

IPC Socket (Unix Domain Socket)

by

Viraj Savaliya

ASU Id: [REDACTED]

Email Id: vsavaliy@asu.edu

Final Report for CSE 530: Embedded Operating Systems Internals  
Master of Science in Computer Engineering (Electrical Engineering)

Course Instructor:

Dr. Yann-Hang Lee

ARIZONA STATE UNIVERSITY

May 2020

## ACKNOWLEDGMENTS

I would like to thank Dr. Yann-Hang Lee for providing me the opportunity to write a report on the given topic. Also, would like to thank him again for being an amazing Professor for the course CSE530 which made me understand the Linux Kernel and its internals better and making the subject more interesting through the assignments provided throughout the course. I would also like to thank Xiangyu Guo who was the Teaching Assistant for the course as he helped me a lot throughout for all the assignment related as well as programming and Linux kernel related questions and dedicated enough time in order to make sure my query is resolved.

## TABLE OF CONTENTS

	Page
PREFACE.....	iii
CHAPTER	
1. INTRODUCTION TO SOCKETS.....	1
Advantages of using Sockets.....	2
2. LET’S TALK UNIX!.....	4
Types of Unix Domain Sockets.....	5
3. UNIX SOCKET API.....	8
Synopsis for Functions.....	9
4. Internal working of Socket Creation.....	14
5. METHODOLOGIES.....	16
6. INVESTIGATION FOR SOCKETS.....	25
7. SAMPLE CODE.....	27
REFERENCES.....	29

## PREFACE

This report is about the Inter Process Communication sockets of Unix Domain. It is written for the Final Exam/ Report Submission for CSE530: EOSI course offered in Spring 2020 at Arizona State University. The report structure and the content are according to my understanding of the subject matter. Various online resources, textbooks as well as lecture notes for the course were referred for understanding the subject matter and to form the detailed report for the topic. All the references which were used are mentioned at the end of the document and any content which is exactly taken from the source is cited as well. The attempt made while writing is with the aim to form a resource which explains the subject matter in as simple way as possible. The Linux kernel version v5.6.11 is used for reference for the report and its source code can be found online <sup>[1]</sup>.

## CHAPTER 1: INTRODUCTION TO SOCKETS

Communication is as important in machines as it is important for humans. We communicate using various medium like phone calls, text messages, messaging via apps, emails, etc. Similarly, there are various mediums for communication between processes running on machine. Sockets are one such medium which provide the medium for communication. They are very similar to making a call and exchanging information over the phone call when we look at the way they are implemented. In a phone call we have 2 endpoints, namely the caller's phone and the receiver's phone. The phone call procedure is like the caller must pick up his handset and dial the number to make a call and wait for the other person to pick up. The other person gets ring on his phone and he needs to accept the phone call to exchange information. Then when they are done talking, they need to end the call. Similarly, in sockets there are two endpoints needed to establish a connection for communication. One is the server endpoint, and another is the client endpoint. Both the endpoints are distinguished clearly in sockets. There can be multiple clients for the server. Another analogy which is more similar to socket mechanism can be like the servers are like doctor's clinic and the clients are like patients who come to visit the clinic. The patients knock on the door to talk with the doctor, the doctor will then allow them to come in and talk. The doctor can have multiple patients coming and it maintains two queues, one of the patients he has treated and the one yet knocking. Also, the doctor makes sure the clients are treated privately and they are talked within closed doors so that no other person can hear about the other persons issue. The sockets working is very similar to these analogies and in addition sockets can also allow communication across different machines or on the same machine and the communication is bi-directional.

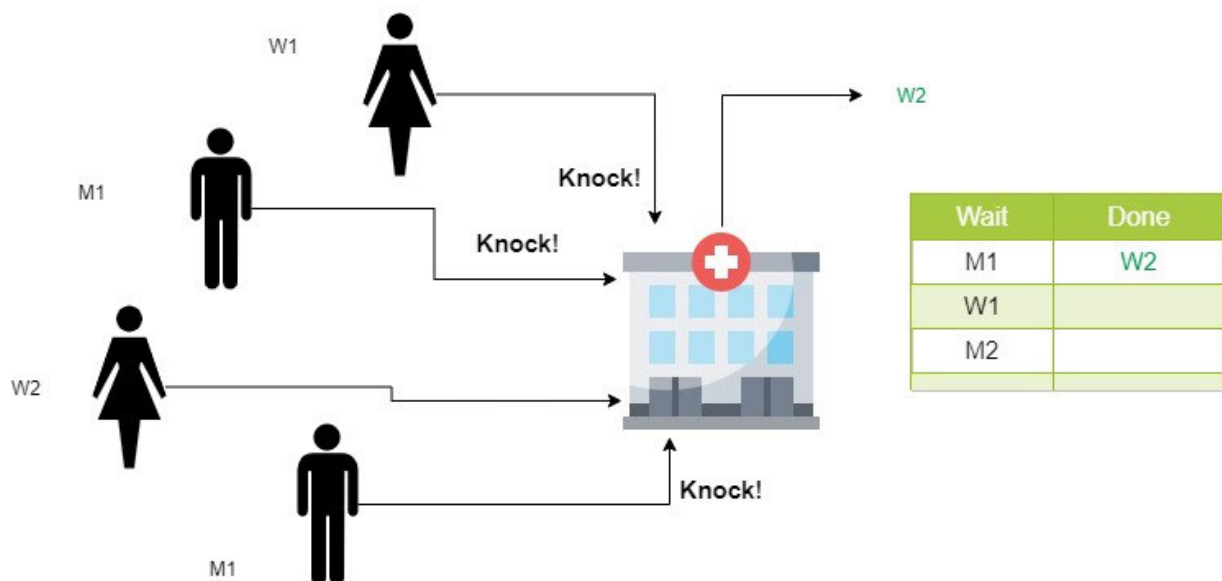


Figure 1. Doctor Clinic- Patient analogy for sockets.

The sockets were originally developed by the University of California, Berkeley (UCB) in 1969 when they developed their own flavor of Unix known as BSD operating system. The sockets come in two types; one is the IPC socket for processes on same physical device and the other is the network sockets for processes running on different hosts. Network sockets rely on the underlying protocols like TCP/IP and UDP whereas IPC sockets rely on the host kernel to support for communication. The basic implementation though for both remains the same. On the server side, the socket created by *socket()* system call and then it is given an address using *bind()* system call. Now the server can be identified by the clients as it had an address. The server listen to any connections initiated by any of its clients using *listen()* system call. On the client side, the client creates a socket using the *socket()* system call and then tries to connect to the server's address using the *connect()* system call. The server accepts the connection using the *accept()* system call if it is available and ready to serve. Now the connection is established and they can exchange data using number of ways but the most simplest one is to use *read()* and *write()* system calls. After the communication is done or the request is served the socket is closed using the *close()* system call. The client and the server can again establish connection again when required using the same steps. Also, for each connection a different channel is created in sockets. The server can be either an iterative server which can handle only one client at a time, or it can be a concurrent server which can serve multiple clients at a time. The server can be selected based on the application.

Socket is just an abstraction for communication endpoint. Like files, sockets use socket descriptors to access sockets. As "Everything is a File" in Unix OS, sockets are just special forms of files. Many of the functions of the sockets are similar to as that of the files. The main difference between the two is that when *open()* system call is called a file or a file descriptor is bound to the device whereas the socket can choose every time the destination it wants to bind to. Also, when allocating unit (file entry) numbers, the kernel makes no distinction between files and sockets. This provides the flexibility for operating the function of files for sockets as well.

### **Advantages of using sockets:**

1. Communication can be bi-directional. You can send and receive messages from both endpoints.
2. Messages can be distributed to multiple processes at the same time using multicast and many clients can be served at a time.
3. The clients can also be distinguished by the server.
4. Sockets can be good for debugging purposes. Like when you want to receive feedback from some client when you are testing, and the client is still in progress to be fully developed.
5. Sockets can be used for communicating on different machines.
6. In sockets you can have different types of messages and of varying sizes. But here you should implement your socket on various aspects like synchronization requirement, blocking/ non-blocking messages, atomicity, etc. which will depend on your application.

7. The message exchanging speed is fast using sockets.
8. Method of creating and opening sockets is easier and favorable for sockets.
9. Sockets can have packetized communication.

## Chapter 2: LET'S TALK UNIX!

Unix domain sockets (UDS) are Inter-Process Communication (IPC) socket for communicating data between the processes on the same host machine. Their application lies mainly for processes for which you want the data to be shared only locally or do not want any process running on some other machine to have access to the data. They are also known as local domain sockets. Unix IPC sockets are represented by the *AF\_UNIX* socket family or also by *AF\_LOCAL* (POSIX name for *AF\_UNIX*). They are simpler like pipes and have less overhead compared to network (TCP/IP or UDP) sockets. But, the implementation of Unix sockets remains similar to that of network sockets. In UDS, kernel is responsible for the transport of data rather than some protocol. UDS are suitable for applications that use the Client-Server Architecture for communication. They communicate between processes running on the same machine very efficiently and are fast as well. Unix sockets have a filesystem pathname rather than an IP address or port for namespace (address). The client and the server agree upon the pathname to find each other. They also support passing file descriptors or process credentials to other processes on the OS. Unix domain sockets can have restrictions like ownership and permission for certain processes to open the file and communicate with the server. These sockets are very much secure, and no routing is required as they are present only on the host machine. Both connectionless and connection-oriented communication is supported by UDS. The three valid types of UDS are *SOCK\_STREAM* for stream oriented (similar to TCP) sockets, *SOCK\_DGRAM* for datagram - oriented (similar to UDP) socket and *SOCK\_SEQPACKET* for sequenced-packet (similar to SCTP) socket. The address of the UDS is represented in the form of following structure:

```
5. File: /include/upai/linux/un.h
6.
7. #define UNIX_PATH_MAX 108
8.
9. struct sockaddr_un {
10.     sa_family_t sun_family; /*AF_UNIX*/
11.     char sun_path[UNIX_PATH_MAX]; /*pathname*/
12. };
```

Where *sa\_family\_t* specifies which family the socket belongs to and for UDS it should be *AF\_UNIX*.

The second field, *sun\_path* is the null terminated pathname for your socket. The maximum bytes for the length of the name allowed in UDS is 108. The pathname including the null terminating byte should not exceed 108. You can also specify the length of the address using *addrlen* argument which should be same as *sizeof(struct sockaddr\_un)* as this will be the socket address used by processes to establish a connection. Also, this socket address is only required to establish



a connection and is not required after it is established, i.e. not required for transferring data. Interestingly, there can also be unnamed and abstract sockets. Unnamed sockets are the ones which are not bound to any pathname, these sockets are created by *socketpair()*. Unnamed sockets are mostly useful when you are creating 2 processes or simply when the two processes you want to communicate are not unrelated to each other (E.g parent and child process are related processes.). Abstract sockets are the ones which have their *sunpath[0]* as '\0' i.e it starts with a null byte. In such a case the socket address is the remainder of the *sunpath[]* and its length can be specified by the length of the structure usign *addrlen*. In abstract sockets the pathname is not required to be null terminated. Abstract sockets have a unique feature of binding to its name rather than binding to an entry name created in file system. The advantage for abstract sockets would be that there is no need to worry about the possible collision of names in the file system, no need to unlink the socket pathname when you are done using the socket. The application for abstract sockets lies when you don't want a chroot (isolated) environment, i.e. when you want to switch between files and access different directories for the running process. The permission and ownership changes of an object have no effect on abstract sockets.

### **Types of Unix domain sockets:**

#### **1. SOCK\_STREAM**

This is the standard socket which preserves the byte sequence of data without preserving the message boundaries. These sockets are quite reliable as they provide a proper sequence of the message transfer which is error-free and in order. It allows the receiver endpoint to read to data in byte by byte or sender to send the information in multiple small messages. It provides out-of-band capabilities, which means creating a separate channel to transfer the data and protecting it from getting it interfered by some other messages. The messages are kept intact and are unduplicated. It has well defined mechanisms to create and destroy connections and report error. The analogy of these sockets is like of the phone call as they are connection-oriented and it follows the standard routine to establish the connection of listening, connecting and accepting the requests for connection establishment. Once the connection is made, the peers can communicate and transfer messages till the time it is destroyed. Even If one end of the socket is closed then it won't allow to transfer the message and will report an error. Multiple streams can be created for each connection to transfer the data separately. These sockets are generally used when the messages are in unstructured stream or have arbitrary long message payloads (no upper limit on message size).

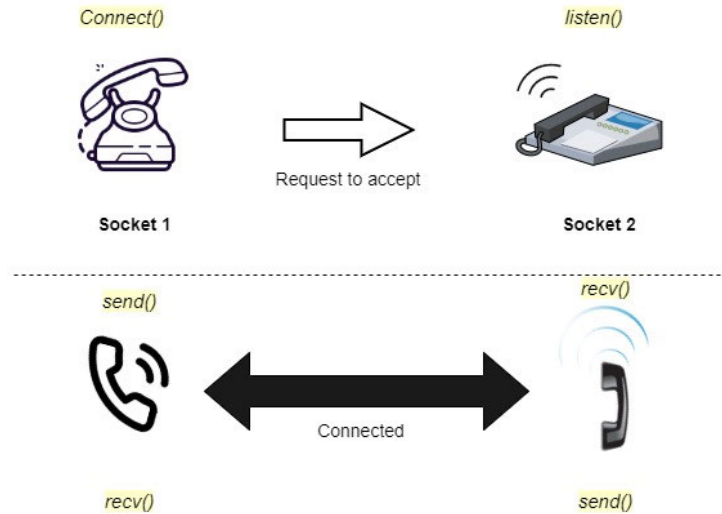


Figure 2. Phone call connection analogy for UNIX Stream Socket

## 2. SOCK\_DGRAM

These sockets are message-oriented, and they preserve the message boundaries. Which means that if one message is written by the sender then the receiver will have to do one read to get that message. The datagram sockets are usually said to be unreliable but in Unix domain they are reliable as the datagrams do not get reordered or duplicated in Unix. All the messages are either delivered or reported as missing to the sender. These sockets are connectionless sockets and the analogy for these sockets can be like the postal message system. Every message sent is like a datagram and has its individual reference to identify (address) it. This is why the number of messages received by the receiver should be the same the number of messages sent by the sender. These sockets don't follow the standard routine for connection establishment and hence there is no listening or accepting the request. Hence, as no establishment of connection is done before sending the messages, they are called connectionless sockets. Every message should have the address of the receiver endpoint in order to be delivered just like the mails in the postal system. In order to send the message in both directions, the client end should also be bound to an address. These sockets are used when you have an upper bound on the message size and small messages need to be transferred. In Unix domain *SOCK\_RAW* is considered to follow and created as *SOCK\_DGRAM*.

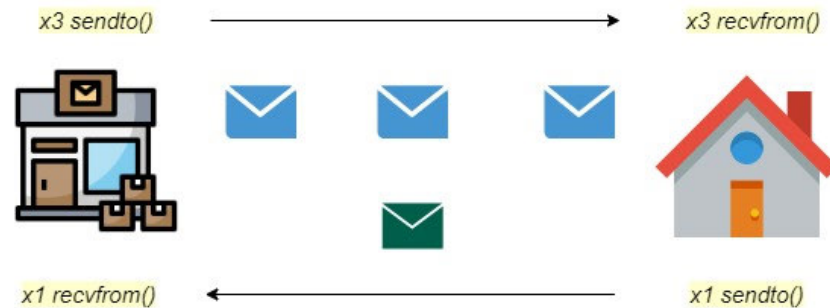


Figure 3. Postal message system analogy for UNIX Datagram Sockets

### 3. SOCK\_SEQPACKET

These sockets are kind of a combination of both the other socket types. These sockets have the combined benefits of both datagram and stream sockets. Sequenced-packet sockets are connection-oriented like stream but are also message oriented like datagram. These sockets are in general reliable and the messages are delivered in order. The message boundaries are preserved which means the messages here have an upper limit on its size. The standard sequence of listening, connecting and accepting is followed here for connection establishment. Like stream sockets the order of the message is preserved, delivery is guaranteed, and no duplication of messages takes place. Like datagram sockets, the number of send by the sender and the number of receive by the receiver has to be same, record boundaries are maintained and an error report can be sent to the sender and also multiple messages are sent of small size. These sockets are not the standard sockets and not generally used but can be used when the benefits of both the sockets are required in the application.

## CHAPTER 3: UNIX SOCKET API

The application interface for Stream and Datagram sockets which you can use from the user space in Unix domain is as follows: -

### 1. Stream socket:

#### a. Server endpoint:

`socket()`, `bind()`, `listen()`, `accept()`, `getpeername()`, `recv()`, `send()`, `close()`

#### b. Client endpoint:

`socket()`, `bind()`, `connect()`, `send()`, `recv()`, `close()`

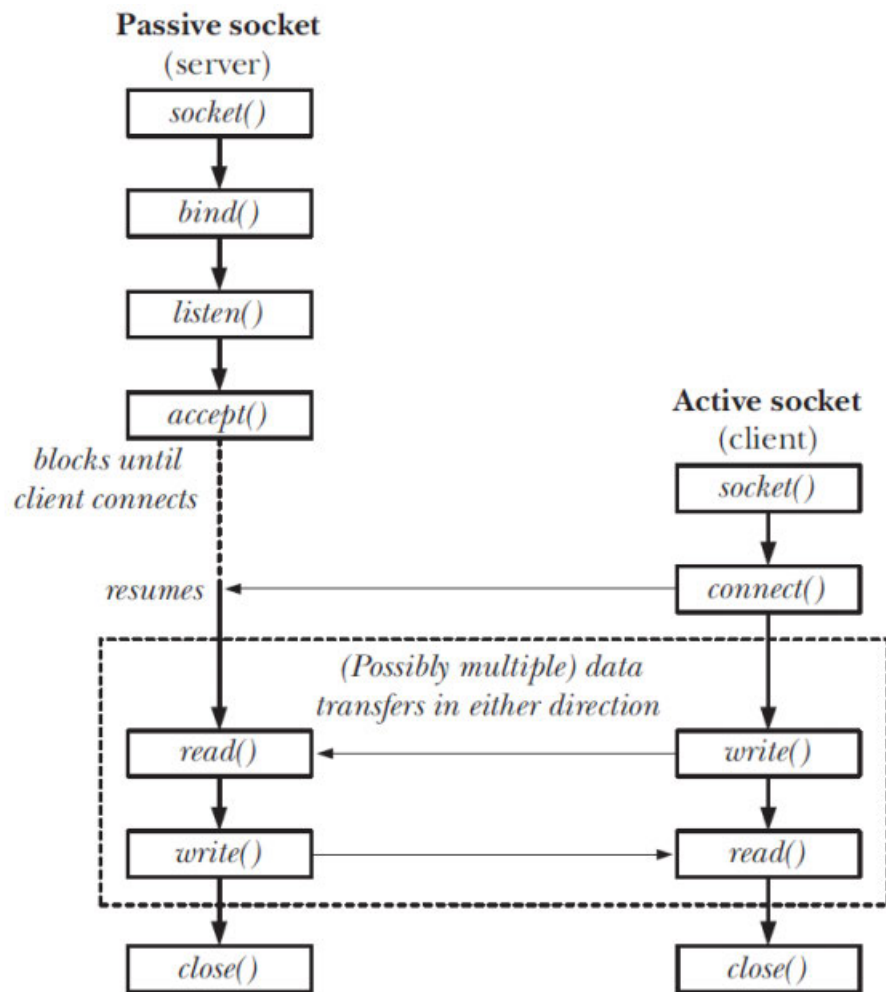


Figure 4. UNIX Stream socket connection establishment sequence<sup>[9]</sup>.

## 2. Datagram socket:

### a. Server endpoint:

socket(), bind(), recvfrom(), , close()

### b. Client endpoint:

socket(), sendto(), close()

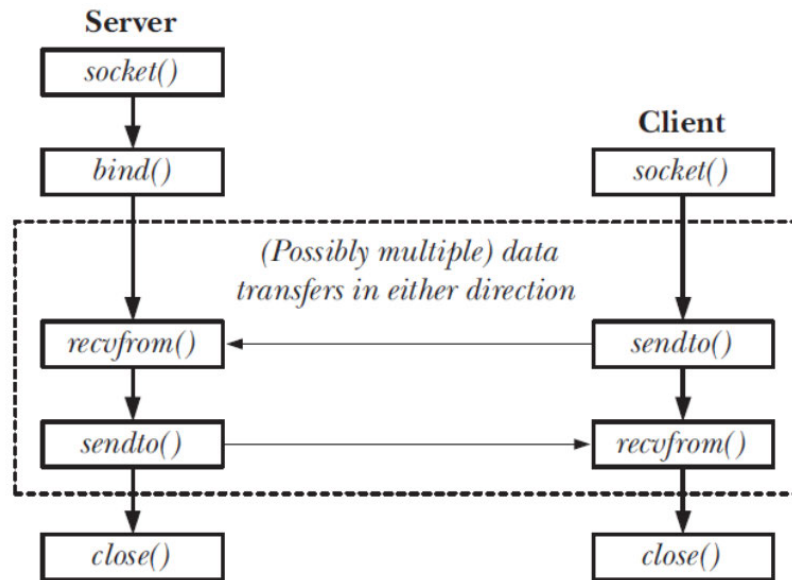


Figure 5. UNIX Datagram socket connection establishment sequence<sup>[9]</sup>.

## 3. Other Similar Function calls =

read(), write(), shutdown(), recvfrom(), select(), poll()

## SYNOPSIS FOR FUNCTIONS:

### 1. `socket()` –

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

It creates unbound socket which is an endpoint for communication and returns a file descriptor (non-negative integer) which can be used in function calls that operate on socket. On failure it returns -1.

Address domain should be `AF_UNIX` or `AF_LOCAL` for Unix sockets.

Type field specifies the type of the socket created. For e.g. *SOCK\_STREAM*, *SOCK\_DGRAM* or *SOCK\_SEQPACKET*.

Protocol field to specify if any protocols used or requested. Mostly the field is 0 as it specifies to use default protocol based on the domain and type selected.

## 2. **bind()** –

```
int bind(int socket, const struct sockaddr *address, socklen_t address_len);
```

It used to bind a name to the socket. It assigns local socket address to the socket. It returns 0 on success otherwise -1.

First field is the file descriptor of the socket to be bound.

Second, pointer that points to the *sockaddr\_un* structure containing the address to be bound to the socket.

Third, specifies the length of the *sockaddr\_un* structure.

## 3. **listen()** –

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

It basically tells that socket is ready to accept connections. It returns 0 on success otherwise -1.

*socket* argument is the file descriptor that refers to a socket.

*backlog* argument specifies the maximum possible length of the wait queue for connections.

When the queue is full, no more connections will be queued to get accepted and the clients will receive an error or might be asked to resend it again later (try again later).

## 4. **accept()** –

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

It is used with connection-oriented sockets (i.e. *SOCK\_STREAM*, *SOCK\_SEQPACKET*). It takes in the first request on the queue of listening and creates a new socket with the same type protocol and family as of the specified socket and allocates a file descriptor for that socket. It returns the non-negative file descriptor of the accepted socket on success or else it returns -1.

*sockfd* is the socket created with *socket()*, bound to an address using *bind()* and issued a successful call using *listen()*. The second field is the pointer of the *sockaddr* structure where the address of the connecting socket should be returned. The third field is the length of the supplied *sockaddr* and on output specifies the length of the stored address.

## 5. **connect()** –

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

It makes an attempt to establish a connection on the socket. It connects the socket referred by the *sockfd* to the address specified by *addr*. The third field specifies the size of the *addr*. It returns 0 on success or else it returns -1.

## 6. **getpeername()** –

```
int getpeername(int socket, struct sockaddr *address, socklen_t *address_len);
```

This system call is called after you have accepted, or you are connected to some peer process. It retrieves the address of the peer sockets specified and stores the address in the *sockaddr* struct buffer. It also stores the length of the address. If the buffer size is too small then the returned address of the peer is truncated. It returns 0 on success otherwise -1. A very similar alternative is to use *getsockname()*.

## 7. **recv()** , **recvfrom()** –

```
ssize_t recv(int socket, void *buffer, size_t length, int flags);
```

These are used to receive messages from a socket. Returns the length of the message in bytes, if no message or peer is shutdown then return 0 otherwise -1.

*recv* is generally used for connection-oriented sockets as it does not take the source's address.

*socket* argument is the socket's file descriptor.

Second field is the buffer to store the message.

Third field is the length in bytes of the message stored in buffer.

Fourth field is a flag for the message received. This indicates the type of message reception, whether to send the entire data back to the user or send only partially back.

If message is too long to fit in the buffer, then the excess bytes are discarded.

```
int recvfrom(int socket, char *buffer,int length, int flags,struct sockaddr *address,int *address_length);
```

This is similar to *recv* but is used for datagram sockets.

In addition to fields of *recv*, it has an address pointer of the *sockaddr* struct from which the data is to be taken and also has a pointer to integer that contains the address length.

Another alternative is to use *recvmsg()*.

## 8. *send()* , *sendto()* –

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

This system call is similar to write. It is used to send message to another socket. It can be used only when the connection is established. It returns the bytes of the message sent or else on error it returns -1.

*sockfd* is the file descriptor of the socket who is sending the message.

*buf* is the buffer for keeping the message and its length is specified in *len*.

*flags* field to indicate various status regarding message like message sent properly, send to hosts only, blocking or non-blocking operation, more data to be sent, etc.

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr  
*dest_addr, socklen_t addrlen);
```

This is similar to send but it can be used anytime, even when connection is not established.

In addition to fields of send, it has an address pointer of the *sockaddr* of the destination socket and its size is given by *addrlen*.

Another alternative to use is *sendmsg()*.

## 9. *close()* –

```
int close(int fd);
```

This system call will close the socket and releases the file descriptor so that it can be reused again. I return 0 on SUCCESS or else -1 on error.

## 10. *read()* , *write()*–

```
ssize_t read(int fd, void *buf, size_t count);
```

```
ssize_t write(int fd, const void *buf, size_t count);
```

These works the same as they work for files. *read* is for reading from a file and *write* for writing to a file.

## 11. *shutdown()* –

```
int shutdown(int sockfd, int how);
```

It causes the connection to terminate by shutting down all or some section of the connection associated with the *sockfd* to *shutdown*. On success it returns 0 or else -1 on error.



## **12. *select()*, *poll()* –**

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

These system calls are used in addition to the socket API calls when you want to monitor multiple files to wait for some files to become ready in order to perform I/O operations.

## CHAPTER 4: Internal working of Socket Creation

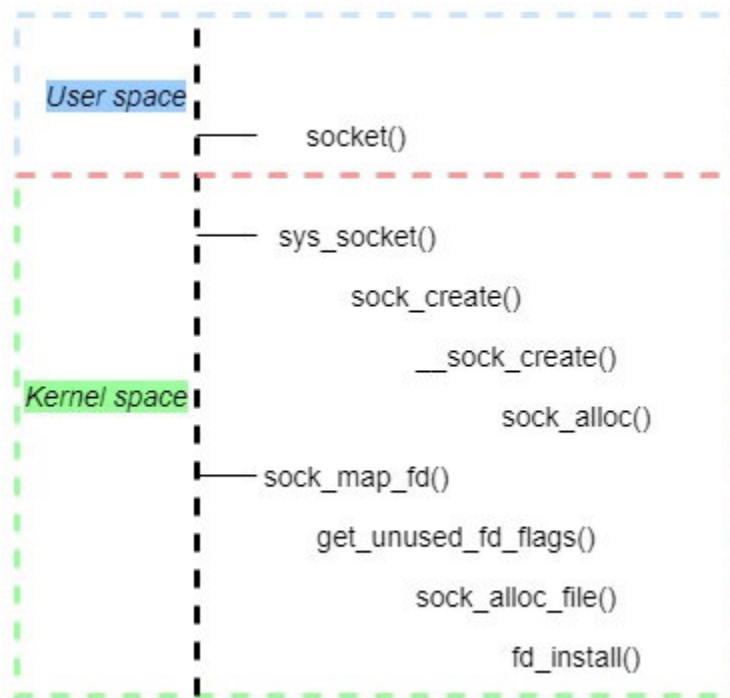


Figure 6. The internal flow of calling functions for socket creation

The first step of creating socket requires an analysis on how it is formed and what all things happen or is changed inside the kernel when a socket is created. The source code file to look about the formation of socket is in `/net/socket.c`<sup>[4]</sup>. Once the socket is created then you can use all the operations provided for it, create connection, transfer messages, etc. Basically, you can use all the functions mentioned for the particular socket domain inside the kernel. For `AF_UNIX` sockets the functions, operations, structures, etc. defined in `/net/unix/af_unix.c` all become available to be used for the socket and can be used as per their definitions explained in the methodologies section of this report.

For creating a socket first the `socket()` system call is invoked from the user space. The socket call needs to be called with the domain, type and protocol arguments. This system call transfers the control to the kernel level. The first thing to be called inside the kernel space is `__sys_socket` which has the same arguments and is the system call definition of the `socket()` in the kernel space. Now, before we dig deeper into it, there are two important structures associated with the socket. The first is struct `socket` and another is struct `sock`. The former is the endpoint definition of the socket for the communication and latter is the structure pointer present inside struct `socket` which represents the information about the operations related to the socket specific to the domain. Inside the `__sys_socket` definition of the system call, first the `sock` pointer and the flags variables are declared (allocated in the stack). Then, the various flags

are checked, and constants are checked for its consistency. Then *sock\_create()* is called to initialize the socket and sock structures. The *sock\_create* takes the family, type, protocol and the socket pointer as its arguments and calls *\_\_sock\_create* to create the socket. Now, first the protocol range is checked, security hooks are triggered and memory is allocated for the socket to be created. Also, the protocol operations are initiated and the necessary information related to the socket are filled up. The *sock\_alloc()* function for allocation of socket allocates a new inode (if available or else return NULL) and the socket object and bounds the two of them together and initializes them. The *sockfs* filesystem binds the inode and the socket object together and is now responsible to keep track of the socket information in the system. The inode for the sockets is a special inode and is different from all other inodes of dev file, block file, etc. The file system will create a tree and this new inode will be the superblock for all the sockets. The counter inside the cpu is incremented to add the socket to the list sockets. We can know the socket count inside the cpu using *sockstat*.

After the socket is created the system call *\_\_sys\_socket* is responsible to return a file descriptor to the user space of the socket created. The function *sock\_map\_fd()* is called to return a file descriptor for the socket. To check for the resource limits *get\_unused\_fd\_flags()* is called and then *sock\_alloc\_file()* is called to allocate a file descriptor for the socket. If there is no error till now then the *fd\_install()* is called to assign the file descriptor for the new file. If there occurs an error, then the *fd* value is made unused and error is returned.

Now the socket is created and all the things inside the kernel is set and you can use socket API as per the need of your application and starts performing transfer of messages. The kernel will create a separate network namespace for each socket. Network namespace is basically a brand-new network stack for each socket and controls how much system resource can be used. Namespaces also isolates resources for all the stacks at the network-level.

## CHAPTER 5: METHODOLOGIES

The following is the source code review for *AF\_UNIX* domain of socket explaining every function, structure, macros and operations in brief:

### 1. *unix\_socket\_table*

This is a linked list to keep track of all the unix socket which are bounded with some address. Its default size is 2\*256 and its size is increased every time a new socket is bounded by the value hash.

### 2. *UNIX\_ABSTRACT*

It is a macro used for abstract unix sockets and it checks for hash size of the unix sockets to be less than 256 or not.

### 3. *CONFIG\_SECURITY\_NETWORK*

When this is enabled, network and security hooks are enabled and allows the security module to use these hooks to implement access controls for socket and network.

For unix sockets, when it is enabled function for getting and setting the security id for the socket are enabled.

### 4. *unix\_peer*

This is a macro used for declaring a socket to be a peer of another socket.

### 5. *unix\_peer\_get*

This is a function used to return get the peer structure pointer of a peer of the socket.

### 6. *unix\_release\_addr*

This function is used to test if the socket address is of zero length or not and releases the address if it is of zero length.

### 7. *unix\_mkname*

This function is to make sure the *sun\_path* is always null terminated.

### 8. *unix\_remove\_socket, unix\_insert\_socket*

These functions are to safely remove and insert the socket from the linked list.

### 9. *unix\_find\_socket\_byname, unix\_find\_socket\_byinode*

These functions are to find the socket in the linked list by iterating over the list using its name and inode value respectively.

**10. *unix\_dgram\_peer\_wake\_relay, unix\_dgram\_peer\_wake\_connect, unix\_dgram\_peer\_wake\_disconnect, unix\_dgram\_peer\_wake\_disconnect\_wakeup, unix\_dgram\_peer\_wake\_me***

These functions are for datagram sockets which are asymmetrically connected. When peer tries to perform some action and the socket cannot take in more messages then a wake up needs to be performed on the waitqueue to test the datagram being propagated. The above functions are to perform wake with relay to check if the waitqueue exists, wake and connect to the waitqueue, wake and disconnect from the waitqueue , to wake and disconnect with relay to check if waitqueue exists and to wake and check if the socket is not dead.

**11. *Unix\_writeable, unix\_write\_space***

The first function is used to check if the socket is writable or not. The second function is used to initialise a write space for the socket.

**12. *unix\_dgram\_disconnected***

This function is used to clear receive queue of a datagram socket when it is disconnected or changes its peers.

**13. *unix\_sock\_destructor***

This the destructor function which is called when socket is dead.

**14. *unix\_release\_sock***

This function is called when the sockets is intended to be released. It clears the state of the socket, shuts down, closes it, disables any write from peers and disconnects them and flushes out all the queues.

**15. *init\_peercred, copy\_peercred***

These functions are to initialize and set all the credentials of the peer socket respectively.

**16. *unix\_listen***

This function is to wait for incoming connections of clients on the server side and store them in a queue before accepting it. It is mostly for providing the feature to the socket to let it handle the incoming connections in some way before accepting them. You can also limit the number of connection requests in the queue. You can also think about this as caching.

**17. *unix\_release***

This function is defined as an operation to the socket when it wants to release the socket. It looks for the pointer to the socket struct and passes on to the *unix\_release\_sock* function to release the socket.

### **18. *unix\_bind, unix\_autobind***

The *bind* function is used to bind the socket to an address. It is like giving a phone number to someone so that they can be called using that number later. The *bind* function checks for all the details like sockets family is *AF\_UNIX* or not, checks if the pathname follows all the rules, if the name or inode already exists and then allocates address and its name to the socket table. If the length of the address is short, then it calls *unix\_autobind* which makes sure to provide an address to the socket.

### **19. *unix\_stream\_connect, unix\_dgram\_connect***

These functions are used to establish connection of one process with another process for which socket is already created. Usually it is used by the client to connect with the server. *unix\_stream\_connect* is used for stream as well as seqpacket type of socket whereas *unix\_dgram\_connect* is used by the datagram type of socket.

The *unix\_stream\_connect* function checks if the pathname of the socket is within the rules, allocates the resources required for making a connection, creates a new socket, allocates socket buffer to send to the listening socket, finds the listening socket and locks its state, rechecks the state and connection, checks for any security concerns, if everything fine till now then it will set all the necessary fields and credentials and sends the information about it being ready to the listening socket.

The *unix\_dgram\_connect* function also works in similar fashion. It first checks for the pathname being correct or not, address is bound to the socket or not, it then checks for the listening socket to confirm if it is not dead, checks for security concerns, rechecks for the connection, wakes up the socket if it is not awake and makes itself ready and declares about being ready to the listening socket.

### **20. *unix\_socketpair***

This function creates a special pair of unnamed sockets. The specialty of this socket is that you are not required to go through the entire process of creating socket, binding address to it, listening, accepting, etc. This function is nice enough to return you a pair of already connected sockets.

It first holds two sockets, declares each as a peer of other and initializes the peer credentials for both. Later, it checks if the required socket is not of datagram type and if not then sets the state for both and declares both as connected.

### **21. *unix\_accept***

This function is used for accepting any requested connection coming after listening. It creates a brand-new socket file descriptor for the sending and receiving data with the socket who requested connection. While, doing so it allows you to keep listening with the old socket file descriptor.

This function is for datagram sockets only and return error for other types of socket. It takes the socket buffer of the other end of datagram socket and checks it is not yet shutdown. It wakes up the peer socket if required and changes and declares its state as connected.

## **22. *unix\_getname***

This function is used to get the name (address) of the peer.

## **23. *unix\_poll*, *unix\_dgram\_poll***

The former is used for stream sockets and the latter is used for datagram sockets. Both the functions are identical in operation. They poll (wait) for the necessary conditions or events to become true so that they can continue their operation successfully. They check for exceptional events like shutdown, socket closed, error in socket, if the socket is still readable or not and also makes sockets writable in order to avoid being stuck in case of one end being shut down.

## **24. *unix\_ioctl*, *unix\_compat\_ioctl***

ioctl in unix socket controls or queries the characteristics of socket descriptor's data. If CONFIG\_COMPACT enabled (kernel allowed to handle system calls from ELF binaries for 31-bit ESA) then *unix\_compat\_ioctl* is used which basically calls *unix\_ioctl* for its functioning.

## **25. *unix\_shutdown***

This function is similar to *close()* but provides more control over the socket. It allows to cut off the communications in certain direction or both. It basically does not close the socket but changes its usability to stop any sending and receiving of data. This function changes the state of the socket to its peers to disable any more exchange of data.

## **26. *unix\_stream\_sendmsg*, *unix\_stream\_recvmsg*, *unix\_dgram\_sendmsg*, *unix\_dgram\_recvmsg*, *unix\_seqpacket\_sendmsg*, *unix\_seqpacket\_recvmsg***

These functions are used to send and receive messages for the various types of Unix domain sockets.

For stream sockets: The send function uses the scm operations to send the message. It checks the state if the socket is ready to send data, checks if the socket is not shutdown, checks for the size of the message if within the max limit, sends the file descriptor in the first buffer and later sends the data. The receive function will take the message and call the actor to perform the reading and it returns the return value of generic read.

For datagram sockets: The send function here also uses scm operation to send the message. It checks for the name of the socket, the message is of correct size or not, if the socket is ready to send, sockets not shutdown, sending to correct peer or not and wakes up the socket process if required, also updates the peer credentials and flags along with the message. The send function also keeps a timestamp of each message passed. The receive function checks for the flags along with the message, receives the data and checks for more packet if coming.

It pools for the peer and copies the data into its buffer when it has completely received entire message. Later it checks for the timestamp associated with each data and sets the scm credentials before freeing it.

For seqpacket sockets: The seqpacket sockets basically uses *unix\_dgram\_sendmsg* and *unix\_dgram\_recvmsg* functions to send and receive messages before checking for the state of the socket.

## **27. *unix\_stream\_sendpage***

This function is used by stream sockets. It basically sends an entire page instead of a message. The sending operation is similar to that of sending a message in stream sockets.

## **28. *unix\_set\_peek\_off***

This function sets the peek off value for the socket. Peek is used to check for out of band data.

## **29. *CONFIG\_PROC\_FS***

If it is enabled, then *unix\_show\_fdinfo* function can be used to read the file descriptor of the socket to get its information.

## **30. *unix\_stream\_ops*, *unix\_dgram\_ops*, *unix\_seqpacket\_ops*, *unix\_proto***

The first three are the structures which define the operations for the three types of the unix domain sockets. The *unix\_proto* is the structure which defines name, owner and size for the module *af\_unix.c*.

```
726. File: /net/unix/af_unix.c
727. static const struct proto_ops unix_dgram_ops = {
728.     .family = PF_UNIX,
729.     .owner = THIS_MODULE,
730.     .release =      unix_release,
731.     .bind =         unix_bind,
732.     .connect =      unix_dgram_connect,
733.     .socketpair =   unix_socketpair,
734.     .accept = sock_no_accept,
735.     .getname =      unix_getname,
736.     .poll =         unix_dgram_poll,
737.     .ioctl =  unix_ioctl,
738. #ifdef CONFIG_COMPAT
739.     .compat_ioctl =  unix_compat_ioctl,
740. #endif
741.     .listen = sock_no_listen,
742.     .shutdown =   unix_shutdown,
743.     .setsockopt =  sock_no_setsockopt,
744.     .getsockopt =  sock_no_getsockopt,
```



```

745.         .sendmsg =      unix_dgram_sendmsg,
746.         .recvmsg =      unix_dgram_recvmsg,
747.         .mmap =         sock_no_mmap,
748.         .sendpage =     sock_no_sendpage,
749.         .set_peek_off =  unix_set_peek_off,
750.         .show_fdinfo =   unix_show_fdinfo,
751.};

```

### **31. *unix\_create1, unix\_create***

These functions are used to create new sockets for complete connections. The *unix\_create* is the create operation assigned to the protocol family *PF\_UNIX*.

The *unix\_create* function checks for the type of the socket to be created and assign operations accordingly and returns the return value of *unix\_create1*. The *unix\_create1* function is called by the stream type of socket and it initialize all the things related to the socket and its data like writespace, destructor, waitqueue, etc.

### **32. *unix\_find\_other***

This function checks for already existing socket of the same name or inode. This function is called whenever the connect operation is to be done and the need to look for the other endpoint socket for sending and receiving data.

### **33. *unix\_mknod***

This function is used to create an inode in the file system for the socket pathname provided.

### **34. *unix\_state\_double\_lock, unix\_state\_double\_unlock***

These functions are used to put lock and unlock the state for both the endpoint sockets when a connection is made.

### **35. *unix\_wait\_for\_peer***

This function is called when you exclusively want to wait for a peer till it gets scheduled.

### **36. *unix\_sock\_inherit\_flags***

This function is called during the accept operation when you need to update the flags with the socket.

### **37. *unix\_scm\_to\_skb, unix\_skb\_scm\_eq, scm\_stat\_add, scm\_stat\_del***

The first function is used to get the scm socket credentials and update them for your socket. The second function is used to verify if the socket credentials are same in scm and your socket or not. The last two functions are used to increment and decrement the count for file pointer when sending and receiving the messages in stream and datagram sockets.

**38. *maybe\_add\_creds, maybe\_init\_creds***

The former is used to add the credentials for your socket when sending message in datagram socket. The latter is used to initialize credentials when sending page in stream socket.

**39. *unix\_copy\_addr***

This function is used to get the copy the socket address in the message header when receiving data in stream and datagram sockets.

**40. *unix\_stream\_data\_wait***

This function is used when receiving message in stream sockets to wait for more data coming.

**41. *unix\_skb\_len***

This function is used to get the length of the socket buffer.

**42. *unix\_stream\_read\_state***

This is a structure used for keeping information about the state of the socket when receiving data in stream sockets.

**43. *unix\_stream\_read\_generic, unix\_stream\_read\_actor, unix\_stream\_splice\_actor, unix\_stream\_splice\_read***

The first function is used to check the state of the socket, keep track of the different messages received in stream socket, maintain the order the messages received in and keeping the messages coming from different writers separate. The second function is used to copy the message into the socket buffer. The third function is used connect the chunks of data together of the same message.

The fourth function is used to read the splices of data and then form the message by calling *unix\_stream\_splice\_actor*.

**44. *unix\_inq\_len, unix\_outq\_len***

These functions are used to measure the length of the socket for incoming and outgoing ioctl queries.

**45. *unix\_open\_file***

This function is used to open the file for the socket and return its file descriptor for an ioctl command request.

**46. *net\_proto\_family***

This is a structure used to define the family operations for the *PF\_UNIX* family.

**47. *unix\_net\_init, unix\_net\_exit, pernet\_operations***

The first two are the init and exit operations defined in the structure *pernet\_operations* for the network namespaces specific data.

**48. *af\_unix\_init, af\_unix\_exit***

The first one is the initcall for the *AF\_UNIX* module which register the family operations and the pernet operations. It is called using the file system initcall. The second one the exit call for unregistering the operations.

**49. *unix\_inflight, unix\_notinflight***

These functions are used to keep the count for the number of times messages sent using the file descriptor.

**50. *unix\_destruct\_scm***

This is the destructor for scm socket.

**51. *unix\_gc, wait\_for\_unix\_gc***

The first function is used to remove the file descriptors associated with sockets are not having any external reference. The second function is used to force a garbage collection when the number of inflight sockets is high.

**52. *unix\_sk***

This is the function used to get the structure pointer for the socket.

**53. *struct unix\_address***

This is the structure used to keep the address related information for the socket.

**54. *struct unix\_skb\_parms***

This is the structure used to keep the socket related parameter information.

**55. *struct scm\_stat***

This is the structure used as a duplicate copy of scm stats (the count of a file pointer in scm).

## 56. *struct unix\_sock*

This is the per socket structure for the *AF\_UNIX* socket.

```
55. File: /include/net/af_unix.h
56. /* The AF_UNIX socket */
57. struct unix_sock {
58.     /* WARNING: sk has to be the first member */
59.     struct sock          sk;
60.     struct unix_address  *addr;
61.     struct path          path;
62.     struct mutex         iolock, bindlock;
63.     struct sock          *peer;
64.     struct list_head     link;
65.     atomic_long_t        inflight;
66.     spinlock_t           lock;
67.     unsigned long        gc_flags;
68. #define UNIX_GC_CANDIDATE    0
69. #define UNIX_GC_MAYBE_CYCLE  1
70.     struct socket_wq     peer_wq;
71.     wait_queue_entry_t    peer_wake;
72.     struct scm_stat      scm_stat;
73. };
```

## CHAPTER 6: INVESTIGATION FOR SOCKETS

To find information about the Unix domain sockets in Linux, you can use the following commands:

1. The list of all unix sockets can be obtained using the command “ss -x -a”

```
virsav@vs-ubuntu_16:~$ ss -x -a
Netid  State      Recv-Q Send-Q Local Address:Port      Peer Address:Port
u_str  LISTEN     0      128   @/tmp/.ICE-unix/6664 29269      * 0
u_dgr  UNCONN     0      0     /run/user/1000/systemd/notify 28970      * 0
u_dgr  UNCONN     0      0     /run/user/108/systemd/notify 15345      * 0
u_str  LISTEN     0      128   /run/user/1000/systemd/private 28971      * 0
u_str  LISTEN     0      128   /run/user/108/systemd/private 15346      * 0
u_seq  LISTEN     0      128   /run/udev/control 16838      * 0
u_str  LISTEN     0      128   /run/user/1000/keyring/control 22356      * 0
u_str  LISTEN     0      128   /run/user/108/keyring/control 23592      * 0
u_dgr  UNCONN     0      0     /run/systemd/cgroups-agent 16833      * 0
u_str  LISTEN     0      128   /run/systemd/private 16834      * 0
u_str  LISTEN     0      128   /run/user/1000/keyring/pkcs11 29935      * 0
u_str  LISTEN     0      128   /run/user/1000/keyring/ssh 29937      * 0
u_dgr  UNCONN     0      0     /run/systemd/journal/dev-log 16839      * 0
u_str  LISTEN     0      128   /run/systemd/fscck.progress 16840      * 0
u_dgr  UNCONN     0      0     /run/systemd/journal/syslog 16842      * 0
u_str  LISTEN     0      128   /run/systemd/journal/stdout 16846      * 0
u_dgr  UNCONN     0      0     /run/systemd/journal/socket 16847      * 0
u_str  LISTEN     0      5     /run/user/1000/pulse/native 30076      * 0
```

Figure 7. Investigation command 1

2. Another way to get list of all unix sockets is by using command “cat /proc/net/unix”

```
virsav@vs-ubuntu_16:~$ cat /proc/net/unix
Num          RefCount Protocol Flags      Type St Inode Path
0000000000000000: 00000002 00000000 00010000 0001 01 29269 @/tmp/.ICE-unix/6664
0000000000000000: 00000002 00000000 00010000 0002 01 28970 /run/user/1000/systemd/notify
0000000000000000: 00000002 00000000 00010000 0001 01 28971 /run/user/1000/systemd/private
0000000000000000: 00000002 00000000 00010000 0005 01 16838 /run/udev/control
0000000000000000: 00000002 00000000 00010000 0001 01 22356 /run/user/1000/keyring/control
0000000000000000: 00000002 00000000 00010000 0002 01 16833 /run/systemd/cgroups-agent
0000000000000000: 00000002 00000000 00010000 0001 01 16834 /run/systemd/private
0000000000000000: 00000002 00000000 00010000 0001 01 29935 /run/user/1000/keyring/pkcs11
0000000000000000: 00000002 00000000 00010000 0001 01 29937 /run/user/1000/keyring/ssh
```

Figure 8. Investigation command 2

3. For checking for a specific state, you can use the command “ss -x state [FILTER-NAME]”

Replace [FILTER-NAME] with whatever filter you wanna provide for the state name.

E.g. established, closed, connected, etc.

```

virsav@vs-ubuntu_16:~$ ss -x state established
Netid Recv-Q Send-Q Local Address:Port Peer Address:Port
u_str 0 0 * 62455 * 68669
u_str 0 0 * 33440 * 35329
u_str 0 0 @/tmp/.ICE-unix/6664 27619 * 27618
u_str 0 0 * 29871 * 26163
u_str 0 0 @/tmp/dbus-cYm5nKZFdN 29836 * 25353
u_str 0 0 /run/systemd/journal/stdout 104968 * 102999
u_str 0 0 * 26409 * 25560
u_str 0 0 * 24242 * 26310
u_str 0 0 * 30156 * 26509
u_str 0 0 * 35327 * 32509
u_str 0 0 * 32839 * 28655
u_str 0 0 /var/run/dbus/system_bus_socket 30792 * 27535

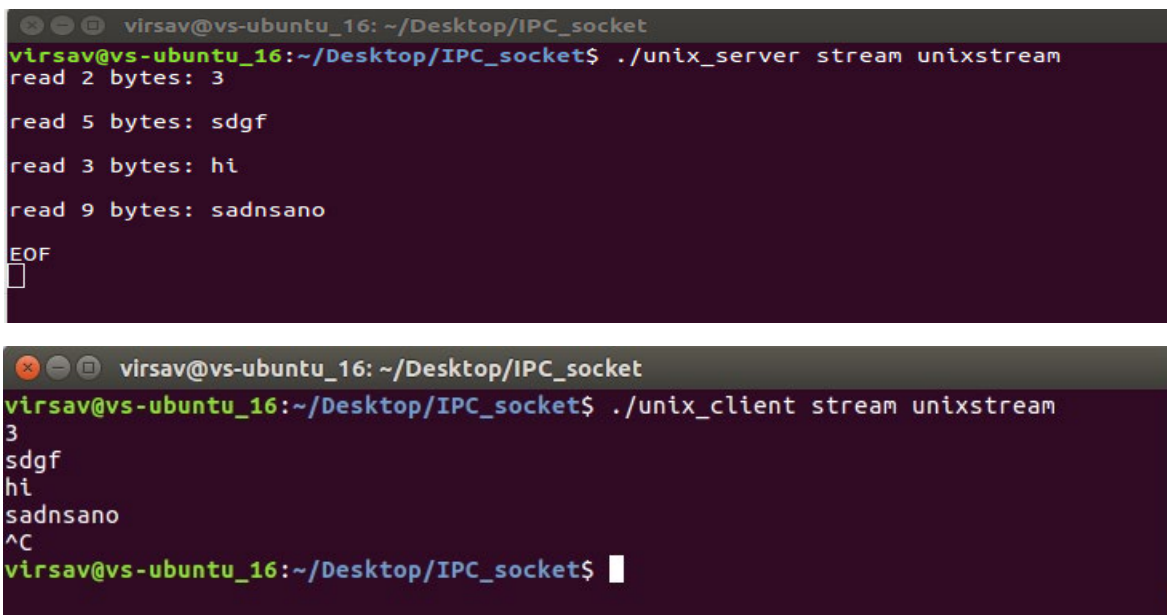
```

Figure 9. Investigation command 3

## CHAPTER 7: SAMPLE CODE

The sample code for running Unix stream and Unix datagram sockets is submitted along with this report. The code submitted is written by referring online sources<sup>[17][18]</sup> and looks very similar to the sources. I have tried to make a single file from which we can create any type of Unix socket required. The basic steps followed for creating the socket remains the same and hence I have used the sources to incorporate my idea of single file for any Unix domain IPC socket. Please check the zip folder for `unix_server.c` and `unix_client.c` files. Also, a header file is included and a Makefile is provided to compile these two user-space programs. The header file submitted is the header file provided in the textbook<sup>[19]</sup>. In both the applications I am taking in command line arguments to create stream or datagram socket. Depending on your choice you can use the application programs provided to create either of the sockets for server as well as client. Both the sockets are an example of a very basic socket implementation.

When Unix stream socket is created, the command line looks like `./unix_server stream unixstream` for server endpoint and `./unix_client stream unixstream` for client endpoint. Always first server end should be established as if you try to create the client endpoint first then it will tell you as “connection refused” as it does not find any server present to transfer data. Also, the second argument is stream which specifies the server/client to create stream socket at its end. The third argument is `unixstream` which creates a socket pathname file named “`unixstream`” in order to successfully transfer data. If the client provides some other pathname at its command line, then it will return no such file present (if a file of that name is not present). The successful establishment of stream socket and its output can be seen below. The client keeps sending data until it is closed, and the server reads the number of bytes of the message received and the message as well. The server returns an EOF when the client socket is closed.

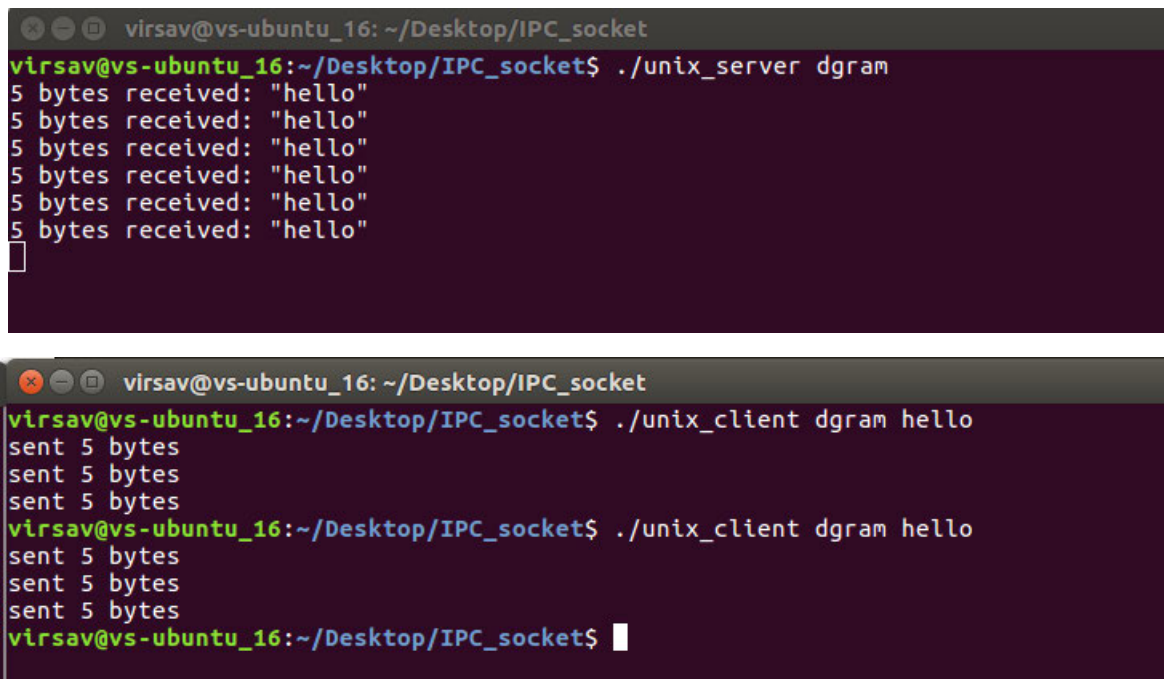


```
virsav@vs-ubuntu_16: ~/Desktop/IPC_socket
virsav@vs-ubuntu_16:~/Desktop/IPC_socket$ ./unix_server stream unixstream
read 2 bytes: 3
read 5 bytes: sdgf
read 3 bytes: hi
read 9 bytes: sadnsano
EOF
^C
```

```
virsav@vs-ubuntu_16: ~/Desktop/IPC_socket
virsav@vs-ubuntu_16:~/Desktop/IPC_socket$ ./unix_client stream unixstream
3
sdgf
hi
sadnsano
^C
virsav@vs-ubuntu_16:~/Desktop/IPC_socket$
```

Figure 10 & 11. Sample output for UNIX stream socket

When Unix datagram socket is created, the command line looks like “./unix\_server dgram” and “./unix\_client dgram hello” for server and client endpoint respectively. For datagram socket as well, always first server endpoint should be established because if you try to create client socket first then it will say “send failed: connection refused” as the client is unable to find the server destination address it is looking for. The second argument is to specify the client/server to create a datagram socket at its endpoint. The successful implementation of Unix datagram socket and the output can be seen below. Whenever the client end socket is created along with a message being the third argument, it will send the message 3 times and the server end will read the message 3 times. The number of times the message to be sent is hardcoded and can be changed.



```
virsav@vs-ubuntu_16: ~/Desktop/IPC_socket
virsav@vs-ubuntu_16:~/Desktop/IPC_socket$ ./unix_server dgram
5 bytes received: "hello"
5 bytes received: "hello"
5 bytes received: "hello"
5 bytes received: "hello"
5 bytes received: "hello"
5 bytes received: "hello"
virsav@vs-ubuntu_16:~/Desktop/IPC_socket$

virsav@vs-ubuntu_16: ~/Desktop/IPC_socket
virsav@vs-ubuntu_16:~/Desktop/IPC_socket$ ./unix_client dgram hello
sent 5 bytes
sent 5 bytes
sent 5 bytes
virsav@vs-ubuntu_16:~/Desktop/IPC_socket$ ./unix_client dgram hello
sent 5 bytes
sent 5 bytes
sent 5 bytes
virsav@vs-ubuntu_16:~/Desktop/IPC_socket$
```

Figure 12 & 13. Sample output for UNIX datagram socket

When a Sequenced-packet socket is tried to be created using the given programs then it will return “Work in progress for SOCK\_SEQPACKET”. I have not tried to add the third type of Unix socket in my program because of time constraint. But it should be straightforward to do it as it is a combination of the first two types. For any other name provided for the Unix socket type to the given programs, they will return “No such socket type in Unix”.



## REFERENCES

1. [https://elixir.bootlin.com/linux/latest/source/net/unix/af\\_unix.c#L126](https://elixir.bootlin.com/linux/latest/source/net/unix/af_unix.c#L126)
2. <http://man7.org/linux/man-pages/man7/unix.7.html>
3. CSE530: Embedded Operating System Internals, EOSI\_8\_2020\_Linux\_Socket.ppt, by Dr.Yann-Hang Lee, ASU, Spring 2020.
4. <https://elixir.bootlin.com/linux/v5.6.11/source/net/socket.c>
5. <https://upsilon.cc/~zack/teaching/1314/progsyst/cours-09-socket-unix.pdf>
6. <https://ops.tips/blog/how-linux-creates-sockets/>
7. [http://openbook.rheinwerk-verlag.de/linux\\_unix\\_programmierung/Kap11-005.htm#top](http://openbook.rheinwerk-verlag.de/linux_unix_programmierung/Kap11-005.htm#top)
8. <https://www.cyberciti.biz/tips/linux-investigate-sockets-network-connections.html>
9. <https://brennan.io/2017/03/08/sock-net/>
10. [http://man7.org/conf/lca2013/IPC\\_Overview-LCA-2013-printable.pdf](http://man7.org/conf/lca2013/IPC_Overview-LCA-2013-printable.pdf)
11. <https://www.ibm.com/developerworks/linux/library/l-vercon/>
12. Rosen, Rami. Linux Kernel Networking: Implementation and Theory. Apress, Jan 2014
13. W. Richard Stevens, Bill Fenner, Andrew M. Rudoff. UNIX Network Programming Volume 1, Third Edition: The Sockets Networking API. Addison Wesley, November 2003.
14. [https://www.ibm.com/support/knowledgecenter/SSB23S\\_1.1.0.15/gtpc1/unixsock.html](https://www.ibm.com/support/knowledgecenter/SSB23S_1.1.0.15/gtpc1/unixsock.html)
15. <https://beej.us/guide/bgnet/html/#datagram>
16. Warren W. Gay. Linux Socket Programming by Example. Que, April 2000.
17. <https://github.com/troydhanson/network/tree/master/unixdomain/01.basic>
18. <http://www.cs.columbia.edu/~jae/4118/L09-domain-sockets.html>
19. W. Richard Stevens, Stephen A. Rago. Advanced Programming in the Unix Environment Third Edition. Addison Wesley, 1992.