Exercise 3 - Socket Programming

CMPE-570/670 Data & Computer Communications – 2215 Spring
Due Date: 4/19/2022

Description

In this exercise you are asked to design and develop a custom *concurrent* FTP Server named **TigerS** and its associated FTP client named **TigerC**. Once **TigerS** starts on a host, it will listen for incoming requests from multiple **TigerC** clients. You are required to implement five basic commands for the **TigerC** as well as supporting functionality in the **TigerS**:

1. **tconnect <TigerS IP Address> <User> <Password>**: This command takes the IP address (dotted decimal notation) of the TigerS, and the client user name and a password as parameters. If the provided user name and password are correct, then the client is allowed to connect TigerS, otherwise an error message is returned.

2. **tget <File Name>:** The command expects the name of the file to download from TigerS as the parameter. If the file exists, it is downloaded from the server, otherwise an error message is displayed. Files can be either ASCII or Binary files.

3. **tput <File Name>**: The tput command takes the name of the file to upload and uploads it to the TigerS host. If the file does not exist, and error should be returned to the user. Files can be either ASCII or Binary files.

4. **tlist:** The tlist command queries the server for the list of files for the server's current working directory. Since the TigerS is a concurrent server, the current working directory for two clients may not be the same, and thus return a different directory listing.

5. **tcwd <directory name>:** The tcwd command tells the server to change the working directory for the calling client's context. The command may take either a relative path (such as "**..**"or "**~/blah**") or an absolute path (such as "**/home/user/folder** "). Issuing this command from one client must not affect the working directory of any other client currently connected to the server or future connections to the server.

What to submit?

Please prepare a professionally written report that will include the following, and submit it to "mycourses dropbox – SocketProgramming" together with your tar-zipped client and server folders:

1. Provide both a high-level requirements and detailed requirements section.

2. Describe your design approach; the transport protocol used and the implementation details and tradeoffs.

3. Provide the Command-and-Control messaging and format of data.

4. How did you implement concurrency? Explain and show your work.

5. What is the maximum number of clients that your server can process concurrently, when both TigerS and TigerC is run on your laptop or PC? Justify your answer.

Additional Remarks

- Clearly articulate any assumption you made for this assignment, if necessary (HINT: You most likely will).

- You MUST use C/C++ for your implementation. The CLI library provided works with both. Make sure you use Linux or MacOSX. ***I will NOT accept Windows Visual Studio projects.***

- You are expected to use the default socket classes. You are not allowed to use a higher level (advanced) framework or 3rd party socket libraries. I will not install 3rd party libraries since there are around 40 students in this class.

- For the server, you may create a file to store the authorized users and passwords. Also, you may use default user directories on TigerS (and TigerC) machines, for the tget and tput commands. Ensure that your server and client use separate directories to avoid unnecessary complications and errors.

- A Command Line Parsing library is already implemented and available in the example client folder posted with this project. Examine the folder and header files to ensure you are aware of what needs to be implemented in your client.

- There is no example server folder. You may use the same Makefile and folder structure from the client. You may need to modify the Makefile if you chose to use pthreads for concurrency. You do not need the CLI library for the server.

Grading

The project will be graded as follows. Note that the sub-components under the Server and Client requirements are broken down into points per sub-topic, as some features are easier to implement than others.

Item	**Weight**	
Well defined Command and Control Protocol	**10**	
Server functions	**30**	
- Supports user login and logout (connect and disconnect)	*4*	
- Supports error free PUT and GET commands	*8*	
- Supports error free transfer of binary and ASCII files	*4*	
- Server supports (up to 100) clients simultaneously	*10*	
- Server supports unique directory traversal for clients	*4*	
Client functions	**25**	
- Supports user login and logout	*5*	
- Supports error free PUT and GET commands	*10*	
- Supports Directory Listing and Directory Changes	*10*	
Sockets detection and error handling	**10**	
Report as defined in the assignment	**25**	
TOTAL	**100**	

Supporting Material

In order to be successful with this project, several topics must be researched, and their underlying concepts understood. This can be done at the student's leisure, but the time constraint of the project should be kept in mind. Leave enough time for implementation and testing, as use of a new API is a learning experience and should be treated as such.

Requirements Development

This exercise requires you to create a custom concurrent FTP Server and associated Client application. Furthermore, it needs to support five basic commands. Immediately, it should be evident that two sets of requirements are needed, one for the server and the other for the Client. The two sets of requirements should be described using the identifiers SRV and CLN.

For the Server, we can immediately identify three basic requirements of the system. The Server is an IPv4 based service, using POSIX based sockets for portability; and thus, we can derive the following six system level requirements:

Identifier	Requirement	Req/D.O.
SRV-1	The Server shall utilize an Internet Protocol based socket for inter-process communications.	Requirement
SRV-2	The Server shall listen for incoming connection requests.	Requirement
SRV-3	The Server shall allow for at least one active client connection at any time.	Requirement
SRV-4	The Server shall provide for controlled access to the Server.	Requirement
SRV-5	The Server shall support file exchanges with the Client	Requirement
SRV-6	The Server shall support limited directory operational commands from the Client.	Requirement

The Client must compliment the Server, for which the following system level requirements can be derived:

Identifier	Requirement	Req/D.O.
CLN-1	The Client shall provide an ASCII based Command Line Interface to the user.	Requirement
CLN-2	The Client shall support an IP based connection to the Server.	Requirement
CLN-3	The Client shall support file exchanges with the Server.	Requirement
CLN-4	The Client shall support limited directory operations on the Server.	Requirement

Reviewing the five commands listed (tconnect, tget, tput, tlist, and tcwd), additional requirements can be described for both the Client and the Server. It is your job to complete this exercise in your design document. ***It is highly suggested that you complete this exercise before you start writing code.***

Command Line Interface Concepts

Requirement CLN-1 states that a CLI for the Client must be created. As review, CLIs parse three types of parameters: Arguments, Options and Flags. For this assignment, there is no need to parse anything but arguments. The CLI consists entirely of simple string parsing of arguments.

Be advised that a common issue when parsing command line arguments is the failure to use a POSIX compliant stream I/O function that can detect a newline delimiter. It is suggested that the student utilize fgetc(), getline(), or other POSIX compliant functions.

Since CLI parsing is not the focus of this project, the CLI is implemented in a C++ file called `processor.cpp`, and is included in the main C or C++ file via the `processing.h` file. The header file is designed to export the C++ library using the C calling conventions for those that chose to use C rather than C++. Please review the `main.cpp`, `processing.h`, and `readme.txt` files to understand how to leverage the lexer program. This will save you a lot of time and allow you to focus on the socket's programming, message protocols, and design document. The lexer is compiled using the re2c version 1.0.1, and does not need to be recreated. The readme provides instructions on how to build the lexer.cpp, but there should be no reason for you to do this.

The following Here String method of launching 100 clients should be utilized to verify program functionality. With small files, it should be noted that file transfers will occur very quickly, so to ensure concurrency of operation, files should be very large (100's of MB). Large file transfer will allow for overlap of operations on most current Operating Systems. ***The following bash script will be run by the instructor when validating programs after submission:***

```
#!/usr/bin/bash
n=1
last=100
while [ $n -lt $last]  do
  ./TigerC <<< "tconnect 127.0.0.1 user pass
  tget down$n.txt
  tput upload$n.txt
  exit" &   n=$(($n+1))
done
```

POSIX Compliance

To learn more about POSIX compliant functions for use in the application, refer to https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/. When a specific function or structure is required, search for the applicable term using the search in the upper left of the page and narrow in on the correct one.

For example, the sockets constructor can be found by searching for the keyword "socket". This search returns several pages, one of them pointing out the header at "sys/socket.h", and another the socket constructor function along with the arguments in its signature.

This type of research may be necessary to discover critical functionality for use in the project. Couple it with the example of sockets communication given below to achieve the functionality required.

Sockets Basics

- **socket** – Constructor takes params socket family (AF_INET), socket type (SOCK_STREAM) and a protocol descriptor. It returns an integer, which is either an integral socket ID or -1 indicating an invalid socket.
- **struct sockaddr** – Data members include sa_len a uint 8_t, sa_family_t sa_family and a 14 byte char array sa_data. Can be used to typecast specific sockaddr structures (_in, _in6) to more general structures, and memcpy is used to create the more specific structures from it.
- **struct sockaddr_in** – Data members include socket family (AF_INET), address (a 32-bit IPv4 address in network byte order) and port for the server (a 16-bit TCP or UDP port number network byte ordered).
- **bind** – Takes a socket ID, a pointer to a sockaddr struct and the size of the struct (using sizeof) and returns a 0 on success or -1 on error. Assigns a local protocol address to a socket.
- **listen** – Takes a socket ID, and a blacklog argument (queue of completed connections). Called by a TCP server, and converts an active unconnected socket into a passive socket, indicating the kernel should accept incoming connection requests directed to the socket.
- **accept** – Takes a socket ID, a pointer to a sockaddr struct, and the size of the struct. Returns a non-negative fd if OK, otherwise -1. Called by a TCP server to return the next completed connection from the front of the completed connection queue. If queue empty, process sleeps.

- **close** – The normal Unix close function is also used to close a socket and terminate a TCP connection. It takes a socket ID and returns a 0 on success or a -1 on error.
- **write** – Takes an fd, a buffer and max buffer size. Writes data to the socket from the buffer.
- **connect** – The connect function is used by a TCP client to establish a connection with a TCP server. It takes a socket ID, pointer to a sockaddr struct and size of the struct. It returns 0 on OK and -1 on error.
- **read** – Takes an fd, buffer address and max buffer size. Reads data from the socket and copies it to the buffer.

Concurrency Basics

Fork is the basic function used to spawn child processes from the main server process. This will create N number of child processes off the main, that will each serve a separate client. The function itself returns a process ID (pid). Pthreads are also an option, and for some systems requires additional linker command line arguments to include the pthread library.

The difference between an iterative and a concurrent server is that one will finish servicing the current client prior to taking the next client request. This is not true concurrency. See the myCourses C and C++ videos for a feel for how to use "fork" effectively to create a server that can handle the 100 clients that will be accessing it, but keep in mind that "fork" will only be a stepping stone to understanding true concurrency. The pthread library should be used in its stead to create lightweight child processes and manipulate them effectively.

Transport Layer

This was not specified in the project requirements at a lower level, so either TCP or UDP can be selected. Even though SRV-2 states that the server will listen for connection requests, this is a high-level requirement, and not explicit enough to map to a TCP or UDP implementation; however, authentication and reliable file transfer should guide the engineer to a specific transport protocol, and a method to associate authenticated clients with a session.

Error Correction

Error correction is a requirement throughout this exchange process, and it is essential to check every function call for an error return. POSIX compliant functions such as socket, inet_pton, connect, read, and fputs can cause the global "errno" variable to change, representing the type of error that occurred with a positive integer. The "errno" global is returned as the return value of the function in multithreaded applications, as opposed to being changed as a global value and accessed directly after a function call.

Client-Server Basics

A Client generally functions to accept and parse command line arguments, create a socket, specify the server's IP address and port, establish connection with the server, read and display the server's reply, and terminate the grogram.

A Server also creates a socket, but then binds the Server's well-known port to the socket, converts the socket to a listening socket, accepts the client connection, sending a reply, and terminate the connection.

Sockets Helper Functions

- int inet_pton(int family, const char* strptr, void* addrptr);
  Converts from dotted-decimal representation of IPv4 address to 32-bit binary (in_addr).

- const char* inet_ntop(int family, const void* addrptr, char* strptr, size_t len);
  Converts from in_addr binary representation to dotted-decimal address notation.

- void bzero(void* dest, size_t nbytes);
  Sets the specified number of bytes to 0 in the dest.

- uint16_t htons(uint16_t host16bitvalue);
  Returns 16-bit value in network byte order

- uint32_t htonl(uint32_t host32bitvalue);
  Returns 32-bit value in network byte order

- uint16_t ntohs(uint16_t host16bitvalue);
  Returns 16-bit value in host byte order

- uint16_t ntohl(uint32_t host32bitvalue);
  Returns 32-bit value in host byte order