Spring 2009 CSCI 551

Warm-up Project #1

Client/Server

Due 11:45PM 1/30/2009

[BC: added 1/17/2009] *(Please check out the FAQ before sending your questions to the TA or the instructor.)*

Assignment

You will write two programs, a **server** and a **client**. The server creates a socket in the Internet domain bound to **`port`** specified in the commandline when the server was started, receives requests from it, acts on the requests, and returns the results to the requester. You must **not** fragment the response message into packets or multiple messages. It should be sent in one message. Therefore, the words *packet* and *message* are interchangeable below.

For this simple exercise, there are only three kinds of requests:

- an ADDR request
- a FILESIZE request
- a GET request

All messages have the same packet structure: a 2-byte (short integer) type field, a 4-byte (integer) data length field and a variable-length data field.

Byte Pos	**Name**	**Description**
0-1	MessageType	The message type (in network byte order). `0xFE10 (ADR_REQ)`: ADDR request `0xFE20 (FSZ_REQ)`: FILESIZE request `0xFE30 (GET_REQ)`: GET request `0xFE11 (ADR_RPLY)`: succeeded ADDR

		response `0xFE21 (FSZ_RPLY)`: succeeded FILESIZE response `0xFE31 (GET_RPLY)`: succeeded GET response `0xFE12 (ADR_FAIL)`: failed ADDR response `0xFE22 (FSZ_FAIL)`: failed FILESIZE response `0xFE32 (GET_FAIL)`: failed GET response `0xFCFE (ALL_FAIL)`: catch-all failure response
2-5	Offset	Zero or file offset (unsigned) for a GET request (in network byte order).
6-9	DataLength	Length of the data field below (in network byte order).
10+	Data	Data byte stream.

Please note that all integers must be sent in *network byte order*.

Ports are a shared resource (meaning each person must have their own ports to use). **You should only use the ports assigned to you.** To obtain port assignments, please follow the instructions. Please note that your code **must** attempt to use the port number given in the commandline and **not** check if it is in the range assigned to you. This is because during grading, the grader will use a port number reserved for the grader.

Requests & Responses

[BC: Paragraph added 1/24/2009]
Please note that for FILESIZE and GET, the **file** in question can also be a **directory**. You can call `stat()` (or equivalent) to get the filesize of a file or a directory.

ADDR:

When the server receives an ADDR request message, it should construct a response packet and send it back to the requester immediately. The ADDR response packet has the same structure as the ADDR request packet. The type field of the ADDR response should be set to `ADR_RPLY` and the data field should be an *ASCII string* (**not** null-terminated) containing the IP Address (in dotted-decimal format) of the host whose name was specified in the data field for the ADR_REQ packet. (You can use `gethostbyname()`, `inet_addr()`, and `inet_ntoa()` to get the IP address of a host.) The value of the data length field must be the same as the length of the string (as returned by the `strlen()` function). Some hosts may have more than one address. In such cases, just return any one of them. If the host does not exist, `ADR_FAIL` packet should be sent instead with no data (data length equals zero).

FILESIZE:

To satisfy the FILESIZE request (type field is set to `FSZ_REQ`), the server will again construct a response packet and send it back to the requester immediately. The type field of the FILESIZE response should be set to `FSZ_RPLY`, the data field should be an *ASCII string* (**not** null-terminated) containing the size of the file whose name was specified in the data field of the `FSZ_REQ` packet. (You can use `sprintf()` to format an integer in a string buffer.) If the file does not exist, `FSZ_FAIL` packet should be sent instead with no data (data length equals zero). Note that the filename may be a relative or fully qualified pathname, so you should not manipulate the given filename in any way. If a relative pathname is given, it is taken to start in the working directory of the server.

GET:

To satisfy the GET request (type field is set to `GET_REQ`), the server will again construct a response packet and send it back to the requester immediately. The type field of the GET response should be set to `GET_RPLY`, the data field should be **the file itself**, starting at **byte offset** specified in bytes 2-5 of the `GET_REQ` message (transmitted in cleartext, i.e., without any encoding and without any additional information). If the file does not exist, `GET_FAIL` packet should be sent instead with no data (data length equals zero). If the Offset

> specified in the `GET_REQ` is greater or equal to the requested file size, `GET_RPLY` packet should be sent with no data (data length equals zero). Note that the filename may be a relative or fully qualified pathname, so you should not manipulate the given filename in any way. If a relative pathname is given, it is taken to start in the working directory of the server.

If the server cannot recognize the request or if the request is malformed, the server should send a `ALL_FAIL` response packet with no data (data length equals zero).

For both request and response messages, if the data is a *string*, it must **not** be null-terminated. This means that the value of the data length field must be the same as the length of the string (as returned by the `strlen()` function).

Although bytes 2-5 of a message is only meaningful for a `GET_REQ` message, it should be set to zero for all other message types although they are not checked (i.e., if it is not zero, you should not treat it as an error).

You should find a convenient way to define the necessary packet types. Remember that all integers must be sent in *network byte order*.

Commandline Syntax & Program Output

The commandline syntax for the **server** and **client** is given below. The syntax is:

```
server [-t seconds] [-d delay] [-m] port

client {adr|fsz|get} [-o offset] [-m] hostname:port string
```

[BC: Moved and updated 1/14/2009]
Square bracketed items are optional. You must follow the UNIX convention that **commandline options** can come in any order. (Note: a **commandline option** is a commandline argument that begins with a **-** character in a commandline syntax specification.)

The **`server`** program can take three optional commandline arguments but **`port`** (port number) is required. If **`-t`** is specified, **`seconds`** is the number of seconds it takes for the server to auto-shutdown. If **`-t`** is not specified, your server should auto-shutdown 60 seconds after it starts. If **`-d`** is specified, **`delay`** is the number of seconds the server must wait before sending a response to any request. If **`-m`** is specified, upon receiving a message from a client, the server should print information about the request message to `stdout`.

The **`client`** program takes three to five commandline arguments: the request type flag (**`adr`** for an ADDR request, **`fsz`** for a FILESIZE request, or **`get`** for a GET request); an optional **`offset`** if **`get`** was specified previously; the name of the host where your server is running; and a **`string`**. In the ADDR request case, the **`string`** is the name of the host (e.g., www.cs.usc.edu) for which the Internet address is to be found. In the FILESIZE request case, it is the name of a file on the server for which you would like the size. In the GET request case, it is the name of a file on the server for which you would like to receive. If **`-o`** is not specified for a GET request, an offset of zero should be assumed. For ADDR and FILESIZE requests, if **`-o`** is specified, you should print an error and not send a request to the server. If **`-m`** is specified, upon receiving a message from the server, the client should print information about the reply message to `stdout`. For all cases, **string** should be copied into the **data** field of a request message.

If **`-m`** is specified when the server starts, the following information should be printed to `stdout` when a message is received from a client (please format the output lines exactly as shown here; please also note that for the client, you should insert a <TAB> in front of every line):

```
Received X bytes from Y1.Y2.Y3.Y4.
  MessageType: 0x????
       Offset: 0x????????
   DataLength: 0x????????
```

where **`X`** is the total number of bytes of data received from the client, **`Y1.Y2.Y3.Y4`** is the IP address of the client obtained by calling `getpeername()` and `inet_ntoa()` on the socket (could simply be 127.0.0.1), and the message field values are printed in hexadecimal format. If the message is less than 10 bytes long, please only print the first line. Please note that you do **not** need a **mutex** for writing to `stdout` for this project (since pthread is not required). So, if you get more than one client connecting to the server simultaneously, the output for one message may be interleaved by other outputs. This is fine for this project.

The requests should have the packet structure described above. The client sends the request to the server, waits for a response, and prints the result. All results should be indented by a single <TAB> character ('\t').

- If it is an ADDR response, you should simply print the string echoed. For example,

```
client adr -m nunki.usc.edu:6001 www.cs.usc.edu
<TAB>Received 23 bytes from Y1.Y2.Y3.Y4.
<TAB>  MessageType: 0xfe11
<TAB>       Offset: 0x00000000
<TAB>   DataLength: 0x0000000d
<TAB>ADDR = 128.125.3.227

client adr -m nunki.usc.edu:6001 www.cs.usc.xxx
<TAB>Received 10 bytes from Y1.Y2.Y3.Y4.
<TAB>  MessageType: 0xfe12
<TAB>       Offset: 0x00000000
<TAB>   DataLength: 0x00000000
<TAB>ADDR request for 'www.cs.usc.xxx' failed.
```

- For the FILESIZE response packet, print the file size (or an error message if the file doesn't exist).

```
client fsz -m nunki.usc.edu:6001 /etc/passwd
<TAB>Received 13 bytes from Y1.Y2.Y3.Y4.
<TAB>  MessageType: 0xfe21
<TAB>       Offset: 0x00000000
<TAB>   DataLength: 0x00000003
<TAB>FILESIZE = 801

client fsz -m nunki.usc.edu:6001 /home/scf-22/csci551b/hello
<TAB>Received 10 bytes from Y1.Y2.Y3.Y4.
<TAB>  MessageType: 0xfe22
<TAB>       Offset: 0x00000000
<TAB>   DataLength: 0x00000000
<TAB>FILESIZE request for '/home/scf-22/csci551b/hello' failed.
```

- For the GET response packet, print the number of bytes received and an MD5 checksum of the file or partial file received (or an error message if the file doesn't exist).

```
client get -m nunki.usc.edu:6001 /etc/hosts.equiv
<TAB>Received 24 bytes from Y1.Y2.Y3.Y4.
<TAB>  MessageType: 0xfe31
<TAB>       Offset: 0x00000000
<TAB>   DataLength: 0x0000000e
<TAB>FILESIZE = 14, MD5 = 42ccd1847b0a219402cd398179cbc5fc

client get -m nunki.usc.edu:6001 /home/scf-22/csci551b/hello
<TAB>Received 10 bytes from Y1.Y2.Y3.Y4.
```

```
<TAB>  MessageType: 0xfe32
<TAB>       Offset: 0x00000000
<TAB>   DataLength: 0x00000000
<TAB>GET request for '/home/scf-22/csci551b/hello' failed.

client get -m nunki.usc.edu:6001 /bin/less
<TAB>Received 104918 bytes from Y1.Y2.Y3.Y4.
<TAB>  MessageType: 0xfe31
<TAB>       Offset: 0x00000000
<TAB>   DataLength: 0x000199cc
<TAB>FILESIZE = 104908, MD5 = f27df2e0c79d4ab69eb37dff2ac0a92f

client get -m -o 123 nunki.usc.edu:6001 /bin/less
<TAB>Received 104795 bytes from Y1.Y2.Y3.Y4.
<TAB>  MessageType: 0xfe31
<TAB>       Offset: 0x00000000
<TAB>   DataLength: 0x00019951
<TAB>FILESIZE = 104785, MD5 = eccfd764fb33b3bc2001e344d474a507
```

When you receive the file content in a GET response message, you should compute the `MD5` checksum of the file on-the-fly. (Do **not** write out the content of the file to the filesystem and call an external program to compute the `MD5` checksum. Also, do **not** send the value of the MD5 checksum in the GET response message since the spec says that you are suppose to only send the file content.)

Information about the MD5 algorithm can be found at http://www.openssl.org/docs/crypto/md5.html. You should use **`MD5_Init()`**, **`MD5_Update()`** and **`MD5_Final()`** to compute `MD5` checksums on-the-fly.

If you cannot get the correct MD5 checksum, chances are that you are not reading, writing, or sending the correct data bytes. You can use a hexdump program, provided in `~csci551b/bin` on nunki, to see if you are getting the correct data bytes. Try running the following command on nunki:

```
~csci551b/bin/hexdump /bin/less | more
```

Any error conditions whose handling methods have not been explicitly specified here, please think about what would make sense and handle them appropriately and print out reasonable and useful error messages.

Compiling and Linking

Please use a single `Makefile` so that when the grader simply enters:

```
make client
```

an executable named **client** is created. If the grader simply enters:

```
make server
```

an executable named **server** is created. Please make sure that your submission conforms to other general compilation requirements and README requirements.

To figure out what linker flags you should use, please look at the top of the man pages for `socket()` by doing the following on nunki.usc.edu:

```
man -s 3socket socket
```

To use openssl on nunki.usc.edu, please see the additional notes on openssl.

Grading Guidelines

The grading guidelines has been made available. Please run the scripts in the guidelines on `nunki.usc.edu`. It is possible that there are bugs in the guidelines. If you find bugs, please let the instructor know as soon as possible.

Please do **not** use the port numbers in the grading guidelines. Please only use ports that are assigned to you. Your code is suppose to work with any port numbers above 1024.

Miscellaneous Requirements & Hints

- Please read the programming FAQ if you need a refresher on C/C++ file I/O and bit/byte manipulications.
- If you are new to sockets programming, start writing and testing your code as soon as you can! Here are some resources that may be useful.
- You are required to use *separate compilation* to compile your source code. You must divide your source code into separate source files in a logical way. You also must **not** put the bulk of your code in header files!

- To appreciate the *stream abstraction* of TCP, when you write to a socket, you are **required** to write only one byte at a time. This means that if you call `send()` or `write()` with the first argument being a socket descriptor, the 3rd argument **must be `1`**. Similarly, when you read from a socket, you are **required** to read only one byte at a time. This means that if you call `recv()` or `read()` with the first argument being a socket descriptor, the 3rd argument **must be `1`**.

- In responding to a `GET_REQ`, if the file is large, you may **not** read the whole file into a memory buffer and write the whole buffer out. Instead, use a small buffer and fill the buffer with data from the file and write out the content of the buffer before you read another block of data into the same buffer. **The maximum size of a memory buffer is limited to 512 bytes.**

  You should shutdown *gracefully*. This means the following after your main server process/thread is asked to quit:

  1. It should refuse to process any new request.
  2. If a request is being processed by a child process/thread, your main server process/thread must notify the child so that the child can close its socket and terminate *as soon as possible*.
  3. Your main server process/thread must wait for all the child processes/threads to terminate before itself exits. This implies that your main server must keep track of the process IDs of all the child processes or the thread IDs of all the child threads. Please note that if you are using `fork()`, you can send a SIGUSR1 or a SIGUSR2 signal to ask a child process to quit. If you are using threads, you can just set a global variable and all the child threads can check this variable.

- Your must not do busy-wait! If you run `"top"` from the commandline and you see that your program is taking up one of the top spots in CPU percentages and show high percentages (more than 1%), this is considered **busy-wait**. You will lose a lot of points if your program does busy-waiting.

  It's quite easy to find the offending code. If you run your program from the debugger, wait a few seconds, then type

<Cntrl+C>. Most likely, your program will break inside your busy-waiting loop! An easy fix is to call `select()` to sleep for 100 millisecond before you loop again.

- You should be aware that the packet structure requires that an integer is 4 bytes long. For some architectures, this is not true. Keep this in mind, but your programs are not required to support this.
- Use `netstat(1)` to see what sockets have been created by your programs (you may have put them in the background; once a program terminates, the sockets it created go away).
- You can use `stat(1)` or equivalent to get the filesize of a file.
- Clean up! If you are using multiple processes you should not leave zombie processes behind. Use `ps(1)` to see what processes you have left lying around and take care of them in your code.
- The Solaris workstations in the ISD lab in SAL have the same setup as nunki.usc.edu. So, if you are logged on to one of these workstations, please do your development locally and not to overload nunki unnecessarily.

Here are some programming hints:

- If you do not know what ports to use for your projects, please follow the procedure to request a set of port numbers. If it has been over a day since you sent your e-mail but have not receive the port assignments, it is possible that you have exceeded disk quota for your account, or the return e-mail address is incorrect. In this case, please send the request again.
- Let's say you got your server running and you use `fork()` to create child processes to handle the messages. Run the following command:

  ```
  ps x
  ```

  You may see something like the following:

  ```
   PID TT       S  TIME COMMAND
  7094 pts/171  S  0:00 -tcsh
  8018 pts/171  S  0:00 ./server
  8035          Z  0:00
  ```

```
8066         Z  0:00
```

The last 2 processes are zombies. This probably means that when a child process of yours terminates, it's sitting in `exit()` waiting for the parent process to handle `SIGCHLD`. You may try having the parent ignore `SIGCHLD` by calling:

```
signal(SIGCHLD, SIG_IGN);
```

at the appropriate place and see if this gets rid of the zombies problem.

If you want to handle SIGCHLD in your code, some helpful hints can be found at the GNU C Library site and also here (although I'm not sure if this is good for Linux systems only).

- Let's say that you want to kill your server process and restart it immediately. But your call to `bind()` fails. Try the following and see if it helps:

```
int my_socket=0, reuse_addr=1;

... [create my_socket]...

if (setsockopt(my_socket, SOL_SOCKET, SO_REUSEADDR,
        (void*)(&reuse_addr), sizeof(int)) == -1) {
    /* report error */
}
```

- To copy from a buffer to an unsigned short integer, try:

```
char *buf;
unsigned short us;

memcpy(&us, buf, sizeof(unsigned short));
```

- Do not use `sprintf/snprintf()` to put a pattern into a piece of memory because `sprintf()` is suppose to output ASCII! If you want to put `0xFE` into the byte 3 of `buf`, do not do:

```
snprintf(&buf[3], sizeof(buf)-3, "%2x", (int)(0x0fe))
```

You will end up with the *character* `'f'` in `buf[3]`, the *character* `'e'` in `buf[4]`, and '\0' in `buf[5]`. This happens because `sprintf()` is doing exactly what you asked it to do. Try

```
buf[3] = (char)(0xfe)
```

When in doubt, get into the debugger and print the value of `buf[3]` ane make sure you are getting what you expected.

- If you are having trouble with `connect()`, here's some information that may be helpful.
- If you are having trouble with figuring out how to set a timer to shutdown the server or for the server to break out of the `accept()` call, here's some information that may be helpful.
- You are encouraged to use *threads* for this project. Although there is *no* penalty for using `fork()`.
- Here is something to think about when you implement your server... If a malicious client connects to your server and specify a Data Length of 1 billion, what should you do? It's not exactly in the spec! What reasonable design decision should you make? Whatever you come up with, make sure you document that in the `README` file in your submission. (What are other things a malicious client can do? Please document how your server handles them.)
- If you have `.nfs*` files you cannot remove, please see notes on `.nfs` files.

A lot of hints have been given above. Please do not ask the TA or the instructor what these functions do and where to put them unless you have tried hard to get them to work and faied. You should always read the man pages, search the web, and try things out first.
