# Computer Network Programming

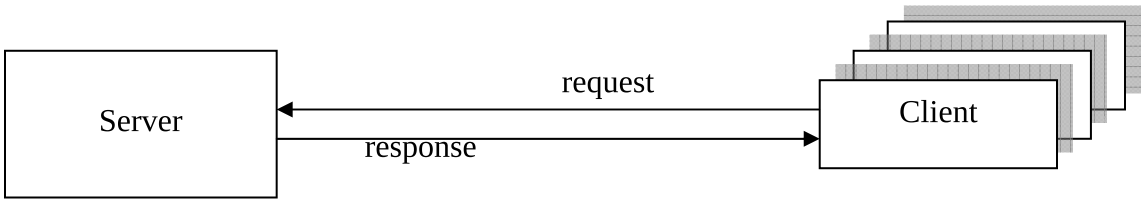
## Intro to Sockets

Dr. Sam Hsu
Computer Science & Engineering
Florida Atlantic University

# Intro to Sockets

- The Client/Server Model
- Layered Network Structure
- Sockets
- Internet Addressing
- Protocol Port Numbers
- Socket Programming
- Network Byte Order
- Connectionless/Connection-oriented Examples

# The Client/Server Model




| | |
|---|---|
| Starts and initializes. | Starts and initializes. |
| Goes to sleep waiting for client to connect. | Waits for user input. |
| When contacted by client, calls/creates a handler to handle. (Goes back to sleep: concurrent) | Upon receiving user request, contacts server, sends request on user's behalf. |
| Handles client request, sends back reply ... | Waits for server reply, sends results to user, sends another user request ... |
| Closes client connection. (Goes back to sleep: iterative). | Closes server connection. |

# Client/Server on an LAN

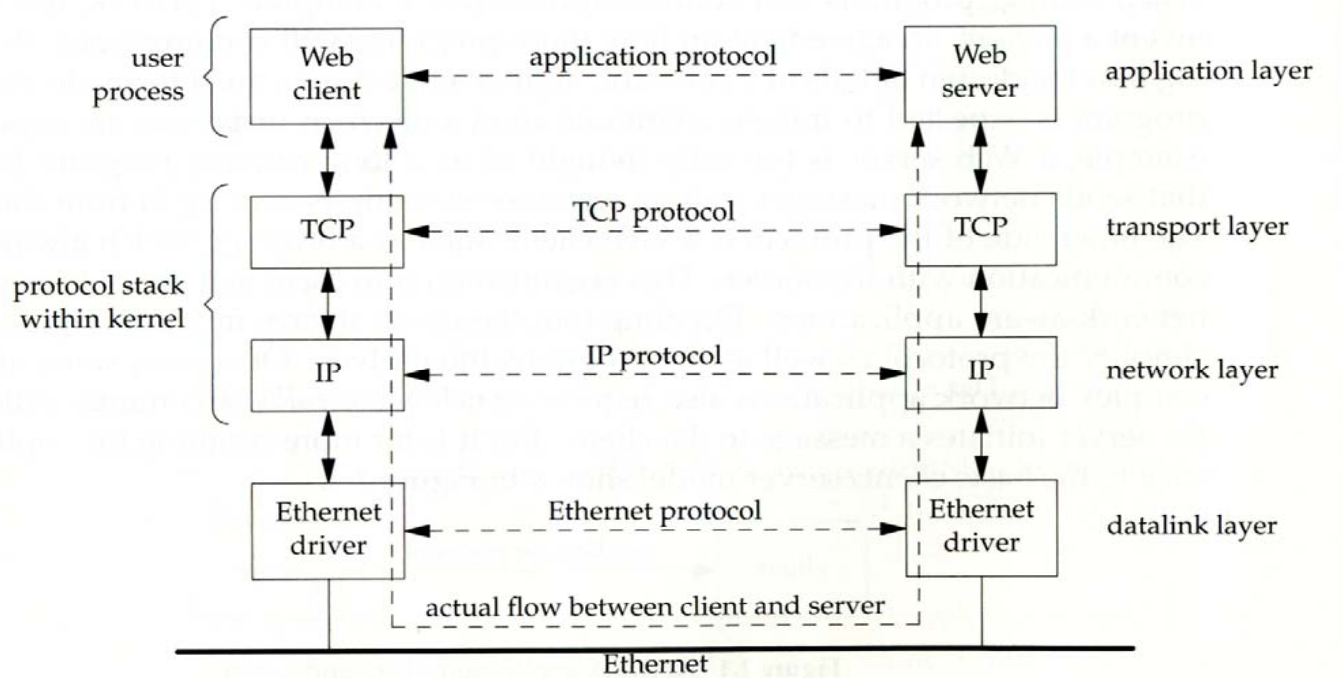



**Figure 1.3** Client and server on the same Ethernet communicating using TCP.

# Client/Server via a WAN

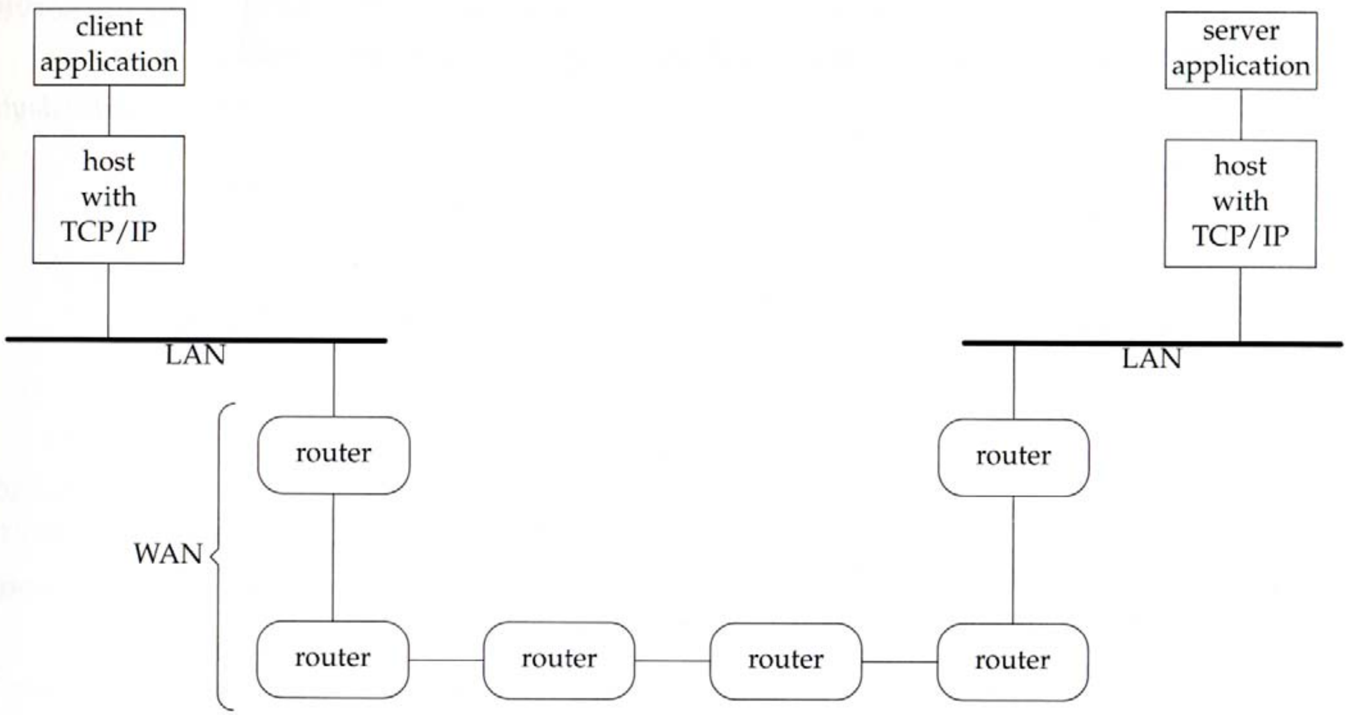



**Figure 1.4** Client and server on different LANs connected through a WAN.

# OSI vs. Internet Protocol Layers

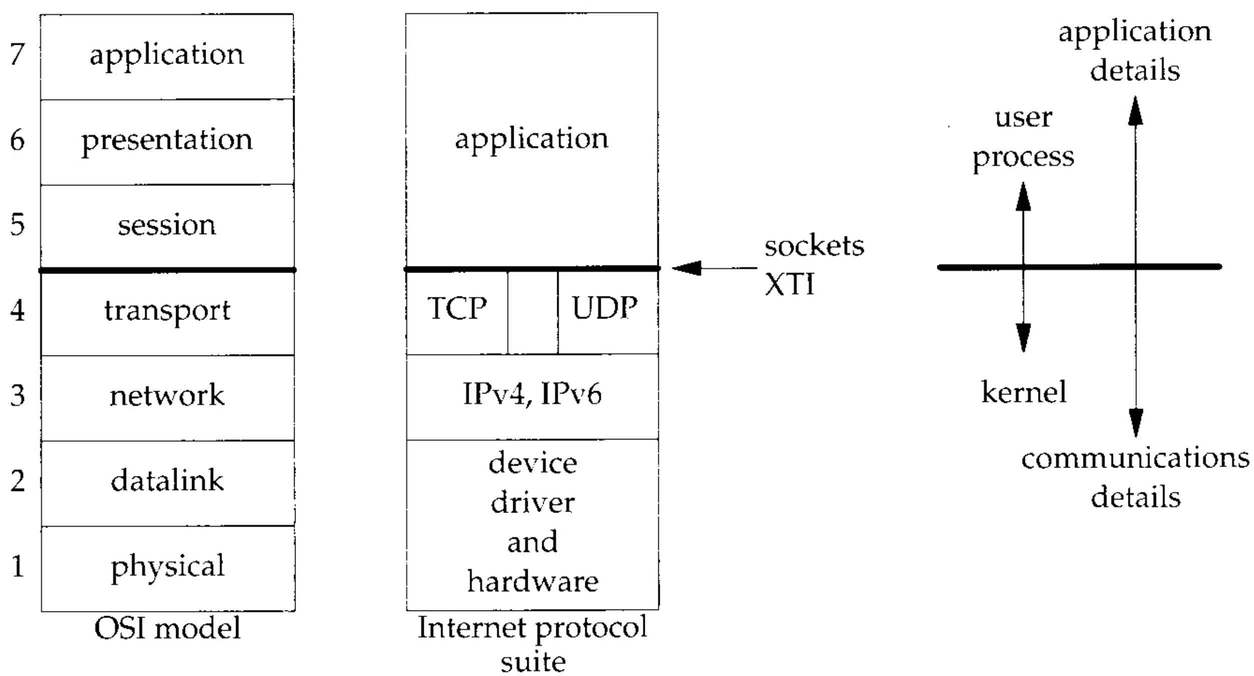
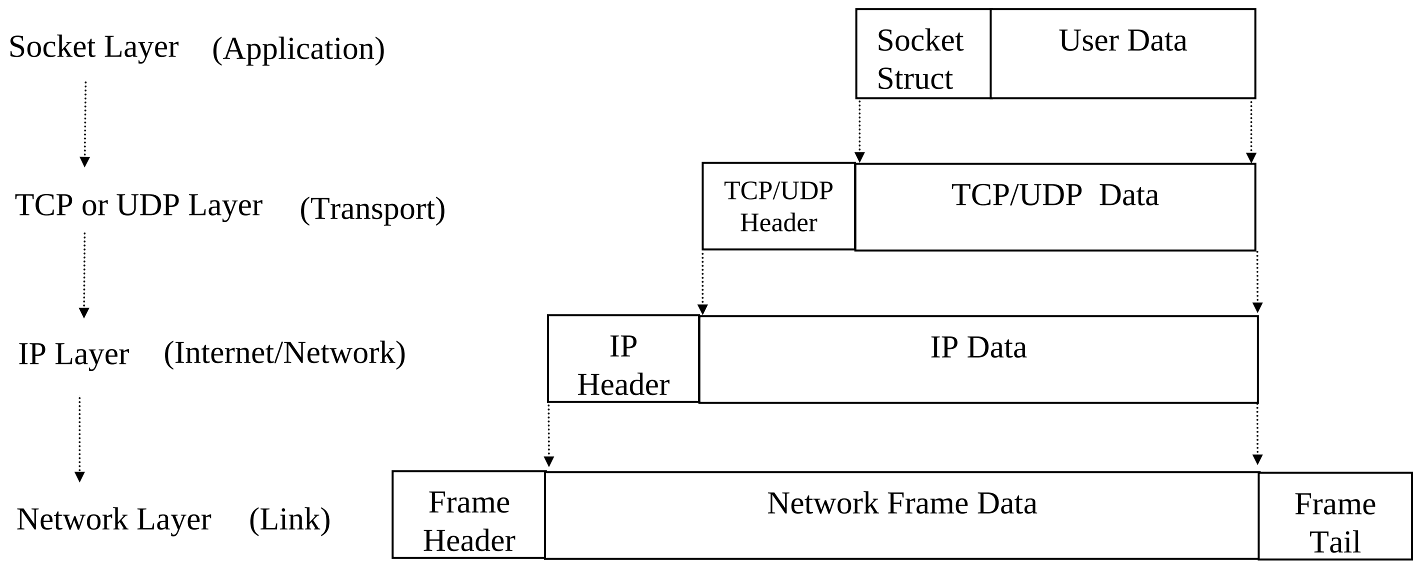



**Figure 1.14** Layers in OSI model and Internet protocol suite.

# Protocol Data and Headers



# What Is A Socket? (1/2)

- A socket is a communication end point.
  - Is equivalent to a computer's network (hardware) interface.
  - Allows a network application to "plug into" the network (not physically, but metaphorically).

# What Is A Socket? (2/2)

- Is a network programming interface.
  - It is used for *interprocess communication* over the network.
  - It is used by a process to communicate with a remote system via a transport protocol.
  - It needs an IP address and a port number.

# Sockets Came From Berkeley UNIX

- Sockets were first introduced in Berkeley UNIX.
  - An extension of the UNIX abstraction of file I/O concepts.
- Now are commonly supported in almost all modern operating systems for inter-systems communications.

# Popular in Client/Server Computing

- Sockets are popularly used in client/server computing.
  - Provides two major types of services:
    - Connection-oriented
    - Connectionless

# Connection-Oriented Services (1/2)

- Implemented on TCP
  - Short for ***T***ransmission ***C***ontrol ***P***rotocol.
  - A connection-oriented protocol.
  - Data transfer unit is known as *segment*.
- An end-to-end connection is established before data exchange can happen.
  - Similar to our phone system.

# Connection-Oriented Services (2/2)

- Data bytes are delivered in sequence.
  - Delivery of data is guaranteed.
- Connection is terminated when finished.
- There are two modes:
  - Iterative (synchronous)
  - Concurrent (asynchronous)

# Connectionless Services (1/2)

- Implemented on UDP
  - Short for ***U***ser ***D***atagram ***P***rotocol.
  - A connectionless protocol.
  - Data transfer unit is known as *datagram*.
- No connection is required before data transfer.
  - Similar to our postal system.

# Connectionless Services (2/2)

- Data bytes may be missing or delivered out-of-order.
- There are also two modes:
  - Iterative (synchronous)
  - Concurrent (asynchronous)

# Sockets Are Bi-directional

- A socket provides a bi-directional communication mechanism.
  - Two way simultaneously.
  - Also know as *full duplex* (FDX) communication.

# Internet Addressing

- A means to identify hosts on the Internet.
- There are two popular ways:
  - Using IP addresses.
  - Using the domain name system (DNS).

# IP Addresses (1/2)

- IP is short for *Internet Protocol*.
- Each host on the Internet is assigned a 32-bit unique address (in current IPv4).
  - An IP address is assigned to a single host only.
  - A host may have more than one IP address (multi-homed host).

# IP Addresses (2/2)

- Dotted representation
  - Internet addresses are represented in the form of four integers separated by decimal points known as *dotted representation*.
  - Examples:
    - 131.91.96.108
    - 131.91.128.73
- For readability by human.

# The Domain Name System (1/2)

- A high-level naming scheme
  - A sequence of characters grouped into sections delimited by decimal points.
    - Labeled in a meaningful way.
  - Examples:
    - earth.cse.fau.edu
    - www.fau.edu

# The Domain Name System (2/2)

- The DNS naming convention is hierarchical.
  - Written in the local-most level first and the top-most level last fashion.
- It is much easier to deal with DNS names than with IP addresses.

# Mapping DNS to IP Addresses

- Delivery of information across the Internet is done by using IP addresses.
  - Need to map DNS names to IP addresses before delivery.
- Three ways:
  - Done at system startup.
  - Via a local table lookup.
  - Going through a *nameserver*

# Port Numbers

- A (protocol) port is an abstraction used by TCP/UDP to distinguish applications on a given host.
  - A port is identified by a 16-bit integer known as the *port number*.
- Three ranges of port numbers:
  - Well-known ports
  - Registered ports
  - Dynamic ports

# Well-known Ports

- Port numbers ranging from 0 to 1,023.
  - A set of pre-assigned port numbers for specific uses.
- Port numbers are managed by ICANN.
  - Short for the ***I***nternet ***C***orporation for ***A***ssigned ***N***ames and ***N***umbers (ICANN)
  - Used to be controlled solely by IANA (***I***nternet ***A***ssigned ***N***umbers ***A***uthority).

# Some Well-known Ports

| Port | Keyword | Description |
|---|---|---|
| 0 | | Reserved |
| 7 | ECHO | Echoes a received datagram to the sender |
| 13 | DAYTIME | Returns the date and the time |
| 20 | FTP-DATA | File Transfer Protocol (data) |
| 21 | FTP | File Transfer Protocol (control) |
| 22 | SSH | Secure Shell |
| 23 | TELNET | Terminal Connection |
| 25 | SMTP | Simple Mail Transport Protocol |
| 53 | DNS | Domain Name Server |
| 67 | BOOTP | Bootstrap Protocol |
| 79 | FINGER | Finger |
| 80 | HTTP | HyperText Transfer Protocol |
| 101 | HOSTNAME | NIC Host Name Server |
| 103 | X400 | X.400 Mail Service |
| 110 | POP3 | Post Office Protocol Vers. 3 |

# Registered Ports

- Port numbers ranging from 1,024 to 49,151.
- Not assigned or controlled by ICANN; however their uses need to be registered via an ICANN-accredited registrar to prevent duplications.

# Dynamic Ports

- Port numbers ranging from 49,152 to 65,535.
- Neither assigned or registered. They can be used by anyone.
  - These are ephemeral ports.
  - Also known as private ports.

# Socket Programming

- To use a socket, one needs a structure to hold address and its associated port number information.

- A generic socket format:

  (*address family*, *address in the family*)

  - Another name for *family* is *domain*.

# Generic Socket Address Structure

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

- Note: This generic socket structure is primarily for declaring variables. “cast” is needed in the actual use of a socket address structure.

# A Popular BSD-derived Socket Implementation (1/3)

```
struct  sockaddr_in {
    sa_family_t  sin_family;   /* address family: AF_XXX */
    in_port_t  sin_port;        /* 16-bit protocol port number */
    struct  in_addr sin_addr;  /* IP addr in NW byte order */
    char  sin_zero[8];          /* unused, set to zero */
}
```

- Note: One may encounter PF_XXX occasionally. It is the same as AF_XXX at present, but is expected to be phased out later.

# A Popular BSD-derived Socket Implementation (2/3)

Where

- sa_family_t sin_family usually holds the value either AF_INET or AF_UNIX.
- in_port_t sin_port is a 16-bit TCP or UDP port number.
  - In need of htons() to convert to network byte order.

# A Popular BSD-derived Socket Implementation (3/3)

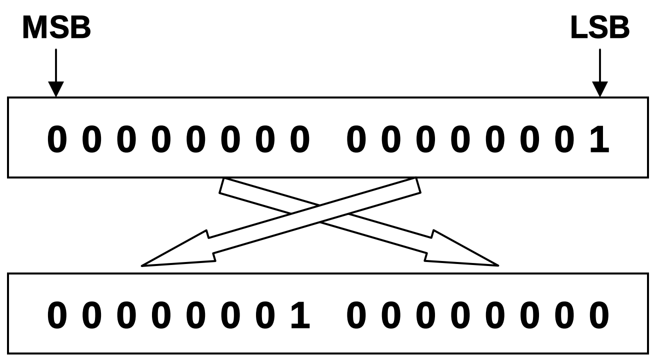
- in_addr sin_addr contains a 32-bit IPv4 address.
- Structure for in_addr:

```
struct in_addr {
    in_addr_t s_addr;     /* 32-bit IPv4 address */
                          /* in network byte order */
}
```

  - In need of htonl() to convert to network byte order.

# Network Byte Order

- Different systems may store numbers in different byte orders internally.
  - *For example, Sparc* machines is *big-endian*, and *i386* is *little-endian.* Taking 1 as an example,




  - Big-endian:
  - Little-endian

- Network byte order uses big-endian ordering.

# Socket Types

| Family | Description |
| --- | --- |
| SOCK_STREAM | stream socket (TCP) |
| SOCK_DGRAM | datagram socket (UDP) |
| SOCK_SEQPACKET | sequenced packet socket (SCTP) |
| SOCK_RAW | raw socket (talk to IP directly) |

# Two Examples

- A connectionless example
  - Algorithms for server and client.
  - An implementation in C.
- A connection-oriented example
  - Algorithms for server and client.
  - An implementation in C.

# Example 1: Connectionless

- To illustrate a simple connectionless client/server example.
  - One server (iterative), multiple clients.
  - The server echoes back requests from clients, one client request at a time.
  - A client sends user request to server and displays response received from server.
  - Programs: echo_seru.c & echo_cliu.c
- It is implemented on UDP.

# Server Algorithm (connectionless)

a) Create a socket.
b) Bind to a predefined address for the service desired.
c) Wait for a datagram to arrive from a client.
d) Send response back to originating client.
e) Return to c) for next client.

# Client Algorithm (connectionless)

a) Create a socket.
b) Send a datagram to server.
c) Wait for response from server.
d) Return to b) for more datagrams.
e) Close the socket when no more datagram to send.

# Example 2: Connection-oriented

- To illustrate a simple connection-oriented client/server example.
  - Similar to the previous one: The server echoes back requests from clients, and a client displays server response.
  - However, a connection is established before data exchange can happen.
  - Programs: echo_ser.c & echo_cli.c
- It is implemented on TCP.

# Server Algorithm (connection-oriented)

a) Create a socket.

b) Bind to a predefined address for the service desired.

c) Place the socket in passive mode.

d) Accept the next connection request from a client.

e) Read a request, process the request, and send back the results.

f) Close the connection when done with a client.

g) Return to d) for next client.

# Client Algorithm (connection-oriented)

a) Create a socket.

b) Connect the socket to the desired server.

c) Send a request, and wait for the response.

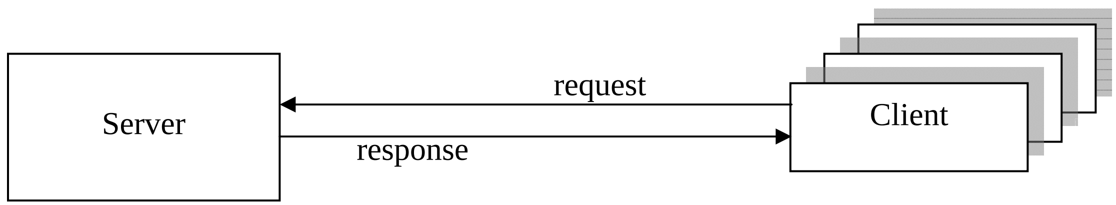
d) Repeat c) until done.

e) Notify server of intention to terminate.

  - May close R/W end either separately or together at the same time.

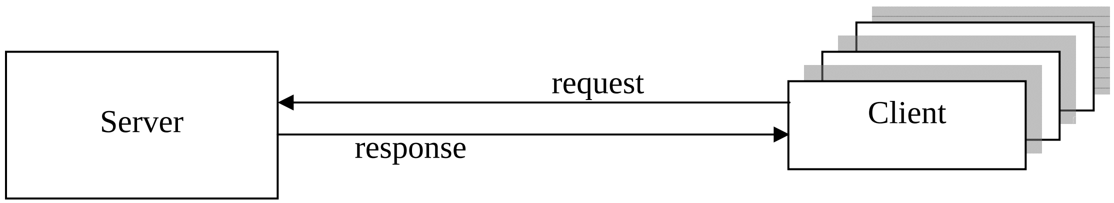
f) Close the socket (connection) when done.

# Connectionless: Functions Used




| Server | Client |
| --- | --- |
| socket() | socket() |
| bind() | bind() → optional |
| recvfrom() → blocks | sendto() |
| sendto() | recvfrom() → blocks |

# Connection-oriented: Functions Used




| Server | Client |
| --- | --- |
| socket() | socket() |
| bind() | connect() → blocks |
| listen() | |
| accept() → blocks | |
| read() | write() |
| write() | read() |
| close() | close() |

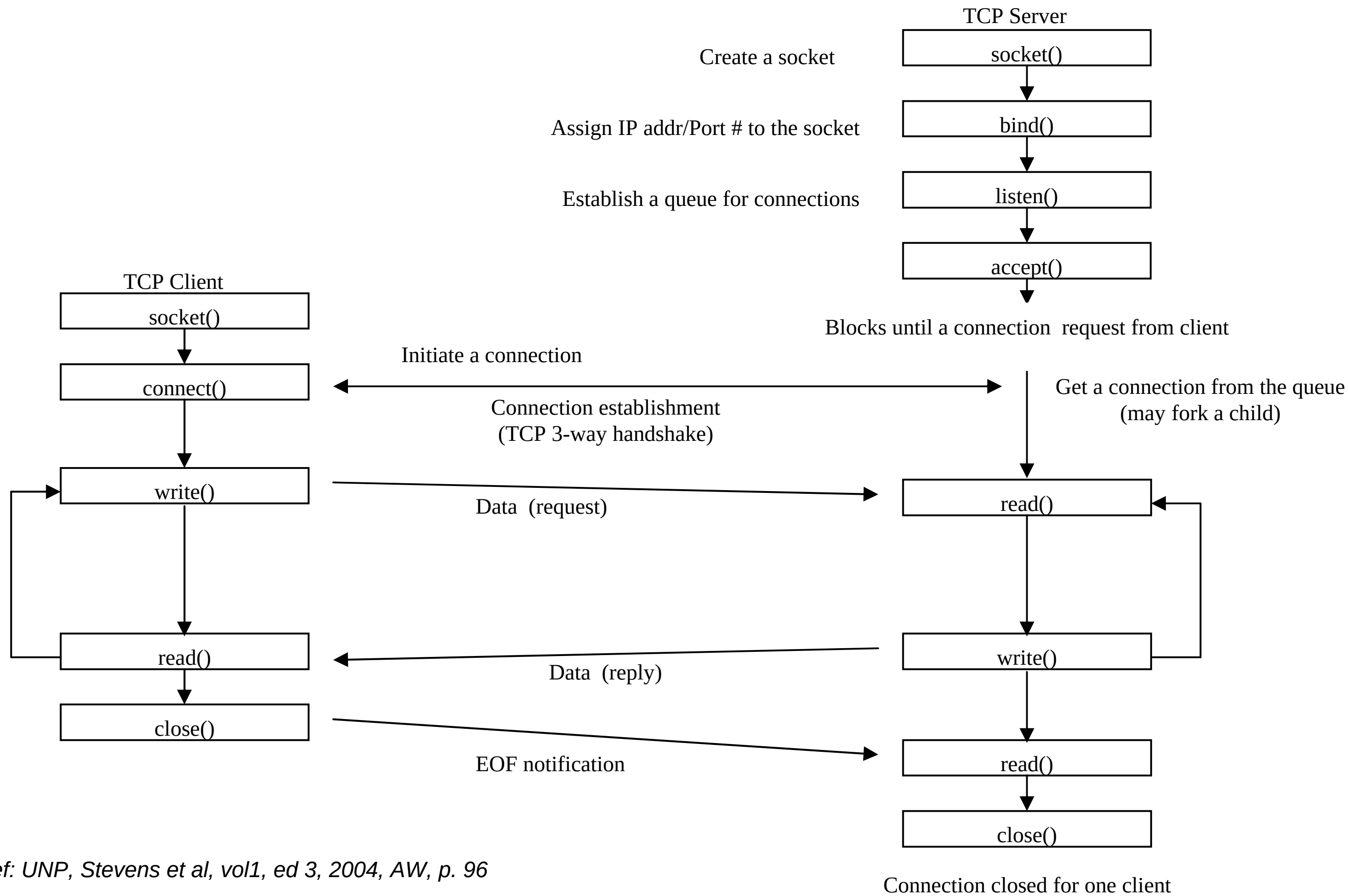
# R&W on a Closed TCP Socket

- In a TCP connection, a *write* to a disconnected socket will generate SIGPIPE.
  - This can be dealt with a proper signal handler.
- A *read* from socket will return 0 if the socket is closed.

# A close() Call for TCP/UDP

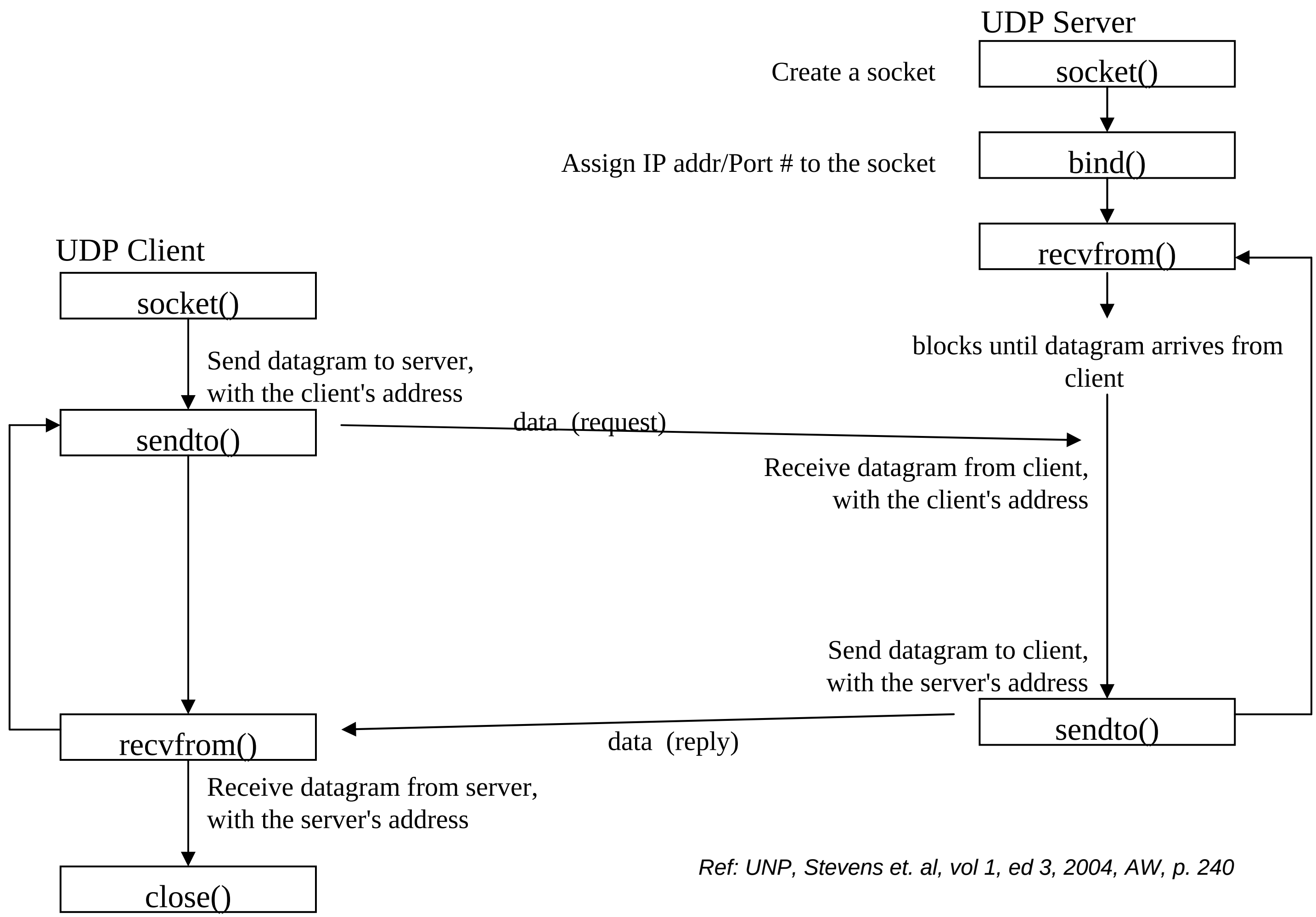
- If the socket is for SOCK_STREAM
  - The kernel will deliver all data received before terminating the connection.
  - close() will block until all outstanding data delivered to the receiving process.
- If the socket is for SOCK_DGRAM
  - The socket is closed immediately.

# TCP Client/Server Socket Functions




*Ref: UNP, Stevens et al, vol1, ed 3, 2004, AW, p. 96*

# UDP Client/Server Socket Functions




*Ref: UNP, Stevens et. al, vol 1, ed 3, 2004, AW, p. 240*

# Some Relevant Socket System Calls and Header Files

int socket(int *family*, int *type*, int *protocol*);

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

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

int accept(int *sockfd*, struct sockaddr **cliaddr*, socklen_t **cliaddrlen*);

int connect(int *sockfd*, struct sockaddr **servaddr*, socklen_t **servaddrlen*);

ssize_t recvfrom(int *sockfd*, void **buff*, size_t *len*, int *flags*, struct sockaddr **from*,
    socklen_t **fromlen*);

ssize_t sendto(int *sockfd*, void **msg*, size_t *len*, int *flags*, struct sockaddr **to*,
    socklen_t *tolen*);

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

int shutdown(int *sockfd*, int *howto*);

<sys/socket.h>

<netinet/in.h>

<netdb.h>

# Some References

- To download textbook source code
  **http://www.unpbook.com/src.html**
- A tutorial on Networking Programming using Sockets
  **http://beej.us/guide/bgnet/output/print/bgnet_USLetter.pdf**
- Networking Programming FAQs
  **http://www.faqs.org/faqs/unix-faq/socket/**
  **http://www.uni-giessen.de/faq/archiv/unix-faq.socket/msg00000.html**
- Some coding examples
  **http://www.xcf.berkeley.edu/~ali/K0D/UNIX/Networking/**

# Reading Assignment

- Scan Chapters 1, 4, and 8