CSCB09 Software Tools and Systems Programming
Process Communication – Sockets

Marcelo Ponce

Winter 2025

Department of Computer and Mathematical Sciences - UTSC

Today's class

Today we will discuss the following topics:

- Sockets

Processes Communication

Quick Recap

- Exit codes
- Signals

Quick Recap

- Exit codes
- Signals
- Pipes

In order to use a pipe for interprocess communication what has to be true?

1. The process writing to the pipe must be the parent of the process reading from the pipe.
2. The process writing to the pipe must be the child of the process reading from the pipe.
3. The processes must be a parent/child combination but the direction of the pipe could go either way.
4. The processes must be on the same machine.

In order to use a pipe for interprocess communication what has to be true?

1. The process writing to the pipe must be the parent of the process reading from the pipe.
2. The process writing to the pipe must be the child of the process reading from the pipe.
3. The processes must be a parent/child combination but the direction of the pipe could go either way.
4. **The processes must be on the same machine.**

Processes Communication

Within the same machine

- Exit codes
- Signals
- Pipes

To communicate processes across different machines...

Processes Communication

Within the same machine

- Exit codes
- Signals
- Pipes

To communicate processes across different machines...

$\Rightarrow$ SOCKETS

Processes communicating

- **process** $\rightsquigarrow$ program running within a host
- within same host, two processes can communicate using *inter-process communication* (defined by OS)
- processes in different hosts communicate by exchanging *messages*

Client-Server Paradigm

Clients & Servers

client process : process that initiates communication

server process : process that waits to be contacted

Client-Server Paradigm

Clients & Servers

client process : process that initiates communication

server process : process that waits to be contacted

Client

- Running on end host
- Requests service
- E.g., Web browser

Client-Server Paradigm

Clients & Servers

client process : process that initiates communication

server process : process that waits to be contacted

Client

- Running on end host
- Requests service
- E.g., Web browser

Server

- Running on end host
- Provides service
- E.g., Web server

Client

- Sometimes ON
- Initiates a request to the server when interested
- E.g., web browser
- Needs to know the server's address
- Shows up and initiates the connection

Server

- Always ON
- Serve services to many clients
- E.g., `www.utsc.utoronto.ca`
- Not initiate contact with the clients
- Needs a fixed address
- Always available at a known location

Sockets

Application Programming Interface – Sockets

- Socket Interface was originally provided by the Berkeley distribution of Unix (BSD)
- Now supported in virtually all operating systems
- Each protocol provides a certain set of services, and the API provides a syntax by which those services can be invoked in this particular OS

Application Programming Interface – Sockets

- Socket Interface was originally provided by the Berkeley distribution of Unix (BSD)
- Now supported in virtually all operating systems
- Each protocol provides a certain set of services, and the API provides a syntax by which those services can be invoked in this particular OS
- Once you set up the socket, the interface behaves like a file (or pipe)
  - entry in the file descriptor table
  - read/write/close
- Except – sockets are full-duplex (2 way)

Across the network...
When you think about the task at hand of sending information across different machines and employing a complex and complicated underlying computer network (e.g. the internet), things may look even much more complicated...

Across the network...
When you think about the task at hand of sending information across different machines and employing a complex and complicated underlying computer network (e.g. the internet), things may look even much more complicated...

- Network Protocols
- layers
- clients & servers
- addresses, ports, TCP/IP
- packets
- routing

Because of the way in which computer networks have been designed, following a *layer approach* we can just focus in learning the basics of the "application" layer.

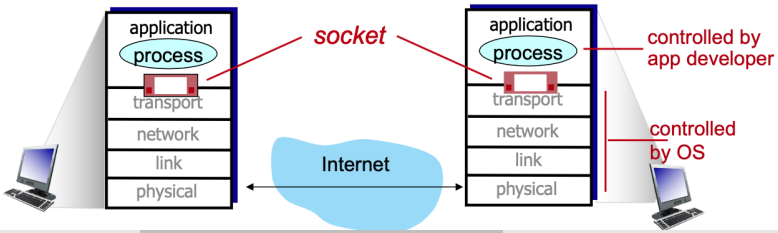
Across the network...
When you think about the task at hand of sending information across different machines and employing a complex and complicated underlying computer network (e.g. the internet), things may look even much more complicated...

- Network Protocols
- layers
- clients & servers
- addresses, ports, TCP/IP
- packets
- routing

Because of the way in which computer networks have been designed, following a *layer approach* we can just focus in learning the basics of the "application" layer.
Having said that, if you are interested in learning the way in which computer networks work, consider taking **CSCD58**

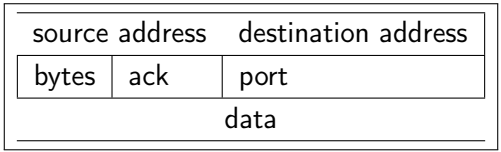
Sockets

- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process
- two sockets involved: one on each side



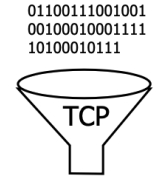
TCP/IP

- Transmission Control Protocol/Internet Protocol
- Tell us how to *package up* the data

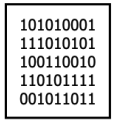


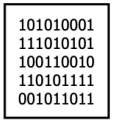
Packet Details

- make packets

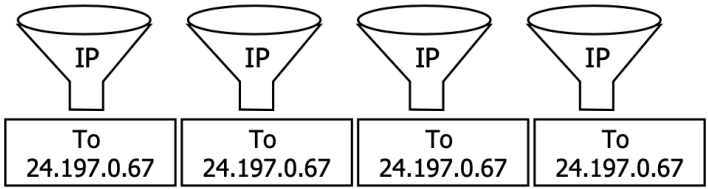


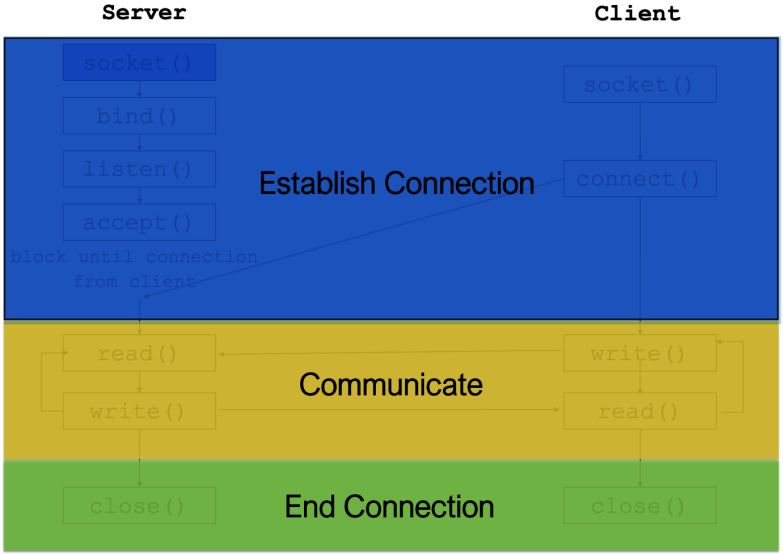
- Each TCP packet is given a header
  - -sequence number
  - -checksum

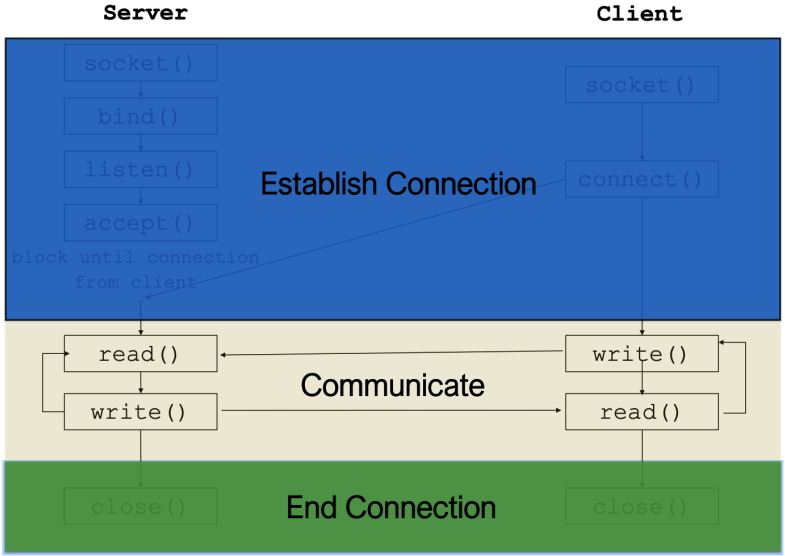


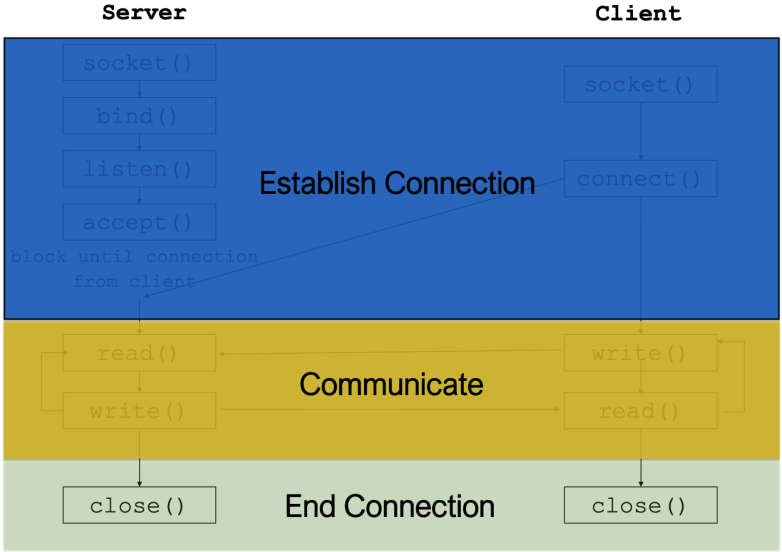


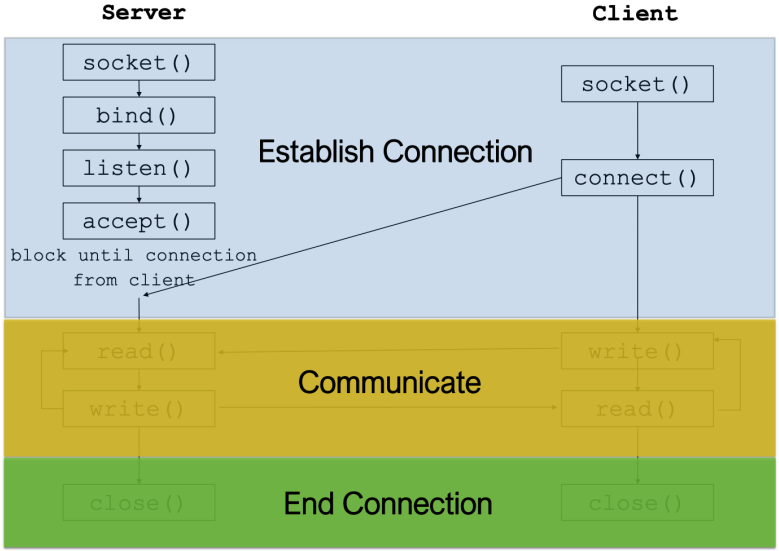
- put in an IP envelope with another header

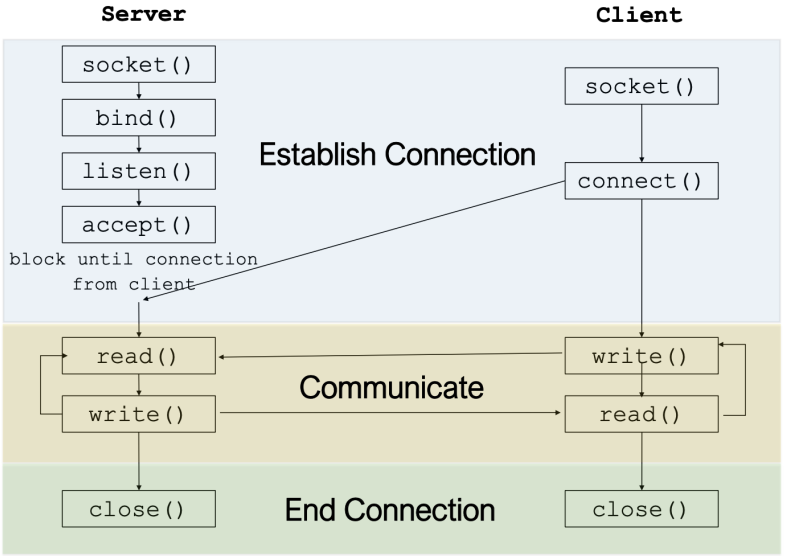



Server
Client
socket()
bind()
listen()
accept()
block until connection
from client
socket()
connect()
Establish Connection
read()
write()
Communicate
write()
read()
close()
End Connection
close()


Server
Client
socket()
bind()
listen()
accept()
block until connection
from client
socket()
connect()
Establish Connection
read()
write()
Communicate
write()
read()
close()
End Connection
close()


Server
Client
socket()
bind()
listen()
accept()
block until connection
from client
socket()
connect()
Establish Connection
read()
write()
Communicate
write()
read()
close()
End Connection
close()


Server
Client
socket()
bind()
listen()
accept()
socket()
connect()
Establish Connection
block until connection
from client
read()
write()
write()
read()
Communicate
close()
close()
End Connection


Server
Client
socket()
bind()
listen()
accept()
block until connection
from client
socket()
connect()
Establish Connection
read()
write()
Communicate
write()
read()
close()
End Connection
close()

Connection-Oriented

Server

- Create a socket: `socket()`
- Assign an address to a socket: `bind()`
- Establish a queue for connections: `listen()`
- Get a connection from the queue: `accept()`

Client

- Create a socket: `socket()`
- Initiate a connection: `connect()`

Addressing Processes

- to receive messages, process must have **identifier**
- host device has unique *32-bit IP address*

Addressing Processes

- to receive messages, process must have **identifier**
- host device has unique *32-bit IP address*

» Q: does IP address of host on which process runs suffice for identifying the process?

» A: no, many processes can be running on same host

Addressing Processes

- to receive messages, process must have **identifier**
- host device has unique *32-bit IP address*

» Q: does IP address of host on which process runs suffice for identifying the process?

» A: no, many processes can be running on same host

- **identifier** includes both **IP address** and **port numbers** associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to gaia.cs.umass.edu web server:
  - IP address: 128.119.245.12
  - port number: 80

Addresses & Ports

- A **socket pair** is the two endpoints of the connection.
- An endpoint is identified by an **IP address and a port**.
- IPv4 addresses are 4 8-bit numbers:
  - 128.100.31.200 = werewolf
  - 128.100.31.201 = seawolf
  - 128.100.31.202 = skywolf
- Ports:
  - because multiple processes can communicate with a single machine we need another identifier.
- command-line tool to explore: `nslookup`

Ports

The *port* number is a 16-bit integer that ranges from 0 to 65535.

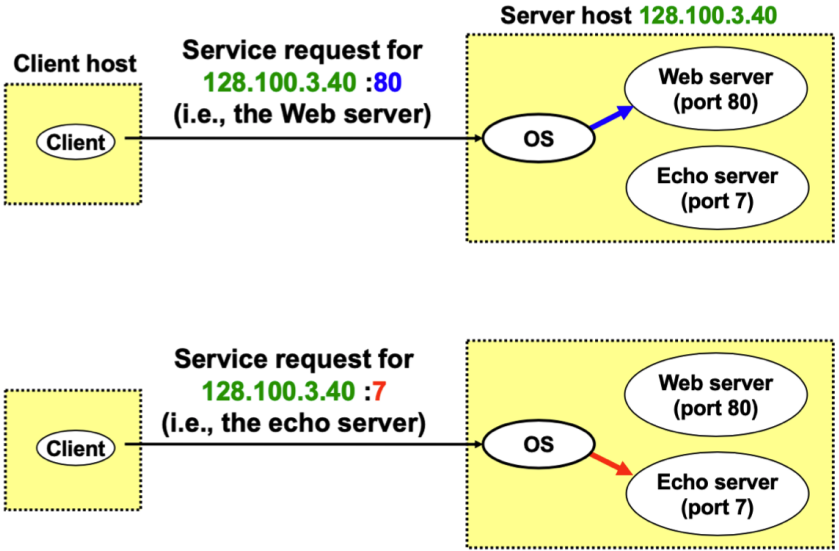
Types of port number space:

- Pre-established ports – **0-1023**
  80: http 21: ftp
  22: ssh 25: smtp (email)
  23: telnet 194: irc
- Registered port – **1024-49151**
  2709: supermon
  26000: quake
  3724: world of warcraft
- Dynamic port – **49152-65535**
  pick ports in this range to avoid overlap

Number	Assignment
20	File Transfer Protocol (FTP) Data Transfer
21	File Transfer Protocol (FTP) Command Control
22	Secure Shell (SSH) Secure Login
23	Telnet remote login service, unencrypted text messages
25	Simple Mail Transfer Protocol (SMTP) email delivery
53	Domain Name System (DNS) service
67, 68	Dynamic Host Configuration Protocol (DHCP)
80	Hypertext Transfer Protocol (HTTP) used in the World Wide Web
110	Post Office Protocol (POP3)
119	Network News Transfer Protocol (NNTP)
123	Network Time Protocol (NTP)
143	Internet Message Access Protocol (IMAP) Management of digital mail
161	Simple Network Management Protocol (SNMP)
194	Internet Relay Chat (IRC)
443	HTTP Secure (HTTPS) HTTP over TLS/SSL
546, 547	DHCPv6 IPv6 version of DHCP

`https://www.iana.org/assignments/service-names-port-numbers/`

Using ports to identify services



Unix Socket API

Socket interface

- A collection of system calls to write a networking program at user-level.
- Originally provided in Berkeley UNIX
- Later adopted by all popular operating systems

In UNIX, everything is like a file

- All input is like reading a file
- All output is like writing a file
- File is represented by an integer file descriptor
- Data written into socket on one host can be read out of socket on other host

System calls for sockets

- Client:
  - create, connect, write, read, close
- Server:
  - create, bind, listen, accept, read, write, close

Socket:

- The point where a local application process attaches to the network
- An interface between an application and the network
- An application creates the socket

The interface defines operations for

- Creating a socket
- Attaching a socket to the network
- Sending and receiving messages through the socket
- Closing the socket

Socket: end point of communication

- Processes send messages to one another
- Messages traverse the underlying network

A process sends and receives through a "socket"

- Analogy: the doorway of the house
- Socket, as an API, supports the creation of network applications

Socket parameters

A socket connection has 5 general parameters:

- The protocol
  Example: TCP and UDP.
- The local and remote address
  Example: 128.100.3.40
- The local and remote port number
  - Some ports are reserved (e.g., 80 for HTTP)
  - Root access require to listen on port numbers below 1024

Socket parameters

A socket connection has 5 general parameters:

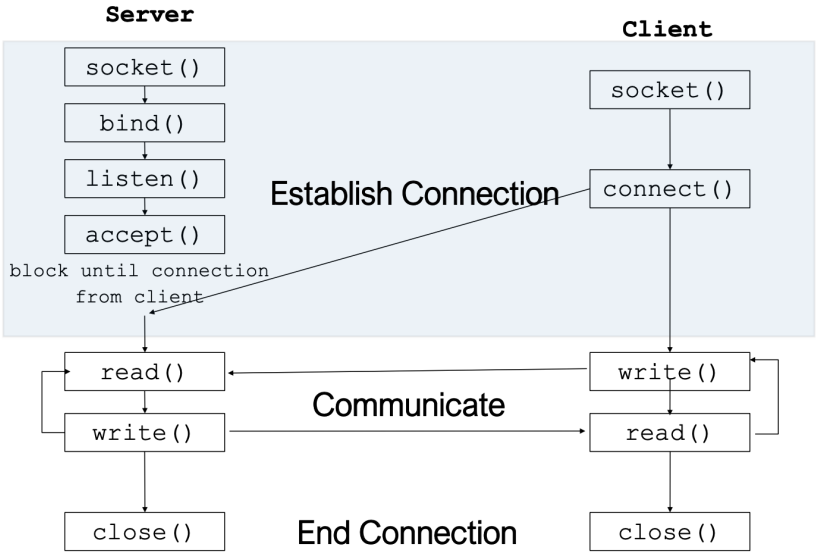
- The protocol
  Example: TCP and UDP.
- The local and remote address
  Example: 128.100.3.40
- The local and remote port number
  - Some ports are reserved (e.g., 80 for HTTP)
  - Root access require to listen on port numbers below 1024

Socket Family

- `PF_INET` denotes the Internet family
- `PF_UNIX` denotes the Unix pipe facility
- `PF_PACKET` denotes direct access to the network interface (i.e., it bypasses the TCP/IP protocol stack)

Socket Type

- `SOCK_STREAM` is used to denote a byte stream
- `SOCK_DGRAM` is an alternative that denotes a message oriented service, such as that provided by UDP


Server
Client
socket()
bind()
listen()
accept()
block until connection
from client
socket()
connect()
Establish Connection
read()
write()
Communicate
write()
read()
close()
End Connection
close()

Typical Client Program

Prepare to communicate

- Create a socket
- Determine server address and port number
- Initiate the connection to the server

Typical Client Program

Prepare to communicate

- Create a socket
- Determine server address and port number
- Initiate the connection to the server

Exchange data with the server

- Write data to the socket
- Read data from the socket
- Do something with the data (eg. render a webpage)

Typical Client Program

Prepare to communicate

- Create a socket
- Determine server address and port number
- Initiate the connection to the server

Exchange data with the server

- Write data to the socket
- Read data from the socket
- Do something with the data (eg. render a webpage)

Close the socket

`socket()`	create the socket descriptor
`connect()`	connect to the remote server
`read()`,`write()`	communicate with the server
`close()`	end communication by closing socket descriptor

Typical Server Program

Prepare to communicate

- Create a socket
- Associate local address and port with the socket

Typical Server Program

Prepare to communicate

- Create a socket
- Associate local address and port with the socket

Wait to hear from a client (passive open)

- Indicate how many clients-in-waiting to permit
- Accept an incoming connection from a client

Typical Server Program

Prepare to communicate

- Create a socket
- Associate local address and port with the socket

Wait to hear from a client (passive open)

- Indicate how many clients-in-waiting to permit
- Accept an incoming connection from a client

Exchange data with the client over new socket

- Receive data from the socket
- Send data to the socket
- Close the socket

Typical Server Program

Prepare to communicate

- Create a socket
- Associate local address and port with the socket

Wait to hear from a client (passive open)

- Indicate how many clients-in-waiting to permit
- Accept an incoming connection from a client

Exchange data with the client over new socket

- Receive data from the socket
- Send data to the socket
- Close the socket

Repeat with the next connection request

`bind()`	associate the local address
`listen()`	wait for incoming connections from clients
`accept()`	accept incoming connection
`read(),write()`	communicate with the client
`close()`	end communication by closing socket descriptor

Server side

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

- `family` specifies protocol family:
  - `AF_INET`: IPv4
  - `AF_LOCAL`: UNIX domain
- `type`

  `SOCK_STREAM`, `SOCK_DGRAM`, `SOCK_RAW`
- `protocol`

  set to `0` except for RAW sockets
- returns a *socket descriptor*

Creating a Socket

```
int sockfd = socket(address_family, type, protocol);
```

- The socket number returned is the socket descriptor for the newly created socket
- TCP – `SOCK_STREAM`: Connection oriented

  ```
  int sockfd = socket (PF_INET, SOCK_STREAM, 0);
  ```

- UDP – `SOCK_DGRAM`: Connectionless

  ```
  int sockfd = socket (PF_INET, SOCK_DGRAM, 0);
  ```

Bind to a name

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

- `sockfd` – returned by `socket()`
-
```
struct sockaddr_in {
    short             sin_family;   /*AF_INET */
    u_short           sin_port;
    struct    in_addr  sin_addr;
    char         sin_zero[8];  /*filling*/
};
```
- Binds the newly created socket to the specified address i.e. the network address of the local participant (the server)
- Address is a data structure which combines IP and port
- `sin_addr.s_addr` can be set to `INADDR_ANY` to communicate on any network interface
- `addrlen` is the size of `struct sockaddr_in`

Set up queue in kernel

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

- after calling `listen()`, a socket is ready to accept connections
- prepares a queue in the kernel where partially completed connections wait to be accepted
- defines how many connections can be pending on the specified socket
- `backlog` is the maximum number of partially completed connections that the kernel should queue

Complete the connection

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

- carries out the passive open
- blocks waiting for a connection (from the queue)
- returns a new descriptor which refers to the TCP connection with the client
- `sockfd` is the listening socket
- `cliaddr` stores the address of the client on return
- reads and writes on the connection will use the socket returned by accept

Client side

- `socket()` – same as server, to say "how" we are going to talk

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

- the kernel will choose a dynamic port and source IP address.
- returns 0 on success and -1 on failure setting `errno`.

Sending and Receiving Data

- Sending Data

  `write(int sockfd, void *buf, size_t len);`

  Arguments: socket descriptor, pointer to buffer of data, and length of the buffer

  Returns the number of characters written, and -1 on error

- Receiving data

  `read(int sockfd, void *buf, size_t len);`

  Arguments: socket descriptor, pointer to buffer to place the data, size of the buffer

  Returns the number of characters read (where 0 implies "end of file"), and -1 on error

Important Network Details

Byte Order – aka “Endianess”

Big-endian
The most significant byte (the "big end") of the data is placed at the byte with the lowest address

$$\mathbf{91,329_{10}} \approx \mathbf{0x164C1} \sim \mathbf{0164C1_{16}} = \boxed{00}\ \boxed{01}\ \boxed{64}\ \boxed{C1}$$

Little-endian
The least significant byte (the "little end") of the data is placed at the byte with the lowest address

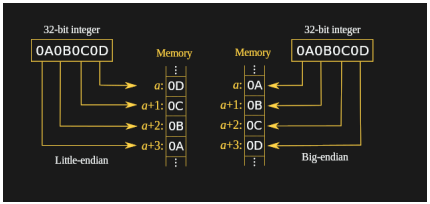
$$\mathbf{91,329_{10}} = \boxed{C1}\ \boxed{64}\ \boxed{01}\ \boxed{00}$$

- Machines differ in how they store data
  E.g. Intel is little-endian, Sparc is big-endian, ...

Network Byte Order

- To communicate between machines with unknown or different "endian-ness" we convert numbers to **network byte order (big-endian)** before we send them.
- There are functions provided to do this:

```
unsigned long htonl(unsigned long)
unsigned short htons(unsigned short)
unsigned long ntohl(unsigned long)
unsigned short ntohs(unsigned short)
```



- Use `htons()` and `htonl()` to convert to network byte order
  Use `ntohs()` and `ntohl()` to convert to host order

Network new-line

“`\r\n`” rather than just “`\n`”

Application: Client-Server Model with TCP

Client-Server Model with TCP

Server

- Passive open
  Prepares to accept connection, does not actually establish a connection
- Hearing from multiple clients
  Allow a backlog of waiting clients
  ... in case several try to start a connection at once
- Create a socket for each client
  Upon accepting a new client
  ... create a new socket for the communication

Client

- Application performs active open
- It say swho it wants to communicate with
- Client invokes
  `int connect (int socket, struct sockaddr *address, int addr_len)`
- Connect
  Does not return until TCP has successfully established a connection at which application is free to begin sending data
  Address contains remote machine's address

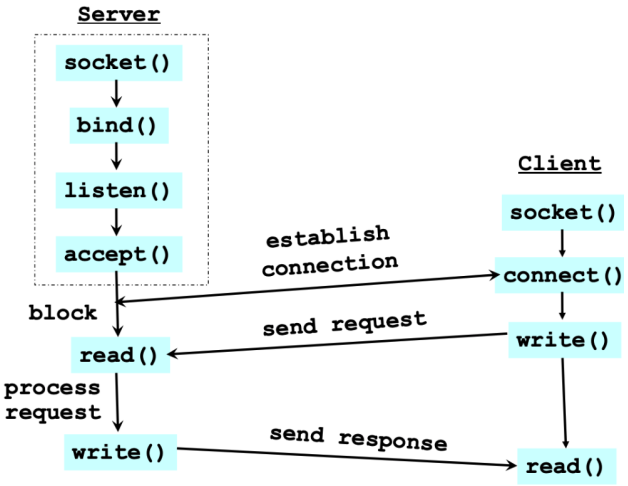
Client-Server Model with TCP

In practice

- The client usually specifies only remote participant's address and let's the system fill in the local information
- Whereas a server usually listens for messages on a well-known port
- A client does not care which port it uses for itself, the OS simply selects an unused one

Once a connection is established, the application process invokes two operation

```
int send (int socket, char *msg, int msg_len, int flags)
int recv (int socket, char *buff, int buff_len, int flags)
```



```
gethostbyname() // get address for given host name (e.g. 128.100.3.40 for name
     "cs.toronto.edu");
getservbyname() // get port and protocol for a given service e.g. ftp, http (e
    .g. "http" is port 80, TCP)
getsockname() // get local address and local port of a socket
getpeername() // get remote address and remote port of a socket

struct sockaddr { // Generic address, "connect(), bind(), accept()"
  u_short sa_family;  // <sys/socket.h>
  char sa_data[14];
};

struct sockaddr_in {  // Client & server addresses TCP/UDP address (inc.port
    #)
  u_short sa_family;  // <netinet/in.h>
  u_short sa_family;
  u_short sin_port;
  struct in_addr sin_addr;
  char sin_zero[8];
};

struct in_addr {  // IP address <netinet/in.h>
  u_long s_addr;
};
```

```
// Convert between system's representation of IP addresses and readable
    strings (e.g. "128.100.3.40")
unsigned long inet_addr(char* str);
char * inet_ntoa(struct in_addr inaddr);

// Relevant header files:
<sys/types.h>
<sys/socket.h>
<netinet/in.h>
<arpa/inet.h>


// man pages
socket, accept, bind , listen
```