Chapter 15

Unix Domain Protocols

Dibyasundar Das

Department of Computer Science and Information Technology
Institute of Technical Education and Research,
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India

Contents

Introduction

What is Unix domain protocol?

The Unix domain protocols are not an actual protocol suite, but a way of performing client/server communication on a single host using the same API that is used for clients and servers on different hosts.

Why we need Unix domain protocol?

1. **Faster :** Unix domain sockets are often twice as fast as a TCP socket when both peers are on the same host.
2. **Convenient :** We can pass descriptors between process by using Unix domain protocol.
3. **Secure :** Unix domain sockets provide the client's credentials to the server, which can provide additional security checking.

Unix Domain Socket Address Structure

```
struct sockaddr_un {
sa_family_t sun_family; /*AF_LOCAL*/
char sun_path[104]; /*null-terminated pathname*/
};
```

- **Unix domain socket address structure, which is defined by including the <sys/un.h> header.**
- **The path name stored in the *sun_path* array must be null-terminated. The unspecified address is indicated by a null string as the pathname, that is, a structure with *sun_path*[0] equal to 0. This is the Unix domain equivalent of the IPv4 *INADDR_ANY* constant and the IPv6 *IN6ADDR_ANY_INIT* constant.**

Binding Socket Names

Why we need to bind name?

- A socket is created without a name. To be used, it must be given a name so that processes can reference it and messages can be received on it.
- Communicating processes are bound by an association. Unlike the Internet domain, an association in the UNIX domain is composed of local and foreign pathnames. (A "foreign pathname" is a pathname created by a foreign process, not a pathname on a foreign system).
- **UNIX domain socket names, like file pathnames, may be either absolute (like /dev/socket) or relative (like ../socket).**

Binding Socket Names (contd.)

Why unlink is necessary?

When a name is bound into the name space, a file is created in the file system. If the file is not removed (unlinked), the name will continue to exist even after the bound socket is closed. This can cause subsequent runs of a program to find that a name is unavailable for use,

NB: Names in the UNIX domain are only used for establishing connections. They are not used for message delivery once a connection is established. Therefore, in contrast with the Internet domain, unbound sockets need not be (and are not) automatically given addresses when they are connected.

Binding Socket Names (contd.) (Example 1)

```
#include<stdio.h>
#include<sys/socket.h>
#include <sys/un.h>
#include<string.h>
#include <unistd.h>
#include<stdlib.h>
int main(int argc, char *argv[])
{
        int sockfd;
        socklen_t len;
        struct sockaddr_un addr1, addr2;
        if (argc != 2)
        {
                printf("Enter path name");
                exit(0);
        }
        sockfd = socket(AF_LOCAL, SOCK_STREAM, 0);
        unlink(argv[1]);

        bzero(&addr1, sizeof(addr1));

        addr1.sun_family = AF_LOCAL;
        strcpy(addr1.sun_path, argv[1]);
        bind(sockfd, (struct sockaddr *) &addr1, strlen(addr1.sun_path)
                                +sizeof (addr1.sun_family));
        len = sizeof(addr2);
        getsockname(sockfd, (struct sockaddr *) &addr2, &len);
        printf("bound name = %s, returned len = %d\n", addr2.sun_path, len);
        return(0);
}
```

Binding Socket Names (contd.) (Example 1 explanation)

- Whenever, we try to bind a Unix domain socket we must provide a file path name. In the above program we are going take a file path at the runtime by *argv* variable.
- Then we must try to unlink or remove the file if exist. If the unlink command fails then the file does not exist. So, we need not handle if it fails.
- Then we copy the path to sun_path and call the bind function.
- **NB: The length argument of the bind function must take sum of address family and length of the path string.**
- If the code runs successfully a file will be created in the given name which will act as Unix domain socket name.
- the last part of the code prints the path and size of the socket address.

Binding Socket Names (contd.) (Example 1 Execution)

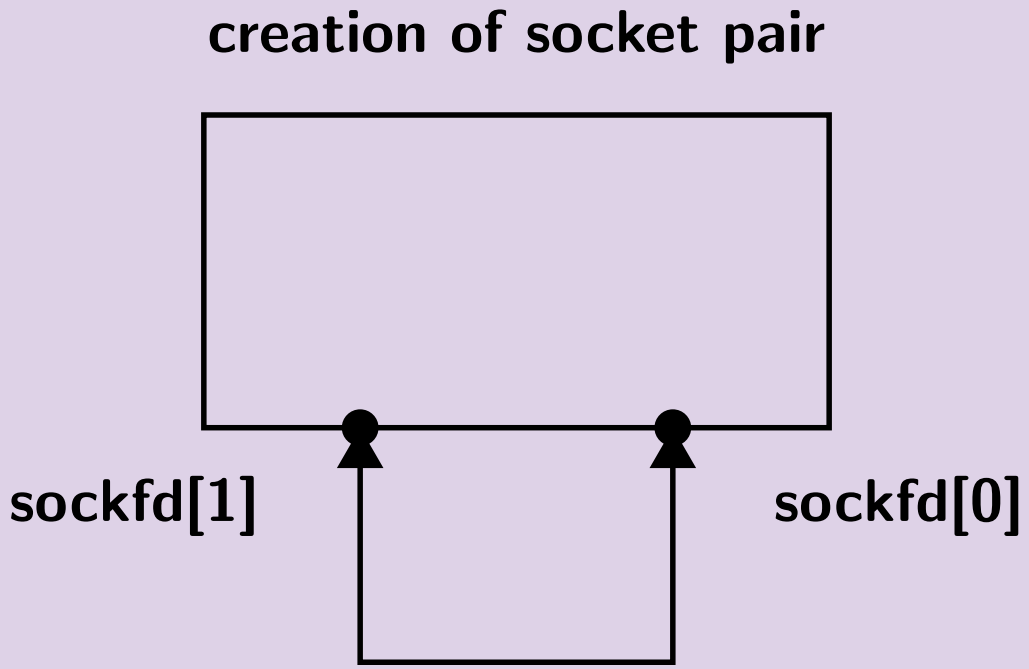
```
drago@drago-pc:~/Desktop/UNP-main/Unix Sockets$ mkdir temp
drago@drago-pc:~/Desktop/UNP-main/Unix Sockets$ ls temp
drago@drago-pc:~/Desktop/UNP-main/Unix Sockets$ gcc bind_emaple.c -o bind_emaple.o
drago@drago-pc:~/Desktop/UNP-main/Unix Sockets$ ./bind_emaple.o temp/new_socket
bound name = temp/new_socket, returned len = 18
drago@drago-pc:~/Desktop/UNP-main/Unix Sockets$ ls temp
new_socket
drago@drago-pc:~/Desktop/UNP-main/Unix Sockets$ ls -l temp
total 0
srwxrwxr-x 1 drago drago 0 Jun 15 08:52 new_socket
drago@drago-pc:~/Desktop/UNP-main/Unix Sockets$
```

- We created a temporary directory and check the available by **ls** command
- In next step we compiled the program.
- When running the object file we provide the relative path(in temp directory) of the file.
- In last we verified if the file is created in temp folder by **ls -l** command.

socketpair Function

Basic code

```
#include <sys/socket.h>
int socketpair(int family, int type, int protocol,
                                        int sockfd[2]);
```



socketpair Function (contd.)

- The *socketpair* function creates two sockets that are then connected together
- Returns: nonzero if OK, -1 on error
- The family must be **AF_LOCAL** and the protocol must be 0. The type, however, can be either **SOCK_STREAM** or **SOCK_DGRAM**. The two socket descriptors that are created are returned as *sockfd*[0] and *sockfd*[1].
- The two created sockets are unnamed; that is, there is no implicit **bind** involved.
- The result of socketpair with a type of **SOCK_STREAM** is called a stream pipe. It is similar to a regular Unix pipe (created by the pipe function), but a stream pipe is full-duplex; that is, both descriptors can be read and written.

Socket Function

There are several differences and restrictions in the socket functions when using Unix domain sockets. Which are as follows:

- The default file access permissions for a pathname created by bind should be 0777.
- The pathname associated with a Unix domain socket should be an absolute pathname, not a relative pathname. Binding a relative pathname to a Unix domain socket gives unpredictable results.
- The pathname specified in a call to connect must be a pathname that is currently bound to an open Unix domain socket of the same type (stream or datagram).
  Errors occur if:
  1. the pathname exists but is not a socket
  2. the pathname exists and is a socket, but no open socket descriptor is associated with the pathname
  3. the pathname exists and is an open socket, but is of the wrong type

Socket Function (contd.)

- The permission testing associated with the connect of a Unix domain socket is the same as if open had been called for write-only access to the pathname.
- Unix domain stream sockets are similar to TCP sockets.
- If a call to connect for a Unix domain stream socket finds that the listening socket's queue is full **ECONNREFUSED** is returned immediately i.e. connection is refused error will be recieved.
- Unix domain datagram sockets are similar to UDP sockets: They provide an unreliable datagram service that preserves record boundaries.
- Unlike UDP sockets, sending a datagram on an unbound Unix domain datagram socket does not bind a pathname to the socket.

Unix Domain Stream Client/Server

server code

```
#include<stdio.h>
#include<sys/socket.h>
#include <sys/un.h>
#include<string.h>
#include <unistd.h>
#include<stdlib.h>
#include<signal.h>
#include<errno.h>
int sockfd;

void handle_int(int sig)
{
        close(sockfd);
        printf("Exiting␣Safely....\n");
        exit(0);
}

int main(int argc, char *argv[])
{
        int connfd, clilen;
        pid_t childpid;
```

Unix Domain Stream Client/Server (contd.)

```
char data[201];
struct sockaddr_un addr1, addr2;
if (argc != 2)
{
        printf("Enter␣path␣name");
        exit(0);
}
sockfd = socket(AF_LOCAL, SOCK_STREAM, 0);
unlink(argv[1]);

bzero(&addr1, sizeof(addr1));

addr1.sun_family = AF_LOCAL;
strcpy(addr1.sun_path, argv[1]);
bind(sockfd, (struct sockaddr *) &addr1,
                        strlen(addr1.sun_path)
                        +sizeof (addr1.sun_family));

listen(sockfd, 10);

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

Unix Domain Stream Client/Server (contd.)

```
while(1)
{
        clilen = sizeof(addr2);
        if ( (connfd = accept(sockfd,
                (struct sockaddr *) &addr2,
                 &clilen)) < 0)
        {
                if (errno == EINTR)
                        continue;
                else
                        printf("error");
                }
                if ( (childpid = fork()) == 0)
                {
                  close(sockfd);
                  while(recv(connfd, data,200,0) != 0)
                  {
                    write(connfd,data,strlen(data));
                  }
                  exit(0);
                }
```

Unix Domain Stream Client/Server (contd.)

```
                        else
                        {
                                close(connfd);
                        }
        }
        return(0);
}
```

The program is available as "Unix stream server.c" file.

Unix Domain Stream Client/Server (contd.)

client code

```
#include<stdio.h>
#include<sys/socket.h>
#include <sys/un.h>
#include<string.h>
#include <unistd.h>
#include<stdlib.h>
#include<signal.h>

int main(int argc, char *argv[])
{
        int sockfd;
        char data[201];
        struct sockaddr_un addr1;
        if (argc != 2)
        {
                printf("Enter␣path␣name");
                exit(0);
        }
        sockfd = socket(AF_LOCAL, SOCK_STREAM, 0);

        bzero(&addr1, sizeof(addr1));
```

Unix Domain Stream Client/Server (contd.)

```
        addr1.sun_family = AF_LOCAL;
        strcpy(addr1.sun_path, argv[1]);
        connect(sockfd, (struct sockaddr *) &addr1,
                        sizeof(addr1));
        while (fgets(data, 200, stdin) !=NULL)
        {
                write(sockfd,data,strlen(data));
                recv(sockfd, data,200,0);
                printf("%s",data);
        }
        close(sockfd);
        return(0);
}
```

The program is available as "Unix stream client.c" file.

Unix Domain Stream Client/Server (contd.)

Execution of Server and client

```
drago@drago-pc:~/Desktop/UNP-main/Unix Sockets$ gcc Unix\ stream\ server.c -o server.o
drago@drago-pc:~/Desktop/UNP-main/Unix Sockets$ ./server.o temp/new_socket
```

```
drago@drago-pc:~/Desktop/UNP-main/Unix Sockets$ gcc Unix\ stream\ client.c -o client.o
drago@drago-pc:~/Desktop/UNP-main/Unix Sockets$ ./client.o temp/new_socket
dibya
dibya
sundar
sundar
```

Unix Domain Datagram Client/Server

server code

```
#include<stdio.h>
#include<sys/socket.h>
#include <sys/un.h>
#include<string.h>
#include <unistd.h>
#include<stdlib.h>

int main(int argc, char *argv[])
{
        int connfd, clilen;
        pid_t childpid;
        char data[201];
        struct sockaddr_un addr1, addr2;
        if (argc != 2)
        {
                printf("Enter␣path␣name");
                exit(0);
        }
        connfd = socket(AF_LOCAL, SOCK_DGRAM, 0);
        unlink(argv[1]);
```

Unix Domain Datagram Client/Server (contd.)

```
        bzero(&addr1, sizeof(addr1));
        addr1.sun_family = AF_LOCAL;
        strcpy(addr1.sun_path, argv[1]);
        bind(connfd, (struct sockaddr *) &addr1,
                        sizeof (addr1));

        clilen=sizeof(addr2);
        while(1)
        {
                recvfrom(connfd, data,200,0,
                                (struct sockaddr*)&addr2,
                                                &clilen);
                sendto(connfd, data, strlen(data), 0,
                                (struct sockaddr *)&addr2,
                                        clilen);
        }

        return(0);
}
```

The code is available as "Unix datagram server.c"

Unix Domain Datagram Client/Server (contd.)

client code

```
#include<stdio.h>
#include<sys/socket.h>
#include <sys/un.h>
#include<string.h>
#include <unistd.h>
#include<stdlib.h>

int main(int argc, char *argv[])
{
        int connfd, clilen;
        pid_t childpid;
        char data[201];
        struct sockaddr_un addr1, addr2;
        if (argc != 2)
        {
                printf("Enter␣path␣name");
                exit(0);
        }
        connfd = socket(AF_LOCAL, SOCK_DGRAM, 0);

        bzero(&addr1, sizeof(addr1));
```

Unix Domain Datagram Client/Server (contd.)

```
        addr1.sun_family = AF_LOCAL;
        strcpy(addr1.sun_path, tmpnam(NULL));
        bind(connfd, (struct sockaddr *) &addr1,
                                sizeof (addr1));
        bzero(&addr2, sizeof(addr2));
        addr2.sun_family = AF_LOCAL;
        strcpy(addr2.sun_path, argv[1]);
        clilen=sizeof(addr2);
        while(1)
        {
                fgets(data, 200, stdin);
                sendto(connfd, data, 200,0,
                                (struct sockaddr*)&addr2,
                                        clilen);
                recvfrom(connfd, data, 200, 0,
                                (struct sockaddr *)&addr2,
                                         &clilen);
                printf("server sent : %s", data);
        }

        return(0);
}
```

Unix Domain Datagram Client/Server (contd.)

The code is available as "Unix datagram client.c".

Execution of Server and client

```
drago@drago-pc:~/Desktop/UNP-main/Unix Sockets$ gcc Unix\ datagram\ server.c -o datagram_server.o
drago@drago-pc:~/Desktop/UNP-main/Unix Sockets$ ./datagram_server.o temp/data_socket
```

```
drago@drago-pc:~/Desktop/UNP-main/Unix Sockets$ gcc Unix\ datagram\ client.c -o datagram_client.o
/usr/bin/ld: /tmp/ccLWQiCn.o: in function `main':
Unix datagram client.c:(.text+0xa2): warning: the use of `tmpnam' is dangerous, better use `mkstemp'
drago@drago-pc:~/Desktop/UNP-main/Unix Sockets$ ./datagram_client.o temp/data_socket
dibya
server sent : dibya
sundar
server sent : sundar
das
server sent : das
```

Passing Descriptors

In general we pass an open descriptor in two cases:

1. A child sharing all the open descriptors with the parent after a call to **fork**
2. All descriptors normally remaining open when **exec** is called.

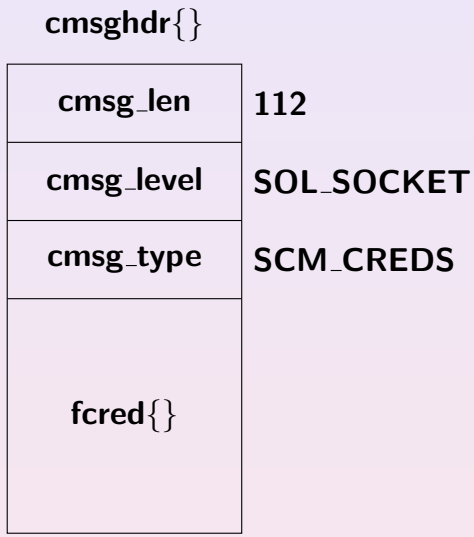
In such case the parent opens the descriptor and the child handles it. However, Unix systems provide a way to pass any open descriptor from one process to any other process. That is, there is no need for the processes to be related, such as a parent and its child.

Steps required:

- First establish a Unix domain socket between the two processes.
- Use **sendmsg** to send a special message across the Unix domain socket.

Receiving Sender Credentials

User credentials is another type of data that can be passed along a Unix domain socket as ancillary data.



Exactly how credentials are packaged up and sent as ancillary data tends to be OS-specific.

The sender must include a **cmsgcred** structure when sending data using sendmsg.

Receiving Sender Credentials (contd.)

```
struct cmsgcred {
  pid_t cmcred_pid; /* PID of sending process */
  uid_t cmcred_uid; /* real UID of sending process */
  uid_t cmcred_euid; /*effective UID of sending process*/
  gid_t cmcred_gid; /* real GID of sending process */
  short cmcred_ngroups; /* number of groups */
  gid_t cmcred_groups[CMGROUP_MAX]; /* groups */
};
```

Why we need credentials?

When a client and server communicate using these protocols, the server often needs a way to know exactly who the client is, to validate that the client has permission to ask for the service being requested.


Thank
you!