Socket Network Programming

Reading:
Advanced Programming in the UNIX Environment, Chapter 11

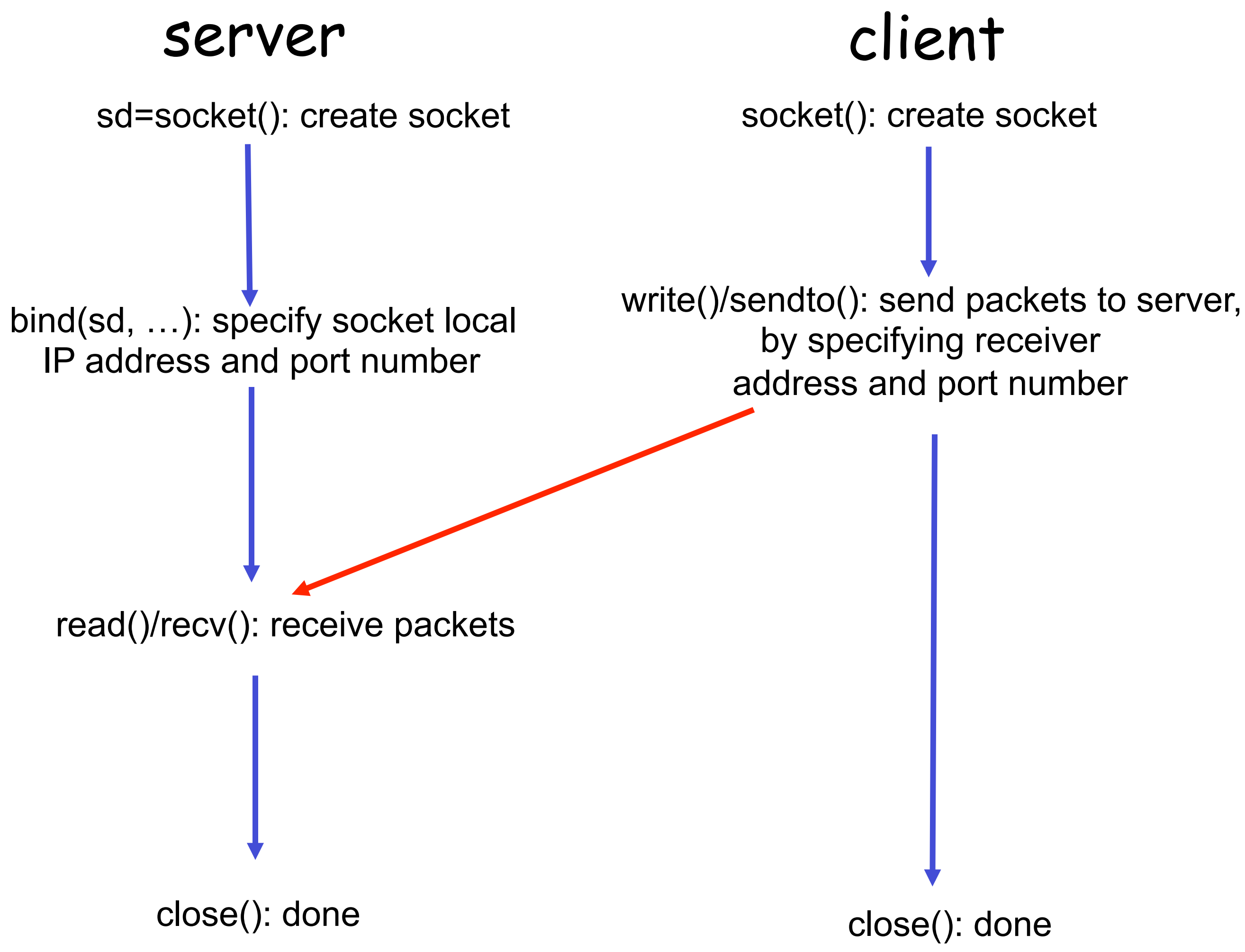
Connection-oriented vs. Connectionless

- Connectionless protocols
  - Use UDP as the transport protocol
  - Need to deal with unreliability at the application level
- Connection-oriented protocols
  - Use TCP as the transport protocol (reliable)
  - May use more resources (e.g., three-way handshake, half-open connections)

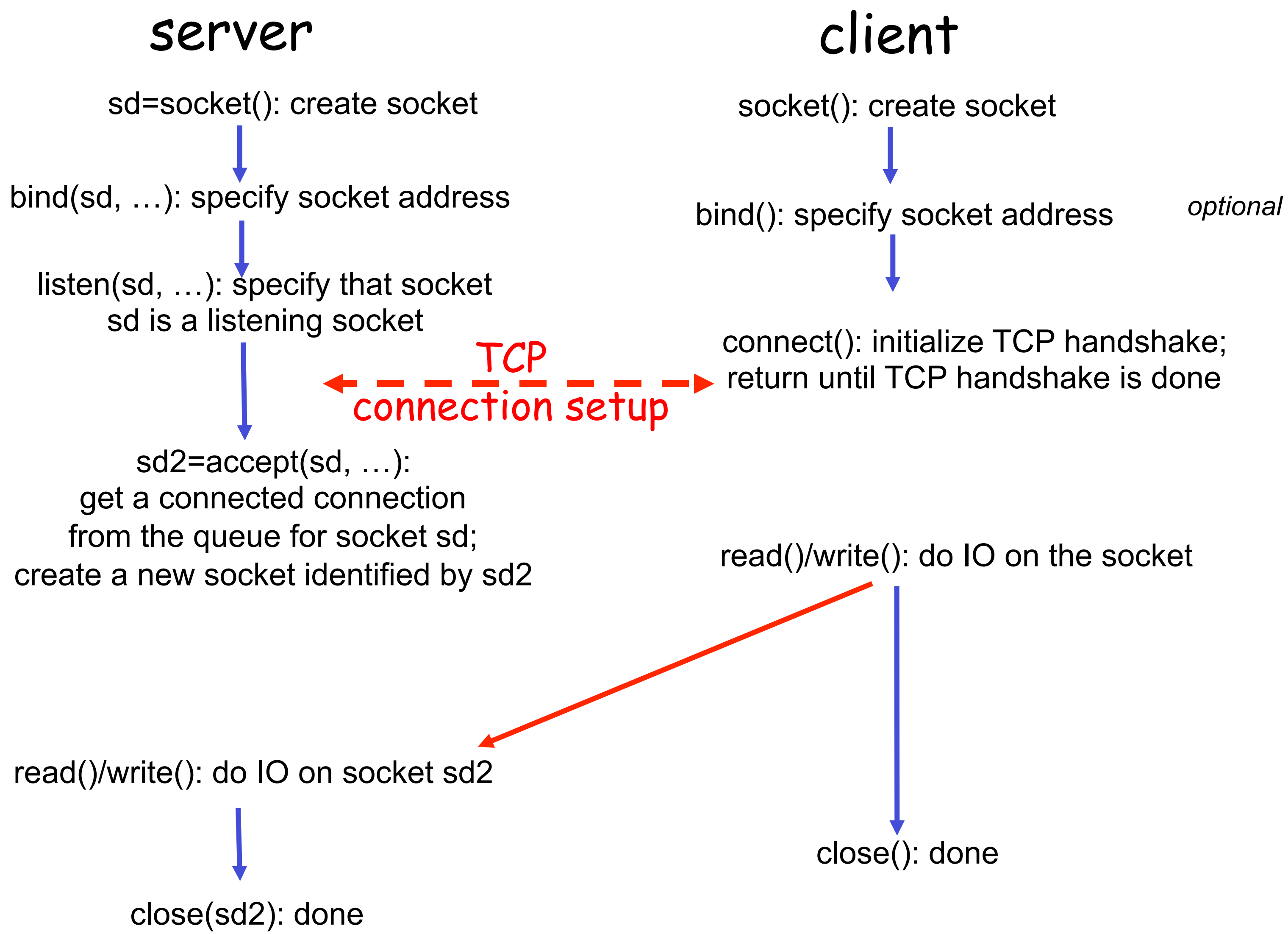
Stateful *vs.* Stateless Protocols

- Stateful
  - A protocol (and therefore a server) can imply a "history" of previous commands
  - The server has to maintain some state associated with the clients
    - Connection-oriented descriptors
    - Client-oriented handles (do not change across connections)
  - Need to deal with client/server failures
- Stateless
  - Each protocol interaction is independent
  - Might create overhead

Connectionless UDP: Big Picture



Connection-oriented: Big Picture



Unix Programming: Mechanism

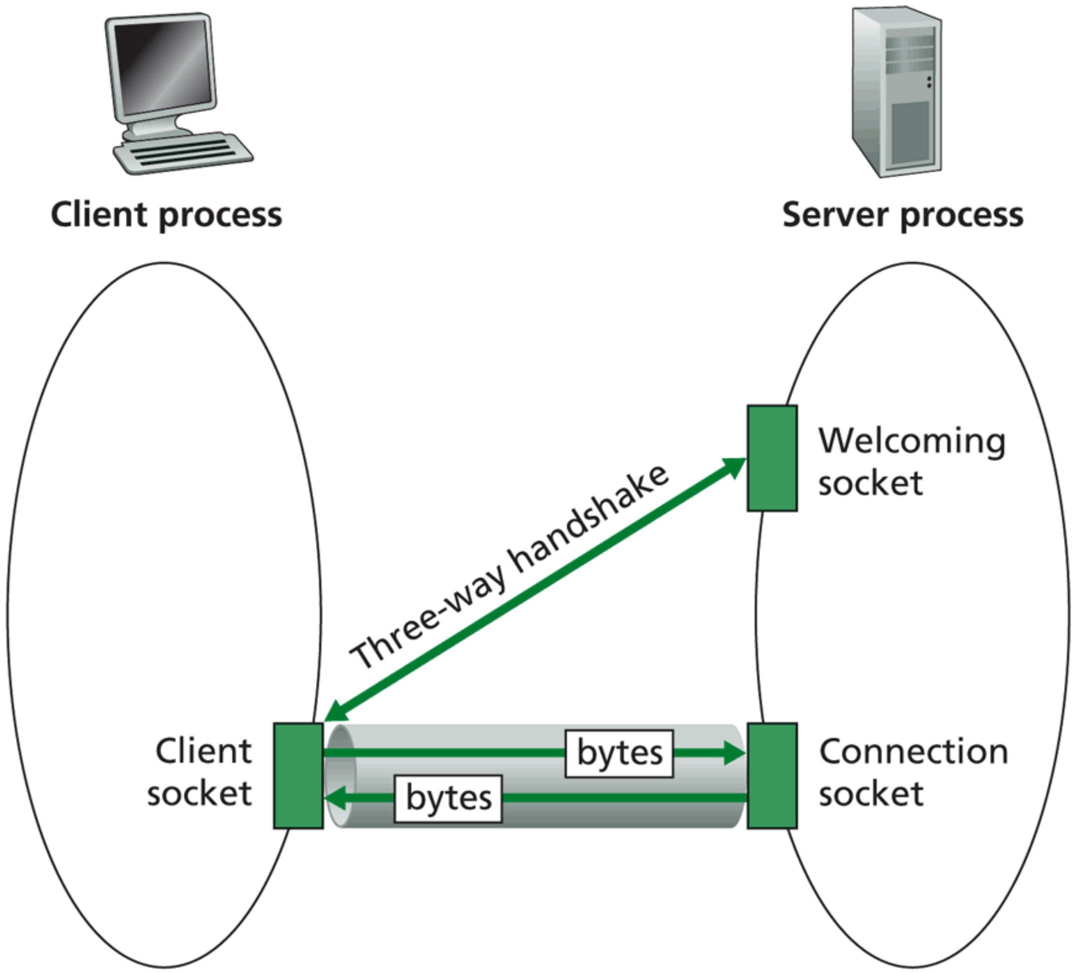
- UNIX system calls and library routines (functions called from C/C++ programs)

- %man 2 <function name>

- A word on style: **check all return codes**

```
if ((code = syscall()) < 0) {
        perror("syscall");
}
```

Big Picture: Connection-Oriented TCP



TCP Connection-Oriented Demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- Host uses all four values to direct segment to appropriate socket
  - Server can easily support many simultaneous TCP sockets: different connections/sessions are automatically separated into different sockets

Socket API

- API used to access the network
- A socket is an abstraction of a communication endpoint
  - Can be used to access different network protocols (not only TCP/IP)
  - A socket's characteristics are determined by calling specific functions
  - A socket can be accessed as a file
    - Similar to some I/O APIs, e.g., read, write, close

Create a Socket

- int socket(int domain, int type, int protocol)
  - domain: PF_UNIX, PF_INET, PF_INET6
  - type: SOCK_STREAM, SOCK_DGRAM
  - protocol: normally 0, can be other values if there are multiple protocols available (IPPROTO_TCP, IPPROTO_UDP)
  - Return value: the file descriptor of the socket or -1 in case of errors
- UDP and TCP are accessed using the same primitives but the semantics are different

Creating Sockets Examples

```
if ((sockfd = socket(AF_INET,SOCK_STREAM,0)) {
    perror("socket");
    exit(1);
}
```

Socket Descriptors

- Similar to file descriptor
  - Internal data structure specifies
    - Protocol used (or family – PF_INET)
    - Type of service (connection-less or connection-oriented)
    - IP addresses involved
    - Ports involved
- Per-process resource

Bind Sockets to Addresses

- A socket can be bound to a specific IP address and port
- int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen)
  - sockfd: socket descriptor
  - my_addr: socket address (usually of sockaddr_in type)
  - addrlen: address structure length
  - Return value: 0 for success, -1 in case of error
- Same semantics for TCP and UDP

Socket Addresses

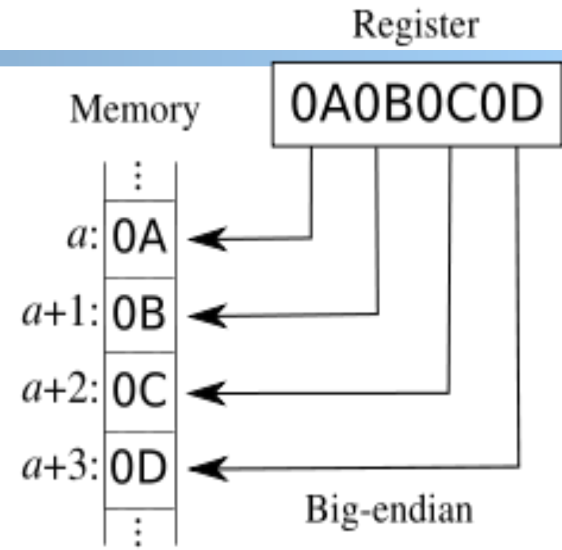
```
struct sockaddr {
    u_short sa_family; /* family */
    char sa_data[14]; /* address data */
};

struct sockaddr_in { /* a TCP endpoint */
    u_int16_t sin_family; /* address family: AF_INET */
    u_int16_t sin_port; /* port in network byte order */
    struct in_addr sin_addr; /* internet address */
};

struct in_addr {
    u_int32_t s_addr; /* address in network byte order */
};
```

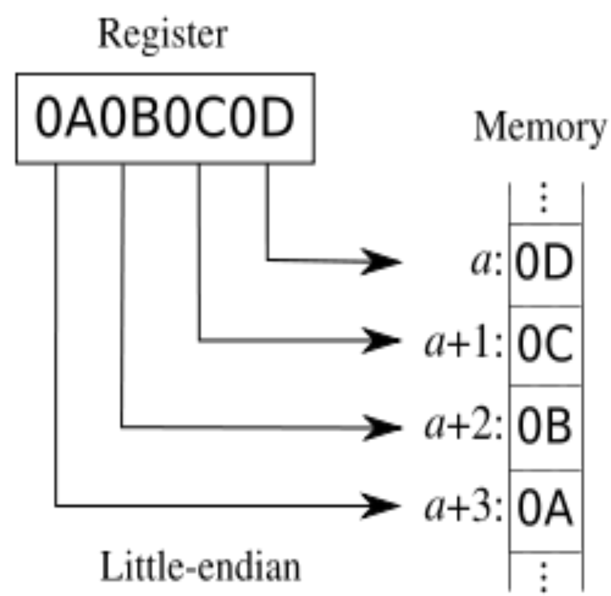
Internet Addresses and Ports

- sin_port
  - 16 bits
  - 0-1024 reserved for system
  - well-known ports are important
  - If you specify 0, the OS picks a port
- s_addr
  - 32 bits
  - INADDR_ANY for any local interface address

Network Byte Order

- *Big-endian* is network byte order
  - Most-Significant-Byte at lowest memory location
  - E.g., ARM, PowerPC (not the PPC970/G5), DEC Alpha, SPARC V9, MIPS, PA-RISC and IA64
- *Little-endian*
  - Least-Significant-Byte at lowest memory location
  - E.g., 6502, Z80, x86, and VAX
- htons, htonl, ntohs, ntohl





Internet Addresses and Ports: Example

```
struct sockaddr_in myaddr;

bzero( (char*)myaddr, sizeof(myaddr) );
myaddr.sin_family = AF_INET;
myaddr.sin_port = htons(80); /* bind to HTTP port*/
myaddr.sin_addr.s_addr = htonl(INADDR_ANY); /* any address*/

if ( (bind(sockfd, (struct sockaddr*)&myaddr, sizeof(myaddr)) < 0 ) {
    perror("bind");
    exit(1);
}
```

Passive Connection-oriented Sockets

- If a socket is used to receive TCP connections, it is necessary to bind it to an address and to specify that the socket is passive
- int listen(int s, int backlog)
  - s: socket descriptor
  - backlog: maximum length of the queue of pending connections (used to be the number of open/half-open connections, now only the open ones that are ready to be accepted are counted)
  - Return value: 0 in case of success, -1 in case of error

Accept Connections

- int accept(int s, struct sockaddr *addr, socklen_t *addrlen)
  - s: socket descriptor
  - addr: structure that will be filled with the parameters of the client (maybe NULL)
  - addrlen: length of the structure in input, it is filled with the actual size of the data returned
  - Return value: a new socket file descriptor in case of success, -1 in case of errors

Accept TCP Connections

- A call to accept() blocks the caller until a request is sent
- When a connection is made, accept() returns a new socket, associated with a specific client (virtual circuit)
- A virtual circuit is identified by:
  - <srcIP, srcPort, dstIP, dstPort>
- There are no two identical virtual circuits at one time in the whole Internet

Create Connections

- Clients use connect() to open a connection to a specific IP address/port
- int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen)
  - sockfd: socket descriptor
  - serv_addr: destination address
  - addrlen: size of the structure
  - Return value: 0 in case of success, -1 in case of errors

Connecting with TCP/UDP

- TCP
  - connect() starts the three-way handshake
- UDP
  - Nothing really happens, but the socket can only be used to send/receive datagrams to/from the specified address

Read/Write to a Socket

- read()/write() of the file interface for connected-oriented
- Socket specific system call
  - send()/sendto()/sendmsg()
  - recv()/recvfrom()/recvmsg()

Read from a socket by using read()

```
#include <unistd.h>
ssize_t read(int sockfd, void *buf, size_t count);
```

- read up to count from the socket
- return value: -1 if an error occurs; 0 if end of file; otherwise number of bytes read
- what are the possible outcomes of this system call?

Write to a socket by using write()

```
#include <unistd.h>
ssize_t write(int sockfd, const void *buf, size_t
    count);
```

- write up to count to the socket
- return value: -1 if an error occurs; otherwise number of bytes write
- what are the possible outcomes of this system call?

Send to a Socket

```
#include <sys/types.h>
#include <sys/socket.h>

int send(int sd, const void *msg, size_t len, int flags);
int sendto(int sd, const void *msg, size_t len, int flags, const
    struct sockaddr *to, socklen_t tolen);
int sendmsg(int sd, const struct msghdr *msg, int flags)
```

- return value: -1 if an error occurs; otherwise the number of bytes sent

sendmsg(): scatter and collect

```
struct msghdr {
    void          *msg_name;       // peer address
    socklen_t      msg_namelen;    // address length
    struct iovec *msg_iov;         // io vector
    size_t         msg_iovlen;     // io vector length
    void           *msg_control;
    socklen_t      msg_controllen;
    int            msg_flags;
};
struct iovec {
    void          *iov_base;
    size_t        iov_len;
};
```

Receive from a Socket

```
#include <sys/types.h>
#include <sys/socket.h>
int recv(int sd, void *buf, size_t len, int flags);
int recvfrom(int sd, void *buf, size_t len, int flags, struct sockaddr
    *from, socklen_t fromlen);
int recvmsg(int sd, struct msghdr *msg, int flags);
```

- return value: -1 if an error occurs; otherwise the number of bytes received
- will block the process if there is no data

Close Sockets

- int close(int fd)
  - fd: socket descriptor
  - Return value: 0 in case of success, -1 in case of errors
- int shutdown(int s, int how)
  - Can be used to close a socket partially
  - s: connected socket
  - how:
    - SHUT_RD, further receptions are disallowed
    - SHUT_WR, further transmissions are disallowed
    - SHUT_RDWR, further receptions and transmissions are disallowed
  - Return value: 0 in case of success, -1 in case of errors

Closing Semantics

- TCP
  - Close: FIN/ACK in both directions
  - Shutdown: FIN/ACK in one direction
- UDP
  - Close: Don't send anything. Just deallocate structure

Data Exchange in TCP/UDP

- TCP
  - recv/read will return a chunk of data
  - Not necessarily the data sent by means of a single send/ write operation on the other side
- UDP
  - recv/read will always return a datagram
  - If message size > buffer, fills buffer, discards rest

Support Routines: Address from and to String Formats

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
```

presentation format to network format

```c
int inet_pton(int af, const char *src, void *dst);
```

return value: positive if successful

```c
const char *inet_ntop(int af, const void *src, char *dst, size_t cnt);
```

return value: NULL is error

Note: inet_addr() deprecated!

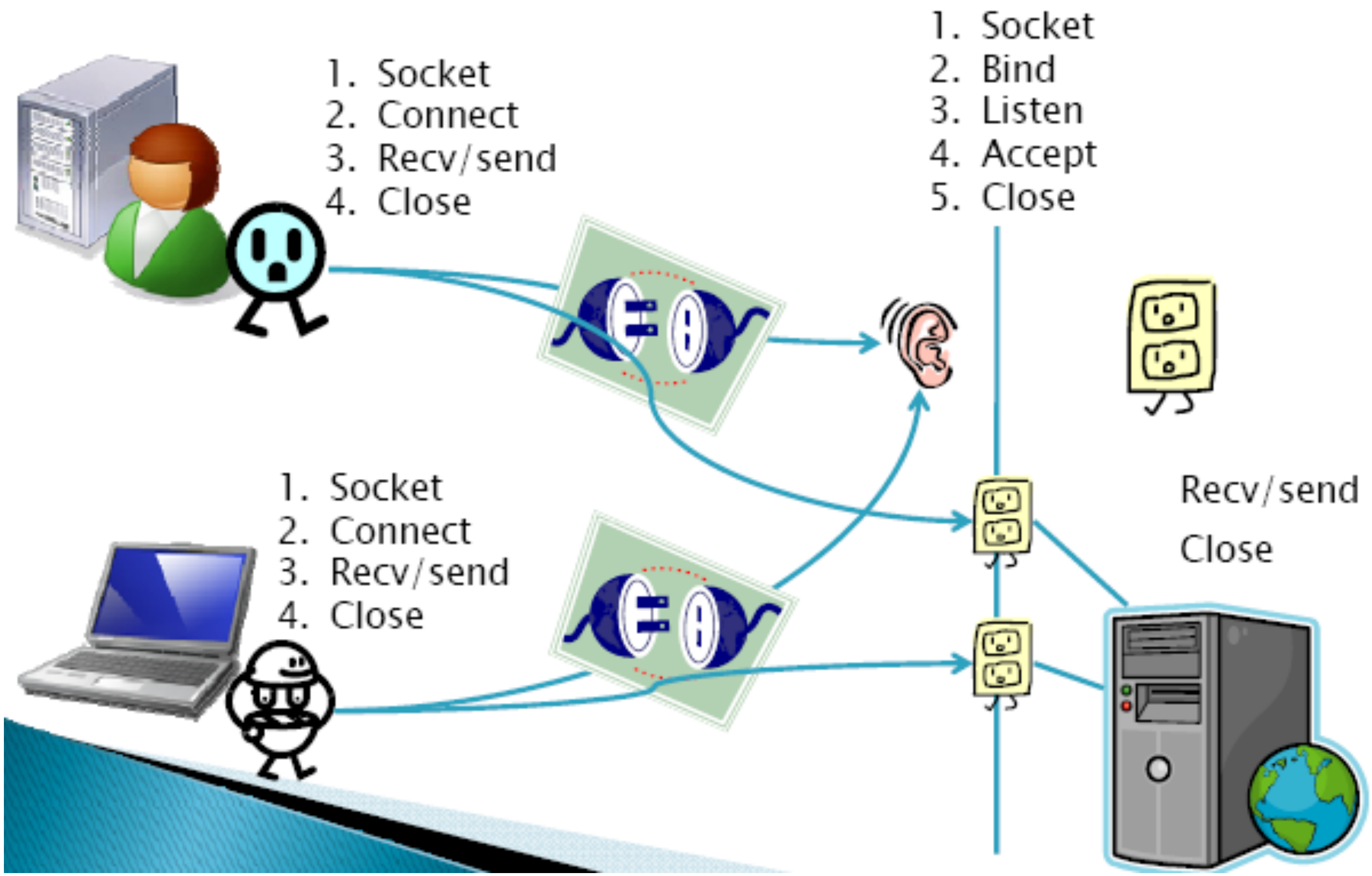
Support Routines: Network/Host Order

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);

unsigned long int ntohl(unsigned long int networklong);
unsigned short int ntohs(unsigned short int networkshort);
```

DNS Service

```
#include <netdb.h>
extern int h_errno;

struct hostent *gethostbyname(const char *name);

struct hostent {
    char   *h_name;        // official name
    char **h_aliases;      // a list of aliases
    int      h_addrtype;
    int      h_length;
    char **h_addr_list;
}
#define h_addr h_addr_list[0]
- return value: NULL if fails
```

Summary: TCP Client and Server



Summary: Berkeley Sockets

Primitive	Meaning
Socket	Create a new communication end point
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections; given queue size
Accept	Block the caller until a connection attempt arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data from the connection
Close	Release the connection

Summary: Socket Programming

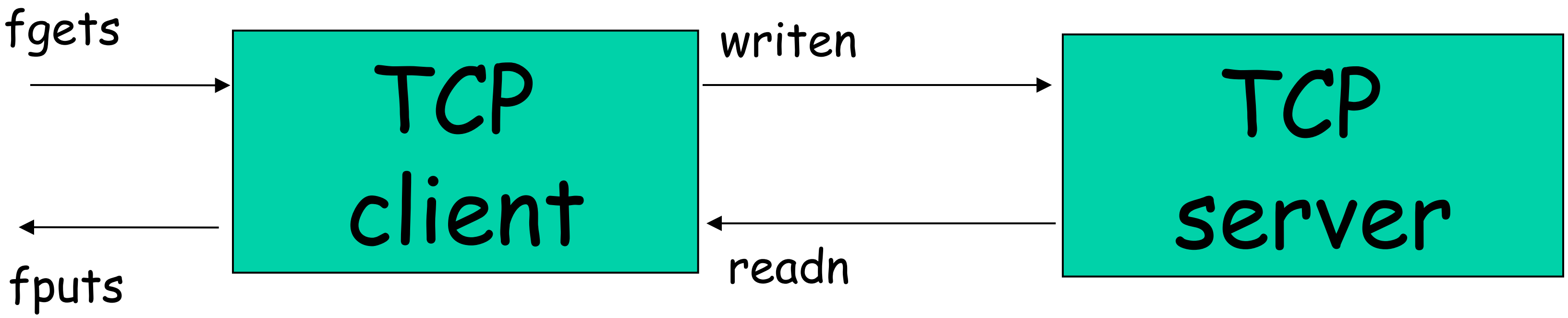
- $ man 2 <name>
- System calls (functions)
  - int socket(int domain, int type, int protocol);
  - int bind(int sd, struct sockaddr *my_addr, socklen_t addrlen);
  - int listen(int sd, int backlog);
  - int connect(int sd, const struct sockaddr *serv_addr, socklen_t addrlen);
  - int accept(int sd, struct sockaddr *peer_addr, socklen_t addrlen);

  - read(int sockfd, void *buf, size_t count);
  - write(int sockfd, const void *buf, size_t count)

Asynchronous Network Programming

(C/C++)

A Relay TCP Client: telnet-like Program



Method 1: Process and Thread

- Process
  - fork()
  - waitpid()
- Thread: light weight process
  - pthread_create()
  - pthread_exit()

pthread

```
void main()
{
 char  recvline[MAXLINE + 1];
  ss = new socketstream(sockfd);

  pthread_t  tid;
  if (pthread_create(&tid, NULL,
     copy_to, NULL)) {
    err_quit("pthread_creat()");
  }

  while  (ss->read_line(recvline,
     MAXLINE) > 0) {
    fprintf(stdout, "%s\n", recvline);
  }
}
```

```
void *copy_to(void *arg) {
  char sendline[MAXLINE];

  if (debug) cout << "Thread
     create()!" << endl;
  while (fgets(sendline,
     sizeof(sendline), stdin))
    ss->writen_socket(sendline,
     strlen(sendline));

  shutdown(sockfd, SHUT_WR);
  if (debug) cout << "Thread done!"
     << endl;

  pthread_exit(0);
}
```

Method 2: Asynchronous I/O (Select)

- select: deal with blocking system call

  int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)

  FD_CLR(int fd, fd_set *set);

  FD_ZERO(fd_set *set);

  FD_ISSET(int fd, fd_set *set);

  FD_SET(int fd, fd_set *set);

Method 3: Signal and Select

- signal: events such as timeout

Examples of Network Programming

- Library to make life easier
- Four design examples
  - TCP Client
  - TCP server using select
  - TCP server using process and thread
  - Reliable UDP
- Warning: It will be hard to listen to me reading through the code. Read the code.

Example 2: A Concurrent TCP Server Using Process or Thread

- Get a line, and echo it back
- Use select()
- For how to use process or thread, see later
- Check the code at:
  http://jingyu.dyndns.org/~jzhou/courses/F11.Networks/examples-c-socket/tcpserver

- Are there potential denial of service problems with the code?

Example 3: A Concurrent HTTP TCP Server Using Process/Thread

- Use process-per-request or thread-per-request
- Check the code at:

http://jingyu.dyndns.org/~jzhou/courses/F11.Networks/examples-c-socket/simple_httpd

Example 4: Reliable UDP

- How to implement timeout?
- Use SIGALARM
- Use setjmp()
- Check the code at:

http://jingyu.dyndns.org/~jzhou/courses/F10.Networks/examples-c-socket/udptimeout

Optional Slides

Writing High Performance Servers: Major Issues

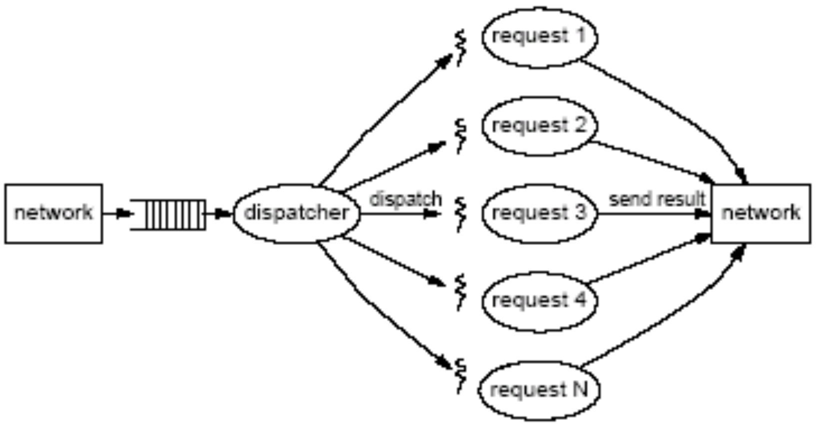
- Many socket/IO operations can cause a process to block, e.g.,
  - `accept`: waiting for new connection;
  - `read` a socket waiting for data or close;
  - `write` a socket waiting for buffer space;
  - I/O `read/write` for disk to finish

- Thus a crucial perspective of network server design is the concurrency design (non-blocking)
  - for high performance
  - to avoid denial of service

- Concurrency is also important for clients!

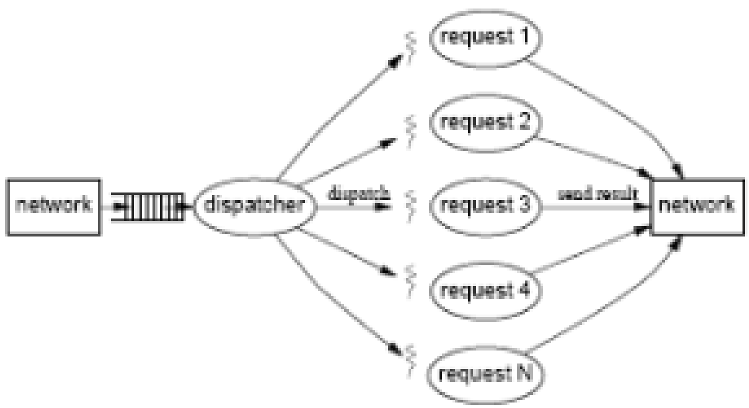
Writing High Performance Servers: Using Multi-Threads

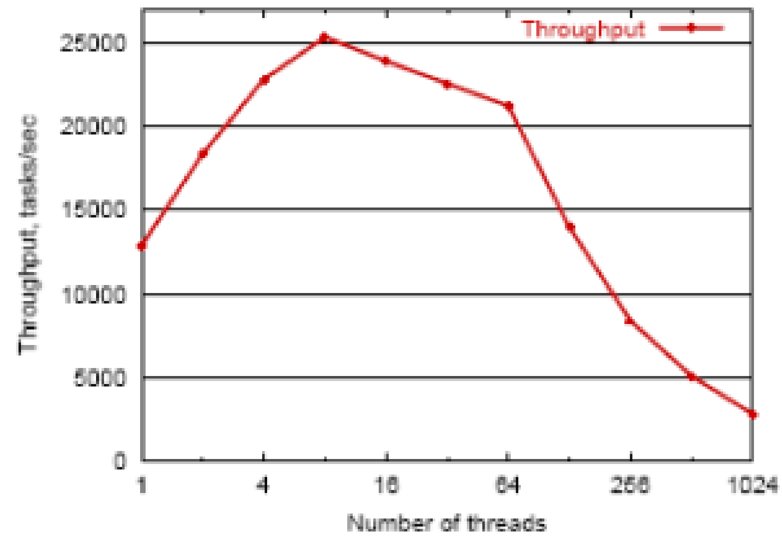
- Using multiple threads
  - So that only the flow processing a particular request is blocked
  - Java: extends Thread or implements Runnable interface
  - C/C++: use pthread package



Example: a Multi-threaded WebServer, which creates a thread for each request

Problems of Multi-Thread Server

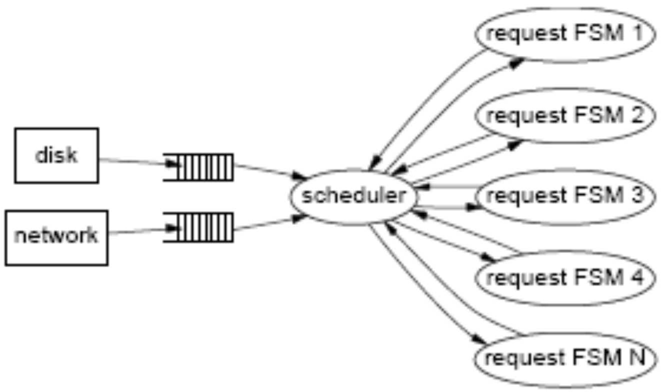


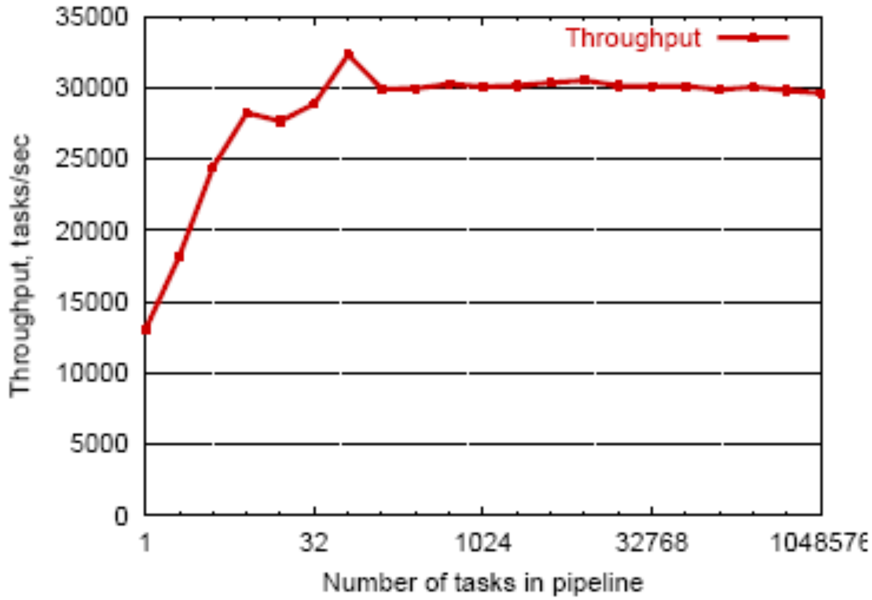


(937 MHz x86, Linux 2.2.14, each thread reading 8KB file)

- High resource usage, context switch overhead, contended locks
- Too many threads → throughput meltdown, response time explosion
- In practice: bound total number of threads

Event-Driven Programming





- Event-driven programming, also called asynchronous i/o
- Using Finite State Machines (FSM) to monitor the progress of requests
- Yields efficient and scalable concurrency
- Many examples: Click router, Flash web server, TP Monitors, etc.

- Java: asynchronous i/o
  - for an example see: http://www.cafeaulait.org/books/jnp3/examples/12/

Problems of Event-Driven Server

- Difficult to engineer, modularize, and tune

- Little OS and tool support: ‘‘roll your own’’

- No performance/failure isolation between FSMs

- FSM code can never block (but page faults, garbage collection may still force a block)
  - thus still need multiple threads