Networking Protocols and Socket Programming

# TFTP Server Using GNS3

This part will focus on the TFTP protocol. Using GNS3, You need to configure a TFTP server router and a TFTP client router both working on the network 192.168.11.0/24. The TFTP server IP is 192.168.11.1/24 and the TFTP client IP is 192.168.11.2 connected via fast Ethernet. Your goal is to configure the server router as a TFTP server that only servers the 192.168.11.0/24 network only. Whenever the TFTP client asks for a file named YourConfig the TFTP server should send its configuration file.  
Cisco specifies the following:`` The router is not a fully functional TFTP server. It can only serve files for download. You cannot use this feature to upload files into the serving router's local flash. However, the router is not limited to just serving IOS images. You can use your router's flash to store configuration files and make them available for download via TFTP as well. Moreover, you can even use it to hold configuration files for non-Cisco equipment.” The following commands may help you to achieve this task:

* tftp-server ?
* copy tftp ?

For more information, follow this [link](https://www.cisco.com/c/en/us/support/docs/routers/2500-series-routers/15092-copyimage.html).  
You will need to capture the resulted TFTP data packets using Wireshark and to compare the protocol header with the one existing in [RFC 1350](https://tools.ietf.org/html/rfc1350).

# Accessing FTP server using command prompt

To initiate an FTP session from the Windows command prompt, follow these steps:

* Establish an Internet connection as you normally do.
* Click Start, and then click Run.
  + type cmd and then click OK.
  + A command prompt will appear in a new window.
* Type ftp <insert your FTP host address here>
  + Example: C:\ftp [ftp.globalscape.com](ftp://ftp.globalscape.com), C:\ftp myftpsite.com
* Press Enter.
* If the initial connection is successful, you should be prompted for a username. Type it in and press Enter again. (If connecting anonymously, type anonymous)
* You should now be prompted for a password. Type it in and press Enter once more.
  + If all is well, then you should now be connected to the remote FTP site.
* Type dir and then press Enter to see a list of files and folders.
* You can get or send files using any of the commands [here](https://www.serv-u.com/features/file-transfer-protocol-server-windows/commands).
* To end your FTP session, type quit and press Enter.

Capture the generated packets using Wireshark and compare the resulted FTP data header with the one in [RFC 959](https://tools.ietf.org/html/rfc959) and with the TFTP data header resulted in part 1.

# FTP Client-Server via Socket programming

You are required to build an FTP-like server socket. This could be realized through implementing two programs. First, a server socket that listens for any incoming connections. The server provides read and write operations that you will tailor to meet your application needs. Second, a client that connects to the server and GET a file from the server.

## Socket Programming Model

You can find a helpful tutorial on socket programming [here](https://www.csd.uoc.gr/~hy556/material/tutorials/cs556-3rd-tutorial.pdf).

|  |  |
| --- | --- |
|  |  |

## Coding

You will need to generate at least two files:  
1-Server.c  
2-Client1.c

## Header Files

Make sure to include the following header files.

|  |  |
| --- | --- |
| Filename | File function |
| <stdio.h> | This header defines three variable types, several macros, and various functions for performing input and output. |
| <stdlib.h> | This header defines four variable types, several macros, and various functions for performing general functions. |
| <String.h> | This header defines one variable type, one macro, and various functions for manipulating arrays of characters. |
| <unistd.h> | This header file that provides access to the POSIX operating system API. |
| <sys/types.h> | This file contains the definitions of a number of data types and functions used in system calls. This file will help in executing the following two files. |
| <sys/socket.h> | This header includes a number of definitions of structures needed for socket |
| <netinet/in.h> | This header includes a number of definitions of structures and constants needed for internet domain address |

## Server model

### **Socket Creation:**

int Sockfd = socket(int domain, int type, int protocol)

|  |
| --- |
| **Sockfd: socket descriptor, an integer (like a file-handle)**  **The socket function has three arguments:**  **Domain: Communication domain (sometimes referred to as family). Two processes can communicate with each other only if their sockets are of the same type and in the same domain. There are two widely used address domains, 1) *the unix domain*, in which two processes which share a common file system communicate, and 2) *the Internet domain*, in which two processes running on any two hosts on the Internet communicate. Each of these has its own address format. The address of a socket in the Unix domain is a character string which is basically an entry in the file system. The address of a socket in the Internet domain consists of the Internet address of the host machine. There is a list for symbolic consatnts e.g., AF\_INET (IPv4 protocol), AF\_INET6 (IPv6 protocol).**  **Type: Communication type. The most famous socket types are *stream sockets* and *datagram sockets*. Stream sockets treat communications as a continuous stream of characters, while datagram sockets have to read entire messages at once. Stream sockets use TCP and datagram sockets use UDP. The two symbolic constants are 1)SOCK\_STREAM: TCP and 2)SOCK\_DGRAM: UDP.**  **Protocol: If this argument is *zero* (and it always should be except for unusual circumstances), the operating system will choose the most appropriate protocol. It will choose TCP for stream sockets and UDP for datagram sockets.** |

### **Socket Binding:**

int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

|  |
| --- |
| The bind function binds the socket to the address and port number specified in a specific data structure of the type sockaddr.  It takes ***three*** arguments:  **Sockfd**: The socket file descriptor  **addr**: The address to which is bound. The second argument is a pointer to a structure of type sockaddr, but what is passed in is a structure of type sockaddr\_in, and so this must be cast to the correct type. This can fail for a number of reasons, the most obvious being that this socket is already in use on this machine.  struct sockaddr {  unsigned short *sa\_family*; // address family, AF\_xxx  char *sa\_data*[14]; // 14 bytes of protocol address  };  **addrlen**: The size of the address to which it is bound. The size of addr in ***bytes***. |

### Server Listen:

int listen(int sockfd, int backlog);

|  |
| --- |
| It puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection i.e. the process to listen on the socket for connections.  It takes two arguments:  **sockfd**: The socket file descriptor.  **backlog**: The size of the backlog queue, i.e., the number of connections that can be waiting while the process is handling a particular connection. So the backlog defines the maximum length to which the queue of pending connections for sockfd may grow. This should be set to ***five***, the maximum size permitted by most systems. If a connection request arrives when the queue is full, the client may receive an error with an indication of ***ECONNREFUSED***. |

### Client Accept:

int new\_socket= accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

|  |
| --- |
| The accept() system call causes the process to block until a client connects to the server. Thus, it wakes up the process when a connection from a client has been successfully established. It returns a new file descriptor, and all communication on this connection should be done using the new file descriptor. In other words, it extracts the first connection request on the queue of pending connections (remember maximum 5) for the listening socket, sockfd, creates a new connected socket, and returns ***a new file descriptor referring to that socket***. At this point, connection is established between client and server, and they are ready to transfer data.  It takes ***three*** arguments:  Refer to Bind function FYR.  **Note**: This function ***waits*** for the ***connection*** function from the client side. |

## Client Model

### Socket Creation:

int Sockfd = socket(int domain, int type, int protocol)

|  |
| --- |
| **Sockfd: socket descriptor, an integer (like a file-handle)**  **The socket function has three arguments:**  **Domain: Communication domain (sometimes referred to as family). Two processes can communicate with each other only if their sockets are of the same type and in the same domain. There are two widely used address domains, 1) *the unix domain*, in which two processes which share a common file system communicate, and 2) *the Internet domain*, in which two processes running on any two hosts on the Internet communicate. Each of these has its own address format. The address of a socket in the Unix domain is a character string which is basically an entry in the file system. The address of a socket in the Internet domain consists of the Internet address of the host machine. There is a list for symbolic consatnts e.g., AF\_INET (IPv4 protocol), AF\_INET6 (IPv6 protocol).**  **Type: Communication type. The most famous socket types are *stream sockets* and *datagram sockets*. Stream sockets treat communications as a continuous stream of characters, while datagram sockets have to read entire messages at once. Stream sockets use TCP and datagram sockets use UDP. The two symbolic constants are 1)SOCK\_STREAM: TCP and 2)SOCK\_DGRAM: UDP.**  **Protocol: If this argument is *zero* (and it always should be except for unusual circumstances), the operating system will choose the most appropriate protocol. It will choose TCP for stream sockets and UDP for datagram sockets.** |

### Client Connection:

int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

|  |
| --- |
| The connect function is called by the client to establish a connection to the server. The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr.  **Note**: Server’s address and port are specified in addr. |

## Generic Functions:

### Read/receive

You may implement these functions as you like.

Hint: you will need at least: buffer to receive the data in it, determine the buffer size, and specify the new sockfd. You may add flags, etc.

### Write/send

You may implement these functions as you like.

Hint: you will need at least: buffer to receive the data in it, determine the buffer size, and specify the new sockfd. You may add flags, etc.

### Close

int close(int newsockfd);

|  |
| --- |
| This function terminates the connection between the client and server and vice versa. |