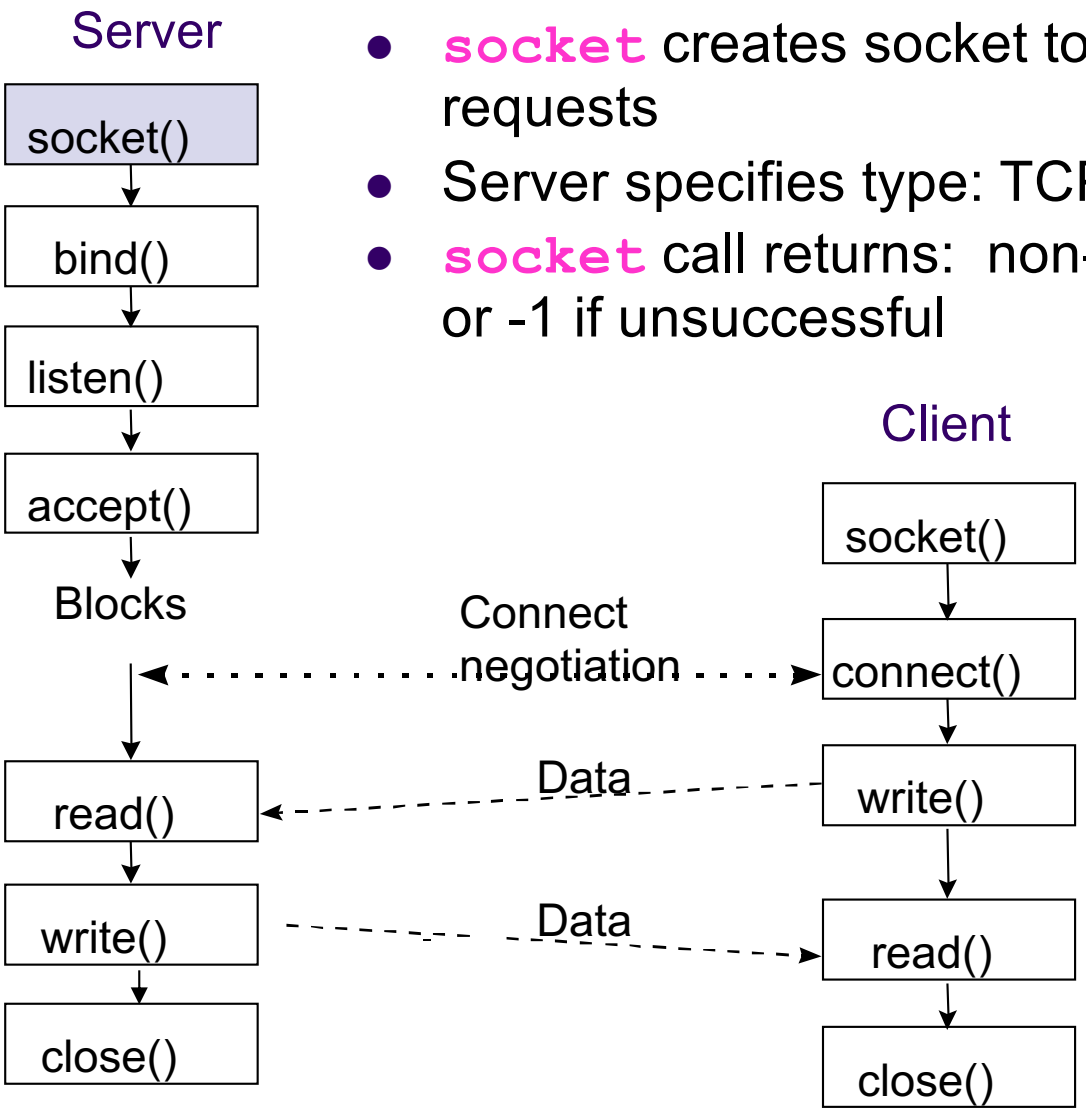
- Server
  - Specifies well-known port # when creating socket
  - May have multiple IP addresses (net interfaces)
  - Waits passively for client requests
- Client
  - Assigned ephemeral port #
  - Initiates communications with server
  - Needs to know server's IP address & port #
    - DNS for URL &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

Socket Calls for Connection-Oriented Mode

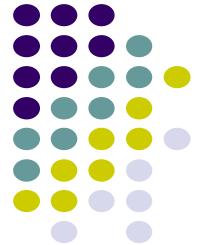


Server does Passive Open

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

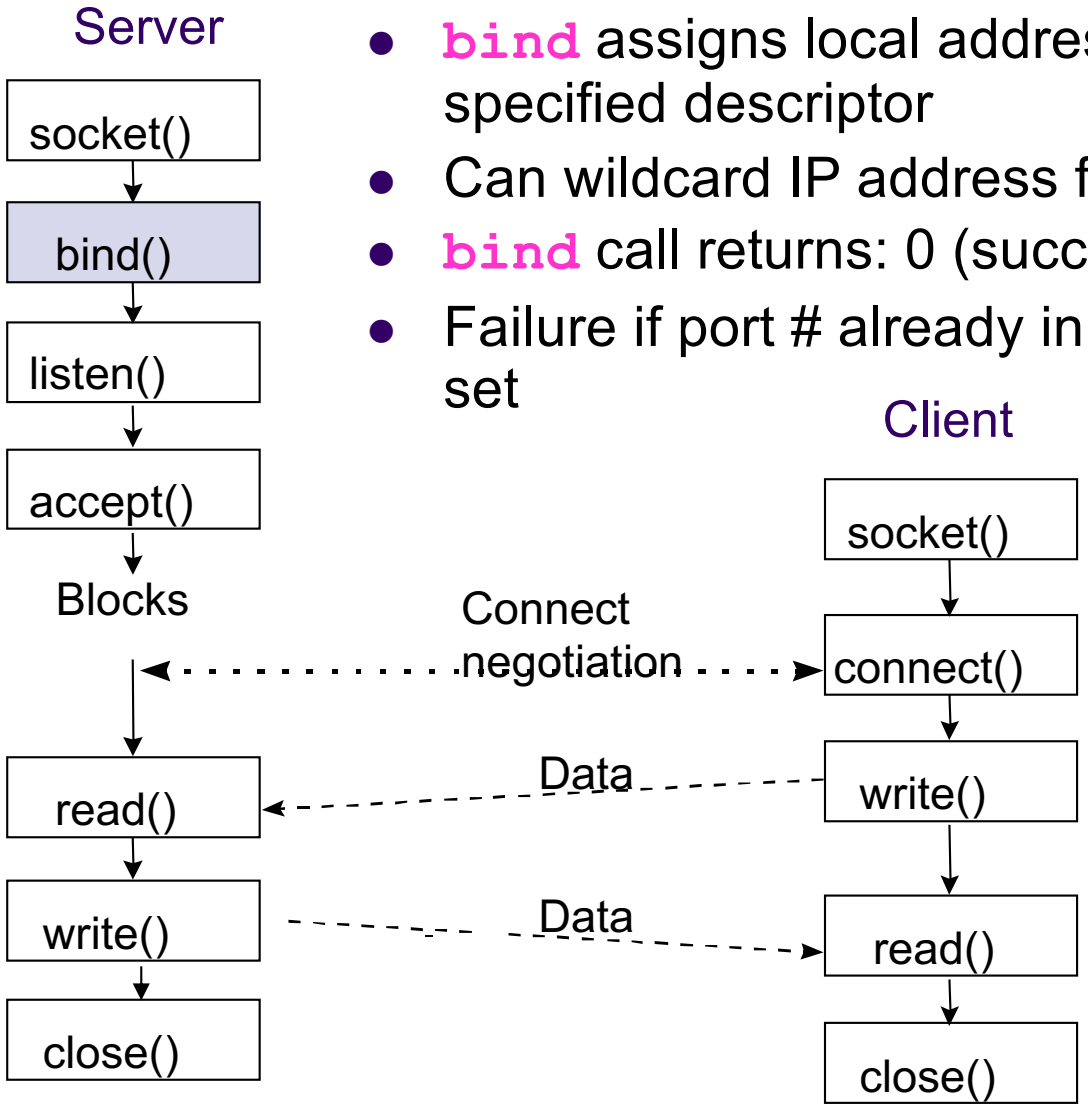


Socket Calls for Connection-Oriented Mode

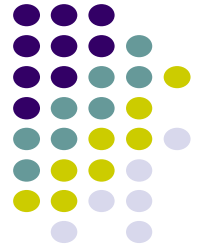


Server does Passive Open

- **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

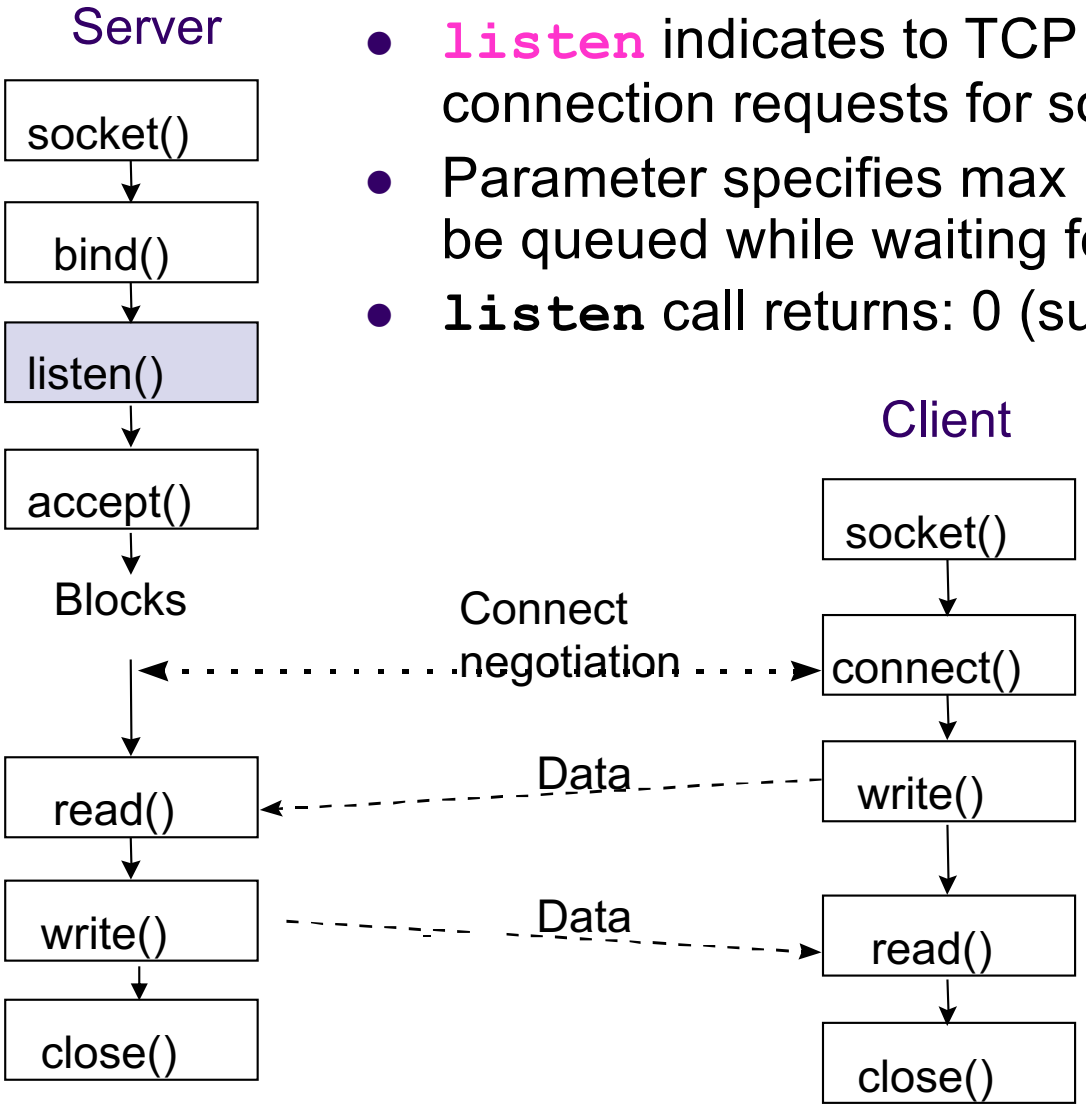


Socket Calls for Connection-Oriented Mode



Server does Passive Open

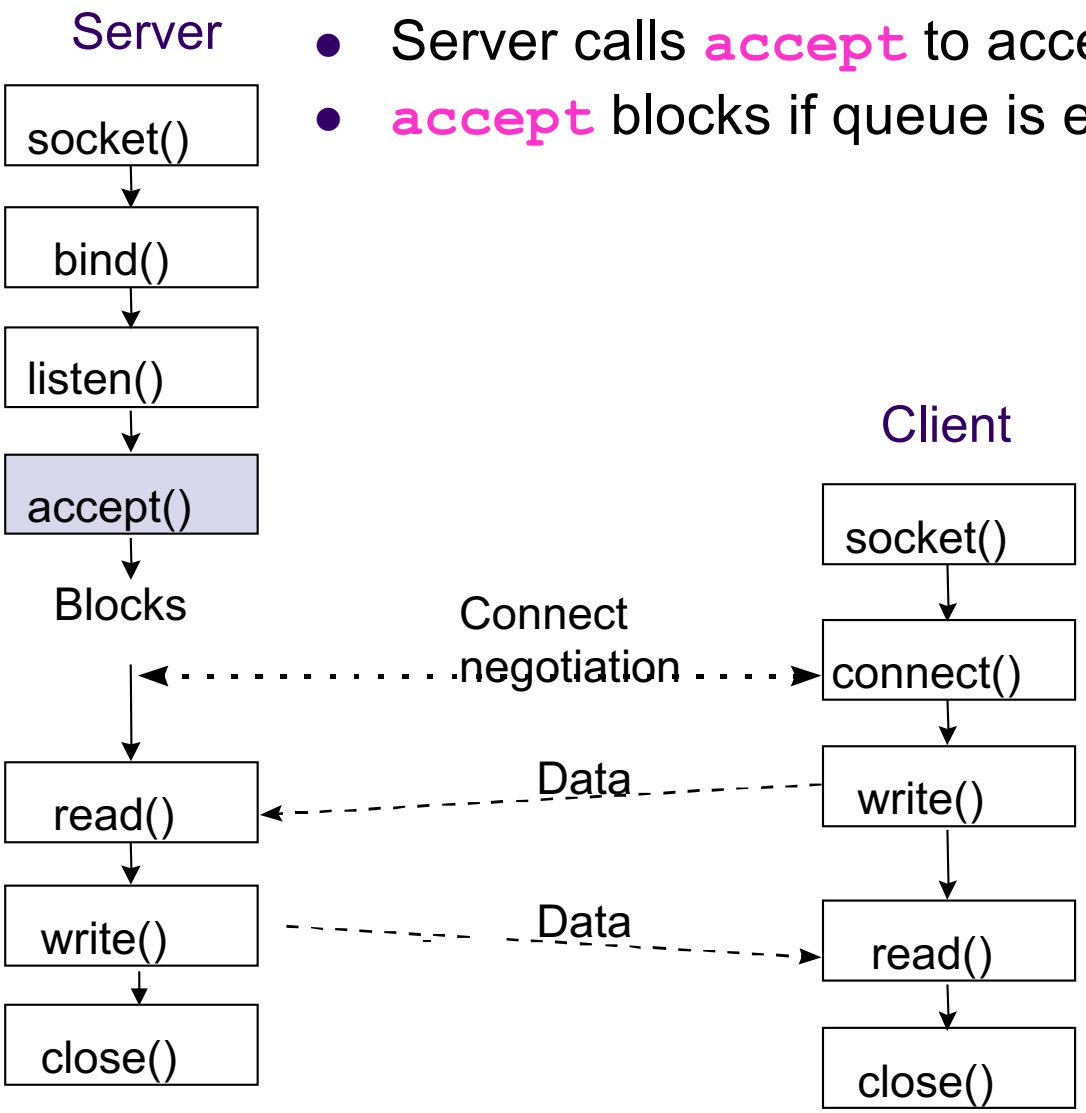
- **`listen`** indicates to TCP readiness to receive connection requests for socket with given descriptor
- Parameter specifies max number of requests that may be queued while waiting for server to accept them
- **`listen`** call returns: 0 (success); or -1 (failure)



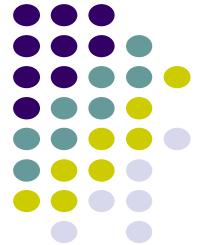
Socket Calls for Connection-Oriented Mode

Server does Passive Open

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

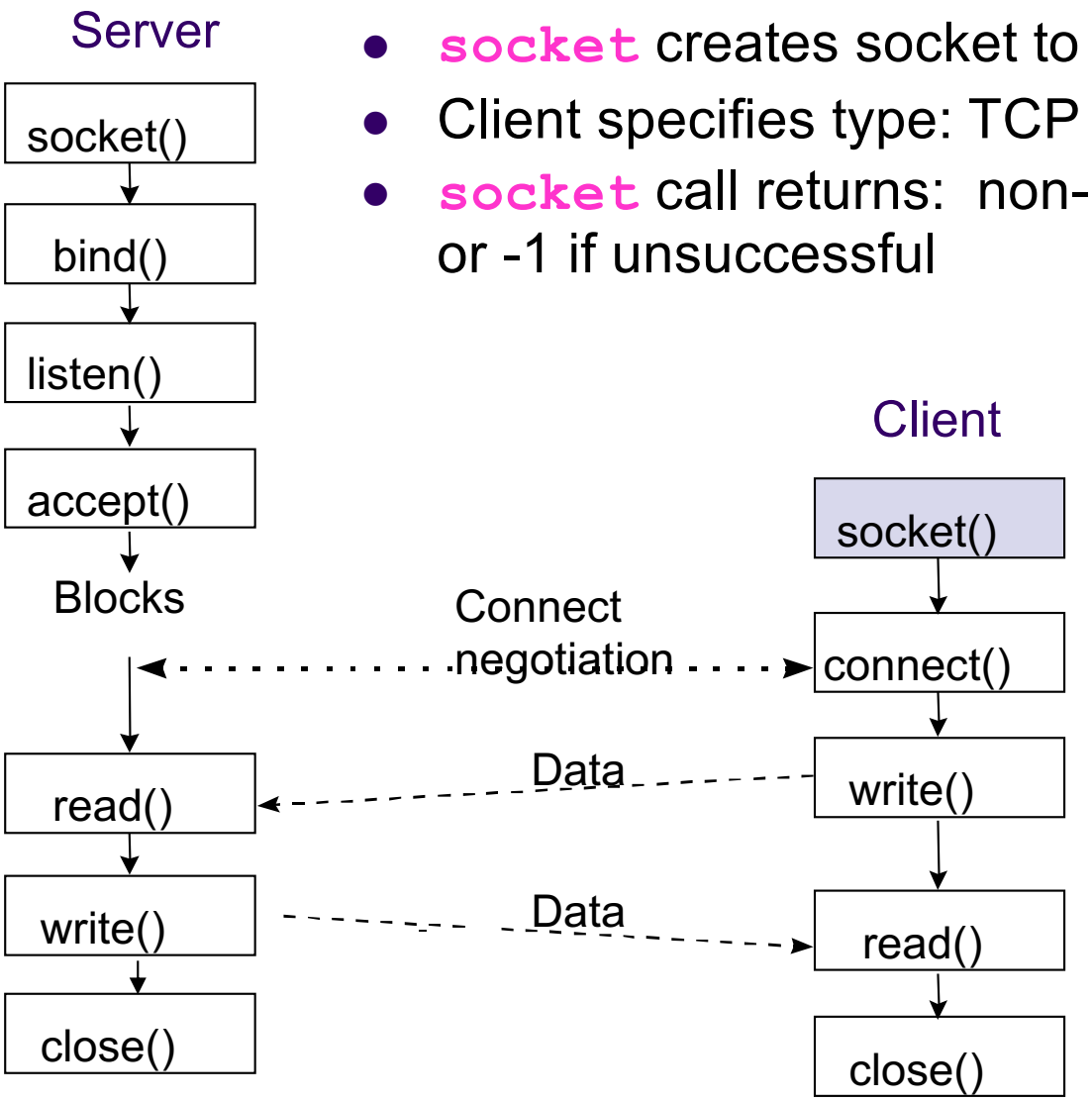


Socket Calls for Connection-Oriented Mode

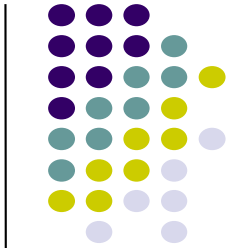


Client does Active Open

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

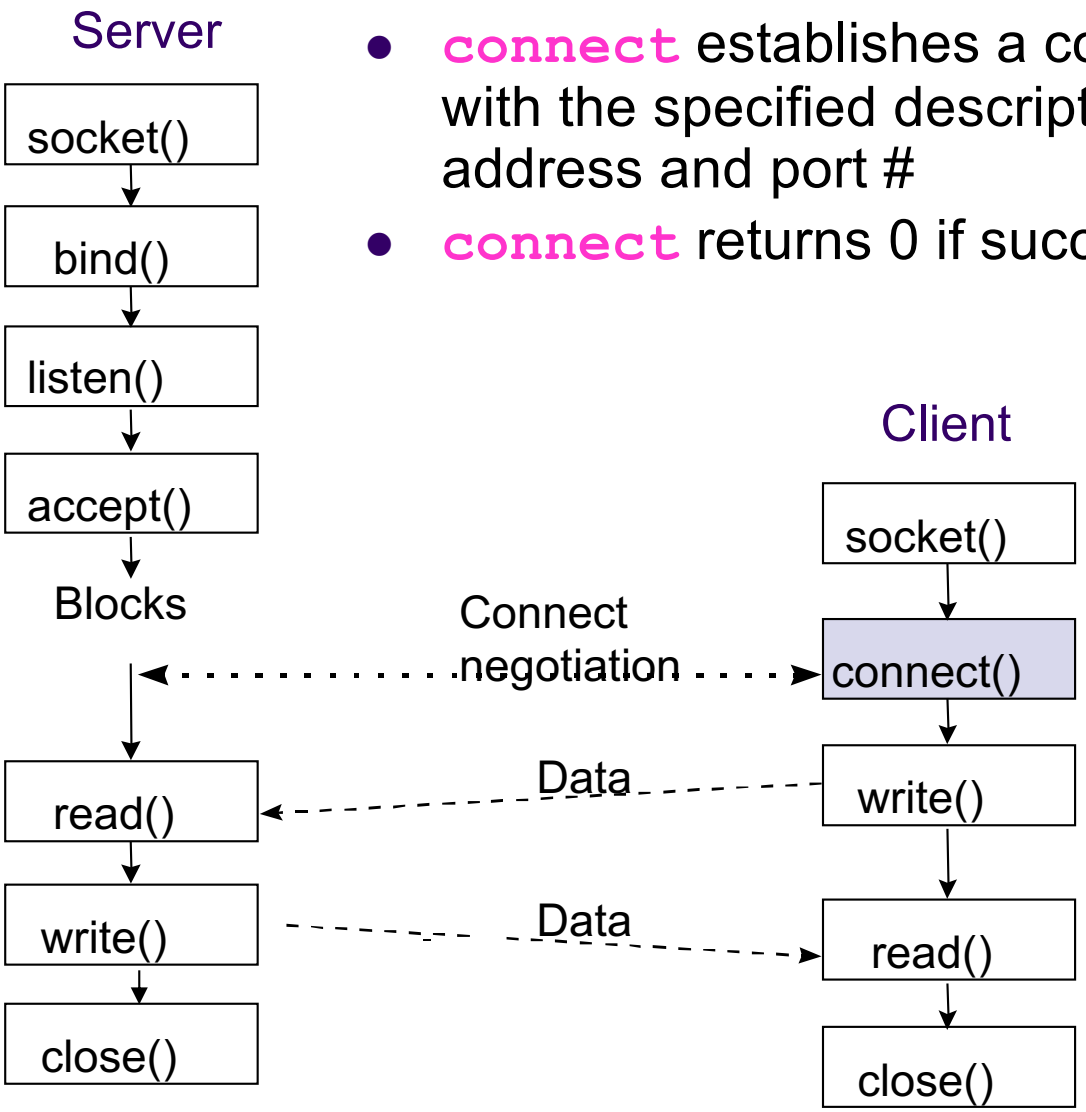


Socket Calls for Connection-Oriented Mode



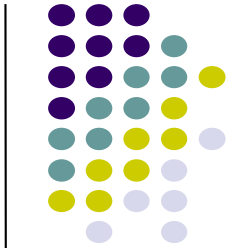
Client does Active Open

- `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

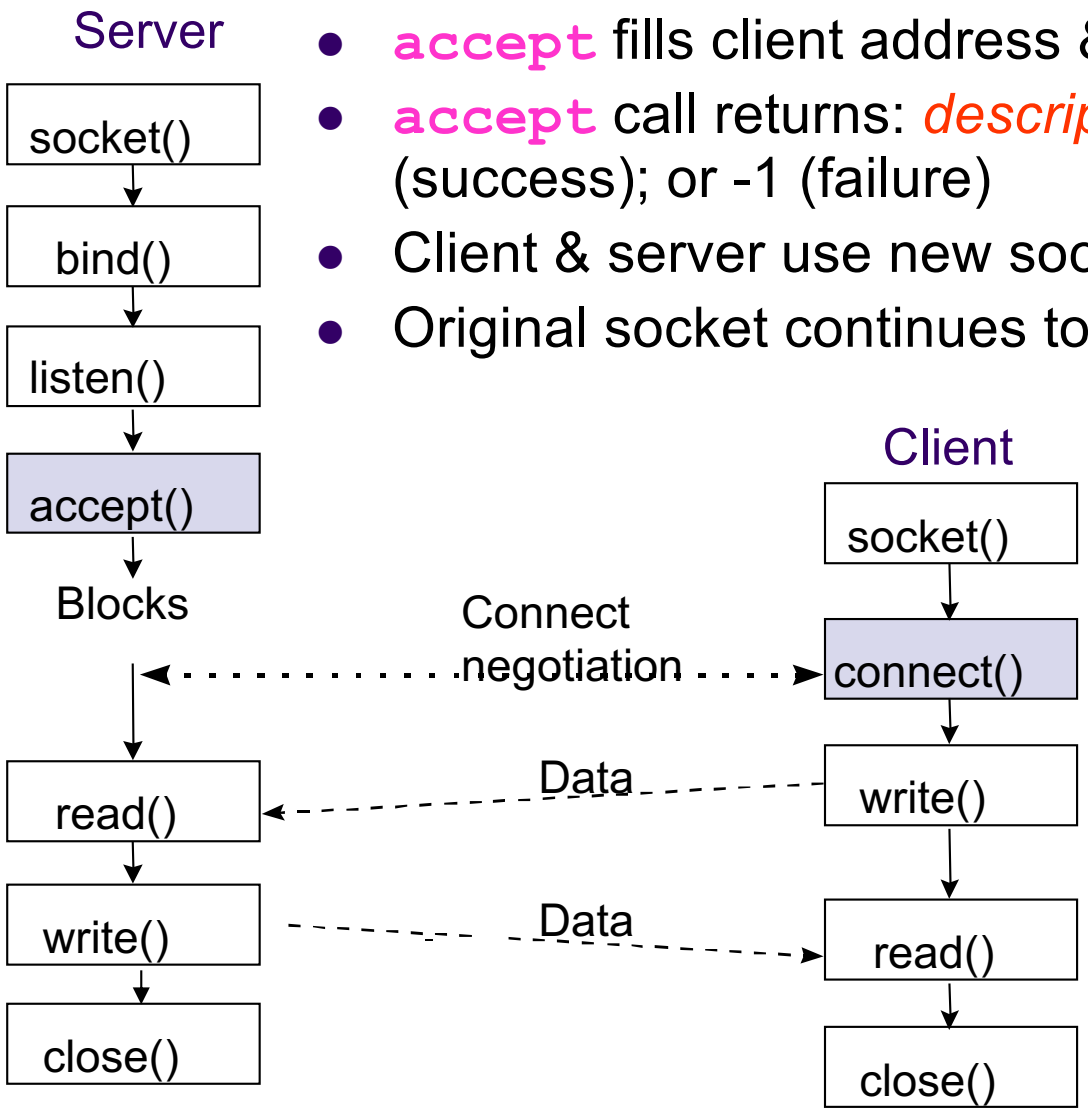


Note: `connect` initiates TCP three-way handshake

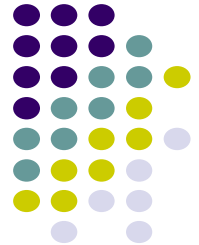
Socket Calls for Connection-Oriented Mode



- `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

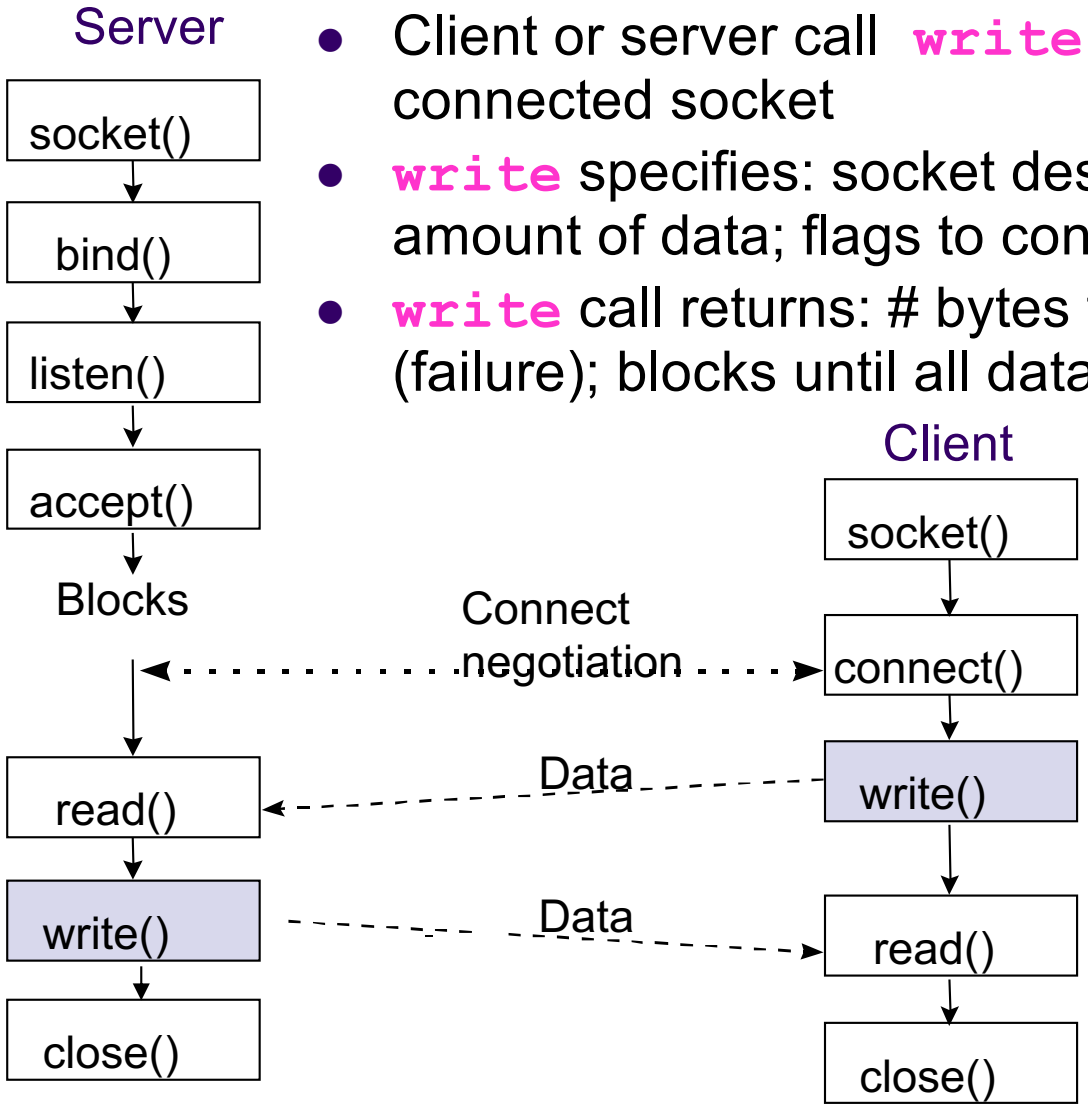


Socket Calls for Connection-Oriented Mode

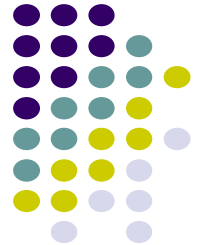


Data Transfer

- 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

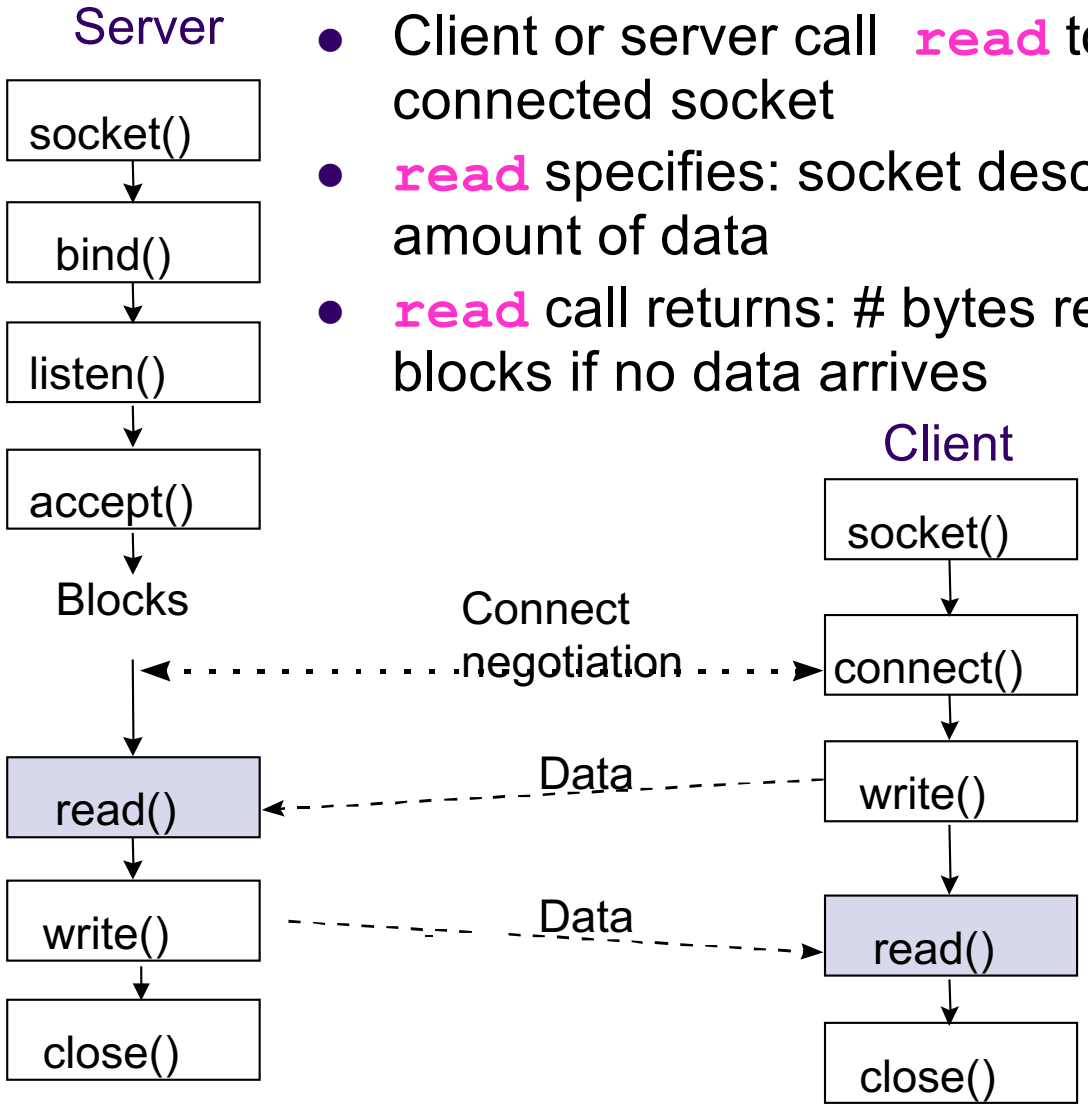


Socket Calls for Connection-Oriented Mode



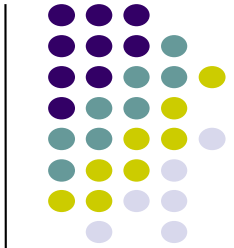
Data Transfer

- 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



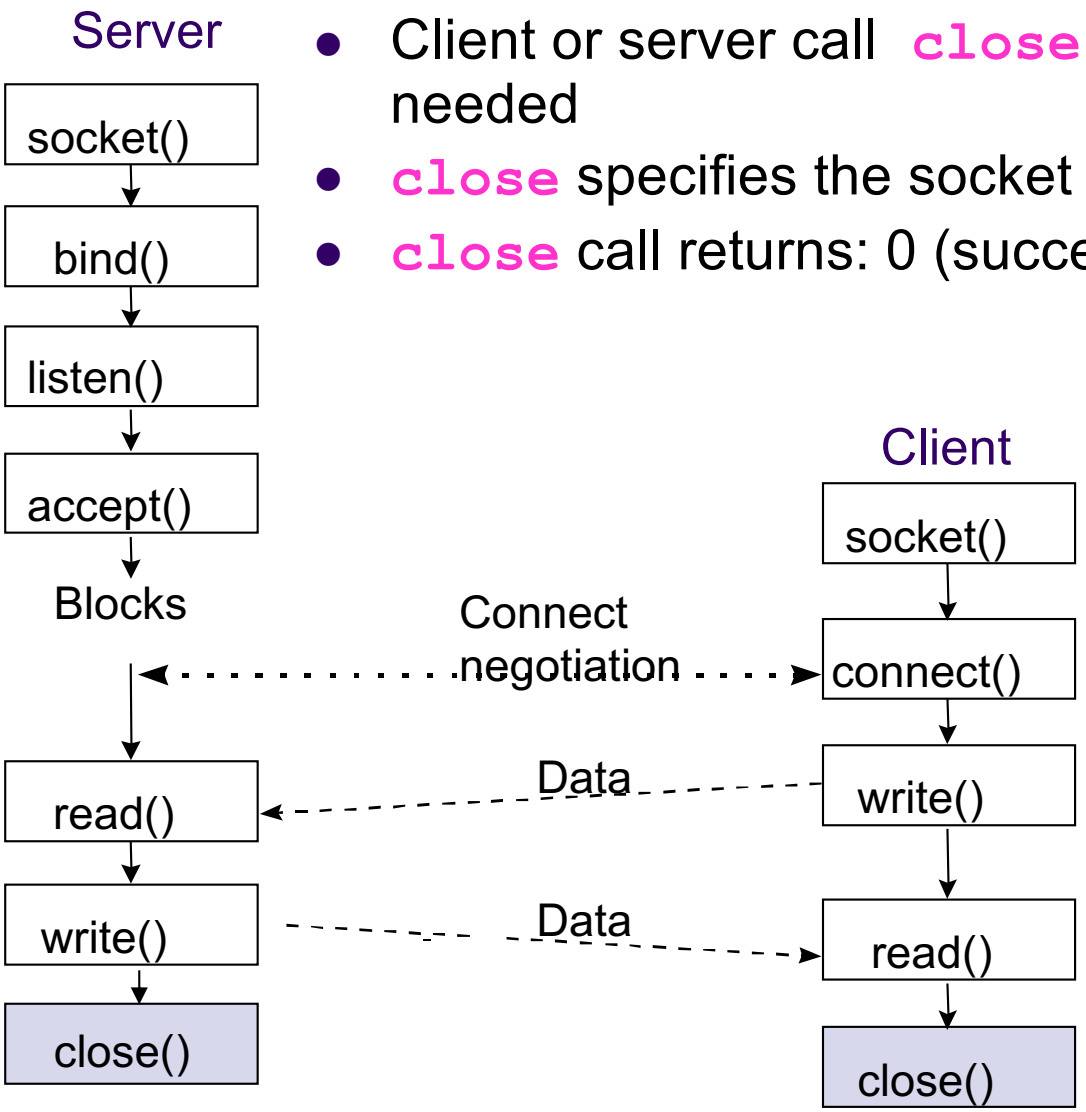
Note: `write` and `read` can be called multiple times to transfer byte streams in both directions

Socket Calls for Connection-Oriented Mode



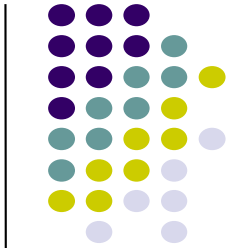
Connection Termination

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



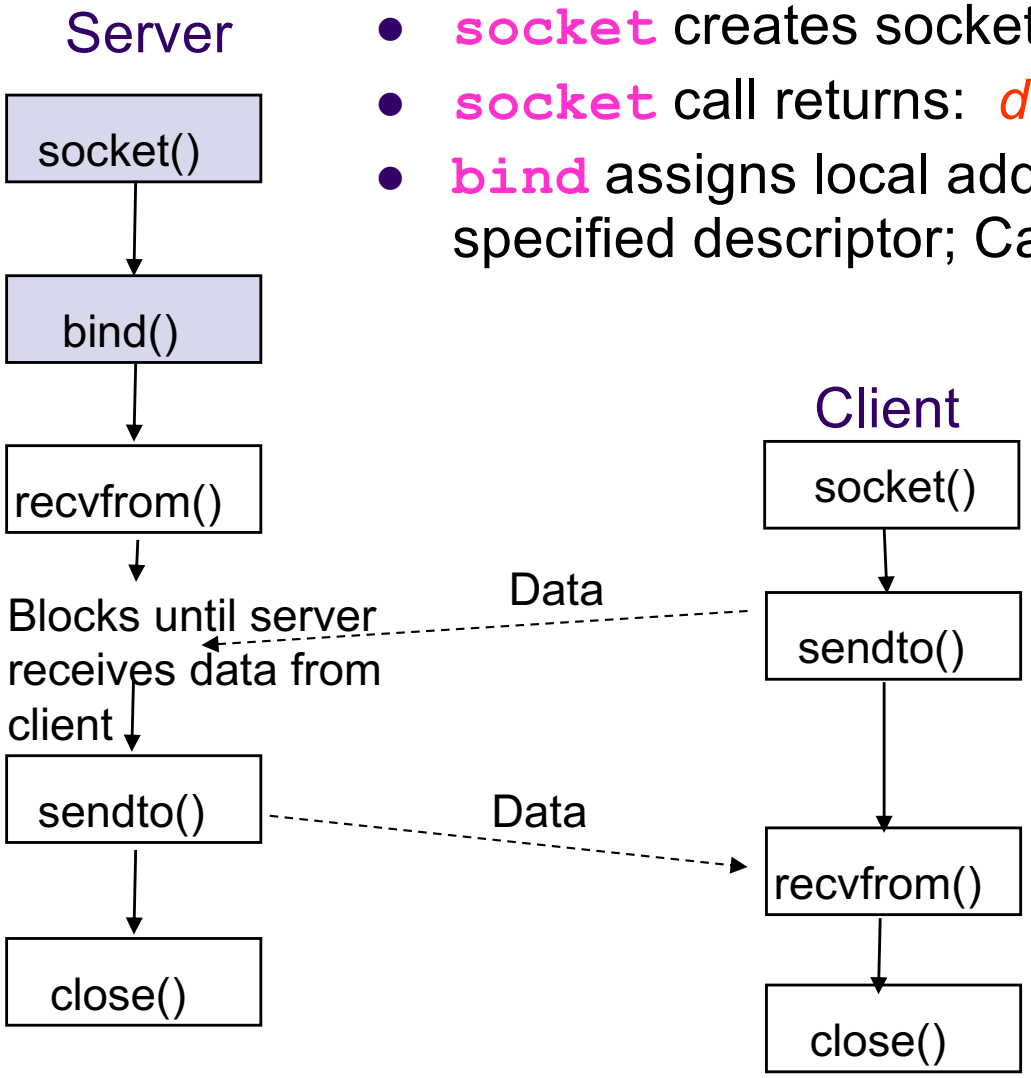
Note: `close` initiates TCP graceful close sequence

Socket Calls for Connection-Less Mode

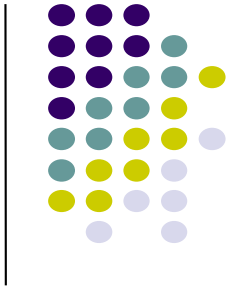


Server started

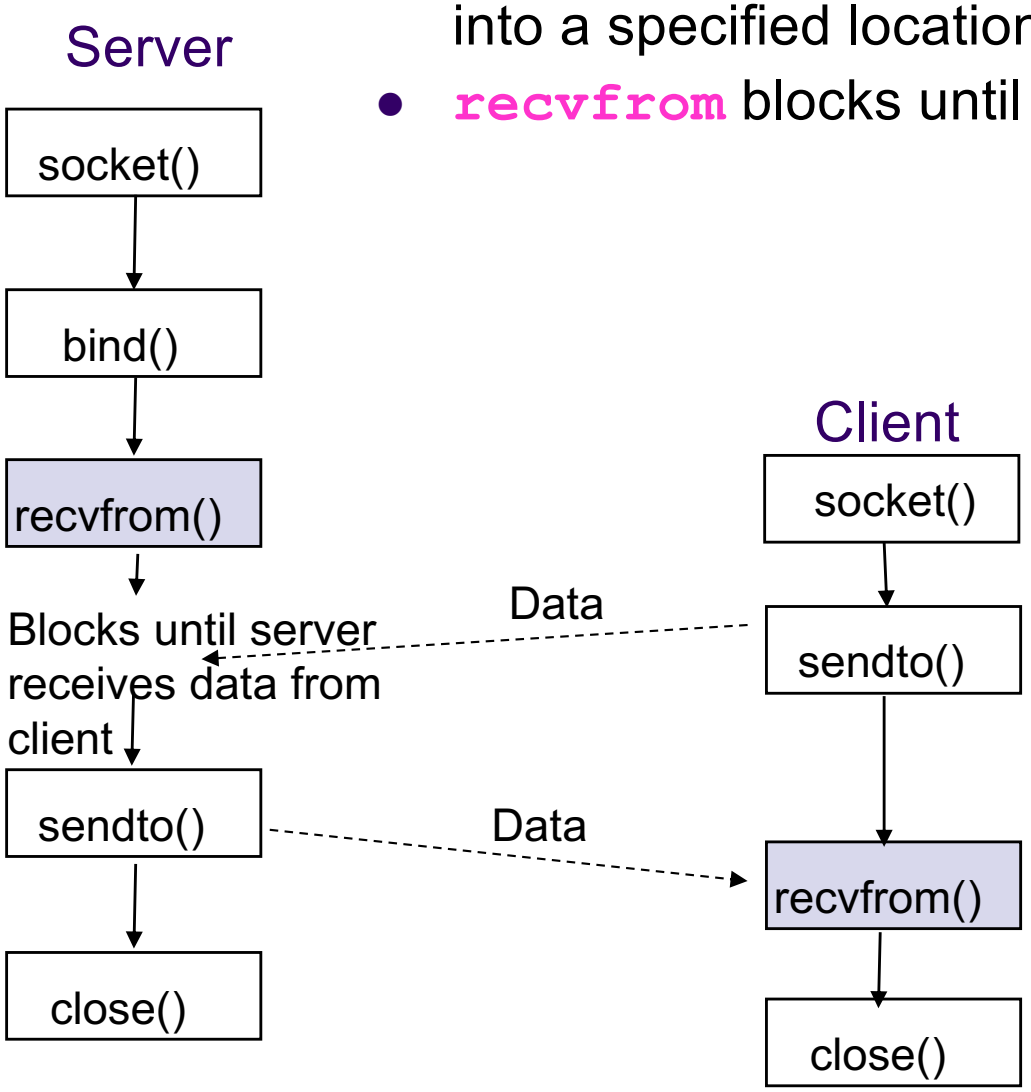
- **`socket`** creates socket of type UDP (datagram)
- **`socket`** call returns: *descriptor*; or -1 if unsuccessful
- **`bind`** assigns local address & port # to socket with specified descriptor; Can wildcard IP address



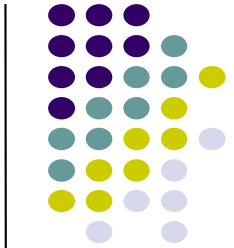
Socket Calls for Connection-Less Mode



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

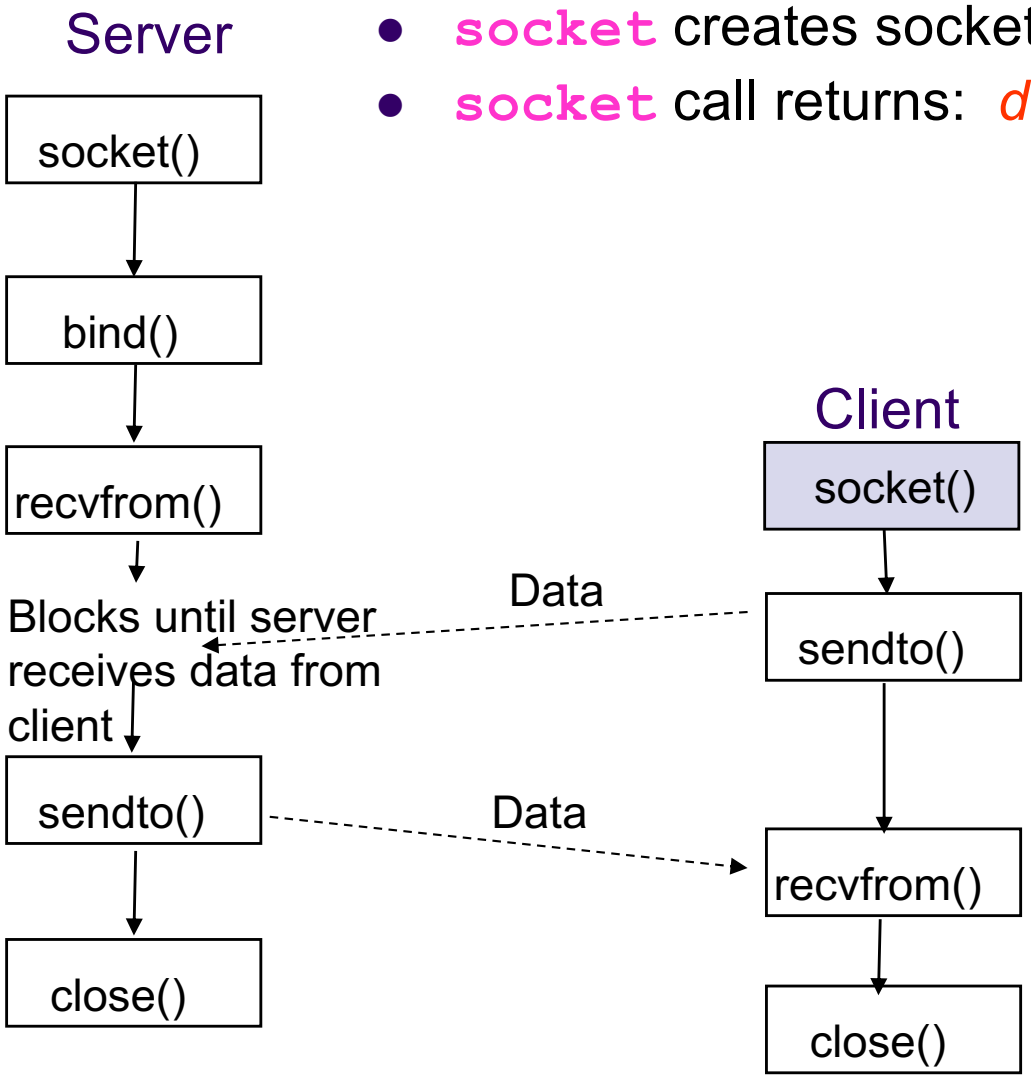


Socket Calls for Connection-Less Mode

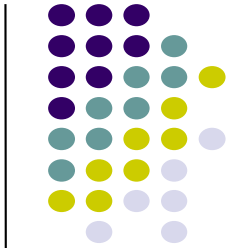


Client started

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

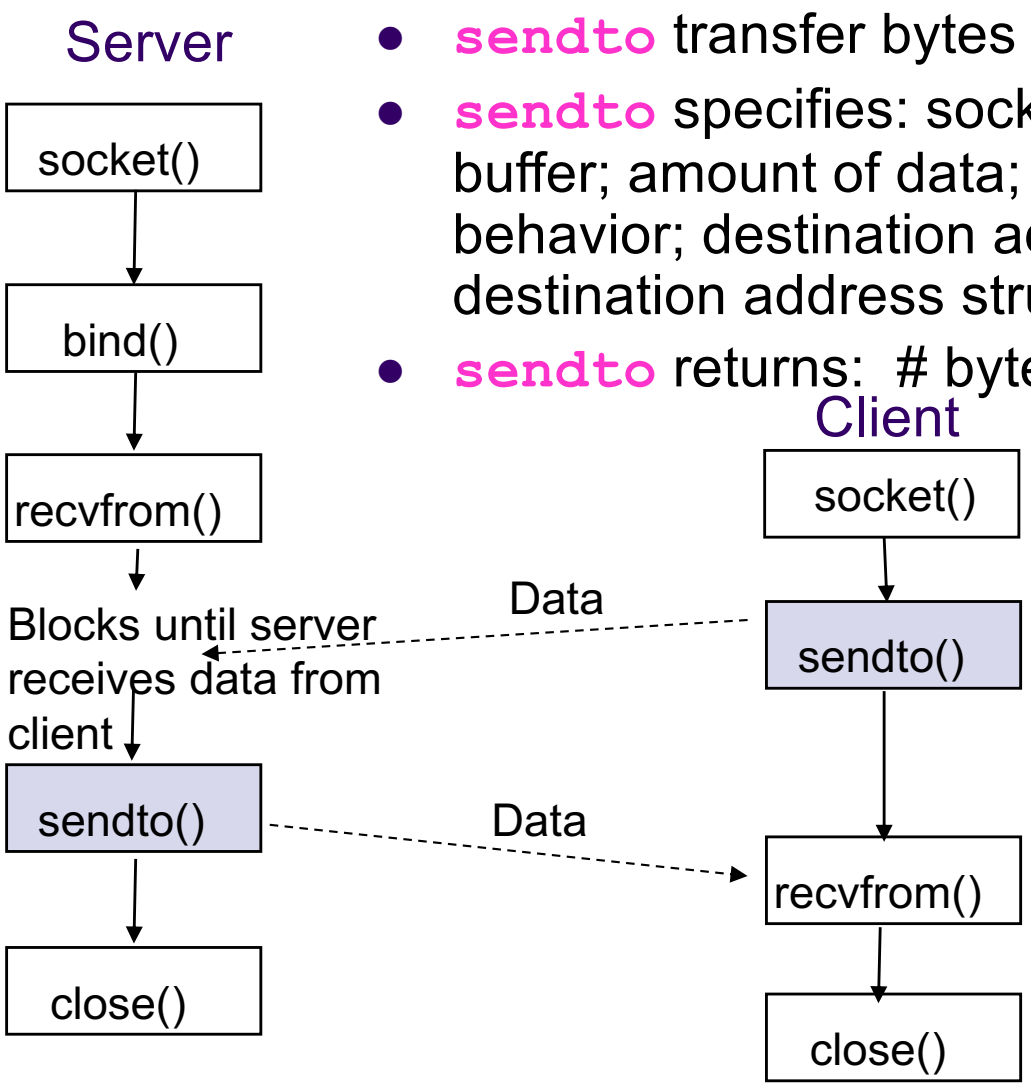


Socket Calls for Connection-Less Mode

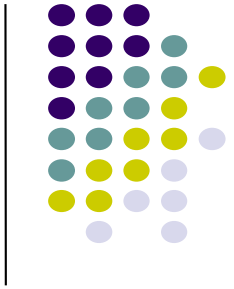


Client started

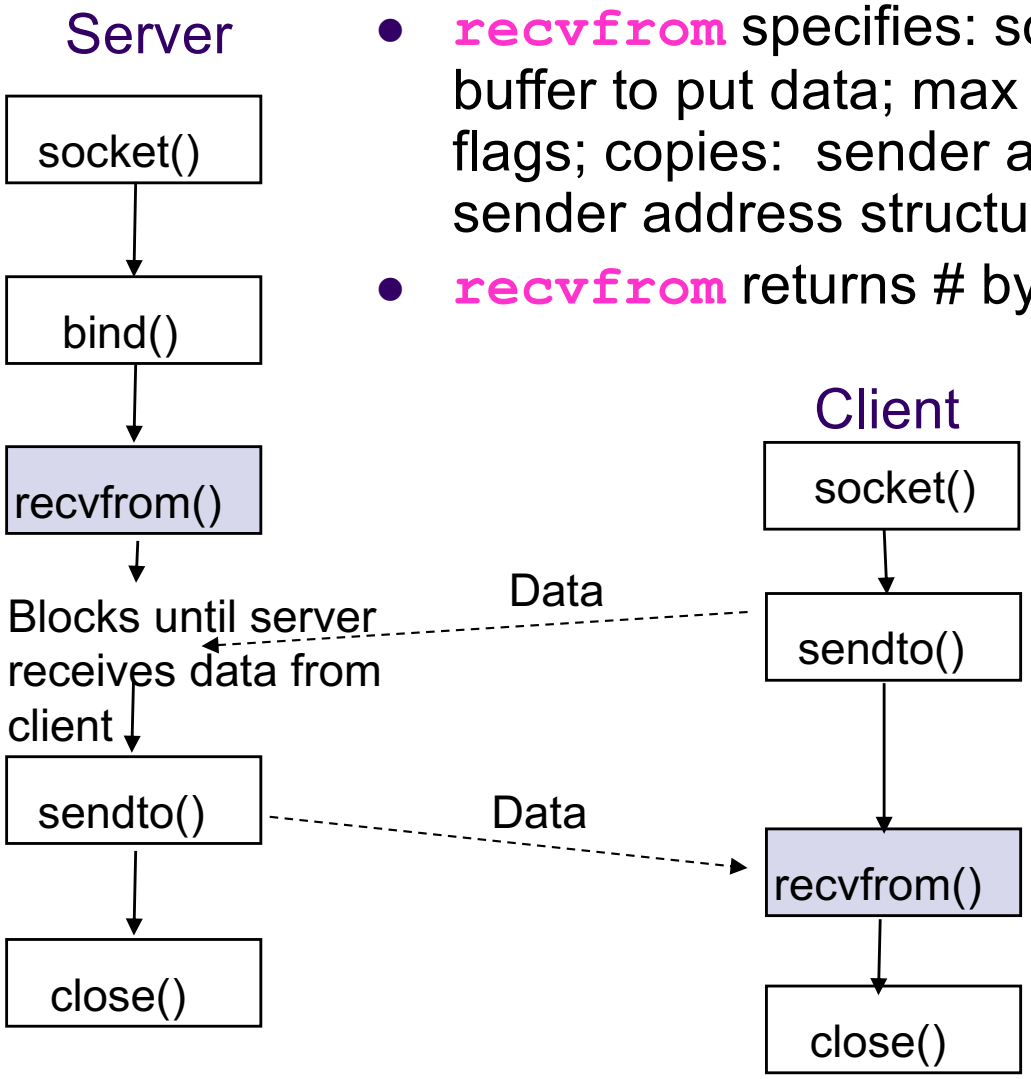
- **sendto** transfer bytes in buffer to specified socket
- **sendto** specifies: socket descriptor; pointer to a buffer; amount of data; flags to control transmission behavior; destination address & port #; length of destination address structure
- **sendto** returns: # bytes sent; or -1 if unsuccessful



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)