Sockets Programming in C

Prof. B.-P. Paris
ECE 465/GMU

Some material from: http://cs.ecs.baylor.edu/~donahoo/practical/CSockets/

The Main sockets functions

- socket: create the socket (server & client)
- connect: initiate connection to server (client)
- bind: Bind socket to an IP address, required before connections are received (server)
- listen: instruct socket to start looking for connections (server)
- accept: counterpart to connect (server)
- send/recev: Send and receive (server & client)
- close: close and delete socket

TCP Client/Server Interaction

Server starts by getting ready to receive client connections...

Server
1. Create a TCP socket
2. Assign a port to socket
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

Client
1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection
TCP Client/Server Interaction

/* Create socket for incoming connections */
if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    DieWithError("socket() failed");

Client
1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server
1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

echoServAddr.sin_family = AF_INET;                         /* Internet address family */
echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY);/* Any incoming interface */
echoServAddr.sin_port = htons(echoServPort);           /* Local port */

if (bind(servSock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
    DieWithError("bind() failed");

Client
1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server
1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

TCP Client/Server Interaction

/* Mark the socket so it will listen for incoming connections */
if (listen(servSock, MAXPENDING) < 0)
    DieWithError("listen() failed");

Client
1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server
1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection
TCP Client/Server Interaction

**Client**
1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

**Server**
1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

---

TCP Client/Server Interaction

Server is now blocked waiting for connection from a client

Later, a client decides to talk to the server...

**Client**
1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

**Server**
1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

---

TCP Client/Server Interaction

/* Create a reliable, stream socket using TCP */
if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
   DieWithError("socket() failed");
TCP Client/Server Interaction

Client
1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server
1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

```
echoServAddr.sin_family = AF_INET;                /* Internet address family */
echoServAddr.sin_addr.s_addr = inet_addr(servIP); /* Server IP address */
echoServAddr.sin_port = htons(echoServPort); /* Server port */

if (connect(sock, (struct sockaddr *) &echoServAddr, sizeof(echoServAddr)) < 0)
   DieWithError("connect() failed");
```

```
echoStringLen = strlen(echoString);          /* Determine input length */
/* Send the string to the server */
if (send(sock, echoString, echoStringLen, 0) != echoStringLen)
   DieWithError("send() sent a different number of bytes than expected");
```
TCP Client/Server Interaction

Client
1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server
1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a. Accept new connection
   b. Communicate
   c. Close the connection

Client Example
```c
int main(int Count, char *Strings[]) {
  int sockfd, bytes_read;
  struct sockaddr_in dest;
  char buffer[MAXBUF];

  /*---Create socket for streaming---*/
  if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
    perror("Socket");
    exit(errno);
  }

  /*---Initialize server address/port struct---*/
  bzero(&dest, sizeof(dest));
  dest.sin_family = AF_INET;
  if ( inet_addr(Strings[1], &dest.sin_addr.s_addr) == 0 ) {
    perror("Strings[1]");
    exit(errno);
  }
  dest.sin_port = htons(atoi(Strings[2]));

  /*---Establish connection---*/
  if (connect(sockfd, (struct sockaddr *)&dest, sizeof(dest)) < 0) {
    perror("connect() failed");
    exit(errno);
  }

  /*---Receive message from client---*/
  if ((recvMsgSize = recv(clntSocket, echoBuffer, RCVBUFSIZE, 0)) < 0) {
    DieWithError("recv() failed");
  }

  /*---Send message to client---*/
  if (send(sockfd, echoBuffer, RCVBUFSIZE, 0) < 0) {
    perror("send() failed");
    exit(errno);
  }

  /*---Close connection---*/
  close(sock);
  close(clntSocket);
}
```
Client Example cont’d

/*---Connect to server---*/
if ( connect(sockfd, (struct sockaddr *)&dest, sizeof(dest)) != 0 ) {
    perror("Connect");
    exit(errno);
}

/*---send message with a ‘\n’ (newline)---*/
sprintf(buffer, "%s\n", Strings[3]);
send(sockfd, buffer, strlen(buffer), 0);

/*---While there’s data, read and print it---*/
do {
    bzero(buffer, MAXBUF);
    bytes_read = recv(sockfd, buffer, MAXBUF, 0);
    if ( bytes_read > 0 ) print("%s", buffer);
} while ( bytes_read > 0 );

/*---Clean up---*/
close(sockfd);
return 0;

Example Server

int main(int Count, char *Strings[]){
    int sockfd;
    struct sockaddr_in self;
    char buffer[MAXBUF];
    /*---Create streaming socket---*/
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
        perror("Socket");
        exit(errno);
    }
    /*---Initialize address/port structure---*/
bzero(&self, sizeof(self));
    self.sin_family = AF_INET;
    self.sin_port = htons(MY_PORT);
    self.sin_addr.s_addr = INADDR_ANY;
    /*---Assign a port number to the socket---*/
    if ( bind(sockfd, (struct sockaddr *)&self, sizeof(self)) != 0 ) {
        perror("socket--bind");
        exit(errno);
    }
    /*---Make it a "listening socket"---*/
    if ( listen(sockfd, 20) != 0 ) {
        perror("socket--listen");
        exit(errno);
    }
    /*---Forever... ---*/
    while (1){
        int clientfd;
        struct sockaddr_in client_addr;
        int addrlen=sizeof(client_addr);
        /*---accept a connection (creating a data pipe)---*/
        clientfd = accept(sockfd, (struct sockaddr*)&client_addr, &addrlen);
        printf("%s:%d connected\n", inet_ntoa(client_addr.sin_addr),
            ntohs(client_addr.sin_port));
        /*---Echo back anything sent---*/
        send(clientfd, buffer, recv(clientfd, buffer, MAXBUF, 0), 0);
        /*---Close data connection---*/
        close(clientfd);
    }
}
**UDP Sockets**

- UDP is fully supported by sockets.
- Differences to TCP sockets include:
  - Use SOCK_DGRAM instead of SOCK_STREAM in socket() call.
  - The functions connect() and accept() are not used, since UDP is connectionless.
  - Use recvfrom() instead of recv() and sendto() instead of send().
    - Both have address information as an additional parameter.

**UDP Client**

```c
int main(int Count, char *Strings[]) {
    char buffer[BUFSIZE];
    struct sockaddr_in addr;
    int sd, addr_size;
    if ( (sd = socket(PF_INET, SOCK_DGRAM, 0)) < 0 ) {
        perror("Socket");
        abort();
    }
    addr.sin_family = AF_INET;
    addr.sin_port = htons(9999);
    if ( inet_aton("127.0.0.1", &addr.sin_addr) == 0 ) {
        perror("127.0.0.1");
        abort();
    }
    sendto(sd, Strings[1], strlen(Strings[1]) + 1, 0,
        (struct sockaddr*)&addr, sizeof(addr));
    bzero(buffer, BUFSIZE);
    addr_size = sizeof(addr);
    if ( recvfrom(sd, buffer, BUFSIZE, 0,
        (struct sockaddr*)&addr, &addr_size) < 0 ) {
        perror("server reply");
    } else
        printf("Reply: %s:%d "%s",
            inet_ntoa(addr.sin_addr), ntohs(addr.sin_port), buffer);
    close(sd);
    return 0;
}
```
int main() {
    char buffer[BUFSIZE];
    struct sockaddr_in addr;
    int sd, addr_size, bytes_read;
    sd = socket(PF_INET, SOCK_DGRAM, 0);
    if (sd < 0) {
        perror("socket");
        abort();
    }
    addr.sin_family = AF_INET;
    addr.sin_port = htons(9999);
    addr.sin_addr.s_addr = INADDR_ANY;
    if (bind(sd, (struct sockaddr*) &addr, sizeof(addr)) < 0) {
        perror("bind");
        abort();
    }

    do {
        bzero(buffer, BUFSIZE);
        addr_size = BUFSIZE;
        bytes_read = recvfrom(sd, buffer, BUFSIZE, 0,
                               (struct sockaddr*) &addr, &addr_size);
        if (bytes_read > 0) {
            printf("Connect: %s:%d \"%s\"

                    Connect: ");
            inet_ntoa(addr.sin_addr), ntohs(addr.sin_port), buffer);
            alltoupper(buffer);
            if (sendto(sd, buffer, bytes_read, 0,
                        (struct sockaddr*) &addr, addr_size) < 0) {
                perror("reply");
            } else
                perror("recvfrom");
        } else
            perror("recvfrom");
    } while (bytes_read > 0);
    close(sd);
    return 0; }

The sockaddr_in structure

/* a structure to contain an internet address defined in the include file in.h */
struct sockaddr_in {
    short sin_family; /* should be AF_INET */
    u_short sin_port; /* 16 bit port number */
    struct in_addr sin_addr;
    char sin_zero[8]; /* not used, must be zero */
};

struct in_addr {
    unsigned long s_addr; /* 32 bit IP address */
};
Multiple Recipients: broadcast and multicast

- Messages can be sent to more than one recipient using either broadcast or multicast.
- Both broadcast and multicast can only use UDP. Obviously!
- Broadcast addresses:
  - 255.255.255.255 all hosts on the local network.
  - 129.174.135.255 all hosts whose IP address begins with 129.174.135.
- Multicast addresses:
  - All addresses between 224.0.0.0 and 239.255.255.255.
  - Think of multicast IP addresses as “meeting points” for groups of hosts.
  - Hosts don’t have to be on the same subnet.
  - Multicast routing is not supported throughout the entire internet.
  - Hosts must still be listening for such messages to receive them.
  - Only difference, more than one receiver is possible.
- Application: discovering hosts on the network

Multicast and Broadcast with sockets

- Principally, broadcasting and multicasting is just like unicasting with UDP.
- Broadcast sender must set broadcast permission:
  ```
  broadcastPermission = 1
  setsockopt(sock, SOL_SOCKET, SO_BROADCAST, (void *)&broadcastPermission, sizeof(broadcastPermission))
  ```
- Multicast sender must set multicast TTL value:
  ```
  setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, (void *)&multicastTTL, sizeof(multicastTTL))
  ```
- Multicast receiver must join multicast group:
  ```
  See MulticastReceiver.c for syntax.
  ```
- Logically, for use in discovery:
  - The “client” use an infinite loop to make periodic announcements.
  - The “server” looks for an announcement until one has been received.
Dealing with Concurrency

- Often, especially for servers, it is necessary to do more than one operation at a time (concurrently).
  - E.g., communicate with multiple clients.
- There are essentially two ways to support concurrency:
  - Multiple processes or threads
  - Asynchronous I/O multiplexing (via select() function)

Server w/ multiple processes

- Idea:
  - create a new process (child) that handles the actual work.
  - Allow original process (parent) to return immediately to monitoring server socket.
- New process is created by fork() system call.
  - fork() duplicates the current process.
  - fork() is called immediately after accept().
  - In parent, fork() returns a positive number.
  - In child, fork() returns zero.
  - Parent must explicitly acknowledge that child has exited via a signal handler.

Server w/ multiple processes

```c
int main(void) {
    int sd;
    struct sigaction act;
    struct sockaddr_in addr;
    /* Install a signal handler to deal with completed spawned processes*/
    bzero(&act, sizeof(act));
    act.sa_handler = sig_handler;
    act.sa_flags = SA_NOCLODSTOP;
    sigaction(SIGCHLD, &act, 0);
    if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0 )
        PANIC("Socket");
    addr.sin_family = AF_INET;
    addr.sin_port = htons(9999);
    addr.sin_addr.s_addr = INADDR_ANY;
    if ( bind(sd, (struct sockaddr*)&addr, sizeof(addr)) != 0 )
        PANIC("Bind");
    if ( listen(sd, 20) != 0 )
        PANIC("Listen");
    return 0;
}
```
Server cont’d

while (1) {
  int client, addr_size = sizeof(addr);
  client = accept(sd, (struct sockaddr*)&addr, &addr_size);
  if (client < 0) {
    perror("Accept");
    continue;
  }
  printf("Connected: %s:%d\n", inet_ntoa(addr.sin_addr), ntohs(addr.sin_port));
  if (fork() ) /* Spawn */
    /* Parent */
    close(client);
  else {
    /* Child */
    close(sd);
    Child(client); /* Talk to Client */
    exit(0);
  }
}  

return 0;

Subroutine Child()

Carries on actual conversation with client.
Is passed file descriptor for Client socket

void Child(void* arg) {
  char line[100];
  int bytes_read;
  int client = *(int*)arg; /* Client Socket file descriptor */
  do { bytes_read = recv(client, line, sizeof(line), 0);
    if (bytes_read < 0)
      perror("Read socket");
    send(client, line, bytes_read, 0);
  } while (strncmp(line, "bye\r", 4) != 0 || bytes_read < 0);
  close(client);
  exit(0);
}

Signal Handler

- When the child process finishes, the parent is informed (via signal SIGCHLD).
- Parent has to acknowledge this signal to free up child process.
  - Otherwise, they turn into Zombies
  - The wait() function does this.
  
void sig_handler(int sig) {
  if (sig == SIGCHLD) {
    int retval;
    wait(&retval);
  }
}
Server with Threads

- An alternative to multiple processes are threads.
- Think of threads as lightweight processes.
- They incur less overhead than processes.
- Structure of resulting program is similar to above.
- Instead of `fork()`, main thread call `pthread_create()`.
- One of the arguments to `pthread_create()` is a pointer to a subroutine which the thread will execute.
- If the thread is detached (via `pthread_detach()`) no clean-up is necessary.

Server main loop

```c
while (1)
{
    int client, addr_size = sizeof(addr);
    pthread_t child;

    printf("PARENT: waiting to accept ...\n");
    client = accept(sd, (struct sockaddr*)&addr, &addr_size);
    printf("PARENT: Connected: %s:%d\n", inet_ntoa(addr.sin_addr), ntohs(addr.sin_port));
    if (pthread_create(&child, NULL, Child, &client) != 0)
    {
        perror("Thread creation");
    } else {
        printf("PARENT: Thread created\n");
        pthread_detach(child); /* disassociate from parent */
        printf("PARENT: Thread detached\n");
    }
}
```

Child subroutine (servlet)

```c
/* Child subroutine */
void* Child(void* arg)
{
    char line[100];
    int bytes_read;
    int client = *(int*)arg;

    bzero(line, 100);
    /*do
    printf("CHILD: Processing request ...\n");
    bytes_read = recv(client, line, sizeof(line), 0);
    send(client, line, bytes_read, 0);
    */
    while (strcmp(line, "bye\r") != 0)
    {
        printf("CHILD: Sending \"%s\"\n", line);
        close(client);
        printf("CHILD: Returning \"\n");
        return arg;
    }
}
Asynchronous I/O Multiplexing

- Functions like `accept()` or `recv()` are blocking.
- The system cannot do anything else until they return.
- E.g., when a new connection arrives `accept()` returns.
- The problem could be prevented if there is a mechanism to alert us when “interesting” things happen.
- The standard IO (stdio) function `select()` does just that:
  - Given a set of file descriptors (including sockets), `select` returns whenever these file descriptors need attention.
  - This approach is often called an event loop.

```
Simple select() Example

```