CT-I Report

SOCKETS INTERFACE

Submitted by:

Muhammad Farooq Khan
Kai Schönewald

Submitted to:

Prof Dr. Ing. Klaus David
Chair for Communication Technology
University of Kassel.
Contents

Contents.................................................................................................................................- 2 -
1. Sockets Introduction ....................................................................................................- 4 -
  1.1 UNIX File descriptor .............................................................................................- 4 -
  1.2 Types of Sockets .....................................................................................................- 6 -
2. TCP sockets.......................................................................................................................- 6 -
  2.1 Introduction ..............................................................................................................- 6 -
  2.2 TCP Socket Primitives ............................................................................................- 7 -
  2.3 TCP Client-Server Architecture .............................................................................- 8 -
3. UDP Sockets....................................................................................................................- 12 -
  3.1 Introduction ............................................................................................................- 12 -
  3.2 Important functions and data structures ............................................................... -12 -
  3.3 Client-Server example ...........................................................................................- 14 -
  3.4 Communicate with one peer ...................................................................................- 19 -
  3.5 Advantages of UDP and conclusion ......................................................................- 20 -
4. SCTP Socket ...................................................................................................................- 20 -
  4.1 What is SCTP? ........................................................................................................- 20 -
  4.2 Two different styles ...............................................................................................- 21 -
  4.3 Some functions .......................................................................................................- 21 -
  4.4 Notifications and events .......................................................................................- 24 -
  4.5 Terminate an association ......................................................................................- 25 -
  4.6 Reasons for SCTP and summary ...........................................................................- 25 -
References ...........................................................................................................................- 26 -
Abstract

Sockets are endpoints for communication. Some types of sockets provide reliable communications. Others offer few guarantees, but consume low system overhead. Socket communication can be used to let processes talk on just one machine or over the Internet. In this report TCP and UDP sockets are discussed in detail and also explained the implementation of client-server architecture of TCP and UDP in C/C++. It has also be explained how STCP sockets are better than TCP sockets, explaining the STCP sockets in detail. The purpose of this report is to discuss inter-process communication in the context of Berkeley UNIX. Special emphasis will be given to those system calls concerned with the creation, management, and use of sockets. More information on all the system calls mentioned in the report can be found in the UNIX Programmer’s Manual.
1. Sockets Introduction

Networking protocols are implemented as part of OS. The networking API exported by most of the OS is *socket interface*. Sockets interface are generated in 1983 by BSD systems. The most general mechanism for inter-process communication offered by Berkeley UNIX is the socket. A socket is an endpoint for communication. Two processes can communicate by creating sockets and sending messages between them. The two processes can be running on the same machine or can be on different machines connected to each other through some network. There are a variety of different types of sockets, differing in the way the address space of the sockets is defined and the kind of communication that is allowed between sockets. A socket type is uniquely determined by a <domain, type, protocol> triple. In order for a remote socket to be reached, it must be possible to assign a name to it. The form that this name assumes is determined by the *communication domain* or *address family* to which the socket belongs. There is also an *abstract type* or *style of communication* associated with each socket. This dictates the semantics of communication for that socket. Finally, there is a specific *protocol* that is used with the socket. A socket can be created with the *socket* system call by specifying the desired address family, socket type, and protocol.

$$\text{Socket\_descriptor} = \text{socket(domain, type, protocol)}$$

```
int socket_descriptor, domain, type, protocol;
```

This call returns a small positive integer called a *socket descriptor* that can be used as a parameter to reference the socket in subsequent system calls. Socket descriptors are similar to file descriptors returned by the *open* system call. Each *open* or *socket* call will return the smallest unused integer. Thus a given number denotes an open file, a socket, or neither (but never both). Socket and file descriptors may be used interchangeably in many system calls. For example, the *close* system call is used to destroy sockets.

1.1 UNIX File descriptor

As we know everything in UNIX is a file, it is the fact that when UNIX programs do any sort of I/O, they do it by reading or writing to a file descriptor. A file descriptor is simply an integer associated with an open file. But, that file can be a network connection, a FIFO, a
pipe, a terminal, a real on-the-disk file, or just about anything else. Following figures explain how sockets connections works like file.

**Fig 1** UNIX Descriptor Table

Fig 1 shows that different files are pointed by the UNIX file descriptor and the table holds the addresses of it. Similarly the address of network connection can also be found in the UNIX file descriptor. Fig 2 explains it in a better way.

**Fig 2** UNIX Descriptor Table for socket connection
1.2 Types of Sockets

There are mainly two types of sockets, only two, well there are more than two types of sockets but we will talk about only two types mainly. One is "Stream Sockets"; the other is "Datagram Sockets", which may hereafter be referred to as "SOCK_STREAM" and "SOCK_DGRAM", respectively. Datagram sockets are sometimes called "connectionless sockets".

Stream sockets are reliable two-way connected communication streams. If you output two items into the socket in the order "1, 2", they will arrive in the order "1, 2" at the opposite end. They will also be error free. As an example telnet uses the stream sockets. Also, web browsers use the HTTP protocol which uses stream sockets to get pages. How do stream sockets achieve this high level of data transmission quality? They use a protocol called "The Transmission Control Protocol", otherwise known as "TCP". TCP makes sure that data arrives sequentially and error-free.

What about Datagram sockets? They are connectionless, unreliable. If some datagram is sent, it may arrive. It may arrive out of order. If it arrives, the data within the packet will be error-free. Datagram sockets also use IP for routing, but they don’t use TCP; they use the "User Datagram Protocol", or "UDP". Why are they connectionless? Well, basically, it doesn’t maintain an open connection unlike in stream sockets. Just build a packet, slap an IP header on it with destination information, and send it out. No connection needed. They are generally used for packet-by-packet transfers of information. Sample applications are tftp, bootp, etc [4].

2. TCP sockets

2.1 Introduction

TCP is an acronym for Transmission Control Protocol. TCP works on Transport layer of OSI model. TCP is a connection oriented service, means; in order to send data first the connection has to be established between the sender and receiver. Through the use of TCP, the transmission between two processes is reliable, means, error free transmission of packets. TCP also takes care of flow control, means; if the sender is sending the packets at high speed
as compare to the capabilities of the speed of receiver, then it takes care of the speed in such a way that both sender and receiver’ speeds can follow it. Since TCP is connection oriented so data is sent as byte stream and it gives point to point connection. Point to point means the transmission connection has exactly two connections.

2.2 TCP Socket Primitives

The transport primitives used in Berkley UNIX for TCP are listed in the following table.

<table>
<thead>
<tr>
<th>Primitives</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOCKET</td>
<td>Creates a new communication endpoint</td>
</tr>
<tr>
<td>BIND</td>
<td>Attach a local address to socket</td>
</tr>
<tr>
<td>LISTEN</td>
<td>Willingness to accept connections</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>Block the caller until a connection attempt arrives</td>
</tr>
<tr>
<td>CONNECT</td>
<td>Actively attempt to establish connection</td>
</tr>
<tr>
<td>SEND</td>
<td>Send some data over that connection</td>
</tr>
<tr>
<td>RECEIVE</td>
<td>Revieve some data over that connection</td>
</tr>
<tr>
<td>CLOSE</td>
<td>Release the connection</td>
</tr>
</tbody>
</table>

Table 1 TCP sockets primitives [4]

The first primitives in the list are executed in the same order. The SOCKET primitive creates a new end point for the communication and allocates table space for it within the transport entity. Newly created sockets do not have addresses. These are assigned using the BIND primitive. Once the server has bound an address to a socket, remote clients can connect to it. The reason for not having the address while creating the sockets some applications are using the same addresses for years and for that matter it has been made a rule to use other addresses (port numbers) for other application. Generally the globally known applications are using first 1024 port numbers. So it is always advisable to use port numbers after 1024.
The LISTEN call allocates space to queue incoming calls for the case that several clients try to connect at the same time. In the socket model, LISTEN is not a blocking call. To block waiting for an incoming connections, the server executes an ACCEPT primitive. When a connection request arrives from the client side then a connection is established and ACCEPT primitive blocks no more then.

Then using the SEND and RECEIVE primitives, communication can be achieved between server and the client. Connection release with sockets is symmetric. When both sides have executed a CLOSE primitive, the connection is released.

2.3 TCP Client-Server Architecture

Fig 3 shows the client server architecture for TCP.

socket ()
The synopsis for the socket() system call is:

```c
#include <sys/types.h>
#include <sys/socket.h>
```
int socket(int domain, int type, int protocol);

*domain* should be set to "AF_INET", *domain* asks for the family of sockets which can be either ARPA internet services, UNIX services or XEROX networks services. Next, the *type* argument tells the kernel what kind of socket this is: SOCK_STREAM or SOCK_DGRAM. Finally, just set *protocol* to "0" to have socket() choose the correct protocol based on the *type*.

socket() simply returns to you a socket descriptor that you can use in later system calls, or -1 on error. The global variable *errno* is set to the error’s value. But what good is this socket? The answer is that it’s really no good by itself, and you need to read on and make more system calls for it to make any sense.

**bind ()**

Once we have sockets then we must assign the addresses (IP and port numbers) to it. In order to perform that action we need to use bind() system call. Bind() actually associate addresses to the sockets. Here is the synopsis for the bind() system call:

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

*sockfd* is the socket descriptor returned by the socket() system call. *my_addr* is a pointer to a struct sockaddr that contains information about your address, namely, port and IP address. *addrlen* can be set to sizeof(struct sockaddr).

**Listen()**

The listen call is fairly simple, but requires a bit of explanation. The listen() call is as follows:

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

*sockfd* is the usual socket file descriptor from the socket() system call. *backlog* is the number of connections allowed on the incoming queue. What does that mean? Well, incoming connections are going to wait in this queue until you accept() them (see below) and this is the limit on how many can queue up. Most systems silently limit this number to about 20; you can probably get away with setting it to 5 or 10.

Again, as per usual, listen() returns -1 and sets *errno* on error. Well, as you can probably imagine, we need to call bind() before we call listen() or the kernel will have us listening on a
random port. So if you’re going to be listening for incoming connections, the sequence of system calls you’ll make is:

    socket();
    bind();
    listen();

**connect ()**

This system call originates from the remote client side. The connect() call is as follows:

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

connect() gives the remote client an opportunity to have connection with the server. Catch is here that if the server is listening only then connection can be established.

*sockfd* is our friendly neighbourhood socket file descriptor, as returned by the socket() call, *serv_addr* is a struct sockaddr containing the destination port and IP address, and *addrlen* can be set to sizeof(struct sockaddr).

**accept()**

A remote client will tries to connect() to machine on a port that is listen()ing on. Their connection will be queued up waiting to be accept()ed. Machine call accept() and tell it to get the pending connection. It’ll return a *brand new socket file descriptor* to use for this single connection. Now it has *two socket file descriptors* for the price of one! The original one is still listening on your port and the newly created one is finally ready to send() and recv(). The call is as follows:

```
#include <sys/socket.h>

int accept(int sockfd, void *addr, int *addrlen);
```

*sockfd* is the listen()ing socket descriptor. Easy enough. *addr* will usually be a pointer to a local struct sockaddr_in. This is where the information about the incoming connection will go (and with it one can determine which host is calling from which port). *addrlen* is a local integer variable that should be set to sizeof(struct sockaddr_in) before its address is passed to
accept(). Accept will not put more than that many bytes into addr. accept() also returns -1 and sets \texttt{errno} if an error occurs.

**send() and recv()**

These two functions are for communicating over stream sockets or connected datagram sockets. If you want to use regular unconnected datagram sockets, you'll need to see the section on sendto() and recvfrom(), below. The send() call:

\begin{verbatim}
int send(int sockfd, const void *msg, int len, int flags);
\end{verbatim}

\texttt{sockfd} is the socket descriptor you want to send data to (whether it’s the one returned by \texttt{socket() or the one you got with accept().}) \texttt{msg} is a pointer to the data you want to send, and \texttt{len} is the length of that data in bytes. Just set \texttt{flags} to 0.

send() returns the number of bytes actually sent out—\emph{this might be less than the number you told it to send.} Sometimes it can't handle the whole chunk of data, so it sends only what it can handle and rest is required to sent it again. Remember, if the value returned by send() doesn’t match the value in \texttt{len}, only then it is required to send the rest of the string. The good news is this: if the packet is small (less than 1K or so) it will probably manage to send the whole thing all in one go. Again, -1 is returned on error, and \texttt{errno} is set to the error number.

The recv() call is similar in many respects:

\begin{verbatim}
int recv(int sockfd, void *buf, int len, unsigned int flags);
\end{verbatim}

\texttt{sockfd} is the socket descriptor to read from, \texttt{buf} is the buffer to read the information into, \texttt{len} is the maximum length of the buffer, and \texttt{flags} can again be set to 0. recv() returns the number of bytes actually read into the buffer, or -1 on error.

\textbf{Note:} All system calls and their explanation have been taken from \cite{3}. 


3. UDP Sockets

3.1 Introduction

When we want to write network applications, which use the UDP, we have to take notice of some things: The User Datagram Protocol (UDP) is different in its functionality and usage in compare to TCP. The major difference is that there is no connection establishment. Each datagram is sent over a network completely independent of all others and with no guarantee that it will reach the target. So the user or programmer always has to keep this in mind when using the UDP.

An option to implement UDP-applications is to use a Socket - API. In the following sections the Unix-Socket-Interface for UDP is described using the programming language C.

3.2 Important functions and data structures

This section describes some important functions and structures, which will be used in the next section with a client-server-example. In figure 4 we consider the Socket Address Structure, whose instances are used to set the port number, the IP-adress and so on:

```c
struct in_addr {
   in_addr_t s_addr; /* 32-bit IPv4 address */
             /* network byte ordered */
};

struct sockaddr_in {
   uint8_t  sin_len; /* length of structure (16) */
   sa_family_t sin_family; /* AF_INET */
   in_port_t  sin_port; /* 16-bit TCP or UDP port number */
       /* network byte ordered */
   struct in_addr sin_addr; /* 32-bit IPv4 address */
       /* network byte ordered */
   char        sin_zero[8]; /* unused */
};
```

Fig.4 [1]

The important members of this structure are `sin_family`, `sin_port` and `sin_addr`. With `sin_family` the network-protocol is determined. For IPv4 this member is set to `AF_INET`, while for IPv6 it is set to `AF_INET6`. In `sin_port` the port number is located. The datatype
in_port_t is normally an unsigned 16-bit integer. sin_addr contains the IP-address. It is an instance of the in_addr structure (see fig.1 top), where the datatype in_addr_t is an unsigned 32-bit integer for IPv4.

For IPv6 another address structure has to be used, called sockaddr_in6, which has nearly the same members. Important is that the IP-address is determined in an array of 16 eight-bit integers (128 bit). Because there are different address structures, sockaddr_in for IPv4 and sockaddr_in6 for IPv6, the common structure sockaddr is needed for the socket-functions, which have to deal with both. So if we want to pass an instance of an address structure as an argument to a function, we always have to do a cast, example (the function sendto() will be described next):

```c
struct sockaddr_in addr;
/* some code */
sendto( ..., ..., ..., ..., (sockaddr *) &addr, ...);
```

The two main functions, needed to write UDP-applications, are sendto() and recvfrom(). sendto() is used to send datagrams and recvfrom() is used to receive datagrams. The prototypes are shown in figure 5:

```c
ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags,
                 struct sockaddr *from, socklen_t *addrlen);

ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags,
                const struct sockaddr *to, socklen_t addrlen);
```

Fig. 5 [1]

sendto() has the following arguments: With sockfd the socket, we want to use, is determined. The second argument points to a buffer, which contains the information, we want to send. Thereby nbytes describes the size of this buffer. The flag argument is not relevant yet. to is a pointer to an address structure, where the IP-address and the port number of the target are located. In the last argument the size of the address structure is specified. The return value
Sockets Interface

shows the length of data, which are sent. The arguments of recvfrom() are lightly different: In buf the received data is stored and with from we can find out, from which source the received data comes. The function returns the length of the received data. An example, which shows how to use this functions is described in the next section.

3.3 Client-Server example

In this part a short client-server example using UDP-sockets is described. If you wish to see the complete source code, watch [1] – pages 241 ff. To get an overview about what we want to do, consider figure 6:

**Fig. 6 [1]**

On the server side there is a socket, which is bound on a well known port. The server starts by beginning to receive the first datagram. Notice that the server will block forever, if no datagram arrives. When a datagrams is received, the server sends it back to the source and receives the next datagram. The client also establishes a socket, sends a datagram and receives the response. If there is no response, the client will also block forever. Thereby the socket is not explicit bound with a port. This is done automatically.
To implement the server, some datastructures have to be initialized first (all following source-code parts are taken from [1]):

```c
int sockfd;
struct sockaddr_in servaddr, cliaddr;
```

In `sockfd` the socket filedescriptor will be saved. While `servaddr` contains the IP-address and the port number of the server, `cliaddr` will be used to save the same information from a client.

In next code part the socket is created. With `SOCK_DGRAM` in the second argument is determined that this socket will use the UDP:

```c
sockfd = Socket( AF_INET, SOCK_DGRAM, 0);
bzero( &servaddr, sizeof( servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl( INADDR_ANY);
servaddr.sin_port = htons( SERV_PORT);
```

With `bzero(...)` the server address structure is initialized to zero. Then IPv4 (`AF_INET`) is set in the member `sin_family` of the address structure. `INADDR_ANY` means that the server can receive datagrams from each available interface. So, for example, datagrams from a LAN, which is directly connected with the server, and from the internet can be received both. In the `sin_port` member the port number of the server is set. Thereby `SERV_PORT` is defined as 9877 (lock unp.h in [1]).

The functions `htonl()` and `htons()` have to be used, because the protocols, used here, work with different byte orders. Some work with big-endian, some with little-endian. The two functions convert the values in the correct orders in each case. The first of the following statements fits all together:

```c
Bind( sockfd, (SA *) &servaddr, sizeof( servaddr));
dg_echo( sockfd, (SA *) &cliaddr, sizeof( cliaddr));
```

It links the created UDP-socket with the address structure, which we already configured above. `SA` means the generic address structure, in which we always have to cast, when using socket functions (see first section). The second statement calls a function that is later described in this section.

In the client source code we are doing nearly the same, except two differences. One difference is that the IP-address of the target-server is taken from the command line:

```c
Inet_pton( AF_INET, argv[1], &servaddr.sin_addr);
```
Inet_pton() takes the value of argv[1] and puts it into sin_addr respecting IPv4 (AF_INET). The other difference is that the socket is not linked with a certain port; the bind-function is not called. The reason for this will become clear, when we watch on the dg_cli() function, which

```
#include "unp.h"

void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
    int n;
    char sendline[MAXLINE], recvline[MAXLINE + 1];

    while (fgets(sendline, MAXLINE, fp) != NULL) {
        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
        n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
        recvline[n] = 0; /* null terminate */
        fputs(recvline, stdout);
    }
```

Fig. 7 [1]

is called by the client, when all initializations are done (see figure 4). Before describing this function notice that both, the server and the client, have only configured the address structure of the server. There is nothing explicitly determined about the port number of the client for example Fig 7.

The source code of figure 7 implements the behavior of the client, which is already showed in figure 4. In detail there is a while-loop (line 7). In each iteration of that loop, first the user is asked for a message he want to send. The user presses enter and the client sends the message using Sendto() (line 8) and waits for a response by starting Recvfrom() (line 9). If a datagram is received, the message is put to stdout, usually the screen (line 11).

As mentioned above the port number of the client is not explicitly set. This is done by the kernel with the first use of sendto(). Because the fifth argument of recvfrom() is undefined (line 9), the client does not care about the sender. The client sends the messages to the correct server, because in sendto() the pservaddr structure is used, but the responses could come from everywhere. To solve this problem we can use an additional address structure preply_addr. When receiving a datagram, preply_addr is used in recvfrom(). To make a decision, if a
current datagram is from the server, where the request was sent to, we just compare `preply_addr` with `pservaddr`:

```c
/* unchanged code .... */

n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
if( memcmp(pservaddr, preply_addr, len) != 0)
    continue;
/* unchanged code ... */
```

A new problem here is that the IP-address of `pservaddr` is the current of the chosen interface, where the request is sent. If the response of the server is sent using another interface, the IP-address will also be another and the comparison does not return 0 (0 = equal, 1 = different). A way to solve this problem is to use the DNS-name, which won't be described here. Whatever, back to the server, who also needs a function to create the responses, see figure 8:

Fig. 8 [1]

The main part of this function is an infinite loop (line 8), where the server receives a datagram (line 10) and sends the message back to the client (line 11). Note that the server also don't care about the sender – what should be normal, because a server usually serves everybody. Because fork() is not used here to create a new process for each incoming request, we have an iterative server (otherwise concurrent server). But a FIFO-queue is implemented in the UDP-sockets to handle a lot of requests at one moment. In detail each socket has a receive- and a send-buffer, where datagrams are stored, if necessary. The socket options `SO_RCVBUF` and
SO_SNDBUF can be used to change the size of this buffers (for more information about socket-options, look [1], section 7.5).

The data transfer between the server and the client is not reliable, because this we are using the UDP. So if only one datagram is lost, the server or the client will block forever. When the request-datagram, from client to server is lost, the server waits for a request and the client waits for a response. Both will block forever. The client also hangs, if the response-datagram from the server is lost.

A solution is to use a timeout with the SO_RCVTIMEO socket option in the source code of the client by adding the following lines in the dg_cli function:

```c
struct timeval tv;
tv.tv_sec = 5;    /* seconds */
tv.tv_usec = 0;  /* microseconds */
Setsockopt( sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof( tv));
/* unchanged code – while-loop */
n = recvfrom( ...);
if( n < 0) {
    if( errno == EWOULDBLOCK)
        continue;
    else
        err_sys( "recvfrom error");
}
```

With `Setsockopt( ...)` is determined that a read operation with the specified socket will be stopped five seconds after the start of the read operation. Thereby a read operation means a receive command. If `recvfrom()` is stopped in this manner, `n` will be smaller than zero and we can check the `errno` variable. (The `errno` variable is set by a function when an error occurs in it to give additional information about the error. [2]). In the case the value of `errno` is `EWOULDBLOCK`, we continue. Naturally the amount of seconds, the timeout is set to, depends on a lot of details like the traffic in the network, the distance between client and server, and so on.
3.4 Communicate with one peer

In the case a client want to talk only with one peer for a long time, the default UDP-sockets are quite slow. For each datagram the kernel connects with the socket, sends the message and then unconnects the socket. We can avoid this by calling the `connect()` function explicit.

But some differences in compare to TCP have to be considered, when using this function with UDP. There is no connection establishment; no three-way-handshake or something like that. When we call `connect()` on a UDP-socket, we only determine the destination for all datagrams, which are sent over this socket. Incoming datagrams to this socket, also have to come from this destination, otherwise they will be refused.

In figure 9 we use the `connect()` function with the client of the previous section:

```c
#include "unp.h"

void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
  int n;
  char sendline[MAXLINE], recvline[MAXLINE + 1];

  Connect(sockfd, (SA *) pservaddr, servlen);

  while (Fgets(sendline, MAXLINE, fp) != NULL) {
    Write(sockfd, sendline, strlen(sendline));

    n = Read(sockfd, recvline, MAXLINE);

    recvline[n] = 0;    /* null terminate */
    Fputs(recvline, stdout);
  }
}

Fig. 9 [1]
```

Note that, because we connected the socket with a specified address structure (line 7), we don't need `sendto()` or `recvfrom()` to determine the destination, respectively the source each time. ( `write()` and `read()` are default system functions in Unix, see also [2] - pages 54, 55).

An additional advantage here is that if the destination is unreachable, the kernel returns a ICMP error to the socket in the most Unix systems. With an unconnected socket we are not able to detect this. The `recvfrom()` function would just block forever, awaiting a response.

To unconnect a connected socket, call `connect()` again, but with a address structure where `sin_family` is set to `AF_UNSPEC`. 
Note that if you want to talk with more than one peer at one time over the same UDP-socket, the connect() function doesn't make sense and you have to use sendto() and recvfrom().

3.5 Advantages of UDP and conclusion

There are problems, partially described in the previous sections, which we have because of the usage of UDP. Unreliability and no flow control for example. But UDP-sockets also have some advantages: If we want to use broadcasting or multicasting, we need UDP-socket, because there is no other way to realize it. For more information about broadcast- and multicasting, look [1] – chapter 20, 21. Another benefit is that UDP needs only two packets to send a request and receive a response, while TCP needs around ten.

The description of UDP-sockets is done now. The main functions and structures, needed to write UDP-programs, were showed. Thereby sendto() and recvfrom() take the important part. With the client-server example we described how to use this functions and considered some problems like lost datagrams. The use of the connect() function was presented, which is quite different to TCP.

4. SCTP Socket

4.1 What is SCTP?

SCTP is a new transport protocol and means Stream Control Transmission Protocol. It provides a lot of the features of TCP like reliability, flow control and it allows connections. But the term "connection" is replaced by the term "association", here. This shall indicate that with this protocol not only a connection between two IP-addresses is possible like in TCP. SCTP can manage multiple associations for each end point. Thereby an end point is called a system, which can contain more than two IP addresses – called multihoming. While in TCP there is only one stream between two endpoints, in SCTP there are multiple streams.

Summarized:
- An association is a connection between two endpoints.
- Such an association can consist of multiple streams.
- A socket can control multiple associations with multiple peers (endpoints).
- An endpoint can contain multiple IP-adiresses.
- Note that "association" and "stream" have different meanings!
Another difference in compare to TCP is that SCTP is message-oriented. This means that there is no byte-stream. Here SCTP is more comparable to UDP, where also messages are used.

The following sections describe a Unix socket API for SCTP in C. Doing this, some principles and functions, needed to write SCTP applications, will be considered. Because SCTP is a quite new protocol, it is not ensured that this API already works on all Unix systems.

4.2 Two different styles

There are two varieties of SCTP sockets available. One is called the "One-to-One Style", the other is the "One-to-Many Style".
The "One-to-One Style" provides a SCTP socket, which can have exactly one association with another socket. So the behaviour of this sort of socket is quite the same as a TCP socket. This style is suitable to port TCP programs to SCTP. Because this text wants to show the new possibilities of SCTP, this style will not be described in more detail here (for more information look [1], section 9.2).

A socket, created with the "One-to-Many Style", is able to manage multiple associations. To distinguish between the associations, an identifier with the type sctp_assoc_t is used (normal integer). The behaviour of this kind of sockets has three important facts: First, if a client closes an association, the corresponding server will also do. Second, if we use for example sendto() with an address, which not already belongs to an association, a new association will be established, it won't be blocked or something like that. And the last rule says that all sending operations on a socket uses the same address by default. The kernel choses this address when the socket is initialized.

4.3 Some functions

In this section some new functions of the SCTP API are described. To get an overview about the use of them, see figure 10, where the "One-to-Many Style" is used:
On the server side the server program creates a socket with `socket()`, links the socket at a well-known port by using `bind()` and calls `listen()` to be prepared for associations from clients. `sctp_recvmsg()` is used to wait for the first message.

The client also creates a socket and calls `sctp_sendto()` to send his request. This implies an association establishment with the server. Thereby a four-way handshake is used. The data of the request is piggybacked on the third packet during this handshake. Note that there is no `connect-function` necessary like in TCP. The client receives the answer (`sctp_recvmsg()`) and closes the association with `close()` and the server do the same, who now waits for new associations. Because the "One-to-Many Style" is used here, it also possible that multiple clients have an association with the server at the same moment using only one socket on the server side.
With the following function it is possible to bind a list of addresses with a socket; remind that in TCP and UDP we are only able to bind one or all IP-addresses of a host:

```c
int sctp_bindx( int sockfd, const struct sockaddr * addrs, int addrcnt, int flags);
```

The second argument is a pointer on that list, which has `addrcnt` elements. All address structures in the list must define the same port, because the port identifies the process in common and the process is always the same here. The `flags` argument can be set to `SCTP_BINDX_ADD_ADDR` for adding addresses or to `SCTP_BINDX_REM_ADDR`, which removes the addresses, declared in `addrs`.

Although an association can be established automatically with the first use of a send operation, there is still a function available, which can be used to connect (or better to associate) with a peer:

```c
int sctp_connectx( int sockfd, const struct sockaddr *addrs, int addrcnt);
```

With this function we are able to connect with a peer on more than one IP-address. For example if the peer has three interfaces to a network, we can define the three IP-addresses of that interfaces in `addrs` and then have three possibilities to reach the peer. All addresses in the list of the second argument have to be valid or the function will fail.

The prototype of the `sctp_sendmsg()` function follows:

```c
ssize_t sctp_sendmsg( int sockfd, const void *msg, size_t msgsz, const struct sockaddr *to, socklen_t tolen, uint32_t ppid, uint32_t flags, uint16_t stream, uint32_t timetolive, uint32_t context);
```

The second argument is a pointer on the message to be sent. `msgsz` defines the size of the message in bytes. The destination is determined in `to` and `tolen` contains the size of the address structure. `ppid` identifies a protocol and in `flags` some SCTP options can be defined. With `stream` a stream, over which the message is sent, is set. It is possible to specify the lifetime of the message in `timetolive` (set it to zero for infinite lifetime). In `context` can be placed an integer to give the message a number for example. The function returns the number of bytes, sent. To receive messages there is also new SCTP function:

```c
ssize_t sctp_recvmsg( int sockfd, void *msg, size_t msgsz, struct sockaddr *from, socklen_t *fromlen, struct sctp_sndrcvinfo *sinfo, int *msf_flags);
```

It returns the number of received bytes of the message. The message itself is stored in `msg`. The maximum size of the message is set with `msgsz`. The sender can can be identified using `from`. Thereby `fromlen` defines the size of the address of the sender. The flags of the message
are delivered in flags and in sinfo additional information about the message are available, like the used stream for example.

Notice that "old" functions, like sendmsg(), can still be used with SCTP, but the new functions provide a more user friendly interface in some cases. Sometimes it is necessary to convert one of the "One-to-Many" associations to a "One-to-One" association. To reach this, the following function exists:

```c
int sctp_peeloff( int sockfd, sctp_assoc_t id);
```

The function returns a new socket descriptor, which is already bound with the converted "One-to-One" association. The association, to be take out of the "One-to-Many" associations of sockfd, is set with id. With the function it is possible to build a concurrent server. For each new association establishment the server calls sctp_peeloff() and works with the new socket descriptor on a new process or thread. To get all addresses of a peer linked with a current association, the following function can be used:

```c
int sctp_getpaddrs( int sockfd, sctp_assoc_t id, struct sockaddr **addrs);
```

sockfd defines the local socket and id defines the association to the remote peer. In addrs a list of the addresses is placed. If we wish to know the local address of an association, we can use sctp_getladdrs(), which has the same arguments as sctp_getpaddrs(). Both functions return the number of elements, which they put in the list. Note that you have to allocate memory for the list, first.

### 4.4 Notifications and events

In SCTP there are two types of messages. One type represents the normal data messages and the other contains notifications about events. To distinguish between them the msg_flags field, which for example can be get with sctp_recvmsg(), is set to MSG_NOTIFICATION, if there is a notification message. Thereby an event means the change of the status of an association or an occurred error for example. When such an event occurs, it generates a certain notification-message, which will be interleaved with the messages from the peer(s). But all events are turned off by default, except the sctp_data_io_event, which gives additional information for send- and receive functions.

There are a lot of events and appendant notifications, which are not described here. (Look [1], section 9.14 for more information.)
4.5 Terminate an association

If we wish to terminate an association, the `close()` function just has to be applied on the concerning socket. But this terminates all associations linked with the socket and the socket can not be used for new associations. The `shutdown()` function closes all associations of the socket, which can still be used for new associations. A new socket has not to be created. For a client this should be enough, but a server may wish to terminate a certain association. To reach this, the server can set the `MSG_EOF` flag in a `Sctp_sendmsg()` function:

```c
Sctp_sendmsg( ..., ..., ..., ..., ..., ..., (sri.sinfo_flags | MSG_EOF), ..., ..., ...);
```

Then after the message is acknowledged, the association, used for the above send()-operation, will be closed. Note that in all kinds of closing an association, there is no half-closed-state like in TCP. An association always will be closed in both directions.

4.6 Reasons for SCTP and summary

SCTP is a new transport protocol, which especially wants to be an alternative for TCP. So this section shows some of the advantages and drawbacks of SCTP in compare to TCP.

One benefit is that SCTP provides multihoming. Hence an end point becomes more reliable, because it can be reached from multiple interfaces in some cases. A TCP connection consists only of one stream. If there is a error in that stream, it is quite expensive to repair the error. But in SCTP there are multiple streams in an association and each stream is completely error-independent from the others. So it is possible to write applications, which are more robust by using multiple streams. Another benefit is that in some cases, it is more easier to implement SCTP-programs. For example a server just has to manage one socket, which can handle all associations to the clients.

A drawback of SCTP is that there is no half-closed-state, although only a few programs really use this possibility in TCP. SCTP also does not provide urgent data, what could be missed by some programmers. But the biggest drawback is that SCTP is not completely ready. It is still possible that the API changes in some details.

The description of SCTP sockets is done now. At first it was showed what SCTP is. Then the "one-to-one"- and the "one-to-many" styles were presented. In following section some new functions were introduced, which make it more easier to deal with SCTP-sockets. The
principle of events and notifications was described, which make it possible to get additional information about the state of associations. In the example a SCTP-server with an appendent client was implemented. Remarkable is that the server only needs one socket. At the end the text shows how to control the number of streams and considered some ways to terminate associations.

References


  http://www.ecst.csuchico.edu/~beej/guide/net/