МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ, КОМП‘ЮТЕРНОЇ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ
Кафедра Комп’ютерних інформаційних технологій

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач випускової кафедри
Аліна САВЧЕНКО
«____» _____ 2022р.

КВАЛІФІКАЦІЙНА РОБОТА

(ДИПЛОМНА РОБОТА, ПОЯСНЮВАЛЬНА ЗАПИСКА)
ВИПУСКНИКА ОСВІТНЬОГО СПУПЕНЯ **“МАГІСТР”**
ЗА ОСВІТНЬО-ПРОФЕСІЙНОЮ ПРОГРАМОЮ
“ІНФОРМАЦІЙНІ УПРАВЛЯЮЧІ СИСТЕМИ ТА ТЕХНОЛОГІЇ”

Тема: “Технологія оновлення IoT систем на базі Embedded Linux”

Виконавець: студент УС-212М Кондрат Максим Сергійович

Керівник: д. т. н., доцент Савченко Аліна Станіславівна

Нормоконтролер: Ігор РАЙЧЕВ

КИЇВ – 2022

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет *Кібербезпеки, комп'ютерної та програмної інженерії*

Кафедра *Комп'ютерних інформаційних технологій*

Галузь знань, спеціальність, спеціалізація: 12 "Інформаційні технології", 122 "Комп'ютерні науки", "Інформаційні управляючі системи та технології"

ЗАТВЕРДЖУЮ
Завідувач випускової кафедри
_____Аліна САВЧЕНКО
"_____"_____2022р.

ЗАВДАННЯ

на виконання кваліфікаційної роботи студента

Кондрата Максима Сергійовича
(прізвище, ім'я, по батькові)

1. **Тема роботи:** «Технологія оновлення IoT систем на базі Embedded Linux» затверджена наказом ректора від "28" вересня 2022р. за № 1774/ст.

2. **Термін виконання роботи:** з 26.09.2022 по 21.11.2022р.

3. **Вихідні дані до роботи:** організація процесу оновлення IoT системи.

4. **Зміст пояснювальної записки:** вступ, аналітичний огляд і постановка задачі, огляд технологій та інструментів для оновлення системи, розробка функціональності системи та результат роботи системи, висновки.

5. **Перелік обов'язкового графічного матеріалу:** стратегія «Подвійна копія», файлова структура системи, етапи розробки системи.

6. Календарний план-графік.

№ п/п	Завдання	Термін виконання	Підпис керівника
1	Отримання завдання на дипломну роботу, створення плану дипломної роботи та побудова плану-графіку виконання робіт.	26.09.2022 – 28.09.2022	
2	Огляд та аналіз наукової літератури по темі дипломної роботи та написання Розділу 1.	29.09.2022 – 09.10.2022	
3	Написання Розділу 2 дипломної роботи.	10.10.2022 – 20.10.2022	
4	Написання Розділу 3 дипломної роботи. Завершення створення пояснювальної записки дипломної роботи.	21.10.2022 – 31.10.2022	
5	Оформлення та друк пояснювальної записки.	01.11.2022 – 07.11.2022	
6	Створення презентації, доповіді та підготовка до захисту дипломної роботи.	08.11.2022 – 15.11.2022	
7	Підготовка матеріалів дипломної роботи для передачі секретарю ДЕК (папка, конверт, диск із файлом диплому, рецензія, відгук).	16.11.2022 – 18.11.2022	

7. Дата видачі завдання: «26» вересня 2022 р.

Керівник кваліфікаційної роботи _____ Аліна САВЧЕНКО
(підпис керівника) (П.І.Б.)

Завдання прийняв до виконання _____ Максим КОНДРАТ
(підпис випускника) (П.І.Б.)

РЕФЕРАТ

Пояснювальна записка до дипломної роботи «Технологія оновлення IoT систем на базі Embedded Linux» складається зі вступу, трьох розділів, висновку, списку бібліографічних посилань та п’яти додатків і містить 80 сторінок та 46 рисунків. Список бібліографічних посилань складається з 29 найменувань.

Ключові слова: ОНОВЛЕННЯ СИСТЕМИ, ТЕХНОЛОГІЇ ОНОВЛЕННЯ, EMBEDDED LINUX, C++, PYTHON, BASH.

Актуальність. Оновлення – це процес заміни продукту на новішу версію того самого продукту. У комп’ютерній техніці та споживчій електроніці оновлення – це, як правило, заміна апаратного забезпечення, програмного забезпечення або мікро-програми на новішу або кращу версію, щоб оновити систему або покращити її характеристики.

Метою дипломної роботи є розробка технології оновлення для IoT систем на базі Embedded Linux, яка дозволяє покращити ефективність оновлення за рахунок розділення скриптів для виконання на серверній та апаратній частині й пересилання тільки різниці між версіями прошивки плати.

Для досягнення поставленої мети необхідно вирішити такі **завдання**:

- написати скрипт mkUpdatePatch.sh, який створює патч (.patch) оновлення;
- створити патч оновлення на серверній частині, який містить тільки різницю між версіями прошивки плати (Raspberry Pi). mkUpdatePatch.sh, що дозволяє скоротити обсяг оновлень і, відповідно, підвищити ефективність процесу;
- створити патч оновлення на серверній частині відбувається за допомогою трьох вхідних параметрів: версії, з якої оновлюється система; версії, на яку оновлюється; назва патчу;
- визначити процес пересилання патчу на плату;

– створити скрипт оновлення для апаратної частини з одним вхідним параметром – назвою патчу оновлення, який розпочинає процес оновлення.

Об’єкт дослідження: процес оновлення для IoT систем.

Предмет дослідження: технології оновлення для IoT систем на базі Embedded Linux.

Методи дослідження включають у себе:

– методи оновлення програмного забезпечення;
– методи створення патчів оновлення;
– методи перевірки оновлення.

Теоретичною основою дипломної роботи стали вітчизняні та зарубіжні дослідження щодо забезпечення оновлення програмного забезпечення та публікації на сайтах, присвячені питанням оновлення систем.

Теоретична і практична значимість роботи полягає в тому, що на основі отриманих знань:

1) по зібраному матеріалу можна сформувати методичний посібник для студентів по оновленню IoT систем т створенню патча оновлення;
2) розроблено технологію оновлення для IoT систем на базі Embedded Linux.

На захист виносяться наступні положення:

1) процес оновлення ПЗ;
2) процес створення патчу оновлення;
3) тестування оновлення.

ЗМІСТ

ВСТУП 8
РОЗДІЛ 1. АНАЛІТИЧНИЙ ОГЛЯД І ПОСТАНОВКА ЗАДАЧІ 10
1.1. Методи OTA оновлення IoT систем 10
1.1.1. Рівні оновлень 13
1.1.2. Стратегії управління пам'яттю 15
1.2. Огляд OTA оновлення IoT систем на базі Embedded Linux 18
1.3. Огляд Embedded систем на основі ядра Linux 20
1.4. Постановка задачі 25
1.5. Висновки до розділу 1 27
РОЗДІЛ 2. ТЕХНОЛОГІЇ ТА ІНСТРУМЕНТИ ДЛЯ ОНОВЛЕННЯ IOT СИСТЕМ НА БАЗІ EMBEDDED LINUX 28
2.1. Технології для розробки функціональної складової системи 28
2.1.1. C++ 28
2.1.2. Python 31
2.1.3. Make 33
2.1.4. Bash сценарії 35
2.1.5. Systemd 36
2.1.6. FTP 38
2.1.7. Linux Utilities 40
2.2. Середовище для розробки системи 41
2.2.1. Visual Studio Code 41
2.2.2. MobaXterm 44
2.3. Висновки до розділу 2 46
РОЗДІЛ 3. РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТЕХНОЛОГІЇ ОНОВЛЕННЯ IOT СИСТЕМ НА БАЗІ EMBEDDED LINUX 47
3.1. Розробка функціональної складової системи 48
3.1.1. Розробка системи для серверної частини 48
3.1.1.1. Створення скрипта mkUpdatePatch.sh 49
3.1.1.2. Створення скрипта mkDelta.sh 51
3.1.2. Процес пересилання патчу на плату 52

3.1.3. Розробка системи для апаратної частини 54
3.1.3.1. Розробка файлу UpdateUnpack.cpp 55
3.1.3.2. Розробка файлу UpdateMoveFile.cpp 56
3.1.3.3. Розробка файлу ScriptChecker.cpp 58
3.1.3.4. Розробка файлу ScriptExecution.cpp 59
3.1.3.5. Розробка скрипта flashUpdate.sh 60
3.1.3.6. Розробка скрипта bankSwitch.sh 61
3.1.3.7. Розробка скрипта applyXdelta.sh 62
3.2. Результат роботи системи оновлення IoT систем 64
3.2.1. Результат роботи системи на серверній частині 64
3.2.2. Результат роботи системи на апаратній частині 66
3.3. Висновки до розділу 3 68
ВИСНОВКИ 69
СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ 71
ДОДАТОК А 73
ДОДАТОК Б 75
ДОДАТОК В 77
ДОДАТОК Г 78
ДОДАТОК Д 79

ВСТУП

Концепція IoT (Internet of Things – Інтернет речей) сповнене світовою індустрією технологій, і очікується, що до 2025 року в усьому світі буде підключено 41 мільярд пристроїв. З огляду на велику кількість підключених пристроїв, які мають доступ до Інтернету через різні засоби, такі як WiFi, Ethernet, 4G і тощо, цілком очевидно, що ці підключені пристрої повинні мати можливість працювати пліч-о-пліч з новими пристроями.

Оскільки вже непрактично замінювати наявний IoT-пристрій щоразу, коли на ринку з'являється нова версія, найкращий можливий спосіб додати функції та виконувати завдання технічного обслуговування – оновити програмне забезпечення і файли на IoT-пристроях. Це може не лише покращити існуючу функціональність пристроїв, наприклад виправлення помилок, але й також може додати певні нові функції програмного забезпечення.

Метою дипломної роботи є розробка технології оновлення для IoT систем на базі Embedded Linux, яка дозволяє покращити ефективність оновлення за рахунок розділення скриптів для виконання на серверній та апаратній частині та пересилання тільки різниці між версіями прошивки плати.

Об'єктом дослідження даної роботи є процес оновлення для IoT систем.

Предмет дослідження: технології оновлення для IoT систем на базі Embedded Linux.

Для досягнення поставленої мети необхідно вирішити такі задачі:

- написати скрипт mkUpdatePatch.sh, який створює патч (.patch) оновлення;
- створити патч оновлення на серверній частині, який містить тільки різницю між версіями прошивки плати (Raspberry Pi). mkUpdatePatch.sh, що дозволяє скоротити обсяг оновлень і, відповідно, підвищити ефективність процесу;

- створення патчу оновлення на серверній частині відбувається за допомогою трьох вхідних параметрів: версії, з якої оновлюється система; версії, на яку оновлюється; назва патчу;
- визначити процес пересилання патчу на плату;
- створити скрипт оновлення для апаратної частини з одним вхідним параметром – назвою патчу оновлення, який розпочинає процес оновлення.

РОЗДІЛ 1. АНАЛІТИЧНИЙ ОГЛЯД І ПОСТАНОВКА ЗАДАЧІ

Оновлення пристроїв не завжди потрібні, але важко придумати будь-яке програмне забезпечення, у якому не було б помилок. Навіть якщо ваше програмне забезпечення ідеальне, але якщо пристрій зв'язується в мережі чи Інтернеті з будь-якими бібліотеками з відкритим кодом, оновлення системи безпеки може стати необхідністю.

1.1. Методи OTA оновлення IoT систем

Оновлення – це процес заміни продукту на новішу версію того самого продукту. У комп'ютерній техніці та споживчій електроніці оновлення – це, як правило, заміна апаратного забезпечення, програмного забезпечення або мікро-програми на новішу або кращу версію, щоб оновити систему або покращити її характеристики.

Що стосується пристроїв IoT, існує два основних способи виконання оновлень:

- фізичне оновлення;
- OTA ((Firmware) Over The Air – по повітрю) оновлення.

Найтрадиційніший спосіб оновлення пристроїв це фізичний доступ до пристрою та виконання оновлення/оновлення. Для невеликої вбудованої системи IoT це може бути лише підключення пристрою до комп'ютера та виконання оновлення. Це може виконуватись до вбудованого комп'ютера, наприклад або одноплатного комп'ютера Raspberry Pi через локальну мережу, встановлення SSH або з'єднання з віддаленим робочим столом і виконання оновлення [1].

Вирішуючи обмеження щодо фізичного оновлення, оновлення OTA використовуються для віддаленого оновлення пристроїв IoT, яке можна виконувати за допомогою бездротових засобів зв’язку, зменшуючи зусилля людини.

В основному існує три основні методи OTA-оновлень:

– оновлення Edge-to-Cloud OTA (E2C);
– оновлення Gateway-to-Cloud OTA (G2C);
– оновлення Edge-to-Gateway-to-Cloud OTA (E2G2C).

Оновлення E2C (рис.1.1) використовують підключення до пристрою IoT для прямого зв’язку з віддаленим сервером і отримання оновлень безпосередньо з сервера. Пристрої IoT, які здебільшого призначені для споживачів, потрапляють у цю категорію, і з’єднання також просте в основному завдяки легкому доступу до Інтернету через Wi-Fi та LAN [1].

Рис.1.1. Схема оновлення E2C

Трохи складніший, але безпечніший метод у порівнянні з методом E2C метод оновлення G2C (рис.1.2) шлюз-хмара включає шлюз IoT-середника для отримання, обробки та розповсюдження оновлень мікропрограми на підключені «вузли» IoT.

Цей метод здебільшого використовується, коли підключені пристрої IoT не можуть самостійно обробити оновлення та не мають прямого підключення до Інтернету.

Пристрої, які використовують методи оновлення G2C OTA, це банкомати, системи дистанційного моніторингу енергоспоживання та інші банківські та

фінансові послуги, такі як кіоски. Цей метод оновлення OTA покращує безпеку системи, оскільки пристрої захищені від зовнішніх атак і вразливостей [1].

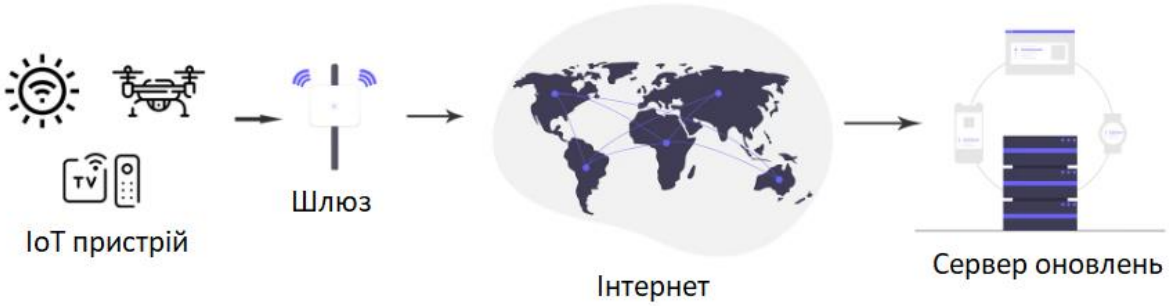

Рис.1.2. Схема оновлення G2C

У методі E2G2C (рис.1.3) оновлень OTA від пристрою до шлюзу в хмару пристрої IoT можуть встановлювати підключення до Інтернету через центральний шлюз і запитувати оновлення через нього. Шлюз – це єдиний пристрій, підключений до Інтернету та запитуючий оновлення. У цьому випадку пристрої IoT повинні мати можливість виконувати оновлення самостійно. Цей метод використовується такими пристроями, як польові датчики (датчики температури, вологості, датчики погоди) та інші промислові сенсорні системи управління [1].

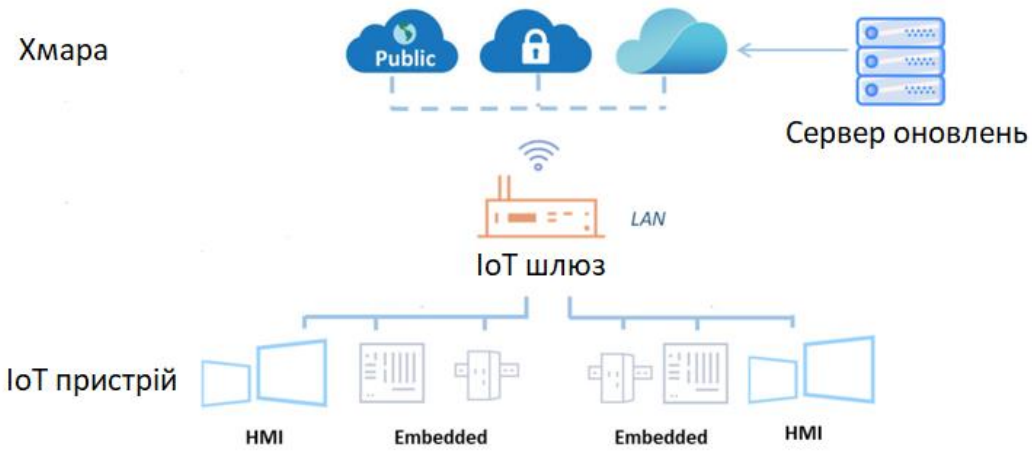

Рис.1.3. Схема оновлення E2G2C

Говорячи про оновлення Linux, можливо побачити «блокові» та «файлові» системи оновлення. Це стосується оновлення всього розділу за раз шляхом запису безпосередньо на блоковий пристрій або оновлення окремих файлів для виконання оновлення.

У Embedded Linux оновлення на базі блоків є правильним шляхом через їх атомарність і той факт, що вся файлова система зазвичай є виходом системи збірки Embedded Linux. Простір для зберігання на кожному вбудованому пристрої буде постійним для певного продукту, тому щоразу ми створюємо розділ однакового розміру. Цей тип оновлення поєднується з наявністю якогось запасного або відновлення образу [2].

1.1.1. Рівні оновлень

Embedded пристрої майже повсюдно використовують флеш-пам'ять для зберігання. Флеш-пам'ять можна розділити на розділи подібно до жорстких дисків. Такі компоненти, як ядро, DTB і RAM-диск, можуть зберігатися в окремому розділі, але частіше всі вони зберігаються разом у кореневій файловій системі. Файлова система має такий формат, як UBIFS або ext4, і міститься в розділі. Отже, ви можете застосовувати оновлення на кількох різних рівнях: файл, пакет або образ розділу. Для кожного є свої плюси і мінуси [3].

Оновлення файлів за файлами є найбільш очевидним підходом, і справді багато проектів мають власний засіб оновлення на основі tar-архівів і сценаріїв, який застосовуватиме оновлення файл за файлом. Вони майже завжди не відповідають деяким критеріям, основний з них це атомарність. Атомарність вимагає ретельного кодування, щоб переконатися, що оновлення окремих файлів є атомарними. Техніка полягає в тому, щоб записати новий вміст у тимчасовий файл, а потім використати функцію перейменування POSIX (набір стандартів, які описують інтерфейси між операційною системою та застосунками), щоб замінити старий файл на новий. Функція перейменування гарантує, що обидва знаходяться в одному розділі, а файлова система сумісна з POSIX. Однак, навіть коли вирішити цю проблему,

виникає більша проблема з забезпечення атомарності над групою оновлень файлів, що вимагає знання про залежності між ними та про те, як відкотити невдале часткове оновлення. Це складна проблема, і її можна частково вирішити за допомогою менеджера пакетів.

Менеджери пакетів зберігають достатньо функцій, щоб мати можливість виявити невдале оновлення та відкотитися до попередньої версії. Вони можуть керувати залежностями між пакетами та виконувати оновлення в живій системі, тому перезавантажуватись потрібно лише в тому випадку, якщо ядро або його модулі були змінені. Вони ведуть контроль того, що було встановлено та коли. Таким чином, менеджери пакетів відповідають принаймні деяким вимогам програми оновлення і завжди є кращим варіантом, ніж домашній засіб оновлення.

Але embedded пристрій відрізняється від сервера, так як існує більша ймовірність невдалого оновлення через втрату живлення або підключення до мережі, і менеджери пакетів не можуть гарантувати відновлення в усіх таких випадках. Це означає, що є шанс, що пристрій не завантажиться після оновлення. У центрі обробки даних, як крайній засіб, ви можете підійти до машини та завершити оновлення вручну. Але embedded пристрої часто знаходяться у віддалених місцях або важкодоступні з різних причин, що робить ручне втручання дорогим.

Таким чином, менеджери пакетів є гарним вибором, якщо є часті невеликі оновлення системи, де випадкові збої оновлення прийнятні, і де ви можете виконати додаткову перевірку якості, необхідну для забезпечення спільного поширення версій програмного забезпечення.

Оновлення цілого розділу, що містить образ файлової системи або двійковий файл ядра, добре вписується в спосіб поєднання більшості embedded дистрибутивів Linux.

Оновлення образу має ту перевагу, що ви можете бути впевнені, що всі пристрої в полі мають однакову збірку програмного забезпечення. Застосування оновлення досить просте, тому реалізація агента оновлення також може бути досить простою. Це означає, що легко підтвердити, що він працює та є надійним

Але оскільки ви оновлюєте цілі розділи, а не файли, розмір оновлення зазвичай більший, ніж пакетна система. Також пам’ятайте, що неможливо оновити живий образ файлової системи, а це означає, що оновлення має бути записано в альтернативний розділ і система має перезавантажитися, перш ніж оновлення набере чинності [3].

1.1.2. Стратегії управління пам'яттю

Ключовим фактором успішного оновлення через OTA є фактор того що пристрій не залишиться у непридатному для використання стані (якщо, наприклад, сталося відключення електроенергії). Вирішити цю проблему можна переконавшись, що завжди можна «повернутися» до попередньої системи, якщо щось у процесі оновлення піде не так. Існує 2 стратегії гарантованої роботи системи: «єдина копія/сховище» та «подвійне копіювання/сховище».

Якщо на сховищі достатньо місця для збереження двох копій всього програмного забезпечення, можна гарантувати, що робоча копія завжди буде, навіть якщо оновлення програмного забезпечення переривається або відбувається відключення живлення.

Кожна копія повинна містити ядро, кореневу файлову систему та кожен інший компонент, який можна оновлювати. Потрібен механізм, щоб визначити, яка версія працює [4].

Механізм роботи полягає в тому, що одна копія буде зберігати програмне забезпечення, яке було вже встановлене на системі. Друга копія зберігатиме нову оновлену систему (рис.1.4).

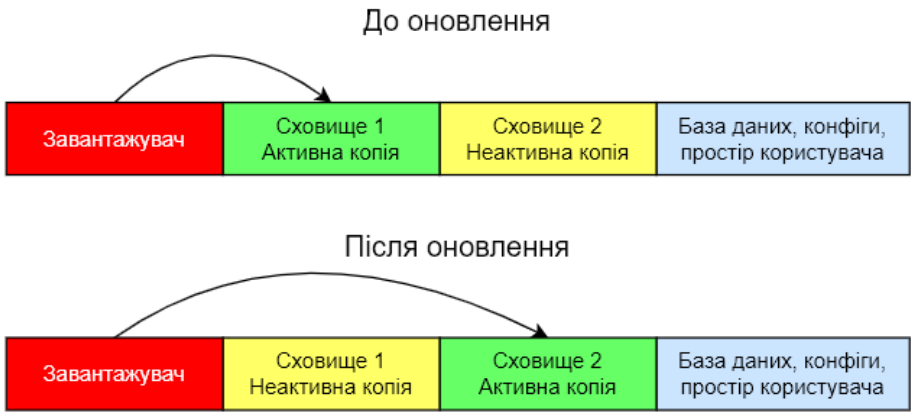

Рис.1.4. Стратегія «Подвійна копія»

Типове OTA-оновлення з використанням стратегії «Подвійне копіювання» містить такі кроки:

– пристрій виконує регулярну перевірку на серверах OTA та отримує сповіщення про наявність оновлення;
– застосовує оновлення до неактивної частини;
– пристрій перезавантажується, а активна та неактивна частини міняються місцями, це означає, що ми зараз завантажуємо нашу оновлену частину, яка раніше була неактивною, а активну частину позначено як неактивну, і саме тут будуть записуватися послідовні оновлення.

Механізм для оновлення програмного забезпечення методом «однієї копії» полягає в тому, що також створюється інша копія в «ОС відновлення», що складається з ядра (можливо, зменшене видаливши непотрібні драйвера) і невеликої кореневої файлової системи з програмою та її бібліотеками. Загальний розмір набагато менший, ніж одна копія системного програмного забезпечення в методі «подвійного копіювання» (рис.1.5).

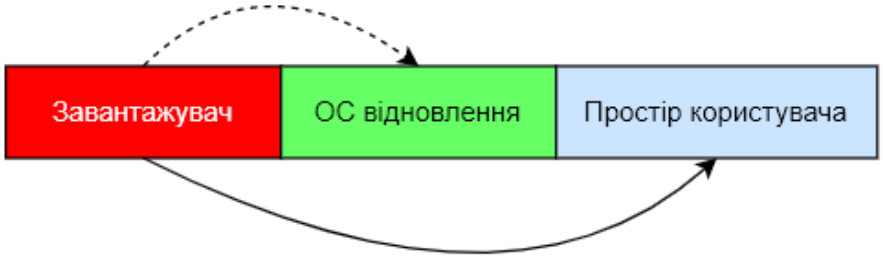

Рис.1.5. Стратегія «Єдина копія»

Типове OTA-оновлення з використанням стратегії «Одна копія» містить такі кроки:

- пристрій виконує регулярну перевірку на серверах OTA та отримує сповіщення про наявність оновлення;
- оновлення завантажень до розділу даних. Користувачеві буде запропоновано встановити оновлення;
- пристрій перезавантажується в «ОС відновлення»;
- дані витягуються з образу оновлення, розташованого в розділі даних, і використовуються для оновлення системи;
- пристрій перезавантажується;
- щойно оновлений завантажувальний розділ завантажується, він монтується та починає виконувати двійкові файли в нещодавно оновленій «Звичайній ОС».

Є кілька очевидних недоліків використання стратегії «Єдиної копії»:

- під час оновлення пристрій не можна використовувати;
- необхідно зарезервувати місце для збереження образу оновлення під час перезавантаження в «ОС відновлення»;
- немає резервного варіанту, тобто якщо програмне забезпечення, яке ми встановили, є несправним або встановлено з помилками, або було перервано, ми отримаємо пристрій, який не завантажує «Звичайну ОС», а замість цього лише завантажує «ОС відновлення» в якій може повторити

спробу оновлення, але якщо нове програмне забезпечення несправне, повторна спроба не допоможе, і потрібна взаємодія користувача.

Є й плюси використання стратегії «Єдиної копії»: порівняно зі стратегією «Подвійне копіювання» вона споживає менше місця для зберігання, і якщо у вас дуже мало місця для зберігання, це може бути єдиним життєздатним варіантом [4].

1.2. Огляд OTA оновлення IoT систем на базі Embedded Linux

Технології оновлення OTA існують у будь-якій галузі, яка включає пристрої, підключені до Інтернету, а не лише в Інтернеті речей (IoT). Будь-який смартфон отримує оновлення OTA, кожен сервер у всьому світі також підтримується дистанційно та підтримується через механізми OTA та багато інших. Кожен із цих продуктів працює однаково та специфічно, тому метод надсилання OTA-оновлення для певного сервера точно такий же, як і для будь-якого іншого сервера на землі. Подібним чином метод надсилання OTA-оновлення для кожної мобільної програми однаковий.

На відміну від цього, якщо розглядати сектор IoT, то ситуація інша. Кожен пристрій поводиться та працює в унікальній архітектурі, отже, механізм надсилання оновлень через OTA має бути «розумним» і достатньо загальним, щоб мати справу з будь-якою ситуацією.

Потрібно пам’ятати, що на відміну від серверів або мобільних додатків, де в разі збою оновлення завжди є хтось, хто може відреагувати та вирішити проблему, перезавантаживши пристрій або навіть зателефонувавши до центру підтримки, у багатьох ситуаціях в галузі Інтернету речей це не так. Ніхто не може навіть наблизитися до пристрою, і потенційна шкода набагато вища. Крім того, час простою пристроїв IoT іноді може завдати серйозної шкоди, наприклад, фабрика буде паралізована в результаті зупинки конвеєра через збій оновлення програмного забезпечення [5].

Незважаючи на ризики надсилання оновлень програмного забезпечення OTA, кожному виробнику Інтернету речей очевидно, що в наші дні немає можливості

продавати розумний продукт без відповідного механізму оновлення OTA. Існує 2 основних підходи до такого механізму:

Старий метод – підключення до Інтернету стандартним способом оновлення вбудованих продуктів було фізичне підключення до кожного пристрою, підключення його до комп'ютера за допомогою кабелю та записування нового програмного забезпечення в його пам'ять. Оскільки ця процедура досить складна і коштує великих грошей, кількість оновлень, які розгорталися на практиці, була вкрай малою. Насправді більшість продуктів ніколи не отримають жодного оновлення програмного забезпечення протягом усього терміну служби.Щойно ці продукти були підключені до Інтернету, природно, багато компаній запровадили механізм OTA на основі методу, з яким вони вже знайомі. Це означає, що з кожним оновленням вони надсилають всю операційну систему (ОС) і замінюють існуючу на нову. Єдина перевага цієї реалізації полягає в тому, що в ній немає сюрпризів. Все, що існує всередині нової ОС, буде працювати на пристрої (звичайно, якщо оновлення пройшло успішно) [5].

У цього методу є ряд недоліків:

- Розмір оновлення. Оскільки надсилається вся ОС, розмір кожного оновлення може становити сотні мегабайт, іноді кілька гігабайт. Це може перешкодити часу розгортання, але найбільша проблема полягає в тому, що він використовуватиме багато Інтернет-даних. Більшість пристроїв IoT підключено до Інтернету через стільникову мережу та можуть мати низькі тарифні плани. Тому розгортання оновлення, яке замінює всю ОС, не має сенсу.
- Проблеми з підключенням до даних. Якщо під час оновлення щось піде не так, існує ризик, що пристрій може більше не вмикатися. Це часто називають «цегляним» пристроєм. Деякі користувачі можуть не встановлювати оновлення OTA через цей ризик.
- Оновлення вимагають перезавантаження. Перезапуск пристрою спричиняє небажані простої, іноді призводячи до втрати доходу через час простою (як у заводському прикладі вище).

Отже, хоча Інтернет, можливо, прискорив доступність і доставку оновлень через OTA, через проблеми, описані вище, кількість оновлень, які надсилаються за допомогою цього методу, залишається низькою [6].

Новий метод – надсилання оновлень лише тієї частини ОС, яку ми хочемо змінити. Наприклад, надсилання лише оновленого двійкового файлу нашої програми або надсилання оновлення для оновлення певного пакета чи модуля, який використовує наша програма. На відміну від старого методу , цей має лише один недолік. Якщо не керувати оновленнями організовано та зрозуміло, ви можете дуже швидко потрапити в ситуацію, коли не знаєте, яку версію розгорнуто на якому пристрої.

Використання цього методу має багато переваг:

- Розмір оновлення зменшується. У більшості випадків розмір OTA-оновлення буде дуже малим. Оскільки ви надсилаєте лише частину ОС, яку потрібно оновити, оновлення є крихітним. Таким чином, пристрої, які мають обмежені тарифні плани Інтернету, можуть легко оновити своє програмне забезпечення, не турбуючись про те, що дані вичерпаються.
- Перезавантаження не потрібно. У деяких випадках програмам навіть не потрібно припиняти роботу після впровадження оновлення. Це запобігає небажаним простоям і незручностям для користувачів.
- У 99% випадків, якщо оновлення OTA виходить з ладу, пристрій не буде заблоковано. Розумний метод оновлення OTA розпізнає, коли під час оновлення виник збій, і повертається до попередньої стабільної версії програмного забезпечення [6].

1.3. Огляд Embedded систем на основі ядра Linux

Linux – це ім'я, дане ядру та ряду операційних систем Unix під GNU GPL. Багато в чому Linux схожий на інші операційні системи, такі як Windows, macOS або iOS. Як і вони, Linux може мати графічний інтерфейс і ті самі типи настільного програмного

забезпечення, до якого ви звикли, наприклад, текстові процесори, редактори фотографій, редактори відео тощо.

Але Linux також відрізняється багатьма важливими параметрами. Перш за все, і, мабуть, найважливіша його особливість, це програмне забезпечення з відкритим кодом. Код, який використовувався для створення Linux, є безкоштовним і доступним для перегляду, редагування та – для обізнаних користувачів – додавання до нього.

Ще одна істотна відмінність полягає в тому, що Linux, хоча основні частини операційної системи Linux загалом широко поширені, багато дистрибутивів Linux включають різні опції програмного забезпечення. Це означає, що Linux неймовірно настроюється. Ми можемо встановити дуже легку систему та додати все необхідне пізніше або за потреби. Користувачі також можуть вибрати основні компоненти, такі як система, яка відображає графіку, та інші компоненти інтерфейсу користувача.

Ймовірно, ви вже використовуєте Linux, навіть якщо не знаєте про це, оскільки він присутній у програмному забезпеченні великої кількості пристроїв, якими ми користуємося щодня. Навіть більшість веб-сторінок, які ми відвідуємо, ймовірно, створено серверами Linux. Подібним чином більшість компаній і окремих осіб обирають Linux для своїх серверів, оскільки він безпечний, гнучкий і може отримати чудову підтримку від великої спільноти користувачів.

Embedded (вбудована) система – це набір комп’ютерного апаратного та програмного забезпечення на основі мікроконтролера або мікропроцесора , який керується операційною системою реального часу, має обмежену пам’ять і може відрізнятися за розміром і складністю. Embedded Linux – це тип операційної системи/ядра Linux, розроблений для встановлення та використання у embedded пристроях або системах.

Незважаючи на використання того самого ядра, embedded Linux значно відрізняється від стандартної операційної системи. Перш за все, він адаптований для embedded систем і, отже, набагато менший за розміром, потребує менше процесорної потужності та має мінімальні функції. Ядро Linux модифіковано та оптимізовано як embedded версію Linux. Такий екземпляр Linux може запускати лише програми, створені спеціально для пристрою.

Embedded Linux є гнучким, недорогим, відкритим кодом і адаптується до мікропроцесорів певного призначення. Порівняно з пропрієтарними embedded операційними системами, Linux допускає декілька постачальників програмного забезпечення, розробки та підтримки; він має стабільне ядро та надає можливість читати, змінювати та перерозповсюджувати вихідний код. Це також дозволяє використовувати високомодульний підхід до створення індивідуальної системи, що забезпечує більшу гнучкість [7].

З цих причин і завдяки своїй універсальності Embedded Linux став дуже популярним серед інженерів embedded систем. Багато споживчих електронних пристроїв, таких як телефони, розумні планшети, цифрові пристрої зберігання даних, персональні відеомагнітофони, фотоапарати, переносні пристрої та багато інших , як правило, розробляються з Linux. Він використовується в програмному забезпеченні автомобілів і багатьох інших прикладах, таких як мережеве обладнання, керування машинами, промислова автоматизація, навігаційне обладнання, програмне забезпечення для польотів космічних кораблів і медичні інструменти в цілому.

Embedded Linux – це компактна версія Linux , яка пропонує функції та послуги, що відповідають вимогам роботи та застосування вбудованої системи. Як і Linux, її основними перевагами перед іншими інтегрованими операційними системами є, серед іншого, використання відкритого вихідного коду і, отже, низька вартість, наявність багатьох програм, розробників і постачальників підтримки, відкритість щодо авторських прав або ліцензій, а також стабільна і надійна ядро. Крім того, можливість читати, змінювати та розповсюджувати вихідний код у поєднанні з високомодульним підходом до розробки вбудованої системи на замовлення є великою перевагою. Це забезпечує більшу гнучкість у можливостях дизайну.

Основні переваги Embedded Linux:

- відкритий код;
- низька ціна;
- кілька постачальників програмного забезпечення, розробки та підтримки;
- висока можливість налаштування;
- висока гнучкість [7].

Серед можливих недоліків Embedded Linux варто відзначити його складність. Завдяки тому, що Embedded Linux є відкритим вихідним кодом і має багато розробок, Embedded Linux багатший функціями, ніж інші вбудовані середовища розробки. Величезна кодова база Linux призводить до майже нескінченної складності.

Крім того, ми повинні враховувати, що при розробці рішень, адаптованих до платформи, яка звикає, будь-яка модифікація апаратного забезпечення може безпосередньо вплинути на програмне забезпечення, відображаючись у ситуаціях заміни продукту або навіть в оновленнях самого програмного забезпечення. З іншого боку, пам’ятайте, що використання Embedded Linux передбачає потребу мати порівняно великий простір пам’яті порівняно з іншими середовищами. Це в основному вплине на пристрої з обмеженими ресурсами.

Сума всіх цих аспектів може призвести до збільшення грошей і часу на розробку нових реалізацій. З цієї причини надзвичайно важливо завжди покладатися на досвідчених розробників , які можуть максимально оптимізувати процес.

Основні недоліки Embedded Linux:

- наявність великої кількості функцій може призвести до більшої складності;
- модифікація апаратного забезпечення може вплинути на програмне забезпечення;
- необхідність більшого простору пам’яті [7].

Embedded Linux складається з 4 основних компонентів (рис.1.6): Bootloader (завантажувач), Linux kernel (ядро Linux), Root filesystem (коренева файлова система), Applications (програми/служби) [8].

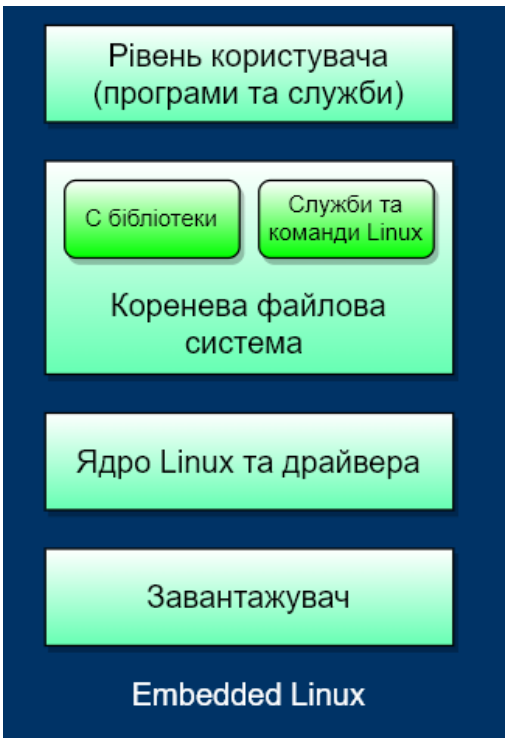

Рис.1.6. Основні компоненти Embedded Linux

Завантажувач технічно знаходиться поза системою та працює інакше, ніж настільна або серверна система, де першим запускається BIOS. У embedded системі апаратне забезпечення запускає завантажувач, відповідальний за базову ініціалізацію та виконання ядра. У embedded системах це можна зробити за допомогою: GRUB , LILO або Das U-Boot . Остання з яких використовується виключно на embedded пристроях.

Ядро Linux містить керування процесами та пам’яттю, мережевий стек, драйвери пристроїв і надає послуги для будь-яких додатків простору користувача. Завантажувач завантажує ядро в пам'ять і запускає його. Ядро шукає програму init для запуску першою. Програма init (програма в UNIX і Unix-подібних системах , що запускає всі інші процеси) відповідає за запуск інших служб, таких як драйвери обладнання, драйвери файлової системи, монтування файлової системи, служби та інші програми. Ядро знає лише програму ініціалізації, і якщо воно не знаходить її, воно викличе «kernel panic».

Ядро embedded системи ідентичне ядру більших систем, таких як настільний комп’ютер. Основна відмінність полягає в тому, що Embedded ядро Linux спеціально розроблено для роботи на іншій архітектурі ЦП.

Коренева файлова система містить бібліотеки C і служби/команди Linux, а також інші необхідні сценарії. Файлова система є інтерфейсом між ядром і програмами простору користувача. Під час ініціалізації ядро завантажує кожну програму в пам’ять і очікує, що вони будуть організовані у файли та каталоги. Це коренева файлова система, її слід створити заздалегідь і змонтувати до '/' до того, як ядро зможе запустити init.

Applications або Userland – це місце, де розміщуються будь-які спеціальні програми чи служби. Програма init також відповідає за завантаження їх у пам'ять. Більшість embedded систем є одноцільовими програмами для виконання певної функції, тому програми у embedded системах, як правило, економні та мінімальні. Для запуску Linux на цільовому вбудованому процесорі потрібно щонайменше 8 МБ оперативної пам’яті, при цьому більшість програм потребують щонайменше 32 МБ оперативної пам’яті. Фактична потреба в оперативній пам’яті може залежати від розміру вашої вбудованої програми. Окрім оперативної пам’яті, також потрібний мінімум 4 МБ пам’яті. Він може бути одного з наступних типів:

- NAND або NOR Flash;
- Картки SD або MMC [8].

1.4. Постановка задачі

Стрімкий розвиток концепції IoT та постійне збільшення кількості пристроїв ускладнює задачу підтримання актуальності їх функцій. Непрактично замінювати наявний IoT-пристрій щоразу, коли на ринку з’являється нова версія. Ефективніше додавати нові функції та виконувати завдання технічного обслуговування через оновлення програмного забезпечення і файлів на IoT-пристроях. Це може як покращити існуючу функціональність пристроїв, наприклад, виправлення помилок, так і додати певні нові функції програмного забезпечення.

Недоліком існуючих систем оновлення є складність використання та погана кастомізація, тобто процес оновлення строго стандартизований та немає інших варіантів реалізацій, в той час як даний метод може бути реалізований за бажанням користувача. Патч оновлення в даному методі може бути різних форматів та створюватися різними методами.

Враховуючи вимогу постійного оновлення програмного забезпечення і файлів на IoT-пристроях та недоліки існуючих методів актуальною є розробка технології оновлення для IoT систем на базі Embedded Linux.

Метою дипломної роботи є розробка технології оновлення для IoT систем на базі Embedded Linux, яка дозволяє покращити ефективність оновлення за рахунок розділення скриптів для виконання на серверній та апаратній частині та пересилання тільки різниці між версіями прошивки плати.

Технологія оновлення для IoT систем на базі Embedded Linux передбачає розділення скриптів на 2 групи: перша група скриптів буде виконуватись на серверній частині, друга – на апаратній. Процес оновлення буде розглянуто на прикладі одноплатного комп'ютера Raspberry Pi.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- написати скрипт mkUpdatePatch.sh, який створює патч (.patch) оновлення;
- створити патч оновлення на серверній частині, який містить тільки різницю між версіями прошивки плати (Raspberry Pi). mkUpdatePatch.sh, що дозволяє скоротити обсяг оновлень і, відповідно, підвищити ефективність процесу;
- створення патчу оновлення на серверній частині відбувається за допомогою трьох вхідних параметрів: версії, з якої оновлюється система; версії, на яку оновлюється; назва патчу;
- визначити процес пересилання патчу на плату;
- створити скрипт оновлення для апаратної частини з одним вхідним параметром – назвою патчу оновлення, який розпочинає процес оновлення.

Запропонована технологія оновлення для IoT систем на базі Embedded Linux дозволить підвищити ефективність процесу оновлення за рахунок розділення скриптів на 2 групи: перша група скриптів буде виконуватись на серверній частині, друга – на апаратній та пересилання тільки різниці між версіями прошивки плати.

1.5. Висновки до розділу 1

В даній роботі було проаналізовано методи OTA оновлення IoT систем, їх рівні та стратегію управління пам'яттю, було проаналізовано переваги та недоліки кожного методу. Оглянуто Embedded системи на основі ядра Linux, їх переваги над іншими ядрами, та проаналізовано OTA оновлення IoT систем на базі Embedded Linux.

Основними недоліками існуючих систем оновлення є складність використання та погана кастомізація, тобто процес оновлення строго стандартизований та немає інших варіантів реалізацій, в той час як даний метод може бути реалізований за бажанням користувача. Патч оновлення в даному методі може бути різних форматів та створюватися різними методами.

Запропонована технологія оновлення для IoT систем на базі Embedded Linux дозволить підвищити ефективність процесу оновлення за рахунок розділення скриптів на 2 групи: перша група скриптів буде виконуватись на серверній частині, друга – на апаратній та пересилання тільки різниці між версіями прошивки плати.

РОЗДІЛ 2. ТЕХНОЛОГІЇ ТА ІНСТРУМЕНТИ ДЛЯ ОНОВЛЕННЯ ІОТ СИСТЕМ НА БАЗІ EMBEDDED LINUX

Технологія оновлення IoT систем на базі Embedded Linux включає багато технологій та інструментів для своєї реалізації. Основними мовами програмування, які використовувались під час розробки оновлення IoT систем на базі Embedded Linux є об’єктно-орієнтовані мови C++ та Python.

Також використовуються такі технології як: Bash сценарії, Make файли, менеджер фонових сервісів systemd, протокол передачі файлів FTP та стандартні команди терміналу Linux.

Технологія оновлення IoT систем на базі Embedded Linux була створена в середовищі Visual Studio Code, а для підключення до одноплатного комп'ютера Raspberry Pi було використано програму для віддаленого робочого столу MobaXterm.

2.1. Технології для розробки функціональної складової системи

2.1.1. C++

C++ – мова програмування загального призначення з підтримкою кількох парадигм програмування: об'єктно-орієнтованої, узагальненої, процедурної та ін. Б'ярн Страуструп почав створювати C++ в AT&T Bell Laboratories у 1979 році. На етапі зародження мова мала назву «Сі з класами». Згодом Страуструп перейменував мову на C++ у 1984 р. Бере витоки з мови програмування C.

У 1990-х роках C++ став однією з найуживаніших мов програмування загального призначення. Мову використовують для системного програмування, розробки прикладного програмного забезпечення, написання драйверів, потужних серверних та клієнтських програм [9].

За своєю конструкцією C++ підходить для вбудованої розробки, оскільки мова знаходиться між програмним і апаратним забезпеченням вищого рівня, дозволяючи отримувати доступ до апаратного забезпечення та керувати ним безпосередньо, не жертвуючи перевагами мови високого рівня. Це особливо ефективно для апаратного забезпечення, яке потребує існувати деякий час, оскільки програми, написані на C++, можуть працювати десятиліттями завдяки високій стабільності мови.

C++ також дає розробникам можливість ефективно використовувати абстракції без надто великих витрат на інфраструктуру. Структура даних C++, як і C, заснована на алгоритмі, що робить його чудовим вибором для вирішення всіх маленьких головоломок, з якими ви стикаєтеся під час розробки вбудованих систем. C++ також не залежить від процесора, і мікропроцесори сьогодні постачаються з компіляторами C++ від 1 долара США.

C++ все ще є стандартною мовою/мовою за замовчуванням, яка використовується для розробки вбудованих систем, і залишатиметься такою протягом тривалого часу. Оскільки наш світ стає все більш зв'язаним через IoT, ми продовжуватимемо бачити, як C++ з'являється навколо нас. Його вже широко використовують такі великі компанії, як Google, Microsoft і Oracle [11].

Переваги C++ для вбудованих систем:

- **Простота використання**: C++ використовує об'єктно-орієнтоване програмування та має низку шаблонів та інструментів, які роблять надійним і легшим створювати частини коду для повторного використання. C++ у вбудованих системах використовується багатьма за його масштабованість і те, як легко повторно використовувати частину його коду. Його шаблони, наприклад, дозволяють інженерам створювати кілька об'єктів на основі дуже подібних типів поведінки. Вони дають вам чудовий спосіб повторного використання та масштабування вашого програмного забезпечення.
- **Портативність**: оскільки частини коду C++ легко використовувати повторно, код також можна переносити з одного пристрою на інший.

- **Стандартна бібліотека**: стандартна бібліотека C++ надає інженерам інструменти для легкого створення оптимального коду. Інженери можуть використовувати те, що вже створили інші інженери, у стандартній бібліотеці основних функцій C++. Замість того, щоб писати все це з нуля, ви просто успадковуєте ці бібліотеки, а потім налаштовуєте та будуєте на них свої частини.
- **Стабільність**: програми, написані на C++ у вбудованій системі, можуть працювати десятиліттями без збоїв завдяки стабільності мови.
- **Мова шлюзу**: C++ є основою багатьох мов таких як Python, Java, JavaScript та інших.
- **Підтримка**: C++ використовується широко, з великою спільнотою підтримки та часто оновлюється.
- **Добре підходить для GUI**: C++ пропонує особливо хороші інструменти для вбудованих систем із графічним інтерфейсом користувача (GUI) [10].

Недоліки C++ для вбудованих систем:

- **Важко вивчити**: C++ – це складна мова, яку важко вивчити. Є більше конструкцій, таких як віртуальні функції та шаблонні класи, тому більше речей для вивчення і складність їх розуміння.
- **Проблеми керування пам’яттю**: у C++ програміст контролює керування пам’яттю, що може бути недоліком. Багато мов мають збирач сміття, який допомагає керувати пам’яттю, який автоматично звільняє непотрібну пам’ять. C++ не має такої функції, тому програмісти повинні керувати нею вручну. Помилки пам'яті можуть спричинити серйозні помилки та навіть збої.

C++ може працювати так само добре або краще, ніж C, у широкому діапазоні вбудованих систем. Перевагами мови C++ є шаблони, стандартні бібліотеки та подібні інструменти, які роблять її цінною в складних системах, які інженери можуть захотіти перебудувати в інших форматах і лінійках продуктів [10].

2.1.2. Python

Python (названа на честь знаменитої комедійної групи Monty Python) – інтерпретована об'єктно-орієнтована мова програмування високого рівня зі строгою динамічною типізацією. Розроблена в 1990 році Гвідо ван Россумом. Структури даних високого рівня разом із динамічною семантикою та динамічним зв'язуванням роблять її привабливою для швидкої розробки програм, а також як засіб поєднування наявних компонентів. Python підтримує модулі та пакети модулів, що сприяє модульності та повторному використанню коду. Інтерпретатор Python та стандартні бібліотеки доступні як у скомпільованій, так і у вихідній формі на всіх основних платформах. В мові програмування Python підтримується кілька парадигм програмування, зокрема: об'єктно-орієнтована, процедурна, функціональна та аспектно-орієнтована [12].

Python став досить популярною мовою програмування за останні кілька років. Мова Python є об’єктно-орієнтованою та інтерпретованою (не скомпільованою). Цей атрибут призвів до використання Python на таких платформах, як Linux і Windows, а також на одноплатних комп’ютерах, таких як Raspberry Pi. З таким широким і зростаючим впровадженням Python став використовуватись у вбудованих системах реального часу. Нижче наведено 4 особливостей, які можна знайти для Python у вбудованих системах:

- Контроль пристрою та налагодження. У процесі розробки вбудованого програмного забезпечення розробники часто аналізують трафік шини, такий як USB, SPI або I2C. Іноді аналіз призначений просто для налагодження, але іноді виникає потреба фактично керувати аналізатором шини та надсилати повідомлення до вбудованої системи. Багато аналізаторів шин і засобів зв'язку мають зручні інтерфейси, які можна використовувати для керування інструментом. Вони також зазвичай забезпечують спосіб розробки сценаріїв, які також можна використовувати для керування інструментом. Python – це одна з мов

сценаріїв, яка зазвичай підтримується, іноді виключно, для взаємодії з інструментом і керування ним.

- Автоматизація тестування. Можливість керувати інструментами, які можуть надсилати й отримувати повідомлення від вбудованої системи за допомогою Python, відкриває можливість використання Python для створення автоматизованих тестів, включаючи регресійне тестування. Можна розробити сценарії Python, які встановлюють вбудовану систему в різні стани, встановлюють конфігурації та перевіряють усі можливі збурення та взаємодії, які система матиме із зовнішнім світом. Одна з переваг використання Python для автоматизованого тестування полягає в тому, що можна розробити регресійні тести, які постійно тестують і тренують систему. Будь-які зміни коду, які призводять до помилок або невідповідностей, будуть негайно виявлені.
- Аналіз даних. Простий пошук бібліотек Python в Інтернеті показує, що існує багато вільнодоступних потужних бібліотек для розробки програм Python. Python можна використовувати для отримання критично важливих вбудованих системних даних, які потім можна зберігати в базі даних або локально для аналізу. Потім розробники можуть використовувати Python для розробки візуалізацій у реальному часі, які показують критичні параметри, або зберігати та зберігати ці параметри для подальшого аналізу. Приємна частина аналізу даних за допомогою Python полягає в тому, що базова робота вже зроблена - функціональність просто встановлюється.
- Програмне забезпечення реального часу. Python довів, що він настільки потужний і простий у використанні, що Python навіть знаходить свій шлях до вбудованих систем реального часу як мова програмування. Саме вбудоване програмне забезпечення написано на Python, а не на C/C++. Найпоширенішою версією Python для реального часу є порт MicroPython, призначений для роботи на мікроконтролерах, таких як ARM Cortex-M3/4. Однак MicroPython не єдиний. Такі компанії, як Synapse і OpenMV,

використовують MicroPython або власний порт Python у вбудованих системах [13].

2.1.3. Make

Make – інструмент, який автоматично визначає, які вихідні файли програми потрібно перекомпілювати та/або зв'язати. Make отримує інформацію про те, як створити вашу програму з файлу під назвою makefile (або Makefile). Ви також можете використовувати make для встановлення (видалення) програм і виконання інших команд із make-файлу.

Можливості Make:

- Make дозволяє кінцевому користувачеві створювати та інсталювати ваш пакет, не знаючи подробиць того, як це робиться, оскільки ці деталі записуються у make-файлі, який ви надаєте.
- Автоматично визначає, які файли потрібно оновити, на основі того, які вихідні файли змінилися. Він також автоматично визначає належний порядок оновлення файлів, якщо один не вихідний файл залежить від іншого не вихідного файлу. Як наслідок, якщо ви змінюєте кілька вихідних файлів, а потім запускаєте Make, не потрібно перекомпілювати всю вашу програму. Він оновлює лише ті не вихідні файли, які прямо чи опосередковано залежать від вихідних файлів, які ви змінили.
- Make не обмежується жодною конкретною мовою. Для кожного не вихідного файлу в програмі makefile визначає команди оболонки для його обчислення. Ці команди оболонки можуть запускати компілятор для створення об'єктного файлу, компонувальник для створення виконуваного файлу, argдля оновлення бібліотеки або TeX або Makeinfo для форматування документації.
- Make не обмежується створенням пакета. Ви також можете використовувати Make, щоб керувати встановленням або видаленням

пакета, генерувати для нього таблиці тегів або будь-що інше, що ви хочете робити досить часто, щоб варто було записати, як це зробити [14].

Правило у make-файлі вказує Make, як виконати серію команд, щоб створити цільовий файл із вихідних файлів (рис.2.1). Він також визначає список залежностей цільового файлу. Цей список має включати всі файли (вихідні файли чи інші цілі), які використовуються як вхідні дані для команд у правилі.

Ось як виглядає просте правило:

```
ціль: залежності ...
        команди
        ...
```

Рис.2.1. Просте правило Makefile

Коли ви запускаєте Make, ви можете вказати конкретні цілі для оновлення; інакше Make оновить першу ціль, зазначеною у make-файлі. Звичайно, будь-які інші цільові файли, необхідні як вхідні дані для створення цих цілей, повинні бути спочатку оновлені.

Make використовує make-файл, щоб визначити, які цільові файли потрібно оновити, а потім визначає, які з них насправді потрібно оновити. Якщо цільовий файл є новішим за всі його залежності, це означає, що він уже оновлений і не потребує повторного створення. Інші цільові файли потрібно оновити, але в правильному порядку: кожен цільовий файл має бути повторно згенерований, перш ніж використовувати його для відновлення інших цілей.

GNU Make має багато потужних функцій для використання у make-файлах, крім того, що є в інших версіях Make. Він також може відновлювати, використовувати та видаляти проміжні файли, які не потрібно зберігати.

GNU Make також має кілька простих функцій, які дуже зручні. Наприклад, параметр -o, який каже “зробити вигляд, що файл вихідного файлу не змінився, навіть якщо він змінився”. Це надзвичайно корисно, коли ви додаєте новий макрос до файлу

заголовка. Більшість версій Make припускають, що вони повинні перекомпілювати всі вихідні файли, які використовують файл заголовка; але GNU Make дає вам спосіб уникнути повторної компіляції, якщо ви знаєте, що ваші зміни у файлі заголовка не потребують цього.

Однак найважливіша відмінність між GNU Make та більшістю версій Make полягає в тому, що GNU Make є безкоштовним програмним забезпеченням [14].

2.1.4. Bash сценарії

Bash – це програма оболонки інтерфейсу командного рядка, яка широко використовується в Linux і macOS. Назва Bash є абревіатурою від «Bourne Again Shell», розробленої в 1989 році як наступник Bourne Shell.

Оболонка (Shell) – це комп'ютерна програма, яка дозволяє безпосередньо керувати операційною системою (ОС) комп'ютера за допомогою графічного інтерфейсу користувача (GUI) або інтерфейсу командного рядка (CLI).

Bash є дуже потужним, оскільки він може спростити певні операції, які важко ефективно виконати за допомогою графічного інтерфейсу користувача. Пам'ятайте, що більшість серверів не мають графічного інтерфейсу користувача, і найкраще навчитися користуватися можливостями інтерфейсу командного рядка (CLI) [15]. Дуже часто коли говорять про bash – мають на увазі bash сценарії.

Сценарій(скрипт) bash – це звичайний текстовий файл, який містить низку команд. Ці команди є сумішшю команд, які ми зазвичай вводимо в командному рядку, і команд, які ми можемо вводити в командному рядку. Файлам, які є сценаріями Bash, прийнято надавати розширення .sh. Проте важливо пам'ятати:

Усе, що ви можете запустити звичайним чином у командному рядку, можна помістити в сценарій, і він робитиме те саме. Подібним чином, усе, що ви можете додати до сценарію, також можна нормально запустити в командному рядку, і воно виконуватиме те ж саме.

У сфері Linux (і комп'ютерів загалом) ми маємо концепцію програм і процесів. Програма – це блок двійкових даних, що складається з серії інструкцій для процесора

та, можливо, інших ресурсів (зображень, звукових файлів тощо), організованих у пакет і зазвичай зберігаються на вашому жорсткому диску. Коли ми говоримо, що ми запускаємо програму, ми насправді не запускаємо програму, а її копію, яка називається процесом. Ми копіюємо ці інструкції та ресурси з жорсткого диска в робочу пам’ять (або оперативну пам’ять). Ми також виділяємо трохи місця в оперативній пам’яті для процесу для зберігання змінних (для зберігання тимчасових робочих даних) і кількох прапорців, щоб дозволити операційній системі (ОС) керувати процесом і відстежувати його під час його виконання.

По суті, процес – це запущений екземпляр програми.

Можуть існувати кілька процесів, що представляють ту саму програму, що виконується в пам'яті одночасно. Наприклад, я міг би відкрити два термінали та запустити команду cp (копіювання) в обох. У цьому випадку в системі зараз буде два процеси cp. Після завершення роботи система знищує їх, і більше не існує процесів, що представляють програму cp.

Коли ми знаходимося на терміналі, у нас працює процес Bash, який надає нам оболонку Bash. Якщо ми починаємо виконувати сценарій, він фактично не виконується в цьому процесі, а замість цього запускає новий процес для виконання всередині [16].

2.1.5. Systemd

systemd – це система ініціалізації Linux і менеджер служб, який включає такі функції, як запуск демонів(служб) на вимогу, обслуговування точок монтування та автоматичного монтування, підтримку знімків і відстеження процесів за допомогою груп керування Linux. systemd надає демон журналювання та інші інструменти та утиліти, які допомагають виконувати типові завдання системного адміністрування.

Леннарт Поеттерінг і Кей Сіверс написали systemd, натхненний launchd і Upstart macOS , з метою створення сучасної та динамічної системи. Примітно, що systemd надає потужні можливості розпаралелювання та логіку керування службами на основі

залежностей, що дозволяє запускати служби паралельно та пришвидшує час завантаження. Ці два аспекти були присутні в Upstart, але вдосконалені системою.

systemd є системою ініціалізації за замовчуванням для основних дистрибутивів Linux, але вона зворотно сумісна зі сценаріями ініціалізації SysVinit. SysVinit – це система ініціалізації, яка передує systemd і використовує спрощений підхід до запуску служби. systemd не тільки керує ініціалізацією системи, але також надає альтернативи для інших відомих утиліт, таких як cron і syslog. Оскільки systemd робить кілька речей у просторі користувача Linux, багато хто критикує його за порушення філософії Unix, яка наголошує на простоті та модульності.

Linux вимагає системи ініціалізації під час завантаження та запуску. Наприкінці процесу завантаження ядро Linux завантажує systemd і передає йому керування, після чого починається процес запуску. Під час цього кроку ядро ініціалізує перший процес простору користувача , процес ініціалізації systemd з ідентифікатором процесу 1, а потім переходить у режим очікування, якщо не буде викликано знову. systemd готує простір користувача та переводить хост Linux у робочий стан, запускаючи всі інші процеси в системі.

Нижче наведено спрощений огляд усього процесу завантаження та запуску Linux:

- система включається. BIOS виконує мінімальну ініціалізацію обладнання та передає керування завантажувачу;
- завантажувач викликає ядро;
- ядро завантажує початковий диск RAM, який завантажує системні диски, а потім шукає кореневу файлову систему;
- після налаштування ядра запускається система ініціалізації systemd;
- systemd переймає та продовжує монтувати файлові системи хоста та запускати служби.

systemd представляє концепцію одиниць systemd, і існує кілька типів, таких як службова одиниця , монтувальна одиниця, сокетна одиниця та слайсова одиниця. Одиниці визначаються у файлах конфігурації одиниць, які містять інформацію про тип одиниці та її поведінку.

Для більшості дистрибутивів, які використовують systemd, файли модулів зберігаються в таких каталогах:

- Каталог /usr/lib/systemd/user/є місцем розташування за замовчуванням, куди пакетами встановлюються одиничні файли. Файли модулів у каталозі за замовчуванням не слід змінювати.
- Каталог /run/systemd/system/є місцем виконання файлів модуля.
- Каталог /etc/systemd/system/зберігає одиничні файли, які розширюють послугу. Цей каталог матиме пріоритет над одиничними файлами, розташованими будь-де в системі [17].

2.1.6. FTP

FTP (File Transfer Protocol, протокол передавання файлів) – стандартний мережевий протокол прикладного рівня, призначений для пересилання файлів між клієнтом та сервером в комп'ютерній мережі.

Клієнт та сервер створюють окремі канали для передачі даних та обміну командами. Можлива автентифікація клієнтів із використанням відкритого тексту, зазвичай це ім'я користувача (логін) та пароль. Також сервер може бути налаштований для роботи без автентифікації користувачів (так звані «анонімні сеанси»).

Для захисту даних (а також процесу автентифікації) використовують побудований на основі SSL/TLS варіант FTPS, або розширення протоколу SSH – SSH File Transfer Protocol (SFTP).

Перші FTP-клієнти були створені ще до появи графічного інтерфейсу користувача в операційних системах і тому мали інтерфейс командного рядка. Проте, такі клієнти досі є складовою сучасних операційних систем сімейства Windows, UNIX-подібних та операційних систем на основі Linux. Відтоді було створено численні версії FTP клієнтів, підтримка протоколу була вбудована в різноманітні утиліти, сервери, пристрої, тощо.

FTP дає можливість абоненту обмінюватися двійковими і текстовими файлами з будь-яким комп'ютером мережі, що підтримує протокол FTP. Установивши зв'язок з віддаленим комп'ютером, користувач може скопіювати файл з віддаленого комп'ютера на свій, або скопіювати файл зі свого комп'ютера на віддалений [18].

FTP використовується для обміну даними та передачі файлів між комп’ютерами в мережі TCP/IP (протокол керування передачею/протокол Інтернету), або ж Інтернет. Користувачі, яким надано доступ, можуть отримувати та передавати файли на сервері протоколу передачі файлів (також відомому як хост/сайт FTP).

FTP-з’єднання потребує двох сторін для встановлення та обміну даними в мережі. Для цього користувачі повинні мати дозвіл, надавши облікові дані FTP-серверу. Деякі загальнодоступні сервери FTP можуть не вимагати облікових даних для доступу до своїх файлів. Ця практика поширена в так званому анонімному FTP.

Під час встановлення з’єднання FTP існує два різних канали зв’язку. Перший канал називається командним каналом, де він ініціює інструкцію та відповідь. Інший називається каналом даних, де відбувається розподіл даних.

Щоб отримати або передати файл, авторизований користувач використовуватиме протокол для запиту на створення змін на сервері. Натомість сервер надасть цей доступ. Цей сеанс називається режимом активного підключення.

Розповсюдження в активному режимі може зіткнутися з проблемою, якщо брандмауер захищає машину користувача. Брандмауер зазвичай не допускає будь-яких неавторизованих сеансів від зовнішньої сторони.

У разі виникнення такої проблеми використовується пасивний режим. У цьому режимі користувач встановлює як канал команди, так і канал даних. Тоді цей режим просить сервер слухати , а не намагатися створити з’єднання з користувачем [19].

Існує три способи встановлення з’єднання FTP. Дуже простим методом є використання FTP командного рядка, наприклад використання командного рядка для Windows або терміналу в Mac/Linux. Розробники все ще використовують його для передачі файлів за допомогою FTP.

Користувач також може використовувати веб-браузер для зв'язку з FTP-сервером. Веб-браузер зручніший, коли користувачі хочуть отримати доступ до

великих каталогів на сервері. Однак це часто менш надійно та повільніше, ніж використання спеціальної програми FTP.

Сьогодні найпоширенішою практикою використання FTP, особливо для веб-розробників, є використання FTP-клієнта.

FTP-клієнт надає більше свободи, ніж командний рядок і веб-браузер. Це також легше в управлінні та потужніший порівняно з іншими методами.

Під час використання такого клієнта також доступні додаткові функції. Наприклад, він дозволяє користувачам передавати великий файл і використовувати утиліту синхронізації [19].

2.1.7. Linux Utilities

Робота кожного розробника полягає в постійному виникненні проблем і пошуку їх вирішення. Швидкість вирішення проблеми залежить від знань, умінь і навичок роботи в розвивальному середовищі.

У повсякденній роботі використовується багато різних інструментів: від пошуку звичайного файлу до створення файлової системи. Далі буде описано обов'язкові інструменти для роботи з Linux.

Основні утиліти для роботи з файловою системою Linux:

Перемістити (перейменувати) файл:

mv /path/to/source /path/to/destination

Копіювати файл до каталогу:

cp /path/to/source /path/to/destination_directory

Видалити певний файл:

rm /path/to/file

Змінити власника файлу:

chown user:group /path/to/file

Більш просунутий набір включає утиліти mount і dd .

За допомогою монтування ви можете «приєднати» файлову систему на блочному пристрої до кореневої файлової системи.

Утиліта dd (як її ще називають «знищувач дисків») забезпечує копіювання по блоках. Використовувати його потрібно вкрай обережно, оскільки необдуманий запуск може назавжди знищити дані на пристрої.

Є багато способів проглянути на файл з різних кутів. За допомогою head і tail ми можемо прочитати початок і кінець файлу відповідно.

Необхідність пошуку інформації у файловій системі виникає досить часто (пошук конфігураційних файлів, пошук файлу, який згенерував повідомлення про помилку в журналах тощо), і таких утиліт, як find і grep:

find -name 'file_name'

grep 'what_find' file_name

Досить часто необхідно керувати процесами або просто отримати інформацію про весь або конкретний процес. Показати список усіх процесів:

ps aux

Зупинити процес:

kill -9 'process_number'

Щоб запустити процес у фоновому режимі, достатньо додати амперсанд (&) наприкінці команди, але ця опція має недолік: якщо завершиться сеанс оболонки – усі його фонові процеси також зупиняться [20].

Повернути процес у фоновий режим: *CTRL+Z*

2.2. Середовище для розробки системи

2.2.1. Visual Studio Code

Visual Studio Code (VS Code) – засіб для створення, редагування та впровадження сучасних застосунків і програм для хмарних систем. Visual Studio Code розповсюджується безкоштовно і доступний у версіях для платформ Windows, Linux і OS X.

Компанія Microsoft представила Visual Studio Code у квітні 2015 на конференції Build 2015. Це середовище розробки стало першим кросплатформовим продуктом у лінійці Visual Studio.

За основу для Visual Studio Code використовуються напрацювання вільного проєкту Atom, що розвивається компанією GitHub. Зокрема, Visual Studio Code є надбудовою над Atom Shell, що використовує браузерний рушій Chromium і Node.js. Примітно, що про використання напрацювань вільного проєкту Atom і на сайті Visual Studio Code, і в пресрелізі, і в офіційному блозі не згадується.

Редактор містить вбудований компілятор, інструменти для роботи з Git і засоби рефакторингу, навігації по коду, автодоповнення типових конструкцій і контекстної підказки. Продукт підтримує розробку для платформ ASP.NET і Node.js, і позиціонується як легковагове рішення, що дозволяє обійтися без повного інтегрованого середовища розробки. Серед підтримуваних мов і технологій: JavaScript, C++, C#, TypeScript, jade, PHP, Python, XML, Batch, F#, DockerFile, Coffee Script, Java, HandleBars, R, Objective-C, PowerShell, Luna, Visual Basic, Markdown, JSON, HTML, CSS, LESS і SASS, Haxe [21].

Visual Studio Code поєднує в собі простоту редактора вихідного коду з потужними інструментами розробника, як-от завершення коду та налагодження IntelliSense. Надзвичайно простий цикл редагування-збирання-налагодження означає менше часу на возитися з вашим середовищем і більше часу на реалізацію ваших ідей.

Visual Studio Code підтримує macOS, Linux і Windows, тож ви можете почати працювати незалежно від платформи.

У своїй основі Visual Studio Code має блискавичний редактор вихідного коду, ідеальний для повсякденного використання. Завдяки підтримці сотень мов VS Code допомагає вам миттєво працювати продуктивно за допомогою підсвічування синтаксису, зіставлення дужок, автоматичного відступу, вибору прямокутника, фрагментів тощо. Інтуїтивно зрозумілі комбінації клавіш, легке налаштування та зіставлення комбінацій клавіш, надані спільнотою, дозволяють легко орієнтуватися в коді.

І коли кодування стає важким, складне отримує налагодження. Налагодження часто є тією функцією, якої розробники найбільше не вистачають у простішому кодуванні, тому ми зробили це. Visual Studio Code містить інтерактивний налагоджувач, тож ви можете покроково переглядати вихідний код, перевіряти змінні, переглядати стеки викликів і виконувати команди в консолі [22].

VS Code також інтегрується з інструментами збірки та сценаріїв для виконання звичайних завдань, що пришвидшує щоденні робочі процеси. VS Code підтримує Git, тому ви можете працювати з керуванням джерелом, не виходячи з редактора, включаючи перегляд змін, що очікують на розгляд.

VS Code включає розширену вбудовану підтримку розробки Node.js за допомогою JavaScript і TypeScript, що базується на тих самих базових технологіях, які керують Visual Studio. VS Code також містить чудові інструменти для веб-технологій, таких як JSX/React, HTML, CSS, SCSS, Less і JSON.

Архітектурно Visual Studio Code поєднує в собі найкраще з веб-технологій, нативних і мовних технологій. Використовуючи Electron , VS Code поєднує такі веб-технології, як JavaScript і Node.js, зі швидкістю та гнучкістю нативних програм. VS Code використовує новішу, швидшу версію того самого промислового редактора на основі HTML, який використовувався в хмарному редакторі "Monaco", інструментах Internet Explorer F12 та інших проектах.

Visual Studio Code містить загальнодоступну модель розширення, яка дозволяє розробникам створювати та використовувати розширення, а також широко налаштовувати свій досвід редагування-складання-налагодження [22].

Розглянемо найпопулярніші середовища розробки (рис.2.2):

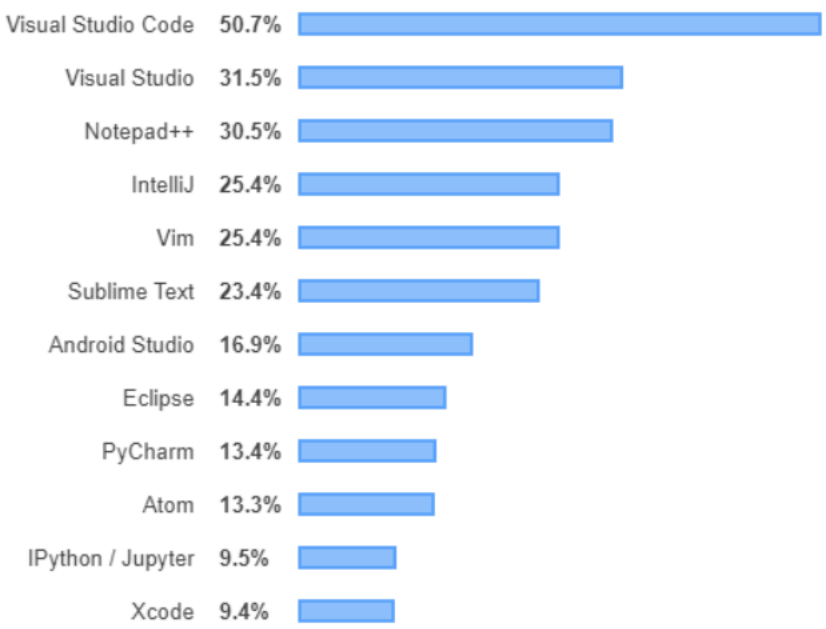

Рис.2.2. Найпопулярніші середовища розробки

Його популярність пояснюється потребою розробників мати легкий добре зроблений редактор з невеликою кількістю функцій, але менш складним, ніж інші, доступні на ринку.

Він також безкоштовний, і його розробляє та підтримує корпорація Майкрософт із сучасним підходом із використанням Electron.

Це деякі характеристики, які виводять VS Code на перше місце серед редакторів кодування [23].

2.2.2. MobaXterm

MobaXterm – це рішення для віддаленого робочого столу, яке надає підприємствам такі мережеві інструменти, як VNC, RDP або FTP, а також різні команди Unix для керування завданнями на віддаленому робочому столі Windows. Функції включають переадресацію X11, спільний доступ до сеансу, керування паролями, моніторинг мережі та передачу файлів.

MobaXterm постачається з вбудованими графічними інструментами, які дозволяють розробникам перетягувати файли безпосередньо з або на віддалені сервери за допомогою безпечного з'єднання SSH. Інтерфейс програми з декількома вкладками дозволяє мережевим інженерам запускати різні незалежні сеанси, запускати світлові демони та відображати термінали в горизонтальному або вертикальному режимах розділення.

Використовуючи інструмент тунелювання, IT-адміністратори можуть створювати тунелі SSH для перенаправлення трафіку з одного порту на інший. MobaXterm містить модуль менеджера сеансів, що дозволяє користувачам створювати та налаштовувати віддалені сесії та зберігати їх для подальшого використання.

MobaXterm – це ваш найкращий інструментарій для віддаленого обчислення. В одній програмі Windows він надає безліч функцій, призначених для програмістів, веб-майстрів, IT-адміністраторів і майже всіх користувачів, яким потрібно простіше виконувати свої віддалені завдання.

MobaXterm надає всі важливі інструменти віддаленої мережі (SSH, X11, RDP, VNC, FTP, MOSH, ...) і команди Unix (bash, ls, cat, sed, grep, awk, rsync, ...) для робочого столу Windows, в одному портативному файлі exe, який працює з коробки. Додаткова інформація про підтримувані мережеві протоколи.

Є багато переваг наявності універсальної мережевої програми для ваших віддалених завдань, наприклад, коли ви використовуєте SSH для підключення до віддаленого сервера, графічний браузер SFTP автоматично з'явиться для безпосереднього редагування ваших віддалених файлів. Ваші віддалені програми також безперешкодно відображатимуться на робочому столі Windows за допомогою вбудованого X-сервера [24].

2.3. Висновки до розділу 2

Для розробки технології оновлення IoT систем на базі Embedded Linux було використано декілька технологій, які виконують різні функції на серверній та апаратній частинах.

Для розробки функціональної складової системи було обрано об'єктно-орієнтовані мови програмування C++ та Python. C++ та Python дозволяє розробникам створювати безліч безпечних, надійних та швидкодіючих програм.

При розробці технологій оновлення IoT систем на базі Embedded Linux були також використані такі скриптові мови, як Bash-сценарії та Makefile. Дані мови є найбільш швидкодіючими та мають безпосередню підтримку для Embedded Linux.

Для підключення до плати було використано MobaXterm – програма для підключення до плати через такі мережеві інструменти, як VNC, RDP або FTP, а також різні команди Unix для керування платою.

Безпосередньо для створення компонентів даної системи було обрано інтегроване середовище розробки Visual Studio Code – засіб для створення, редагування та впровадження сучасних застосунків і програм. У своїй основі Visual Studio Code має дуже добре налаштований редактор вихідного коду, ідеальний для повсякденного використання. Завдяки підтримці сотень мов VS Code допомагає вам миттєво та продуктивно працювати за допомогою підсвічування синтаксису, зіставлення дужок, автоматичного відступу, вибору прямокутника, фрагментів тощо. Інтуїтивно зрозумілі комбінації клавіш, легке налаштування та зіставлення комбінацій клавіш, надані спільнотою, дозволяють легко орієнтуватися в коді.

РОЗДІЛ 3. РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТЕХНОЛОГІЇ ОНОВЛЕННЯ IOT СИСТЕМ НА БАЗІ EMBEDDED LINUX

Файлова структура проекту, яка зображена на рисунку 3.1, містить основні файли, які відповідають за основну функціональність системи та її працездатність. Процес оновлення можна розділити фактично на 4 послідовні дії (рис. 3.2):

- створення патчу оновлення за допомогою bash-сценаріїв;
- пересилання патчу на одноплатний комп'ютер Raspberry Pi використовуючи WinSCP;
- застосування патчу використовуючи C++ та Python;
- перевірка виконання оновлення за допомогою bash-сценаріїв.

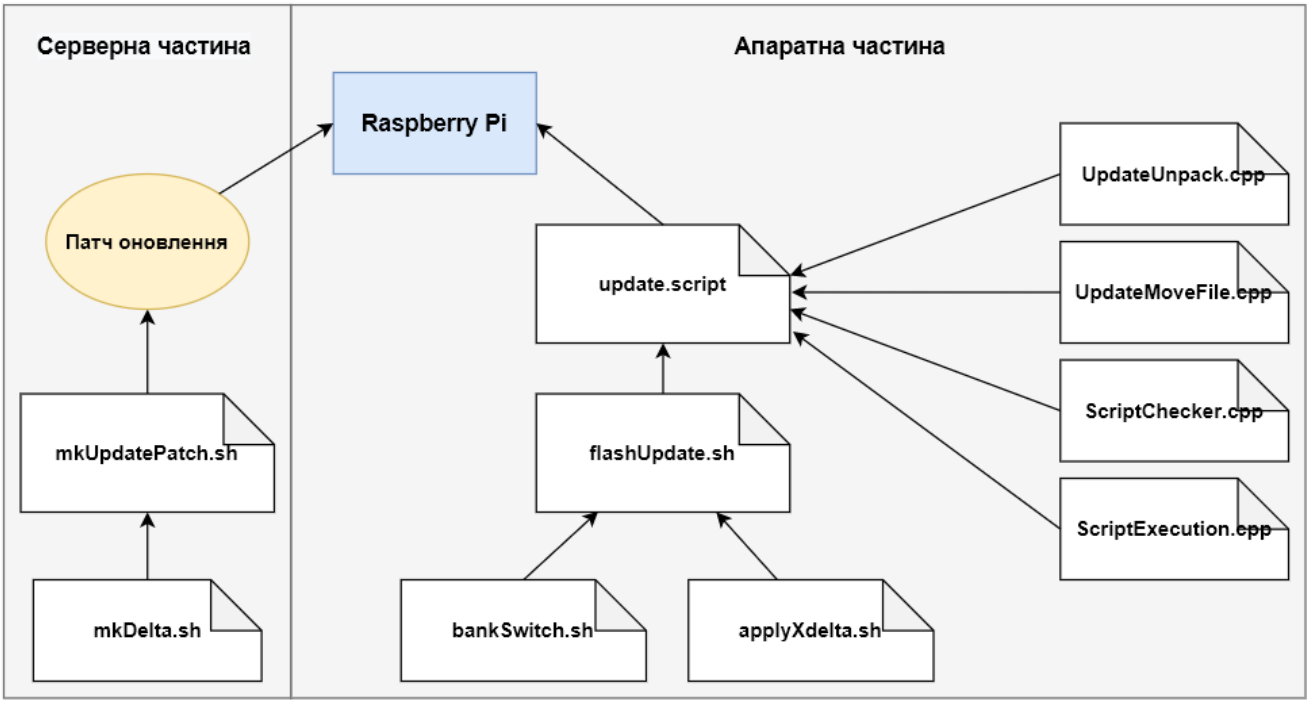

Рис. 3.1. Файлова структура системи

Рис. 3.2. Етапи розробки системи

3.1. Розробка функціональної складової системи

Технологія оновлення для IoT систем на базі Embedded Linux передбачає розділення скриптів на 2 групи:

- перша група скриптів буде виконуватись на серверній частині;
- друга група скриптів буде виконуватись на апаратній.

Процес оновлення буде розглянуто на прикладі одноплатного комп'ютера Raspberry Pi.

3.1.1. Розробка системи для серверної частини

Процес оновлення для серверної частини починається зі створення патчу оновлення, який містить тільки різницю між версіями прошивки плати (Raspberry Pi). Патч оновлення створюється за допомогою скрипта mkUpdatePatch.sh, який повинен містити 3 вхідні параметри (рис.3.3):

- версію, з якої оновлюється система;
- версію, на яку оновлюється;
- назва патчу.

```
usage()
{

    usage="$(basename "$0") [-h] [-p -c -o]
                                    \n
    where:                                                                  \n
        -h  show this help text with available options                      \n
        -p  previous version from which the patch is created                \n
        -c  the current version that will be after update                   \n
        -o  Specify the name of the generated patch                         \n
    "

    echo -e "$usage"
    exit
}
```

Рис.3.3. Вхідні параметри mkUpdatePatch.sh

3.1.1.1. Створення скрипта mkUpdatePatch.sh

Для початку розглянемо вхідні параметри для створення патчу, які виконуються за допомогою циклу *while getopts*.

Перший вхідний параметр -p, який відповідає за попередню версію системи. Другий вхідний параметр -c, який відповідає за поточну версію системи. Третій параметр -o відповідає за назву вихідного патчу (рис.3.4).

```
while getopts "p:c:o:" opt; do
    case $opt in
    p)
        PREVIOUS_VERSIONS=$OPTARG
        echo "PREVIOUS_VERSIONS: $PREVIOUS_VERSIONS"
        ;;
    c)
        CURRENT_VERSION=$OPTARG
        echo "CURRENT_VERSIONS: $CURRENT_VERSIONS"
        ;;
    o)
        OUTPUT_PATCH_NAME=$OPTARG
        ;;
    :)
        echo "Option -$OPTARG requires an argument." >&2
        usage
        ;;
    esac
done
```

Рис.3.4. Реалізація вхідних параметрів mkUpdatePatch.sh

Після того як вхідні параметри були отримані, починається створення патчу за допомогою методу *createPatch()* (рис.3.5). Даний метод створює папки, в яких будуть міститися xdelta-файли та файли з md5sum різних типів партішинів (дисків), в даній системи їх всього 3: bootfs, dmtable та rootfs. Безпосередньо xdelta різницю створює скрипт mkDelta.sh. Після того як xdelta різниця створена, формується вихідний патч (.patch) з назвою переданою параметром -o.

xdelta – це вільна програма, що працює з командного рядка для отримання різниці між двома файлами. Її призначення таке ж, як у програм diff та patch, але xdelta працює і з двійковими (не текстовими) файлами. Також, на відміну від diff, результат порівняння файлів не зручний для прочитання людиною [25].

```
createPatch()
{
    echo "Generating patch..."
    GENERATED_PATCH_NAME=$OUTPUT_PATCH_NAME

    echo "Started deltas generation"

    rm -rf "$PREVIOUS_VERSIONS/deltas"   # In case it exists
    mkdir -p "$PREVIOUS_VERSIONS/deltas/"

    DELTA_OUTPUT_DIR="$PREVIOUS_VERSIONS/deltas/"

    rm -rf "$CURRENT_VERSIONS/deltas"   # In case it exists
    mkdir -p "$CURRENT_VERSIONS/deltas/"

    mkDelta.sh -p "${PREVIOUS_VERSIONS}/bootfs.img" -c "$CURRENT_VERSIONS/bootfs.img" -o "$DELTA_OUTPUT_DIR/bootfs" -t "bootfs"
    mkDelta.sh -p "${PREVIOUS_VERSIONS}/dmtable.img" -c "$CURRENT_VERSIONS/dmtable.img" -o "$DELTA_OUTPUT_DIR/dmtable" -t "dmtable"
    mkDelta.sh -p "${PREVIOUS_VERSIONS}/rootfs.img" -c "$CURRENT_VERSIONS/rootfs.img" -o  "$DELTA_OUTPUT_DIR/rootfs" -t "rootfs"

    IMAGE_BASED_DIFF_DIR="$DELTA_OUTPUT_DIR"

    cd "$PATCH_DIR"

    tar -cvf "$GENERATED_PATCH_NAME" "$IMAGE_BASED_DIFF_DIR"
}
```

Рис.3.5. Реалізація методу createPatch() в mkUpdatePatch.sh

md5sum – програма, що дозволяє обчислювати значення хеш-сум (контрольних сум) файлів за алгоритмом MD5. У звичайному випадку обчислені хеші виводяться (можна зберегти у файлі для подальшого використання). В інших випадках програма звіряє обчислені значення зі значеннями, збереженими у файлі (це зручно для масової перевірки цілісності файлів). Найчастіше програма використовується для перевірки правильного завантаження файлів через мережу. Програма має безліч версій для різних ОС - наприклад, Linux, UNIX, Microsoft Windows, MacOS [26].

3.1.1.2. Створення скрипта mkDelta.sh

mkDelta.sh має 4 вхідні параметри (рис.3.6):

- -p файли попередньої версії;
- -c файли нової версії;
- -o назва вихідної директорії;
- -t тип партішину.

```
while getopts "p:c:o:t:h" opt; do
    case $opt in
    p)
        if [[ $OPTARG == *.bin ]] || [[ $OPTARG == *.img ]]
        then
            PREVIOUS_PATH=$OPTARG
        fi
        ;;
    c)
        if [[ $OPTARG == *.bin ]] || [[ $OPTARG == *.img ]]
        then
            CURRENT_PATH=$OPTARG
        fi
        ;;
    o)
        OUTPUT_DIRECTORY=$OPTARG
        mkdir -p $OUTPUT_DIRECTORY
        ;;
    t)
        TITLE_BASE=$OPTARG
        ;;
    :)
      echo "Option -$OPTARG requires an argument." >&2
      usage
      ;;

    esac
done
```

Рис.3.6. Реалізація вхідних параметрів mkDelta.sh

Після того як вхідні параметри були отримані, починається генерація .xdelta та .info файлів за допомогою методу *generate_xdelta_files()* (рис.3.7). На виході даного методу ми отримуємо 6 файлів.

```
generate_xdelta_files()
{
    DELTA_VAR=xdelta3

    PATCH_NAME="${TITLE_BASE}.xdelta3"
    INFO_NAME="${TITLE_BASE}.info"

    $DELTA_VAR -D0Nq -s "${PREVIOUS_PATH}" "${CURRENT_PATH}"  "$OUTPUT_DIRECTORY/$PATCH_NAME"

    MD5_SUM=$(md5sum -b "${CURRENT_PATH}")
    echo "$MD5_SUM" > "$OUTPUT_DIRECTORY/$INFO_NAME"
}
```

Рис.3.7. Реалізація методу generate_xdelta_files() в mkDelta.sh

3.1.2. Процес пересилання патчу на плату

Процес пересилання патчу на плату реалізується за допомогою WinSCP, який виконує безпечна передача файлів між локальним комп'ютером і платою за допомогою протоколів SSH та FTP (рис.3.8).

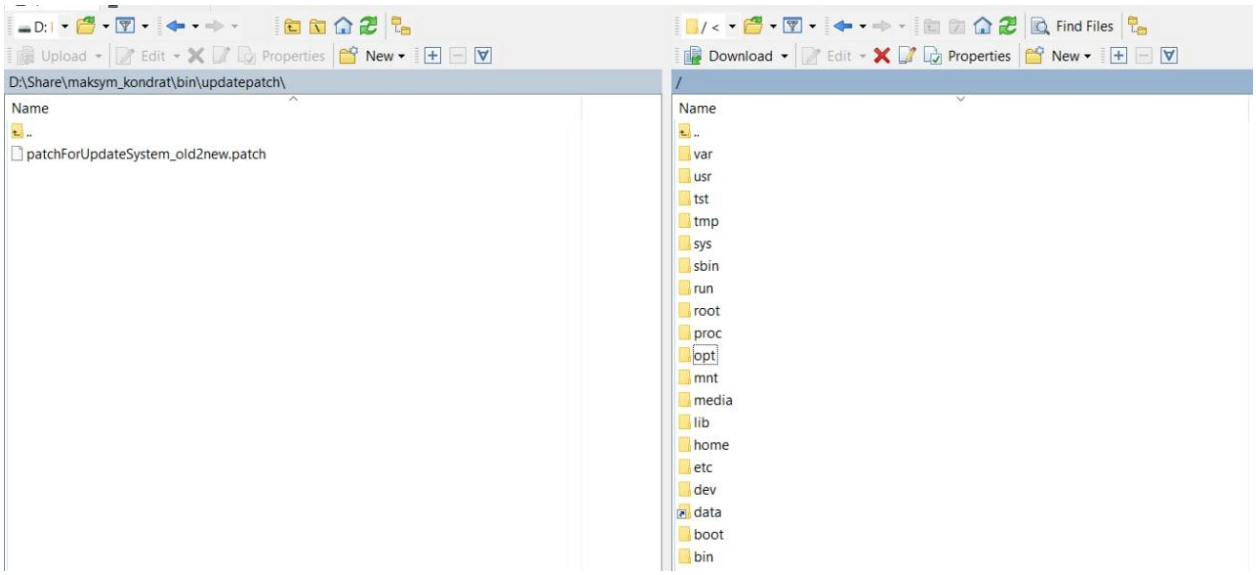

Рис.3.8. Інтерфейс WinSCP

WinSCP (Windows Secure Copy) – це безкоштовний та відкритий клієнт протоколу передачі файлів SSH (SFTP), протоколу передачі файлів (FTP), WebDAV, Amazon S3 і протоколу безпечного копіювання (SCP) для Microsoft Windows. Його

основною функцією є безпечна передача файлів між локальним комп’ютером і віддаленим сервером. Окрім цього, WinSCP пропонує базовий файловий менеджер і функції синхронізації файлів. Для безпечної передачі він використовує протокол Secure Shell (SSH) і підтримує протокол SCP на додаток до SFTP. WinSCP базується на реалізації протоколу SSH від PuTTY і протоколу FTP від FileZilla [27].

Підключення з локального комп’ютера до одноплатного комп'ютера Raspberry Pi відбувається за допомогою PuTTY, який використовує протокол передачі файлів SSH (рис.3.9).

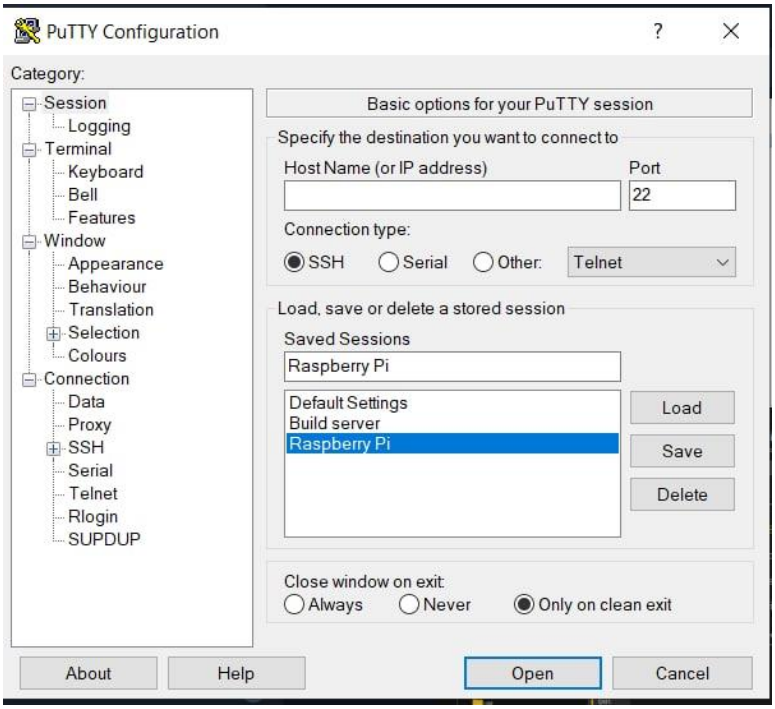

Рис.3.9. Інтерфейс PuTTY

Робота з Raspberry Pi відбувається за допомогою інтегрованого терміналу MobaXTerm для Windows, який дозволяє мати додаткові можливості для Unix-подібних систем (рис.3.10).

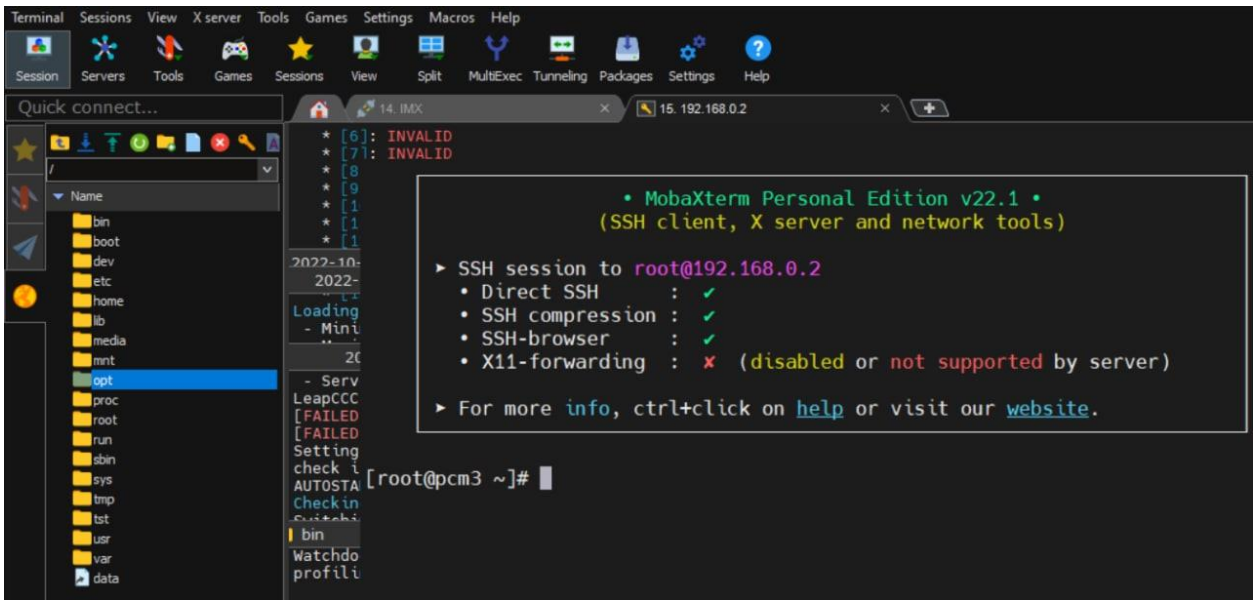

Рис.3.10. Інтерфейс MobaXTerm

PuTTY – вільно розповсюджуваний клієнт для протоколів SSH, Telnet, rlogin і чистого TCP. Спочатку розроблявся для Windows, проте пізніше був портований на різні операційні системи. Наявні офіційні порти для UNIX`оподібних платформ. Порти для класичної Mac OS й для Mac OS X знаходяться в стадії розробки. Існують неофіційні порти на інші платформи, зокрема для мобільних телефонів під управлінням Symbian OS та Windows Mobile. Програма випускається під ліцензією MIT License [28].

3.1.3. Розробка системи для апаратної частини

Патч оновлення після того як потрапляє на плату через WinSCP починає застосовуватись через *update.script*. *update.script* вмістить в собі 4 файли, написаних на C++, які виконуються по черзі:

- UpdateUnpack.cpp – розпаковує патч оновлення;
- UpdateMoveFile.cpp – переміщає файли оновлення у відповідні папки;
- ScriptChecker.cpp – перевіряє правильність знаходження файлів оновлення та їх md5sum;

- ScriptExecution.cpp – містить методи для перевірки md5sum, які використовуються в ScriptChecker.cpp.

Після того як *update.script* завершив розпаковку та перевірку архіву, викликає flashUpdate.sh скрипт, який починає застосовувати оновлення до системи. В свою чергу flashUpdate.sh викликає bankSwitch.sh та applyXdelta.sh.

3.1.3.1. Розробка файлу UpdateUnpack.cpp

Файл UpdateUnpack.cpp має головний метод *unpackPatch()* (рис.3.11), який виконує розархівування патчу. Даний метод має 2 вхідні параметри: updatePatch (патч оновлення) та destinationDir (шлях, в який буде виконуватись розархівування). Тип значення, що повертається з цього методу – void.

В методі виконується перевірка на можливість відкриття патчу оновлення за допомогою методу is_open(). Метод is_open() перевіряє можливість відкриття та одночасно відкриває патч.

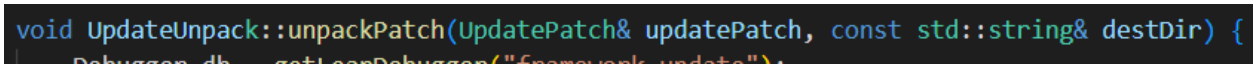

Рис.3.11. Метод unpackPatch() в UpdateUnpack.cpp

Якщо перевірка пройшла успішно (рис.3.12), то закриваємо патч за допомогою методу close(). Далі виконується метод unzipFiles(), який має вхідний параметр destinationDir та виконує розархівування файлів за заданим шляхом.

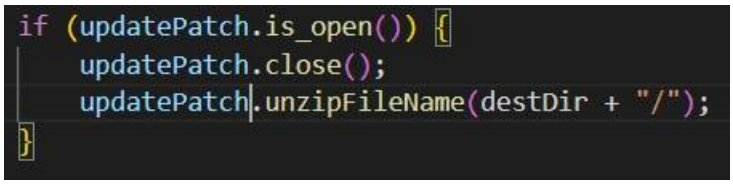

Рис.3.12. Успішне відкриття патчу оновлення в UpdateUnpack.cpp

Якщо перевірка не пройшла успішно та не вдалося відкрити архів (рис.3.13), то виводимо помилку в консоль та виходимо з даного методу з помилкою.

```
else if ((!updatePatch.is_open()) {
    logError(db, "Update patch is not found.");
    THROW(UpdateException, ("Update patch is not found.", INVALID_PATCH));
}
```

Рис.3.13. Невдале відкриття патчу оновлення в UpdateUnpack.cpp

3.1.3.2. Розробка файлу UpdateMoveFile.cpp

Файл UpdateMoveFile.cpp має головний метод *copyFileFromZip()* (рис.3.14), який виконує переміщення файлів з загальної у відповідні папки для кожного файлу. Даний метод має 2 вхідні параметри: fileName (старе ім'я файлу) та destFile (нове ім'я файлу). Тип значення, що повертається з цього методу – void.

```
void UpdateMoveFile::copyFileFromZip(const std::string& fileName, const std::string& destFile) {
```

Рис.3.14. Метод copyFileFromZip() в UpdateMoveFile.cpp

Метод copyFileFromZip() спочатку перевіряється можливість створення файлу з параметром destFile за допомогою методу fopen() (рис.3.15). Якщо створити не вдалося, то програма повідомляє про помилку користувачу.

```
outFile = fopen(destFile.c_str(), "w");
if (outFile == NULL) {
    logError(db, "Unable create output file: %s", destFile.c_str());
    THROW(UpdateException, ("Unable create output file: " + destFile));
}
```

Рис.3.15. Створення файлу в UpdateUnpack.cpp

Метод copyFileFromZip() виконує переміщення кожного файлу через пайп (pipe). Присвоєння кожного файлу в пайп відбувається за допомогою методу getFile() (рис.3.15).

Далі створюється тимчасовий файл методом fread() такого ж розміру, як і файл з патчу оновлення, та виконується перевірка, що він не пустий (рис.3.16). Створюється ще одним тимчасовий файл, в який записуется вміст файлу за допомогою методу fwrite(), і виконується перевірка на збіжність файлів.

```
pipe = getFile(fileName);
while (!feof(pipe)) {
    hasReadFile = fread(pipe);
    if (hasReadFile > 0) {
        writtenFile = fwrite(hasReadFile, outFile);
        if (writtenFile != outFile) {
            logError(db, "Unable to write to output file: %s", destFile.c_str());
            break;
        }
    }
}
```

Рис.3.16. Переміщення файлу в UpdateUnpack.cpp

Pipe – це зв’язок між двома процесами, таким чином стандартний вихід одного процесу стає стандартним входом іншого процесу. В операційній системі UNIX канали корисні для зв’язку між пов’язаними процесами (міжпроцесовий зв’язок).

Pipe є лише одностороннім зв’язком, тобто ми можемо використовувати канал таким чином, що один процес записує в канал, а інший процес читає з каналу. Він відкриває канал, який є областю основної пам’яті, яка розглядається як «віртуальний файл» .

Pipe може використовуватися процесом створення, а також усіма його дочірніми процесами для читання та запису. Один процес може писати в цей «віртуальний файл» або канал, а інший пов’язаний процес може читати з нього [29].

3.1.3.3. Розробка файлу ScriptChecker.cpp

Файл ScriptChecker.cpp має головний метод *checkScript()* (рис.3.17), який перевіряє правильну локацію скопійованих файлів. Даний метод має вхідний параметр: fileName (ім'я файлу). Тип значення, що повертається з цього методу – bool.

```
bool ScriptChecker::checkScript(const std::string& fileName) {
```

Рис.3.17. Метод checkScript() в ScriptChecker.cpp

Метод checkScript() спочатку перевіряється ім'я файлу з параметром fileName за допомогою методу parse() (рис.3.18). Якщо файл не знайдений, то метод повертає значення false, і повідомляє користувачу про помилку.

```
if (checkManager.parse(fileName) != 0) {···
} else {
    logError(db, "Unsuccessfully checked files in checkScript!");
    return false;
}
```

Рис.3.18. Перевірка файлу в ScriptChecker.cpp

Метод checkScript() робить 2 перевірки: перевірку, що ім'я файлу правильно, та перевірку, що md5sum збігається. Для перевірки ім'я, створюється тимчасова зміна file_name, якій присвоюється назва поточного файлу методом getName(). Далі перевіряється збіг значень тимчасової змінної та параметру fileName (рис.3.19).

```
filePtr file_name = checkManager.getName();

if (file_name != fileName)
    return false;
```

Рис.3.19. Перевірка назви файлу в ScriptChecker.cpp

Для перевірки md5sum файлу, також створюється тимчасова зміна prevMd5sum та має значення за допомогою методу loadPrevMd5sum(), яке дорівнює md5sum попереднього файлу. Далі виконуємо 2 перевірки, що md5sum поточного файлу не дорівнює 0, та, що md5sum поточного файлу дорівнює prevMd5sum. Дана перевірка виконується за допомогою методу getMd5sum() для змінної file_name (рис.3.20).

```
std::string prevMd5sum = scriptExecution->loadPrevMd5sum();

if (file_name.getMd5sum() != 0) {
    if (file_name.getMd5sum() == prevMd5sum) {
        return true;
    }
    else
        return false;
} else {
    return false;
}
```

Рис.3.20. Перевірка md5sum файлу в ScriptChecker.cpp

3.1.3.4. Розробка файлу ScriptExecution.cpp

Файл ScriptExecution.cpp має головний метод *loadPrevMd5sum()* (рис.3.21), який знаходить md5sum файлу, яка була до оновлення (вона має збігатися з теперішньою). Даний метод не має вхідних параметрів. Тип значення, що повертається з цього методу – void.

```
void ScriptExecution::loadPrevMd5sum() {
```

Рис.3.21. Метод *loadPrevMd5sum*() в ScriptExecution.cpp

Метод loadPrevMd5sum() проходить по партішину та підсумовує всі значення md5sum кожного файлу (рис.3.22).

```
if (!s.empty()) {
    stringstream ss(s);
    string md5Sum;

    for (u32 i = 0; i < 2; ++i) {
        vector<string>& v = prevMd5Sums[i];
        v.clear();
        vector<string>::size_type size;
        ss >> size;

        for (vector<string>::size_type j = 0; j < size; ++j) {
            ss >> md5Sum;
            v.push_back(md5Sum);
        }
    }
}
```

Рис.3.22. Реалізація методу *loadPrevMd5sum*() в ScriptExecution.cpp

3.1.3.5. Розробка скрипта flashUpdate.sh

Скрипт flashUpdate.sh не має ніяких вхідних параметрів, та просто викликає 3 рази applyXdelta.sh скрипт з різними параметрами. Якщо з якихось причин не вдалося викликати applyXdelta.sh хоча б один раз з трьох, то flashUpdate.sh завершується з відповідною помилкою. Якщо ніяких помилок не з'явилось, то скрипт завершується зі значенням 0, після чого система продовжує виконувати процес оновлення (рис.3.23).

```
# patch the bootfs
/opt/flash/applyXdelta.sh "$(/opt/flash/bankSwitch.sh info)" "bootfs"
if [ $? != 0 ]; then
  exit 2
fi

# patch the dmtable
/opt/flash/applyXdelta.sh "$(/opt/flash/bankSwitch.sh info)" "dmtable"
if [ $? != 0 ]; then
  exit 3
fi

# patch the rootfs
/opt/flash/applyXdelta.sh "$(/opt/flash/bankSwitch.sh info)" "rootfs"
if [ $? != 0 ]; then
  exit 4
fi

exit 0
```

Рис.3.23. Реалізація скрипта flashUpdate.sh

3.1.3.6. Розробка скрипта bankSwitch.sh

Скрипт bankSwitch.sh має один вхідний параметр (ис.3.24), який може дорівнювати лише: info, 0, 1, futstate. Якщо bankSwitch.sh отримує інше значення від цих 4 значень, то викликається метод *help()* (рис.3.25), який повідомляє, що вхідне значення некоректне.

```
case "$1" in
    "info")
        echo $STATE
        exit
    ;;
    "0")
        STATE="0"
    ;;
    "1")
        STATE="1"
    ;;
    "futstate")
        STATE=$FUTURESTATE
    ;;
    *)
        help
esac
```

Рис.3.24. Реалізація вхідного параметра bankSwitch.sh

```
help()
{
    echo "usage: $(basename "$0") [ info | 0 | 1 | futstate ]";
}
```

Рис.3.25. Реалізація методу help() в bankSwitch.sh

Скрипт bankSwitch.sh використовуєтесь для того щоб дізнатися на якому з двох сховищ(розділів) встановлена система: першому або другому. Це потрібно для того щоб знати на якому зі сховищ встановлюється оновлення, а на якому встановлена стара система. В разі невдалого оновлення, або помилок під час оновлення, система автоматично переходить на сховище, де встановлена стара система. Тобто скрипт bankSwitch.sh є критично важливим, так як він містить значення сховища, на яке

застосовується оновлення системи. В разі вдалого оновлення системи, одноплатний комп'ютер Raspberry Pi автоматично виконує перезавантаження, та вже завантажується з нового сховища.

Скрипт bankSwitch.sh є фактично обгорткою файлу bank_selection (рис.3.26), який містить в собі одну цифру, тобто номер сховища з якого зараз завантажена система.

```
BANK_VALUE_PATH="/proc/board/bank_selection"

SWITCH=$( cat "${BANK_VALUE_PATH}" )

if [ "${SWITCH}" = "0"  ];then
    STATE="0"
    FUTURESTATE="1"
elif [ "${SWITCH}" = "1" ];then
    STATE="1"
    FUTURESTATE="0"
else
    echo "Incorect bankswitch partition"
fi
```

Рис.3.26. Реалізація скрипта bankSwitch.sh

3.1.3.7. Розробка скрипта applyXdelta.sh

Скрипт applyXdelta.sh має 2 вхідні параметри (рис.3.27):

- номер сховища – 0 або 1;
- назва партішина (диска): bootfs, dmtable або rootfs.

Рис.3.27. Значення PARTITION та NEWSTATE змінних

Скрипт applyXdelta.sh виконує перевірку чому дорівнює номер сховища, і залежно від цього зміні PARTITION та NEWSTATE мають різні значення (рис.3.27).

Після того як зміні PARTITION та NEWSTATE визначені виконується команда dd, яке копіює партішин оновлення в тимчасовий, так як це більш безпечно ніж копіювати напряму в системний партішин. Команда має 2 вхідні параметри: if (вхідний файл/партішин) та of (вихідний файл/партішин) (рис.3.28).

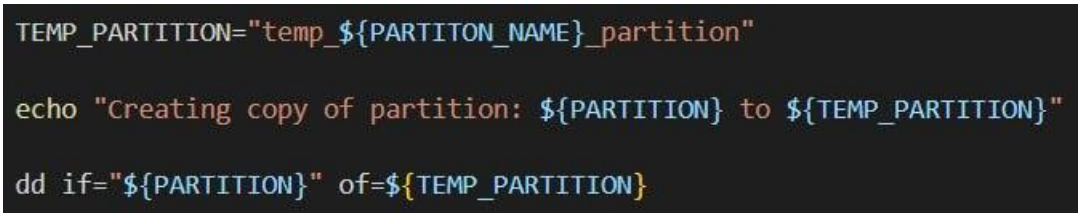

Рис.3.28. Реалізація команди dd в applyXdelta.sh

Після того як копіювання партішину відбулося, виконується метод *apply_xdelta_patch()*, який в залежності від значення NEWSTATE, визначає значення змінної NEWPARTITION (рис.3.29).

Далі виконується команда xdelta3, яка копіює з тимчасового партішину в новий. Команда xdelta3 має 3 вхідні параметри, які повинні бути в такому вигляді: “партішин з якого копіюють” < “назва партішину” > “партішин в який копіюють”. Також є перевірка на виконання команди xdelta3, яке в негативному сценарії зупинить процес оновлення, та повідомить про це користувача (рис.3.29).

```
apply_xdelta_patch()
{
    if [ "${NEWSTATE}" = "0" ]; then
        DEVICE_NAME="${PARTITON_NAME}0Device"
        NEWPARTITION=$(configuration os/partitions.config ${DEVICE_NAME})
    else
        DEVICE_NAME="${PARTITON_NAME}1Device"
        NEWPARTITION=$(configuration os/partitions.config ${DEVICE_NAME})
    fi

    xdelta3 -c -d -s "${TEMP_PARTITION}" < "${XDELTA_NAME}" > "${NEWPARTITION}"
    if [ $? != 0 ]; then
        echo "Applying partition patch failed"
        exit 1
    fi
}
```

Рис.3.29. Реалізація методу apply_xdelta_patch() в applyXdelta.sh

На цьому процес оновлення завершується, та пристрій перезавантажується на сховище з новою системою.

3.2. Результат роботи системи оновлення IoT систем

3.2.1. Результат роботи системи на серверній частині

Створення патчу оновлення відбувається таким чином:

mkUpdatePatch.sh -p /new_system/system1.0.zip -c /old_system/system2.0.zip -o patchForUpdateSystem_old2new.patch

Результат даної команди показано на рисунку 3.30:

```
mkUpdatePatch.sh -p /old_system/system1.0.zip -c /new_system/system2.0.zip -o patchForUpdateSystem_old2new.patch
Generating patch...
USE_FLASH_IMAGE_DIFF: true
Current version is: /new_system/system2.0.zip
Previous version is: /old_system/system1.0.zip
Current version option has the system2.0. Processing...
Archive:  /new_system/system2.0.zip
  inflating: current_image_location/bootfs.img
  inflating: current_image_location/dmtable.img
  inflating: current_image_location/rootfs.img
Previous version option has the system1.0. Processing...
Archive:  /old_system/system1.0.zip
  inflating: previous_image_location/bootfs.img
  inflating: previous_image_location/dmtable.img
  inflating: previous_image_location/rootfs.img
Started deltas generation
Archive:  /new_system/system2.0.zip
VERSION_DIR_DELTA_GEN: /old_system/system1.0.zip
VERSION_DIR_DELTA_GEN: /bin/updatepatch/previous_image_location
CUR_BIN_SIZE: 33554432
CUR_PAD_BYTESIZE: 0
PAD_BYTES_COUNT: 0
CUR_BIN_SIZE: 33554432
CUR_PAD_BYTESIZE: 0
PAD_BYTES_COUNT: 0
CUR_BIN_SIZE: 1048576
CUR_PAD_BYTESIZE: 0
PAD_BYTES_COUNT: 0
CUR_BIN_SIZE: 1048576
CUR_PAD_BYTESIZE: 0
PAD_BYTES_COUNT: 0
CUR_BIN_SIZE: 98566144
CUR_PAD_BYTESIZE: 425721856
PAD_BYTES_COUNT: 831488
831488+0 records in
831488+0 records out
425721856 bytes (426 MB, 406 MiB) copied, 0.922455 s, 462 MB/s
CUR_BIN_SIZE: 98566144
CUR_PAD_BYTESIZE: 425721856
PAD_BYTES_COUNT: 831488
831488+0 records in
831488+0 records out
425721856 bytes (426 MB, 406 MiB) copied, 0.93425 s, 456 MB/s
Generating update patch...
Patch generated in: /bin/updatepatch
```

Рис.3.30. Результат роботи mkUpdatePatch.sh

У вихідному патчі повинно містися 6 файлів: 3 xdelta-файли з різницею та 3 файли, які містять md5sum (рис.3.31).

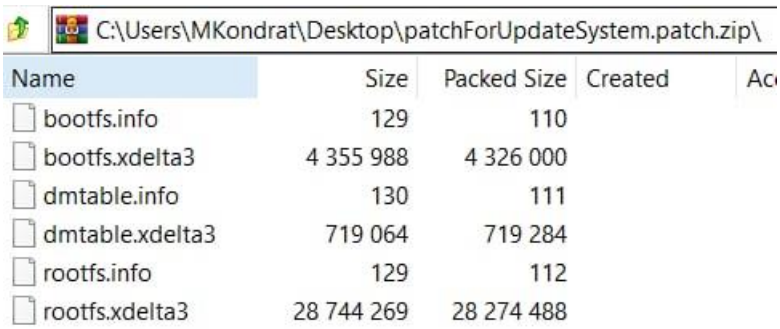

Рис.3.31. Вміст вихідного патчу оновлення

3.2.2. Результат роботи системи на апаратній частині

Перевірка роботи системи буде відбуватися таким чином:

- створення файлу для тестування test.txt (рис.3.32);

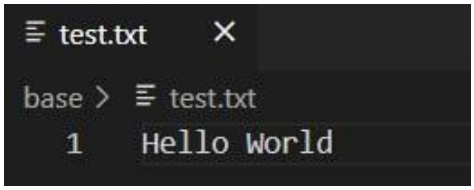

Рис.3.32. Тестовий файл test.txt

- додаємо файл test.txt до файлу Rules.mk, щоб він з’явився в системі після оновлення (рис.3.33);

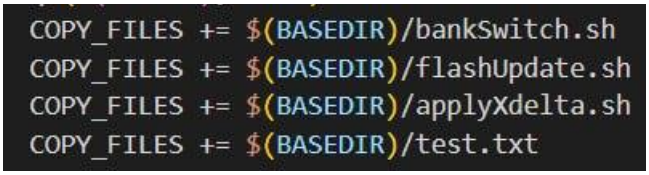

Рис.3.33. Файл Rules.mk

- створюємо патч оновлення, як це показано в попередньому розділі;
- пересилаємо патч оновлення у відповідну папку на одноплатний комп'ютер Raspberry Pi через WinSCP;
- перевіряємо на якому сховищі ми знаходимось за допомогою файлу bank_selection (рис.3.34);

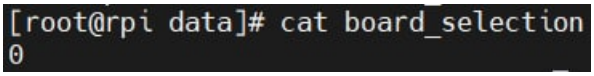

Рис.3.34. Сховище до процесу оновлення

- виводимо файли в папці в яку буде копіюватися тестовий файл test.txt, та бачимо, що там його нема (рис.3.35);

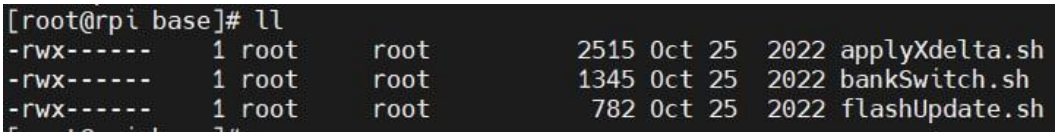

Рис.3.35. Вміст папки до оновлення

- викликаємо скрипт update.script на платі (рис.3.36);

Рис.3.36. Виклик процесу оновлення

- чекаємо завершення процесу оновлення та перезавантаження плати;
- перевіряємо з якого сховища перезавантажилась плата (рис.3.37);

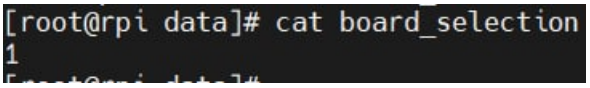

Рис.3.37. Сховище після процесу оновлення

– виводимо файли в папці в якій повинен бути тестовий файл test.txt (рис.3.38);

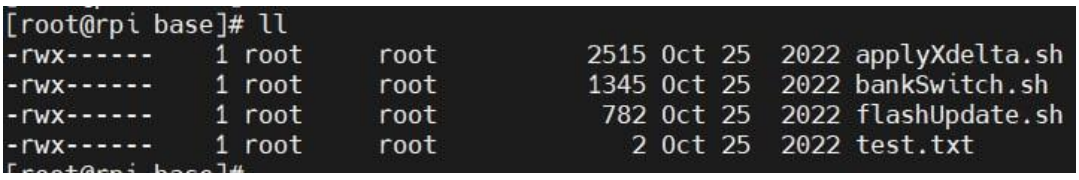

Рис.3.38. Вміст папки після оновлення

Отже, процес оновлення завершився та все працює правильно.

3.3. Висновки до розділу 3

Основними етапами розробки технології оновлення IoT систем на базі Embedded Linux є розробка систем для серверної та апаратної частин.

Для розробки серверної складової системи оновлення IoT систем на базі Embedded Linux були розроблені методи:

- для створення патчу оновлення;
- для створення xdelta файлів.

Для розробки апаратної складової системи оновлення IoT систем на базі Embedded Linux були розроблені методи:

- для розархівування патчу;
- для переміщення файлів;
- для перевірки переміщених файлів у відповідне місцезнаходження;
- для застосування оновлення з xdelta файлів;
- для перемикання між сховищами системи.

MobaXterm була використана як графічний інтерфейс за допомогою якої, користувачі могли взаємодіяти з одноплатним комп'ютером Raspberry Pi. Підключення до плати відбувалось за допомогою проколів SSH та FTP.

В результаті створена технологія оновлення IoT систем на базі Embedded Linux може бути використана розробником програмного забезпечення для оновлення IoT систем на базі Embedded Linux через графічний інтерфейс MobaXterm.

ВИСНОВКИ

Оновлення пристроїв не завжди потрібні, але важко придумати будь-яке програмне забезпечення, у якому не було б помилок. Навіть якщо програмне забезпечення ідеальне, але якщо пристрій зв’язується в мережі чи Інтернеті з будь-якими бібліотеками з відкритим кодом, оновлення системи безпеки може стати необхідністю.

Оновлення – це процес заміни продукту на новішу версію того самого продукту. У комп’ютерній техніці та споживчій електроніці оновлення – це, як правило, заміна апаратного забезпечення, програмного забезпечення або мікро-програми на новішу або кращу версію, щоб оновити систему або покращити її характеристики.

При розглядані існуючих технологій оновлення IoT систем на базі Embedded Linux було виявлено деякі недоліки, як погана кастомізація, тобто процес оновлення строго стандартизований та немає інших варіантів реалізацій, та складність використання. Отже, виникла необхідність створення власної системи для уникнення зазначених проблем.

В процесі розробки системи були використані такі основні технології:

- C++ – мова програмування загального призначення з підтримкою кількох парадигм програмування: об'єктно-орієнтованої, узагальненої, та процедурної;
- Bash – це програма оболонки інтерфейсу командного рядка, яка широко використовується в Linux. Сценарій(скрипт) bash – це звичайний текстовий файл, який містить низку команд;
- MobaXterm – це рішення для віддаленого робочого столу, яке надає підприємствам такі мережеві інструменти, як VNC, RDP або FTP, а також різні команди Unix для керування завданнями на віддаленому робочому столі Windows;
- Visual Studio Code – засіб для створення, редагування та впровадження сучасних застосунків і програм для хмарних систем. Visual Studio Code

розповсюджується безкоштовно і доступний у версіях для платформ Windows, Linux і OS X.

Основні можливості, які надає технологія оновлення IoT систем на базі Embedded Linux:

- створення патчу оновлення;
- створення xdelta файлів;
- розархівування патчу;
- переміщення файлів;
- перевірки переміщених файлів у відповідне місцезнаходження;
- застосування оновлення з xdelta файлів;
- перемикання між сховищами системи.

Для використання технології оновлення IoT систем на базі Embedded Linux її потрібно завантажити на пристрій користувача та запустити для створення патчу. Додаткового налаштування та завантаження система не потребує.

Даний функціонал системи в майбутньому можна розширити додавши:

- додати автоматичне пересилання патчу на пристрій;
- збереження інформації у хмарі.

Перевагами розробленої системи є безкоштовне використання, легкість в розумінні та швидкість процесу оновлення. Її використання підвищить швидкість процесу оновлення для систем будь-якого розміру.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ

1. What’s IoT device Updating? [Електронний ресурс]. – Режим доступу: https://jfrog.com/
2. OTA updates for Embedded Linux [Електронний ресурс]. – Режим доступу: https://www.embedded.com/
3. Software updates for embedded Linux: requirement and reality [Електронний ресурс]. Режим доступу: https://mender.io/
4. Strategies for an application doing software upgrade [Електронний ресурс]. – Режим доступу: https://sbabic.github.io/
5. Software updates for IoT edge Linux devices [Електронний ресурс]. – Режим доступу: https://jfrog.com/
6. Best Methods To Updating Embedded Linux Devices [Електронний ресурс]. – Режим доступу: https://jfrog.com/
7. Embedded LINUX | What is it, When and How to use it [Електронний ресурс]. – Режим доступу: https://tech.tribalyte.eu/
8. Guide to Building Embedded Linux Systems [Електронний ресурс]. – Режим доступу: https://pantacor.com/
9. C++ [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/
10. C++ for Embedded: Advantages, Disadvantages, and Myths [Електронний ресурс]. – Режим доступу: https://www.qt.io/
11. Pros and Cons of C++ for Embedded Systems [Електронний ресурс]. – Режим доступу: https://www.verypossible.com/
12. Python [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/
13. Python’s role in developing real time embedded systems [Електронний ресурс]. – Режим доступу: https://www.edn.com/
14. GNU Make [Електронний ресурс]. – Режим доступу: https://www.gnu.org/
15. What Is Bash Used For? [Електронний ресурс]. – Режим доступу: https://www.codecademy.com/

16. What is a Bash Script? [Електронний ресурс]. – Режим доступу: https://ryanstutorials.net/

17. What is systemd? [Електронний ресурс]. – Режим доступу: https://www.linode.com/

18. FTP [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/

19. What is FTP: FTP Explained for Beginners [Електронний ресурс]. – Режим доступу: https://www.hostinger.com/

20. Linux utilities that every developer should know [Електронний ресурс]. – Режим доступу: https://webbylab.com/

21. Visual Studio Code [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/

22. Why did we build Visual Studio Code? [Електронний ресурс]. – Режим доступу: https://code.visualstudio.com/

23. The Reasons why you Must Use Visual Studio Code [Електронний ресурс]. – Режим доступу: https://blog.devgenius.io/

24. MobaXterm [Електронний ресурс]. – Режим доступу: https://mobaxterm.mobatek.net/

25. Xdelta [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/

26. md5sum [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/

27. WinSCP [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/

28. PuTTY [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/

29. pipe() System call [Електронний ресурс]. – Режим доступу: https://www.geeksforgeeks.org/

ДОДАТОК А

Програмна реалізація файлу mkUpdatePatch.sh

```
PREVIOUS_VERSIONS=""
CURRENT_VERSION=""
OUTPUT_PATCH_NAME=""
DELTA_OUTPUT_DIR=""
IMAGE_BASED_DIFF_DIR=""
PATCH_DIR="/bin/update_patch"

createPatch()
{
	echo "Generating patch..."
	GENERATED_PATCH_NAME=$OUTPUT_PATCH_NAME

	echo "Started deltas generation"

	rm -rf "$PREVIOUS_VERSIONS/deltas"   # In case it exists
	mkdir -p "$PREVIOUS_VERSIONS/deltas/"

	DELTA_OUTPUT_DIR="$PREVIOUS_VERSIONS/deltas/"

   rm -rf "$CURRENT_VERSIONS/deltas"   # In case it exists
	mkdir -p "$CURRENT_VERSIONS/deltas/"

	mkDelta.sh -p "${PREVIOUS_VERSIONS}/bootfs.img" -c "$CURRENT_VERSIONS/bootfs.img" -o
"$DELTA_OUTPUT_DIR/bootfs" -t "bootfs"
	mkDelta.sh -p "${PREVIOUS_VERSIONS}/dmtable.img" -c "$CURRENT_VERSIONS/dmtable.img" -o
"$DELTA_OUTPUT_DIR/dmtable" -t "dmtable"
	mkDelta.sh -p "${PREVIOUS_VERSIONS}/rootfs.img" -c "$CURRENT_VERSIONS/rootfs.img" -o
"$DELTA_OUTPUT_DIR/rootfs" -t "rootfs"

	IMAGE_BASED_DIFF_DIR="$DELTA_OUTPUT_DIR"

   cd "$PATCH_DIR"

   tar -cvf "$GENERATED_PATCH_NAME" "$IMAGE_BASED_DIFF_DIR"
}


usage()
{
	usage="$(basename "$0") [-h] [-p -c -o]
                                                                                \n
	where:
                                                                                \n
		-h  show this help text with available options
                                                                                        \n
		-p	previous version from which the patch is created
                                                                                \n
		-c	the current version that will be after update                   \n
		-o  Specify the name of the generated patch
                                                                                \n
	"

	echo -e "$usage"
	exit
}
```

```
while getopts "p:c:o:h" opt; do
        case $opt in
        p)
                PREVIOUS_VERSIONS=$OPTARG
                echo "PREVIOUS_VERSIONS: $PREVIOUS_VERSIONS"
                ;;
        c)
                CURRENT_VERSION=$OPTARG
    echo "CURRENT_VERSIONS: $CURRENT_VERSIONS"
                ;;
        o)
                OUTPUT_PATCH_NAME=$OPTARG
                ;;
        :)
                echo "Option -$OPTARG requires an argument." >&2
                usage
                ;;
        esac
done

create_patch

echo -e "-> Patch generated in: $PATCH_BIN_DIR \n"


echo "Current versions is: $CURRENT_VERSION"
```

ДОДАТОК Б

Програмна реалізація файлу mkDelta.sh

```
PREVIOUS_PATH=""
CURRENT_PATH=""
OUTPUT_DIRECTORY=""
TITLE_BASE=""

generate_xdelta_files()
{
   DELTA_VAR=xdelta3

   PATCH_NAME="${TITLE_BASE}.xdelta3"
        INFO_NAME="${TITLE_BASE}.info"

        $DELTA_VAR -D0Nq -s "${PREVIOUS_PATH}" "${CURRENT_PATH}"
"$OUTPUT_DIRECTORY/$PATCH_NAME"

        MD5_SUM=$(md5sum -b "${CURRENT_PATH}")
        echo "$MD5_SUM" > "$OUTPUT_DIRECTORY/$INFO_NAME"
}


while getopts "p:c:o:t:h" opt; do
        case $opt in
        p)
                if [[ $OPTARG == *.bin ]] || [[ $OPTARG == *.img ]]
                then
                        PREVIOUS_PATH=$OPTARG
                fi
                ;;
        c)
                if [[ $OPTARG == *.bin ]] || [[ $OPTARG == *.img ]]
                then
                        CURRENT_PATH=$OPTARG
                fi
                ;;
   o)
                OUTPUT_DIRECTORY=$OPTARG
                mkdir -p $OUTPUT_DIRECTORY
                ;;
   t)
```

```
        TITLE_BASE=$OPTARG
        ;;
            :)
      echo "Option -$OPTARG requires an argument." >&2
              usage
      ;;

            esac
done

generate_xdelta_files

exit
```

ДОДАТОК В

Програмна реалізація файлу UpdateUnpack.cpp

```
void UpdateUnpack::unpackPatch(UpdatePatch& updatePatch, const std::string& destDir) {
    if (updatePatch.is_open()) {
                updatePatch.close();
                updatePatch.unzipFileName(destDir + "/");
        }
    else if ((!updatePatch.is_open()) {
                logError(db, "Update patch is not found.");
                THROW(UpdateException, ("Update patch is not found.", INVALID_PATCH));
        }
}
```

ДОДАТОК Г

Програмна реалізація файлу UpdateMvFiles.cpp

```
void UpdateMoveFile::copyFileFromZip(const std::string& fileName, const std::string& destFile) {


    outFile = fopen(destFile.c_str(), "w");
        if (outFile == NULL) {
            logError(db, "Unable create output file: %s", destFile.c_str());
            THROW(UpdateException, ("Unable create output file: " + destFile));
        }

        pipe = getFile(fileName);
        while (!feof(pipe)) {
            hasReadFile = fread(pipe);
            if (hasReadFile > 0) {
                writtenFile = fwrite(hasReadFile, outFile);
                if (writtenFile != outFile) {
                    logError(db, "Unable to write to output file: %s", destFile.c_str());
                    break;
                }
            }
        }
}
```

ДОДАТОК Д

Програмна реалізація файлу applyXdelta.sh

```
XDELTA_NAME="${PARTITON_NAME}.xdelta3"


apply_xdelta_patch()
{
        if [ "${NEWSTATE}" = "0" ]; then
                DEVICE_NAME="${PARTITON_NAME}0Device"
                NEWPARTITION=$(configuration os/partitions.config ${DEVICE_NAME})
        else
                DEVICE_NAME="${PARTITON_NAME}1Device"
                NEWPARTITION=$(configuration os/partitions.config ${DEVICE_NAME})
        fi

   xdelta3 -c -d -s "${TEMP_PARTITION}" < "${XDELTA_NAME}" > "${NEWPARTITION}"
   if [ $? != 0 ]; then
      echo "Applying partition patch failed"
      exit 1
   fi
}

#########################################################################################################
#####

BANK_VALUE=${1}
PARTITON_NAME=${2}

        if [ "${BANK_VALUE}" = "" ]; then
                BANK_VALUE=$(/opt/base/script/root/flash/bootSwitch.sh info)
        fi

        case "${BANK_VALUE}" in
                "0" )
                        DEVICE_NAME="${PARTITON_NAME}0Device"
                        PARTITION=$(configuration partitions.config ${DEVICE_NAME})
                        NEWSTATE=1
                ;;
                "1")
                        DEVICE_NAME="${PARTITON_NAME}1Device"
```

```
                PARTITION=$(configuration partitions.config ${DEVICE_NAME})
                NEWSTATE=0
        ;;
        *)
        echo "wrong parameter for set of partition (0 or 1 expected)!"
        exit 1
    esac

  TEMP_PARTITION="temp_${PARTITON_NAME}_partition"

    echo "Creating copy of partition: ${PARTITION} to ${TEMP_PARTITION}"

    dd if="${PARTITION}" of=${TEMP_PARTITION}

    applyXdeltaPatch

exit 0
```