Advanced Computer Communications Assignment 1
Anonymous File Storage

Jhi Morris (19173632) - 25/09/2020

Usage

The server does not support any user input once launched. The server can be launched from the command-line as **'./server k t1 t2 [port]'**, where k is the number of attempts a user is given to submit a valid key, t1 is the number of seconds the user is locked out once out of attempts, t2 is the number of seconds allowed between requests before the connection is closed, and optionally, port is the port to run the server at. The default port is 52000.

Example: **'./server 5 10.5 120'** to start the server that locks the user out for ten and a half seconds after five incorrect keys, and will close an idle connection after two minutes.

The client can be launched as **'./client ip port',** where ip is the hostname or IP address of the server, (IP addresses must be in dot-decimal notation [IPv4] or colon-hexadecimal notation [IPv6]), and port is the network port that the server is running at.

Example: **'./client 192.168.1.234 1234'** to connect to a server running at 192.168.1.234 on port 1234

Example: **'./client localhost 52001'** to connect to a server running on the same machine as the client, on port 52001.

Once launched, commands can be input into the client. The following commands are accepted, along with their expected arguments and a usage description:

- **'STORE filename'** where filename is the path of the file to upload to the server.
- **'GET key filename'** where key is the key of the file to retrieve, and filename is the path to save the downloaded file at.
- **'DELETE key'** where key is the key of the file to delete from the server.
- **'HISTORY key'** where key is the key of the file to retrieve the history of.
- **'QUIT'** to close the connection to the server.

Server / Client Communication

The format in which the client and server communicate via sockets is as defined below:

- **Command** (unsigned 8bit int)
- **Length** (unsigned 64bit int)
- **Body** (Length bytes of chars

The Command is an integer code, whose value corresponds with each command in the following way:

1: **STORE** Body will contain file to be stored.
2: **GET** Body will contain key of file to retrieve.
3: **DELETE** Body will contain key of file to delete.
4: **HISTORY** Body will contain key of file to view history of.
5: **QUIT** Body is ignored.
6: **FILECONT** Body is file to be saved by the client.
7: **MESSAGE** Body is text to be printed by the client.
8: **DISCON** Body is text to be printed by the client, before closing the connection.

Note that the server ignores commands 6-8 (FILECONT, MESSAGe, DISCON), and the client ignores commands 1-5 (STORE, GET, DELETE, HISTORY, QUIT). The client will ignore a 6 (FILECONT) when it does not expect it.

The Length corresponds with the number of bytes in the Body.

The Body contains the message, encoded key, or file contents required for an operation. If the client makes a request for a file, it will save the file to the prepared filename if a file is sent by the server. If a message is sent by the server, it will instead print the message.

Mutual Exclusion

Upon a new connection from a non-banned IP being established with the server, a new thread is created to handle requests made by the connecting client. The thread is then detached, so as not to consume system resources once the connection is finished and the thread closes.

Access to the following resources is shared with all threads of the program:

FileList - which is a list containing nodes for each file stored by the server.
BanList - which is a list containing nodes for each IP that has been banned.
Integers **timeout**, **attempts** - the ban duration, and the number of fails before a ban.

As the timeout and max attempts values are never written past initial startup (when they are parsed from the program arguments) no mutual exclusion is required for them.

The FileList and BanList each have a single mutex which enforces that only a single thread may read or write to the list at a time, in order to avoid dirty reads or data corruption. Threads obtain a lock of the lists's mutex before attempting any read or write operations to the list or its nodes.

Known Issues

This cannot transfer files of a size bigger than 2^64 bytes.

Upon an IP being banned by a different concurrent connection, other connections are not closed until timeout or upon sending an additional request.

The server has no way of storing persistence between restarts. Thus, file history is lost when the server stops.

When the client is connecting using a hostname, it tries only the first IP address resolved from the name, not all of them.

The server handles each client connection using threads, not processes.

If two files with the same hash are stored, any requests will return the last non-deleted file with that hash stored. Any requests to a hash will apply to the last non-deleted file with that hash.

If the client inputs a hash longer than the maximum hash length, it will silently be truncated to the max key length before being transmitted to the server.

Source Code

common.h

```
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <signal.h>

#define MAXPATHLENGTH 4096
#define MAXPATHLENGTHSTR "4096"
#define KEYLENGTH 33 //128bit MD5, represented in hex as 32 chars, with a null terminator

//COMMAND defines
#define COMMANDMIN 1
#define STORE 1
#define GET 2
#define DELETE 3
#define HISTORY 4
#define QUIT 5
#define FILECONT 6 //file content response
#define MESSAGE 7 //message response
#define DISCON 8 //message + disconnect notice
#define COMMANDMAX 8

//used for string comparison. commands[i] should match above defines name and value
extern const char* const commands[];
extern const size_t commandsLen;

typedef struct Message {
  uint8_t command;
  uint64_t length;
  char *body;
} Message;

int writeFile(const char* const fileContents, const uint64_t length, const char* const filename);
int readFile(char** fileContents, uint64_t* length, const char* const filename);

int sendMessage(const Message msg, int sock);

int recieveMessage(Message* msg, int sock);
```

common.c

```
#include "common.h"

//used for string comparison to commands, or for printing command names
const char *const commands[] = {"store", "get", "delete", "history", "quit", "filecont", "message",
"discon"};
const size_t commandsLen = sizeof(commands) / sizeof(commands[0]);


/* sendMessage
*PURPOSE: Sends the contents of the Message struct over the socket connection. It returns 'true' if
an error occurs.
*INPUT: Message message, int sock descriptor
*OUTPUTS: int error occured (boolean)
*/
int sendMessage(const Message msg, int sock)
{
  int error = false;

  if(send(sock, &(msg.command), sizeof(msg.command), 0))
  {
   if(send(sock, &(msg.length), sizeof(msg.length), 0))
   {
   if(!send(sock, msg.body, sizeof(char) * msg.length, 0))
   { //failed to send body
          error = true;
   }
   }
   else
   { //failed to send length
   error = true;
   }
  }
  else
  { //failed to send command
   error = true;
  }

  return !error;
}

/* recieveMessage
*PURPOSE: Recieves the contents of a message from the socket connection and writes it into the
Message struct at the pointer passed into it. It returns 'true' if an error occurs.
*INPUT: int sock descriptor
*OUTPUTS: int error occured (boolean), Message msg
*/
int recieveMessage(Message *msg, int sock)
{
  int error = false;

  if(recv(sock, &(msg->command), sizeof(msg->command), MSG_WAITALL) == sizeof(msg->command))
  {
   if(recv(sock, &(msg->length), sizeof(msg->length), MSG_WAITALL) == sizeof(msg->length))
   {
   msg->body = calloc(msg->length + 1, sizeof(char));

   if(recv(sock, msg->body, sizeof(char) * msg->length, MSG_WAITALL) == sizeof(char) * msg->length)
   {
          msg->body[msg->length] = '\0'; //ensure null termination
          error = false;
   }
   else
   { //failed to get full body
           error = true;
   }
   }
   else
   { //failed to get length info
   error = true;
   }
  }
  else
  { //failed to get command info or connection closed
   error = true;
  }

  return !error;
}

/* writeFile
*PURPOSE: Writes the file contents (of length as per the second parameter)
*  into a file at the path filename. Returns 'true' if an error occurs.
*INPUT: char* file contents, unsigned int length, char* filename.
*OUTPUTS: int error occured (boolean)
*/
int writeFile(const char *const fileContents, const uint64_t length, const char *const filename)
{
  FILE *file = fopen(filename, "w");
  int error = false;

  if(file != NULL)
  {
   if(fwrite(fileContents, sizeof(char), length, file) == length)
   {
   error = false;
   }
   else
   { //not all bytes written
   error = true;
   }

   fclose(file);
  }
  else
  { //failed to open file
   error = true;
  }

  return !error;
}

/* readFile
*PURPOSE: Finds the length of the file at the path filename, writes it into
*  the length parameter, allocates memory for the file contents and then reads
*  the file into it. Returns 'true' if an error occurs.
*INPUT: char* filename.
*OUTPUTS: int error occured (boolean), char** file contents, unsigned int* length
*/
int readFile(char **fileContents, uint64_t *length, const char *const filename)
{
  FILE *file = fopen(filename, "rb");
  int error = false;

  if(file != NULL)
  {
   fseek(file, 0, SEEK_END);
   *length = ftell(file);

   if(*length > 0)
   {
   rewind(file);
   *fileContents = calloc(*length, sizeof(char));

   if(fread(*fileContents, sizeof(char), *length, file) == *length)
   {
        error = false;
   }
   else
   { //failed to read, or file length changed mid-read and buffer may be corrupt.
        error = true;
   }
   }
   else
   { //empty file
   error = true;
   }

   fclose(file);
  }
  else
  { //failed to open file
   error = true;
  }

  return !error;
}
```

server.h

```
#include "common.h"
#include <time.h>
#include <pthread.h>
#include <poll.h>
#include <errno.h>

#define DEFAULT_PORT 52000
#define MAX_BACKLOG 10 //max incoming client connections backlog length

typedef struct Connection
{
  int sd; //socket descriptor
  struct sockaddr_in6 client;
  socklen_t len;
  int fails;
} Connection;

typedef struct AddressNode
{
  struct in6_addr ip;
  time_t banTime;
  struct AddressNode* next;
} AddressNode;

typedef struct AddressList
{ //linked list of ip addresses
  pthread_mutex_t* mutex;
  AddressNode* head;
} AddressList;

typedef struct FileHistoryNode
{
  struct FileHistoryNode* next;
  uint8_t command;
  time_t time;
  struct in6_addr ip;
} FileHistoryNode;

typedef struct FileHistory
{ //linked list of file operations
  FileHistoryNode *head;
} FileHistory;

typedef struct FileNode
{
```

```c
  struct FileNode* next;
  char path[MAXPATHLENGTH];
  char key[KEYLENGTH]; //128bit MD5 hash (as hex, 32 characters)
  FileHistory history;
} FileNode;

typedef struct FileList
{ //linked list of file info
  pthread_mutex_t* mutex;
  unsigned int count; //used for naming files
  FileNode* head;
} FileList;

typedef struct ConnectionThread
{ //used for handleConnection() threads
  Connection* con;
  AddressList* banList;
  FileList* fileList;
  int attempts;
  int timeout;
} ConnectionThread;

int server(const int attempts, const int lockout, const int timeout, int sock);

void *handleConnection(void *arg);

int bannedAddr(AddressList *banList, struct in6_addr ip);

void unbanAddrs(AddressList *banList, const int lockout);

void banAddr(AddressList *banList, struct in6_addr ip);

int store(Message* msgIn, Message* msgOut, FileList* fileList, struct in6_addr ip);

int get(Message* msgIn, Message* msgOut, FileList* fileList, struct in6_addr ip);

int delete(Message* msgIn, Message* msgOut, FileList* fileList);

int history(Message* msgIn, Message* msgOut, FileList* fileList, struct in6_addr ip);

void addHistory(FileHistory* history, uint8_t command, struct in6_addr ip);

void removeNode(FileNode* node, FileList* list);
```

server.c

```c
#include "server.h"

/* main
*PURPOSE: Reads in and validates the server parameters from the command line
*  arguments and binds a two-stack (ipv4 & ipv6) socket for server().
*INPUT: argv[1] max attempts, argv[2] seconds timeout, argv[3] seconds of idle
*  before connection timeout, optional argv[4] port.
*OUTPUTS: -
*/
int main(int argc, char *argv[])
{ //parses args and starts server
  int error = false;
  long port = DEFAULT_PORT;
  long attempts, lockout, timeout;
  char *endptr;

  if(argc != 4)
  {
   if(argc != 5)
   {
   printf("Invalid number of arguments.\n");
   error = true;
   }
```

```
 else
 { //if argc is 5, optional port argument may have been included
 port = strtol(argv[4], NULL, 10);
 }
}

if(!error)
{
 attempts = strtol(argv[1], &endptr, 10);

 if(!error && attempts < 1)
 {
 printf("First argument (number of attempts) must be an integer greater than zero.\n");
 error = true;
 }

 lockout = strtol(argv[2], &endptr, 10);

 if(!error && (lockout < 0 || argv[2] == endptr))
 {
 printf("Second argument (lockout time) must be a positive integer.\n");
 error = true;
 }

 timeout = strtol(argv[3], &endptr, 10);

 if(!error && (timeout < 0 || argv[3] == endptr))
 {
 printf("Third argument (timeout time) must be a positive integer.\n");
 error = true;
 }
}

if(!error && (port < 1 || port > 65535))
{
 printf("Forth argument (port) must be an integer between 1 and 65535, inclusive.\n");
 error = true;
}

int sock = -1;

if(!error && (sock = socket(AF_INET6, SOCK_STREAM, 0)) < 0)
{
 printf("Socket error.\n");
 error = true;
}

int mode = 0;
if(!error && setsockopt(sock, IPPROTO_IPV6, IPV6_V6ONLY, (char*)&mode, sizeof(mode)) < 0)
{
 printf("setsockopt(IPV6_V6ONLY, 0) failed: only operating in IPv6 mode. . .\n");
}

//bind to both ipv4 and ipv6
struct sockaddr_in6 ip;
memset(&ip, 0, sizeof(ip));
ip.sin6_family = AF_INET6;
ip.sin6_port = htons(port);
ip.sin6_addr = in6addr_any;

if(!error && bind(sock, (struct sockaddr*)&ip, sizeof(ip)) < 0)
{
 printf("Failed to bind to the port. Is there already a service running at this port?\n");
 error = true;
}

if(!error && listen(sock, MAX_BACKLOG))
{
 printf("Failed to listen on port.\n"); //no clue how it might reach this condition
 error = true;
}

if(!error)
{
    printf("Starting server. . .\n");
    error = server(attempts, lockout, timeout, sock);
    printf("Server shutting down. . .\n");
   }
   else
   {
    printf("Expected usage: './server k t1 t2 [port]', where k is the number "\
    "of attempts a user is given to submit a valid key, t1 is the number of "\
    "seconds the user is locked out once out of attempts, t2 is the number "\
    "of seconds allowed between requests before the connection is closed, "\
    "and optionally, port is the port to run the server at. The default "\
    "port is 52000.\nExample: './server 5 10.5 120' to start the server that "\
    "locks the user out for ten and a half seconds after five incorrect keys, "\
    "and will close an idle connection after two minutes.\n");
   }

   if(sock != -1)
   {
    close(sock);
   }

   return !error;
}

/* server
*PURPOSE: Manages the banlist and recieves connections to the socket and creates threads to handle
them if they are not from a banned address.
*INPUT: int max fails, int lockout time, int idle time limit
*OUTPUTS: int error occured (boolean)
*/
int server(const int attempts, const int lockout, const int timeout, int sock)
{
   int error = false;
   AddressList banList;
   banList.head = NULL;
   banList.mutex = malloc(sizeof(pthread_mutex_t));
   pthread_mutex_init(banList.mutex, NULL);

   FileList fileList;
   fileList.head = NULL;
   fileList.count = 0;
   fileList.mutex = malloc(sizeof(pthread_mutex_t));
   pthread_mutex_init(fileList.mutex, NULL);

   while(!error)
   {
    Connection *connection = calloc(1, sizeof(Connection));

    unbanAddrs(&banList, lockout);

    if((connection->sd = accept(sock, NULL, NULL)) < 0)
    {
    printf("Connection error.\n");
    error = true;
    }
    else
    {
    socklen_t addrlen = sizeof(connection->client); //needed to pass ref to getpeername()

    getpeername(connection->sd, (struct sockaddr*)&(connection->client), &addrlen);

    if(bannedAddr(&banList, connection->client.sin6_addr))
    { //if connecting IP is still banned, we send rejection and close the connection
            Message msgOut;
            char msg[] = "Error: This address is banned.";
            msgOut.command = DISCON;
            msgOut.length = sizeof(msg);
            msgOut.body = calloc(1, sizeof(msg));
            memcpy(msgOut.body, msg, sizeof(msg));

            sendMessage(msgOut, connection->sd);

            printf("SERVER Info: Rejected connection from banned address.\n");
            if(connection->sd != -1)
            {
            close(connection->sd);
            }
     }
     else //TODO convert to use PROCESSES, not threads
     { //if not banned, make thread to handle connection
            pthread_t thread;
            ConnectionThread *cont = calloc(1, sizeof(ConnectionThread));
            cont->con = connection;
            cont->attempts = attempts;
            cont->timeout = timeout;
            cont->banList = &banList;
            cont->fileList = &fileList;

            printf("SERVER Info: New connection.\n");

            pthread_create(&thread, NULL, handleConnection, (void*)cont);
            pthread_detach(thread);
     } //the thread will handle closing its own connection
     }
    }

  pthread_mutex_destroy(banList.mutex);
  pthread_mutex_destroy(fileList.mutex);

  free(banList.mutex);
  free(fileList.mutex);

  return error;
}

/* handleConnection
*PURPOSE: Thread function to handle a client connection. Recieves Message requests and sends Message
responses over the socket.
*INPUT: void* to a ConnectionThread struct
*OUTPUTS: -
*/
void *handleConnection(void *arg)
{ //thread function, expects ConnectionThread *argument
  ConnectionThread *cont = (ConnectionThread*)arg;
  int quit = false;
  int valid = false;
  Message msgIn;
  Message msgOut;

  struct sockaddr_in6 addr; //used for logging ip in history
  socklen_t addrlen = sizeof(addr);
  getpeername(cont->con->sd, (struct sockaddr*)&addr, &addrlen);

  signal(SIGPIPE, SIG_IGN); //failed socket operations are handled as they occur

  struct pollfd polld;
  polld.fd = cont->con->sd;
  polld.events = POLLIN;

  cont->con->fails = 0;

  char welcome[] = "Welcome to our anonymous storage.";
  msgOut.length = sizeof(welcome);
  msgOut.body = calloc(1, sizeof(welcome));
  msgOut.command = MESSAGE;
  memcpy(msgOut.body, welcome, sizeof(welcome));

  if(sendMessage(msgOut, cont->con->sd))
  {
   free(msgOut.body);

   while(!quit)
   {
   switch(poll(&polld, 1, (cont->timeout)*1000))
   {
            case -1: //error
            printf("NETWORK Error: Failed to poll connection.\n");
		valid = false;
		break;
		case 0: //timeout
		printf("NETWORK Info: Connection timed out.\n");
		valid = false;
		break;
		default: //data available or connection died
		valid = recieveMessage(&msgIn, cont->con->sd);
		break;
	}

	if(valid)
	{
		switch(msgIn.command)
		{
		case STORE:
		store(&msgIn, &msgOut, cont->fileList, addr.sin6_addr);
		break;
		case GET:
		if(get(&msgIn, &msgOut, cont->fileList, addr.sin6_addr))
		{
		cont->con->fails = 0;
		}
		else
		{ //invalid key
		cont->con->fails++;
		}
		break;
		case DELETE:
		if(delete(&msgIn, &msgOut, cont->fileList))
		{
		cont->con->fails = 0;
		}
		else
		{ //invalid key
		cont->con->fails++;
		}
		break;
		case HISTORY:
		if(history(&msgIn, &msgOut, cont->fileList, addr.sin6_addr))
		{
		cont->con->fails = 0;
		}
		else
		{ //invalid key
		cont->con->fails++;
		}
		break;
		case QUIT: ;
		char msg[] = "Thank you for using our anonymous storage.";
		quit = true;
		msgOut.command = MESSAGE;
		msgOut.length = sizeof(msg);
		msgOut.body = calloc(1, sizeof(msg));
		memcpy(msgOut.body, msg, sizeof(msg));
		break;
		default: ;//server ignores FILECONT, MESSAGE and DISCON commands
		char errorMsg[] = "Error: Unrecognized command.";
		msgOut.command = MESSAGE;
		msgOut.length = sizeof(errorMsg);
		msgOut.body = calloc(1, sizeof(errorMsg));
		memcpy(msgOut.body, errorMsg, sizeof(errorMsg));
		quit = true;
		break;
		}

		if(cont->con->fails > cont->attempts)
		{
		banAddr(cont->banList, cont->con->client.sin6_addr);
		printf("Info: Error limit exceeded. IP address banned.\n");
		char errorMsg[] = "Error: Error limit exceeded. IP address banned.";
		msgOut.command = DISCON;
		msgOut.length = sizeof(errorMsg);
		msgOut.body = calloc(1, sizeof(errorMsg));
```

```
          memcpy(msgOut.body, errorMsg, sizeof(errorMsg));
          quit = true;
          }

          if(!sendMessage(msgOut, cont->con->sd))
          { //failed to send message
          quit = true;
          printf("NETWORK Error: Failed to send message.\n");
          }

          free(msgOut.body);
          free(msgIn.body);
   }
   else
   {  //connection has died or invalid request made
          quit = true;
          printf("NETWORK Info: Invalid connection dropped.\n");
   }

   }
  }
  else
  { //failed to send welcome
   quit = true;
   printf("NETWORK Error: Failed to send welcome message.\n");
   free(msgOut.body);
  }

  if(cont->con->sd != -1)
  {
   close(cont->con->sd);
  }

  free(cont->con);
  free(cont);

  printf("SERVER Info: Closed connection.\n");

  return NULL;
}


/* bannedAddr
*PURPOSE: Returns true if the IP address is found in the banlist.
*INPUT: AddressList ban list, struct in6_addr IP.
*OUTPUTS: int banned (boolean)
*/
int bannedAddr(AddressList *banList, struct in6_addr ip)
{
  int found = false;

  pthread_mutex_lock(banList->mutex);

  AddressNode *addr = banList->head;

  while(!found && addr != NULL)
  {
   if(!memcmp(&(addr->ip), &ip, sizeof(ip)))
   {
   found = true;
   }
   else
   {
   addr = addr->next;
   }
  }

  pthread_mutex_unlock(banList->mutex);

  return found;
}

/* unbanAddrs
*PURPOSE: Unbans any addresses on the banlist whose ban time is older than
```

```
*  the lockout duration.
*INPUT: AddressList* ban list, int lockout duration
*OUTPUTS: -
*/
void unbanAddrs(AddressList *banList, const int lockout)
{
  pthread_mutex_lock(banList->mutex);

  time_t unbanTime = time(NULL) - lockout; //latest bantime an address could have and still be ready
to be unbanned
  AddressNode *addr = banList->head;
  AddressNode *prevAddr;

  if(addr != NULL)
  { //handle first element manually, as it has no previous element
   if(addr->banTime < unbanTime)
   {
   banList->head = addr->next;
   }

   prevAddr = addr;
   addr = addr->next;

   while(addr != NULL)
   { //handle other elements
   if(addr->banTime < unbanTime)
   { //if unbanned, remove from list.
           prevAddr->next = addr->next;
           free(addr);
           addr = prevAddr->next;
   }
   }
  }

  pthread_mutex_unlock(banList->mutex);
}

/* banAddr
*PURPOSE: Adds the IP address to the ban list.
*INPUT: AddressList* ban list, struct in6_addr IP
*OUTPUTS: -
*/
void banAddr(AddressList *banList, struct in6_addr ip)
{
  AddressNode *newNode = calloc(1, sizeof(AddressNode));

  pthread_mutex_lock(banList->mutex);

  newNode->ip = ip;
  newNode->banTime = time(NULL);

  newNode->next = banList->head;
  banList->head = newNode;

  pthread_mutex_unlock(banList->mutex);
}

/* checkKey
*PURPOSE: Searches the file list for a node with a matching key, and returns
*  a pointer to it if found. Returns NULL if no match is found.
*INPUT: char* key, FileList* file list
*OUTPUTS: FileNode* file node
*/
FileNode *checkKey(char *key, FileList *list)
{ //mutex for this operation handled by calling function
  FileNode *node = list->head;

  while(node != NULL && memcmp(node->key, key, KEYLENGTH-1))
  {
   node = node->next;
  }

  return node;
}
```

```c
/* addHistory
*PURPOSE: Modifies the history list to then include a new node containing
*  a timestamp, the operation code, and the IP address.
*INPUT: FileHistory* history list, int command code, struct in6_addr IP
*OUTPUTS: -
*/
void addHistory(FileHistory *history, uint8_t command, struct in6_addr ip)
{ //mutex for this function handled by calling function
  FileHistoryNode *newNode = calloc(1, sizeof(FileHistoryNode));
  newNode->command = command;
  newNode->time = time(NULL);
  memcpy(&(newNode->ip), &(ip), sizeof(ip));

  if(history->head == NULL)
  {
   history->head = newNode;
  }
  else
  {
   FileHistoryNode *node = history->head;

   while(node->next != NULL)
   {
   node = node->next;
   }

   node->next = newNode;
  }
}

/* removeNode
*PURPOSE: Modifies the file list to not include the passed node.
*INPUT: FileNode node, FileList file list.
*OUTPUTS: -
*/
void removeNode(FileNode *node, FileList *list)
{ //mutex for this function handled by calling function
  FileHistoryNode *prevNode = node->history.head;

  if(prevNode != NULL)
  { //free node's history list
   FileHistoryNode *histNode = prevNode->next;

   while(histNode != NULL)
   {
   free(prevNode);
   prevNode = histNode;
   histNode = histNode->next;
   }

   free(prevNode);
  }

  if(list->head == node)
  {
   list->head = node->next;
   free(node);
  }
  else
  {
   FileNode *listNode = list->head;

   while(listNode->next != node && listNode->next != NULL)
   {
   listNode = listNode->next;
   }

   if(listNode->next != NULL)
   { //listNode->next is the node we're looking for
   listNode->next = node->next;
   free(node);
   }
   else
   { //node not found - may have been deleted already. . .
   free(node);
   }
  }
}

/*
* Below (until the end of the file) are the functions which handle client
* commands. All of them take pointers for an input message and an output
* message, as well as the IP address of the client.
* They all return 'true' if the request constitutes an action which should
* increment the counter towards a ban, and modify the struct at the pointer to
* the output message to be the response to the request.
*/

int store(Message *msgIn, Message *msgOut, FileList *fileList, struct in6_addr ip)
{
  FileNode *fileNode;

  msgOut->command = MESSAGE;

  char path[MAXPATHLENGTH];

  pthread_mutex_lock(fileList->mutex);

  snprintf(path, MAXPATHLENGTH, "file_%d", fileList->count);

  if(writeFile(msgIn->body, msgIn->length, path))
  {
   FILE *hashStream;
   char prog[] = "md5sum ";
   char *command = calloc((strlen(prog) + strlen(path) + 1), sizeof(char));
   snprintf(command, (strlen(prog) + strlen(path) + 1), "%s%s", prog, path);

   fileNode = calloc(1, sizeof(FileNode));
   strcpy(fileNode->path, path);

   hashStream = popen(command, "r");

   free(command);

   if(hashStream != NULL)
   {
   if(fscanf(hashStream, "%s ", fileNode->key) == 1)
   {
           if(strlen(fileNode->key) == (KEYLENGTH - 1)) //-1 to ignore expected '\0' which strlen
does not count
           {
           fileNode->history.head = NULL;
           addHistory(&(fileNode->history), STORE, ip);
           printf("SERVER Info: Stored file %s.\n", path);

           fileNode->next = fileList->head;
           fileList->head = fileNode;
           fileList->count++;

           char errorMsg[] = "Info: File has been stored with hash key: ";
           msgOut->length = KEYLENGTH + sizeof(errorMsg);
           msgOut->body = calloc(1, sizeof(fileNode->key) + sizeof(errorMsg));
           memcpy(msgOut->body, errorMsg, sizeof(errorMsg));
           memcpy(msgOut->body + sizeof(errorMsg) - 1, fileNode->key, sizeof(fileNode->key));
           }
           else
           { //failed to read key of valid length
           printf("SERVER Error: Failed to read valid hash from md5sum.");
           char errorMsg[] = "Info: Error while trying to hash file.";
           msgOut->length = sizeof(errorMsg);
           msgOut->body = calloc(1, sizeof(errorMsg));
           memcpy(msgOut->body, errorMsg, sizeof(errorMsg));
           }
   }
   else
   { //failed to read
           printf("SERVER Error: Failed to read from md5sum.");
```

```
        char errorMsg[] = "Info: Error while trying to hash file.";
        msgOut->length = sizeof(errorMsg);
        msgOut->body = calloc(1, sizeof(errorMsg));
        memcpy(msgOut->body, errorMsg, sizeof(errorMsg));
   }

   pclose(hashStream);
   }
   else
   { //failed to open process to md5sum
   printf("SERVER Error: Failed to load md5sum to hash file.");
   char errorMsg[] = "Info: Error while trying to hash file.";
   msgOut->length = sizeof(errorMsg);
   msgOut->body = calloc(1, sizeof(errorMsg));
   }
  }
  else
  { //failed to write
   printf("SERVER Error: Failed to write to file %s for STORE operation.\n", path);
   char msg[] = "Info: File failed to save. Please try again later.";
   msgOut->length = sizeof(msg);
   msgOut->body = calloc(1, sizeof(msg));
   memcpy(msgOut->body, msg, sizeof(msg));
  }

  pthread_mutex_unlock(fileList->mutex);

  return true; //no bannable offences possible with this function
}

//command function, see above
int get(Message *msgIn, Message *msgOut, FileList *fileList, struct in6_addr ip)
{
  int error = false;

  pthread_mutex_lock(fileList->mutex);

  FileNode *node = checkKey(msgIn->body, fileList);

  if(node != NULL)
  {
   if(readFile(&(msgOut->body), &(msgOut->length), node->path))
   {
   msgOut->command = FILECONT;
   error = false;
   printf("SERVER Info: Retrieved file %s.\n", node->path);
   }
   else
   { //failed to read file for valid key. should never happen
   printf("SERVER Error: Failed to read file %s.\n", node->path);
   char msg[] = "Info: Key found, but the file cannot be read. Please try again later.";
   msgOut->length = sizeof(msg);
   msgOut->command = MESSAGE;
   msgOut->body = calloc(1, sizeof(msg));
   memcpy(msgOut->body, msg, sizeof(msg));
   error = false; //not the user's fault; system error
   }

   addHistory(&(node->history), GET, ip);
  }
  else
  { //key not found
   char msg[] = "Error: Hash key not valid.";
   msgOut->length = sizeof(msg);
   msgOut->command = MESSAGE;
   msgOut->body = calloc(1, sizeof(msg));
   memcpy(msgOut->body, msg, sizeof(msg));
   error = true;
  }

  pthread_mutex_unlock(fileList->mutex);

  return !error;
}
```

```
//command function, see above
int delete(Message *msgIn, Message *msgOut, FileList *fileList)
{
  int error = false;

  msgOut->command = MESSAGE;

  pthread_mutex_lock(fileList->mutex);

  FileNode *node = checkKey(msgIn->body, fileList);

  if(node != NULL)
  {
   if(!remove(node->path))
   {
   printf("SERVER Info: Deleted file %s.\n", node->path);
   removeNode(node, fileList);
   char msg[] = "Info: File with hash key has been deleted.";
   msgOut->length = sizeof(msg);
   msgOut->body = calloc(1, sizeof(msg));
   memcpy(msgOut->body, msg, sizeof(msg));
   error = false;
   }
   else
   { //failed to delete file for valid key. should never happen
   printf("SERVER Error: Failed to delete file %s.\n", node->path);
   char msg[] = "Info: Key found, but the file cannot be deleted.";
   msgOut->length = sizeof(msg);
   msgOut->body = calloc(1, sizeof(msg));
   memcpy(msgOut->body, msg, sizeof(msg));
   error = false; //not the user's fault; system error
   }
  }
  else
  { //key not found
   error = true;
   char msg[] = "Error: Hash key not valid.";
   msgOut->length = sizeof(msg);
   msgOut->body = calloc(1, sizeof(msg));
   memcpy(msgOut->body, msg, sizeof(msg));
  }

  pthread_mutex_unlock(fileList->mutex);

  return !error;
}

//command function, see above
int history(Message *msgIn, Message *msgOut, FileList *fileList, struct in6_addr ip)
{
  int error = false;

  msgOut->command = MESSAGE;

  pthread_mutex_lock(fileList->mutex);

  FileNode *file = checkKey(msgIn->body, fileList);

  if(file != NULL)
  {
   char ipBuff[INET6_ADDRSTRLEN];
   char dateBuff[20]; //space of, eg "26-08-2020 22:59, "

   FileHistoryNode *node = file->history.head;
   msgOut->length = 0;

   while(node != NULL)
   { //calculate length of body required
   msgOut->length += strlen(commands[node->command - 1]) + 2; //space of, eg "STORE: "

   strftime(dateBuff, sizeof(dateBuff), "%d-%m-%Y %H:%M, ", localtime(&(node->time)));
   msgOut->length += strlen(dateBuff); //space of, eg "26-08-2020 22:59, "
```

```
    inet_ntop(AF_INET6, &(node->ip), ipBuff, sizeof(ipBuff));
    if(!strncmp("::ffff:", ipBuff, strlen("::ffff:")))
    { //ipv4-mapped address
            msgOut->length += strlen(ipBuff) + 1 - strlen("::ffff:"); //space of, eg
"128.225.212.210\n"
    }
    else
    { //actual ipv6 address
            msgOut->length += strlen(ipBuff) + 1; //space of, eg "2001:0DB8:AC10:FE01::1A2F:1A2B\n"
    }

    node = node->next;
    }

    if(msgOut->length != 0)
    {
    node = file->history.head;
    msgOut->body = calloc(msgOut->length, sizeof(char));

    while(node != NULL)
    { //copy data into body
            strcat(msgOut->body, commands[node->command - 1]);
            strcat(msgOut->body, ": ");

            strftime(dateBuff, sizeof(dateBuff), "%d-%m-%Y %H:%M, ", localtime(&(node->time)));
            strcat(msgOut->body, dateBuff);

            inet_ntop(AF_INET6, &(node->ip), ipBuff, sizeof(ipBuff));
            if(!strncmp("::ffff:", ipBuff, strlen("::ffff:")))
            { //ipv4-mapped address
            strcat((msgOut->body), ipBuff + strlen("::ffff:"));
            }
            else
            { //actual ipv6 address
            strcat(msgOut->body, ipBuff);
            }

            node = node->next;

            if(node != NULL)
            { //avoids printing extra newline on last line of history output
            strcat(msgOut->body, "\n");
            }
    }

    msgOut->length--; //as there is no newline on last line of output
    printf("SERVER Info: Retrieved history for file %s.\n", file->path);
    }
    else
    { //no history found; not an error but shouldn't happen
    printf("SERVER Error: Missing history for file %s.\n", file->path);
    char msg[] = "Info: File found, but no history recorded.";
    msgOut->length = sizeof(msg);
    msgOut->body = calloc(1, sizeof(msg));
    memcpy(msgOut->body, msg, sizeof(msg));
    }

    addHistory(&(file->history), HISTORY, ip);
   }
   else
   { //key not found
    char errorMsg[] = "Error: Key does not match any known file.";
    error = true;
    msgOut->length = sizeof(errorMsg);
    msgOut->body = calloc(1, sizeof(errorMsg));
    memcpy(msgOut->body, errorMsg, sizeof(errorMsg));
   }

  pthread_mutex_unlock(fileList->mutex);

  return !error;
}
```

client.h

```
#include "common.h"
#include <netdb.h>

int client(int sock);

Message prepareMessage(char **fileName);
```

client.c

```
/* client.c
*AUTHOR: Jhi Morris (19173632)
*MODIFIED: 2020-09-12
*PURPOSE: Handles client user interface and communication
*/

#include "client.h"

/* main
*PURPOSE: Reads in and validates the connection parameters from the command
*  line arguments and establishes a connection to the server.
*INPUT: argv[1] server address (ip or hostname), argv[2] port
*OUTPUTS: -
*/
int main(int argc, char *argv[])
{ //parses args and starts client cli
  int error = false;
  int sock;
  struct sockaddr_in6 ip;
  long portTest;

  if(argc != 3)
  {
   printf("Invalid number of arguments.\n");
   error = true;
  }
  else
  {
   portTest = strtol(argv[2], NULL, 10); //used to check for negative or out of range values
   ip.sin6_port = htons(strtol(argv[2], NULL, 10));
  }

  struct hostent *hostname = gethostbyname(argv[1]);

  if(hostname != NULL)
  {
   ip.sin6_addr = *((struct in6_addr*)hostname->h_addr_list[0]);
   ip.sin6_family = hostname->h_addrtype;
  }
  else
  { //not routable hostname
   error = true;
   if(!error && inet_pton(AF_INET, argv[1], &ip.sin6_addr) > 0)
   { //check ip validity and type
   ip.sin6_family = AF_INET;
   }
   else
   { //not IPv4
   if(inet_pton(AF_INET6, argv[1], &ip.sin6_addr) > 0)
   {
          ip.sin6_family = AF_INET6;
   }
   else
   { //not IPv6
```

```
          printf("First argument is not a valid hostname or IP address. IP addresses must be in
dot-decimal notation (IPv4) or colon-hexidecimal notation (IPv6).\n");
   }
   }
  }

  if(!error && (sock = socket(ip.sin6_family, SOCK_STREAM, 0)) < 0)
  { //check able to make socket for ip type
   printf("Socket error. Check network connectivity of this machine.\n");
   error = true;
  }

  if(!error && (portTest < 1 || portTest > 65535))
  { //check port validity
   printf("Second argument (port) must be an integer between 1 and 65535, inclusive.\n");
   error = true;
  }

  if(!error && connect(sock, (struct sockaddr*)&ip, sizeof(ip)) < 0)
  {
   printf("Connection error. Check network connectivity of this machine and the remote server.\n");
   error = true;
  }

  if(!error)
  {
   signal(SIGPIPE, SIG_IGN); //failed socket operations are handled as they occur
   error = client(sock);
  }
  else
  {
   printf("Expected usage: './client ip port', where ip is the hostname or "\
   "IP address of the server, (IP addresses must be in dot-decimal notation "\
   "[IPv4] or colon-hexidecimal notation [IPv6]), and port is the network "\
   "port that the server is running at.\nExample: './client 192.168.1.234 "\
   "1234' to connect to a server running at 192.168.1.234 on port 1234\n"\
   "Example: './client localhost 52001' to connect to a server running on"\
   "the same machine as the client, on port 52001.\n");

  }

  return !error;
}

/* client
*PURPOSE: Sends & recieves messages to/from the server.
*INPUT: int sock descriptor
*OUTPUTS: int error occured (boolean)
*/
int client(int sock)
{
  Message msgOut;
  Message msgIn;

  int error = false;
  int quit = false;

  if(recieveMessage(&msgIn, sock))
  { //catch welcome message (or ban notice)
   if(msgIn.command == MESSAGE)
   {
   printf("SERVER %.*s\n", (int)msgIn.length, msgIn.body); //print response
   }
   else
   { //not general welcome
   if(msgIn.command == DISCON)
   {
          quit = true;
          printf("SERVER %.*s\n", (int)msgIn.length, msgIn.body); //print response
   }
   else
   { //invalid command
          error = true;
          printf("NETWORK Error: Connection established, but invalid welcome recieved.\n");
```

```
 }
 }

 free(msgIn.body);
}
else
{ //failed to get welcome
 printf("NETWORK Error: Connection established, but invalid welcome recieved.\n");
 error = true;
}

while(!error && !quit)
{
 char *fileName; //used only for GET operations

 msgOut = prepareMessage(&fileName);

 if(sendMessage(msgOut, sock))
 {
 free(msgOut.body);

 if(recieveMessage(&msgIn, sock))
 {
        if(msgIn.command == MESSAGE)
        { //MESSAGE is always a valid response for any request
        printf("SERVER %.*s\n", (int)msgIn.length, msgIn.body); //print response

        if(msgOut.command == QUIT)
        {
        quit = true;
        }
        }
        else
        {
        if(msgIn.command == FILECONT && msgOut.command == GET)
        { //a FILECONT response is only valid for the GET command
        if(writeFile(msgIn.body, msgIn.length, fileName))
        {
        printf("LOCAL Info: File retrieved successfully.\n");
        }
        else
        { //failed to save file - not a communication error, program does not exit
        printf("LOCAL Error: Failed to save file.\n");
        }

        if(fileName != NULL)
        {
        free(fileName);
        }
        }
        else
        { //invalid or discon response

        if(msgIn.command == DISCON)
        {
        printf("SERVER %.*s\n", (int)msgIn.length, msgIn.body); //print response
        quit = true;
        }
        else
        {
        error = true;
        printf("LOCAL Error: Server sent an invalid response.\n");
        }
        }
        }

        free(msgIn.body);
 }
 else
 { //failed to recv message
        error = true;
        printf("NETWORK Error: Failed to recieve message. Connection closed.\n");
 }
 }
```

```
   else
   { //failed to send message
   printf("NETWORK Error: Failed to send message.\n");
   error = true;
   }

  }

  if(sock != -1)
  {
   close(sock);
  }

  return error;
}

/* prepareMessage
*PURPOSE: Takes command and parameter inputs from the user for the server.
*  Returns a Message struct containing all the information to be sent to the
*  server for the request. If the user has selected GET, then the filename for
*  the response to be saved at will be written to the filename pointer.
*INPUT: -
*OUTPUTS: Message request message, char** file name
*/
Message prepareMessage(char **fileName)
{ //handles ui and file io to prepare a message
  Message msg;
  int valid = false;
  char input[MAXPATHLENGTH]; //space for command, key, or max length file path
  *fileName = NULL;

  while(!valid)
  {
   printf(">");
   if(scanf("%"MAXPATHLENGTHSTR"s", input) == 1) //handle command
   {
   //used for string input comparison

   for(int i = 0; i < strlen(input); i++)
   { //convert to lower case
          if(input[i] >= 'A' && input[i] <= 'Z')
          {
          input[i] += 'a' - 'A';
          }
   }

   int i = 0;
   int found = false;
   while(!found && i < commandsLen)
   { //compare to known command strings
          if(!strcmp(commands[i], input))
          {
          found = true;
          }

          i++; //even once found we increment, as the integers for commands starts at 1
   }

   if(found)
   {
          msg.command = i;

          switch(i)
          {
          case STORE: //second argument will be file path
          if(scanf("%"MAXPATHLENGTHSTR"s", input) == 1)
          {
          if(readFile(&(msg.body), &(msg.length), input))
          {
                 valid = true;
          }
          else
          { //failed to load file
                 valid = false;
```

```
                printf("LOCAL Error: Failed to load file.\n");
        }
        }
        else
        { //no filename to read
        valid = false;
        printf("LOCAL Error: Filename required.\n");
        }

        break;
        case GET: //second argument will be key
        case DELETE: //falls through
        case HISTORY:
        if(scanf("%4096s", input) == 1)
        {
        msg.length = strlen(input);
        msg.body = calloc(msg.length + 1, sizeof(char)); //+1 for '\0'
        strcpy(msg.body, input);

        valid = true;

        if(i == GET)
        { //also need to get filename
                if(scanf("%"MAXPATHLENGTHSTR"s", input))
                {
                *fileName = calloc(strlen(input) + 1, sizeof(char));
                strcpy(*fileName, input);
                }
                else
                { //no destination file path to save file
                valid = false;
                printf("LOCAL Error: Destination filename required.\n");
                }
        }
        }
        else
        { //no key to read
        valid = false;
        printf("LOCAL Error: File key required.\n");
        }

        break;
        case QUIT: //no second argument
        msg.length = 1;
        msg.body = calloc(1, sizeof(char));
        msg.body[0] = '\0';
        valid = true;
        break;
        }
    }
    else
    { //unknown command
        valid = false;
        printf("LOCAL Error: Unknown command.\n");
    }
    }
    else
    { //no input
    valid = false;
    printf("LOCAL Error: Unknown command.\n");
    }

    fflush(stdin);
   }

  return msg;
}
```