CS 720/820 – Russell
Fall Semester, 2013
Assignment 3
Due: Monday, November 25, 2013

For this assignment, modify your recently improved version of “plcs” in order to be able to search remote files using TCP connections. First, fix all your bugs and clean up your code. Then split your existing code into two separate programs: one, entitled “rplcs”, will be the remote client, the other, “rplcsd”, the remote server.

The remote server, “rplcsd”, should run on any node and should do all its printing to standard error. It is run with up to 2 optional command line parameters (as in the class demo “servermain.c”): the first is the port number on which the server should listen; the second is the interface name on which the server should listen. The server starts by calling the “openlistener” function given in the class demo “tcpblockio.c”, which creates a socket, binds it to a TCP port number, and sets up a listening backlog. The server then goes into an infinite loop that advertises this listening socket to the world by printing its hostname and port number, and then accepts simultaneous SOCK_STREAM connections from multiple clients on various remote nodes. Once a connection is established, the server should print a “connected to client” message containing the fd number, client’s IP address, and client’s port number (as returned by the “accept”), then create a new “agent” thread operating at descent level 0 to service that connection. This new agent thread should immediately print a “starting agent” message containing the fd number it is servicing and its thread id. An agent thread will terminate only after finishing its search and exchanging all messages involved in the search, printing a “terminating agent” message containing the fd number it is servicing and its thread id, and finally closing its connection to the client.

The main server thread (i.e., the thread doing the listening), should be able to support any number of simultaneous agent threads, should run in parallel to all its agent threads (and any threads they create during their searches), and should run forever — it will have to be manually terminated.

Once a new connection is made, the client will send messages to the agent thread servicing that connection. These messages, in the format detailed below, will contain the search options, the search string, and the name of a file, directory, etc. to be searched. Once it has received all this information, the agent thread will start a search of the indicated file, directory, etc. and will send all the output, both error messages and normal output, back to the client, using messages in the format detailed below. When the search is finished, the agent thread will send a final message containing its search statistics, and then that agent thread will print its “terminating” message (as outlined above), close the connection to the client, and terminate itself. To do its searching, each of these agent threads uses the full value of both “-t” and “-d” options when spawning threads and descending into directories. Neither the server listening thread nor any agent thread created by it to service a connection counts against any thread limit set by either the “-t” or “-d” options — only threads created by each agent’s search itself count against this limit.

The remote client, “rplcs”, should run on any node and may direct simultaneous search requests to multiple servers on various nodes. The client processes command line options as in the previous version of “plcs”. Any errors in processing the options, or a missing search string following the options, will cause the client to terminate without processing the “list-of-file-names”, and therefore, without contacting any servers. When the client processes names in the optional “list-of-file-names”, it needs to differentiate between “local” names (those that refer to files, directories, etc. on the client’s machine) and “remote” names (those that refer to files, directories, etc. on a remote server’s machine). The client processes local names exactly as in the previous assignment (i.e., by searching files, spawning threads to process directories, detecting directory loops, etc.). The client connects to a server during the processing of the “list-of-file-names” each time it detects an item with the format:

```
node:service/remote_name
```

where there are no spaces between any of these components. In this format, “node” is either an IPv4 address in dotted-decimal notation or a DNS name of a remote server, “service” is the decimal port number on which to connect to that remote server, and “remote_name” is the name of a file, directory, etc. on that remote server. Both the colon (:) and the slash (/) are required punctuation, and are not themselves part of the “node”, “service,” or “remote_name” components. This punctuation can be used to syntactically identify a request for remote searching. If an item on the command line does not have this format, then the client should do a local search for that item in exactly the same way as was done in the previous assignment.

The following rules define “remote-name” in this assignment:

(1) if a colon (:) appears anywhere in an item from the “list-of-file-names” then the client interprets this item as a remote item. Everything before the colon in that item is the “node”, everything between the colon (:) and the next slash (/) is the “port”, and everything after the slash is the “remote_name”.

(2) The “node”, “service”, and “remote_name” components must not be empty.

(3) “.” and “..” in a remote_name are resolved on the server. For example, the remote_name “.” refers to the directory in which “rplcsd” was executed on the server machine.

(4) A remote_name of “-” is a filename on the server — it does **not** mean “read from stdin” on the server.

(5) A remote_name starting with a slash (/) is an absolute path name on the server. A remote_name starting with any other character is relative to the directory in which “rplcsd” was executed on the server.

(6) These syntactic conditions should be checked by the client, and if they are not met, it should give an appropriate error message and continue with the next item in “list-of-file-names” (without spawning a new client thread and without attempting to connect to the server).

When the client encounters a valid “node:service/remote_name” item while scanning the “list-of-file-names”, it should spawn a new client thread which will create a socket, make a SOCK_STREAM connection to the server on the remote node, then process all message exchanges with the remote agent created by that server. After spawning this new client thread, the main client thread continues in parallel to process remaining items in the “list-of-file-names”. None of these client threads are counted against the client’s “-t” limit (if any), and a client thread servicing a connection with an agent will terminate only after that connection terminates and all post-processing involving that connection is finished. Any errors encountered when creating a connection to a server should cause the client thread spawned for that connection to give an appropriate error message and terminate.

NOTE: the format `node:service/remote_name` should be checked for **only** in the “list-of-file-names” on the client’s command line. The client should **not** check for it in a name taken from a directory, (i.e., they are always processed as in your previous assignment). An agent **never** checks for it. Note: this is not to say that names in directories cannot be links to names in this format — they can! It is just that for this assignment we will not treat them as remote_names.

Also note that all options and option parameters are resolved and checked for correct syntax and validity by the client, not the server or its agents. Finally, if there are no remote names in the “list-of-file-names”, your client should not attempt to spawn any client threads, nor should it contact any servers. It should just process items in the “list-of-file-names” as in the previous assignment.

In order to make your client, server, and agent interoperable with all other clients, servers, and agents in the class, the protocol on the connection between client and agent must conform to the following specifications.

There are 6 different types of messages sent between the client and the agent:

(1) A message from client to agent containing option values as described below. This must be the first message the client sends on a new connection.

(2) A message from client to agent containing the search string. This must be the second message the client sends on a new connection.

(3) A message from client to agent containing the “remote_name” of the file, directory, etc. that the agent is to use in its search. Note that this message contains **only** the “remote_name” component of the “node:service/remote_name” item from the client’s command line. This must be the third message the client sends on a new connection.

(4) A message from agent to client containing one formatted output line resulting from the search.

(5) A message from agent to client containing one formatted error message line resulting from the search. Each line in a directory loop stack trace is also sent using this message type.

(6) A message from agent to client containing some search statistics. This must be the last message the agent sends on a connection because it also tells the client that the agent is finished.

All messages in both directions must have the “TLV” format, where each message starts with a header having the following format (each field is a 32-bit unsigned integer that must be transmitted in network byte order):

```
struct our_header {
     unsigned int message_type; /* type of this message */
     unsigned int var_length;   /* no. of bytes after header */
};
```

A message header is immediately followed by a variable length part, containing “var_length” bytes (if any).

The “message_type” is simply the number given in the list above for each message (1, 2, 3, 4, 5 or 6), and the “var_length” is the number of bytes in the variable part of the message (which can be 0).

When the message_type is 2, the variable part is the search string, including the nul-terminator at the end.

When the message_type is 3, the variable part is the remote_name string, including the ending nul-terminator.

When the message_type is 4, the variable part is the properly formatted output line produced by the search, including the new-line and nul-terminator at the end. Upon receiving a message_type 4 from the agent, the client should print this line as a single string (with no further formatting) to standard output prefaced (on the same line with no intervening spaces) by the “IPv4_address:port_number/” of the agent sending that line. (For an output line produced locally by a client search, this preface will be omitted.) Note that if the “-p” option is in effect, the agent should be producing output lines that start with the full path name of a file, so that when these lines are printed by a client they will contain 2 consecutive slashes (i.e., “//”) after the “port_number”.

When the message_type is 5, the variable part is the output error line produced by the search, including the new-line and nul-terminator at the end. An agent also uses message_type 5 to send the levels and names of directories in a directory loop detected by that agent, one message for each level and name in the loop that includes the required indentation. Upon receiving a message_type 5 from the agent, the client should print this line as a single string (with no further formatting) to its standard error, prefaced on that line (with no intervening spaces) by the “IPv4_address:port_number/” of the agent sending that line. (For an error line produced locally by a client search, this preface will be omitted.) The agent should also print on one line the variable part of every message_type 5, exactly as it is sent to the client, but prefaced by the “IPv4_address: port_number/” of the client.

For message_type 1, the variable part is the following 24-byte structure (each item in network byte order):

```
32-bit integer flags value
32-bit integer value of -d option
32-bit integer value of -l option
32-bit integer value of -m option
32-bit integer value of -n option
32-bit integer value of -t option
```

where the value of “flags” is the inclusive-OR of the following 8 bit-masks for the command line options that do not take parameters:

```
0x01 if -a option is present in command line
0x02 if -b option is present in command line
0x04 if -e option is present in command line
0x08 if -f option is present in command line
0x10 if -i option is present in command line
0x20 if -p option is present in command line
0x40 if -q option is present in command line
0x80 if -v option is present in command line
```

The value of an option that does take a command line parameter (-d, -l, -m, -n, -t) is the numeric value of that parameter if the option was present in the command line, and -1 otherwise. When the agent receives a message_type 1, it should print a one-line “client’s options” message containing the agent thread’s fd and thread id, followed by 13 lines, each containing 1 option value received from the client plus that option’s switch character in the order shown above, starting with the options in “flags”.

For message_type 6, the variable part is the following 56-byte structure (each item in network byte order):

```
unsigned integer value of Total soft links ignored due to -f
unsigned integer value of Total directories opened successfully
unsigned integer value of Total directory loops avoided
unsigned integer value of Total directory descents pruned by -d
unsigned integer value of Maximum directory descent depth
unsigned integer value of Total dot names not ignored due to -a
unsigned integer value of Total descent threads created
unsigned integer value of Total descent threads pruned by -t
unsigned integer value of Maximum simultaneously active descent threads
unsigned integer value of Total errors not printed due to -q
unsigned integer value of Total lines matched
unsigned integer value of Total lines read
unsigned integer value of Total files read
unsigned integer value of Total bytes read
```

where all 14 values are accumulated by that agent thread on that search. Upon receiving a message_type 6 from the agent, the client should incorporate its values into a corresponding table of cumulative totals being kept in the client for all searches, local and remote. (This is done by adding "Total" entries to a running total kept in the client, and by keeping the maximum of "Maximum" entries and a running maximum kept in the client.) This cumulative table is printed by the client to standard output **only** after all searches, local and remote, have been completed. (During debugging you may wish to print this cumulative table each time it is updated, and you may also wish to print each table as it is received from an agent.) Before sending a message_type 6 to the client, the agent should print a one-line "search statistics" message containing the agent thread's fd and thread id, followed by a statistics table (using the values sent to the client) in the format below.

All printed statistics tables should contain 16 lines (one per statistic), each line in the format:

```
label: value
```

where label is the string given in the left column in the following table, and value is the numeric value of the corresponding statistic. Please use the exact labels and print the statistics in the exact order shown in the following. Also, **please** align the columns so the result actually looks like a table!

label	statistic
Total soft links ignored due to -f	Number of soft links not followed
Total directories opened successfully	Number of directories processed
Total directory loops avoided	Number of directories ignored due to a loop
Total directory descents pruned by -d	Number of descents prevented by -d limit
Maximum directory descent depth	Maximum directory descent depth actually attained
Total dot names not ignored due to -a	Number of names starting with ´.´ processed
Total descent threads created	Number of descent threads created successfully
Total descent threads pruned by -t	Number of descent threads not created due to -t limit
Maximum simultaneously active descent threads	Maximum active descent threads at an instant
Total errors not printed due to -q	Number of errors not printed
Total lines matched	Number of lines matched in all files
Total lines read	Number of lines read (including each line from stdin)
Total files read	Number of files read (including each instance of stdin)
Total bytes read	Total number of bytes read from all files
Total search time in seconds	Total elapsed time taken to perform the search
Processing rate in megabytes per second	Number of megabytes read per second

The value of the "Total search time in seconds" should be printed as a number in the format:

```
seconds.microseconds
```

where 6 digits are always printed to the right of decimal point. On the client, compute this value as the elapsed time between the point in your client where you start to process items in "list-of-file-names" (or the client's standard input if "list-of-file-names" is empty) and the point in your client when you have terminated all local and remote searches and would start to print the final table of cumulative statistics. On the agent, compute this value as the elapsed time between the point where your agent thread for this connection starts running and the point where this thread has terminated all searching and would start to print the table of statistics before sending it to the client. This value is **not** sent from the agent to the client in a message_type 6.

The value of the "Processing rate in megabytes per second" should be printed as a number with 6 digits to the right of the decimal point. It is computed as 0 if the "Total search time in seconds" is 0. Otherwise it is (the value of "Total bytes read" divided by the value of "Total search time in seconds") divided by 1,000,000 bytes per megabyte. This value is **not** sent from the agent to the client in a message_type 6.

As a "log" of their activities, the "rplcsd" server and agents will print 7 types of lines to standard error on the server machine (all of these have been mentioned above, and are just summarized here):

(1) The "advertisement" of the listener's hostname and port number.

(2) The "connect" line when a new client connects to the listener.

(3) The "starting" line when a new agent thread starts.

(4) The "terminating" line when an existing agent thread finishes.

(5) The "client's options" line and subsequent options table when an agent receives a message_type 1.

(6) The variable part of every message_type 5 an agent sends.

(7) The "search statistics" line and subsequent statistics table when an agent sends a message_type 6, plus the 2 extra "Total search time" and "Processing rate" lines.

By following these message specifications exactly, your clients, servers, and agents should be interoperable with the clients, servers, and agents of all other students (as well as those of the instructor!).

Except for the additions and modifications given above, your new version of "rplcs" should be identical to the previous version of "plcs" from assignment 2. Because you are using threads and remote connections, the order in which files are processed may be different from that seen in the previous assignment when using the same options and files. In fact, the order could differ from run to run using just your new version, even when using the same options and files on each run. Furthermore, because of the thread and connection parallelism, it is possible (in fact, highly likely), that the output created by the client will interleave lines from many searches, local and remote. This is fine. However, you must ensure that this interleaving is done between full lines, not between components of lines. For example, if one output line is to have components in the format:

```
IPv4_address:port_number//realpath: line_number: text
```

then all those components must appear together on exactly one output line, and no components from other output lines should appear anywhere on this output line.

By this time, there should be huge sections of your code that are either not changed at all by these additions, or are changed in only a trivial way. If you find this is not the case, please think about why this is, and how your original design could have been better structured and why (even without the new features).

Using the gcc compiler, both programs should compile, link and run with no changes on agate.cs.unh.edu, newton.cs.unh.edu, and topaz.cs.unh.edu. You may copy (and modify) any files from the "˜cs720/demos" area, but please be sure to submit these files along with your own files, so that the submission is a complete package. Also, if you change a copy of a class demo, please put your name in the changed version you submit.

You must create a Makefile that can be used by gmake to build both your executables on all platforms. The executable for the client must be called "rplcs", and the executable for the server must be called "rplcsd".

You do **not** have to create or submit a "dotest" file for this assignment.

Submit all your source files (no object, executable, or "tar" files, please) and your Makefile by typing:

```
˜cs720/submit 3 file1 file2 file3 ...
```

where 3 is the number of this assignment. You MUST be on "agate.cs.unh.edu" to run submit! Please include your name in all your source files you create or modify, and in your Makefile.