МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Тольяттинский государственный университет»

Кафедра «Прикладная математика и информатика»
(наименование)

09.03.03 Прикладная информатика
(код и наименование направления подготовки, специальности)

Разработка программного обеспечения
(направленность (профиль) / специализация)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)

на тему «Разработка микросервиса для автоматического обновления встроенной linux-системы»

Обучающийся М.В. Киян
(Инициалы Фамилия) (личная подпись)

Руководитель О.В. Оськина
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Тольятти 2025

Аннотация

Тема выпускной квалификационной работы «Разработка микросервиса для автоматического обновления встроенной linux-системы» [20].

Ключевые слова: обновления, система A/B обновлений, GRPC, микросервис, встраиваемая система.

Предмет исследования – разработка системы A/B обновлений с консольным интерфейсом и grpc api для встраиваемой системы.

Цель выпускной квалификационной работы – разработка удобного микросервиса с возможностью скачивать обновления с облака, устанавливать обновления из локального архива, как вручную через консоль, так и используя grpc интерфейс.

Практическая значимость выпускной квалификационной работы [19] заключается в разработке и внедрения программного обеспечения для эффективного и безопасного обновления встраиваемой системы без участия человека.

Данная работа состоит из введения, четырёх разделов, заключения и списка используемой литературы.

Бакалаврская работа состоит из 48 страниц текста, 20 рисунков, 25 листингов.

Содержание

Аннотация 2
Введение 4
1 Анализ требований 5
1.1 Анализ входных и выходных данных 6
2 Проектирование программного продукта 7
2.1 Проектирование UML 7
2.2 Выбор средств разработки 10
2.3 Описание алгоритма 14
3 Используемые средства разработки 18
3.1 Среда разработки Visual Studio Code 18
4 Разработка программного обеспечения 23
4.1 Разработка интерфейса пользователя 23
4.2 Описание основы приложения 29
4.3 Описание процесса скачивания обновления 32
4.4 Подготовка к обновлению 34
4.5 Установка обновления 39
4.6 Откат системы 41
4.7 Демонстрация сборки 42
4.8 Демонстрация запуска 43
Заключение 46
Список используемых источников 47

Введение

Обновление – это важное составляющие каждой информационной системы, обеспечивающие внедрение новых функций и исправление возможных ошибок и уязвимостей. Особо важно автоматическое обновление для встраиваемых систем так как к ним не всегда есть доступ оператора и такое устройство должно уметь безопасно обновляться без прямого участия человека.

Предметом исследования является применение системы A/B обновлений во встраиваемой системе Linux на базе процессора ARM.

Цель выпускной квалификационной работы – создание микросервиса обновления из заранее предопределённого архива управляемый с помощью grpc и с возможностью скачивать обновления из облака. Проект направлен на создание удобного микросервиса обновлений, который станет частью большого программного комплекса.

Для достижения цели выпускной квалификационной необходимо написать программный модуль, реализующий следующие функции:

- Скачивание архива из облака;
- Распаковка архива с проверкой на повреждение;
- Установка обновления используя механизм A/B;
- Возврат к состоянию до обновления;
- Удобные grpc методы для управления;
- Консольный интерфейс для ручного управления.

Практическая значимость данной бакалаврской работы заключается в внедрении сервиса обновлений тем самым позволив обновлять систему просто и безопасно в автоматическом режиме.

1 Анализ требований

Современные встраиваемые системы часто выполняют важные функции в составе более крупных комплексов. По мере развития программного обеспечения, появления новых функций и устранения уязвимостей и ошибок возникает необходимость обновления системы. Обновление позволяет:

- исправлять ошибки и повышать стабильность работы устройства;
- обеспечивать актуальность и безопасность программного обеспечения (в том числе закрывать известные уязвимости);
- добавлять новую функциональность без необходимости физического вмешательства в устройство;
- продлевать срок службы устройства за счёт адаптации к новым условиям эксплуатации.

Встраиваемые системы имеют ряд особенностей, таких как ограниченные ресурсы (память, вычислительная мощность), специфическая аппаратная конфигурация и высокие требования к надёжности. Стандартные решения часто не рассчитаны на эти особенности.

Необходимо разработать приложение подходящие под следующие требования:

- использование механизма A/B обновлений [1][2];
- обновление из файла архива;
- скачивание файла архива обновления из облака;
- возможность отката обновлений;
- простота, без избыточной функциональности;
- малый объём занимаемой памяти и быстродействие;
- надёжность;
- консольный режим, без GUI;
- управление через GRPC [8].

1.1 Анализ входных и выходных данных

В качестве входных данных выступает архив, который содержит файл манифеста и образы. Архив из себя представляет файл cpio [4] сжатый с помощью gzip [9] и содержит образы разделов требующие обновления и файл манифеста с перечислением этих образов и их контрольными суммами.

К выходным данным относятся обновлённые разделы системы. В системе используется A/B-схема.

A/B-схема обновления – это метод безопасного обновления встроенных систем, минимизирующий риск выхода устройства из строя в случае сбоя при обновлении. Эта схема широко используется в Android-устройствах, а также в встроенных системах, использующих Buildroot [3] и других систем сборки.

Система имеет две копии разделов rootfs и ядра:

- слот A – активный слот (с которого работает система сейчас);
- слот B – пассивный слот (зарезервирован под обновление).

Во время обновления:

- образ новой системы записывается в неактивный слот (B);
- после успешной записи и верификации загрузчик (например, U-Boot [14]) переключается на слот B;
- при следующей перезагрузке система загружается со слота B;
- если загрузка прошла успешно – слот B становится новым активным;
- если произошёл сбой – загрузчик откатывается обратно к слоту A.

2 Проектирование программного продукта

2.1 Проектирование UML

UML – это стандартный язык для визуального моделирования в области программных систем. Он используется для описания архитектуры, процессов, классов, взаимодействий и других свойств системы на разных этапах её проектирования и разработки, более подробно о UML [15].

2.1.1 Диаграмма классов

Диаграмма классов – один из главных инструментов в UML. Она показывает структуру системы с точки зрения объектно-ориентированного программирования: какие классы существуют, какие у них свойства, методы и как они связаны между собой. Из минусов диаграммы классов, то что при использовании функционального или смешенного программирования нет возможности отобразить полную картину.

На рисунке 1 показана диаграмма классов отображая точку зрения реализации показывая классы используемы в программном коде.

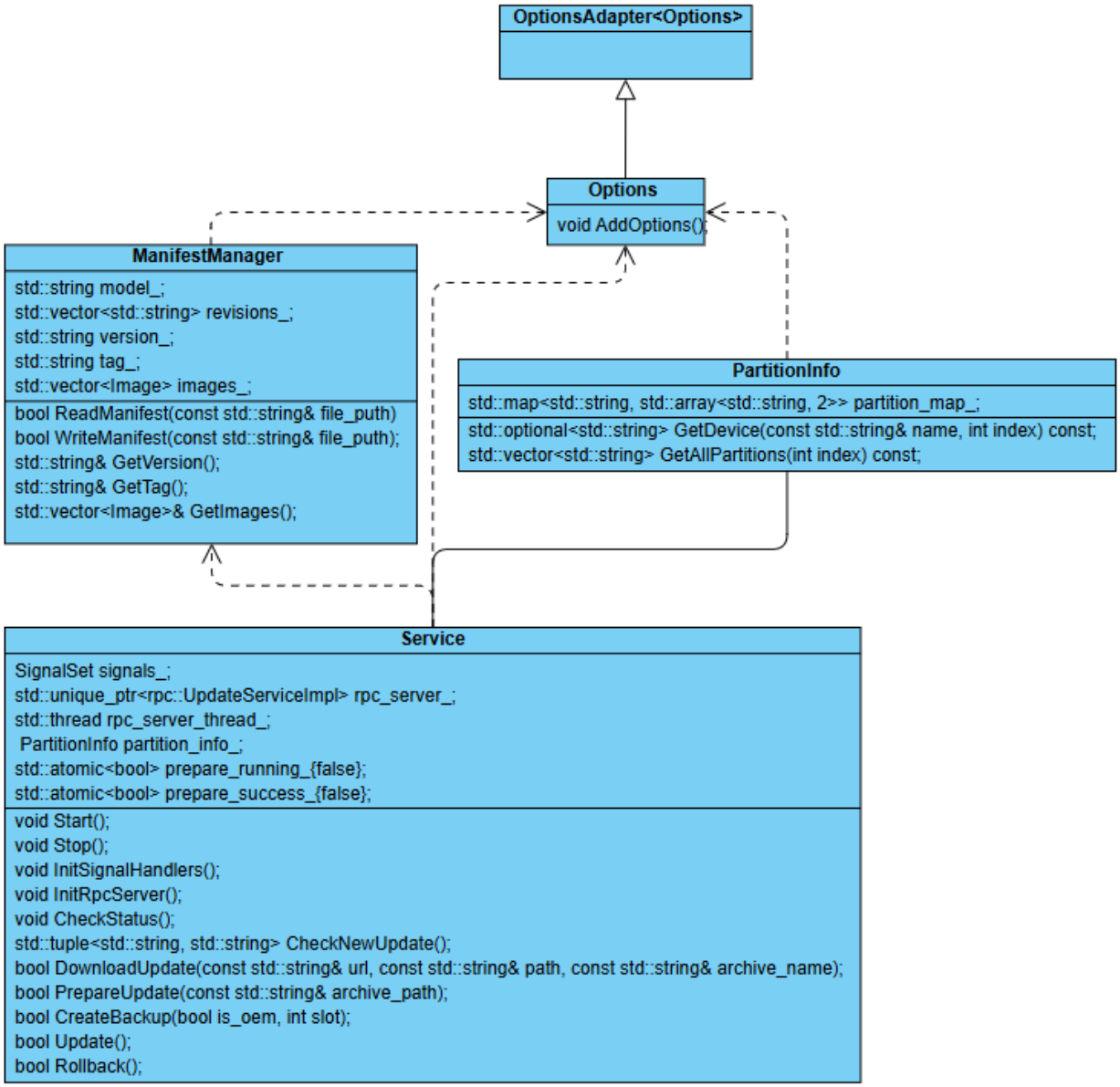

Рисунок 1 – диаграмма классов

Помимо классов в приложении используются вспомогательные функции в анонимном пространстве имён, но данная диаграмма не подразумевает их показ.

2.1.2 Диаграмма вариантов использования

Диаграмма вариантов использования – это UML диаграмма, которая отображает взаимодействие участников (акторов) с системой через набор функций (вариантов использования), рисунок 2.

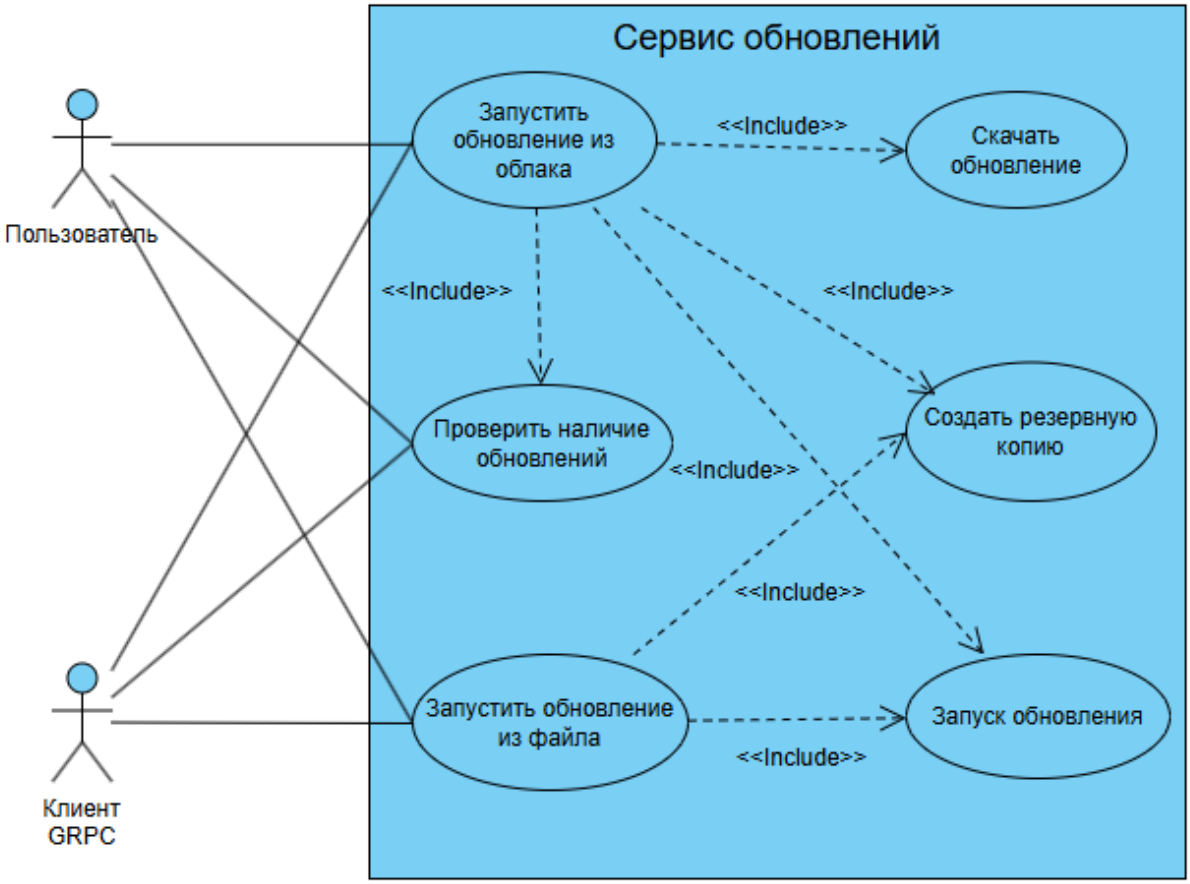

Рисунок 2 – диаграмма вариантов использования

У пользователя по мимо этих действий есть ещё дополнительные, но они большее для отладки и тестирования и поэтому не были отражены на данной диаграмме.

2.1.3 Диаграмма потоков данных

Диаграмма потоков данных – это схема, которая показывает, как информация проходит через систему. В ней отображаются процессы, которые обрабатывают данные, места хранения информации, а также внешние участники, от которых данные поступают или которым они отправляются. Такая диаграмма помогает понять, откуда берутся данные, как они преобразуются и куда в итоге попадают, рисунок 3.

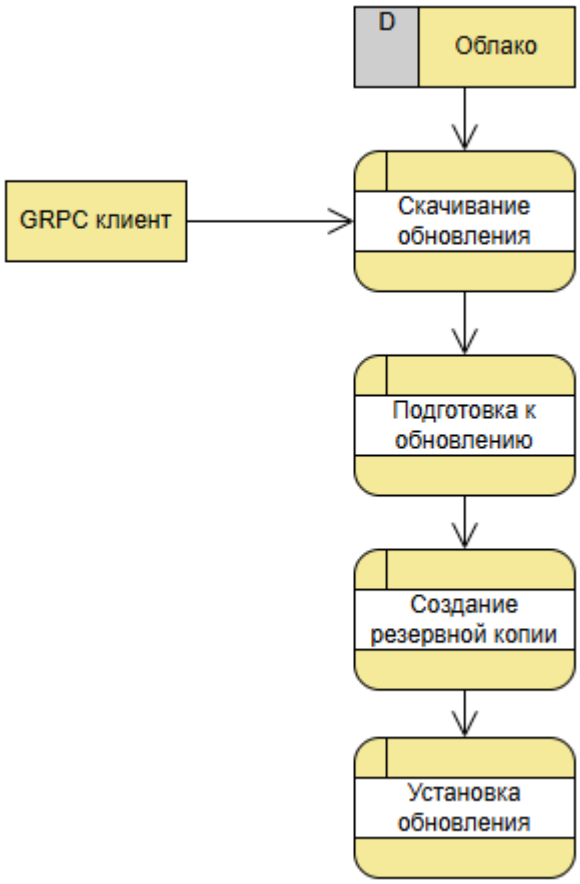

Рисунок 3 – диаграмма потоков данных для облачного обновления через rpc

На рисунке 3 показано обновление используя облако с помощью rpc вызова. В случаи невозможности обновления rpc метод вернёт код ошибки.

2.2 Выбор средств разработки

В рамках реализации программного обеспечения проекта был выбран язык программирования C++. Это компилируемый язык общего назначения, ориентированный как на системное, так и на прикладное программирование. Являясь наследником языка C, он предоставляет как низкоуровневый доступ к аппаратным ресурсам, так и высокоуровневые абстракции, включая объектно-ориентированное, обобщённое и функциональное программирование.

Преимущества языка C++:

- высокая производительность: программы на C++ компилируются в эффективный машинный код, что критично в задачах, требующих минимального времени отклика или ограниченного энергопотребления;
- широкие возможности управления ресурсами: ручное управление памятью и низкоуровневые конструкции обеспечивают полный контроль над системой;
- мощная стандартная библиотека (STL): включает множество готовых контейнеров и алгоритмов, упрощающих реализацию сложной логики;
- кроссплатформенность: C++ поддерживается большинством операционных систем и аппаратных архитектур;
- распространённость и зрелость: язык существует более 40 лет и активно используется в промышленной разработке, что обеспечивает обширную экосистему, документацию и инструменты.

Недостатки классического C++:

- сложность синтаксиса и крутая кривая обучения: язык требует глубокого понимания множества концепций;
- вероятность ошибок при ручном управлении памятью: неправильная работа с указателями может приводить к утечкам и сбоям;
- отсутствие встроенных механизмов для работы с многопоточностью в ранних версиях;
- сложности в сопровождении больших проектов из-за отсутствия модульности в традиционном виде.

С принятием стандарта C++11 и последующих редакций язык получил значительное развитие. Современный C++ активно ориентирован на повышение удобства разработки, безопасность и выразительность кода, стремясь сохранить при этом производительность и контроль над ресурсами.

Ключевые усовершенствования:

- автоматическое управление памятью: с помощью умных указателей (std::unique_ptr, std::shared_ptr) риск утечек значительно снижается;
- Вывод типов (auto), лямбда-выражения и шаблоны делают код более читаемым и обобщённым;
- стандартизированная многопоточность (std::thread, std::mutex, std::async);
- новые абстракции для обработки ошибок – std::optional, std::variant;
- корутины, концепты, модули и диапазоны позволяют реализовывать сложные архитектуры с меньшими усилиями и лучшей читаемостью.

Современные версии C++ во многом устраняют исторические недостатки языка, но при этом вносят и новые:

- усложнение стандарта: быстрый рост языка делает его всё менее доступным для начинающих разработчиков;
- высокие требования к компиляторам: не все платформы оперативно поддерживают последние стандарты;
- сложности в интеграции с устаревшим кодом, особенно при использовании новых абстракций и модульной системы.

Язык C++ был выбран в силу сочетания высокой производительности, широких возможностей управления ресурсами и активного развития в сторону более безопасной и выразительной разработки. Современный C++ предоставляет необходимые инструменты для построения надёжного и расширяемого программного решения, соответствующего как текущим требованиям проекта, так и перспективам его развития.

В качестве дополнения стандартной библиотеки C++ используется библиотека Boost.

Библиотека Boost представляет собой коллекцию высококачественных и хорошо протестированных библиотек для языка C++. Она охватывает широкий спектр функциональности – от работы со строками и контейнерами до многопоточности, математических вычислений и сетевого взаимодействия.

Так же используются дополнительные специализированные библиотеки:

– yaml-cpp [17],
– zlib [18],
– nlohmann_json [11],
– curl [6],
– libarchive [10].

Для сборки проекта используется CMake. CMake – это инструмент для автоматизации сборки проектов. Он генерирует файлы сборки (например, для make, ninja) на основе описания проекта в файлах CMakeLists.txt. Позволяет легко управлять зависимостями, конфигурациями и переносимостью сборки между разными платформами.

2.3 Описание алгоритма

Процесс упрощённо показан на рисунке 4.

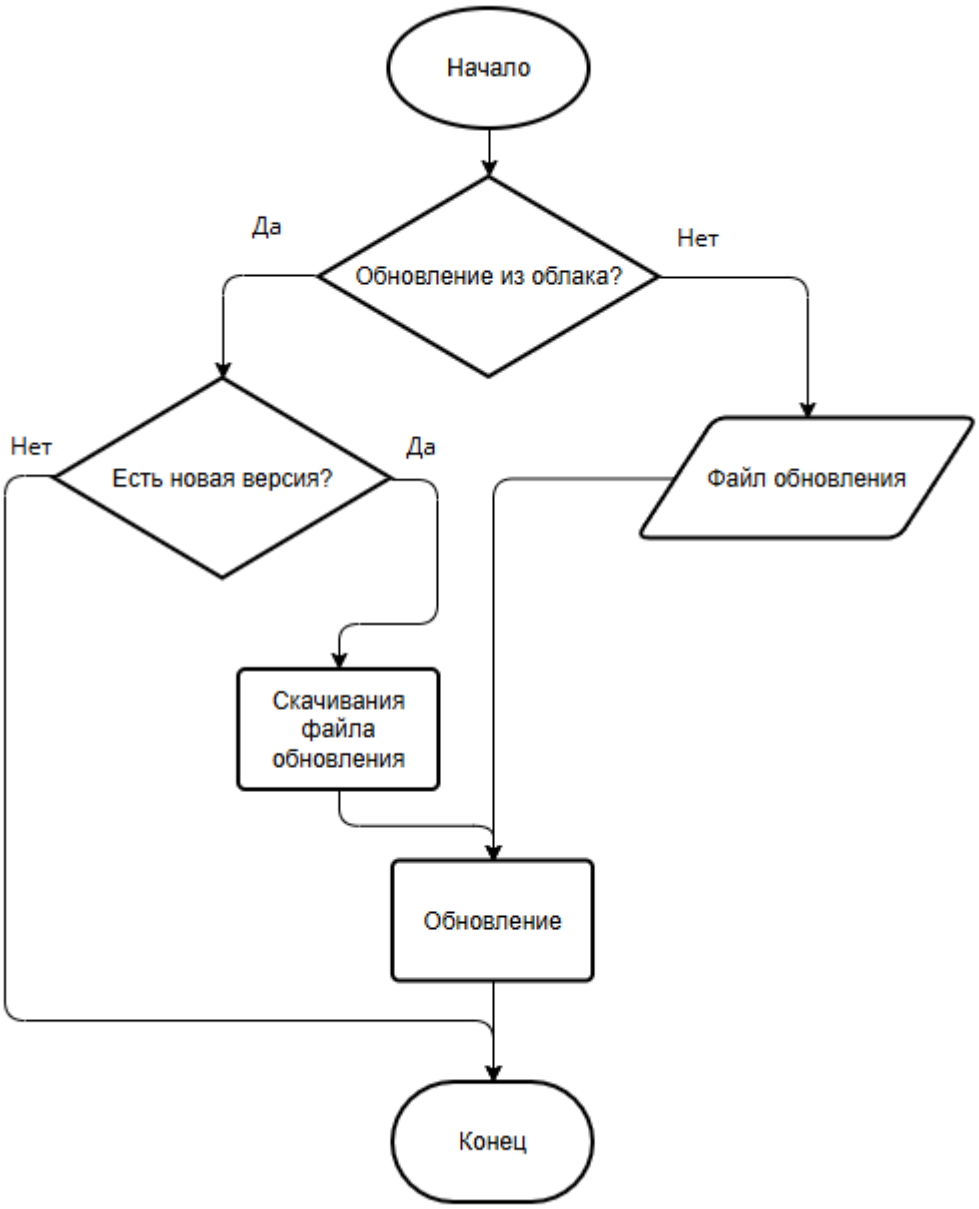

Рисунок 4 – процесс обновления в упрощённом виде

Обновление может быть из облака со скачиванием архива с обновлениями или из файла архива, который уже должен заранее быть загружен на устройство.

Сам процесс разбит на несколько этапов, на рисунке 5 показан процесс обновления.

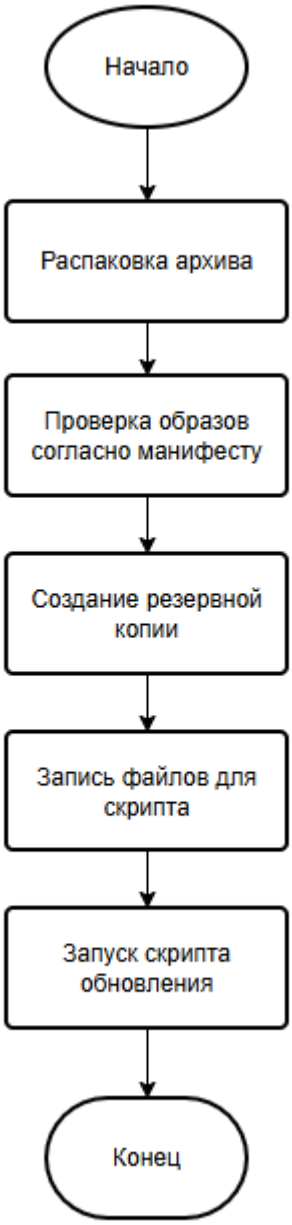

Рисунок 5 – основные этапы обновления

В начале архив с обновлениями распаковывается во временную директорию, а далее хеш sha256 [13] у каждого образа проверяется согласно манифесту.

В системе используется механизм A/B обновлений и в этом случаи нет необходимости делать резервные копии системных разделов, но раздел с приложениями в единственном экземпляре и необходимо сделать и сохранить его образ в специальную директорию с резервными копиями.

Сам процесс обновления запускает bash-скрипт и для него необходимо создать файл, в котором содержится номер нового слота, а также файл, где перечислены образы и разделы, на которые следует записать эти образы.

В последнею очередь создаётся файл с информацией что обновление инициировано и запуск скрипта обновления.

На рисунке 6 показан алгоритм работы скрипта обновления.

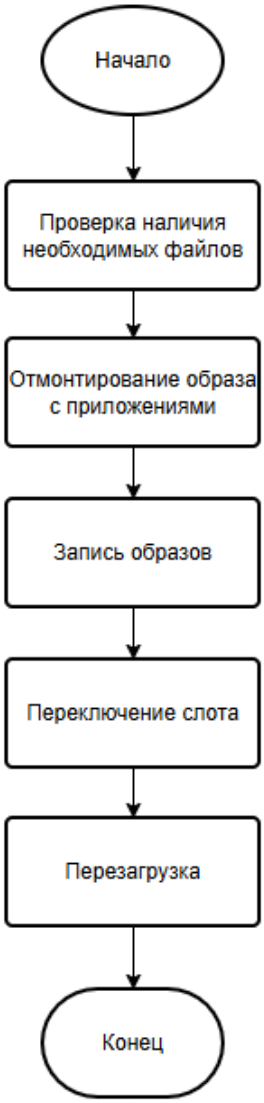

Рисунок 6 – скрипт обновления

Скрипт в начале проверяет наличие необходимых файлов с новым слотом и с сопоставлениями разделов с образами.

Дальше происходит размонтирование раздела с приложениями и запись всех образов в цикле.

Заключительным этапом происходит переключение на новый слот и перезагрузка системы.

После перезагрузки сервис обновлений проверяет что обновление успешно, слот соответствует и все приложения запущенны без ошибок. Если всё нормально, происходит удаление файла с информацией о инициации обновления.

В случаи каких-то ошибок запускается механизм отката. Откат из резервной копии представляет из себя переключение на предыдущий слот и запуск скрипта обновления, но с указанием образа с резервной копии.

3 Используемые средства разработки

В качестве среды разработки используется Visual Studio Code, документация в [16].

3.1 Среда разработки Visual Studio Code

Visual Studio Code (VSCode) – это бесплатный, кроссплатформенный, легковесный, но мощный редактор кода, разработанный Microsoft. Он предназначен для разработки на множестве языков программирования и поддерживает различные расширения, которые значительно увеличивают его функциональность. Главное окно VSCode показано на рисунке 7.

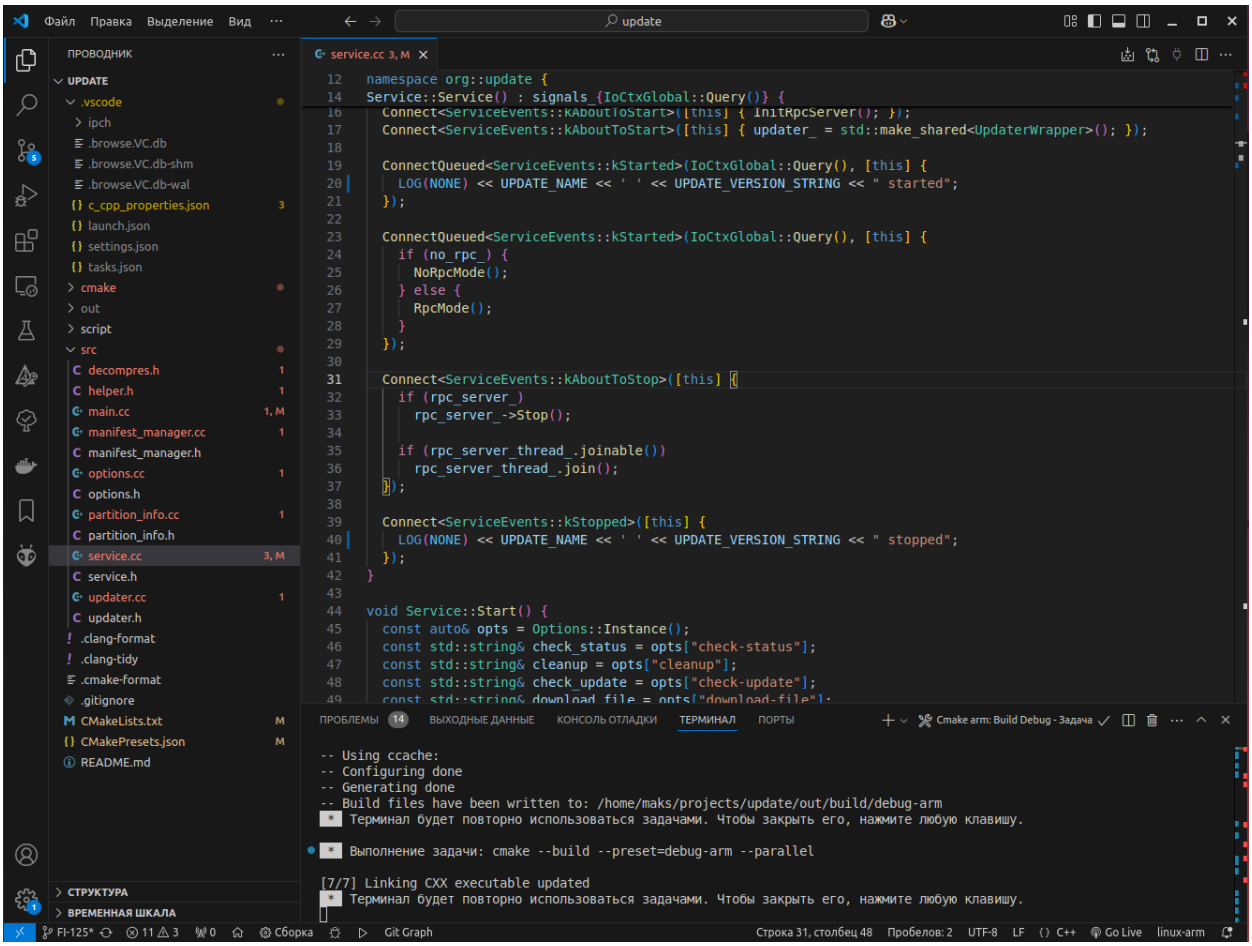

Рисунок 7 – окно VSCode

3.1.1 Основные особенности VSCode

Кроссплатформенность:

Доступен для Windows, macOS и Linux.

Широкая поддержка языков программирования:

– JavaScript,
– TypeScript,
– Python,
– C++,
– C#,
– Java,
– Go,
– PHP.

И многих других. Можно добавлять поддержку новых языков через расширения.

Редактор кода с интеллектуальными возможностями:

– Подсветка синтаксиса;
– Автодополнение (IntelliSense) на основе типов, модулей и документации;
– Переход по коду (к определению, ссылкам, структуре файлов).

Отладка кода:

– Встроенная поддержка отладки для JavaScript, TypeScript и Node.js;
– Возможность интеграции с отладчиками для Python, C++, Java и других языков.

Встроенный терминал:

– Позволяет запускать команды прямо в редакторе без необходимости переключаться между окнами.

Интеграция с Git и другими системами контроля версий:

– Поддерживает работу с Git: коммиты, история изменений, разрешение конфликтов;

– Можно подключать другие SCM через расширения.

Гибкость и кастомизация:

– расширяется с помощью Marketplace, где доступно множество плагинов (темы, новые функции, интеграции с сервисами);
– настройка пользовательских горячих клавиш, тем оформления и параметров редактора.

Поддержка контейнеров и удалённой работы:

– разработка в Docker-контейнерах через Remote Containers;
– подключение к удалённым серверам через SSH или WSL (Windows Subsystem for Linux).

Легковесность и высокая производительность:

– работает быстрее, чем полноценные IDE, но предоставляет схожие возможности.

3.1.2 Плагины VSCode

Поддержка в VSCode главным образом реализована через плагины, на рисунке 8 показана часть установленных плагинов и в том числе те, что отвечают за C++.

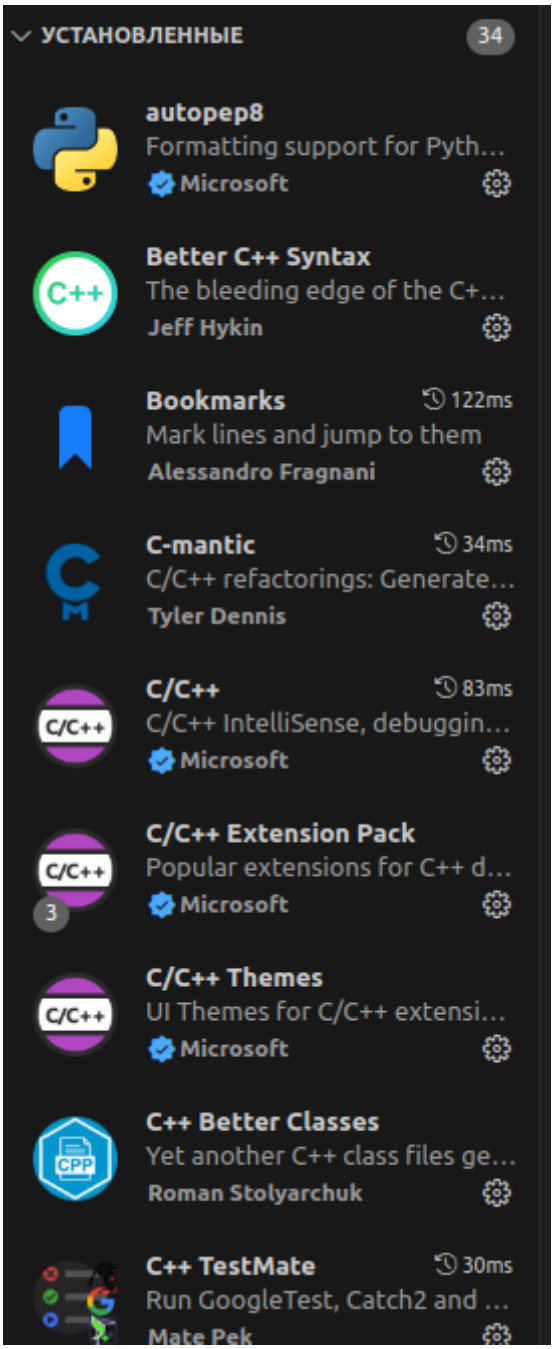

Рисунок 8 – часть списка плагинов

3.1.3 Задачи VSCode

В VSCode сборка проекта главным образом осуществляются через задачи. На рисунке 9 показана выплывающие меню, содержащие задачи.

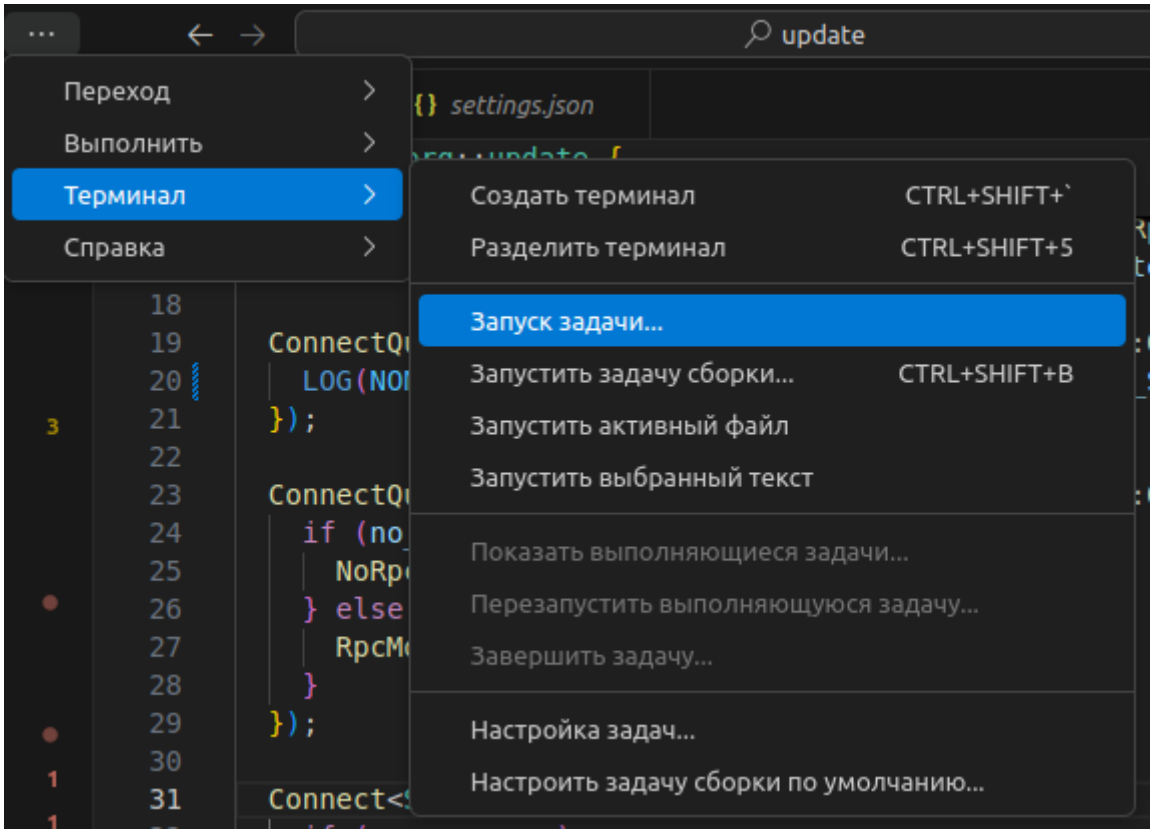

Рисунок 9 – выплывающее меню с задачами

Список задач можно увидеть на рисунке 10. Все задачи в этом списке созданы вручную, более подробно как создаются задачи описано в следующие главе.

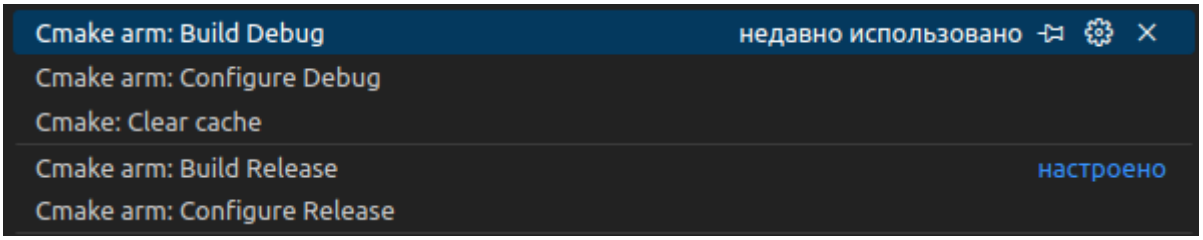

Рисунок 10 – выплывающее меню с задачами

4 Разработка программного обеспечения

Данный раздел содержит подробное описание разработки программного обеспечения с техническими нюансами.

4.1 Разработка интерфейса пользователя

Разрабатываемое приложение не содержит привычного интерфейса пользователя, а имеет интерфейс командной строки и grpc api для межпроцесного взаимодействия.

Приложение принимает следующие опции:

- io-threads – количество потоков для boost asio;
- rpc-name – имя микросервиса;
- rpc-deadline – максимальное время запроса rpc;
- check-status – проверка статуса обновления;
- cleanup – очистка временных директорий;
- check-update – проверить наличия обновлений в облаке;
- download-file – скачать обновления из облака;
- prepare-update – подготовка к обновлению;
- update – начало установки обновления;
- rollback – восстановление предыдущей версии.

Для опций используется класс Options который представляет из себя хранилище параметров используя шаблон CRTP [5].

```
#pragma once

#include <helper/options_adapter.h>

class Options : public OptionsAdapter<Options> {
 public:
  void AddOptions();
};

void Options::AddOptions() {
  AddOption("io-threads", "t", "IO threads", 2);
  AddOption("rpc-name", "RPC service name", "update-service");
  AddOption("rpc-deadline", "RPC deadline", 100u);
  AddOption("check-status", "check update/rollback status, cleanup and
exit (no rpc)", "");
  AddOption("cleanup", "cleanup temp, downloads folder and exit (no
rpc)", "");
  AddOption("check-update", "check for updates and exit (no rpc)",
"");
  AddOption("download-file", "download update file and exit (no rpc)",
"");
  AddOption("prepare-update", "prepare update and exit (no rpc)", "");
  AddOption("update", "update and exit (no rpc)", "");
  AddOption("rollback", "rollback and exit (no rpc)", "");
}
```

Листинг 1 – опции запуска

Как видно из листинга 1 у опций есть краткая форма и описание.

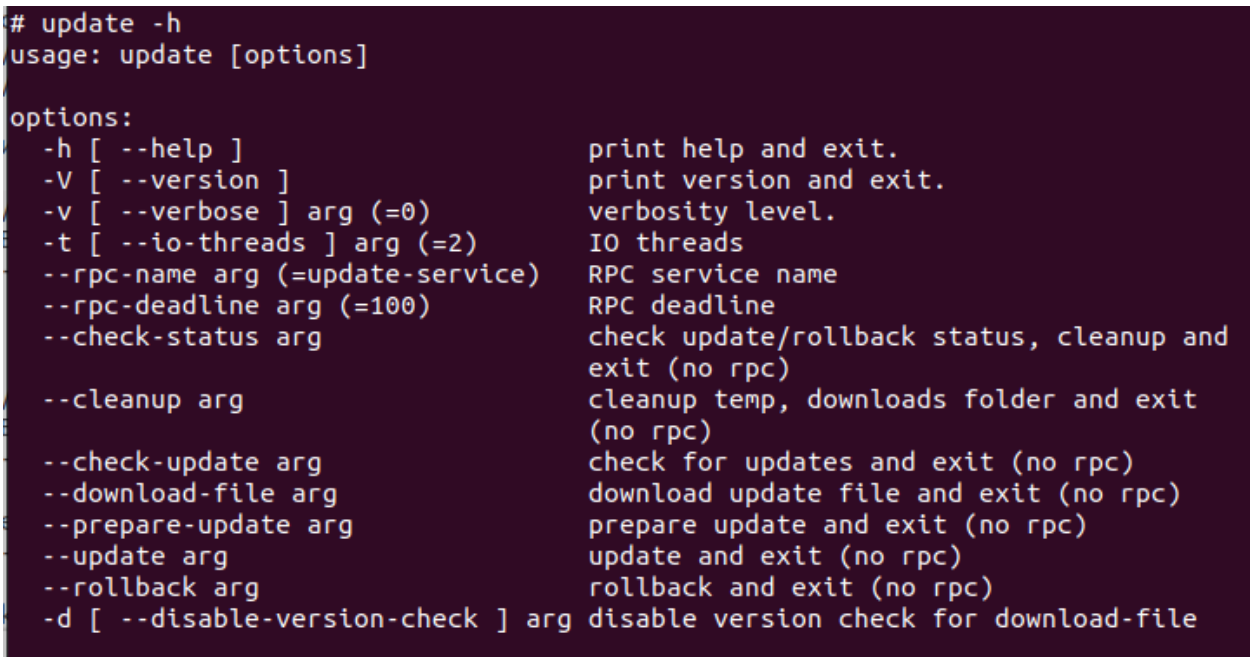

Рисунок 11 – параметры запуска

Часть опций реализует сам OptionsAdapter, например вывод справки или вывод версии.

```
const auto& opts = Options::Instance();
const std::string& rpc_name = opts["rpc-name"];
```

Листинг 2 – получение опции в приложении

На листинге выше показан пример получения опции.

В grpc реализовано несколько методов:

– CheckUpdates – возвращает наличие новой версии и её описание;

– OnlineUpdate – скачивает обновление и устанавливает его;

– OfflineUpdate – применяет обновление.

Данные функции описаны в файле update_service.proto.

```
syntax = "proto3";
package rpc;
option cc_enable_arenas = true;

message CheckUpdatesRes {
  string new_version = 1;
}
message OfflineUpdateReq {
  string path = 1;
}

service Config {
  rpc CheckUpdates(Empty) return (CheckUpdatesRes);
  rpc OnlineUpdate (OnlineUpdateReq) return (Empty);
  rpc OfflineUpdate (OfflineUpdateReq) return (Empty);
}
```

Листинг 3 – proto файл

Из листинга выше видно, что при обновлении из файла нужно предварительно на устройство загрузить файл обновлений и потом вызвать этот rpc метод указав путь до файла в сообщении.

```
grpc::Status RPC:: CheckUpdates (const rpc::Empty*, rpc::
CheckUpdatesRes * response) {
  if (!init_)
    return grpc::Status(grpc::StatusCode::UNAVAILABLE, "Server
configure, retry later");
response->set_ new_version(version_);

return grpc::Status::OK;
}
```

Листинг 4 – метод rpc сервера

По примеру на листинге 4 обрабатываются все rpc методы. Сначала проверяется атомарный флаг init_ что все данные инициализированы, далее уже идёт работа с данными и возврат статуса rpc.

```
RPC::RPC() {
  const auto& opts = Options::Instance();
  const std::string& rpc_name = opts["rpc-name"];
  const auto listen_addr = fmt::format("unix:///tmp/org-{}.sock",
rpc_name);

  rpc_server_ = std::make_unique<rpc::UpdateServiceImpl>(listen_addr);

  rpc_server_->Register<rpc::UpdateServiceMethods::kCheckUpdates>(
      [this](auto request, auto response) { return
CheckUpdates(request, response); });

  rpc_server_->Register<rpc::UpdateServiceMethods::kOnlineUpdate>(
      [this](auto request, auto response) { return
OnlineUpdate(request, response); });

  rpc_server_->Register<rpc::UpdateServiceMethods::kOfflineUpdate>(
```

```
      [this](auto request, auto response) { return
OfflineUpdate(request, response); });

  rpc_server_->Build();
}
```

Листинг 5 – создание и запуск rpc сервера

Как видно из листинга 5 процесс создания и запуска rpc сервера разбит на несколько этапов:

– Полученные опций и создание полного пути сокета;

– Создание сервера с указанием пути до сокета;

– Регистрация методов в сервере;

– Запуск сервера.

Rpc сервер в методе Build сам создаёт поток в котором он и работает, сам же объект сервера хранится в умном указателе unique_ptr.

Для форматирования строки пути до сокета используется библиотека fmt [7]. Хотя C++ 20 поддерживает форматирование текста, но версия компилятора в sdk не имеет такой поддержки.

Типы данных request и response вычисления автоматически компилятором, тем самым сокращая код.

4.2 Описание основы приложения

В основе используется приложения используется сигналы из библиотеки boost. Реализовано 4 события, которые перечислены в enum:

– kAboutToStart – испускается перед стартом;

– kStarted – старт приложения;

– kAboutToStop – сигнал перед остановкой;

– kStopped – сигнал после остановки.

Часть кода используемые эти сигналы:

```
Connect<ServiceEvents::kAboutToStart>([this] { InitSignalHandlers();
});
Connect<ServiceEvents::kAboutToStart>([this] { InitRpcServer(); });
Connect<ServiceEvents::kAboutToStart>([this] {
curl_global_init(CURL_GLOBAL_DEFAULT); });
Connect<ServiceEvents::kAboutToStart>([this] {
partition_info_.Initialization(); });
```

Листинг 6 – подготовка к запуску

```
  ConnectQueued<ServiceEvents::kStarted>(IoCtxGlobal::Query(), [this]
{
    LOG(NONE) << UPDATE_NAME << ' ' << UPDATE_VERSION_STRING << "
started";
  });
```

Листинг 7 – сообщение о запуске

```
  Connect<ServiceEvents::kAboutToStop>([this] {
    if (rpc_server_)
      rpc_server_->Stop();
    if (rpc_server_thread_.joinable())
```

```
      rpc_server_thread_.join();
  });
```

Листинг 8 – подготовка к остановке

```
  Connect<ServiceEvents::kStopped>([this] {
    LOG(NONE) << _UPDATE_NAME << ' ' << UPDATE_VERSION_STRING << "
stopped";
  });
```

Листинг 9 – сообщение о остановке

Данный подход позволяет гибко настроить запуск и остановку приложения. Так же из кода выше видно, что сигналы могут обрабатываться как синхронно, так и асинхронно. Для всех асинхронных событий используется boost io_context который можно получить через класс IoCtxGlobal. Для обработки сигналов операционной системы используется метод InitSignalHandlers показанный в листинге 10.

```
void Service::InitSignalHandlers() {
  LOG(TRACE) << "InitSignalHandlers";

  signals_.Add(SIGINT, [this] { Stop(); });
  signals_.Add(SIGTERM, [this] { Stop(); });
  signals_.Add(SIGQUIT, [this] { Stop(); });
  signals_.AsyncWait();
}
```

Листинг 10 – обработка сигналов ОС

Для чтения опций используется класс Options, его использование показано в листинге 11.

```
const auto& opts = Options::Instance();
const std::string& check_status = opts["check-status"];
const std::string& cleanup = opts["cleanup"];
const std::string& check_update = opts["check-update"];
const std::string& download_file = opts["download-file"];
const std::string& prepare_update = opts["prepare-update"];
const std::string& update = opts["update"];
const std::string& rollback = opts["rollback"];
```

Листинг 11 – чтение параметров запуска

Таким образом запуск приложения осуществляется методом Start, а остановка методом Stop.

```
void Service::Start() {
  Emit<ServiceEvents::kAboutToStart>();
  Emit<ServiceEvents::kStarted>();

  if (IoCtxGlobal::Stopped())
    IoCtxGlobal::Start();  // blocks
}
```

Листинг 12 – запуск приложения

```
void Service::Stop() {
  Emit<ServiceEvents::kAboutToStop>();

  if (!IoCtxGlobal::Stopped())
    IoCtxGlobal::Stop();
```

```
  Emit<ServiceEvents::kStopped>();
}
```

Листинг 13 – остановка приложения

Из листинга 12 и 13 видно, что для запуска и остановки приложения нужно вызвать соответствующие сигналы и не забыть про IoCtxGlobal.

4.3 Описание процесса скачивания обновления

Для проверки и скачивания обновления используется библиотека curl.

cURL – это утилита командной строки и библиотека (libcurl) для передачи данных по различным протоколам, таким как HTTP, HTTPS, FTP и другие. Она поддерживает аутентификацию, прокси, загрузку и отправку файлов, работу с заголовками и куками. Пример кода работы с curl показан в листинге 14.

```
  static constexpr auto CurlDeleter = [](CURL* curl) {
curl_easy_cleanup(curl); };
  std::unique_ptr<CURL, decltype(CurlDeleter)> curl(curl_easy_init(),
CurlDeleter);

  std::string read_buffer;

  if (curl) {
    curl_easy_setopt(curl.get(), CURLOPT_URL, api_url.c_str());
    curl_easy_setopt(curl.get(), CURLOPT_WRITEFUNCTION,
WriteCallback);
    curl_easy_setopt(curl.get(), CURLOPT_WRITEDATA, &read_buffer);

    CURLcode res = curl_easy_perform(curl.get());
```

```
    if (res != CURLE_OK) {
      LOG(ERROR) << "error: " << curl_easy_strerror(res);
      return {};
    } else {
// обработка ответа
      return {uri, version};
    }
  }
```

Листинг 14 – пример работы с curl

Для разбора ответа используется nlohmann/json.

nlohmann/json – это удобная и популярная библиотека для работы с JSON в C++. Она предоставляет простой синтаксис для парсинга, социализации и манипуляции JSON-данными, используя стандартные контейнеры STL.

Часть функции отвечающий за разбор ответа показана на листинге 15.

```
nlohmann::json parsed_data = nlohmann::json::parse(read_buffer);

      int quantity = parsed_data["meta"]["quantity"];

      if (quantity < 1) {
        LOG(ERROR) << "firmware list is empty";
        return {};
      }

      std::string id = parsed_data["data"][0]["id"];
      std::string uri = parsed_data["data"][0]["uri"];
      std::string version = parsed_data["data"][0]["version"];
      std::string tag_name = parsed_data["data"][0]["tag_name"];
```

Листинг 15 – пример работы с nlohmann/json

Метод CheckNewUpdate проверяет с помощью curl наличие свежего обновления, а метод DownloadUpdate скачивает обновление в временную папку для дальнейшего разбора.

4.4 Подготовка к обновлению

Первым этапом подготовке идёт проверка временной директории и очистка её в случаи необходимости.

Дальше идёт этап распаковки архива и для этого используется библиотеки libarchive и zlib.

```
struct archive* a = archive_read_new();
struct archive_entry* entry;
int r;

archive_read_support_format_cpio(a);
archive_read_support_filter_gzip(a);

r = archive_read_open_filename(a, archive_path.c_str(), 10240);

if (r != ARCHIVE_OK) {
  LOG(ERROR) << "no input file: " << archive_path;
   archive_read_free(a);
  return false;
}
```

Листинг 16 – подготовка libarchive к распаковке архива

Далее в цикле извлекаются файлы и записываются на диск.

```
  while ((r = archive_read_next_header(a, &entry)) == ARCHIVE_OK) {
    std::string filename = archive_entry_pathname(entry);
    std::string full_path = output_dir + filename;

    LOG(DEBUG) << "extracting: " << full_path;

    std::ofstream outfile(full_path, std::ios::binary);

    if (!outfile) {
      LOG(ERROR) << "error creating file: " << full_path;
      archive_read_free(a);
      return false;
    }

    ssize_t size;
    while ((size = archive_read_data(a, buffer.data(), buffer.size()))
> 0) {
      outfile.write(buffer.data(), size);
    }

    outfile.close();

    if (size < 0) {
      LOG(ERROR) << "error reading data: " << archive_error_string(a);
      break;
    }
  }
```

Листинг 17 – извлечение файлов

После успешного извлечения файлов создаётся ManifestManager для работы с манифест файлом. ManifestManager это простой класс для удобного чтения и предоставления данных из манифест файла, для чтения данных использует библиотеку yaml-cpp.

```
  auto manifest = std::make_unique<ManifestManager>();
  manifest->ReadManifest(temp_path + "/manifest.yml");

  if (!manifest->IsOk()) {
    LOG(ERROR) << "error parse manifest";
    return false;
  }
```

Листинг 18 – чтение манифест файла

Манифест содержит список образов для обновления с дополнительными данными к ним. Первым делом необходимо проверить что все образ, описанные в манифесте удачно извлеклись и для этого необходимо проверить sha256 каждого из образов.

```
for (size_t i{}; i < manifest->GetImages().size(); i++) {
    LOG(TRACE) << "calculate sha256: " << temp_path << manifest-
>GetImages()[i].filename;
    std::string sha256 = CalculateSHA256(temp_path + manifest-
>GetImages()[i].filename);
    if (!sha256.empty()) {
      if (manifest->GetImages()[i].sha256 == sha256) {
        LOG(DEBUG) << "file " << manifest->GetImages()[i].filename <<
" ok";
      } else {
        LOG(ERROR) << "file " << manifest->GetImages()[i].filename <<
" not ok";
        return false;
      }
    } else {
      LOG(ERROR) << "error calculate SHA256, file :" << manifest-
>GetImages()[i].filename;
      return false;
```

```
  }
}
```

Листинг 19 – проверка образов в цикле

Для расчёта sha256 используется библиотека openssl [12].

```
SHA256_CTX sha256;
SHA256_Init(&sha256);

const size_t bufferSize = 64 * 1024;
std::vector<char> buffer(bufferSize);

while (file) {
  file.read(buffer.data(), bufferSize);
  std::streamsize bytesRead = file.gcount();
  if (bytesRead > 0) {
    SHA256_Update(&sha256, buffer.data(), bytesRead);
  }
}

SHA256_Final(hash, &sha256);
```

Листинг 20 – расчёт sha256 с помощью openssl

Дальнейшим шагом необходимо создать файлы part.txt и slot.txt. Эти файлы позволяют скрипту установки правильно установить образы на нужные разделы и переключить загрузку системы.

Файл part.txt содержит список образов с соотнесёнными к ним разделами, на которые необходимо установить образы. ManifestManager предоставляет к каждому образу его тип и по этому типу и текущему слоту

системы с помощью класса PartitionInfo можно узнать раздел куда необходимо установить образ. Текущий слот можно узнать с помощью приложения bootctl.

Файл slot.txt содержит новый номер слота, с которого загрузится после установки обновлений, слот узнаём с помощью bootctl.

```
int GetSlotNumber() {
  namespace bp = boost::process;
  try {
    bp::ipstream pipe_stream;

    bp::child c("/usr/bin/bootctl -p", bp::std_out > pipe_stream);

    std::string line;
    if (pipe_stream && std::getline(pipe_stream, line)) {
      c.wait();

      int slot_number = std::stoi(line);
      return slot_number;
    }
  } catch (...) {
    return -1;
  }
  return -1;
}
```

Листинг 21 – чтение текущего слота

Следующим этапом следует создание бэкапа. Бэкап требуется только для раздела с приложениями, остальные же разделы в двух экземплярах. Этап создания бэкапа немного напоминает этап подготовки, также создаются part.txt и slot.txt, так как для установки обновления и отката используется один и тот же скрипт. Только теперь slot.txt содержит слот не новый, а текущий, а в

part.txt только раздел с приложениями. Текущий образ раздела с приложениями создаётся с помощью dd прямо во время работы, так как этот раздел только для чтения.

Образы с обновлениями и файлы part.txt, slot.txt находятся в одной папке. А бэкап создаётся в отдельную папку, где только образ oem и так же файлы part.txt, slot.txt.

4.5 Установка обновления

Для обновления первым делом необходимо проверить наличие файлов part.txt и slot.txt. А далее достаточно запустить скрипт install.sh с указанием нужной директории.

Скрипт обновления первым шагом проверяет что part.txt и slot.txt на месте.

```
if [[ ! -f ${PART_FILE} || ! -f ${SLOT_FILE} ]]; then
      do_log "Not found part.txt or slot.txt"
      do_fail
fi
```

Листинг 22 – проверка наличия part.txt и slot.txt

Дальше идёт остановка всех приложения и размонтирование oem.

```
if mountpoint -q /opt/oem; then
    do_log "/opt/oem смонтирован, отмонтируем..."
    umount /opt/oem
    if [ $? -ne 0 ]; then
        do_log "Ошибка при отмонтировании /opt/oem"
        exit 1
    fi
```

```
else
    do_log "/opt/oem уже отмонтирован, продолжаем..."
fi
```

Листинг 23 – размонтирование раздела с приложениями

Следующий шаг – это запись разделов из part.txt.

```
while read -r IMAGE PARTITION; do
     do_log "Устанавливаем ${IMAGE} на ${PARTITION}..."
     dd if="${UPDATE_DIR}/${IMAGE}" of="/dev/${PARTITION}" bs=512
     if [ $? -ne 0 ]; then
          do_log "Ошибка записи образа ${IMAGE} на ${PARTITION}"
          do_fail
     fi
done < ${PART_FILE}
```

Листинг 24 – запись образов

После записи образов устанавливается новый слот для загрузки.

```
/usr/bin/bootctl --set-slot="$SLOT"
if [ $? -ne 0 ]; then
    do_log "Ошибка при установке слота системы на $SLOT"
    do_fail
fi
```

Листинг 25 – установка нового слота системы

Ну а далее уже идёт перезапуск системы с помощью команды reboot. Если обновление установилось удачно, то система загрузится с нового слота и все приложения из oem запустятся.

4.6 Откат системы

При невозможности загрузки с нового слота u-boot после нескольких попыток загрузит старый слот. Или же если новый слот загрузился, но не все необходимые приложения смогли загрузится тогда средство обновления запустит откат системы.

Откат системы не сильно отличается от самого обновления, отличие только в том, что скрипту установки обновления в параметрах передаётся директория с бэкапом который был создан на этапе подготовки к обновлению, а не с самим обновлением.

Бэкап системы созданный во время подготовки обновления не удаляется никогда, а только перезаписывается новым бэкапом во время следующего обновления. Тем самым при необходимости откат системы можно запустить вручную в любое время.

4.7 Демонстрация сборки

Для сборки проекта используем задачи что были описаны ранее, процесс конфигурации и сборки показаны на рисунках 12 и 13.

```
*  Выполнение задачи: cmake --preset=debug-arm /home/maks/projects/update

Preset CMake variables:

  CMAKE_BUILD_TYPE="Debug"
  CMAKE_EXPORT_COMPILE_COMMANDS="ON"
  CMAKE_TOOLCHAIN_FILE:FILEPATH="/opt/arm_sdk/sdk/usr/share/buildroot/toolchainfile.cmake"
  UPDATE_TARGET="ipc-debug"

Preset environment variables:

  LD_LIBRARY_PATH="/opt/arm_sdk/sdk/lib"

-- The CXX compiler identification is GNU 12.3.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /opt/arm_sdk/sdk/usr/bin/arm-buildroot-linux-gnueabihf-g++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Selected target: ipc-debug
-- SDK path: /opt/arm_sdk/sdk
-- Arch: arm
-- Found PkgConfig: /opt/arm_sdk/sdk/usr/bin/pkg-config (found version "1.6.3")
-- Found Protobuf: /opt/arm_sdk/sdk/usr/arm-buildroot-linux-gnueabihf/sysroot/usr/lib/libprotobuf.so (found version "3.21.
12")
-- Found ZLIB: /opt/arm_sdk/sdk/usr/arm-buildroot-linux-gnueabihf/sysroot/usr/lib/libz.so (found version "1.2.13")
-- Found OpenSSL: /opt/arm_sdk/sdk/usr/arm-buildroot-linux-gnueabihf/sysroot/usr/lib/libcrypto.so (found version "1.1.1u")

-- Found Threads: TRUE
-- Found RE2 via pkg-config.
-- Found Boost: /opt/arm_sdk/sdk/usr/arm-buildroot-linux-gnueabihf/sysroot/usr/include (found version "1.82.0") found comp
onents: system log log_setup timer program_options date_time filesystem thread regex chrono atomic
-- Found Helper: 0.3.0
-- Found nlohmann_json: /opt/arm_sdk/sdk/usr/arm-buildroot-linux-gnueabihf/sysroot/usr/share/cmake/nlohmann_json/nlohmann_
jsonConfig.cmake (found suitable version "3.11.2", minimum required is "3.2.0")
-- Found CURL: /opt/arm_sdk/sdk/usr/arm-buildroot-linux-gnueabihf/sysroot/usr/lib/libcurl.so (found version "8.1.2")
-- Checking for module 'libarchive'
--   Found libarchive, version 3.6.2
-- Logger channel name: update
-- Build type: Debug
-- Install path: /opt/arm_sdk/sdk/usr/arm-buildroot-linux-gnueabihf/sysroot/opt/update
-- Using ccache:
-- Configuring done
-- Generating done
-- Build files have been written to: /home/maks/projects/update/out/build/debug-arm
*  Терминал будет повторно использоваться задачами. Чтобы закрыть его, нажмите любую клавишу.
```

Рисунок 12 – конфигурация проекта

```
*  Выполнение задачи: cmake --build --preset=debug-arm --parallel

[7/7] Linking CXX executable updated
*  Терминал будет повторно использоваться задачами. Чтобы закрыть его, нажмите любую клавишу.
```

Рисунок 13 – сборка проекта

4.8 Демонстрация запуска

В данном разделе показан этап подготовке к обновлению, так как этот этап наиболее сложный. Само же обновление после подготовке может быть и выполнено вручную запуском скрипта или же с помощью ручной установки образов.

С устройством взаимодействие происходит через ssh.

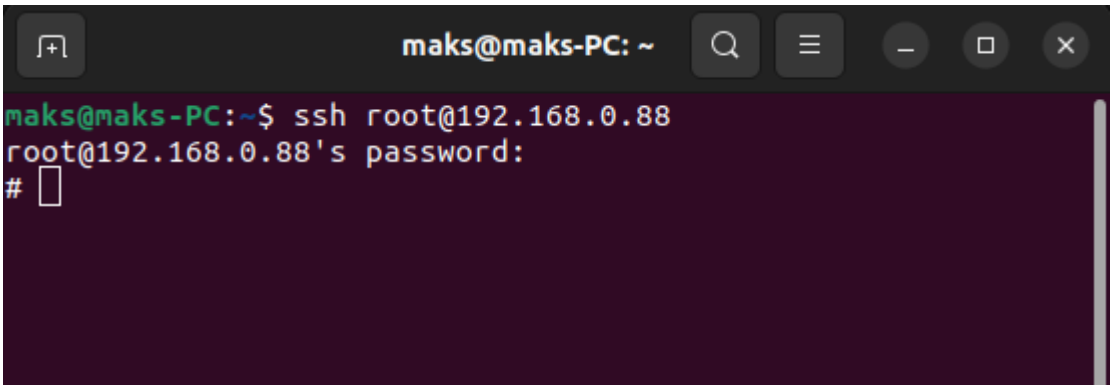

Рисунок 14 – подключение к устройству

Для начала нужно проверить наличие директории для нужд обновлений, рисунок 15.

Рисунок 15 – директория для нужд обновления

Перед запуском ещё нужно проверить временную директорию куда подготавливаются образы после распаковки, рисунок 16.

```
# pwd
/opt/update/temp
# ls
```

Рисунок 16 – временная директория

Заблаговременно нужно поместить архив с обновлением в папку с загрузками, рисунок 17.

```
# cd /opt/update/downloads
# ls
current.cpio.gz
```

Рисунок 17 – директория с загрузками

Запуск приложения в режиме справки показан на рисунке 18.

```
# update -h
usage: update [options]

options:
  -h [ --help ]                     print help and exit.
  -V [ --version ]                  print version and exit.
  -v [ --verbose ] arg (=0)         verbosity level.
  -t [ --io-threads ] arg (=2)      IO threads
  --rpc-name arg (=update-service)  RPC service name
  --rpc-deadline arg (=100)         RPC deadline
  --check-status arg                check update/rollback status, cleanup and
                                    exit (no rpc)
  --cleanup arg                     cleanup temp, downloads folder and exit
                                    (no rpc)
  --check-update arg                check for updates and exit (no rpc)
  --download-file arg               download update file and exit (no rpc)
  --prepare-update arg              prepare update and exit (no rpc)
  --update arg                      update and exit (no rpc)
  --rollback arg                    rollback and exit (no rpc)
  -d [ --disable-version-check ] arg disable version check for download-file
```

Рисунок 18 – возможные параметры запуска

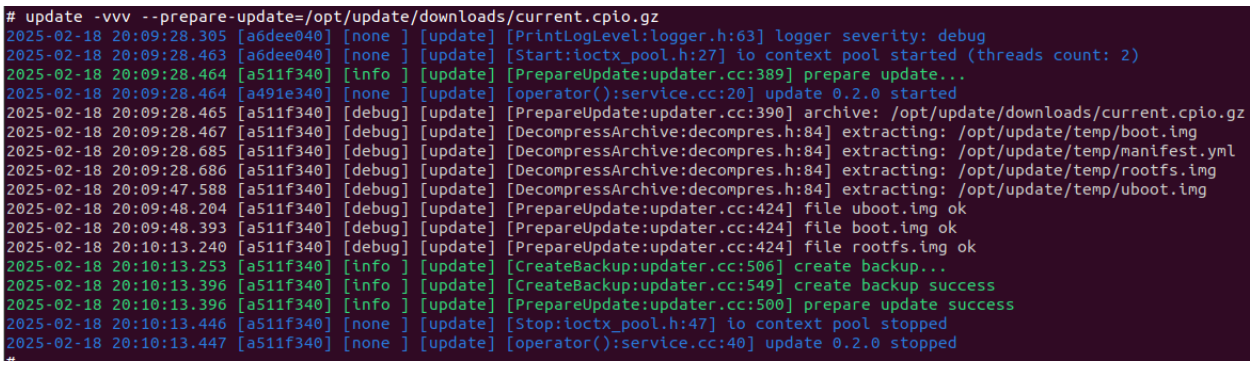

Рисунок 19 – запуск подготовки обновлений

На рисунке 19 показан запуск подготовки обновления с указанием архива с обновлением. Как видно из вывода в консоль, сначала файлы извлекаются, а потом каждый образ проверятся. Результат работы можно увидеть на рисунке 20.

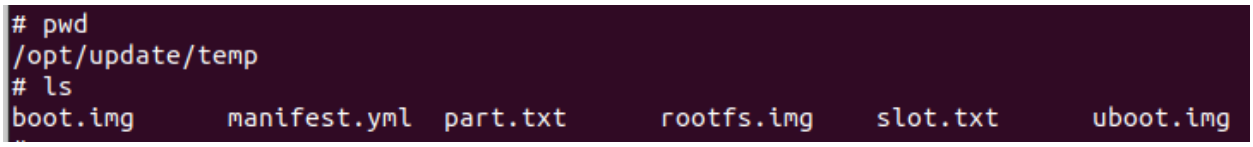

Рисунок 20 – распакованные файлы образов

По мимо образов и манифест файла появились part.txt и slot.txt необходимые для скрипта обновления.

Заключение

В результате данной выпускной квалификационной работы был проведён анализ предметной области, входных данных, изучен механизм A/B обновлений и исходя из этих сведений разработано приложение.

Разработанное приложение имеет как консольный режим, так и grpc интерфейс, имеет возможность скачивать обновления с облака, а также производить установку и локального архива содержащий обновления.

Все поставленные задачи выполнены, а пути их решения описаны в данной работе, а именно:

- Проведён анализ требований;
- Изучены предоставленные входные данные и требуемые выходные;
- Выполнено проектирование с использованием uml, а также созданы основные диаграммы;
- Выбраны средства разработки;
- Разработано приложение;
- Показана демонстрация запуска и работы.

Созданное приложение продемонстрировало стабильность и эффективность во время работы.

Не маловажным плюсом разработанного приложения является расширяемость благодаря простой, но удобной архитектуре.

Список используемых источников

1. A/B – Бесшовные A/B-обновления в Android: как они устроены [Электронный ресурс]. URL: https://habr.com/ru/companies/sberdevices/articles/521036/ (дата обращения: 22.05.2025).

2. A/B (Android Open Source Project): Жизненный цикл обновления A/B [Электронный ресурс]. URL: https://source.android.com/devices/tech/ota/ab?hl=ru#life-of-an-a-b-update (дата обращения: 22.05.2025).

3. Buildroot – Wikipedia: Buildroot [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/Buildroot (дата обращения: 22.05.2025).

4. CPIO – Архивация в Linux: как это работает [Электронный ресурс]. URL: https://habr.com/ru/articles/130092/ (дата обращения: 22.05.2025).

5. CRTP – Паттерн Curiously Recurring Template Pattern в C++ [Электронный ресурс]. URL: https://habr.com/ru/companies/otus/articles/803601/ (дата обращения: 22.05.2025).

6. curl – Command line tool and library for transferring data with URLs [Электронный ресурс]. URL: https://curl.se/ (дата обращения: 22.05.2025).

7. fmt – A modern formatting library [Электронный ресурс]. URL: https://github.com/fmtlib/fmt (дата обращения: 22.05.2025).

8. gRPC – A high performance, open source universal RPC framework [Электронный ресурс]. URL: https://grpc.io/ (дата обращения: 22.05.2025).

9. Gzip – Wikipedia: GNU zip [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/Gzip (дата обращения: 22.05.2025).

10. libarchive – Multi-format archive and compression library [Электронный ресурс]. URL: https://www.libarchive.org/ (дата обращения: 22.05.2025).

11. nlohmann_json – JSON for Modern C++ [Электронный ресурс]. URL: https://github.com/nlohmann/json (дата обращения: 22.05.2025).

12. OpenSSL – Cryptography and SSL/TLS Toolkit [Электронный ресурс]. URL: https://www.openssl.org/ (дата обращения: 22.05.2025).

13. SHA-2 – Wikipedia: Secure Hash Algorithm 2 [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/SHA-2 (дата обращения: 22.05.2025).

14 U-Boot – Wikipedia: Das U-Boot [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/Das_U-Boot (дата обращения: 22.05.2025).

15. UML – Wikipedia: Unified Modeling Language [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/UML (дата обращения: 22.05.2025).

16. Visual Studio Code – Code Editing [Электронный ресурс]. URL: https://code.visualstudio.com/ (дата обращения: 22.05.2025).

17. yaml-cpp – YAML parser and emitter in C++ [Электронный ресурс]. URL: https://github.com/jbeder/yaml-cpp (дата обращения: 22.05.2025).

18. zlib – Wikipedia: zlib compression library [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/Zlib (дата обращения: 22.05.2025).

19. Гущина О.М., Очеповский А.В., Рогова Н.Н. Прикладная информатика. Бизнес-информатика. Выполнение бакалаврской работы: электронное учебно-методическое пособие. Тольятти: Изд-во ТГУ, 2022.

20. Сайт ТГУ [Электронный ресурс]. URL: https://www.tltsu.ru/ (дата обращения: 22.05.2025).