Brno University of Technology

Faculty of Information Technology

POP3 client with the TLS support

Vladislav Sokolovskii [xsokol15]

October 23, 2021

Contents

1 Introduction

The main goal of this project was to develop a client type application which will download e-mails from the given server via the POP3/POP3S protocol. The resulting application supports all the required options and fully corresponds to the task.
`Usage: popcl <server> [-p <port>] [-T|-S [-c <certfile>] [-C <certaddr>]] [-d] [-n] -a <auth_file> -o <out_dir>`
`Compulsory arguments: <server>, -o, -a`

The program will handle any acceptable order of arguments.

- The client will try to connect to the specified `<server>` on the specified `<port>`, if port is not specified the default one is used (110 for the unsecured and 995 for the secured communication). The default port is chosen according to the connection security level.
- If `-T` flag is set then the whole communication is encrypted from the beginning and port 995 is chosen.
- If `-S` flag is set then the unencrypted communication on server port 110 starts, communication will be switched to the encrypted version once the connection with the server is established.
- In case when neither `-T` nor `-S` flag is set then the client will try to initiate the unsecured communication on the port 110, however most of the mail servers do not support this type of communication due to the security reasons.
- During the establishment of secure connection server certificates are being verified, user can specify the certificate location using `-c and -C options`, where `-c` specifies the particular file containing certificates and `-C` specifies the folder where files with certificates can be found.
- If connection to the server was successful then message retrieving process starts.
- If `-d` flag is set then all the messages downloaded withing the current session will be deleted from the mail server.
- If `-n` flag is set then only the new messages will be downloaded. Here you can read more about how this parameter is implemented.
- `auth_file` containing the credentials is expected in the particular format.
- `out_dir` where the downloaded e-mails will be saved **must exist beforehand**.

2 Implementation

2.1 Overview

The entry point of the application is `popcl.cpp` file, which contains the `main()` function where the two key methods of the application are called. Function `args_parse()` parses the command line arguments and stores the given options to the `ArgumentsParser` private class attributes. Subsequently method `set_tcp_connection()` is called, this is the core method of the application. It establishes the TCP connection according to the provided options, calls method which starts retrieving messages from the server and stores them to the specified folder.

2.2 Arguments parsing

CLI application `popcl` accepts arbitrary order of the arguments. Initially `getopt()` function was used but within the development process it was decided to write a parsing function `args_parse()` from scratch. This function checks whether the command line arguments are in the acceptable order and checks if all the required arguments were provided.

2.3 Connection establishment

Connection establishment was the second thing to implement, the main source of information during this phase was a tutorial provided in the assignment [5]. `OpenSSL documentation` was a good source of information regarding some particular `BIO/SSL` functions [3].

According to the given options the client will try to connect to the port number `110` in case of the unencrypted connection or to the port number `995` in case of the encrypted connection, if port is not explicitly specified by the user.

2.3.1 Unsecured connection

In case of the unsecured connection the process of connection establishment is quite trivial. In the beginning `bio` object is created (`BIO_new_connect()`) which is then used for connection and communication with the server. After `bio` object is successfully created communication with the server starts.

In case of the **unencrypted connection with the subsequent transition to the encrypted communication** (`-S option`) the client establishes the unsecured connection with the server as described above and sends `STLS` request to the server. If the answer is positive (`+OK`) then TLS negotiation begins immediately and `init_context()` function is called. This function initializes all the `BIO` and `SSL` objects which are necessary for the secured communication and verifies the server certificates. In comparison to the implicit SSL connection where `BIO_write/BIO_read` functions were able to process the encrypted data, the encrypted connection established via the `STLS` request could not be handled by the `BIO_write/BIO_read` functions. Therefore `SSL_write/SSL_read` function were used in this case.

2.3.2 Secured connection

In order to establish SSL communication `init_context()` function is called and secured `bio` object has to be created (`BIO_new_ssl_connect(context)`). Then some auxiliary function from the `OpenSSL` library are called. After successfully verifying the server certificates encrypted communication with the server can start. User can set where the file(s) with the certificates should be looked for (`-c <particular_file>`, `-C <folder>` options), if not specified the default path is used (`SSL_CTX_set_default_verify_paths()`).

2.4 Authorization

To retrieve the credentials `parse_auth_file()` function is called. This function expects the file containing the username and the password on the first two lines, according to the assignment.

```
username = test-user-name
password = 12345
```

The application considers any unexpected file format as invalid and terminates with an error.

If everything was as expected the tuple of strings is returned. Afterwards username and password are passed to the `authorize()` function call. Here the client-server communication starts according to the POP3 protocol standard[2]. Authorizing function sends two requests to the server containing username and password and analyzes the response. If credentials are valid and server returns the positive response communication proceed.

2.5 Retrieving only new messages

It is a good time to find out how the `-n option` is handled. It was decided to create a hidden file called `.oldmails` in the same directory where `popcl` binary file is located. This file will store message identifiers of all downloaded via the *popcl* e-mails, one message-id per line.

Every time after successful login to the server the `.oldmails` file is loaded to the global `std::map<std::string,bool>` structure (`load_old_mails_map()`). This data structure **will enable checking** if an e-mail with the particular message-id was downloaded before **in the constant time**.

In order to retrieve the message-id `get_message_id()` function is called. This function send `TOP n 0` request for every message and then retrieves the message-id via the regular expression.**This is much more efficient** then waiting for the whole message content to be downloaded. Message identifiers of all successfully downloaded messages are stored to the `std::map<std::string,bool>`.

If the program execution will be interrupted by `SIGINT` or `SIGTERM` signals the downloaded messages will be successfully stored to the folder specified by the user and `store_old_mails_file()` function will be called from the signal handler [1]. This function saves all the map items (message ids) to the `.oldmails` file in order to make them accessible during the next execution. Message identifiers of the downloaded messages are stored despite of the fact if `-n` flag was set or no. Therefore during the next execution with the `-n` flag set the messages will not be downloaded again.

The possible downside of this approach could be that in case when the `popcl` binary will be moved to another directory the history of downloaded messages will be lost.

2.6 Message retrieving

After loading `.oldmails` file to the map structure and checking if there are any messages to download (`get_number_of_emails()`) the message retrieving process starts. First of all, it sets the signal handlers and then it will loop through all the messages in the inbox.

The client checks message-id of every message it processes without downloading the whole content (`is_email_old`). If `-n` flag is set then all messages that were downloaded before will be skipped and only new messages will be downloaded (`RETR n`), otherwise the client will send `RETR n` request for every message and store its content to the file called `mail-n`. The message content arrives by chunks, so the `get_response()` waits for the concluding period (.) at the end of the message, then it returns the message content as a `std::string`. The final period and the first line containing (`+OK ...`) are deleted from the message content. The final stored message corresponds to the Internet Message Format [4].

If the `-d` flag is set then every successfully downloaded message will be marked as deleted on the server (`DELE n`), **however** the deletion will happen only if the connection will be successfully finished via the `QUIT` request.

After retrieving all messages `store_old_mails_file()` function is called to store the identifiers of the downloaded messages to the `.oldmails` file. Then `QUIT` request is sent to the server. The application will provide user with the number of actually downloaded messages (not counting the old ones).

2.7 Application compatibility

Great emphasis was placed on the program compatibility. The main implementation phase was carried out on the `macOS Catalina 10.15.7` and the resulting application is fully compatible with the operating systems with Darwin kernel. The testing phase was carried out on Linux and FreeBSD, the application is fully functional on all of these platforms. It was achieved thanks to this source, CMake smart built-in functions and parameters which can easily adjust the building process for every platform.

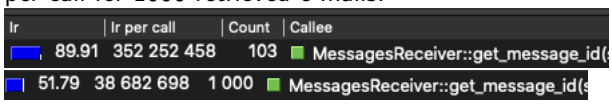
3 Testing

The application was tested on Darwin, Linux and FreeBSD. Since project is quite small it was decided not to implement any kind of unit tests. All the testing was performed manually.

The project was tested on the following e-mail servers:

- eva.fit.vutbr.cz
- pop.gmx.net
- pop.safe-mail.net
- pop.gmail.com (security complications)
- local POP3 server

After the first phase of the implementation when the number of e-mails was >100 and some of them had images or video attached the application worked very slow. According to the `callgrind` tool the majority of instructions executed per call fell under `get_message_id()`. In the top image you can see that the number of instructions executed per call for 103 retrieved messages was more than 352 millions, after the code refactoring and optimization it became possible to achieve 38 682 698 number of instructions per call for 1000 retrieved e-mails!



The final application is almost 88 times faster than the first fully functional working version. It was achieved due to the usage of `POP3 TOP` command for the message-id retrieval and making the `std::regex` static.

3.1 Load test

Downloading **1156 e-mails** from the school mail server via the secured connection took only **38 seconds**! Worth mentioning that some of these messages have images or pdf documents attached. The testing was performed on the `merlin` school server.

```
./popcl -T eva.fit.vutbr.cz -a ./eva_auth -o ./out_mail
1156 e-mails were downloaded
38s
```

Listing 1: bash version

References

[1] *C++ Signal Handling*. URL: https://www.tutorialspoint.com/cplusplus/cpp_signal_handling.htm.

[2] J. Myers. *Post Office Protocol - Version 3*. RFC 1939. RFC Editor, May 1996. URL: https://datatracker.ietf.org/doc/html/rfc1939.

[3] Inc. OpenSSL Foundation. *OpenSSL*. URL: https://www.openssl.org/.

[4] Ed. P. Resnick. *Internet Message Format*. RFC 5322. RFC Editor, Oct. 2008. URL: https://datatracker.ietf.org/doc/html/rfc5322.

[5] *Secure programming with the OpenSSL API*. URL: https://developer.ibm.com/tutorials/l-openssl/.