Class: CSE 130
Professor: Faisal Nawab
Assignment 2
Fall 2020

Design Document: Back-up and Recovery in the HTTP server

Contributors:
Adriel Naranjo - adjnaran
Abbas Engineer - afengine

Objective & Synopsis

Modifying the HTTP server to have 3 additional features:

1) **Recovery** - The HTTP server will have the ability to store a backup of all the files in the server.

   To Recover and restore the most recent backup the request will be GET /r. This will allow all the files from the most recent backup to be restored from the backup folder.

   Another type of request is Get localhost:port/r/backup-[timestamp] where the httpserver will restore the backup with the provided timestamp.
   If no backup with that timestamp exists or there exists no backups at all:
   error 404 is returned.

2) **Backups** - The HTTP server will have the ability to recover files in the server using a backup copy of the server files. Each backup copy is named backup-[timestamp], where timestamp is the number of seconds since 1970.

   A backup is a copy of the HTTP server data, which consists of all the files that can be accessed with GET requests.

   To request to create a backup is to make a localhost:port/b request to the server. When the HTTP server receives a GET /b request, then it creates a new folder called "./backup-[timestamp]" where [timestamp] is the timestamp when the backup was created. Inside this new backup folder will be a copy of all the
   files (not folders) in the httpserver that are asset files, meaning they contain data the server wants to use.

3) **List** - When a Get /l is called (this is the small case of L), then the HTTP server will return to the client a nicely printed list of backups that exists in the server that can be used for recovery.

In this project we utilize the tools provided by C/C++ libraries which are listed below and should be looked into before considering the rest of the design. Emphasis on the following:
`<sys/socket.h>, <netdb.h>, <netinet/in.h>, <errno.h>, <sys/stat.h>`
`<stdio.h>, <stdlib.h>, <sys/types.h>, <fcntl.h>, <unistd.h>, <string.h>, <iostream>,<ctype.h>, <vector>, <dirent.h>, <climits>, <ctime>, <sys/dir.h>, <stack>, and <cstring>`

Initial Assumptions:

- The GET requests will have a slash in front of the special file name. For example, GET /b and GET /r
- The timestamp format is the Unix timestamp which is the number of seconds since Jan 01 1970. (UTC)
- Assume that there will be no read or write to *.c and *.o files.
- No multi-threading or redundancy testing
- No brackets in the timestamp

Let's get started:

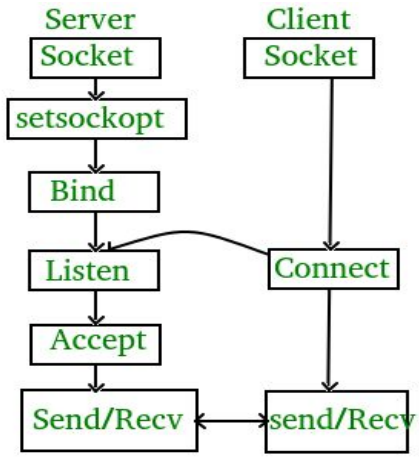

Diagram Citation: https://www.geeksforgeeks.org/socket-programming-cc/

To create a client-server model in C/C++ alone there are 3 major steps that need to be taken.

Additionally in order to make backups, recover, and list work, 3 extensions/modules were created in order to provide these features for the client. **CHECK MAJOR STEP 2 OF CREATING A SERVER TO SEE THE DESIGN OF backups, recover, and lists.**

Steps to creating server:

Major Step 1: create a root socket that can listen for incoming connections.
Get the information needed in order to create a host socket. A lot of references of this step can be found at: https://man7.org/linux/man-pages/man3/getaddrinfo.3.html. Other functions seen in this step can also be found in their own manpage(s).

```
root_socket_init()
            case 1:
                  preferred_port = argv[2];
                  // just checks if the port is all digits 0-9
                  check_alphabet(preferred_port);

            case 2: preferred_port = "80";

// Check the manpages for these,
//understanding these structures will help you
// create various types of sockets.
struct addrinfo *addrs, hints;
getaddrinfo(hostname,preferred_port, &hints, &addrs);

// initialize the root socket using socket()
root_socket = socket(AF_INET, SOCK_STREAM, 0);
// see link at beginning of step 1 to see how to do
Set_socket_attributes(root_socket);

//After validating the socket, put the socket to use by listening.
bind(root_socket, address);
listen(root_socket);
```

Major Step 2: accept the connection and interpret the request from the new connection (client) to the root.

```
// keeps the program state to listening
//so incoming client requests may come in
while(1){
      // where is address set using getaddrinfo & set_socket_attr
      New_connection = accept(root_socket, address);

      // read incoming message
      read(new_connection, buffer)
      // check manpages
      // sscanf extracts info from formatted messages from http
      sscanf(buffer, "%s %s %s", request, file, protocol);
```

Server listens for special request GET /b request

When the HTTP server receives a GET /b request, then it creates a new folder called `./backup-[timestamp]` the timestamp will be when the backup was created. This new backup folder will contain all the files in the current server directory.

```
      if(filestringname == "b"){
             // get the current time seconds since 1970 date
                int new_dir;
                time_t current_time;
                time(&current_time);

             // give the backup file a path to be reference at
                string time_as_str = to_string(current_time);
                string directory_path = "./backup-" + time_as_str;

             // create the directory and give it all permissions
                new_dir = mkdir(directory_path.c_str(), S_IRWXU | S_IRWXG |
S_IROTH | S_IXOTH);

                if(new_dir == -1){
                        perror("directory could not be created");
                }

             // save the backup file names in the vector recovery_files
                recovery_files.push_back(directory_path);

             // needed for parsing through server directory cwd
                DIR *directory = opendir((char*)".");
                struct dirent *direntStruct;

            if (directory != NULL) {
               // while there are directories or files in the main directory
                 while ((direntStruct = readdir(directory))) {

             // get the name of the file, the length of the name of the file
               string name = string(direntStruct->d_name);
               size_t name_length = name.length();

             // to see if the file is a backup, we dont want backups
                int found_position = name.find("backup");
             // sanity check so we only get files that are assets to the
            server
                if(check if the file is an asset){

             // create path for backup to pass to destination
               string destination = directory_path + "/" + name;
```

```
        // open source, open dest, read from server source to buf,  //then
        buff to backup.
        // continue for rest of files that are assets
        int src_fd = open(direntStruct->d_name, O_RDONLY);
        int dst_fd = open(destination.c_str(), O_CREAT | O_WRONLY, 00700);

        // You have all the info needed to do the read from source and
        write into destination. Do this how you please then close all the
        directories.

        close(src_fd);
        close(dst_frd)
```

recovery_files stems from its declaration: **vector<string>recovery_files;** found in the main function. This vector will be used in order to manage the List of backups that are recorded and allows the client to access this vector in order to recover a specific backup.

Server listens for special request GET /r request

The HTTP server restores the more recent backup. Therefore, all the files from the most recent backup folder will be copied to the current folder where the httpserver executable is. Another request is Get /r/[timestamp] where the httpserver will restore the backup with the given timestamp.

Using `DIR` to access the main directory. This struct help us see inside of the directory, So if the directory exists, then go into it and find the subdirectory and files.

```
if(filestringname == "r" || target_found == true){
// checks if there are recoveries, if not then not found
      if(recovery_files.size() == 0){
               close(new_socket);
               continue;
      }
      string str;
      // sets recovery file to most recent if not specified
         if(is_spec_recovery == false){
              str = recovery_files.back();
         }

// sets recovery file to specified backup name if the backup exists and
// it was requested
      else if(is_spec_recovery == true && target_found == true){
```

```
                    str = recovery_files.at(target_index);
            }
        // needed for parsing through server directory
            DIR *directory = opendir((char*)".");
            struct dirent *direntStruct;

            if (directory != NULL) {
               // while there are directories or files in the main directory
                 while ((direntStruct = readdir(directory))) {

             // get the name of the file, the length of the name of the file
               string name = string(direntStruct->d_name);
               size_t name_length = name.length();

             // to see if the file is a backup, we dont want backups
                int found_position = name.find("backup");
             // sanity check so we only get files that are assets to the
            server
                if(check if the file is an asset){

             // create path for backup to pass to destination
               string destination = directory_path + "/" + name;

            // open source, open dest, read from BACKUP source to buf,
            // then buff to SERVER DIRECTORY.
            // continue for rest of files that are assets
            int src_fd = open(direntStruct->d_name, O_RDONLY);
            int dst_fd = open(destination.c_str(), O_CREAT | O_WRONLY, 00700);

            // You have all the info needed to do the read from source and
            write into destination. Do this how you please then close all the
            directories.

            close(src_fd);
            close(dst_frd)
```

Server listens for special request GET /l request

```
// simply appends all backup names to a string then writes the string out as
// the response
```

```
string list_of_recoveries;
if(filestringname == "l"){
// checks if there are recoveries, if not then not found
          if(recovery_files.size() == 0){
               send not found
               close(new_socket);
               continue;
          }
for(unsigned int i = 0; i < recovery_files.size(); i++){
           string a_backup = recovery_files.at(i);
           list_of_recoveries += a_backup + "\n";
}
dprintf("HTTP/1.1 200 OK\r\nContent-Length: 0);
write(new_socket,list_of_recoveries.c_str(),list_of_recoveries.length();
close(new_socket);
continue;
}
```

Major Step 3: from the server side (server), process the request

Due to the restrictions of the file name we must first validate the file resources that meet the constraints of this project.

```
// This important check checks if this request is a special request or a
//normal serve request like get or put
If (file != "b" && file != "r" && file != "l")

      If the file is longer than 10 characters long:
            Response 400 Bad request
      If the file contains any special characters, non-alphabet, non-numeric
            Response 400 Bad request
```

Next, since we know we have a valid file resource name we must check if the request type is either a PUT or a GET. Once this information is obtained we can continue responding depending on the context.

```
      If GET:
            Create buffer to read from file if opened successfully
            get_file_info () // use stat() check manpages
            open(file);
            If permission to open file is not granted
                  Response forbidden 403
            If file does not exist in server directory
```

```
            Response not found 404
        If file has permissions and exists
            Create 200 OK Response
            read file contents from the file directory into the buffer.
            Then write buffer contents to the socket to give the client
            the requested data.
    If PUT:
        Retrieve information about the content so we can use it to create
        the response.
        Create a buffer for data transfer.
        // this is the file that exists in the server to be
        open(file); written

        // information to be PUT into file is received into the buffer
        recv(new_socket, buf)
        // write the information into the file that exists on the server
        write(fd, buf)
    If None:
        Response 400 Bad request
    close(new_connection)
```

} // **NOTICE:** this while loop start in **step 2** and ended in **step 3**

Design Questions to keep in mind:

- How would this backup/recovery functionality be useful in real-world scenarios?
    - The concept of Backups and Recovery is commonly used in many real world situations. One situation that has occurred during the development of this server. When working with Gitlab and with another partner, the functionality of backups and recovery can be the key to restoring potentially corrupted repositories. We had encountered a strange when pushing or pulling from the repository:
      `error: object file .git/object/commitID is empty`
      `Fatal: commitID is not a valid object`
      When the repository is corrupted like this we would want to **Recover** our repository by cloning from a remote repository if there weren't many local changes. In order to do so first a **Backup** of the .git directory is required. Git has its own implementation of backups and recovery that allow the users to access what is essentially a **List** previous commits and revert back to them. Git provides something called a "Rake Task" which is a software task management and build automation tool to backup and restore gitlab instances. Similar to our project, the application data backup creates an archive file that contains all the repositories and attachments.

Testing Issues:
Through modularization, unit testing was the most effective way to debug. We created each of the modules independent of the httpserver then moved over the code and patched it into the server. This method allowed us to test smaller pieces of code and reduced the time spent for debugging.

- Discuss how you Create multiple backups and recovery to the most recent one or an earlier one.
    - Whenever and however many times the special request is called for multiple backups, each backup will sequentially be added to the recovery_files vector, which essentially also acts as the list of all backups.
    - The vectors repertoire of functions made it easy to access the most recently created backup with a call to vector.back() which returns said backup.
    - Now if the user wants a specified backup then the vector of backup names makes it easy to also do a look up for a backup directory that may exist in the server.
    - Finally using this way of managing backups allows the client to be given a list to see what backups have been made and then they can call to restore any backup based the backup-timestamp Id.

Conclusion:
Throughout the Design implementation, we wanted to incorporate as much modularity as possible, given 3 functionalities to implement. As development persisted, **Backups** and **Recovery modules** were seen to be very similar in programmatic & algorithmic structure. One key difference is that with backups we are physically opening a new directory, backup-[timestamp] and for recovery there is nothing to be actually created.
The other key difference was that just the source of where we get the data and the destination to where the data is written differs entirely as well. For backups, we are always copying the asset files from the server directory into a backup. With recovery, the files are being copied from the recovery directory to the server directory.