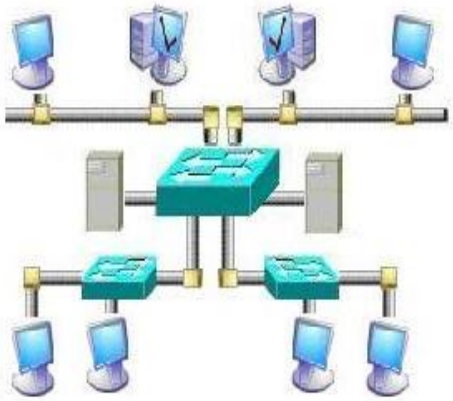
MC833
Pragramação em Redes de Computadores

Primeiro Semestre 2018

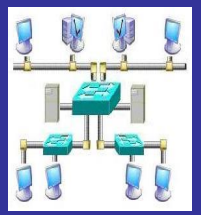
Prof. Edmundo R. M. Madeira



INSTITUTO DE COMPUTAÇÃO
Universidade Estadual de Campinas
UNICAMP

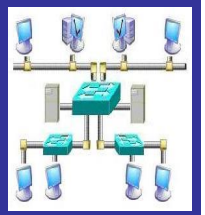
Programa da Disciplina

- Tecnologias de Comunicação:
  1. Sockets TCP
  2. Sockets UDP
  3. RMI
  4. Web Services



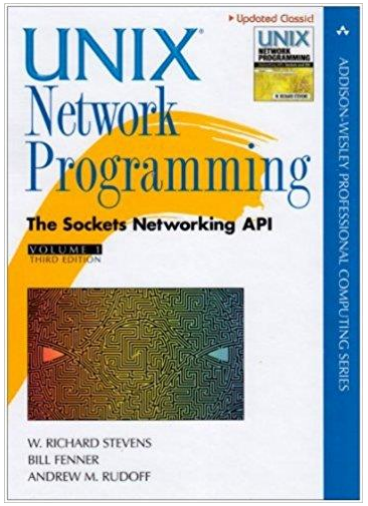
Critério de Avaliação

- Três projetos com relatórios técnicos de comparação (pesos iguais).
- Todos os projetos devem ter notas superiores ou iguais a 5. Se não, a média final é o menor valor entre 4,9 e a média dos três projetos.

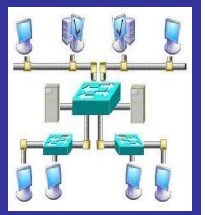


Bibliografia

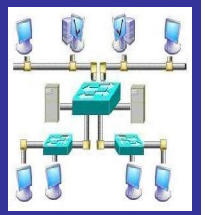
- Stevens, R. "Unix Network Programming – Networking APIs: Sockets and XTI" – Vol. 1, Third Edition, Addison, 2003.
  - **BIMECC 005.43 St47u**



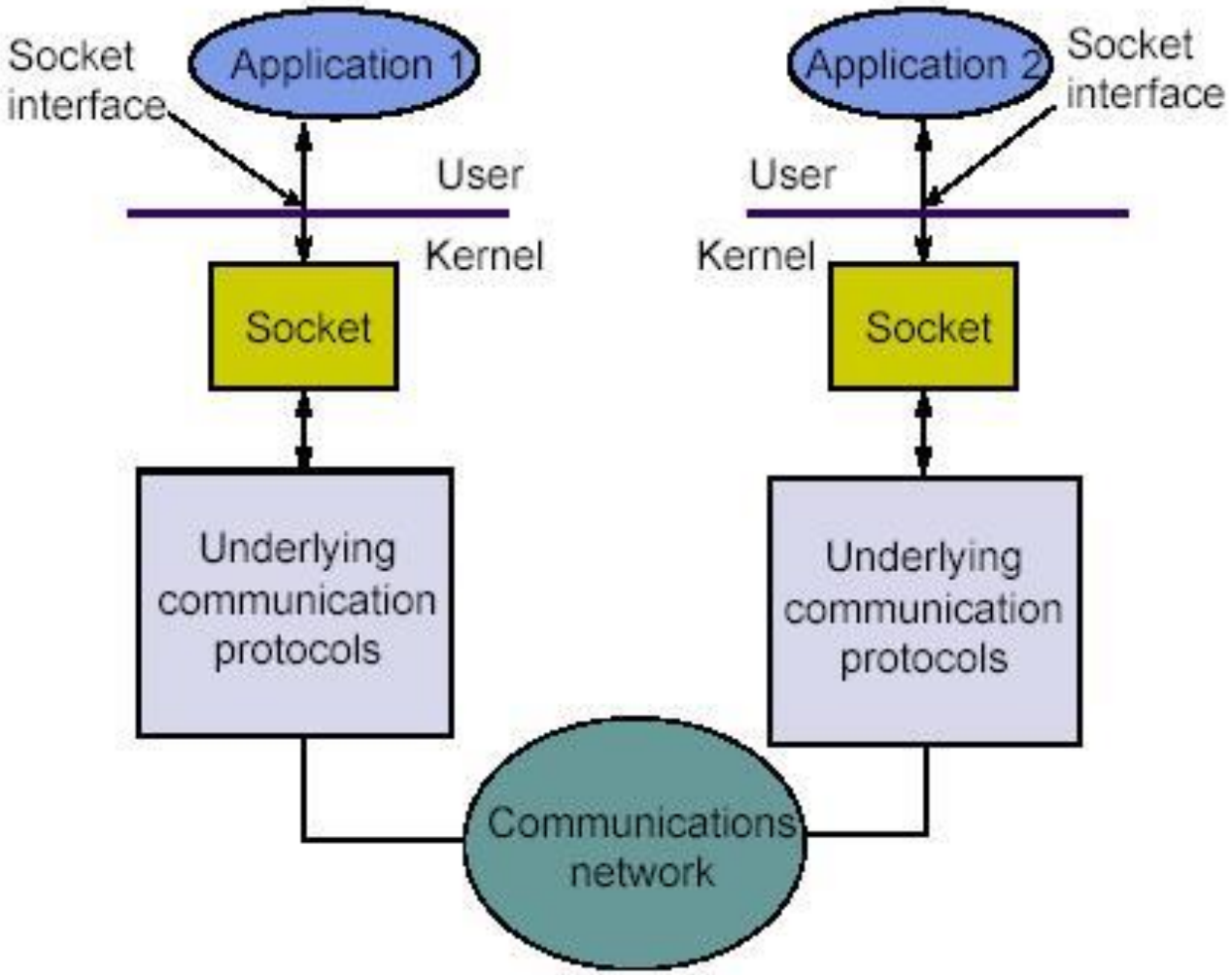
- Beej's Guide to Network Programming: http://beej.us/guide/bgnet/
- RMI: https://docs.oracle.com/javase/tutorial/rmi/index.html
- Web Services: https://docs.oracle.com/javaee/6/tutorial/doc/bnayk.html

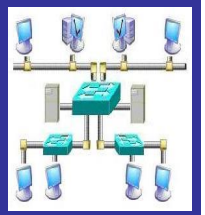


Sockets



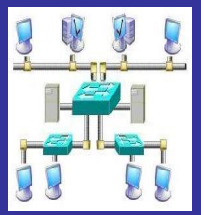
Basics





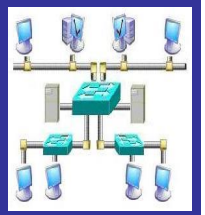
Basics

- End point determined by two things:
  - Host address: IP address is *Network Layer*
  - Port number: is *Transport Layer*
- Two end-points determine a connection:
  socket pair
  - ex: 206.62.226.35,p21 + 198.69.10.2,p1500
  - ex: 206.62.226.35,p21 + 198.69.10.2,p1499



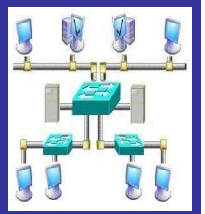
Ports

- Numbers (vary in BSD, Solaris):
  - 0-1023 “reserved”, must be root
  - 1024 – 49151 (registered with IANA)
  - 49152 – 65535 “ephemeral”
- /etc/services:
  - ftp 21/tcp
  - telnet 23/tcp
  - finger 79/tcp
  - snmp 161/udp



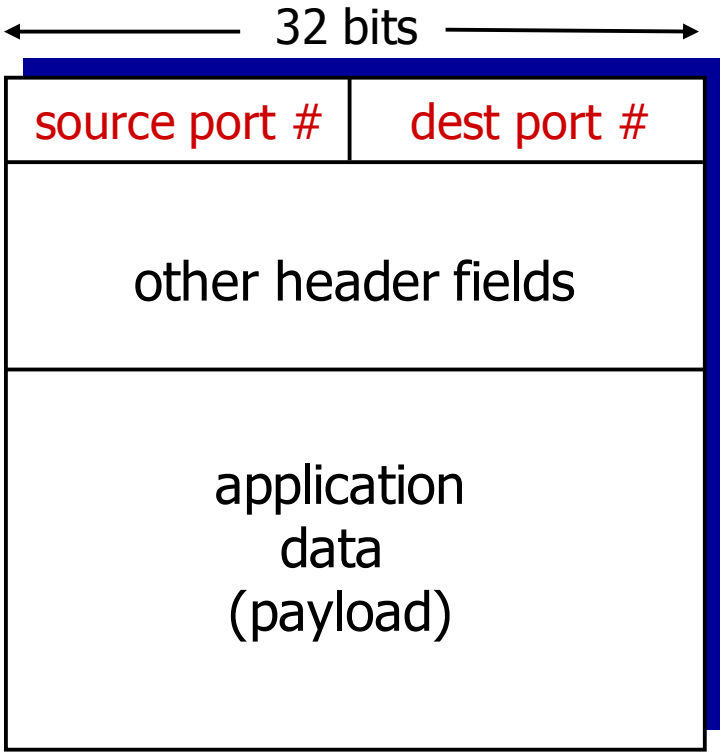
Transport Layer

- UDP: User Datagram Protocol
  - no acknowledgements
  - no retransmissions
  - out of order
  - connectionless
- TCP: Transmission Control Protocol
  - reliable (in order, all arrive, no duplicates)
  - flow control
  - connection
  - duplex

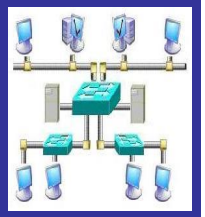


How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries one transport-layer segment
  - each segment has source, destination port number
- host uses *IP addresses & port numbers* to direct segment to appropriate socket



TCP/UDP segment format



Connectionless demultiplexing

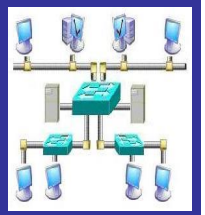
- *recall:* created socket has host-local port #:

```
DatagramSocket
mySocket1                = new
DatagramSocket(12534);
```

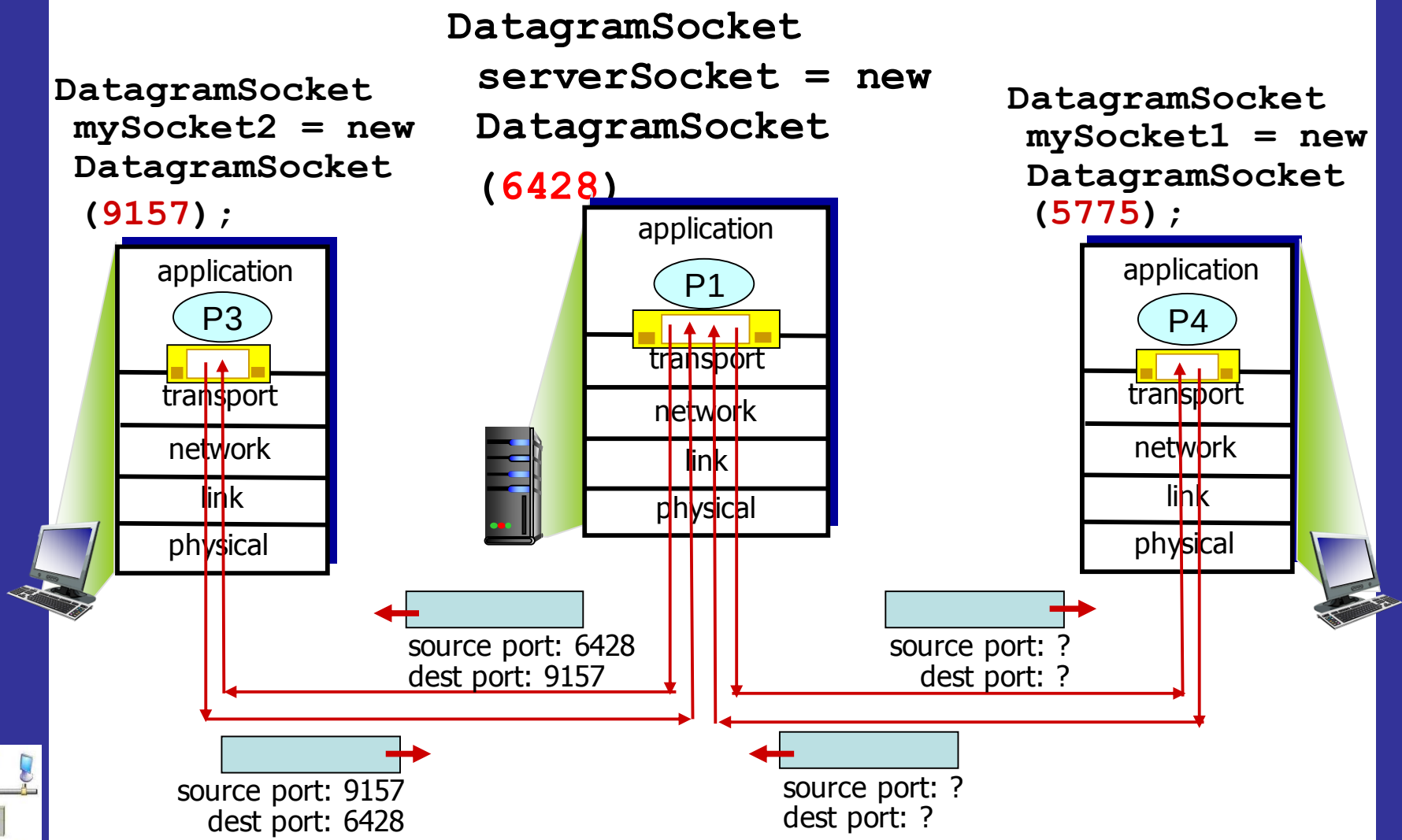
- *recall:* when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

- when host receives UDP segment
  - checks destination port # in segment
  - directs UDP segment to socket with that port #

IP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

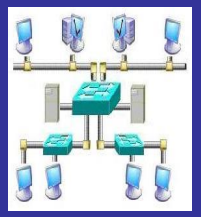


Connectionless demux: example

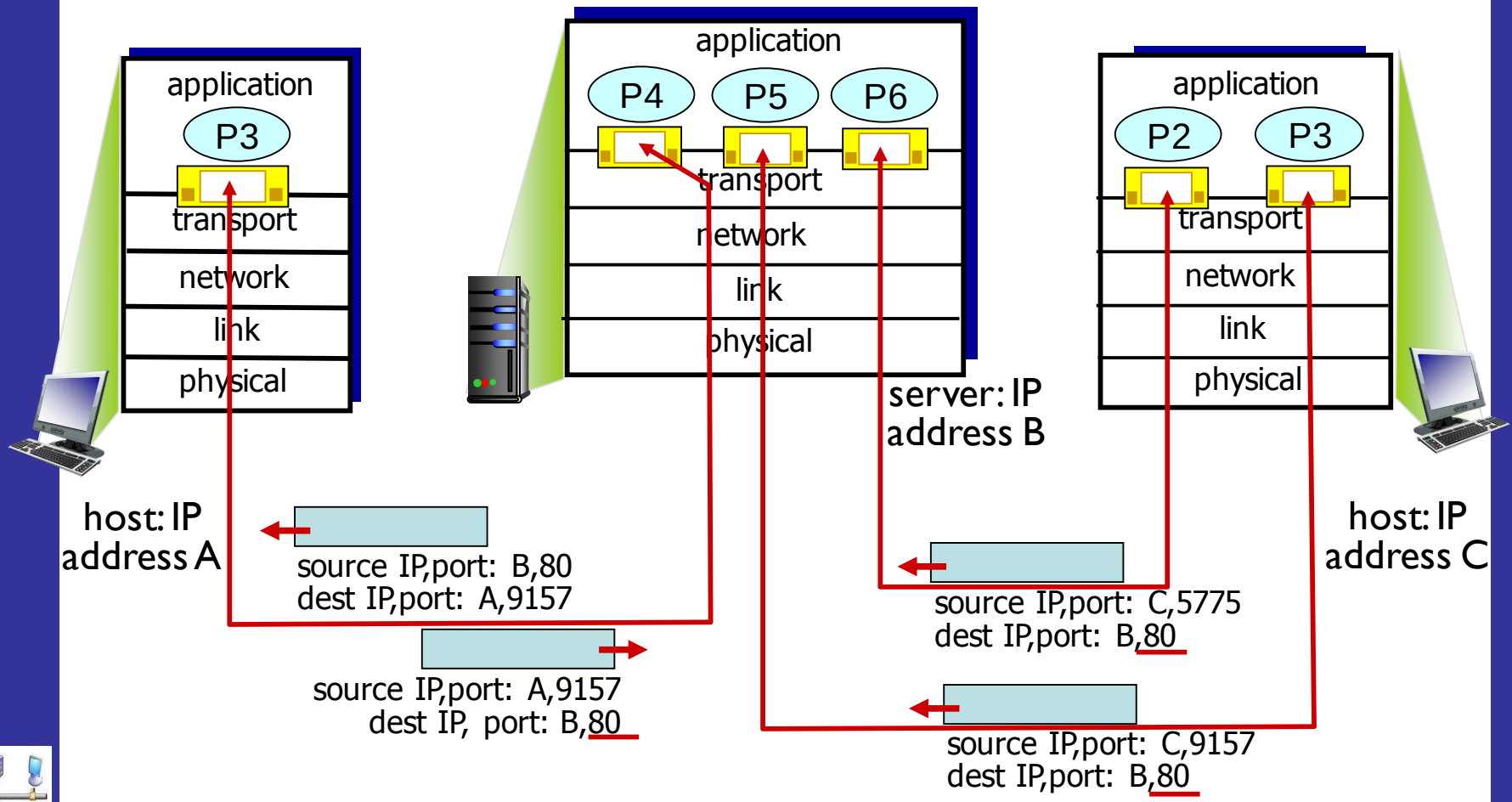


Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate sock
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting clien
  - non-persistent HTTP will have different socket for each request

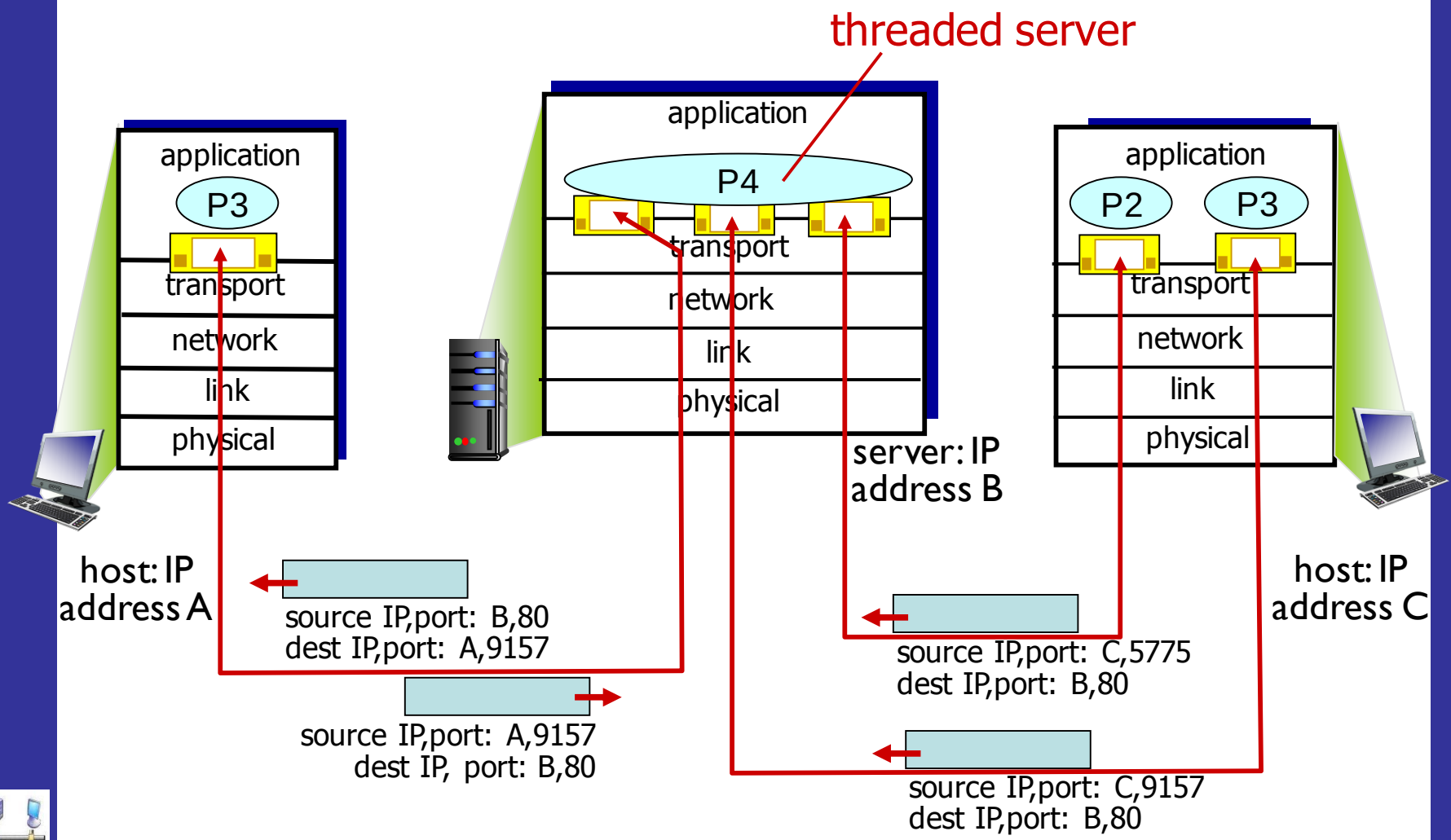


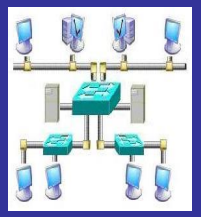
Connection-oriented demux: example



three segments, all destined to IP address: B, dest port: 80 are demultiplexed to *different* sockets

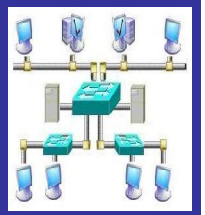
Connection-oriented demux: example





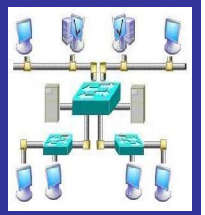
Socket Address Structure


```
struct addrinfo {
    int              ai_flags;      // AI_PASSIVE, AI_CANONNAME, etc.
    int              ai_family;     // AF_INET, AF_INET6, AF_UNSPEC
    int              ai_socktype;   // SOCK_STREAM, SOCK_DGRAM
    int              ai_protocol;   // use 0 for "any"
    size_t           ai_addrlen;    // size of ai_addr in bytes
    struct sockaddr *ai_addr;       // struct sockaddr_in or _in6
    char            *ai_canonname; // full canonical hostname
    struct addrinfo  *ai_next;      // linked list, next node
};
struct sockaddr {
    unsigned short sa_family;    // address family, AF_xxx
    char           sa_data[14]; // 14 bytes of protocol address
};
```



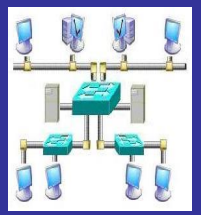
Socket Address Structure (IPv4)


```
struct in_addr {
    uint32_t s_addr; // that's a 32-bit int (4 bytes)
};

struct sockaddr_in {
    short int          sin_family;  // Address family, AF_INET
    unsigned short int sin_port;    // Port number
    struct in_addr     sin_addr;    // Internet address
    unsigned char      sin_zero[8]; // Same size as struct sockaddr
};
```

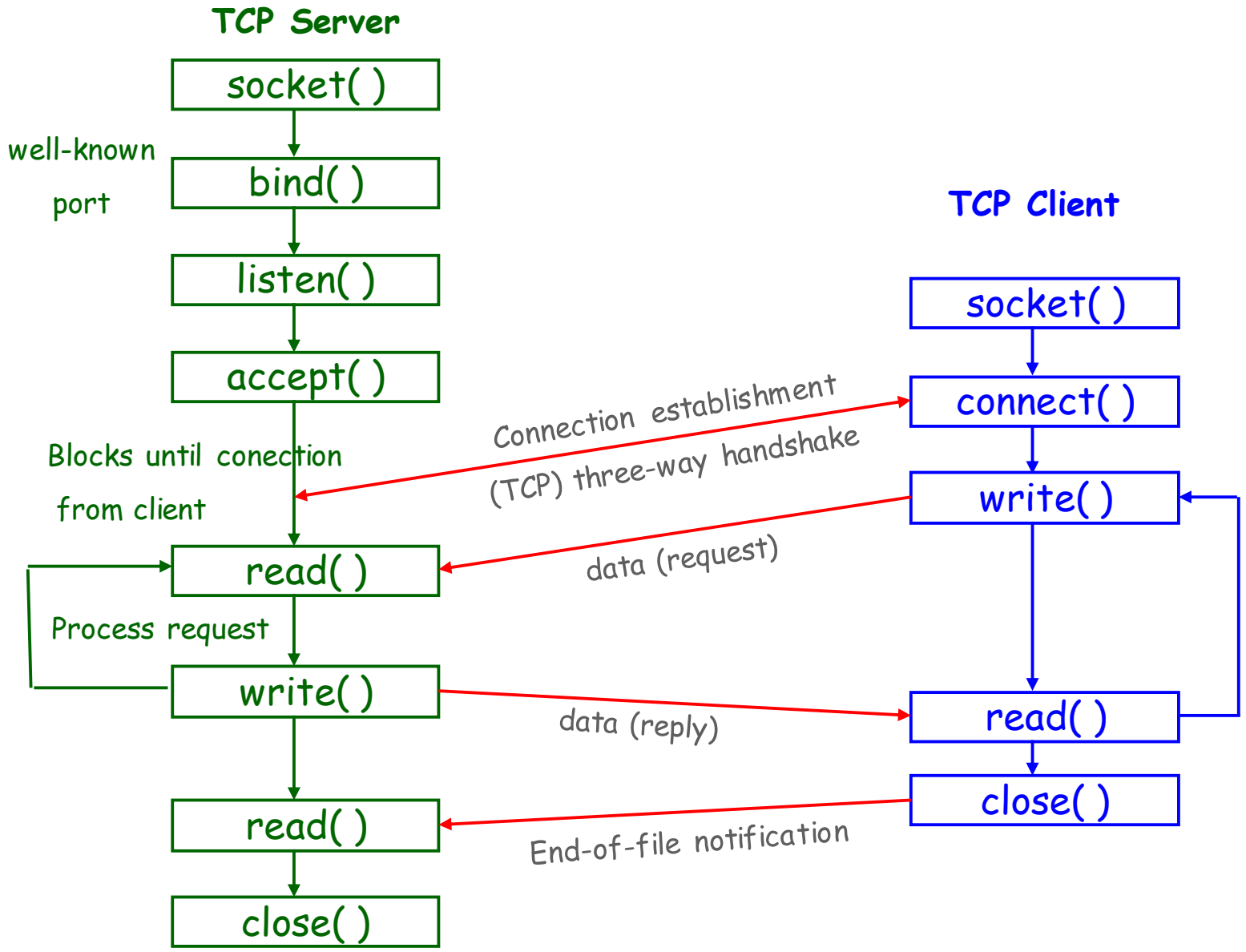


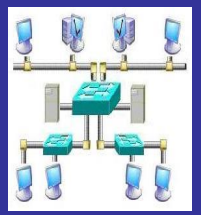
Socket Address Structure (IPv6)


```
struct in6_addr {
    unsigned char s6_addr[16]; // IPv6 address
};

struct sockaddr_in6 {
    u_int16_t       sin6_family;    // address family, AF_INET6
    u_int16_t       sin6_port;      // port number, Network Byte Order
    u_int32_t       sin6_flowinfo;  // IPv6 flow information
    struct in6_addr sin6_addr;      // IPv6 address
    u_int32_t       sin6_scope_id; // Scope ID
};
```



TCP Client-Server





Getaddrinfo Function

int getaddrinfo(const char *node, const char *service,
const struct addrinfo *hints, struct addrinfo **res);

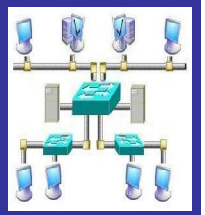
Do DNS and service name lookups, and fills out the structs you need

Parameters:

- *node: host name to connect to, or an IP address*
- *service: port number our a particular service (/etc/services)*
- *hints: points to a struct addrinfo that you have already filled out with relevant information*

upon success, fills out the res struct

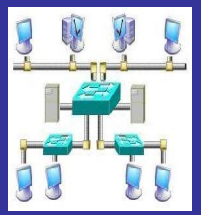
Return a non zero value if failure



Getaddrinfo Function

Example:

```
int status
struct addrinfo hints;
struct addrinfo *servinfo // will point to the results

menset(&hints, 0, sizeof(hints)); // make sure the struct is empty
hints.ai_family = AF_UNSPEC; //don't care IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM; //TCP stream sockets
hints.ai_flags = AI_PASSIVE  // fill in my IP for me

if (( status = getaddrinfo (NULL, "3490", &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(status));
    exit(1);
}
...
```



socket Function

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

Create a socket, giving access to transport layer service.

Parameters:

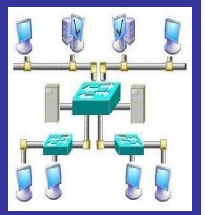
- *domain:*
  - *PF_INET - socket uses internet IPv4*
  - *PF_INET6 - socket uses internet IPv6*
- *type: um dentre as constantes:*
  - *SOCK_STREAM: TCP socket*
  - *SOCK_DGRAM: UDP Socket*
- *protocol: 0 to choose the proper protocol for the given type*

upon success returns socket file descriptor

like file descriptor => -1 if failure

Example:

```
getddrinfo("www.example.com", "http", &hints, &res)
s = socket (res->ai_family, res->ai_socktype, res->ai_protocol)
```



bind Function

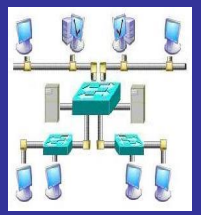
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);

Assign a local protocol address ("name") to a socket.

- *Sockfd* is the socket file descriptor returned by socket()
- *myaddr* is a pointer to address struct with:
  - *port number* and *IP address*
- *addrlen* is the length in bytes of that structure
- returns 0 if ok, -1 on error

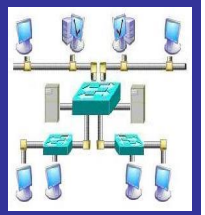
Example:

```
getaddrinfo(NULL, "3490", &hints, &res)
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol)
if (bind (sockfd, res->ai_addr, res->ai_addrlen) < 0)
    perror("bind call error");
```



bind Function

Process specifies		Result
IP address	port	
Wildcard	0	Kernel chooses IP address and port
Wildcard	nonzero	Kernel chooses IP address, process specifies port
Local IP address	0	Process specifies IP address, kernel chooses port
Local IP address	nonzero	Process specifies IP address and port



connect Function

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

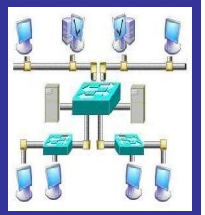
Connect to server.

- *sockfd:*
  It is the socket file descriptor returned by **socket()** call
- *servaddr:*
  It is a pointer to a structure with:
  - Server *port number* and *IP address*
  - **must** be specified!
- *addrlen* is the length of the structure

returns socket descriptor if ok, -1 on error

Example:

```
if (connect (sockfd, res->ai_addr, res->ai_addrlen)) < 0)
        perror("connect call error");
```



listen Function

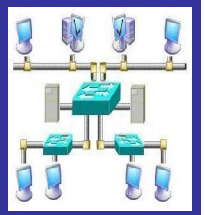
int listen(int *sockfd*, int *backlog*);

Announce willingness to accept connections, give queue size, change socket state for TCP server.

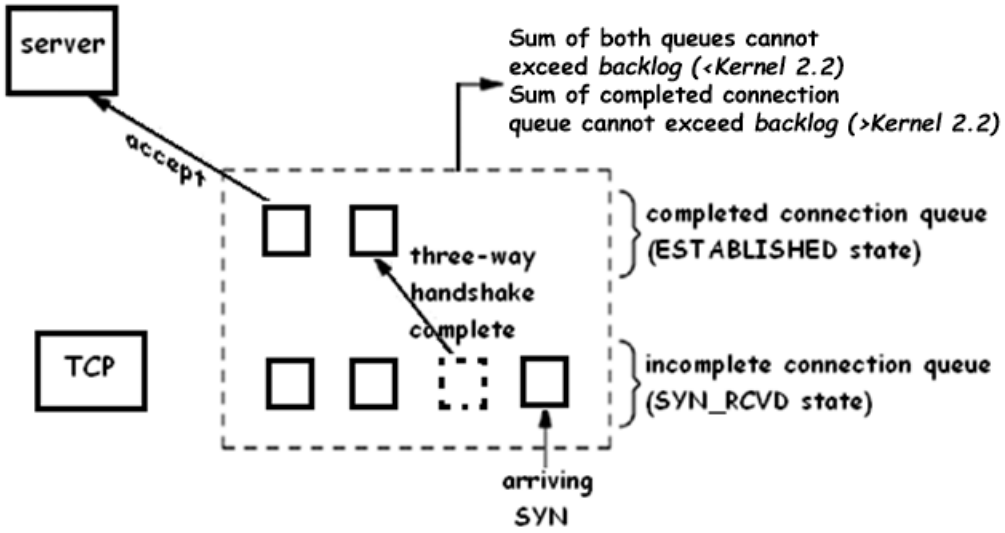
- *sockfd* is socket descriptor from **socket()** system call
- *backlog*
  - **Before Kernel 2.2**, it is the maximum number of connections allowed on the incoming queue (established + incomplete)
  - **After kernel 2.2**, it specifies the queue length for completely established sockets waiting to be accepted, instead of the number of incomplete connection requests. The maximum length of the queue for incomplete sockets can be set using /proc/sys/net/ipv4/tcp_max_syn_backlog. . (128 by default).
  - Tipically 5. Rarely above 15 on a even moderate web server!

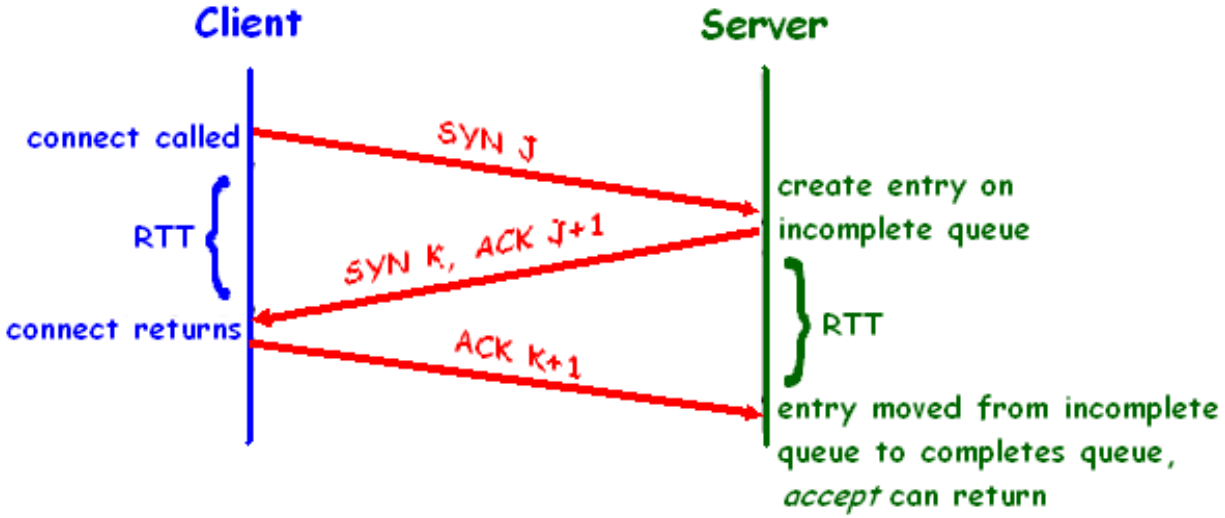
Example:

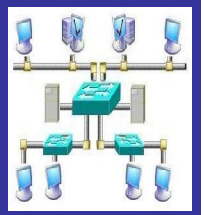
```
if (listen (sd, 2) < 0)
    perror ("listen call error");
```



listen Function

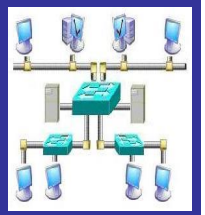






backlog

backlog	Maximum actual number of queued connections				
	MacOS 10.2.6 AIX 5.1	Linux 2.4.7	HP-UX 11.11	FreeBSD 4.8 FreeBSD 5.1	Solaris 2.9
0	1	3	1	1	1
1	2	4	1	2	2
2	4	5	3	3	4
3	5	6	4	4	5
4	7	7	6	5	6
5	8	8	7	6	8
6	10	9	9	7	10
7	11	10	10	8	11
8	13	11	12	9	13
9	14	12	13	10	14
10	16	13	15	11	16
11	17	14	16	12	17
12	19	15	18	13	19
13	20	16	19	14	20
14	22	17	21	15	22



accept Function

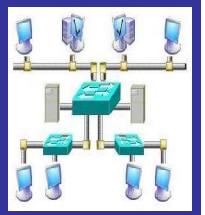
int accept (int sockfd, struct sockaddr *addr, socklen_t *addrlen);

Return next completed connection.

- *sockfd* is the socket file descriptor from **socket()** call
- *addr* and *addrlen* return protocol address from client
- returns brand new descriptor, created by OS
- if used with **fork()**, can create concurrent server (more later)

Example:

```
getaddrinfo(NULL, MYPORT, &hints, &res);
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
bind(sockfd, res->ai_addr, res->ai_addrlen);
listen(sockfd, BACKLOG);
// now accept an incoming connection:
addr_size = sizeof their_addr;
new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &addr_size);
```

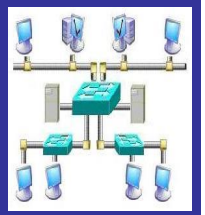


read and write Functions

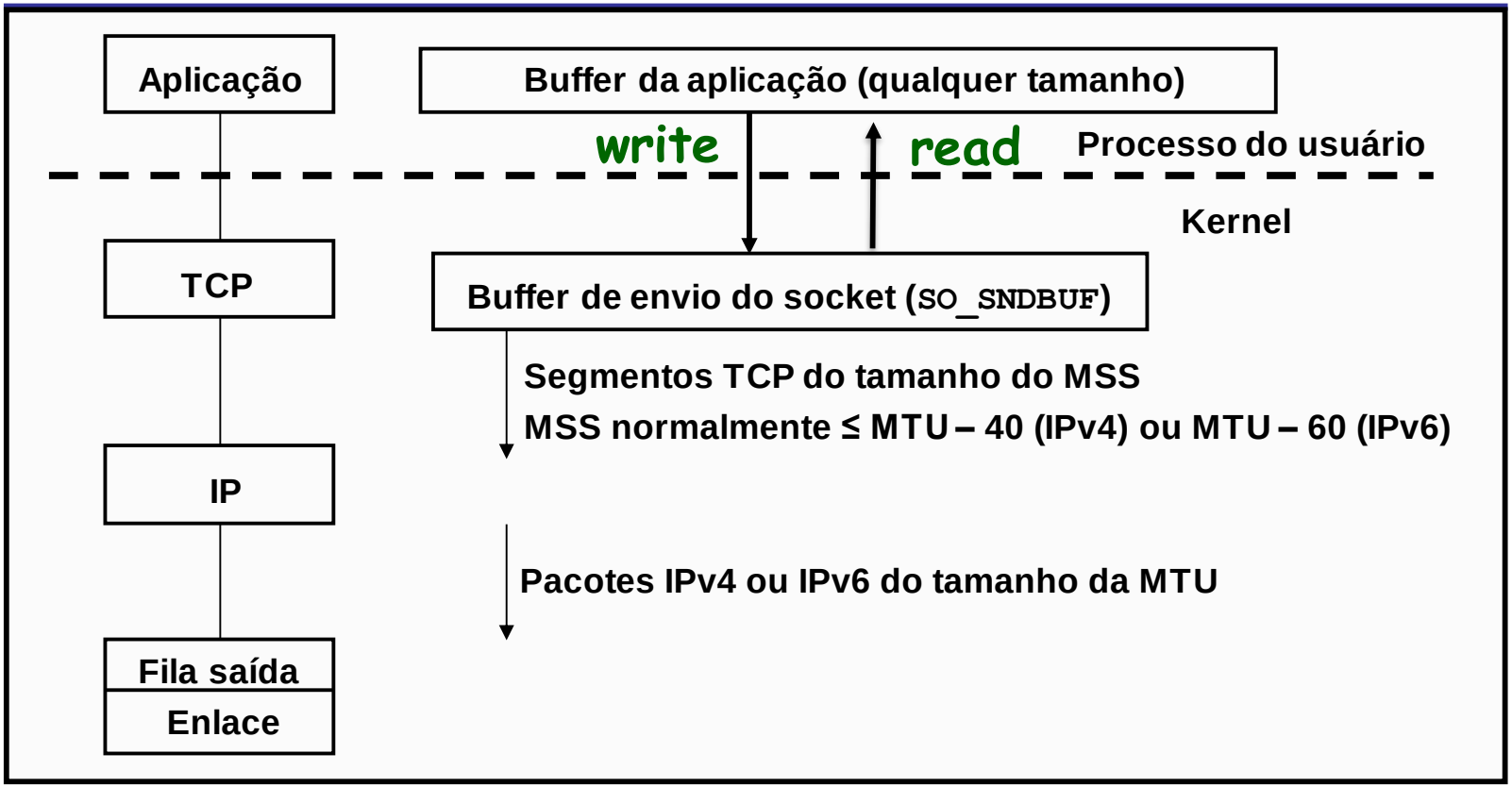
- Pode ler/escrever menos do que necessita devido ao tamanho restrito do buffer.
- É necessário chamar a função novamente.
- Read and *write* functions

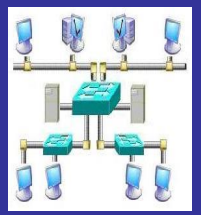
int read(int *sockfd*, void **buff*, size_t *mbytes*);

int write(int *sockfd*, void **buff*, size_t *mbytes*);



write Function





Sending and Receiving

int send(int sockfd, const void *msg, int len, int flags);

int recv(int sockfd, void *buf, int len, int flags);

Same as **read()** and **write()** but for flags

sockfd is the socket descriptor you want to send data to / read data from (whether it's the one returned by **socket()** or the one you got with **accept()**.)

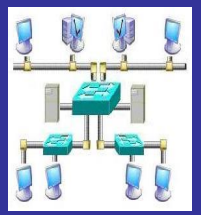
msg is a pointer to the data you want to send,.

buf is the buffer to read the information into.

len is the length of that data in bytes.

flags for I/O functions (see man pages)

- MSG_DONTWAIT (this send non-blocking)
- MSG_OOB (out of band data, 1 byte sent ahead)
- MSG_PEEK
- MSG_WAITALL (don't give me less than max)
- MSG_DONTROUTE (bypass routing table)

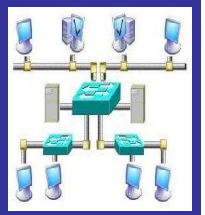


close Function

int close(int *sockfd*);

Close socket for use.

- *sockfd* is the socket file descriptor from **socket()** call
- It closes socket for reading/writing
  - returns (doesn't block)
  - attempts to send any unsent data
  - -1 if error



getpeername() - Who are you?

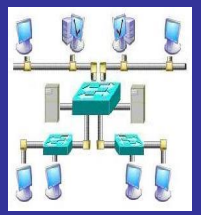

```
#include <sys/socket.h>
int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);
```

This function will tell you who is at the other end of a connected stream socket.

- *sockfd* is the descriptor of the connected stream socket,
- *addr* is a pointer to a struct sockaddr (or a struct sockaddr_in) that will hold the information about the other side of the connection, and
- *addrlen* is a pointer to an int, that should be initialized to sizeof *addr or sizeof(struct sockaddr).

The function returns -1 on error and sets errno accordingly.

Once you have their address, you can use **inet_ntop()**, **getnameinfo()**, or **gethostbyaddr()** to print or get more information.



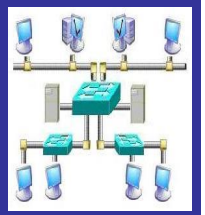
gethostname() - Who am I?


```
#include <unistd.h>
int gethostname(char *hostname, size_t size);
```

It returns the name of the computer that your program is running on. The name can then be used by gethostbyname() to determine the IP address of your local machine.

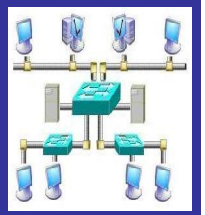
- *hostname* is a pointer to an array of chars that will contain the hostname *upon the function's return, and*
- *size* is the length in bytes of the hostname array.

The function returns 0 on successful completion, and -1 on error, setting errno as usual.



Servidor Concorrente

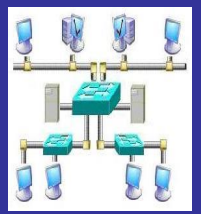
- Após **accept** e **fork**, o processo filho executa no new_fd e o pai continua a escutar no sock_fd.
- **close** decrementa contador de referências.
- FYN é enviado somente quando contador de referências possui valor zero.



Servidor Concorrente – fork

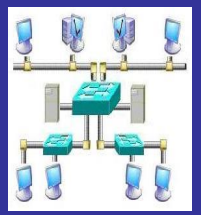
- **fork** executa uma cópia idêntica do processo.
- Retorna o identificador do processo filho ao pai (*process ID*) e o valor 0 (*process ID*) ao filho.
- Os descritores abertos antes do **fork** são compartilhados com os filhos.
- o "*connected socked*" após um accept seguido de **fork** é compartilhado com o filho.

```
# include <unistd.h>
pid_t fork(void);
    Returns: 0 in child, process ID of child in parent, -1 on error
```

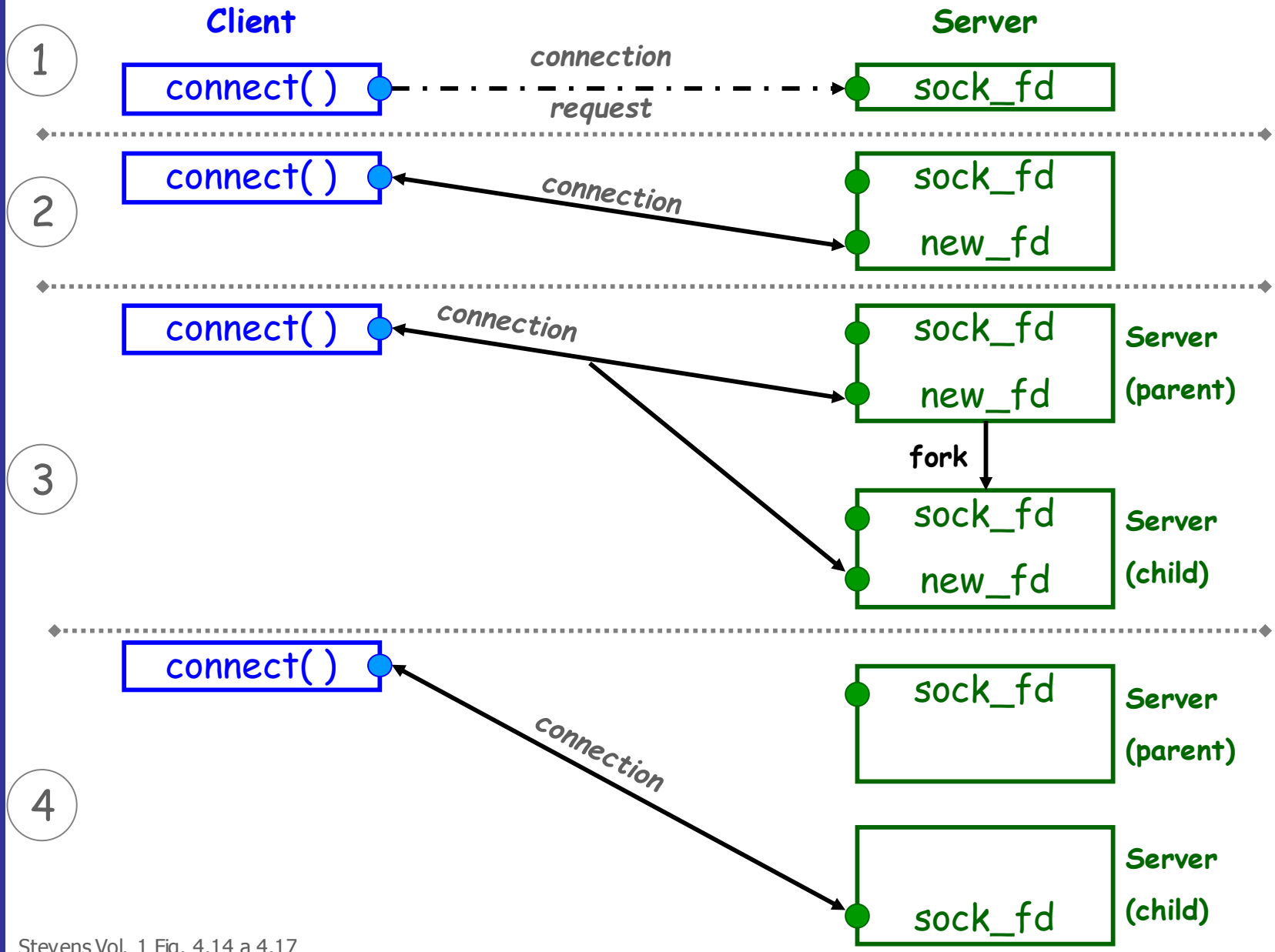


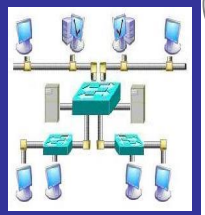
Servidor Concorrente


```
pid_t pid;
int sock_fd, new_fd;
/* fill in sockaddr_in{ } with server's well-known port */
sock_fd = socket( ... );
bind(sock_fd, ... );
listen(sock_fd, LISTENQ);

for ( ; ; ) {
    new_fd = accept (sock_fd, ... );   /* probably blocks */

    if( (pid = fork() ) == 0) {
          close(sock_fd);      /* child closes listening socket */
          doit(new_fd);        /* process the request */
          close(new_fd);       /* done with this client */
          exit(0);             /* child terminates */
    }

    close(new_fd);    /* parent closes connected socket */
}
```



Servidor Concorrente

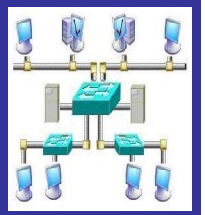




Stevens Vol. 1 Fig. 4.14 a 4.17

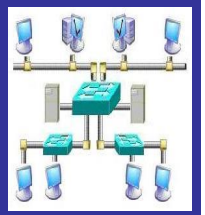
Servidor Concorrente - Exemplo

- Servidor Echo
- Cliente Echo



Servidor Echo (*main*)

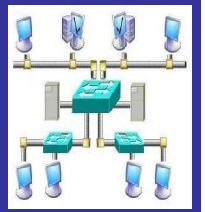

```
1  #include "unp.h"
2  int
3  main(int argc, char **argv)
4  {
5     int sock_fd, new_fd;
6     pid_t childpid;
7     socklen_t clilen;
8     struct sockaddr_in cliaddr, servaddr;
9     sock_fd= Socket (PF_INET, SOCK_STREAM, 0);
10    bzero(&servaddr, sizeof(servaddr));
11    servaddr.sin_family = AF_INET;
12    servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
13    servaddr.sin_port = htons (SERV_PORT);
14    Bind(sock_fd, (SA *) &servaddr, sizeof(servaddr));
15    Listen(sock_fd, LISTENQ);
16    for ( ; ; ) {
17            clilen = sizeof(cliaddr);
18            new_fd= Accept(sock_fd, (SA *) &cliaddr, &clilen);
19            if ( (childpid = Fork()) == 0) {      /* child process */
20                    Close(sock_fd);         /* close listening socket */
21                    str_echo(new_fd);       /* process the request */
22                    exit (0);
23            }
24            Close(new_fd);    /* parent closes connected socket */
25    }
26 }
```



Stevens Vol. 1 Fig. 5.2 TCP echo server

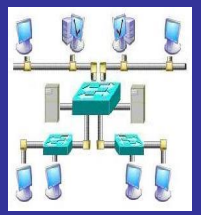
Servidor Echo (*str_echo*)


```
1 #include "unp.h"
2
3 void str_echo(int new_fd)
4 {
5   ssize_t n;
6   char buf[MAXLINE];

7   again:
8         while ( (n = read(new_fd, buf, MAXLINE)) > 0)
9         writen(new_fd, buf, n);
10        if (n < 0 && errno == EINTR)
11                  goto again;
12        else if (n < 0)
13                  perror("str_echo: read error");
14 }
```



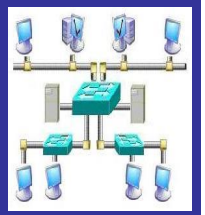
Cliente Echo (*main*)


```
1 #include "unp.h"
2
3 int main(int argc, char **argv)
4 {
5   int sock_fd;
6   struct sockaddr_in servaddr;
7   if (argc != 2)
8         err_quit("usage: tcpcli <IPaddress>");
9   sock_fd = Socket(PF_INET, SOCK_STREAM, 0);
10 bzero(&servaddr, sizeof(servaddr));
11 servaddr.sin_family = AF_INET;
12 servaddr.sin_port = htons(SERV_PORT);
13 Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
14 Connect(sock_fd, (SA *) &servaddr, sizeof(servaddr));
15 str_cli(stdin, sock_fd);    /* do it all */
16 exit(0);
17 }
```



Cliente Echo (*str_cli*)


```
1 #include "unp.h"
2
3 str_cli(FILE *fp, int sock_fd)
4 {
5   char sendline[MAXLINE], recvline[MAXLINE];
6   while (Fgets(sendline, MAXLINE, fp) != NULL) {
7         Writen(sock_fd, sendline, strlen (sendline));
8         if (Readline(sock_fd, recvline, MAXLINE) == 0)
9                   rr_quit("str_cli: server terminated prematurely");
10        Fputs(recvline, stdout);
11  }
12 }
```



Stevens Vol. 1 Fig. 5.5 str_cli function: client processing loop