Sockets Tutorial

Reference: https://www.linuxhowtos.org/C_C++/socket.htm

This is a simple tutorial on using sockets for interprocess communication.

The client server model

Most interprocess communication uses the client server model. These terms refer to the two processes which will be communicating with each other. One of the two processes, the client, connects to the other process, the server, typically to make a request for information. A good analogy is a person who makes a phone call to another person.

Notice that the client needs to know of the existence of and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established. Notice also that once a connection is established, both sides can send and receive information.

The system calls for establishing a connection are somewhat different for the client and the server, but both involve the basic construct of a socket. A socket is one end of an interprocess communication channel. The two processes each establish their own socket.

establishing a socket on the client side	establishing a socket on the server side
• Create a socket with the **socket()** system call • Connect the socket to the address of the server using the **connect()** system call • Send and receive data. There are a number of ways to do this, but the simplest is to use the **read()** and **write()** system calls.	• Create a socket with the **socket()** system call • Bind the socket to an address using the **bind()** system call. For a server socket on the Internet, an address consists of a port number on the host machine. • Listen for connections with the **listen()** system call • Accept a connection with the **accept()** system call. This call typically blocks until a client connects with the server. • Send and receive data

Socket Types

When a socket is created, the program has to specify the **address domain** and the **socket type**. Two processes can communicate with each other only if their sockets are of the same type and in the same domain.

There are two widely used address domains, the **unix domain** and the Internet domain. Each of these has its own address format.

unix domain	**Internet domain**
two processes which share a common file system communicate. The address of a socket is a character string which is basically an entry in the file system.	two processes running on any two hosts on the Internet communicate. The address of a socket in the Internet domain consists of the Internet address of the host machine (every computer on the Internet has a unique 32 bit address, often referred to as its IP address).

In addition, each socket needs a port number on that host. Port numbers are 16 bit unsigned integers.

The lower numbers are reserved in Unix for standard services. For example, the port number for the FTP server is 21. It is important that standard services be at the same port on all computers so that clients will know their addresses.

However, port numbers above 2000 are generally available.

There are two widely used socket types, stream sockets, and datagram sockets. Each uses its own communciations protocol.

stream sockets	**datagram sockets**
Stream sockets treat communications as a continuous stream of characters. Stream sockets use TCP (Transmission Control Protocol), which is a reliable, stream oriented protocol	datagram sockets have to read entire messages at once. datagram sockets use UDP (Unix Datagram Protocol), which is unreliable and message oriented.

The examples in this tutorial will use sockets in the Internet domain using the TCP protocol.

Sample code

C code for a very simple client and server are provided for you. These communicate using stream sockets in the Internet domain. The code is described in detail below. However, before you read the descriptions and look at the code, you should compile and run the two programs to see what they do.

server.c

```
/* A simple server in the internet domain using TCP
   The port number is passed as an argument */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void error(const char *msg)
{
    perror(msg);
    exit(1);
}

int main(int argc, char *argv[])
{
     int sockfd, newsockfd, portno;
     socklen_t clilen;
     char buffer[256];
     struct sockaddr_in serv_addr, cli_addr;
     int n;
     if (argc < 2) {
         fprintf(stderr,"ERROR, no port provided\n");
         exit(1);
     }
     sockfd = socket(AF_INET, SOCK_STREAM, 0);
     if (sockfd < 0)
        error("ERROR opening socket");
     bzero((char *) &serv_addr, sizeof(serv_addr));
     portno = atoi(argv[1]);
     serv_addr.sin_family = AF_INET;
     serv_addr.sin_addr.s_addr = INADDR_ANY;
```

```
     serv_addr.sin_port = htons(portno);
     if (bind(sockfd, (struct sockaddr *) &serv_addr,
              sizeof(serv_addr)) < 0)
              error("ERROR on binding");
     listen(sockfd,5);
     while(1){
          clilen = sizeof(cli_addr);
          newsockfd = accept(sockfd,
                 (struct sockaddr *) &cli_addr,
                 &clilen);
          if (newsockfd < 0)
               error("ERROR on accept");
          bzero(buffer,256);
          n = read(newsockfd,buffer,255);
          if (n < 0) error("ERROR reading from socket");
          printf("Here is the message: %s\n",buffer);
          n = write(newsockfd,"I got your message",18);
          if (n < 0) error("ERROR writing to socket");
          close(newsockfd);
          //close(sockfd);
     }
     return 0;
}
```

client.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void error(const char *msg)
{
    perror(msg);
    exit(0);
}

int main(int argc, char *argv[])
{
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;

    char buffer[256];
    if (argc < 3) {
       fprintf(stderr,"usage %s hostname port\n", argv[0]);
       exit(0);
       return 1;
    }
    portno = atoi(argv[2]);
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");
    server = gethostbyname(argv[1]);
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);
        return 1;
    }
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
         (char *)&serv_addr.sin_addr.s_addr,
         server->h_length);
    serv_addr.sin_port = htons(portno);
    if (connect(sockfd,(struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0){
        error("ERROR connecting");
        return 1;
    }
    printf("Please enter the message: ");
    bzero(buffer,256);
    fgets(buffer,255,stdin);
```

```
    n = write(sockfd,buffer,strlen(buffer));
    if (n < 0){
         error("ERROR writing to socket");
         return 1;
    }
    bzero(buffer,256);
    n = read(sockfd,buffer,255);
    if (n < 0){
         error("ERROR reading from socket");
         return 1;
    }
    printf("%s\n",buffer);
    close(sockfd);
    return 0;
}
```

Enhancements to the server code

The sample server code above has the limitation that it only handles one connection, and then dies. A "real world" server should run indefinitely and should have the capability of handling a number of simultaneous connections, each in its own process. This is typically done by forking off a new process to handle each new connection.

The following code has a dummy function called dostuff(int sockfd).

This function will handle the connection after it has been established and provide whatever services the client requests. As we saw above, once a connection is established, both ends can use read and write to send information to the other end, and the details of the information passed back and forth do not concern us here.

To write a "real world" server, you would make essentially no changes to the main() function, and all of the code which provided the service would be in dostuff().

To allow the server to handle multiple simultaneous connections, we make the following changes to the code:

1. Put the accept statement and the following code in an infinite loop.

2. After a connection is established, call fork()#### to create a new process.

3. The child process will close sockfd#### and call #dostuff#####, passing the new socket file descriptor as an argument. When the two processes have completed their conversation, as indicated by dostuff()#### returning, this process simply exits.

4. The parent process closes newsockfd####. Because all of this code is in an infinite loop, it will return to the accept statement to wait for the next connection.

Here is the code.

```
while (1)
 {
  newsockfd = accept(sockfd,
          (struct sockaddr *) &cli_addr, &clilen);
  if (newsockfd < 0)
    error("ERROR on accept");
  pid = fork();
  if (pid < 0)
    error("ERROR on fork");
  if (pid == 0)
  {
    close(sockfd);
```

```
    dostuff(newsockfd);
    exit(0);
  }
  else
    close(newsockfd);
 } /* end of while */
```

for a complete server program which includes this change. This will run with the program client.c.

```
/* A simple server in the internet domain using TCP
   The port number is passed as an argument
   This version runs forever, forking off a separate
   process for each connection
*/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void dostuff(int); /* function prototype */
void error(const char *msg)
{
    perror(msg);
    exit(1);
}

int main(int argc, char *argv[])
{
     int sockfd, newsockfd, portno, pid;
     socklen_t clilen;
     struct sockaddr_in serv_addr, cli_addr;

     if (argc < 2) {
         fprintf(stderr,"ERROR, no port provided\n");
         exit(1);
     }
     sockfd = socket(AF_INET, SOCK_STREAM, 0);
     if (sockfd < 0)
        error("ERROR opening socket");
     bzero((char *) &serv_addr, sizeof(serv_addr));
     portno = atoi(argv[1]);
     serv_addr.sin_family = AF_INET;
     serv_addr.sin_addr.s_addr = INADDR_ANY;
     serv_addr.sin_port = htons(portno);
     if (bind(sockfd, (struct sockaddr *) &serv_addr,
              sizeof(serv_addr)) < 0)
              error("ERROR on binding");
     listen(sockfd,5);
     clilen = sizeof(cli_addr);
     while (1) {
         newsockfd = accept(sockfd,
               (struct sockaddr *) &cli_addr, &clilen);
         if (newsockfd < 0)
             error("ERROR on accept");
         pid = fork();
         if (pid < 0)
             error("ERROR on fork");
         if (pid == 0)  {
             close(sockfd);
             dostuff(newsockfd);
```

```
            exit(0);
        }
        else close(newsockfd);
    } /* end of while */
    close(sockfd);
    return 0; /* we never get here */
}

/******** DOSTUFF() *********************
 There is a separate instance of this function
 for each connection.  It handles all communication
 once a connnection has been established.
 *****************************************/
void dostuff (int sock)
{
   int n;
   char buffer[256];

   bzero(buffer,256);
   n = read(sock,buffer,255);
   if (n < 0) error("ERROR reading from socket");
   printf("Here is the message: %s\n",buffer);
   n = write(sock,"I got your message",18);
   if (n < 0) error("ERROR writing to socket");
}
```

The zombie problem

The above code has a problem; if the parent runs for a long time and accepts many connections, each of these connections will create a zombie when the connection is terminated. A zombie is a process which has terminated but cannot be permitted to fully die because at some point in the future, the parent of the process might execute a wait and would want information about the death of the child. Zombies clog up the process table in the kernel, and so they should be prevented. Unfortunately, the code which prevents zombies is not consistent across different architectures. When a child dies, it sends a SIGCHLD signal to its parent. On systems such as AIX, the following code in main() is all that is needed.

```
signal(SIGCHLD,SIG_IGN);
```

This says to ignore the SIGCHLD signal. However, on systems running SunOS, you have to use the following code:

```
void *SigCatcher(int n)
{
  wait3(NULL,WNOHANG,NULL);
}
...
int main()
{
  ...
  signal(SIGCHLD,SigCatcher);
  ...
```

The function SigCatcher() will be called whenever the parent receives a SIGCHLD signal (i.e. whenever a child dies). This will in turn call wait3 which will receive the signal. The WNOHANG flag is set, which causes this to be a non-blocking wait (one of my favorite oxymorons).

Alternative types of sockets

This example showed a stream socket in the Internet domain. This is the most common type of connection. A second type of connection is a datagram socket. You might want to use a datagram socket in cases where there is only one message being sent from the client to the server, and only one message being sent back. There are several differences between a datagram socket and a stream socket.

Datagram socket	Stream socket
unreliable, which means that if a packet of information gets lost somewhere in the Internet, the sender is not told (and of course the receiver does not know about the existence of the message)	the underlying TCP protocol will detect that a message was lost because it was not acknowledged, and it will be retransmitted without the process at either end knowing about this.
Message boundaries are preserved in datagram sockets. If the sender sends a datagram of 100 bytes, the receiver must read all 100 bytes at once	if the sender wrote a 100 byte message, the receiver could read it in two chunks of 50 bytes or 100 chunks of one byte.
The communication is done using special system calls sendto() and receivefrom()	The communication is done using system calls read() and write()

There is a lot less overhead associated with a datagram socket because connections do not need to be established and broken down, and packets do not need to be acknowledged. This is why datagram sockets are often used when the service to be provided is short, such as a time-of-day service.

The following codes are for datagram socket server and client

server.c

```
/* Creates a datagram server.  The port
   number is passed as an argument.  This
   server runs forever */

#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <netdb.h>
#include <stdio.h>

void error(const char *msg)
{
    perror(msg);
    exit(0);
}

int main(int argc, char *argv[])
{
   int sock, length, n;
   socklen_t fromlen;
   struct sockaddr_in server;
   struct sockaddr_in from;
```

```
  char buf[1024];

  if (argc < 2) {
    fprintf(stderr, "ERROR, no port provided\n");
    exit(0);
  }

  sock=socket(AF_INET, SOCK_DGRAM, 0);
  if (sock < 0) error("Opening socket");
  length = sizeof(server);
  bzero(&server,length);
  server.sin_family=AF_INET;
  server.sin_addr.s_addr=INADDR_ANY;
  server.sin_port=htons(atoi(argv[1]));
  if (bind(sock,(struct sockaddr *)&server,length)<0)
     error("binding");
  fromlen = sizeof(struct sockaddr_in);
  while (1) {
     n = recvfrom(sock,buf,1024,0,(struct sockaddr *)&from,&fromlen);
     if (n < 0) error("recvfrom");
     write(1,"Received a datagram: ",21);
     write(1,buf,n);
     n = sendto(sock,"Got your message\n",17,
            0,(struct sockaddr *)&from,fromlen);
     if (n  < 0) error("sendto");
  }
  return 0;
}
```

client.c

```
/* UDP client in the internet domain */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

void error(const char *);
int main(int argc, char *argv[])
{
  int sock, n;
  unsigned int length;
  struct sockaddr_in server, from;
  struct hostent *hp;
  char buffer[256];

  if (argc != 3) { printf("Usage: server port\n");
               exit(1);
  }
  sock= socket(AF_INET, SOCK_DGRAM, 0);
  if (sock < 0) error("socket");

  server.sin_family = AF_INET;
  hp = gethostbyname(argv[1]);
  if (hp==0) error("Unknown host");

  bcopy((char *)hp->h_addr,
      (char *)&server.sin_addr,
       hp->h_length);
```

```
   server.sin_port = htons(atoi(argv[2]));
   length=sizeof(struct sockaddr_in);
   printf("Please enter the message: ");
   bzero(buffer,256);
   fgets(buffer,255,stdin);
   n=sendto(sock,buffer,
         strlen(buffer),0,(const struct sockaddr *)&server,length);
   if (n < 0) error("Sendto");
   n = recvfrom(sock,buffer,256,0,(struct sockaddr *)&from, &length);
   if (n < 0) error("recvfrom");
   write(1,"Got an ack: ",12);
   write(1,buffer,n);
   close(sock);
   return 0;
}

void error(const char *msg)
{
    perror(msg);
    exit(0);
}
```

These two programs can be compiled and run in exactly the same way as the server and client using a stream socket.