Programming Assignment 1 - `richftp`

Due Date — March 6, 2008
An extension until March 8, 2008 is granted to all who may need it, however, note that the next project may possibly be assigned prior to March 8.

This project is worth 10% of the final grade.
You may perform this project in a team of up to 2 students.

How does this project fit into the Big Picture?

As the first stage in building a peer-to-peer application, this assignment requires you to write code for file transfer between two hosts. In particular, you will implement a version of ftp called **richftp** that supports **concurrent downloads and uploads of files across any pair of hosts**. By the end of this project, you will have obtained experience of the following programming primitives/libraries.

- Network Sockets.
- Multi-threaded programming.
- Thread Synchronization.

What Do I Implement?

`richftp` will link into a searching capability later on in the semester. The ftp functionality is implemented in the **FTP layer**, and the other application functionalities are implemented in the **Application layer** (later, we will build the searching capability in here). The two layers are connected by a well-known interface. In other words, the application layer calls functions that are implemented by the FTP layer, and vice-versa.

The functions in the interface are specified below. A sample implementation of the Application layer is provided to you. **Your task in this project is to implement the functions in the FTP layer only.** *You may modify the Application layer implementation provided to you, but your implementation of the FTP layer must run correctly even if the application layer functions have different implementations.* In other words, we will replace the functions of the Application layer with our implementations for automatically testing your implementation of the FTP layer.

The FTP layer provides four functions to the Application layer - `richftp_put`, `richftp_get`, `richftp_abortftp`, and `richftp_init`. The Application layer implements a function `app_result` that is called by the FTP layer for nonblocking requests. We describe these in detail now.

FTP layer

The functions you will implement are defined in the file <`richftp.h`>. You may write your code into the templates in file `richftp.c`.

1. `ResultType richftp_put(const char *hostname, const char *fname, char *buff, int bufflen)`: Called by the application layer to transfer a file to a remote host. The parameters are : (i) the IP address of the remote host `hostname` (a string of the form "126.0.4.32"); (ii) `fname`, the name of the file to be transferred to the remote host. The remaining parameters are: (iii-iv) a character buffer `buff` containing all the `bufflen` bytes that are the contents of the file to be transferred. The application layer must not modify the contents of this buffer (or free the space allocated to it) until the request is complete.

   This function should be implemented only as a nonblocking call. For a blocking call, a request is complete when the above function returns. A nonblocking call, on the other hand, returns immediately with an integer that specifies the unique local identifier `requestid` for this particular request (the return type `ResultType` is a `typedef int`). It should create a separate thread to attempt to finish the request within `TIMEOUT` seconds, and whichever happens earlier, leads to an `app_result` upcall. The request completes when either `app_result` or `richftp_abortftp` have been called. It may be easier to implement the function as a blocking call first, and then modify it to a nonblocking call.

   Your code will have to run appropriate checks for errors (e.g., hostname is in the required format).

2. `ResultType richftp_get(const char *hostname, const char *fname, char *buff, int bufflen)`: Called by the application layer to transfer a file from a remote host. The parameters are (i-ii) the same as in `richftp_put()` above; (iii-iv) a buffer `buff` allocated at least `bufflen` bytes of space in the application layer before the call. The application layer must not modify or free this buffer until the request completes (same as in `richftp_put`).

   It should be implemented as a nonblocking call. A nonblocking call returns immediately with an integer that specifies the unique local identifier `requestid` for this particular request (the return type `ResultType` is a `typedef int`). It should create a thread to attempt to finish the request within `TIMEOUT` seconds, and whichever happens earlier, lead to an `app_result` upcall. The application layer must not modify the contents of this buffer (or free the space allocated to it) until the request is complete. The request completes when either `app_result` or `richftp_abortftp` have been called. It may be easier to implement the function as a blocking call first, and then modify it to a nonblocking call.

   Your code will have to run appropriate checks for errors (e.g., hostname is in the required format).

3. `ResultType richftp_abortftp(int requestid)`: This function is called by the application layer with the previously returned `requestid`, obtained from a call to `richftp_get` or `richftp_put`. This function aborts the request with the id, cleans up the necessary thread and data structures, and returns the number of bytes transferred so far (to the remote host if this was for a `richftp_put` request, and from the remote host if this was for a

`richftp_get` call). If the `requestid` is invalid, a negative integer specifying an error code of `NOSUCHREQUEST` is returned (see <`errorcodes.h`>).

4. `int richftp_init(void)`: This function is called by the application soon after your program starts up. This initializes the needed data structures that are maintained inside the FTP layer, creates the server daemon thread, and initializes mutex/synchronization variables. There are no parameters. The function returns -1 if the initialization fails, and 0 if it succeeds.

With respect to error checking, your code is in general responsible for catching any errors that are not mentioned above as being a responsibility of the application.

Application layer

Implementations of the following functions are provided to you. You may modify these implementations as you wish, but your FTP layer code (above) must run even if we replace these with our own implementations. Sample implementations are provided in the file `app.c`.

- `void app_result(int requestid, ResultType res, char *buff, int bufflen)`: This is an **upcall function** implemented in the application layer, and **is called by the FTP layer**. It is called when a (nonblocking) `richftp_get()` or `richftp_put()` call completes successfully. This function is called exactly once for each request (i.e., for each valid `requestid`) that was not subjected to an `richftp_abortftp()` call. It should not be called for requests for which `richftp_abortftp` was called and returned. For requests that are currently being aborted, the behavior is undefined.

  The parameters in this call are (i) `requestid`, the id of the request for which the result is being upcalled; (ii) `res` that specifies the result being returned - this could either be a positive integer specifying the number of bytes transferred so far (to the remote host if this was a put request, or from the remote host if this was for a get request), or a negative integer that specifies one of the errors defined in <`errorcodes.h`> - `HOSTNOTFOUND`, `FILENOTFOUND`, `XFERTIMEOUT`, `FILEXFERABORT`. The other parameters are : (iii-iv) `buff`, a buffer of length `bufflen` containing the characters in the file that were transferred (for both `richftp_get` and `richftp_put` requests). **Notice that the values of pointer `buff` and `bufflen` here are the same values that were passed to the original `richftp_get` or `richftp_put` call that created this requestid.**

- `int main(void)`: This function in the application layer first calls `richftp_init()` before any of the other `richftp_*` functions. Then, it runs into an infinite loop where it makes the necessary calls to the FTP layer.

You can write your code into the file `richftp.c`. The file <`errorcodes.h`> contains a list of errorcodes that `richftp_get()` and `richftp_put()` return pass back to the `app_result()` callback function. The file `app.c` contains a sample `main()` function.

Please do not change the structure of the code provided to you, e.g., do not delete any of the errorcodes already provided, do not modify the interface definitions, do not rename the files

already provided to you. However, you may add on more errorcodes, more functions (e.g., inside `richftp.c`), and more files.

Also, feel free to modify `main()` so that you can test your code appropriately.

A sample Makefile and README are provided for you. Use the Makefile while compiling your code. The purpose of a README file is to describe to the reader the purpose and design of your implementation, as well as summarized details about the .c and .h files. When your project is complete, record the details of your implementation in the README file.

What Is My Implementation Supposed to Do?

To support concurrent file transfers in a correct and consistent fashion, your implementation will need to access three kinds of libraries: (i) synchronization, (ii) network sockets, and (iii) multithreading. Our suggestions for good libraries that you can use are given in the next section, but first let us find out why we need these three.

Data Structure for Concurrent Accesses: Create a data structure to hold information about outstanding requests (nonblocking gets and puts currently being processed). This could be an array, a linked list, or any other implementation that you choose. Access to this data structure is protected by a mutex or semaphore that needs to be locked before access, and unlocked afterwards (we will soon see why).

The code that you write has to implement functionalities on both the Client Side and the Server Side. We describe each of these in turn, and then describe common characteristics that your code needs to implement.

I. Client Side

richftp_put(): Suppose the application layer (read: `main()` function) makes a `richftp_put()` call.

Your implementation of `richftp_put()` does some things before returning : (i) it records details of the request into the data structure maintained for outstanding requests (and obtains the request id for this request), (ii) it creates a new thread T for this request. Then the function call returns with the appropriate `requestid`. Notice that step (i) will need to lock a mutex or a semaphore in order to ensure that two concurrent requests do not access the data structure simultaneously.

The thread T essentially does the same operations as a call to a blocking version of `richftp_put()` would do - open a socket connection to the remote host, transfer the file to the remote host, and return the result within `TIMEOUT` seconds. However, the result can no longer be returned to a calling function in the application layer (since this is a nonblocking call). So, an upcall is made to the application layer function `app_result()`, specifying the results of the request. See previous section for detailed specification of the parameters for this function call. Note that the `buff` and

`bufflen` parameters passed to the `app_result` function are the same that were passed down in the original `richftp_put` call.

richftp_get(): The steps for the function `richftp_get()` are similar to those described above for `richftp_put()`. The `buff` and `bufflen` parameters passed back to `app_result` are the same as were passed in the original call to `richftp_get`.

richftp_abortftp(): This function is called by the application layer to abort an ongoing ftp request in the FTP layer. Your implementation should access the data structure for outstanding request, check if the `requestid` is valid (if invalid, it returns an errorcode as specified in the last section), stop the thread that was created for this request, and clean up the entry in the data structure for this `requestid`. Then the call returns the number of bytes transferred successfully so far.

II. Server Side

Your program will have a persistent (long-running) server daemon thread (created by `richftp_init` - see below) that listens to connection requests on an incoming well-known port : you are free to choose a suitable port number. The server daemon thread spawns off one thread for each new request. These threads will have to read (for a `richftp_get` request on the client side) or write (for a `richftp_put` request on the client side) the appropriate file from the file system. You may assume that all files are located in the same directory as your implementation. All files are read and written into this directory. If a file with the same name already exists, you may overwrite it. We suggest you use the library functions `fopen, fprintf, fscanf, fclose`. Don't forget to close the file after you are done writing or reading it.

III. Everywhere

void richftp_init(void): This call is made immediately after `main()` starts. It initializes the data structure for outstanding requests, the mutex / semaphores used to access it, and other global variables that your `richftp` implementation maintains.

Most importantly, the call to `richftp_init(void)` creates a special thread called the server daemon. The server daemon thread runs forever, i.e., until the end of the application. The server daemon thread listens to incoming `richftp` requests on a well-known port (choose your own). When the server daemon thread receives a new request (e.g., from a remote host also running the `richftp` protocol), it accepts the incoming connection and creates a new thread to process the messages on the incoming connection. Such new threads terminate by themselves.

Synchronization: Besides access to the data structure for concurrent accesses, the upcall to the application layer function `app_result()` has to be controlled through a mutex so that only one thread can call it at a time. In the absence of such synchronization, `printf`'s from concurrent threads that are simultaneously calling `app_result()` may overlap on your console!

Deadlocks: When using synchronization objects such as mutexes or semaphores, you need to be careful that your code is not written in a way that it deadlocks!

Local File System: You may assume that the local file system at each host is flat. In other words, all files are present in the same directory as the `richftp` implementation. All fetched files are deposited here.

Freeing `malloc`ed data structures: All malloced data structures have to be freed. All buffers (parameter `buff`) created by the application layer that are passed into `richftp_put` or `richftp_get` calls are freed only in the application layer, and not until the request is completed (a `richftp_abortftp` or an `app_result` call).

Suggested Libraries: We suggest that you use the `pthreads` library (<`pthread.h`>) for multi-threading, and IPC semaphores (<`semaphore.h`>).

What Do I Turn in?

Please include comments in your code – at a minimum, for each procedure, include comments that specify what the procedure does, and explain the inputs and outputs. Your code should compile with the Makefile that you submit. **Your code will be tested automatically by replacing the entire file `app.c` with our implementation, i.e., different `main()` and `app_result()`**. So, you should **ensure before handing in that your final code compiles and runs with different implementations of `app.c`**. Write a concise README (less than two pages) about your design, and the effort put in by each team member.

When your project is complete, turn in all your files (`.c, .h, README, Makefile`) using Compass.

How can I maximize the points I can receive?

Please, make sure that your code works on the EWS machines before you turn it in. To do this, you need to test sufficiently before you turn your code in.

If there are special requirements for your code (e.g. C++ instead of C, lib??? required (except pthreads), ...), you must note these in your README file. If you do not make sufficient notes concerning your alterations, you will be penalized.

The underlying advice here is, if we can't get your code to run on our computers easily, or if your code doesn't pass our tests, you'll lose points. Our tests are thorough, so make sure your testing is thorough. Hence, documentation of extra requirements and thorough testing is highly advised.

If you need help, please make use of office hours and the Google group.

What Resources do I Have?

A. D. Marshall's online guide to C contains tutorials; some useful links from that tutorial:

Sockets: `http://www.cs.cf.ac.uk/Dave/C/node28.html#SECTION00280000000000000000`
Semaphores: `http://www.cs.cf.ac.uk/Dave/C/node26.html#SECTION00264000000000000000`
Pthreads: `http://www.cs.cf.ac.uk/Dave/C/node29.html#SECTION00294000000000000000`

Beej's online guide to network programming
(`http://beej.us/guide/bgnet/`) and W. R. Stevens' book on "Unix Network Programming" are the best for sockets programming. Pthreads tutorials are available at
`http://www.llnl.gov/computing/tutorials/pthreads/`. These guides have plenty of sample code. A gdb tutorial can be found at
`http://www.unknownroad.com/rtfm/gdbtut/gdbtoc.html`. Also see the handout for programming assignment 0.

The Unix `man` pages are an authoritative guide for all library functions. Please do not hesitate to ask questions in the course discussion board on compass, or during office hours.