COP 290 Assignment 2

DeadDrop

Faran Ahmad
2013CS10220

Kartikeya Gupta
2013CS10231

Prateek Kumar Verma
2013CS10246

February 2015

Contents

1 Objectives

We have to build an on-line file management system "Dead Drop" . A server machine maintains the files of multiple users. The user should use a simple desktop application to login into the system. The content of user's account should remain synced with the server.

2 Overall Design

1. We will begin with creating different sub components like a File Transferring System, Credential Verifier, GUI part.
2. Once the components are ready, we will Link this to the network and get basic functionality working on the local-host.
3. Once the local interface is ready, we will take this to the web portal. We will use a server to store data and users will have to send queries to it
4. Once the backend and front end is complete, we will link the two together.

3 Sub Components

3.1 User Verification

Listing 1: Class Parameters for User

```
1 class User
2 {
3     private:
4         std::string UserName;
5         std::string PassWord;
6 };
```

Listing 2: Class Parameters for UserBase

```
1 class UserBase
2 {
3 private:
4     std::unordered_map<std::string, std::string> UsersList;
5 };
```

The User Base is a hash table in which the keys are user-names and the stored values are passwords. When the credentials of the user are to be verified, the key is looked up in the table. Inserting users is also achieved easily using this model. The features which we will be provided to the user will be to verify credentials, add new users and change password. On the server, the credentials will be stored in an encrypted file which will be decrypted by the server program.

3.2 Files of User

We will use boost library to detect changes in files. For each file, the path of the file and last modified time of file is stored in a database.

Listing 3: Class Parameters for File History

```
class FileHistory
{
    private:
        std::string FolderLocation;
        int TimeOfData;
        std::vector< std::pair<std::string, int> > FileTimeBase;
};
```

Folder Location is the path of the synced folder. The parameter "TimeOfData" contains the system time at which the data detection was done. This will be used to determine if the server or client side file is newer and then do changes accordingly. "FileTimeBase" is a vector of a string and an integer. The string is the path of the file and the time is the time at which the file was last modified.

3.3 Network Managing Part

3.3.1 Basic Networking

We will use the Transmission Control Protocol (TCP) to design and implement the network aspect of the assignment. It accepts data from a data stream, divides it into chunks, and adds a TCP header creating a TCP segment. The TCP segment is then encapsulated into an Internet Protocol (IP) datagram, and exchanged with peers. A TCP connection is managed through a programming interface that represents the local end-point for communications, the *Internet Socket*. Sockets allow applications to communicate using standard mechanisms built into network hardware and operating systems. In a nutshell, a socket represents a single connection between exactly two pieces of software. Sockets are bidirectional, meaning that either side of the connection is capable of both sending and receiving data.
We will use *getaddrinfo()* for obtaining domain name system (DNS) hostnames and IP addresses, which are required to create a socket. This function deals with all the complicated structs. It also uses a new struct called called *addrinfo*.

Listing 4: The structure "addrinfo"

```
struct addrinfo {
    int ai_flags;
        // This field specifies additional options.
    int ai_family;
        // IPv4, IPv6 or IP agnostic.
    int ai_socktype;
```

```
7           // TCP or UDP.
8       int ai_protocol;
9           // The protocol for the returned socket addresses.
10      size_t ai_addrlen;
11          // Size of ai_addr in bytes
12      struct sockaddr *ai_addr;
13          // Containing the IP address and port.
14      char *ai_canonname;
15          // The Canonical hostname.
16      struct addrinfo *ai_next;
17          // linked list, next address.
18  };
```

This function and the struct addrinfo are all we need to create and connect a socket.The function prototype looks like this :

Listing 5: getaddrinfo() function

```
1  int getaddrinfo(const char *node, const char *service,
2      const struct addrinfo *host_info, struct addrinfo **res);
```

The parameters are:

- **node** : The host we want to connect to. This can be a hostname or IP address.
- **service** : This is the port number we want to connect to. Usually an integer, but can also be a known service name like 'http'.
- **host_info** : Points to the addrinfo struct to fill.
- **res** : Points to the linked list of filled addrinfo structs.
- **return value** : The function returns 0 if all succeeded or a non-zerro error code in case of an error.

To create a socket we use the *socket()* system call

Listing 6: socket() function

```
1  int socket(int domain, int type, int protocol);
```

The parameters are:

- **domain** : The domain argument specifies a communication domain. In our case this value is AF_INET or AF_INET6 (the Internet using ip4 or ip6).
- **type** : The type of socket. In our case it is SOCK_STREAM (tcp).
- **protocol** : The protocol to be used with the socket-type. In our case the right protocol is automatically chosen.

- **return value** : The socket system call returns a socket descriptor. If the socket call fails, it returns -1.

To connect from client to the server, we will use the *connect()* system call

Listing 7: connect() function

```
1 int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

The parameters are:

- **sockfd** : the socket descriptor the socket() call returns.
- **addr** : The address we need to connect to. In our case stored in 'host_info_list → ai_addr'.
- **addrlen** : The addrlen argument specifies the size of addr. In our case stored in 'host_info_list→ai_addrlen'.
- **return value** : If the connection succeeds, zero is returned. On error, -1 is returned, and errno is set appropriately.

To accept connections from a client, we will use system calls : *bind()*, *listen()* and *accept()*.

Listing 8: bind(), listen() and accept() function

```
1 int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
2 int listen(int sockfd, int backlog);
3 accept(sockfd, (struct sockaddr *)&their_addr, &addr_size);
```

The parameters are:

- *sockfd* : The socket descriptor the socket() call returns.
- *addr* : The address we want to listen on (localhost).
- *addrlen* : The lenght of this address.
- *backlog* : Our server can only handle 1 client at a time. What if more clients want to connect to our server at the same time? With backlog we can specify how many connections will be put in que. For example, if we set it to 5, and 7 connections to our server are made, 1 will fail, 1 will connect and the other 5 will be put"on hold".
- *their_addr* : their_addr will usually be a pointer to a local struct sockaddr_storage. This is where the information about the incoming connection will be stored (Like the client's IP address and port).
- *addr_size* : addr_size is the size of the their_addr struct.

To send and receive messages, we will use the system calls *send()* and *recv()*.

Listing 9: send() and recv() function

```
send(int sockfd, const void *buf, size_t len, int flags);
recv(int sockfd, void *buf, size_t len, int flags);
```

The parameters are:

- *sockfd* : The socket descriptor the socket() call returns.
- *buf* : The message we want to send/receive.
- *len* : The length of this message. Equals strlen(buf).
- *flags* : Without this parameter, this call would be the same as the write() call. This parameter gives us some extra options.

3.3.2 Encryption of Data Transfer

We will use Secure Sockets Layer (SSL) to implement and design a secure connection transfer. It is a cryptographic protocols designed to provide communication security over a computer network. To implement this protocol, we shall use the *OpenSSL* library, which is an open-source implementation. They use X.509 certificates and hence asymmetric cryptography to authenticate the counterparty with whom they are communicating, and to exchange a symmetric key. This session key is then used to encrypt data flowing between the parties.
The initialization is done using the following:

Listing 10: Initialization

```
SSL_CTX* InitServerCTX(void)
{
    SSL_METHOD *method;
    SSL_CTX *ctx;
    OpenSSL_add_all_algorithms(); // load & register all cryptos, etc.
    SSL_load_error_strings();     // load all error messages
    method = SSLv3_server_method(); // create new server-method instance
    ctx = SSL_CTX_new(method);    // create new context from method
    return ctx;
}
```

We shall make the following functions to implement loading and showing the X.509 certificates.

Listing 11: X.509 Certificates

```
void LoadCertificates(SSL_CTX* ctx, char* CertFile, char* KeyFile)
void ShowCerts(SSL* ssl)
```

3.4 GUI interface

The interface of the application would be designed using QtCreator. It would consists of the following windows.

- **User Login**
  Running the application would display a user login window. The users can access into their accounts by entering their user name and password in the respective fields. It also provides options for new user to sign-up for a new account in dead drop. In case, the user forgets his password, he can reset it using 'forgot password' option. If the user name or password entered is wrong, a message box showing this message is displayed and the user can again enter the required informations to login to his account.

- **User Files**
  Once the login procedure is complete, the user is directed to a new window. It contains the list of files of the user, both on the client and server side. The following buttons would be provided on the window for various functions

  - *sync :-* Clicking this button would sync the files on the user and the client side.
  - *Share :-* This button allows the user to share files with other users. Once the user has selected the files, clicking on this button would open a new dialog box, where the user can enter the name of the ones, with whom he wants to share the selected files. It also provide users with the options to set permissions for the shared files and folders.
  - *Delete from local :-* This button can be used to delete the selected files on the client side.
  - *Move To Drive :-* It can be used to move the selected files on the client side to the server.
  - *Get :-* This button can be used to transfer the selected files on the server to the client side.
  - *Delete from Server :-* This button can be used to delete the selected files on the server.
  - *Open File :-* This button will allow the user to open the selected file or folder by the default application.
  - *Add File :-* This button will allow the user to add a file from his PC to the syncing folder.
  - *Exit :-* This button can be used to exit the application.

- **Server Side User Interface**
  The window on the server side displays the list of all on-line users. It also provides the following buttons for different functions

- *View All Users :-* It can be used to display the list of all the users.
- *Remove A User :-* Clicking this button would open a new dialog box, displaying the list of the users. The users to be deleted can be selected from here for this operation.
- *Files and Folders :-* It can be used to view all the files and folders stores on the server.
- *Shut Down :-* This would shut down the server side window.

4 Interaction amongst Sub Components

4.1 User Authentication

- **Client**
  The user will enter their credentials in the provided text boxes. These text boxes belong to the class *LineEdit*, which contains a function to extract *QString* and then converted to *string*. Now, this string is transfered from the client to the server over a secure connection using *OpenSSL*.

- **Server**
  The server receives the credentials one by one over a secure connection using *OpenSSL*. The network part mentioned above is linked with the user base file. The instruction to be performed is decoded to be a new user or credential verification. The data base of user names and passwords are accessed for this to take place and changes if needed are made accordingly to it.

4.2 File Transfer

- **Files to Transfer**
  The files which have to be transferred can be new files created on the client, new files on the server, modifications on the client, server or file getting removed from the client or server. For each of these an instruction will be given to the transferring mechanism which will execute the instructions to perform the sync operation.

- **Transferring**
  Using the absolute path of a file, the file is opened using *ifstream class*. The file is opened in chunks of size 10 MB, thus file of any size can be opened and transfered. Now, each of these chunks are split into 100 KB of *char array* and transfered over a secure connection using the network part mentioned above.
  The file is also received over a secure connection. It is received in chunks of 100 KB and then combined to create 10 MB parts. We wish to store the files in parts of 10 MB to tackle the problem of "de-duplication".

4.3 File Sharing and Syncing

- **File changes**
  To detect changes in files we will use the last modified time stamp of the file. If the files differ in time stamp and are linked together then the newer file is considered to be a more recent version.
  Once this is done we will work on version control which will allow us to consider the differences in files and only sync them.
- **File Sharing and Permissions**
  For each file on the server the default property is user only. If the user wishes to share a file with another user then he can give read or read and write permissions to him.
- **File Syncing** This will involve use of all of the above mentioned properties. A set of instructions will be given to the network part which will then execute them to get the client and server in sync.

4.4 Front End and Back End

To link the front end and back end of the program, we will associate all of the buttons and actions in the GUI part with methods in the back end.

5 Testing Of Components

5.1 General Unit Tests

Listing 12: Class Parameters for Test

```
1 class Test
2 {
3 private:
4     bool verbose;                //If test is to be conducted
5     std::string description;     //String description of the test
6     bool isPass;                 //Boolean if the test has passed
7     void PrintPassFail(bool);    //Prints the status of the test
8 };
```

We will use the aforementioned class "Test" to perform unit tests on the different files created. This will ensure that all the functions work correctly against some tests.

5.2 File Discovery

To test file discovery, a folder with different files will be used. The program will be run on this to obtain the list of files with their modified time and verified to check if it is in accordance with expectations. This will involve new files being created, files being modified and removed.

5.3 File Transferring

To test file transferring, we will transfer 'test files'. First, we shall transfer a small file of size lesser than 100 KB to ensure proper transferring and receiving of each chunk. Then, we shall transfer a file of size lesser than 10 MB to ensure all the chunks are transferred properly. After doing that, we will transfer files of arbitrary large size and to ensure proper working of all the components. We will use the system command 'diff' to check whether the transfered file and the receive files contain the same contents.

5.4 UI Testing

- **User verification Testing**
  To test user verification part, new users would be created in the back end. Then, their validity would be tested by typing their user name and password in their respective fields to see if they are able to login to their account. The output of user name and password entered would also displayed on the terminal to check our input manually.

- **New User Testing**
  To test the new user sign-up part, new users would be added using sign-up button on the main window. The entered informations would then be checked in the back end by seeing the list of all the users and their informations.

- **File Managing Part**
  The user files part would be checked manually by comparing that list of files of a user, displayed in his account is same as the list of files that can obtained by browsing into the user folder. Similarly, list of files of a user on the server side being displayed in his account should be same as the one obtained by browsing into the server.

5.5 Overall Testing

Once all of the components are tested individually and assembled together, we will perform all possible actions on files and check if the sync is taking place as expected. We will compare the size of the files on the clients and on the server to check if de-duplication is working correctly. We will share files between users with both possible permissions and check if the permissions are getting reflected appropriately.

6 Extra Features

6.1 Keeping Files On-line only.

We will give the user an option to keep a file on the cloud only and not on his PC so that lesser space will be used on the users PC. The user can get these

files or remove them from the cloud if he wishes.

6.2 Intelligent UI for server

The server UI will display the list of users currently signed in, the amount of data being sent, received, total size of data stored, total number of users etc.

6.3 Allowing Incomplete Downloads

Allowing the user to resume downloads of files from where he left off if the connection with the server breaks in the middle.

6.4 De-duplication

Dividing the files on the server into smaller files and hashing them will allow us to perform de-duplication of data and save space on the server.

6.5 MultiThreading

Each server-client connection will be handled by a different thread, thus enabling more than one client connecting to the server at the same time.