Poshet(A)

COZLOSCHI LUCA-MATEI

January 15, 2024

1 Introduction

The purpose of this project is to develop a network-based e-mail service application, in accordance with the Post Office Protocol Version 3 (POP3) standard. Additionally, this application provides support for file transfers, adhering to the Multipurpose Internet Mail Extensions (MIME) standard, an essential aspect of modern communication.

2 Applied Technologies

For this project, the Transmission Control Protocol (TCP) was chosen as the primary mode of data transmission, aligning with the requirements of the POP3 standard. The decision to use TCP over the User Datagram Protocol (UDP) stems from several key factors:

1. **Reliable and Ordered Data Transfer:** TCP provides a reliable channel of communication, ensuring the integrity and sequential order of data packets. This is crucial for an e-mail service where the accuracy and completeness of message delivery are paramount. In contrast, UDP, while faster and used in applications like online gaming, does not guarantee packet order or integrity, making it less suitable for a service where reliability is a priority.

2. **Error Checking and Correction:** TCP's built-in mechanisms for error detection and correction ensure that any data corruption during transmission is identified and rectified. This aspect is particularly important for e-mail communications, which often involve the exchange of important information where errors can have significant consequences.

Concurrent Server Implementation: To replicate a real-life e-mail service environment, we implemented a concurrent server that employs child processes to interact with multiple clients simultaneously. This architecture ensures that no client experiences unnecessary delays in service, as each client is handled by a separate process.

3 Application Structure

The application structure is represented by the following sequence diagram, which illustrates the interaction between the client, server, and other components of the e-mail service (jump to the last page of this document)

User Current Implementation: The e-mail service application is structured to enable essential functionalities upon successful user authentication. The login mechanism is crucial, as it gates access to e-mail specific commands and operations such as:

Sending E-mails: Users can compose and send e-mails, with the option to attach files,if all the fields are passing the validation process. E-mail Listing: Users can view lists of sent and received e-mails, which can also be organized and managed. If the user wants to search for a specific data, he can use the search box to input the string that should be included in the e-mail content.

Forward and Reply: Users can reply to and forward e-mails. Attachment Download: Users can download specific attachments associated with their e-mails. This method facilitates the demonstration of the e-mail service's capabilities in a controlled environment.

Dedicated Databases: Dedicated database for storing e-mails.

Attachment Storage: Implemented a storage system for attachments, where each file is be stored.

Graphical Interface Interaction[QT]: Enhancing the user experience with a graphical interface that allows for intuitive interaction with the e-mail service, using QT. For instance, clicking an attachment button will initiate the DOWNLOAD attachmentname.extension command, saving the file locally, in the .../Downloads folder.

Security: Access to the system's databases is exclusively reserved for server operations. Clients can only request and interact with information that is pertinent to their user account, such as their own e-mails and attachments. This maintains a secure boundary between user operations and the underlying data storage.

4. Implementation Aspects

This section highlights specific and innovative sections of the code that contribute to the project's functionality. It includes documentation of the implemented application-level protocol and describes real-world usage scenarios.

E-mail Listing

The e-mail listing function enables users to view their received emails in a list format. This functionality is essential for managing large volumes of e-mails efficiently. The following code snippet demonstrates how the server processes a request to list received e-mails:

```
//server
else if (strncmp(msg, "LIST_RECV", 9) == 0) {
    if (is_user_logged_in(username)) {

            sqlite3 *db;
            sqlite3_stmt *stmt;

            if (sqlite3_open("email_database.db", &db)) {
                strcpy(msgrasp, "-ERR Eroare la deschiderea bazei de date\n");
            } else {
                std::string sql = "SELECT ID, TITLE, EXPEDITOR FROM emails WHERE DESTINATAR = ? ;";
                sqlite3_prepare_v2(db, sql.c_str(), -1, &stmt, NULL);
                sqlite3_bind_text(stmt, 1, username, -1, SQLITE_STATIC);

                std::string response;
                while (sqlite3_step(stmt) == SQLITE_ROW) {
                    response += "ID: " + to_string(sqlite3_column_int(stmt, 0)) + "\n";
                    response += "Title: " + string(reinterpret_cast<const char*>(sqlite3_column_text
                    response += "From: " + string(reinterpret_cast<const char*>(sqlite3_column_text(
                }
                if (response.empty()) {
                    strcpy(msgrasp, "Nu exista e-mail-uri primite pe aceasta pagina.\n");
                } else {
                    strcpy(msgrasp, response.c_str());
                }

                sqlite3_finalize(stmt);
                sqlite3_close(db);
            }

    } else {
        strcpy(msgrasp, "-ERR Logheaza-te mai intai.\n");
    }
    write(client, msgrasp,strlen(msgrasp));
}
```

The server reads the appropriate e-mail entries from e-mails database, ensuring that only the e-mails that are retrieved by the client are sent back .

```
//client
void SecondWindow::on_button_received_clicked()
{
        const string request = "LIST_RECV";
    write(sd, request.c_str(), request.length());

        char buffer[20000];
        ssize_t bytesReceived = read(sd, buffer, sizeof(buffer) - 1);
        if (bytesReceived > 0) {
            buffer[bytesReceived] = '\0';
        }            QString serverResponse = QString::fromUtf8(buffer);

    processServerResponse(serverResponse,false);
}

void SecondWindow::processServerResponse(const QString& response, bool isSent)
{
    ui->listWidget->clear();

    QStringList emails = response.split("\n\n", Qt::SkipEmptyParts);
    for (const QString& email : emails) {
        QStringList details = email.split("\n");
        qDebug() << "Email details:" << details;

        if (details.size() > 2) {
            QString id = details.at(0).split(": ").at(1);
            QString title = details.at(1).split(": ").at(1);
            QString from_or_to = details.at(2).split(": ").at(1);

            QString itemText = QString("ID: %1 Title: %2 %3: %4")
                                   .arg(id)
                                   .arg(title)
                                   .arg(isSent ? "To" : "From")
                                   .arg(from_or_to);

        QListWidgetItem* item = new QListWidgetItem(itemText);
        ui->listWidget->addItem(item);
    } else {
        qDebug() << "Email format is incorrect, skipping";
    }
}
}
```

This client-side code effectively handles the communication with the server, processes the received data, and updates the UI to present the email list in an organized and readable format. This facilitates easy viewing and management of emails for the user.

3.1 Downloading a file from E-mail

In this system, we have a server-client architecture for downloading files. The server listens for download requests and sends the requested file to the client, which then receives and saves the file locally.

```
//server
else if (strncmp(msg, "DOWNLOAD", 8) == 0) {
    char file_name[100];
    sscanf(msg, "DOWNLOAD %s", file_name); // Extract file name from command

    char file_path[200];
    sprintf(file_path, "/home/kali/Desktop/attachments/%s", file_name); // Path where the file is st
    FILE *file = fopen(file_path, "rb");
    if (file == NULL) {
        perror("File not found or unable to open");
        strcpy(msgrasp, "ERROR: File not found or unable to open.\n");
        write(client, msgrasp, strlen(msgrasp));
        continue;
    }


    // Sending file content
    char buffer[1024];
    int bytes_read;
    while ((bytes_read = fread(buffer, 1, sizeof(buffer), file)) > 0) {
        send(client, buffer, bytes_read, 0);
        printf("bytes:%d\n",bytes_read);
    }

    // Send end-of-file marker
    const char* end_of_file_marker = "EOF";
    send(client, end_of_file_marker, strlen(end_of_file_marker), 0);

    fclose(file);
    printf("File has been sent.\n");
}
```

The server code handles a download request by extracting the file name from the message, locating the file, and sending it to the client in chunks. If the file is not found, an error message is sent back. After the entire file is transmitted, an ”EOF” (end-of-file) marker is sent to signal completion.

```
//client
    void EmailDetailWindow::on_downloadButton_clicked()
{   // Trimite comanda de download la server
    const string request = "DOWNLOAD " + attachment.toStdString();
    write(sd, request.c_str(), request.length());

    char file_name[100];
    strncpy(file_name, attachment.toStdString().c_str(), sizeof(file_name));
    char file_path[200];
    sprintf(file_path, "/home/kali/Downloads/%s", basename(file_name));


    FILE *file = fopen(file_path, "wb");
    if (file == NULL) {
        qDebug()<<"Error opening file for writing";
        return;
    }


        char buffer[1024];
    int bytes_read;
    bool end_of_file = false;
    while (!end_of_file && (bytes_read = recv(sd, buffer, sizeof(buffer), 0)) > 0) {
        if (bytes_read >= 3 && strncmp(buffer + bytes_read - 3, "EOF", 3) == 0) {
            // Scrie totul in fisier in afara de EOF
            fwrite(buffer, 1, bytes_read - 3, file);
            end_of_file = true;
            qDebug()<<"bytes final:"<<bytes_read-3;
            break;
        } else {
            fwrite(buffer, 1, bytes_read, file);
            qDebug()<<"bytes: "<<bytes_read;

        }
    }
    bool downloadSuccessful = false;
    if (file != NULL && end_of_file) {
        downloadSuccessful = true;
    }

    fclose(file);
    if (downloadSuccessful) {
        ui->downloadButton->setText("Succes!");
    } else {

        ui->downloadButton->setText("Failed.");
    }
}
```

On the client side, upon clicking the download button, a download request is sent to the server. The client then receives the file in chunks, writing it to a local file. It checks for the ”EOF” marker to determine the end of the file. Finally, the download success is indicated on the UI.

The described functionalities showcase the server's ability to handle typical e-mail operations, adhering to standard protocols.

5. Conclusions

This project is successfully representing a functional e-mail service that aligns with the POP3 standard and provides MIME support for attachments. The current implementation serves as a proof of concept for the core features such as user authentication, e-mail transmission, listing, opening, and attachment handling.

Looking ahead, the following enhancements are proposed to elevate the service to a more sophisticated and secure level:

- Implementing an **encrypted file storage mechanism** for attachments, ensuring privacy and protection against unauthorized access.

The ultimate goal is to ensure that while the service evolves in complexity and utility, it remains secure, and in compliance with established e-mail communication protocols.

6. Bibliography

RFC 1939 - Post Office Protocol - Version 3. Available at: https://www.ietf.org/rfc/rfc1939.txt

Wikipedia contributors. *MIME - Wikipedia, The Free Encyclopedia.* Available at: https://en.wikipedia.org/wiki/MIME

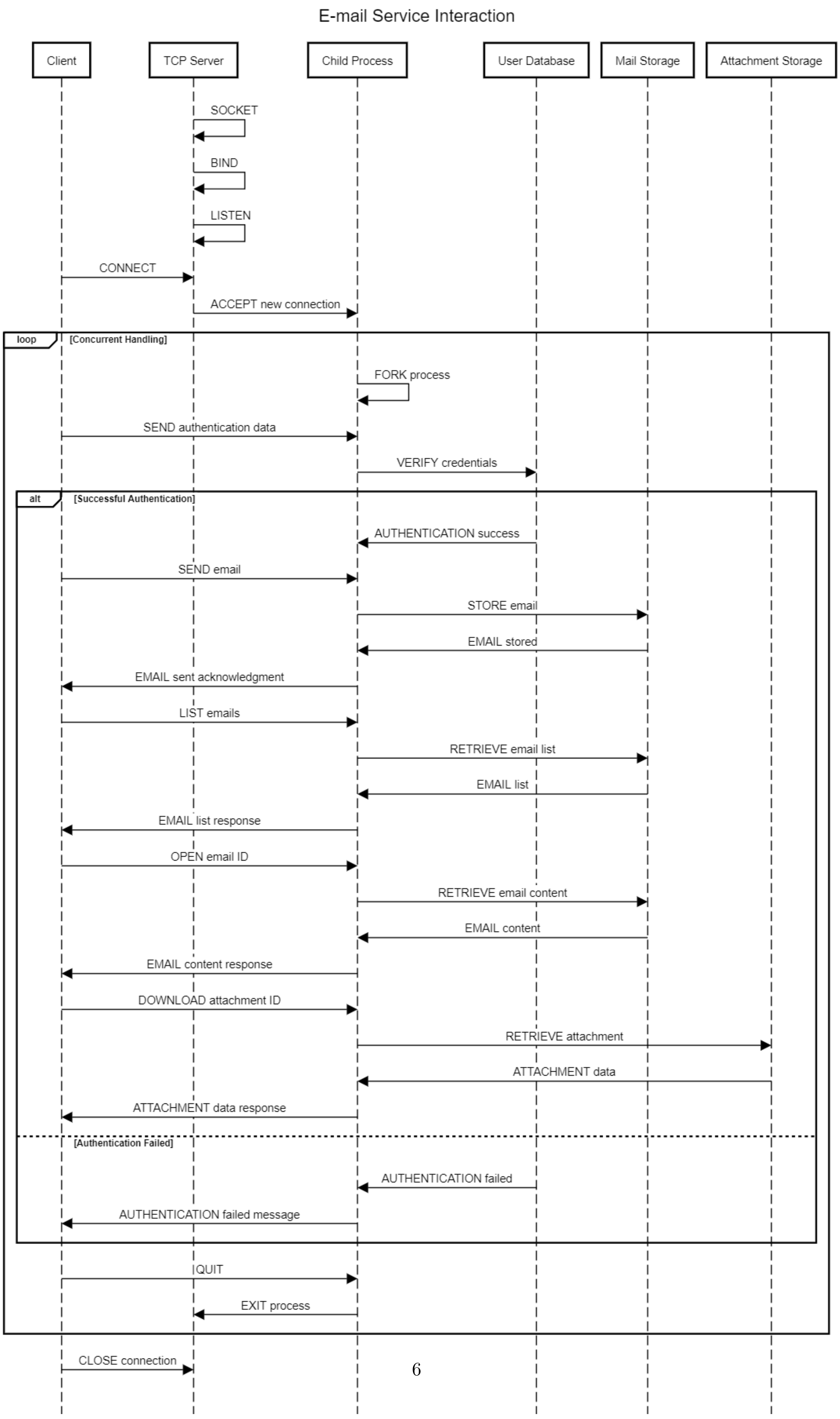

Figure 1: E-mail Service Interaction Diagram