# The Unix Socket Protocol

Michael Jantz

Prasad Kulkarni

## Introduction

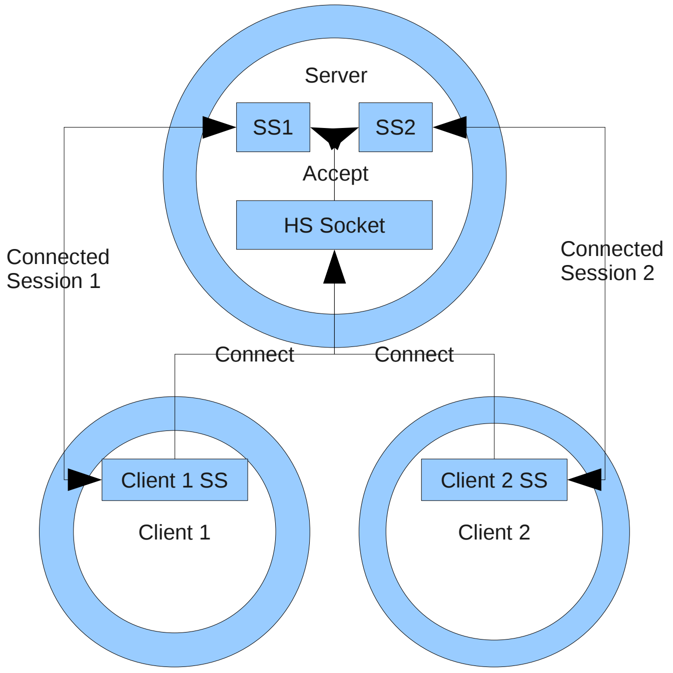
- In this lab, we will use the Unix socket protocol to send data from a client to a server and back to the client.
- As you work through this lab, the following man pages might be helpful:
  - socket(2), unix(7), tcp(7), bind(2), listen(2), accept(2), connect(2), read(2), write(2)
  - Man pages come in different sections. System calls are all in section 2. To access a page from a section other than section 1, give the section as an argument (e.g. man 2 socket)

## What is a Socket?

- A socket is a two-way communication channel between two processes. Sockets can be thought of as bidirectional pipes.
- Each process has access to its own socket endpoint, which can be used to read data from or write data to other endpoints.
- Socket endpoints can be on the same machine or across a network.

# Client / Server Model

- In this lab, we will create two programs which communicate using the client / server model.
- In this model, the server establishes a **handshake socket** with a particular address. Client programs create a **session socket**, which attempts to connect to the handshake socket. When the handshake socket receives a request to connect, the server accepts the connection and creates another session socket which is connected to the client's session socket. These session sockets can be used to exchange data between the client and server processes.



# Creating Sockets

- Sockets are created using the socket() system call:
  - int socket(int domain, int type, int protocol)
- The domain parameter specifies a communication domain; this selects the protocol family which we will use.
  - The socket protocol governs how a socket is accessed and how it can be used. For this lab, we will use the PF_UNIX protocol family.
- The type parameter specifies the communication semantics.
  - e.g. SOCK_STREAM. More on this later.
- The protocol parameter specifies a particular protocol to be used with the socket (within the protocol family selected).
  - Most protocol families only have one associated protocol option, and thus, this parameter is usually 0.

## The Unix Socket Protocol

- To create a Unix (local) socket in C:
  - sockfd = socket(PF_UNIX, SOCK_STREAM, 0)
- The Unix socket family can be used to efficiently communicate between processes on the same machine.
- Because sockets can be used to connect unrelated processes, socket endpoints must be bound to an address which both processes are aware of.
- Protocol families typically differ in how an address is established and what is used as an address.

# Streaming Sockets

- The SOCK_STREAM argument specifies that we will use the socket as a two-way connection-based socket.
- This is in contrast to datagram sockets, which do not establish safe connections and support packet broadcast (sending to all endpoints in a network)
- This requires that we establish a connection between the two socket endpoints before exchanging data.
- In order to establish a connection, we must declare a common address that both endpoints can refer to.

# Establishing an Address

- Unix sockets are addressed by creating a file on the file system which socket endpoints refer to.
  - As a point of comparison, the IPv4 and IPv6 socket families can be used to communicate between processes on different machines. They use an IP address and port number to address socket endpoints.
- The format of a Unix socket address is given in the Unix manual:

```
#define UNIX_PATH_MAX 108
struct sockaddr_un {
        sa_family_t sun_family;               /* AF_UNIX */
        char sun_path[UNIX_PATH_MAX];         /* pathname */
}
```

- In both the client and server, populate the sockaddr_un structure with appropriate values:
  - sun_family should be set to the predefined macro AF_UNIX (e.g. saun.sun_family = AF_UNIX)
  - sun_path should be a common pathname. Use strcpy to copy this pathname from SOCKET_ADDRESS defined constant. Note the path name is relative to current working directory..

## Creating the Handshake Socket

- In server.c, let's first create the handshake socket we will use to accept connections.
- Use the socket system call with the arguments shown above to create a Unix socket.
- The return value is a file descriptor representing the socket.

# Binding to an Address

- Use the bind() system call to bind a socket to an address:
  - int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)
  - sockfd is the file descriptor referring to the socket you just created (record the return value of socket() to get this file descriptor)
  - addr is the address you wish to bind the socket to. Observe that there is only one bind() system call for every socket protocol. Socket protocols typically have different address formats. Therefore, you will have to cast the sockaddr_un structure to a sockaddr structure.
  - addrlen is the size of the address structure. You can get this easily using the builtin sizeof() function, e.g.
    - sizeof(saun)
  - bind() returns 0 on success. On error, -1 is returned and errno is set appropriately.

## Getting the Socket to Listen

- Now that we have the socket bound to an address, we need to tell the operating system that this socket is a handshake socket (i.e. one that does not transfer data, but only listens for sockets trying to connect).
- This is performed with the listen system call:
  - int listen(int sockfd, int backlog)
  - sockfd will be the file descriptor of the socket you've already created
  - backlog is the maximum length to which the queue of pending connections to sockfd may grow. It is irrelevant for our example (any positive integer will do), but is necessary for the following reason:
    - When the client sends a connection request to the server, the server receives the request, sends the client an acknowledgement packet, and puts the client in a pending connections queue.
    - When the client receives the acknowledgement packet, it will send another packet to the server acknowledging this. Only when the server receives this packet will the client be taken off the pending connections queue, and the connection established.
    - Without a maximum length for the pending queue, a hostile client could very easily drain the resources of a server by sending many connection requests without ever sending the final acknowledgement packet.
  - listen() returns 0 on success. On error, -1 is returned and errno is set appropriately.

# Accepting Connections

- When you have established a listening socket bound to an address, you can now use this socket to accept connections:
  - int accept(int sockfd, struct sockaddr *addr, socklen_t addrlen)
  - sockfd is the file descriptor of the listening socket which will accept connections
  - The other arguments are not relevant for the protocol we are using. You can just pass in NULL for these arguments.
- accept() will block until a client successfully connects to the listening socket.
- When this happens, a new socket is created that is connected to the client's endpoint, and accept() returns a file descriptor for this new socket.

# Connecting the Client

- Now that the server is setup, time to connect the client:
  - Create another Unix socket in client.c using the socket() system call.
  - If you have not done so already, populate the sockaddr_un struct (saun) with the same values you had used for the server
  - Now, call connect() to tell the client to attempt to connect to the server:
    - int connect(int sockfd, struct sockaddr *addr, socklen_t addrlen)
    - sockfd should be the file descriptor for the newly created socket
    - addr is the socket address you wish to attempt a connection on. Remember to cast the sockaddr_un variable in the client here.
    - addrlen is the size of the address struct passed to this system call. This can be obtained easily with sizeof(saun).
    - On success, 0 is returned. On error, -1 is returned, and errno is set appropriately.
  - If the connection succeeds, the socket pointed to by sockfd will now be connected to another socket endpoint at the address provided (in our case, the socket returned by the accept() call in the server).

## An Uppercase String Converter

- At this point, each the client and the server should have access to a file descriptor that points to a connected socket endpoint.
- In this lab, we will use these sockets to implement a server which performs uppercase string conversion. The client will send several strings to the server, the server will convert these strings to uppercase, and then send the converted strings back to the client. This is very similar to what you will do in the first class project.
- I have predefined an array of strings to pass at the top of client.c.
  - The array is called strs. As it is defined, strs[i] refers to the i'th string in strs.
- You should use a for loop in client.c to iterate over the strings in the strs array, send them to the server, and receive them from the server.
- The server should read the strings until no more are sent, convert each string to uppercase (using the predefined convert_string), and send each string back to the client.

## Using Read and Write

- You can use the read() and write() system calls to actually read and write the strings to and from the socket endpoints.
  - Although, it should be noted, you do not have to. Sockets are just like any other file. Any file I/O methods can be used to read and write to them (e.g. you could convert the file descriptor to a file pointer using fdopen() and use the file stream functions (fgets, fputs, etc.) to read and write to the stream).
- The read() system call looks like:
  - ssize_t read(int fd, void *buf, size_t count)
  - It should be clear by now what the arguments for read mean. If you forget what any one of them means, a quick glance at the man page will answer your questions (man 2 read).
  - The write() system call is similar.
- If you use read() and write(), a word on the size argument:
  - The size argument given to the system call does not have to be exact. It just has to be sufficiently large to read or write the whole string into the buffer. When you print out the buffer, it will only print the string because it is terminated by a null character ('\0').

# Finishing Up

- When you are done, uncomment the printf statements in  each the client and the server. When you run both the client and the server (note you should always start the server first), your output should look like this:

```
-bash-3.2$ ./server
RECEIVED:
this is the first string from the client
SENDING:
THIS IS THE FIRST STRING FROM THE CLIENT

RECEIVED:
this is the second string from the client
SENDING:
THIS IS THE SECOND STRING FROM THE CLIENT

RECEIVED:
this is the third string from the client
SENDING:
THIS IS THE THIRD STRING FROM THE CLIENT
```

```
-bash-3.2$ ./client
SENDING:
this is the first string from the client
RECEIVED:
THIS IS THE FIRST STRING FROM THE CLIENT

SENDING:
this is the second string from the client
RECEIVED:
THIS IS THE SECOND STRING FROM THE CLIENT

SENDING:
this is the third string from the client
RECEIVED:
THIS IS THE THIRD STRING FROM THE CLIENT
```