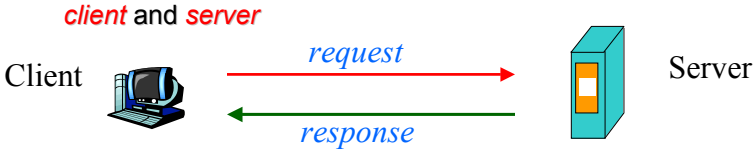
BSD UNIX transport API: The Socket Interface

The Client-Server Programming Paradigm

- most networking applications can be divided into two pieces: *client* and *server*



Server waits for requests from clients and serves them.
- The server can either handle requests iteratively, or concurrently (by spawning child processes)

Client asks a server to do some work and send back the results.
- Client is usually short-lived process, communicates with one server at a time, simpler design
- Server usually runs forever, communicates with multiple clients at any given moment, design is complex

UNIX I/O Paradigm and Network I/O

- UNIX open-read-write-close paradigm
- network communication functionality is more complex:
  - specify communication endpoints
  - client vs server behavior
  - specify transport protocol (TCP or UDP)
  - etc
- Several new operating system calls were added as well as new library routines.
- General mechanism provided to accommodate many protocols and addressing schemes.

/etc/protocols

```
0    HOPOPT     IPv6 Hop-by-Hop Option          [RFC1883]
1    ICMP       Internet Control Message        [RFC792]
2    IGMP       Internet Group Management       [RFC1112]
3    GGP        Gateway-to-Gateway              [RFC823]
4    IP         IP in IP (encapsulation)        [RFC2003]
5    ST         Stream                          [RFC1190,RFC1819]
6    TCP        Transmission Control            [RFC793]
...
17   UDP        User Datagram                   [RFC768,JBP]
...
```

Specifying A Protocol Interface

Designers at Berkeley wanted to accommodate multiple set of communication protocols.

They provided generality far beyond TCP/IP:

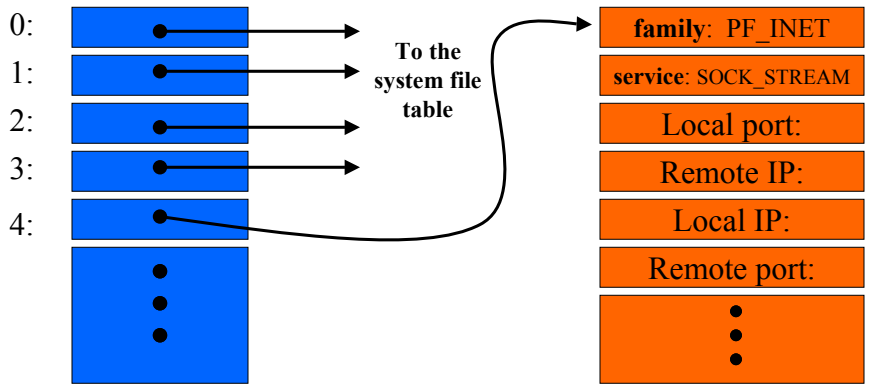
- multiple *protocol families*
- multiple *types of service*
- multiple *addressing families*

The Socket Abstraction

- The socket is a new abstraction for network communication.
- Like files, each socket is identified by a small integer called its *socket descriptor*.

  UNIX allocates socket descriptors in the same table as file descriptors.
- Single system call is sufficient to create any socket.
- Once the socket has been created, an application must make additional system calls to specify the details of exact use.

System Data Structures for Sockets



Creating a Socket

The *socket* system call creates sockets on demand :

sd = **socket**(protocol_family, service_type, protocol);

TCP/IP internet protocol family: PF_INET

Example Service Types:

- reliable stream delivery service (SOCK_STREAM)
- connectionless datagram delivery service (SOCK_DGRAM)
- raw type (SOCK_RAW)

Socket Inheritance

UNIX uses the ***fork*** and ***exec*** system calls to start new application programs. It is a two-step procedure:

- ***fork*** creates a separate copy of the currently executing process.
- By calling ***exec***, the new copy replaces itself with the desired application program.

When a program calls ***fork***, the newly created process inherits access to all open sockets just as it inherits access to all open files.

When a program calls ***exec***, the new application retains access to all open sockets.

Socket Termination

When a process finishes using a socket it calls *close*:

close(sock_d);

The *shutdown* call causes all or part of a full-duplex connection on the socket associated with sock_d to be shut down.

- If how is 0, further receives will be disallowed.
- If how is 1, further sends will be disallowed.
- If how is 2, further sends and receives will be disallowed.

shutdown(sock_d, how);

TCP Communication Endpoint

- A protocol family is free to choose one or more addressing schemes to define address representations.

  The socket abstraction defines an *address family* for each type of address.

  The TCP/IP protocols (PF_INET protocol family) all use a single address representation: AF_INET.

- TCP/IP *communication endpoint*:

  (IP address, port number)

  Other protocols define their endpoint addresses in other ways.

TCP/IP Endpoint Address

The *sockaddr_in* structure (/usr/include/netinet/in.h) specifies the TCP/IP endpoint format:

```
struct sockaddr_in {
  u_char  sin_len;      /* total address length, don't touch it! */
  u_char  sin_family;   /* address family (AF_INET) */
  u_short sin_port;           /*  port number */
  struct  in_addr sin_addr;   /* IP address */
  char    sin_zero[8];        /* unused, must be zero */
};
```

Specifying Endpoint Addresses

- Newly created socket has no association to local or destination address.
- Server processes that operate at a well-known port must be able to specify that port to the system:

  bind(sd, local_addr, addr_len);
- For a client, the protocol software chooses local address and port automatically.

  After creating a socket, a client calls *connect* to establish an

  active reliable byte stream connection to remote server:

  connect(sd, dest_addr, addr_len);

Sending and Receiving Data through a Socket

Once an application has established a socket, it can use the socket to transmit and receive data.

write(sd, buffer, buffer_len);

read(sd, buffer, buffer_len);

- ***Caveat !***

  Since TCP is a byte-stream protocol *read()* / *write()* can return with less bytes than it was requested.

  When reading/writing from/to a TCP socket *always* use loop!

Specifying a Queue Length for a Server

If computing a response takes nontrivial amount of time, a new request may arrive before a server finishes responding to an old request.

The server may tell the underlying protocol software that it wishes to have that requests enqueued until it has time to process them.

```
listen(sd, backlog);
```

- converts socket to the *listening* socket on which incoming connections from clients can be accepted
- *backlog* specifies the maximum number of client connections that the kernel will queue for the socket.

How a Server accepts Connections

Once a socket has been established (socket, bind, listen), a server needs to wait for a connection. To do so, it uses system call *accept*.

- A call to *accept* blocks until a connection request arrives.
- When a client initiates the connection (*connect*) the TCP *three-way handshake (3WHS)* protocol is initiated
- If 3WHS is successful a new TCP connection is established and *accept* returns a new descriptor (*connected* descriptor) for communication with the new client (*read()/write()*)

```
newsock = accept(sd, addr, addr_len);
```

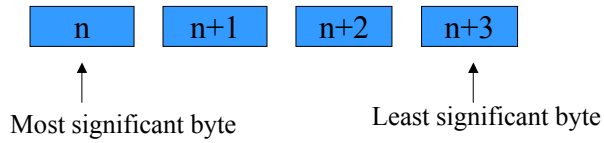
Servers that handle Multiple Services

A server may wait for connections on multiple sockets.
The system call, *select*, applies to I/O in general, not just to communication over sockets.
A call to *select* blocks waiting for one of a set of file descriptors to become ready:

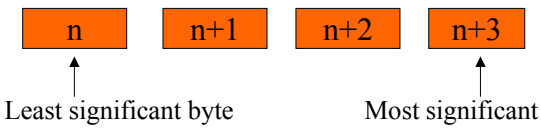
nready = **select**(max_desc, in_desc, out_desc, exc_desc, timeout);

Network Byte Order: Big Endian

- Big Endian (SPARC, IBM 370, PDP-10, Motorola, RISC):



- Little Endian (Intel, VAX, PDP-11):



- Middle Endian (2-3-0-1 or 1-0-3-2)

BSD UNIX Network Library Calls

- network byte order conversion routines

```
local_short = ntohs(net_short);    local_long  = ntohl(net_long);
net_short   = htons(local_short);  net_long  = htonl(local_long);
```

- address manipulation routines

```
ip_addr = inet_addr(dotted_decimal);
dotted_decimal = inet_ntoa(ip_addr);
```

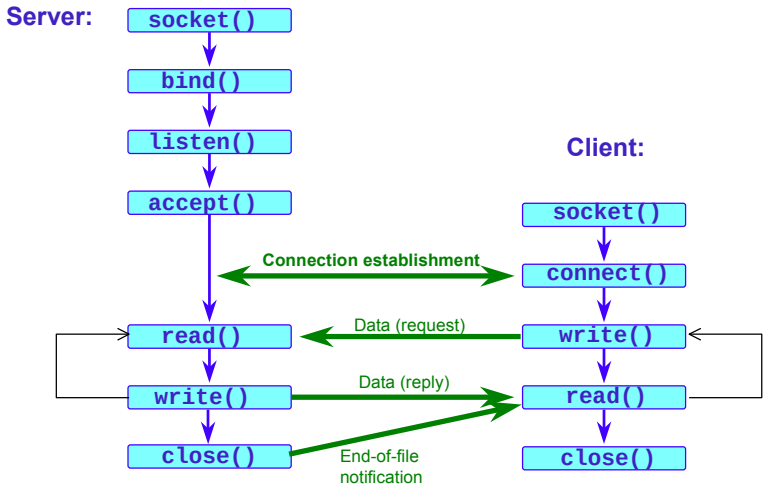
- obtaining info about protocols

```
protocol_struct = getprotobyname(name);
```

Obtaining Info about Hosts

```
host_struct = gethostbyname(domain_name);
struct  hostent {
    char    *h_name;       /* official name of host */
    char    **h_aliases;   /* alias list */
    int     h_addrtype;    /* host address type */
    int     h_length;      /* length of an address */
    char    **h_addr_list; /* list of addresses from name server */
#define h_addr  h_addr_list[0]  /* first IP address */
  };
```

Using Socket Calls in A Program



Our First Socket Program

- A simple TCP day-time client/server:
- Client: establishes a TCP connection with a server
- Server: accepts the connection and sends back the current time and date in human readable form
  - iterative: serves only a single client at a time

Day-Time TCP Client

```
1 int main(int argc, char **argv) {
4       int         sockfd, n;
5       char        recvline[MAXLINE + 1];
6       struct sockaddr_in  servaddr;
8       sockfd = socket(PF_INET, SOCK_STREAM, 0);
9       bzero(&servaddr, sizeof(servaddr));
10      servaddr.sin_family = AF_INET;
11      servaddr.sin_port   = htons(13);       /* daytime server */
12      inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
13      connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr));
14      while ( (n = read(sockfd,  recvline,  MAXLINE)) > 0) {
15              recvline[n] = '\0';      /* null terminate */
16              fputs(recvline, stdout);
17      }
18      exit(0);
19 }
```

Day-Time TCP Server

```
1 int  main(int argc, char **argv)   {
2   int     listenfd, connfd;
3   struct sockaddr_in  servaddr;
4   char  buff[MAXLINE];
5   time_t   ticks;
6
7   listenfd = socket(PF_INET, SOCK_STREAM, 0);
8   bzero(&servaddr, sizeof(servaddr));
9   servaddr.sin_family      = AF_INET;
10  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
11  servaddr.sin_port   = htons(13);  /* daytime server */
```

Day-Time TCP Server (contd)

```
12 bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr));
13 listen(listenfd, LISTENQ);
14
15 for ( ; ; ) {
16          connfd = accept(listenfd, (struct sockaddr *) NULL, NULL);
17          ticks = time(NULL);
18          snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
19          write(connfd, buff, strlen(buff));
20          close(connfd);
21    }
22 }
```

UDP

- s = socket(PF_INET, SOCK_DGRAM, 0)
- connect(s, sockaddr, addr_len)
- sendto(s, message, msg_len, 0 , sockaddr_to, addr_len)
- send(s, message, msg_len, 0)
- recvfrom(s, &buf, buf_len, 0, &sockaddr_from, from_len)
- recv(s, &buf, buf_len, 0)

Further Info Sources

- /etc/protocols
- man pages
- /usr/include/
  - sys/types.h
  - sys/socket.h
  - netinet/in.h
  - netdb.h