Computer System Principles

Network Programming

UMASSCS
SCHOOL OF COMPUTER SCIENCE
Announcement

• We will drop the lowest assignment grade, except that we won’t drop the grade for the last assignment if that grade is not a passing grade (70%).
NETWORK TOPOLOGY
Network Topology

Host A → Router → Router → Host B
**Client-server architecture**

**server:**
- Generally an “always-on” host
- (e.g. Amazon, telephone)
- Generally at a *permanent* IP address
  - data centers for scaling

**clients:**
- communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do not typically communicate directly with each other
Client-Server Model

- An application process is assigned a process identifier number (process ID) - likely to be different each time that process is started.
- Process IDs differ between OS platforms - they are not uniform.
- A server process can have multiple connections to multiple clients at a time - simple connection identifiers at one end are not unique.
A Client-Server Transaction

- Most network applications are based on the client-server model:
  - A server process and one or more client processes
  - Server manages some resource
  - Server provides service by manipulating resource for clients
  - Server activated by request from client (vending machine analogy)

1. Client sends request
2. Server handles request
3. Server sends response
4. Client handles response

Note: clients and servers are processes running on hosts (can be the same or different hosts)
To a host, a network is just another I/O device.
Protocols

a human protocol and a computer network protocol:

protocols define format, order of msgs sent and received among network entities, and actions taken on msg transmission, receipt
IP: Internet Protocol, provides unreliable delivery of packets

Network Layer: IP
Data Link Layer

Internet  
  ↓
  Link  
  ↓
  Ethernet

Internet  
  ↓
  Link  
  ↓
  Satellite/Fibe

Internet  
  ↓
  Link  
  ↓
  Ethernet

Internet  
  ↓
  Link  
  ↓
  Ethernet
UDP, Unreliable Data Protocol, Unreliable data delivery

Transport Layer: TCP/UDP
Network Layer: IP
Data Link Layer

Transport Layer:
- Transport
- Internet
- Link
  - Ethernet
  - Satellite/Fibre

Transport Layer:
- IP
- Data Network Layer:
- Process-to-process
  - TCP/UDP
  - Reliable Data Protocol,
    Reliable data delivery

Link
- Ethernet
TCP, Transmission Control Protocol, reliable data delivery

Transport Layer: TCP/UDP
Network Layer: IP
Data Link Layer

Transport

Internet

Link

Ethernet

Satellite/Fibre

Internet

Link

Ethernet
Application Layer: HTTP/FTP

Transport Layer: TCP/UDP

Network Layer: IP

Data Link Layer

- Ethernet
- Satellite/Fibre
TCP OVERVIEW
TCP

The TCP data unit - segment

- does not recognize messages,
- sends a block of bytes from the byte stream between sender and receiver.
TCP

• The primary purpose of TCP is to provide a reliable logical circuit or connection service between pairs of processes.
  – does not assume reliability from the lower-level protocols (such as IP), so TCP must guarantee this itself
    • Heavy network utilization and resulting congestion
    • Faulty network hardware or connectors
TCP

• The primary purpose of TCP is to provide a reliable logical circuit or connection service between pairs of processes.
  – does not assume reliability from the lower-level protocols (such as IP), so TCP must guarantee this itself
    • Heavy network utilization and resulting congestion
    • Faulty network hardware or connectors
  – When a packet is successfully delivered to its destination, the destination should send an acknowledgement
TCP

• The primary purpose of TCP is to provide a reliable logical circuit or connection service between pairs of processes.
  – does not assume reliability from the lower-level protocols (such as IP), so TCP must guarantee this itself
    • Heavy network utilization and resulting congestion
    • Faulty network hardware or connectors
  – When a packet is successfully delivered to its destination, the destination should send an acknowledgement
  – otherwise, the source system retransmits the packet
    • the destination never receive the packet
    • the ack is lost
TCP Three-Way Handshake
TCP Three-Way Handshake

Client

syn=1, seq=x

Server

Time

Time
TCP Three-Way Handshake

Client

<table>
<thead>
<tr>
<th>Time</th>
<th>syn=1, seq=x</th>
</tr>
</thead>
</table>

Server

<table>
<thead>
<tr>
<th>Time</th>
<th>syn=1, seq=y, ack=x+1</th>
</tr>
</thead>
</table>

Client

<table>
<thead>
<tr>
<th>Time</th>
<th>syn=1, seq=x</th>
</tr>
</thead>
</table>
TCP Three-Way Handshake

Client

syn=1, seq=x

syn=1, seq=y, ack=x+1

syn=0, seq=x+1, ack=y+1

Server

Time

Time
SOCKET
What is a socket?

• Clients and servers communicate by sending streams of bytes over connections:
  – Point-to-point, full-duplex (2-way communication), and reliable.
What is a socket?

• Clients and servers communicate by sending streams of bytes over connections:
  – Point-to-point, full-duplex (2-way communication), and reliable.

• A socket is an endpoint of a connection
  – Socket address is an IP address:port pair
What is a socket?

• Clients and servers communicate by sending streams of bytes over connections:
  – Point-to-point, full-duplex (2-way communication), and reliable.

• A **socket** is an endpoint of a connection
  – Socket address is an IPAddress:port pair

• A **port** is a 16-bit integer that identifies a process
What is a socket?

• Clients and servers communicate by sending streams of bytes over **connections**:
  – Point-to-point, full-duplex (2-way communication), and reliable.

• A **socket** is an endpoint of a connection
  – Socket address is an IP address:port pair

• A **port** is a 16-bit integer that identifies a process:
  – **Ephemeral port**: Assigned automatically on client when client makes a connection request
  – **Well-known port**: Associated with some service provided by a server (e.g., port 80 is associated with Web servers)
What is a socket?

- Clients and servers communicate by sending streams of bytes over connections:
  - Point-to-point, full-duplex (2-way communication), and reliable.

- A **socket** is an endpoint of a connection
  - Socket address is an IP address:port pair

- A **port** is a 16-bit integer that identifies a process:
  - **Ephemeral port**: Assigned automatically on client when client makes a connection request
  - **Well-known port**: Associated with some service provided by a server (e.g., port 80 is associated with Web servers)

- A connection is uniquely identified by the socket addresses of its endpoints (**socket pair**) (and its protocol, e.g., TCP).
  - (cliaddr:cliport, servaddr:servport)
TCP Sockets - Ports

- TCP uses **well-known** and **ephemeral ports**
- **Well Known Ports**: range 0-1023
  - assigned to the server side of an application.
- **Registered Ports**: range 1024-49151
  - publicly defined; convenience for the Internet community; avoids vendor conflicts
- **Dynamic and/or Private Ports**: range 49152-65535
  - Server
    - can be used freely by any client or server
Anatomy of a Connection

• A connection is uniquely identified by the socket addresses of its endpoints (*socket pair*) – *(cliaddr:cliport, servaddr:servport)*

  - Connection socket pair: *(128.2.194.242:51213, 208.216.181.15:80)*
  - Client socket address: 128.2.194.242:51213
  - Server socket address: 208.216.181.15:80
  - Client host address: 128.2.194.242
  - Server host address: 208.216.181.15

51213 is an ephemeral port allocated by the kernel

80 is a well-known port associated with Web servers
Using Ports to Identify Services

Client host

Service request for 128.2.194.242:80 (i.e., the Web server)

Client

Server host 128.2.194.242

Kernel

Web server (port 80)

Echo server (port 7)

Service request for 128.2.194.242:7 (i.e., the echo server)

Client

Kernel

Web server (port 80)

Echo server (port 7)
Sockets Interface

• Set of system-level functions used in conjunction with Unix I/O to build network applications.

• Created in the early 80’s as part of the original Berkeley distribution of Unix that contained an early version of the Internet protocols.

• Available on all modern systems
  – Unix variants, Windows, OS X, IOS, Android, ARM
Socket is like file I/O

- To the kernel, a socket is an endpoint of communication
- To an application, a socket is a file descriptor that lets the application read/write from/to the network

• **Remember:** All Unix I/O devices, including networks, are modeled as “files”
  - Clients and servers communicate with each other by reading from and writing to socket descriptors

• The main distinction between regular file I/O and socket I/O is how the application “opens” the socket descriptors
Socket v.s. pipe

- Fundamental difference between a socket and a pipe: sockets serve to interconnect two processes that execute on arbitrary machines. (Pipes work only on the same machine.)
  - Two processes can execute on the same host, or
  - on networked hosts across the world from each other.
  
  The set up and operations on the sockets are the same!

- TCP sockets implement a high level abstraction:
  - It gives the programmer a byte-stream communication channel across networked hosts that is reliable and order-preserving.
Suppose there is a connection from:
   192.168.0.17:53268 to 128.119.40.1:80
Which of these statements is true?
A) 192.168.0.17 is the client
B) 53268 is an ephemeral port
C) 128.119.40.1 is the server
D) 80 is a well-known port (for HTTP)
E) All of the above
Suppose there is a connection from:
   192.168.0.17:53268 to 128.119.40.1:80

Which of these statements is true?
A) 192.168.0.17 is the client
B) 53268 is an ephemeral port
C) 128.119.40.1 is the server
D) 80 is a well-known port (for HTTP)
E) All of the above
SOCKET ADDRESSING
IPv4 address: a dotted quad

4 bytes

________________
/          \
128.119.240.84
\/
1 byte
Storing an IP address in struct in_addr

typedef uint32_t in_addr_t;
struct in_addr
{
    in_addr_t s_addr;
};
Storing an IP address in struct in_addr

```c
int main()
{
    struct in_addr addr;
    unsigned char * ip;

    ip = (unsigned char *) & (addr.s_addr);

    ip[0]=128;
    ip[1]=119;
    ip[2]=240;
    ip[3]=84;

    printf("Hello %s\n", inet_ntoa(addr));
}
```
Storing an IP address in struct in_addr

```c
int main(){
    struct in_addr addr;
    unsigned char * ip;

    ip = (unsigned char *) &(addr.s_addr);

    ip[0]=128;
    ip[1]=119;
    ip[2]=240;
    ip[3]=84;

    printf("Hello %s\n", inet_ntoa(addr));
}
```
Storing an IP address in struct in_addr

```c
int main(){
    struct in_addr addr;
    inet_aton("128.119.240.84", &(addr));
    printf("Hello %s\n", inet_ntoa(addr));
}
```
Socket Address Structures

```c
struct sockaddr {
    uint16_t   sa_family;    /* Protocol family */
    char       sa_data[14]; /* Address data. */
};
```

```
struct
  struct
    struct
      struct
        struct
          struct
            struct
              struct
                struct
                  struct
                    struct
                      struct
`
Socket Address Structures

• Generic socket address:
  - For address arguments to `connect`, `bind`, and `accept`
  - Necessary only because C did not have generic (`void *`) pointers when the sockets interface was designed

```
struct sockaddr {
    uint16_t sa_family;  /* Protocol family */
    char sa_data[14];    /* Address data. */
};
```

`sa_family`
Socket Address Structures

- Internet-specific socket address

```c
struct sockaddr_in  {
    uint16_t    sin_family; /* Protocol family (always AF_INET) */
    uint16_t    sin_port;  /* Port num in network byte order */
    struct in_addr sin_addr; /* IP addr in network byte order */
    unsigned char sin_zero[8];/* Pad to sizeof(struct sockaddr) */
};
```
Socket Address Structures

- Internet-specific socket address
  - sockaddr_in is like a subtype of sockaddr

```c
struct sockaddr_in  {
    uint16_t sin_family; /* Protocol family (always AF_INET) */
    uint16_t sin_port;  /* Port num in network byte order */
    struct in_addr sin_addr; /* IP addr in network byte order */
    unsigned char sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};
```
Socket Address Structures

- Internet-specific socket address:
  - `sockaddr_in` is like a subtype of `sockaddr`
  - **Must cast** `(struct sockaddr_in *)` to `(struct sockaddr *)` for functions that take socket address arguments.

```c
struct sockaddr_in {
    uint16_t sin_family; /* Protocol family (always AF_INET) */
    uint16_t sin_port;  /* Port num in network byte order */
    struct in_addr sin_addr; /* IP addr in network byte order */
    unsigned char sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};
```

![Diagram of `sockaddr_in` structure with AF_INET and family specific fields highlighted]
int main()
{
    struct sockaddr sadr;
    struct sockaddr_in * saddr_in = (struct sockaddr_in *) &sadr;
    inet_aton("128.119.240.84", &(saddr_in->sin_addr));
    printf("Hello %s\n", inet_ntoa(saddr_in->sin_addr));
}
Struct sockaddr_in
{
    uint16_t sin_family; /* Protocol family (always AF_INET) */
    uint16_t sin_port;  /* Port num in network byte order */
    struct in_addr sin_addr; /* IP addr in network byte order */
    unsigned char sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};
Is this correct in little endian systems?

```c
struct sockaddr_in  {
    uint16_t sin_family; /* Protocol family (always AF_INET) */
    uint16_t sin_port; /* Port num in network byte order */
    struct in_addr sin_addr; /* IP addr in network byte order */
    unsigned char sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};

struct sockaddr saddr;
struct sockaddr_in * saddr_in = (struct sockaddr_in *) &saddr;
saddr->sin_port = 80;
```
Is this correct in little endian systems?

```c
struct sockaddr_in  {
    uint16_t     sin_family;  /* Protocol family (always AF_INET) */
    uint16_t     sin_port;    /* Port num in network byte order */
    struct in_addr sin_addr; /* IP addr in network byte order */
    unsigned char sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};

struct sockaddr saddr;
struct sockaddr_in * saddr_in = (struct sockaddr_in *) &saddr;
saddr->sin_port = 80;
```

Little endian: The most significant byte is stored in the biggest address

Increasing address: 
```
bits: 0 0 0 0 1 0 1 0
```
Is this correct in little endian systems?

```c
struct sockaddr_in  {
    uint16_t    sin_family; /* Protocol family (always AF_INET) */
    uint16_t    sin_port;  /* Port num in network byte order */
    struct in_addr sin_addr; /* IP addr in network byte order */
    unsigned char sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};
```

```c
struct sockaddr saddr;
struct sockaddr_in * saddr_in = (struct sockaddr_in *) &saddr;
saddr->sin_port = htons(80);
```

Little endian: The most significant byte is stored in the biggest address.

Increasing address:

```
bits: 0 0 0 0 1 0 1 0
```
SOCKET API
1. **Start server**

   - Server
     - `getaddrinfo`
     - `socket`
     - `bind`
     - `listen`
     - `accept`
   - Client
     - `getaddrinfo`
     - `socket`
     - `connect`
     - `send`
     - `receive`
     - `close`

   - **Await connection request from next client**

   **Sockets Interface**

   **open_clientfd**

   **Client / Server Session**
2. **Start client**

Client

1. **Start server**

Server

- `open_clientfd`
- `getaddrinfo`
- `socket`
- `connect`
- `send`
- `receive`
- `close`

- `open_listенfd`
- `getaddrinfo`
- `socket`
- `bind`
- `listen`
- `accept`
- `send`
- `receive`
- `close`
2. **Start client**
   **Client**
   - getaddrinfo
   - socket
   - connect
   - receive
   - close

3. **Exchange data**

   Await connection request from next client

4. **Start server**
   **Server**
   - getaddrinfo
   - socket
   - bind
   - listen
   - accept
   - receive
   - send
   - close

**Client / Server Session**

open_clientfd

open_listenfd
1. Start server

Server

- getaddrinfo
- socket
- bind
- listen
- accept

Connection request

2. Start client

Client

- getaddrinfo
- socket
- connect

3. Exchange data

- send
- receive

Await connection request from next client

4. Disconnect client

- close
Host and Service Conversion: `getaddrinfo`

```c
int getaddrinfo(const char *host,    /* Hostname or address */
                const char *service,  /* Port or service name */
                const struct addrinfo *hints, /* Input parameters */
                struct addrinfo **result);  /* Output linked list */
```

- **Given** `host` and `service`, `getaddrinfo` **returns** `result` that points to a linked list of `addrinfo` structs, each of which points to a corresponding socket address struct, and which contains arguments for the sockets interface functions.
Host and Service Conversion: `getaddrinfo`

- **Given** host and service, `getaddrinfo` returns result that points to a linked list of `addrinfo` structs, each of which points to a corresponding socket address struct, and which contains arguments for the sockets interface functions.

- **Helper functions:**
  - `freeaddrinfo` frees the entire linked list.
  - `gai_strerror` converts error code to an error message.
Linked List Returned by `getaddrinfo`
**Linked List Returned by** `getaddrinfo`  

- **Clients:** walk this list, trying each socket address in turn, until the calls to `socket` and `connect` succeed.
- **Servers:** walk the list until calls to `socket` and `bind` succeed.
Each `addrinfo` struct returned by `getaddrinfo` contains arguments that can be passed directly to `socket` function.

Also points to a socket address struct that can be passed directly to `connect` and `bind` functions.
Host and Service Conversion: `getnameinfo`

- `getnameinfo` is the inverse of `getaddrinfo`, converting a socket address to the corresponding host and service.
  - Replaces obsolete `gethostbyaddr` and `getservbyport` funcs.
  - Reentrant and protocol independent.

```c
int getnameinfo(const SA *sa, socklen_t salen, /* In: socket addr */
                 char *host, size_t hostlen,       /* Out: host */
                 char *serv, size_t servlen,       /* Out: service */
                 int flags);                        /* optional flags */
```
#include "csapp.h"

int main(int argc, char **argv)
{
    struct addrinfo *p, *listp, hints;
    char buf[MAXLINE];
    int rc, flags;

    /* Get a list of addrinfo records */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET; /* IPv4 only */
    hints.ai_socktype = SOCK_STREAM; /* Connections only */
    if ((rc = getaddrinfo(argv[1], NULL, &hints, &listp)) != 0) {
        fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(rc));
        exit(1);
    }
}

hostinfo.c
/* Walk the list and display each IP address */
flags = NI_NUMERICHOST; /* Display address instead of name */
for (p = listp; p; p = p->ai_next) {
    Getnameinfo(p->ai_addr, p->ai_addrlen,
               buf, MAXLINE, NULL, 0, flags);
    printf("%s\n", buf);
}

/* Clean up */
Freeaddrinfo(listp);
exit(0);
Running hostinfo

```
student@osboxes:~/Code/lec24$ ./hostinfo amazon.com
54.239.25.192
54.239.25.208
54.239.25.200
54.239.17.6
54.239.26.128
54.239.17.7

student@osboxes:~/Code/lec24$ ./hostinfo google.com
216.58.219.206

student@osboxes:~/Code/lec24$ ./hostinfo localhost
127.0.0.1

student@osboxes:~/Code/lec24$ ./hostinfo umass.edu
128.119.103.148
```
**Sockets Interface**

**Client**
- `getaddrinfo`
- `socket`
- `connect`
- `send`
- `receive`
- `close`

**Server**
- `getaddrinfo`
- `socket`
- `bind`
- `listen`
- `accept`
- `receive`
- `send`
- `close`

Connection request:
- Client: `connect`
- Server: `accept`

Await connection request from next client:
- Server: `receive`
- Client: `close`
Sockets Interface: `socket`

- Clients and servers use the `socket` function to create a `socket descriptor`:

- Example:

```c
int socket(int domain, int type, int protocol)
```

Indicate protocol number, normally 0

```c
int clientfd = Socket(AF_INET, SOCK_STREAM, 0);
```

Indicates that we are using 32-bit IPV4 addresses

Indicates socket type
Sockets Interface: \textit{socket}

- Clients and servers use the \texttt{socket} function to create a \texttt{socket descriptor}.
- Example:

  ```c
  int socket(int domain, int type, int protocol);
  int clientfd = Socket(AF_INET, SOCK_STREAM, 0);
  ```
  
  Indicates that we are using 32-bit IPV4 addresses

  Indicates socket type

  Indicate protocol number, normally 0

Protocol specific! Best practice is to use \texttt{getaddrinfo} to generate the parameters automatically, so that code is protocol independent.
Each addrinfo struct returned by getaddrinfo contains arguments that can be passed directly to socket function.

Also points to a socket address struct that can be passed directly to connect and bind functions.
Sockets Interface: `socket`

- Clients and servers use the `socket` function to create a `socket descriptor`.
- Example:

  ```c
  int socket(int domain, int type, int protocol)
  
  int clientfd = Socket(AF_INET, SOCK_STREAM, 0);
  ```

  Indicates protocol number, normally 0
  A socket descriptor
  Indicates that we are using 32-bit IPV4 addresses
  Indicates socket type

Protocol specific! Best practice is to use `getaddrinfo` to generate the parameters automatically, so that code is protocol independent.
A server uses `bind` to ask the kernel to associate the server’s socket address with a socket descriptor:

- The process can read bytes that arrive on the connection whose endpoint is `addr` by reading from descriptor `sockfd`.
- Similarly, writes to `sockfd` are transferred along connection whose endpoint is `addr`.
Sockets Interface: `bind`

- A server uses `bind` to ask the kernel to associate the server’s socket address with a socket descriptor:

  ```c
  int bind(int sockfd, SA *addr, socklen_t addrlen);
  ```

  ```c
  typedef struct sockaddr SA;
  ```
A server uses `bind` to ask the kernel to associate the server's socket address with a socket descriptor:

Best practice is to use `getaddrinfo` to supply the arguments `addr` and `addrlen`.

```c
int bind(int sockfd, SA *addr, socklen_t addrlen);
```

```
typedef struct sockaddr SA;
```
Each `addrinfo` struct returned by `getaddrinfo` contains arguments that can be passed directly to `socket` function.

Also points to a socket address struct that can be passed directly to `connect` and `bind` functions.
Sockets Interface: `listen`

- By default, kernel assumes that descriptor from socket function is an *active socket* that will be on the client end of a connection.
- A server calls the `listen` function to tell the kernel that a descriptor will be used by a server rather than a client:

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

- Converts `sockfd` from an active socket to a *listening socket* that can accept connection requests from clients.
- `backlog` is a hint about the number of outstanding connection requests that the kernel should queue up before starting to refuse requests.
Client

getaddrinfo
socket
connect
send
receive
close

Server

getakndrinfo
socket
bind
listen
accept
receive
send
receive
close

open_clientfd

open_listenfd

Connection request

Await connection request from next client

Client / Server Session
A client establishes a connection with a server by calling connect:

- Attempts to establish a connection with server at socket address `addr`
  - If successful, then `clientfd` is now ready for reading and writing.
  - Resulting connection is characterized by socket pair:
    - `(x, addr.sin_addr, addr.sin_port)`
    - `x` is client address
    - `y` is ephemeral port that uniquely identifies client process on client host
  Best practice is to use `getaddrinfo` to supply the arguments `addr` and `addrlen`.

```c
int connect(int clientfd, SA *addr, socklen_t addrlen);
```
• Each addrinfo struct returned by getaddrinfo contains arguments that can be passed directly to socket function.

• Also points to a socket address struct that can be passed directly to connect and bind functions.
Sockets Interface

Client

- getaddrinfo
- socket
- connect
- send
- receive
- close

Server

- getaddrinfo
- socket
- bind
- listen
- accept
- receive
- send
- close

open_clientfd

Connection request

Await connection request from next client
Sockets Interface: `accept`

- Servers wait for connection requests from clients by calling `accept`:

```c
int accept(int listenfd, SA *addr, int *addrlen);
```

- Waits for connection request to arrive on the connection bound to `listenfd`, then fills in client’s socket address in `addr` and size of the socket address in `addrlen`.

- Returns a `connected descriptor` that can be used to communicate with the client via Unix I/O routines.
accept Illustrated

1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`
accept Illustrated

1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`

2. Client makes connection request by calling and blocking in `connect`
1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`.

2. Client makes connection request by calling and blocking in `connect`.

3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`. 
Connected vs. Listening Descriptors

• Listening descriptor
  – End point for client connection requests
  – Created once and exists for lifetime of the server

• Connected descriptor
  – End point of the connection between client and server
  – A new descriptor is created each time the server accepts a connection request from a client
  – Exists only as long as it takes to service client

• Why the distinction?
  – Allows for concurrent servers that can communicate over many client connections simultaneously
    • E.g., Each time we receive a new request, we fork a child to handle the request
ECHO SERVER
Sockets Interface

Client

- `getaddrinfo`
- `socket`
- `connect`
- `send`
- `receive`
- `close`

Server

- `getaddrinfo`
- `socket`
- `bind`
- `listen`
- `accept`
- `send`
- `receive`
- `close`

Connection request

Await connection request from next client

open_clientfd

open_listenfd
Sockets Helper: open_clientfd

- Establish a connection with a server

```c
int open_clientfd(char *hostname, char *port) {
...
```
Sockets Helper: open_clientfd

- Establish a connection with a server

```c
int open_clientfd(char *hostname, char *port) {
    int clientfd;
    struct addrinfo hints, *listp, *p;

    /* Get a list of potential server addresses */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_socktype = SOCK_STREAM; /* Open a connection */
    hints.ai_flags = AI_NUMERICSERV; /* ...using numeric port arg. */
    hints.ai_flags |= AI_ADDRCONFIG; /* Recommended for connections */
    Getaddrinfo(hostname, port, &hints, &listp);
}
```
Sockets Helper: open_clientfd (cont)

/* Walk the list for one that we can successfully connect to */
for (p = listp; p; p = p->ai_next) {
    /* Create a socket descriptor */
    if ((clientfd = socket(p->ai_family, p->ai_socktype,
                            p->ai_protocol)) < 0)
        continue; /* Socket failed, try the next */

    /* Connect to the server */
    if (connect(clientfd, p->ai_addr, p->ai_addrlen) != -1)
        break; /* Success */
    Close(clientfd); /* Connect failed, try another */
}

/* Clean up */
Freeaddrinfo(listp);
if (!p) /* All connects failed */
    return -1;
else /* The last connect succeeded */
    return clientfd;
Sockets Interface

**Client**
- `getaddrinfo`
- `socket`
- `connect`
- `send`
- `receive`
- `close`

**Server**
- `getaddrinfo`
- `socket`
- `bind`
- `listen`
- `accept`
- `receive`
- `send`
- `close`

Clients

```
open_clientfd
  
  getaddrinfo
  
  socket
  
  connect
  
  send
  
  receive
  
  close
```

Servers

```
open_listenfd
  
  getaddrinfo
  
  socket
  
  bind
  
  listen
  
  accept
  
  receive
  
  send
  
  close
```

Connection request from next client
Sockets Helper: open_listenfd

- Create a listening descriptor that can be used to accept connection requests from clients.

```c
int open_listenfd(char *port)
{
    struct addrinfo hints, *listp, *p;
    int listenfd, optval=1;

    /* Get a list of potential server addresses */
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_socktype = SOCK_STREAM;  /* Accept connect. */
    hints.ai_flags = AI_PASSIVE | AI_ADDRCONFIG;  /* ...on any IP addr */
    hints.ai_flags |= AI_NUMERICSERV;  /* ...using port no. */
    Getaddrinfo(NULL, port, &hints, &listp);
```
/* Walk the list for one that we can bind to */
for (p = listp; p; p = p->ai_next) {
    /* Create a socket descriptor */
    if ((listenfd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) < 0)
        continue; /* Socket failed, try the next */

    Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
               (const void *)&optval, sizeof(int));

    /* Bind the descriptor to the address */
    if (bind(listenfd, p->ai_addr, p->ai_addrlen) == 0)
        break; /* Success */
    Close(listenfd); /* Bind failed, try the next */
}
Sockets Helper: open_listenfd (cont)

```c
/* Clean up */
Freeaddrinfo(listp);
if (!p) /* No address worked */
    return -1;

/* Make it a listening socket ready to accept conn. requests */
if (listen(listenfd, LISTENQ) < 0) {
    Close(listenfd);
    return -1;
}
return listenfd;
```

**Key point:** open_clientfd and open_listenfd are both independent of any particular version of IP.
#include "csapp.h"

int main(int argc, char **argv)
{
    ...
    clientfd = open_clientfd(host, port);
    
    printf("type: ");
    fflush(stdout);
    
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        send(clientfd, buf, strlen(buf), 0);
        recv(clientfd, buf, MAXLINE, 0);
        
        // Display and read again:
        printf("echo: ");
        fputs(buf, stdout);
        printf("type: ");
        fflush(stdout);
    }
    ...
}
echoclient.c
Iterative Echo Server: Main Routine

```c
#include "csapp.h"

void echo(int connfd);

int main(int argc, char **argv)
{
    ...
    port = atoi(argv[1]);

    listenfd = open_listenfd(port);
    while (1) {
        clientlen = sizeof(clientaddr);
        connfd = accept(listenfd, (SA *)&clientaddr, &clientlen);
        /* determine the domain name and IP address of the client */
        hp = gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
                           sizeof(clientaddr.sin_addr.s_addr), AF_INET);
        haddrp = inet_ntoa(clientaddr.sin_addr);
        client_port = ntohs(clientaddr.sin_port);
        printf("server connected to %s (%s), port %u\n",
               hp->h_name, haddrp, client_port);
        echo(connfd);
        printf("Connection closed\n");
        close(connfd);
    }
    exit(0);
}
```

void echo (int connfd) { 
  // Local variable declarations:
  size_t n;
  char buf[MAXLINE];

  // Keep reading lines until client closes connection:
  while((n = recv(connfd, buf, MAXLINE, 0)) != 0) {
    printf("server received %d bytes\n", (int) n);
    upper_case(buf);
    send(connfd, buf, n, 0);
  }
}
**Socket Programming**

1. **Server Process**
   - `socket()`
   - `bind()`
   - `listen()`
   - `accept()` (Wait for connection())

2. **Client Process**
   - Request for connection establishment
   - `open_clientfd()`
   - `fdopen()`
   - `fprintf()`
   - `getline()` (Response)

3. **Server Response**
   - `fprintf()`
   - `getline()` (Processing)

**Flow Diagram**

- Server starts with `socket()`, `bind()`, `listen()`, and `accept()`.
- Client sends a request for connection establishment.
- Server opens the client's file descriptor and reads the request.
- Server processes the request and sends a response.
- Client reads the response.
- The diagram shows the interaction between the server and the client, highlighting the use of file operations and socket programming functions.
TCP connection

**Server Process**
- socket()
- bind()
- listen()
- accept()
  
  Wait for connection()

**Client Process**
- open_clientfd()
- fopen()
- fprintf()
- getline()

Wait for connection establishment

processing

request

response

request for connection establishment

sscanf
TCP connection

Server Process:
- socket()
- bind()
- listen()
- accept()

Wait for connection()

Client Process:
- open_clientfd()
- fopen()
- fprintf()
- getline()
- sscan

Request for connection establishment

Processing

Request

Response
i-clicker question

Which of the following option is wrong?

A. If `open_clientfd()` returns -1, it may indicate that your internet is not working.

B. client can send request to the server using `getline()` and read response from the server using `fprintf()`

C. We use `getline` to get the whole line, and `sscanf` to break the input into its pieces

D. The `fdopen()` function shall associate a stream with the socket descriptor.
Which of the following option is wrong?

A. If `open_clientfd()` returns -1, it may indicate that your internet is not working.

B. client can send request to the server using `getline()` and read response from the server using `fprintf()`

C. We use `getline` to get the whole line, and `sscanf` to break the input into its pieces

D. The `fdopen()` function shall associate a stream with the socket descriptor.