POPpy
Ein minimalistischer POP3 Client
Aufgabe Nr. 29

Rafael Schreiber (5BHIF/KatNr. 20)

April 2021

Inhaltsverzeichnis

1 Einleitung

1.1 Aufgabenstellung

Laut der Angabe soll ein einfacher POP3 Client programmiert werden, mit welchem man E-Mails abrufen, herunterladen und löschen kann. Zudem sollte mithilfe von GnuTLS [1] auch StartTLS unterstützt werden.
Eine weitere Anforderung ist, dass die Netzwerkbibliothek Asio [2] verwendet werden soll. Asio eine Bibliothek für eine moderne Netzwerk und low-level I/O Programierung mit einem konsitenten asynchronen Modell.

1.2 Probleme

1.2.1 GnuTLS mit Asio

Ein großes Problem bei der Programmierung mit Asio war, dass GnuTLS standardmäßig nicht die Asio Socket Objekte unterstützt. GnuTLS benötigt nämlich den nativen Socket Deskriptor, auf welchem man bei Asio keinen Zugriff hat.

```
gnutls_transport_set_int(_sess, _socket_descriptor);
```

Es gibt zwar GnuTLS Wrapper für Asio, diese sind aber nur kleine Projekte von einzelnen Entwicklern auf GitHub und außerdem sollten so wenige Bibliotheken wie möglich verwendet werden. Deshalb wurde auf Asio verzichtet und stattdessen die native POSIX Socket Bibliothek verwendet. GnuTLS funktioniert einfach damit am besten.

1.2.2 StartTLS

StartTLS ist ein Verfahren zum herstellen einer gesicherten Netzwerkverbindung mittles TLS. Der unterschied zu implizitem TLS besteht darin, dass der erste Verbindungsaufbau unverschlüsselt erfolgt. Das bietet die Möglichkeit für einen möglichen Man-in-the-Middle-Angriff.

Seit 2018 wird davon abgeraten StartTLS für das Abrufen für E-Mails zu verwenden [3]. Auch weil StartTLS für POP3, sowohl als auch für IMAP, keinen Vorteil gegenüber implizitem TLS bietet und weil es praktisch keine POP3-Server mehr gibt, welche auf StartTLS setzen, wurde auf dessen

implementierung verzichtet und stattdessen auf „herkömmliches“ TLS getzt. Zum Glück bietet GnuTLS auch eine Unterstützung für implizites TLS.

2 Implementierung

Zu aller erst wurden sämtliche Codierungsrichtlinien von Professor Dr. Günter Kolousek eingehalten. Für die Benennung von Variablen und Funktionen wurde „`sneak_case`“ verwendet, ausgenommen die gRPC Implementierungen, weil die vom gRPC-Compiler generierte C++ Bibliothek „`PascalCase`“ nutzt.

2.1 Architektur

POPpy besteht aus einer klassischen Client-Server Architektur, mit dem Unterschied, dass Frontend und Backend via gRPC [4] kommunizieren und nicht wie bei Web-Apps üblich mit einer REST-Schnittstelle. Das Backend ist in C++ programmiert und bietet deswegen auch eine hohe Geschwindigkeit.

Das Frontend hingegen wurde in JavaScript programmiert, welches über eine HTML Datei eingebunden wird. Dieses HTML File ist lokal gespeichert und wird beim starten von POPpy automatisch, wenn möglich, mit dem Standardprogramm für HTML Dateien, überlicherweise ein Webbrowser, vom Backend geöffnet.

2.1.1 gRPC & protobuf

gRPC ist ein von Google entwickeltes open-source Remote Procedure Call (RPC) Framework. Mit gRPC kann der Client direkt Methoden auf dem Server aufrufen, auch wenn sich dieser nicht auf der gleichen Maschine befindet wie der Client. Die Kommunikation bei gRPC erfolgt über HTTP/2 welches i. Allg. zwar schneller und sicherer ist aber keine Abwärtskompatibilität zu HTTP/1.1 bietet.

Dabei ist protobuf [5], ebenfalls von Google entwickelt, für die Strukturierung der Daten verantwortlich. Deshalb ist protobuf ein integraler bestandteil von gRPC. Protobuf serialisiert strukturierte Daten anders wie JSON nicht text-basiert, sonder binär. Das reduziert die Nachrichtengröße, senkt dadurch auch den Traffic und steigert somit die Geschwindigkeit.

Probleme mit gRPC im Web: Wegen HTTP/2 Limitierungen in Browsern war gRPC eine lange Zeit Web-Entwicklern vorenthalten. Google widmete sich diesem Problem und entwickelte gRPC-Web. [6]
Das Problem wurde gelöst indem gRPC-Web HTTP/1.1 anstelle von HTTP/2 nutzt. Das Problem mit der Abwärtskompatibilität wird mit einem Proxyserver gelöst. Google empfiehlt auf ihrer Website zu gRPC-Web den Envoy Proxy [7].

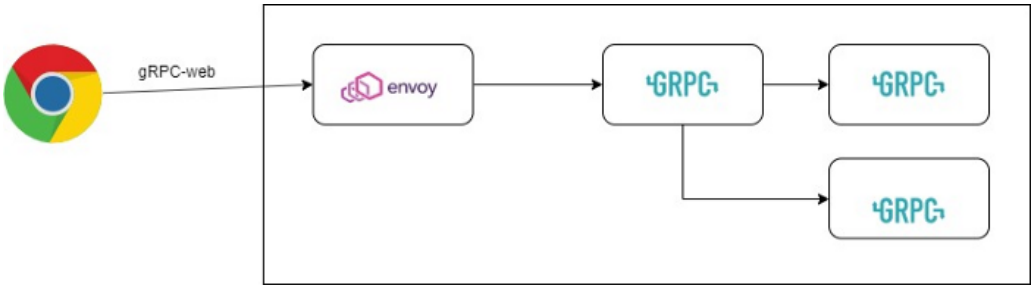

Abbildung 1: Kommunikation zwischen Browsern und gRPC Diensten [9]

2.2 Aufbau

2.2.1 Frontend

Das Frontend ist in JavaScript programmiert und wurde in einem npm Projekt angelegt. Der Paketmanager npm eignet sich gut dafür, weil das Frontend mehrere Bibliotheken benötigt darunter das wichtigste `grpc-web`. Dazu müssen die vom protobuf- und gRPC-Compiler generierten Dateien im Frontend eingebunden werden. Außerdem bietet webpack [10] eine praktische Lösung alle Abhängigkeiten und Module in eine statische „minified“ .js Datei zusammenzufassen welche dann einfach im HTML Dokument eingebunden wird.

2.2.2 Backend

Das Backend wurde in C++17 programmiert und bietet jeweils eine gRPC Schnittstelle für die Kommunikation mit dem Frontend und eine eigens Programmierte API für den Datenaustausch mit dem POP3 Server Schnittstelle für die Kommunkation. Die Mailbox Klasse ist dafür zuständig, E-Mails während der Laufzeit zu cachen, Verbindungen zu verwalten und eine einfache Repräsenatation auf die E-Mails zu bieten.

2.2.3 Envoy

Envoy ist ein Proxy Server entwickelt von den Programmieren bei Lyft. Google selbst empfiehlt die Nutzung von Envoy bei gRPC-Web Applikationen. Envoy wird automatisch mit der richtigen Konfiguration vom Backend gestartet und verwaltet. Wenn Envoy abstürzt, dann schließt sich auch automatisch das Backend. Der Endanwender muss sich dabei um nichts kümmern.

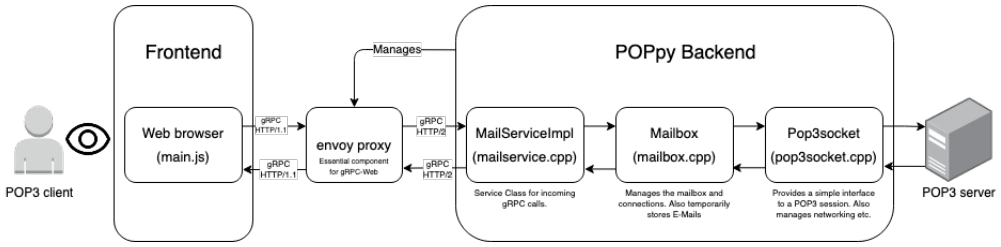

Abbildung 2: Interne Programmstruktur von POPpy

2.3 Ablauf

2.3.1 Start

POPpy wird über die Kommandozeile gestartet. Beim Start muss der Name eines Bookmarks angegeben werden. Ein Bookmark gibt dabei Zugangsdaten zu einem POP3 Server an. Bookmarks werden standardmäßig von der Datei `~/.config/poppy/poppy.conf` gelesen. Alternativ kann auch eine andere Datei angegeben werden. Diese TOML Datei ist folgendermaßen aufgabaut.

```
# POPpy config file
# This files contains bookmarks to your pop3 servers

[beispielverbindung] # ERFORDERLICH
  host = "mail.domain.tld" # ERFORDERLICH
  port = 995               # Optional
  user = "user@domain.tld" # ERFORDERLICH
  pass = "changeit"        # NICHT EMPFOHLEN
  ssl = true               # Optional
```

Wenn kein Passwort gespeichert wird, was auch Empfohlen ist, dann fragt POPpy automatisch noch vor dem Start des Frontends nach dem Passwort.

Wenn die Felder `port` oder `ssl` nicht angegeben werden dann werden sie automatisch auf 995 und true gesetzt.

Beim Start werden zwei Threads gestartet. Diese werden mittels `async` Aufruf als `future` in einem `vector<future<int>*>` gespeichert. Diese Futures liefern als Integer einen Statuscode zurück. Diese Statuscodes sind in der Datei `include/globals.h` gespeichert.
Der erste Thread startet und verwaltet den Envoy Proxy. Zum Starten wird die Bibliothek `cpp-subprocess` verwendet. Die Funktion hinter diesem Thread sieht folgendermaßen aus:

```
int start_envoy_proxy() {
    auto envoy_process = Popen({"envoy", "-c", prefix + "/share/poppy/"
      "envoy_poppy_proxy.yaml", "--log-path", "/tmp/envoy_poppy.log.txt"},
      input{PIPE}, output{"/dev/null"});
    envoy_process_pid = envoy_process.pid();
    logger->debug("Envoy proxy started with pid: {}", envoy_process_pid);
    logger->debug("Check envoy log file after exit under "
      "/tmp/envoy_poppy.log.txt");
    logger->debug("You can also access the envoy admin panel under "
      "http://localhost:42963");
    size_t envoy_exit_status = envoy_process.wait();

    if (envoy_exit_status != SUCCESS) {
        logger->warn("Envoy proxy (PID: {}) exited with status code: {}",
          envoy_process_pid, envoy_exit_status);
        envoy_error = true;
    } else {
        logger->debug("Envoy proxy (PID: {}) exited with status code: {}",
          envoy_process_pid, envoy_exit_status);
    }
    return envoy_exit_status;
}
```

Während der zweite Thread startet, wartet dieser davor ob Envoy richtig gestartet hat. Falls das nicht der Fall sein sollte, dann verweigert dieser den Start. Ein Teil der Funktion hinter dem zweiten Thread sieht so aus:

```
int start_grpc_server() {
    /* ... */
```

```
    int status = service->ConnectMailService(cred.hostname, cred.port,
      cred.username, cred.password, cred.encrypted);
    if (status != SUCCESS) {
        logger->error("An error occurred while starting the gRPC service");
        free(service);
        return status;
    }

    grpc::ServerBuilder builder;
    builder.AddListeningPort(server_addr, grpc::InsecureServerCredentials());
    builder.RegisterService(service);
    server = builder.BuildAndStart();
    logger->info("gRPC server successfully started and is now "
      "listening on: {}", server_addr);
    thread check_shutdown_thread(check_shutdown);

    if (open_frontend() != SUCCESS) {
        logger->warn("Failed to start frontend. Please open the html file "
          "manually located under: {}/share/poppy/index.html", prefix);
    }

    server->Wait();
    check_shutdown_thread.join();
    service->ExitMailService();
    server->Shutdown();
    free(service);
    return SUCCESS;
}
```

Die Variablen `cred` und `service` sind dabei globale Variablen. Davon ist `service` ein Pointer zum gRPC Service und `cred` ein Objekt vom Typ `cred_t`, welches vom Bookmark Parser mit den Zugangsdaten befüllt wird. Die Methode `ConnectMailService` startet die Mailbox, welche dann im Endeffekt mithilfe von Pop3socket die POP3 Sitzung startet. Wenn die Sitzung gestartet ist hört der gRPC Dienst nach eingehenden RPCs.

2.3.2 Laufzeit

Der gRPC Service nimmt eingehende Verbindungen und RPCs an und bearbeitet sie. Die Daten bekommt der Dienst von dem Mailbox Objekt welches er beim Start angelegt hat. Die RPC Handler Funktionen sehen so aus:

```
class MailServiceImpl final : public MailService::Service {
  private:
    Mailbox _pop3mailbox{};
    std::string _account;

  public:
    MailServiceImpl() {}
    int ConnectMailService(std::string hostname, uint16_t port,
      std::string username, std::string password, bool encrypted);
    grpc::Status GetMailBoxInfo(grpc::ServerContext* context,
      const Empty* request, MailBoxInfo* reply) override;
    grpc::Status GetMailPreviews(grpc::ServerContext* context,
      const MailPreviewRequest* request, MailPreviewResponse* reply) override;
    grpc::Status UpdateMailbox(grpc::ServerContext* context,
      const Empty* request, StatusResponse* reply) override;
    grpc::Status DeleteMail(grpc::ServerContext* context,
      const SpecifiedMail* request, StatusResponse* reply) override;
    grpc::Status ResetMailbox(grpc::ServerContext* context,
      const Empty* request, StatusResponse* reply) override;
    grpc::Status DownloadMail(grpc::ServerContext* context,
      const SpecifiedMail* request, DownloadedMail* reply) override;
    grpc::Status ExitApplication(grpc::ServerContext* context,
      const Empty* request, StatusResponse* reply) override;
    void ExitMailService();
    ~MailServiceImpl() {}

};
```

Das ist die einzige Klasse, welche in „`PascalCase`“ geschrieben ist.
Beim Start der Mailbox Klasse wird außerdem ein Thread gestartet, welcher alle 10 Sekunden den POP3 Server „anpingt“ um zu Kontrollieren ob die Verbindung noch aufrecht ist und damit der Server die Verbindung nicht wegen inaktivität abbricht. Diese dazugeörige Funktion zu diesem Ping Thread

sieht so aus:

```
void Mailbox::_ping_thread_function() {
    int noop_status;
    this_thread::sleep_for(10000ms);

    while (!shutdown_initiated) {
        if (_pop3sess.in_update_state()) {
            noop_status = _pop3sess.ping();
            if (noop_status != SUCCESS) break;
        }
        this_thread::sleep_for(10000ms);
    }

    if (!shutdown_initiated) {
        logger->error("Connection timeout. Server is not reachable. Initiating"
          " shutdown...");
        shutdown_initiated = noop_status;
    }
}
```

Die Variable `shutdown_initiated` ist dabei eine Hilfsvariable welche indiziert, ob das Backend gestoppt wird unabhängig davon ob der Nutzer POPpy geschlossen hat oder irgendein Fehler aufgetreten ist.

2.3.3 Shutdown

Wie vorhin erwähnt wird beim Herunterfahren die Variable `shutdown_initiated` gesetzt welche jedem Thread mitteilt, dass er sich beenden soll. So auch der Thread der Funktion `void check_shutdown()` . Dieser Thread wird in der Funktion `int start_grpc_server()` gestartet und beendet sich selbst sowie den gRPC Server wenn die `shutdown_initiated` Variable gesetzt wurde. Die Funktion zu diesem Thread ist sehr minimal und sieht so aus:

```
void check_shutdown(){
    while (!shutdown_initiated) {
        this_thread::sleep_for(1000ms);
    }
    logger->info("gRPC server successfully shut down");
    server->Shutdown();
}
```

Während sich der gRPC Server beendet, wird auch der Envoy Proxy mittels `SIGTERM` beendet. Wenn alle Dienste beendet sind terminiert das Programm.

3 Verwendung

3.1 Abhängigkeiten

Zuerst müssen folgende Programme installiert sein oder zumindest in `$PATH` vorhanden sein. Meson überprüft vor dem kompilieren, ob diese vorhanden sind, wenn das nicht der Fall ist, dann wird bei einem Fehler der Link zum Download ausgegeben.

- Einen C++ Compiler
- The Meson Build System [12]
- protobuf [5]
- gRPC mit protoc-gen-grpc-web [11]
- Envoy Proxy [7]
- npm [8]

Daneben werden außerdem auch noch Bibliotheken benötigt:

- GnuTLS [1]
- GNU Mailutils [13]

Dazu kommen noch ein paar Header-Only Bibliotheken, dessen Elternpfade in der Datei `meson_options.txt` angegeben werden:

- CLI11 [14]
- cpp-subprocess [15]
- spdlog [16]
- toml++ [17]

3.2 Komplilierung

Zuerst wechselt man in den Ordner `build/`, dieser sollte zu beginn leer sein. Danach kann Meson mit folgendem Befehl gestartet werden:

```
$ meson --prefix $PWD/.. ..
# Falls das Programm danach auch installiert werden soll, dann
# sollte bei --prefix der gewünschte Installationspfad angegeben
# werden. Zum Beispiel /usr/local
```

Währenddessen überprüft Meson, ob auch alle Abhängigkeiten installiert sind. Eine Ausnahme dabei ist GNU Mailutils. Außerdem werden auch alle protobuf und gRPC Dateien generiert und das Frontend via npm kompiliert. Dieser Befehl könnte deshalb ein wenig länger dauern.
Zum kompilieren danach folgenden Befehl ausführen

```
$ ninja
```

Falls gewünscht kann das Programm danach auch installiert werden:

```
$ meson install
```

Die Kompilierung wurde auf folgenden Plattformen getestet:

- macOS Big Sur (Apple clang version 12.0.0)
- Ubuntu 20.04 LTS (gcc version 9.3.0)

Beide Plattformen basieren auf einen Prozessor mit x86 Befehlssatz.

3.3 Benutzung

Vor dem ersten Start muss die Datei `~/.config/poppy/poppy.conf` bearbeitet werden. Dort einfach ein Bookmark mit den Anmeldedaten definieren und speichern.

Achtung! Falls POPpy nicht installiert wurde, muss die Datei manuell erstellt werden. Eine Beispielkonfiguration ist unter `poppy_sample.conf`.

3.3.1 Kommandozeilenparameter

bookmark TEXT REQUIRED: Gibt den Namen des Bookmarks an zu welchem eine POP3 Sitzung hergestellt werden soll.

-c,–config TEXT:FILE OPTIONAL: Falls nicht die Standard Konfigurationsdatei unter `poppy_sample.conf` verwendet werden soll, kann mit diesem Parameter eine alternative Datei angegeben werden.

-d,–debug: Wenn dieser Flag gesetzt ist, wird POPpy im Debug Modus gestartet. Dieser Modus bietet mehr Transparenz über die Abläufe im Programm und ist hilfreich bei Fehlern.

Literatur

[1] The GnuTLS Transport Layer Security Library
https://www.gnutls.org

[2] Asio C++ Library
https://think-async.com/Asio

[3] SMTP TLS Reporting
https://tools.ietf.org/html/rfc8460

[4] gRPC: A high performance, open source universal RPC framework
https://grpc.io/docs/what-is-grpc/introduction

[5] Protocol Buffers | Google Developers
https://developers.google.com/protocol-buffers

[6] A basic tutorial introduction to gRPC-web.
https://grpc.io/docs/platforms/web/basics/

[7] Envoy Proxy - Home
https://www.envoyproxy.io/

[8] npm (Node Package Manager)
https://www.npmjs.com

[9] Ditching REST with gRPC-web and Envoy
https://medium.com/swlh/ditching-rest-with-grpc-web-and-envoy-bfaa89a39b32

[10] webpack
https://webpack.js.org

[11] protoc-gen-grpc-web
https://github.com/grpc/grpc-web/releases

[12] The Meson Build System: Project Website
`https://mesonbuild.com`

[13] GNU Mailutils: General-Purpose Mail Package
`https://mailutils.org`

[14] GitHub: CLIUtils/CLI11 command line parser
`https://github.com/CLIUtils/CLI11`

[15] GitHub: arun11299/cpp-subprocess Subprocessing with modern C++
`https://github.com/arun11299/cpp-subprocess`

[16] GitHub: gabime/spdlog Fast C++ logging library.
`https://github.com/gabime/spdlog`

[17] GitHub: marzer/tomlplusplus Header-only TOML config file parser and serializer
`https://github.com/marzer/tomlplusplus`