Lab04 : Design and implement a file transfer server using the Socket API

Learning Objectives

- Use socket API to connect to a remote application.
- FTP protocol can send a file with socket API.
- Handle errors

1. Introduction

In today's lab, you will rewrite Lab 03's simple File Transfer Protocol(FTP) server and client to use the Unix/Linux Socket API instead of the simplified API. Start with the code from last week's client and server and an example set of socket calls we will supply. Remove the simplified API calls from your code, and replace them with actual socket calls.

2. Client and Server specification

You will wipe out simplified API from your FTP server / client application to replace it with Socket API. Of course, you must start with a valid client and server from Lab 3 Lab 03. Examine the textbook and the code for socket access functions available at socket-access-functions.tar to understand how the Socket API function calls in creating a socket and establishing a connection between a client and a server.

3. Overview of the socket interface

An example client-server dialogue
3.1. Socket API functions

Here are the list of socket API functions that you should use.

socket()

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

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

socket() creates an endpoint for communication and returns a file descriptor for the socket. socket() takes three arguments:

- domain, which specifies the protocol family of the created socket. For example:
  - AF_INET for network protocol IPv4 or
  - AF_INET6 for IPv6.
  - AF_UNIX for local socket (using a file).
- type, one of:
  - SOCK_STREAM (reliable stream-oriented service or Stream Sockets)
  - SOCK_DGRAM (datagram service or Datagram Sockets)
  - SOCK_SEQPACKET (reliable sequenced packet service), or
  - SOCK_RAW (raw protocols atop the network layer).
- protocol specifying the actual transport protocol to use. The most common are IPPROTO_TCP, IPPROTO_SCTP, IPPROTO_UDP, IPPROTO_DCCP. These protocols are specified in
<netinet/in.h>. The value 0 may be used to select a default protocol from the selected domain and type.

The function returns -1 if an error occurred. Otherwise, it returns an integer representing the newly-assigned descriptor.

Since you should use TCP and IPv4 for your application, use SOCK_STREAM and AF_INET.

**bind()**

```c
int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);
```

bind() assigns a socket to an address. When a socket is created using socket(), it is only given a protocol family, but not assigned an address. This association with an address must be performed with the bind() system call before the socket can accept connections to other hosts. bind() takes three arguments:

- sockfd, a descriptor representing the socket to perform the bind on.
- my_addr, a pointer to a sockaddr structure representing the address to bind to.
- addrlen, a socklen_t field specifying the size of the sockaddr structure.

bind() returns 0 on success and -1 if an error occurs.

**listen()**

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

After a socket has been associated with an address, listen() prepares it for incoming connections. However, this is only necessary for the stream-oriented (connection-oriented) data modes, i.e., for socket types (SOCK_STREAM, SOCK_SEQPACKET). listen() requires two arguments:

- sockfd, a valid socket descriptor.
- backlog, an integer representing the number of pending connections that can be queued up at any one time. The operating system usually places a cap on this value.

Once a connection is accepted, it is dequeued. On success, 0 is returned. If an error occurs, -1 is returned.

**accept()**

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

When an application is listening for stream-oriented connections from other hosts, it is notified of such events (cf. select() function) and must initialize the connection using the accept() function. The accept() function creates a new socket for each connection and removes the connection from the listen queue. It takes the following arguments:

- sockfd, the descriptor of the listening socket that has the connection queued.
The accept() function returns the new socket descriptor for the accepted connection, or -1 if an error occurs. All further communication with the remote host now occurs via this new socket. Datagram sockets do not require processing by accept() since the receiver may immediately respond to the request using the listening socket.

connect()

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

The connect() system call connects a socket, identified by its file descriptor, to a remote host specified by that host's address in the argument list. Certain types of sockets are connectionless, most commonly user datagram protocol sockets. For these sockets, connect takes on a special meaning: the default target for sending and receiving data gets set to the given address, allowing the use of functions such as send() and recv() on connectionless sockets. connect() returns an integer representing the error code: 0 represents success, while -1 represents an error.

gethostbyname() and gethostbyaddr()

```c
struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const void *addr, int len, int type);
```

The gethostbyname() and gethostbyaddr() functions are used to resolve host names and addresses in the domain name system or the local host's other resolver mechanisms (e.g., /etc/hosts lookup). They return a pointer to an object of type struct hostent, which describes an Internet Protocol host. The functions take the following arguments:

- **name** specifies the name of the host. For example: `www.cs.purdue.edu`
- **addr** specifies a pointer to a struct in_addr containing the address of the host.
- **len** specifies the length, in bytes, of addr.
- **type** specifies the address family type (e.g., AF_INET) of the host address.

The functions return a NULL pointer in case of error, in which case the external integer h_errno may be checked to see whether this is a temporary failure or an invalid or unknown host. Otherwise a valid struct hostent * is returned. These functions are not strictly a component of the BSD socket API, but are often used in conjunction with the API functions. Furthermore, these functions are now considered legacy interfaces for querying the domain name system. New functions that are completely protocol-agnostic (supporting IPv6) have been defined. These new function are getaddrinfo() and getnameinfo(), and are based on a new addrinfo data structure.

recv()
ssize_t recv(int sd, void *buf, size_t nbytes, int flags);

Read at most nbytes bytes into buf from sd. Returns -1 on error, 0 on EOF (connection closed), or the actual number of bytes read. The function take the following arguments:

- sd: socket descriptor.
- buf: address of char buffer.
- nbytes: sizeof buf.

You should use 0 for the flags in your application.

send()

ssize_t send(int sd, const void *buf, size_t nbytes, int flags);

Attempts to write nbytes from buf to sd. Returns -1 on error or the number of bytes actually written. Arguments similar to recv().

You should use 0 for the flags in your application.

close()

int close(int sd);

Close socket descriptor sd. Returns -1 on failure, otherwise 0.

3.2 Steps for setting up a server and a client

Server

Setting up a simple TCP server involves the following steps:

1. Creating a TCP socket, with a call to socket().
2. Binding the socket to the listen port, with a call to bind(). Before calling bind(), a programmer must declare a sockaddr_in structure, clear it (with memset()), and the sin_family (AF_INET), and fill its sin_port (the listening port, in network byte order) fields. Converting a short int to network byte order can be done by calling the function htons() (host to network short).
3. Preparing the socket to listen for connections (making it a listening socket), with a call to listen().
4. Accepting incoming connections, via a call to accept(). This blocks until an incoming connection is received, and then returns a socket descriptor for the accepted connection. The initial descriptor remains a listening descriptor, and accept() can be called again at any time with this socket, until it is closed.
5. Communicating with the remote host, which can be done through send() and recv() or write() and read().
6. Eventually closing each socket that was opened, once it is no longer needed, using close().

Client

Programming a TCP client application involves the following steps:

1. Creating a TCP socket, with a call to socket().
2. Connecting to the server with the use of connect(), passing a sockaddr_in structure with the sin_family set to AF_INET, sin_port set to the port the endpoint is listening (in network byte order), and sin_addr set to the IP address of the listening server (also in network byte order.)
3. Communicating with the server by using send() and recv() or write() and read().
4. Terminating the connection and cleaning up with a call to close().

4. Protocol

The protocol to be implemented in Lab 04 is the same as the Lab 03. Please refer Lab 03.

5. Implementation Guidelines

You have to program a server and client using the socket API only. Do not use simplified API to implement your server and client. We encourage you to examine and thoroughly understand the socket access functions provided in socket-access-functions.tar. You may reuse code form the socket-access-functions in your program, but you must remove the extra layers of functions and call the socket functions directly in your programs (i.e., get rid of the connectsock and connecttcp functions).

6. Test your application

Here are some suggested list of tests to check whether your application works as it supposed to be. You should check:

1. Does it work for a single file?
2. Does it work for a big file? (i.e. greater than 100M)
3. Does it work for multiple files? (i.e. multiple GET / PUT commands)
4. Does it work with empty file and binary files?
5. Does it work for both PUT and GET requests?
6. Does it work across different machines?

You may test to program to see if it runs correctly by performing a diff on the original file and the one that has been received via your FTP program.

Compilation and Makefile

You must wipe out any files related to simplified API. Therefore, remove all files under the following subdirectory:
Lab04 : Design and implement a file transfer server using the Socket API

- api/*
- h/*

Also, update your Makefile under compile_linux/ and compile_solaris/ so that make can compile both of ftpserver.c and ftpclient.c. For example,

```
make
```
must compile ftpserver.c and ftpclient.c.

**Testing ftpserver in binary**

You can test your ftp client's interoperability with our ftpserver. Please download it to one of the xinuXX machine and run from there. It may not work from the other machines such as sslab. For your convenience, our server prints out the all messages and its length verbosely. Please use it for your debugging. (Note that your server shouldn't print out such thing)

Download it from here : [ftp-server-binary](ftp-server-binary)

## 7. Extra-credit

Create a second program that uses the socket API to become an SMTP client and transfer an email message. To demonstrate success, contact an SMTP server and send a short email message to cs422-ta@cs.purdue.edu to demonstrate that you correctly implemented the SMTP protocol. You can find the protocol specification in RFC 2821 ([http://www.ietf.org/rfc/rfc2821.txt](http://www.ietf.org/rfc/rfc2821.txt)).

You can connect any SMTP server in xinuXX.cs.purdue.edu with port number 25. Filename of your code will be smtpclient.c and make this file to be compilable with make - You need to change Makefile under /compile_linux and /compile_solaris.

## 8. Turn-in

You should use turnin command to submit your whole directory.

```
cd ~/cs422
turnin -c cs422 -p lab04 lab04
```

You can check with turnin -v.

```
turnin -c cs422 -p lab04 -v
```

**Turn in your work by 2/6 Friday 11:59pm EST.**

## 9. Grading
### Grading Rubric (tentative)

<table>
<thead>
<tr>
<th>Points</th>
<th>Grading Rubric (tentative)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+5</td>
<td>Client and server communicate with socket API</td>
</tr>
<tr>
<td>+3</td>
<td>Interoperability with different server/client</td>
</tr>
<tr>
<td>+2</td>
<td>Handling errors</td>
</tr>
<tr>
<td>+2</td>
<td>Output correctness</td>
</tr>
<tr>
<td>+3</td>
<td>Overall organization of the program</td>
</tr>
<tr>
<td>+3</td>
<td>Coding style, commenting, etc</td>
</tr>
<tr>
<td>+2</td>
<td>Makefile, compile and run</td>
</tr>
</tbody>
</table>

### Extra Credits

<table>
<thead>
<tr>
<th>Points</th>
<th>Grading Rubric (tentative)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+3</td>
<td>SMTP Client</td>
</tr>
</tbody>
</table>