Introduction to Networking

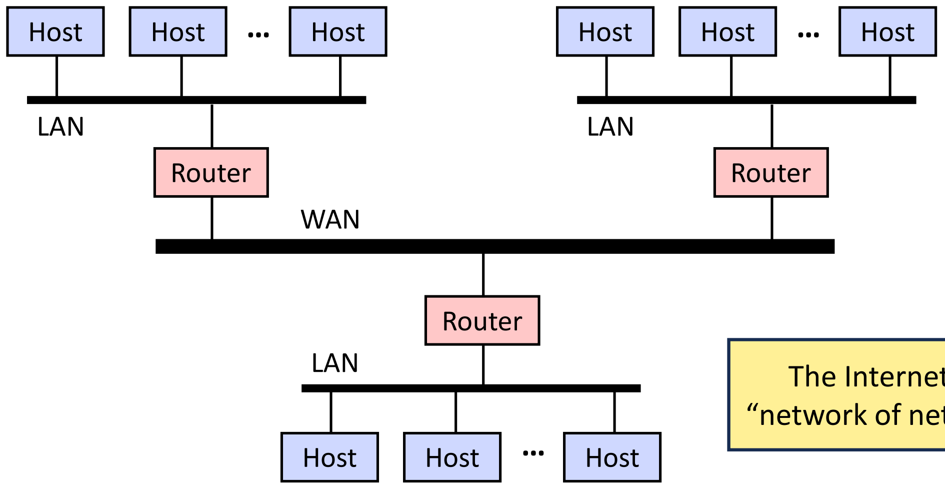
COMP 321

Dave Johnson

Background: Networking Layers (ISO Model)

7	Application	The actual application programs that are communicating
6	Presentation	Handling data representation issues
5	Session	Managing related transport layer connections
4	Transport	Addressing from and to individual processes on source and destination computer, reliability issues
3	Network	Routing: sending “packets” (through “routers”) to even not directly connected computers
2	Data Link	Framing: sending “frames” from one computer to another directly connected computer
1	Physical	Wires, connectors, voltages, etc., sending bits to a directly connected computer

Background: A Small Internet Example

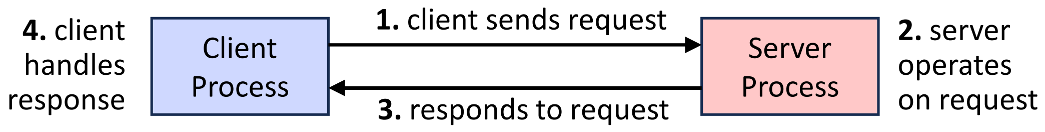


The Internet is a "network of networks"

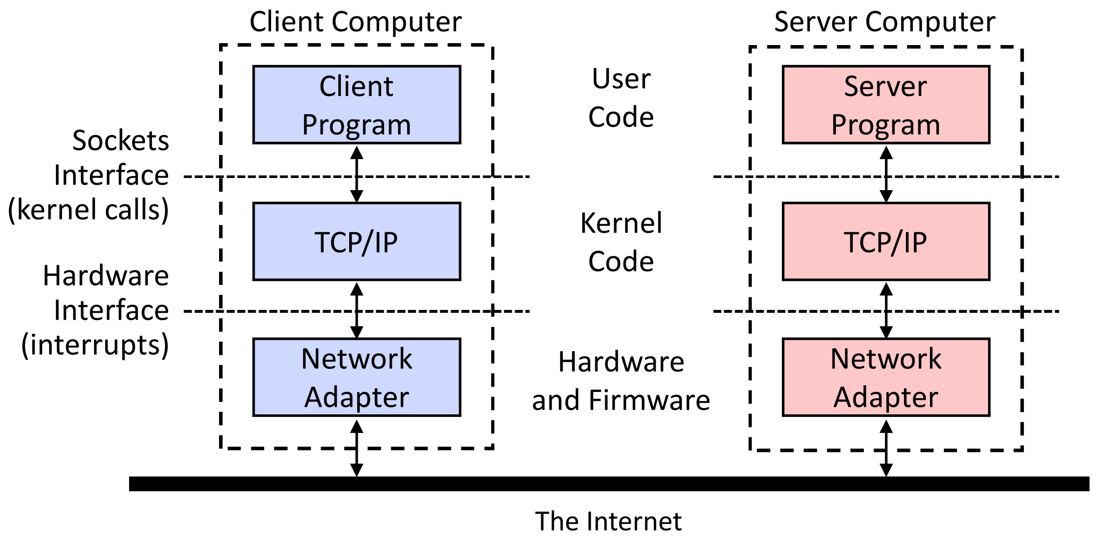
Client-Server Communication

Most networking communication is clients requesting services from servers

- Client – A computer (or program on some computer), requests some service
  - Sends a request for the service to some server that provides that service
  - Typically waits for a reply (response) from the server
- Server – A computer (or program on some computer), provides some service
  - Receives requests from clients, operates on request, and replies to client
  - Examples: web server, ssh server (sshd), echo server
  - Usually runs "forever", handling requests from any number of clients


4. client handles response
Client Process
1. client sends request
3. responds to request
Server Process
2. server operates on request

Organization of an Internet Application

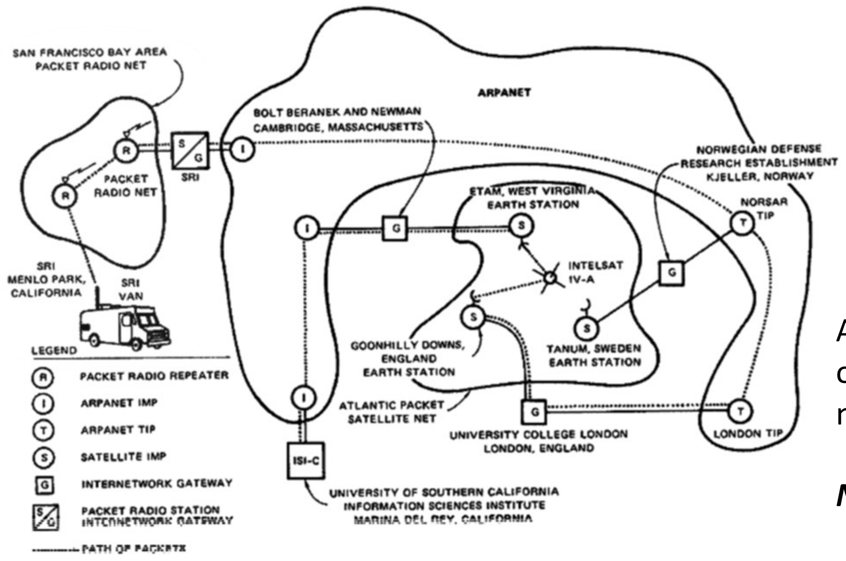


Evolution of the Internet

The ARPANET (Advanced Research Projects Agency Network) starting in 1966
- Initiated by U.S. Government research agency to enable “resource sharing”
- Connected some universities and other government contractors

The original protocols were not “TCP/IP”
- Conceptually similar to today’s IP (network layer) and TCP (transport layer) together in one protocol
- Called NCP (Network Control Program) (before the word “protocol” was used)
- Designed originally only for communicating directly over the ARPANET, running on the ARPANET physical layer and data link layer
- Beginning in 1973, IP designed as a replacement to connect other networks together (originally called “concatenating” networks, thus the “Catenet”)

The First Official Internet – Connected By TCP/IP


SAN FRANCISCO BAY AREA PACKET RADIO NET
ARPANET
BOLT BERANEK AND NEWMAN CAMBRIDGE, MASSACHUSETTS
PACKET RADIO NET
SRI
SRI MENLO PARK, CALIFORNIA
SRI VAN
NORWEGIAN DEFENSE RESEARCH ESTABLISHMENT KJELLER, NORWAY
NORSAR TIP
ETAM, WEST VIRGINIA EARTH STATION
INTELSAT IV-A
GOONHILLY DOWNS, ENGLAND EARTH STATION
TANUM, SWEDEN EARTH STATION
ATLANTIC PACKET SATELLITE NET
UNIVERSITY COLLEGE LONDON LONDON, ENGLAND
LONDON TIP
ISI-C
UNIVERSITY OF SOUTHERN CALIFORNIA INFORMATION SCIENCES INSTITUTE MARINA DEL REY, CALIFORNIA
LEGEND
PACKET RADIO REPEATER
ARPANET IMP
ARPANET TIP
SATELLITE IMP
INTERNETWORK GATEWAY
PACKET RADIO STATION INTERNETWORK GATEWAY
PATH OF PACKETS

A public demonstration of a 3-network TCP/IP network

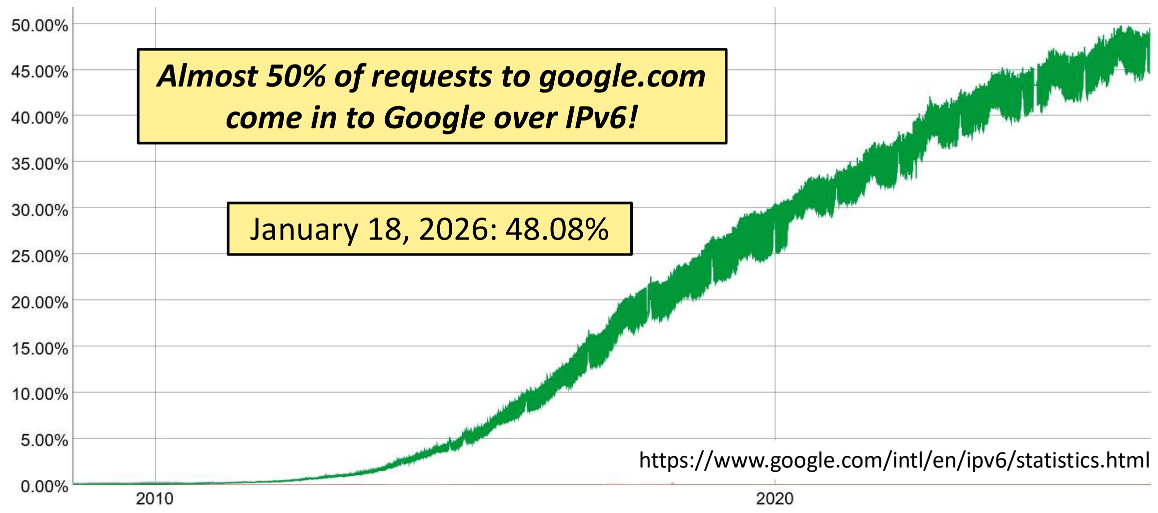
November 27, 1977

Evolution of the Internet

Today's commonly used IP is IP Version 4 (IPv4)

- In the late 1980's and early 1990's, becoming clear IPv4 had limitations
  - Chief among them was that it was running out of unique host IP addresses
- ***IPng*** ("IP next generation") effort started to design a successor for IPv4, with many proposals
  - SIP: Simple Internet Protocol
  - PIP: "P" Internet Protocol
  - TUBA: TCP and UDP over Big Addresses
  - CATNIP: Common Architecture Technology for the Next generation IP
  - SIPP: Simple Internet Protocol "Plus"
- SIPP selected as the "winner" and renamed IP Version 6 (IPv6), January 1995
  - First official protocol specification published in December 1995

IPv6 Global Adoption as Measured by Google



Almost 50% of requests to google.com come in to Google over IPv6!

January 18, 2026: 48.08%

https://www.google.com/intl/en/ipv6/statistics.html

TCP vs. UDP

The Internet uses two common transport layer protocols

Transmission Control Protocol (TCP)

- Used for “most” communication
- Provides a bi-directional reliable stream of bytes (carried by IP packets)
- Guarantees no loss, duplication, errors, etc.
- Addresses individual sender and receiver processes

User Datagram Protocol (UDP)

- Provides roughly the same service as IP packets, no guarantees of reliability
- A bi-directional “best effort” datagram service
- Addresses individual sender and receiver processes

Internet Host Addressing

IP Version 4 (IPv4)

- Each computer has one (or more) IPv4 addresses
  - 32-bit number, consisting (roughly) of a network number and host number
  - Commonly written with the 4 bytes in "dotted decimal" notation

    128.42.124.180

IP Version 6 (IPv6)

- Conceptually, roughly the same basic addressing architecture as in IPv4
  - But now 128 bits rather than 32 bits
  - Commonly written in 8 hex chunks of 16 bits (4 hex digits) each

    fe80:0000:0000:0000:164c:5887:c02c:f6a2
    fe80**::**164c:5887:c02c:f6a2

TCP and UDP Port Numbers

Both use the same scheme for port numbers to identify endpoints

- A nonzero unsigned 16-bit integer
- This combination must be unique:
  - source IP address, source port number,
  - destination IP address, destination port number,
  - and protocol (TCP or UDP)
- Can specify your own port numbers, or they generally can be filled in for you
- Port numbers less than 1024 are reserved for "well known" services
  - Examples: TCP port 80 reserved for web servers, 443 for secure web (https)
- The kernel keeps track of which local process has any given port number open

Internet Host Naming

Hierarchically assigned and managed through the Domain Name System

- A very large replicated, distributed database system and query protocol
- Example: Looking up the IP address for "www.rice.edu"
  - Ask for it from any server for the (unnamed) root of zone
  - It replies, "don't ask me, ask any of these servers for the .edu zone"
  - Ask one of those .edu servers for the "www.rice.edu" IP address
  - It replies, "don't ask me, ask any of these servers for the .rice.edu zone"
  - Ask one of those .rice.edu servers for the "www.rice.edu" IP address
  - It replies with the answer IP address for "www.rice.edu"
- Caching means this full lookup process is usually bypassed or short circuited

The Domain Name System

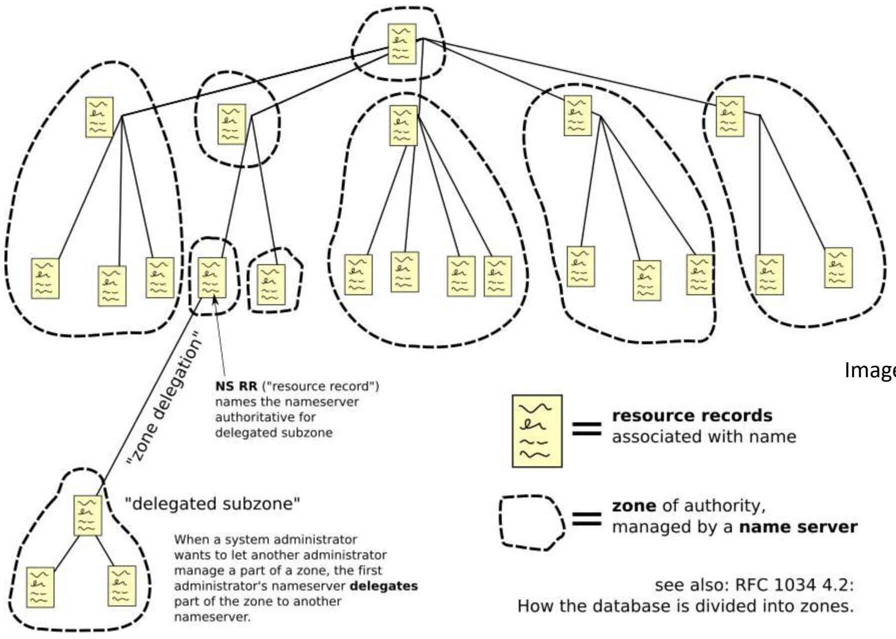
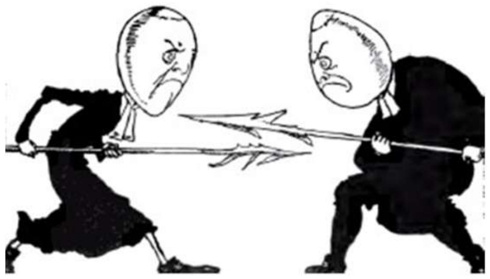

"zone delegation"
NS RR ("resource record") names the nameserver authoritative for delegated subzone
"delegated subzone"
When a system administrator wants to let another administrator manage a part of a zone, the first administrator's nameserver delegates part of the zone to another nameserver.
= resource records associated with name
= zone of authority, managed by a name server
see also: RFC 1034 4.2: How the database is divided into zones.

Image: Wikipedia

Byte Ordering: Big-Endian vs. Little-Endian

Naming comes from "Gulliver's Travels", Jonathan Swift (Lemuel Gulliver), 1726

"It is allowed on all hands, that the primitive way of ***breaking eggs*** before we eat them, was ***upon the larger end***: but his present Majesty's grandfather, while he was a boy, going to eat an egg, and breaking it according to the ancient practice, happened to cut one of his fingers. Whereupon the Emperor his father published an edict, commanding all his subjects, upon great penalties, to break ***the smaller end*** of their eggs. The people so highly resented this law, that our histories tell us there have been six rebellions raised on that account; wherein one Emperor lost his life, and another his crown. . . ."



15

Byte Ordering: Big-Endian vs. Little-Endian

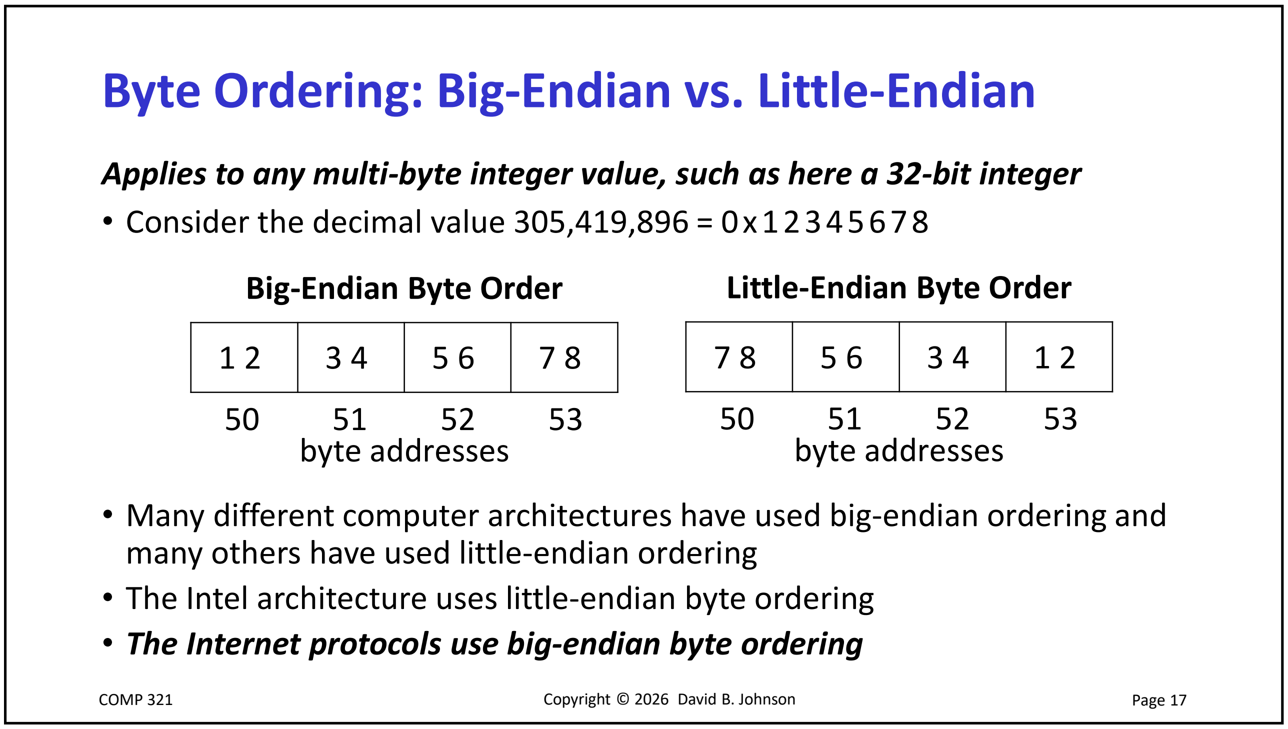
"On Holy Wars and a Plea for Peace," Danny Cohen, IEN 137, April 1, 1980

"It may be interesting to notice that the point which Jonathan Swift tried to convey in Gulliver's Travels in exactly the opposite of the point of this note.

"Swift's point is that the difference between breaking the egg at the little-end and breaking it at the big-end is trivial. Therefore, he suggests, that everyone does it in his own preferred way.

"We agree that the difference between sending eggs with the little- or the big-end first is trivial, but we insist that everyone must do it in the same way, to avoid anarchy. Since the difference is trivial we may choose either way, but a decision must be made."

16



Byte Ordering: Big-Endian vs. Little-Endian

Applies to any multi-byte integer value, such as here a 32-bit integer

- Consider the decimal value 305,419,896 = 0x12345678

Big-Endian Byte Order

50	51	52	53
12	34	56	78

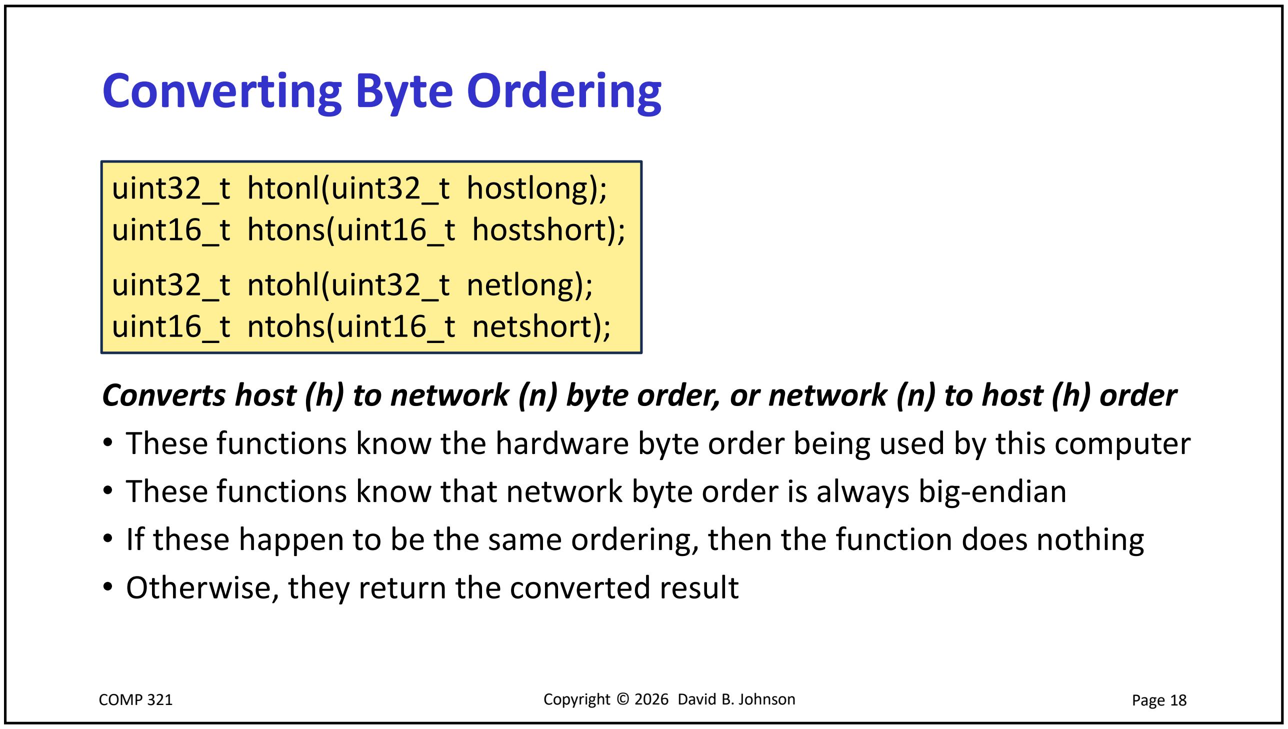
byte addresses

Little-Endian Byte Order

50	51	52	53
78	56	34	12

byte addresses

- Many different computer architectures have used big-endian ordering and many others have used little-endian ordering
- The Intel architecture uses little-endian byte ordering
- ***The Internet protocols use big-endian byte ordering***



Converting Byte Ordering

```
uint32_t  htonl(uint32_t  hostlong);
uint16_t  htons(uint16_t  hostshort);

uint32_t  ntohl(uint32_t  netlong);
uint16_t  ntohs(uint16_t  netshort);
```

Converts host (h) to network (n) byte order, or network (n) to host (h) order

- These functions know the hardware byte order being used by this computer
- These functions know that network byte order is always big-endian
- If these happen to be the same ordering, then the function does nothing
- Otherwise, they return the converted result

Converting To/From Printable IP Addresses

```
const char *inet_ntop(AF_INET,  const void *restrict src,
      char dst[restrict .size],  socklen_t size);
int  inet_pton(AF_INET,  const char *restrict src,  void *restrict dst);
```

Converts "network" (n) to "presentation" (p) IP address format or the reverse

- The "network" format of an IP address is the 32-bit integer (***in network byte order***)
- The "presentation" format here is a character string like "128.42.124.180"
- The AF_INET means we are using IPv4 addresses ("address family Internet")
- inet_ntop returns char *dst on success, NULL on error
- inet_pton returns 1 on success, 0 if not valid format, -1 if address family unknown

Looking up IP Addresses

```
int  getaddrinfo(const char *restrict node,  const char *restrict service,
      const struct addrinfo *restrict hints,  struct addrinfo **restrict res);
void freeaddrinfo(struct addrinfo *res);
const char *gai_strerror(int  errcode);
```

Query the DNS using a hostname or IP address specified by "node"

- You pass in the ***address of*** a "struct addrinfo * " as res
- Lookup fills in the pointer at address res with a pointer to a linked list of results, each as a "struct addrinfo"
  - And returns a getaddrinfo-specific error code for any errors
  - Calling gai_strerror will give you a string explanation of that error code
- ***You must later call freeaddrinfo to free the memory in that linked list!***

Calling getaddrinfo()

Suppose you want to open a TCP connection using a numeric port number

- Initialize things and make the actual getaddrinfo call

```
char *hostname, *port;
struct addrinfo *results;
struct addrinfo hints;

memset(&hints, 0, sizeof(hints));
hints.ai_socktype = SOCK_STREAM;
hints.flags = AI_NUMERICSERV | AI_ADDRCONFIG;
error = getaddrinfo(hostname,  port,  &hints,  &results);
```

The port is like “1234”

Return IPv4 and/or IPv6 addresses depending on own config

- “results” now points to beginning of a linked list of results …
- Must eventually free the memory in this linked list using

```
freeaddrinfo(results);
```

The Sockets API

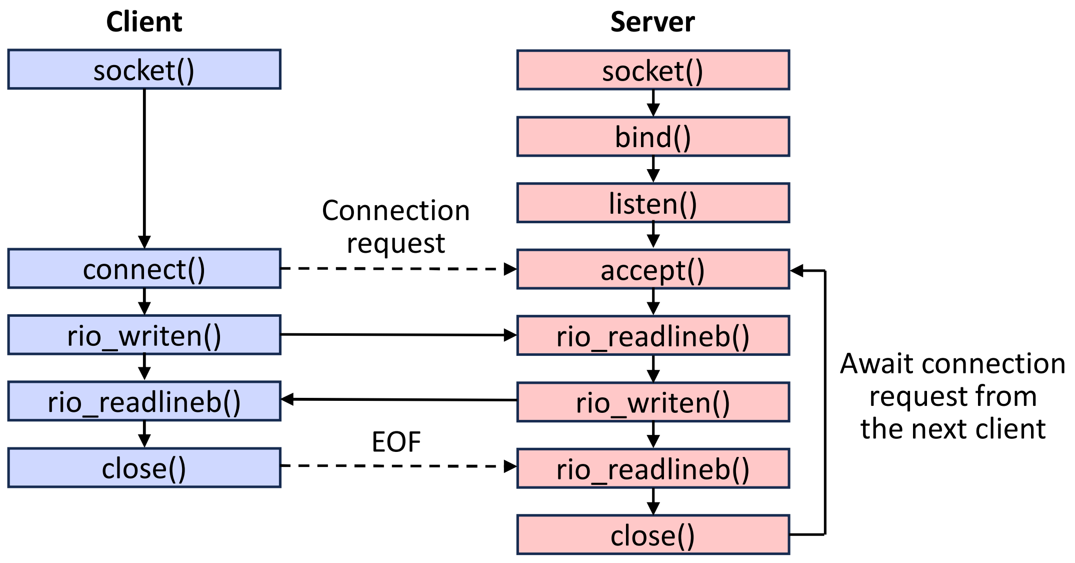
Provides the programming interface for network applications

- Created in the early 1980’s as part of the original Berkeley BSD distribution of Unix that contained an early implementation of TCP/IP protocols
- Based on the client/server network model

What is a socket?

- To the kernel, a socket is an endpoint of network communication
- To a user program, a socket is a type of file descriptor that lets the program read from and write to the network
- Clients and servers communicate with each through socket file descriptor I/O
- The main distinction from “regular” file descriptors and socket file descriptors is how the program “opens” the socket file descriptor

Overview of the Sockets Interface



23

Listening vs. Connected Socket Descriptors

The return from listen() gives you a "listening" socket file descriptor

- Represents the server process waiting for incoming connection requests
  - Listening at the address and port number from the bind() call
- Remembers up to limited number of new requests until the server gets to them
- Normally, created once by the server and reused as each new connection request comes in (as long as this server is willing to accept new connections)

The return from accept() gives you a <u>new</u> "connected" socket file descriptor

- Each return from accept() on listening socket represents a <u>new</u> connection, giving a <u>new</u> connected socket file descriptor for talking with that new client
- Separate listening socket remains, listening for new connections from clients
- Server closes each new connected socket when done with that particular client

24

Creating a Socket (Client and Server)

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

Creates a new socket, but not "open" yet to anything

- To create a socket for TCP communication, use

  `int  socket(AF_INET,  SOCK_STREAM,  0);`

- To create a socket for UDP communication, use

  `int  socket(AF_INET,  SOCK_DGRAM,  0);`

- The "protocol" field is generally unused (thus, 0), since there is only a single AF_INET protocol for SOCK_STREAM or for SOCK_DGRAM

The Client Connecting to the Server

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

Does not return to the client until connected to the server (or error)

- Must have already used socket() to create the socket this will then connect
- addr points to a struct sockaddr that specifies, e.g., the IP address and port number for the client to connect to
  - "struct sockaddr" is the "generic" address structure
  - The IPv4-specific version is a "struct sockaddr_in"

Binding the Server Socket for What to Listen To

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

Needs a socket previously created by the server by calling socket()

- addr points to a struct sockaddr that specifies, e.g., the IP address and port number to listen for new connections to
  - "struct sockaddr" is the "generic" address structure
  - The IPv4-specific version is a "struct sockaddr_in"

Setting up the Server to Wait for New Connections

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

Sets up this socket as a "listening" socket that can wait for new connections

- Should have already used bind() to specify which connections to listen for
- Tells the kernel to listen for new connections to this socket
  - The kernel will remember up to "backlog" number of pending connections, if they come in before the server process gets a chance to accept each

Waiting at the Server for the Next Connection

```
int  accept(int sockfd,  struct sockaddr *_Nullable restrict addr,
       socklen_t *_Nullable restrict addrlen);
```

Does not return until the next client tries to connect

- Should have already used bind() to specify which connections to listen for
- Should have already used listen() to tell the kernel to listen for connections
- Returns a new "connected" file descriptor representing the new connection
  - And the "listening" file descriptor is still there, listening for additional new connections – Thus now both a listening and a separate connected socket
  - Each call to accept() gives the server one of the pending new connections, giving you a <u>new</u> connected socket and leaving the separate listening socket still listening for more connections