Clustered Disk Space Allocation and Network Programming

Overview

- This project is due by 11:59:59 PM on Thursday, December 10, 2015. Projects are to be submitted electronically.
- The due date for late submissions is 11:59:59 PM on Tuesday, December 15, 2015.
- **NOTE:** No late penalty will be applied to late submissions. In other words, feel free to submit this project by 11:59:59 PM on Tuesday, December 15, 2015 at no penalty.
- This project will count as 18% of your final course grade.
- This project is to be completed either **individually** or in a **team of two**. Do not share your code with anyone else.
- You **must** use one of the following programming languages: C, C++, Java, or Python.
- Your program **must** successfully compile and run on Ubuntu v14.04.3 LTS or v15.04 (consider using `http://c9.io`).

Project Specifications

In this fourth (and final) project, you will focus on disk storage and network programming. More specifically, you will write server code to implement a storage server using server sockets. As part of your server implementation, you will implement (i.e., simulate) a clustered disk space allocation scheme.

For your server, clients connect via TCP sockets to server port 8765 (set this as `listener_port`). Your server must **not** be a single-threaded iterative server. Instead, your server must use multiple threads, multiple child processes, or a multiplexing `select()`-based approach.

Note that your server must support clients implemented in any language (e.g., Java, C, Python, etc.). Though you only need to submit your server code, feel free to create one or more test clients. Test clients will **not** be provided to you. You should also use `telnet` and `netcat` to test your server.

Application-Level Protocol

Clients connect to the server and can add, delete, and read files; clients can also request a list of files available on the server. Note that both text and binary (e.g., image) files must be supported. The application-level protocol between client and server is a line-based protocol (see below). Streams of bytes (i.e., characters) are transmitted between clients and your server.

Note that this protocol requires a connection-based protocol at the transport layer (i.e., TCP). Once a client is connected, your server must handle as many client commands as necessary, closing the socket connection and terminating only when it detects that the remote client has closed its socket.

The application-level protocol must be implemented exactly as shown below:

```
STORE <filename> <bytes>\n<file-contents>
-- add file <filename> to the storage server
-- if the file already exists, return an "ERROR: FILE EXISTS\n" error
-- in general, return "ERROR: <error-description>\n" if unsuccessful
-- return "ACK\n" if successful


READ <filename> <byte-offset> <length>\n
-- server returns <length> bytes of the contents of file <filename>,
   starting at <byte-offset>
-- note that this does NOT remove the file on the server
-- if the file does not exist, return an "ERROR: NO SUCH FILE\n" error
-- if the file byte range is invalid, return an "ERROR: INVALID BYTE RANGE\n" error
-- in general, return "ERROR: <error-description>\n" if unsuccessful
-- return "ACK" if successful, following it with the length and data, as follows:

      ACK <bytes>\n<file-excerpt>


DELETE <filename>\n
-- delete file <filename> from the storage server
-- if the file does not exist, return an "ERROR: NO SUCH FILE\n" error
-- in general, return "ERROR: <error-description>\n" if unsuccessful
-- return "ACK\n" if successful


DIR\n
-- server returns the list of files currently stored on the server
-- the list must be in alphabetical order
-- the format of the message containing the list of files is as follows:

      <number-of-files>\n<filename1>\n<filename2>\n...\n

-- therefore, if no files are stored, "0\n" is returned
```

For error messages, use a short human-readable description. Expect clients to display error descriptions to users.

Note that commands are case-sensitive. You must ensure that invalid commands received by the server do not crash the server. In general, return an error and an error description if something is incorrect or goes wrong. To delimit messages or message fields, note the use of newline characters. This should help to determine message boundaries.

Subdirectories are not to be supported. A filename is simply a valid filename without any relative or absolute path specified. We will test using files containing alphanumeric and `'.'` characters; you may assume that a filename starts with an alpha character.

Be very careful to stick to the protocol or else your server might not work with all clients (and with all tests we use for grading).

Note that you may assume that the correct number of bytes will be sent and received by client and server (and vice versa). In practice, this is not a safe assumption, but it should greatly simplify your implementation.

Simulating a Clustered Disk Space Allocation Scheme

To further study the clustered disk space allocation scheme, your server must simulate this scheme for storing file data in your running server program.

For the simulation component, assume that your disk space consists of a fixed number of equally sized blocks. Define this fixed number of blocks as `n_blocks`, with block size defined as `blocksize`. Use a default `n_blocks` value of 128 and a default `blocksize` of 4096 bytes.

For each file to be stored in your server, first determine how many blocks are required. Next, allocate blocks in one or more clusters. Note that a cluster is defined as a contiguous set of one or more blocks.

Your server must keep track of where each file's blocks are stored in your simulated disk.

Aside from simulating this clustered disk space allocation, you may simply store files on disk. More specifically, when a valid `STORE` command is received, simulate the clustered allocation, then just store the file data to disk.

Note that if the simulated disk space allocation fails (because there is not enough space for the given file), return an error (i.e., `ERROR: INSUFFICIENT DISK SPACE`) and do not store the file.

Store files on the server in a hidden directory called `.storage`, which your program creates, if necessary. If the directory already exists, remove all existing files from this directory (be careful!).

When a valid `READ` command is received, use your internal data structure to determine which block(s) must be read, displaying this as part of the server output. Next, read the file from disk using the appropriate byte offset and length.

Output Requirements

Your server is required to output one or more lines describing each command that it executes. Required output is illustrated in the example below; note that this is sample multi-threaded output (so for multiple processes, show the process IDs instead of the thread IDs). Also note that your server must display the simulated disk space after a file is added or deleted.

```
Block size is 4096
Number of blocks is 128
Listening on port 8765
Received incoming connection from <client-hostname-or-IP>
[thread 134558720] Rcvd: STORE abc.txt 25842
[thread 134558720] Stored file 'A' (25842 bytes; 7 blocks; 1 cluster)
[thread 134558720] Simulated Clustered Disk Space Allocation:
================================
AAAAAAA.........................
................................
................................
................................
================================
[thread 134558720] Sent: ACK
[thread 134558720] Rcvd: READ xyz.jpg 5555 2000
[thread 134558720] Sent: ERROR: NO SUCH FILE
[thread 134558720] Client closed its socket....terminating

...

[thread 144513911] Simulated Clustered Disk Space Allocation:
================================
AAAAAAABBBBBBBBBBBBBBBBBBBBBCCCC
CCCC.........EEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE.
................GGGGGGGGGGGGGGG.
================================

...

Received incoming connection from <client-hostname-or-IP>
[thread 119384882] Rcvd: STORE def.txt 79112
[thread 119384882] Stored file 'H' (79112 bytes; 20 blocks; 2 clusters)
[thread 119384882] Simulated Clustered Disk Space Allocation:
================================
AAAAAAABBBBBBBBBBBBBBBBBBBBBCCCC
CCCCHHHHHHHHHEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEH
HHHHHHHHHH......GGGGGGGGGGGGGGG.
================================
[thread 119384882] Sent: ACK
[thread 119384882] Client closed its socket....terminating

...

Received incoming connection from <client-hostname>
[thread 134559232] Rcvd: READ abc.txt 4090 5000
[thread 134559232] Sent: ACK 5000
[thread 134559232] Sent 5000 bytes (from 3 'A' blocks) from offset 4090
[thread 134559232] Rcvd: DELETE abc.txt
[thread 134559232] Deleted abc.txt file 'A' (deallocated 7 blocks)
[thread 119384882] Simulated Clustered Disk Space Allocation:
================================
.......BBBBBBBBBBBBBBBBBBBBBCCCC
CCCCHHHHHHHHHEEEEEEEEEEEEEEEEEEE
EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEH
HHHHHHHHHH......GGGGGGGGGGGGGGG.
================================
[thread 134559232] Sent: ACK
[thread 134559232] Client closed its socket....terminating

...
```

Submission Instructions

To submit your project, please create a single compressed and zipped file (using `tar` and `gzip`). Please include only source and documentation files (i.e., do not include executables or binary files!).

To package up your submission, use `tar` and `gzip` to create a compressed `tar` file using one of your team member's RCS userid, as in `goldsd.tar.gz`, that contains your source files (e.g., `main.c` and `file2.c`); include a `readme.txt` file only if necessary.

Here's an example showing how to create this file:

```
bash$ tar cvf goldsd.tar main.c file2.c readme.txt
main.c
file2.c
readme.txt
bash$ gzip -9 goldsd.tar
```

Submit the resulting `goldsd.tar.gz` file via the corresponding project submission link available in LMS (`http://lms9.rpi.edu`). The link is in the Assignments section.

For team-based submissions, only one student is required to submit the code. Be sure all team member names are included at the top of each source file in a comment.