CPSC 527

ASSIGNMENT 1

VSFTP Protocol Implementation

(Due Monday Feb 6, 2006)

(*This assignment can be done individually or in teams of* ***two***.)

You are to write a program to implement a very simple file transfer protocol (VSFTP) system that will be used to fetch (Get) any kind of file from a (remote) host. You are to use the Berkeley Unix interprocess communication (IPC) system calls and library functions. Although in this assignment description, we talk about both service primitives: Put (send) and Get (fetch) to send and fetch a file to/from a remote host, respectively. You are only required to implement Get (in addition to Open, Ls and Close). A sample program for Put is also provided to make your assignment easier.

Your program will use the unreliable datagrams provided by the UDP protocol. This simple protocol provides a way for application programs to send data with a minimum of protocol processing overhead. The protocol does not guarantee delivery or duplicate protection. The maximum amount of data that can be sent in one UDP datagram is about 65,000 octets, but operating systems may support a smaller maximum size. (See the manual page for "setsockopt()". Determine this maximum size by experimenting. For more information on UDP, do:

```
man udp
```

You must write two programs: a Client and a Server.

The Client process take no argument. Once started it can accept one of the following commands service primitives) from the user:

1. **Open** rhost socket

   This is the first command issued by the user before any other commands.
   where rhost is the name of the remote host (i.e. Server)

and socket is port number that the server is listening to

For example, the command:

```
Open rhost 2017
```

would allow the Client process to establish a connection with the Server running on the remote host "rhost" that is listening to port 2017. Note that each server running on the same host must communicate on a different port number. Also note that the port number must be greater than 1023.

2. **Get** rfoofile foolfile

to fetch rfoofile on the remote host (the Server's side) and saves it in foofile in the local host (the Client's side)

3. **Put** foofile rfoofile (not required to implement)

to send foofile (from the Client host) to rfoofile on the remote host (the Server's side)

4. **Ls**

to list the current directory on the remote machine "rhost"
(Hint: use the system shell command to execute "ls" at the remote machine to store result to a temp file, get this temp file and do a system shell command to execute "more" at the local machine. Do "man system" for more info on this shell command.)

5. **Close**

to close the file tranfer session, i.e. to terminate the Client process.

The details of the protocol involved are given as follows.

Get

The Client accepting a Get command will try to fetch a file from the Server and then prompt for a further command from the user. The Client first sends the name of the file to be fetched to the Server and wait for a reply (use the "recvfrom()" or "recvmsg()" system calls). After receiving from the Client the first datagram which conveys the Get command (type=1) (other type, e.g. type=2 can be used to indicate other command, e.g. Put) which contains the filename, the Server first ensures that requested file does exist and is readable, then sends back a positive (or negative) response and starts the file transfer by sending datagrams using the "sendto()" or "sendmsg()" system calls. Each datagram exchanged between the Client and Server might contain C data structures that look like this:

```
struct file_transfer {
    unsigned int type           /* type=1 for Get; =2 for Put;
                               =0 for data   */
    unsigned int sequence_number;  /* Unique ID for this datagram. */
    unsigned int last_fragment;    /* If non-zero, this is the last one. */
    unsigned int data_length;      /* Bytes of data that follow. */
    ... length bytes of data ...
";
```

In this example, within each datagram there is a "header" followed by the actual data. The header here consists of a type, a sequence number, end-of-message indicator, and a byte count. Each datagram is assigned a "sequence_number" so that the client can tell if it has missed one. The "last_fragment" flag is used to indicate to the client whether this is the last fragment and is zero for all fragments except the last.

The Server program takes three command line arguments:

* the port number to use (i.e., the same port number that the
  Client will use).
* the maximum fragment size to use
* the delay between sending fragments (in seconds)

For example, the command:

```
Server 2017 1024 2
```

listens on port 2017, and sends requested files with a maximum fragment size 1024, including both the header and the data parts of the fragment, and it delays 2 seconds between sending fragments. Use the "sleep()" library routine to delay. If necessary, you can assume the maximum file size is 200,000 bytes.

The Server is started before the Client and waits (indefinitely) to receive from the Client the first datagram of type 1 (indicating the Get command) which contains the name of the file to be fetched. The Client will then receive a response from the Server. If the response is negative, it will print an appropriate error message (e.g. "The file does not exist or inaccessible!"). Otherwise, the Client upon receiving a positive response will print a message (e.g. "File transfer begins..."). Subsequently, it expects to receive the datagrams of the file requested and saves them in the specified local file. After receiving each datagram of the file transfer, however, the Client sets a timer to some time-out value. Use a time-out value of 10 seconds. If a datagram is not received during the time-out period, the Client prints a message and gives up. Likewise, it simply prints an error message and exits if it detects some other problem during the transfer (e.g., it finds it has missed a fragment). If the transfer is successful, a "Successful transfer" message is printed which includes the file size in bytes, the total file transfer time, and the mean transfer rate (i.e., the file size divided by the total time to receive the file starting from when the first datagram was received). In any case, the program should print the number of bytes of the file successfully received so you can see how far it got. See the documentation for the "select()" system call to see one way of handling the time-out timer.

<u>Put</u>

The Client accepting the Put command will send the file to the remote host (the Server's side) and prompt for further commands from the user. The first datagram sent by the Client is of type 2 (for Put command) and contains the filename to be used by the Server to save the file it receives from the Client. The Server first check if the file exist; it then send an appropriate reply to the Client (a negative reply if the file exists). Receiving a negative reply from the Server, the Client will print an appropriate message; otherwise receiving a positive reply, it will start the file transfer, in the same way as in the Get case, but the data transfer takes place in the opposite direction, from the Client to the Server. The maximum fragment size and the delay between sending fragments can be read from a profile file (or alternatively entered as arguments of the Put command). Similarly, these parameters associated with the Server process call could have been obtained from the first datagram of type 1 (Get command) sent by the Client which previously read them from the profile file.

After you have written and debugged your programs, you will run a factorial experiment. Choose any file you want to transfer, but it must be between 20,000 and 200,000 bytes in size. Use the same file for each run and note the outcome of the file transfer. Run each experiment 2 times:
once using the Get command and once using the Put command.

- *Factor1*: Run with the Client and Server on the same computer and with them on different computers.

- *Factor2*: Run the Server with maximum fragment sizes of 1000, 10000, and the maximum supported datagram size.

- *Factor3*: Run the Server with delays of 0, 2, and 10 seconds.

Since Factor1 has 2 levels, Factor2 has 3 levels, and Factor3 has 3 levels, this makes a total of 2 * 3 * 3 = 18 configurations. With 2 runs of each configuration, this makes a total of 36 runs. You might find it helpful to write a simple shell script to run the client.

Answer the following questions:

1. Summarize the results of your factorial experiment in tabular form, indicating for each configuration what the mean transfer rate was (report zero if the transfer was unsuccessful). How big was your test file? What is the maximum datagram size supported by the system?

2. What are the conclusions of your experiment?

3. Briefly describe the weaknesses of this file transfer system and suggest some ways of improving it.

You must hand in the answers to this assignment as follows (in addition to a hard copy of your code listings and your answers):

1. Use "handin" to submit your answers. Do "man handin" to get more information on handin. The course account is cs527, and the assignment directory is a1.

2. The source for the Server program must be in a file called server.c.

3. The source for the Client program must be in a file called client.c.

4. You may have some number of header files (i.e., .h files) used by client.c or server.c.

5. Create a makefile called "Makefile" in the directory. Running "make" with no arguments will create both receiver and sender. Running "make clean" will remove all binary files, object files, core dumps, etc. from the directory.

6. Please do not put any other files in the directory.

<u>HINTS:</u>

Your **Server** program will probably have a segment of code that looks something like this:

```
   int s;
   struct sockaddr_in sin;

   ...

   if ((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
      perror("socket");
      exit(1);
   }
   sin.sin_family = AF_INET;
   sin.sin_port = server_port_number;
   sin.sin_addr.s_addr = INADDR_ANY;
   if (bind(s, (struct sockaddr *) &sin, sizeof(sin)) < 0) {
      perror("bind");
      exit(1);
   }
   ...
```

Your **Client** program will probably have a segment of code that looks something like this:

```
   int s;
   struct hostent *hp;
   struct sockaddr_in serv_addr;
   ...
  /* Read_Next_Command     */
  /* If Command = Connect  */
```

```
if ((hp = gethostbyname(remote)) == NULL) {
    fprintf(stderr, "Unknown host: %s\n", remote);
    exit(1) /* or go back to  Read_Next_Command */;
}
bcopy(hp->h_addr, (char *) &serv_addr.sin_addr, hp->h_length);
serv_addr.sin_family = hp->h_addrtype;
serv_addr.sin_port = server_port_number;

if ((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
    perror("socket");
    exit(1);
}
...
```