Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет Автоматики і комп'ютеризованих технологій
(повна назва)

Кафедра Комп'ютерно-інтегрованих технологій, автоматизації та мехатроніки
(повна назва)

АТЕСТАЦІЙНА РОБОТА

Пояснювальна записка

другий (магістерський)
(рівень вищої освіти)

Розробка удосконалених засобів програмного керування мобільним роботом Festo Robotino
(тема)

Виконав: студент 2 курсу, гр. КТРСм-19-1
Крапивін В.С.
(прізвище, ініціали)

Спеціальність 151 – Автоматизація та комп'ютерно-інтегровані технології

освітньої програми Комп'ютеризовані та робототехнічні системи
(код і повна назва напряму)

Тип програми освітньо-професійна
(повна назва освітньої програми)

Керівник проф. Сінотін А.М.
(посада, прізвище, ініціали)

Допускається до захисту
зав. кафедри _____ Невлюдов І.Ш.
(підпис) (прізвище, ініціали)

2020

Харківський національний університет радіоелектроніки

Факультет	Автоматики і комп'ютеризованих технологій
Кафедра	Комп'ютерно-інтегрованих технологій, автоматизації та мехатроніки
Рівень вищої освіти	другий (магістерський)
Спеціальність	151 – Автоматизація та комп'ютерно-інтегровані технології
Тип програми	освітньо-професійна
Освітня програма	Комп'ютеризовані та робототехнічні системи

(код і повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____

(підпис)
«_____»_____ 2020 р.

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

студентові Крапивіну Владиславу Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка удосконалених засобів програмного керування мобільним роботом Festo Robotino
затверджена наказом по університету від 02.11. 2020 р. № 1509 Ст
2. Термін подання студентом роботи до екзаменаційної комісії 09.12. 2020 р.
3. Вихідні дані до роботи
4. Зміст пояснювальної записки (перелік питань, що потрібно розробити)
4.1. Вступ.
4.2. Аналіз особливостей роботи мобільного роботу Robotino.
4.3. Розробка архітектури засобів програмного керування мобільним роботом.
4.4. Розробка алгоритмів та програмного забезпечення
4.6. Висновки.
4.7. Перелік посилань.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) Демонстраційний матеріал представлений у форматі презентації PowerPoint (*.ppt) 15 с., формату А4.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Аналіз вихідних даних та літератури		
	за темою атестаційної роботи	05.09.2020	Виконано
2	Аналіз особливостей роботи		
	мобільного роботу Robotino	12.09.2020	Виконано
3	Постановка мети та задач		
	дослідження	16.09.2020	Виконано
4	Розробка архітектури засобів		
	програмного керування мобільним		
	роботом	14.10.2020	Виконано
5	Розробка програмної бібліотеки для		
	моделювання функції сенсорної	04.11.2020	Виконано
6	Оформлення пояснювальної записки		
	та презентації	22.11.2020	Виконано
7	Подання атестаційної роботи до		
	екзаменаційної комісії	09.12.2020	Виконано

Дата видачі завдання 01 вересня 2020 р.

Студент _____ Крапивін В.С.

(підпис) (прізвище, ініціали)

Керівник роботи _____ проф. Сінотін А.М.

(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 108 с., 1 табл., 34 рис., 3 дод., 28 джерел.

АЛГОРИТМ, СИСТЕМА, СЕРВЕР, АВТОМАТИЗАЦІЯ, СТРУКТУРА, СЕРЕДА РОЗРОБКИ, C#, C++, ROBOTINO, РОБОТИ.

Об’єкт дослідження – модель робота Robotino.

Предмет дослідження – моделі та методи віддаленого керування роботом Robotino.

Мета дослідження – удосконалення методів автоматизованого віддаленого керування роботом Robotino

Новизна дослідження полягає у розробці удосконалених програмних засобів, які дозволяють віддалене керування роботом на основі Web-сервісу.

Методи розробки – проектування структури системи керування базуючись на основних типах архітектур програмного забезпечення, дослідження та вибір основних методологій для розробки програмного забезпечення, які дозволяють ефективно використовувати ресурси, необхідні для створення програми. Основною задачею програми є автоматизоване керування елементами робота, використовуючи автоматичну компіляцію без наявності доступу до робота. Розробка ПК за допомогою середовища розробки Microsoft Visual Studio та мов C#, C++, розробка графічного інтерфейсу програми за допомогою Unity.

Було спроектовано архітектуру програми, яка має компонентну структуру; розроблені базові сценарії керування роботом Robotino для демонстрації роботи програми та її тестування.

ABSTRACT

Explanatory note: 108 pp., 1 tabs., 34 figs., 3 apps., 28 sources.

ALGORITHM, SYSTEM, SERVER, AUTOMATION, STRUCTURE, DEVELOPMENT ENVIRONMENT, C #, C ++, ROBOTINO, WORK.

The object of research is the Robotino robot model.

The subject of research - models and methods of remote control of Robotino robot.

The purpose of the study is to improve the methods of automated remote control of the Robotino robot

The novelty of the study is the development of advanced software that allows remote control of the robot based on the Web-service.

Development methods - designing the structure of the control system based on the main types of software architectures, research and selection of basic methodologies for software development that allow efficient use of resources needed to create the program. The main task of the program is automated control of robot elements, using automatic compilation without access to the robot. Development of a PC using the Microsoft Visual Studio development environment and C #, C ++ languages, development of the program's graphical interface using Unity.

The architecture of the program, which has a component structure, was designed; developed basic control scenarios for the Robotino robot to demonstrate the operation of the program and its testing.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

ПІД – Пропорційно-інтегрально-диференційний;

HTTP – HyperText Transfer Protocol;

JSON – JavaScript Object Notation;

LAN – Local Area Network;

REST – Representational State Transfer;

RPC – Remote Procedure Call;

TCP – Transmission Control Protocol;

WLAN – Wireless Local Area Network;

XML – eXtensible Markup Language.

ВСТУП

Автономні роботи є універсальними та популярними засобами у багатьох сферах таких як промисловість, будівництво, медицина, робота у середовищах небезпечних для людей та інші. Мобільні автономні роботи використовують власні сенсори для орієнтації та прийняття рішень для навігації у просторі без допомоги людини. Деякі роботи дозволяють роботу у гібридному режимі з можливістю передачі керування оператору. Програмування роботів може відбуватись або на етапі будування робота, що не дає можливості змінювати поведінку у процесі роботи, або у процесі виконання. Другий спосіб є найбільш гнучким, хоча і має складнощі із забезпеченням безпеки та стабільності виконання. Це дозволяє завантаження нового образу програми синхронно з централізованого сховища або терміналу.

Однією з реалізацій таких автономних мобільних систем є сімейство роботів Robotino. Вони використовуються для таких цілей як навчання, тренування та дослідження. Ці роботи мають систему двигунів, сенсори, камеру, набір інтерфейсів для зв’язку та апаратне забезпечення, що працюють під керуванням операційної системи Linux. Оскільки на базі даної платформи доцільно проводити різнопланові науково-практичні дослідження вдосконалення засобів програмного керування мобільного робота Robotino залишається актуальним завданням.

Об’єкт дослідження – модель робота Robotino.

Предмет дослідження – моделі та методи віддаленого керування роботом Robotino.

Новизна роботи полягає у розробці удосконалених програмних засобів, які дозволяють віддалене керування роботом на основі Web-сервісу.

Для досягнення поставленої мети необхідно вирішити такі завдання:

– проаналізувати структуру та характеристики роботи робота Robotino;

– проаналізувати особливості роботи робота Robotino;

– проаналізувати протокол керування роботом Robotino;

– розробити систему керування роботом Robotino;

– розробити систему компіляції та розповсюдження образу програми для керування роботом Robotino;

– розробити систему зворотного зв’язку для терміналу управління роботом Robotino;

– створити образ на базі операційної системи Linux для централізованого контролю, збору інформації роботами Robotino у середовищі Robotino View за допомогою прикладної програми використовуючи практики розробки інтерфейсів користувача.

– оформити атестаційну роботу відповідно ДСТУ 3008-2015 [1], навчальному посібнику з дипломного проектування [2], методичних вказівок до випускної кваліфікаційної роботи рівня «Магістр» [3] та положенню про протидію академічному плагіату [4].

1 АНАЛІЗ ОСОБЛИВОСТЕЙ МОБІЛЬНОГО РОБОТА

1.1 Аналіз засобів керування мобільними роботами

Для мобільної робототехники однією з головних проблем є керування роботом без людини-оператора. Це вимагає аналіз роботом своїх координат у просторі, координат інших об’єктів, відстань до них та інше [5]. Як наслідок, для того, щоб виконати переміщення з однієї точки до іншої робот має коректувати свій рух [6].

У випадку автономної роботи є 2 типи прийняття рішень: жорсткі та гнучкі.

Жорсткий тип є програмним алгоритмом, що заданий людиною-розробником. Цей тип є найбільш простим та не потребує великої кількості апаратних ресурсів для роботи. Із недоліків можна вказати неможливість адаптації або навчання і різних середовищах, що має наслідок у обмеженості середовищ, де цей тип безпечно або можливо використовувати.

Згладити цю проблему можливо завдяки збільшенню кількості та ускладненню сенсорів робота, таких як лідар, датчики зіткнення, камери. Прикладом цього типу може бути ARAGON USV, опис сенсорів якого можна побачити на рисунку 1.1 [7].

До гнучких типів відносяться машинне навчання. Це дозволяє створення роботів, що можуть адаптуватися до навколишнього середовища. Такі роботи використовують такі ж сенсори, але замість перевірки жорстких умов, вхідні дані оброблюються моделлю, створеною завдяки навчальному набору даних [8]. Головною проблемою, що ускладнює впровадження таких типів є відсутність достатньої кількості даних для навчання.

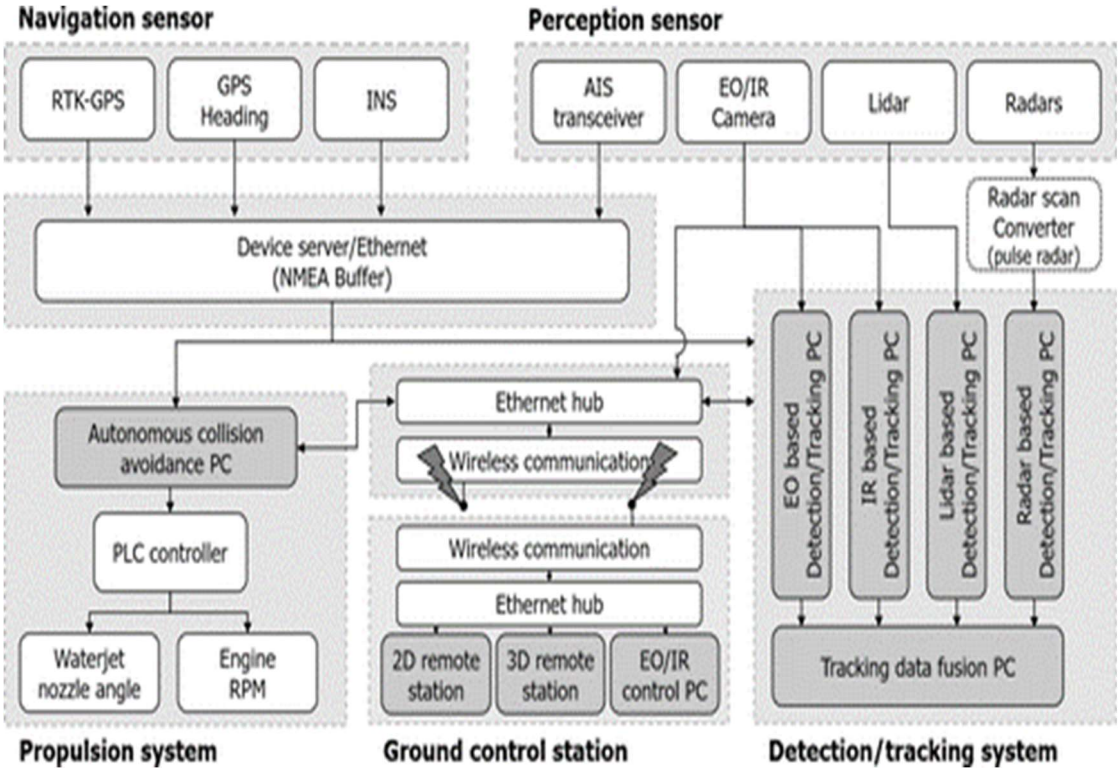

Рисунок 1.1 – Апаратні сенсори ARAGON USV

Управління роботом можна зобразити на рисунку 1.2 або більш конкретну версію – на рисунку 1.3.

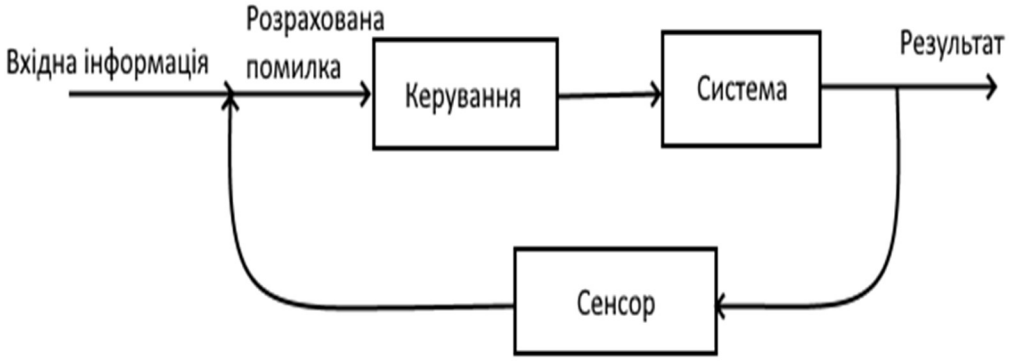

Рисунок 1.2 – Управління системою

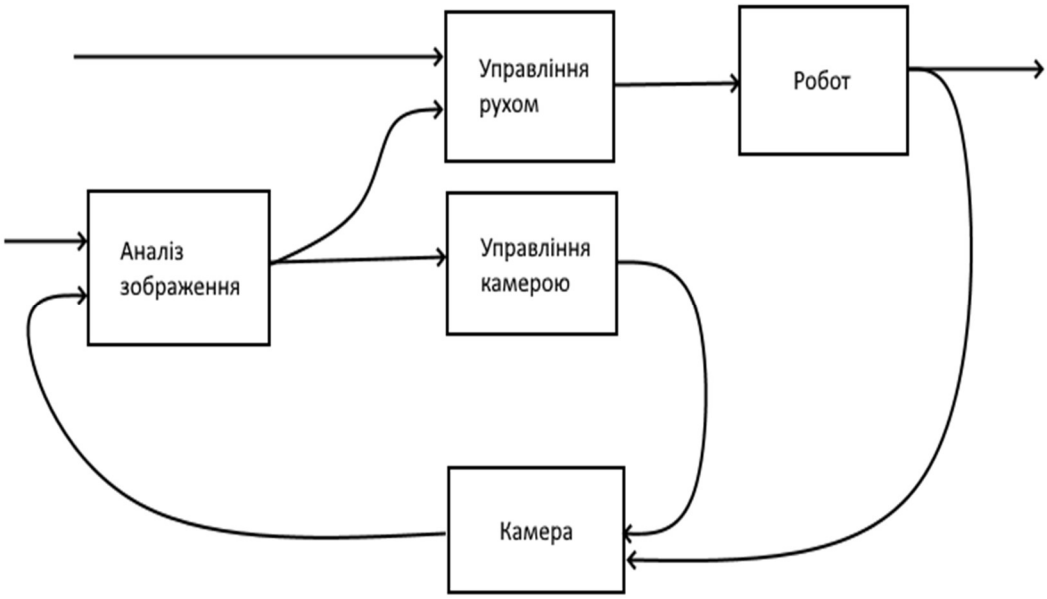

Рисунок 1.3 – Управління системою з камерою

Автономний робот має таки частини:

– система керування; завдяки цієї частини робот може аналізувати різницю між очікуваним результатом та фактичним, це дає можливість адаптувати поведінку робота у випадку автономної роботи;
– операційна система; безпосередньо система, керування якої відбувається у даному циклі;
– сенсорна система; важливий елемент, що дає можливість зворотнього зв’язку та зміни поведінки робота.

1.2 Опис структури робота Robotino

Мобільний робот Robotino розроблений для цілей навчання та тренування у сферах автоматизації та технологій (зображений на рисунку 1.4).

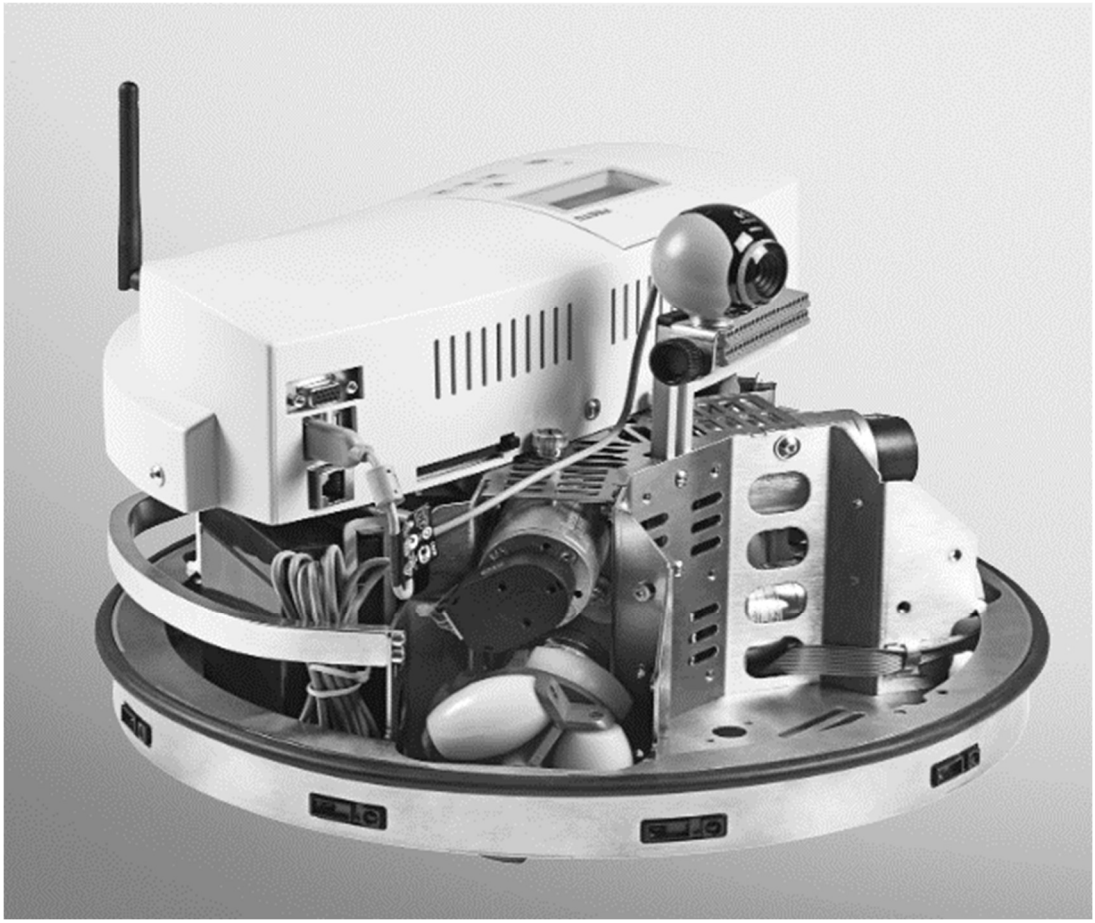

Рисунок 1.4 – Вигляд мобільного робота

Найголовнішими властивостями цього робота можна назвати:

– система керування рухом з колесами;
– ПІД керовані мотори;
– инфракрасні сенсори для контролю дистанції;
– бампери для контролю дотику;
– вбудований комп’ютер;
– інтегрована камера;
– комунікація з комп’ютером за допомогою WLAN.

Робот може рухатися у горизонтальній площині та обертатися навколо своєї осі. Це можливо завдяки 3 незалежним колесам. Також з причини симетричності робота, не можливо цілком вказати передню частину або позитивний напрям руху так як робот може вільно рухатися у любий напрям.

Кожне з колес робота, які розташовані на рівних відстанях один від одного на колі, оснащено мотором. Та завдяки комбінуванню інтенсивності обертання цих колес можна реалізувати рух у будь-якому напряму.

Схема модулю руху з сайта документації робота Robotino зображена на рисунку 1.5 [9].

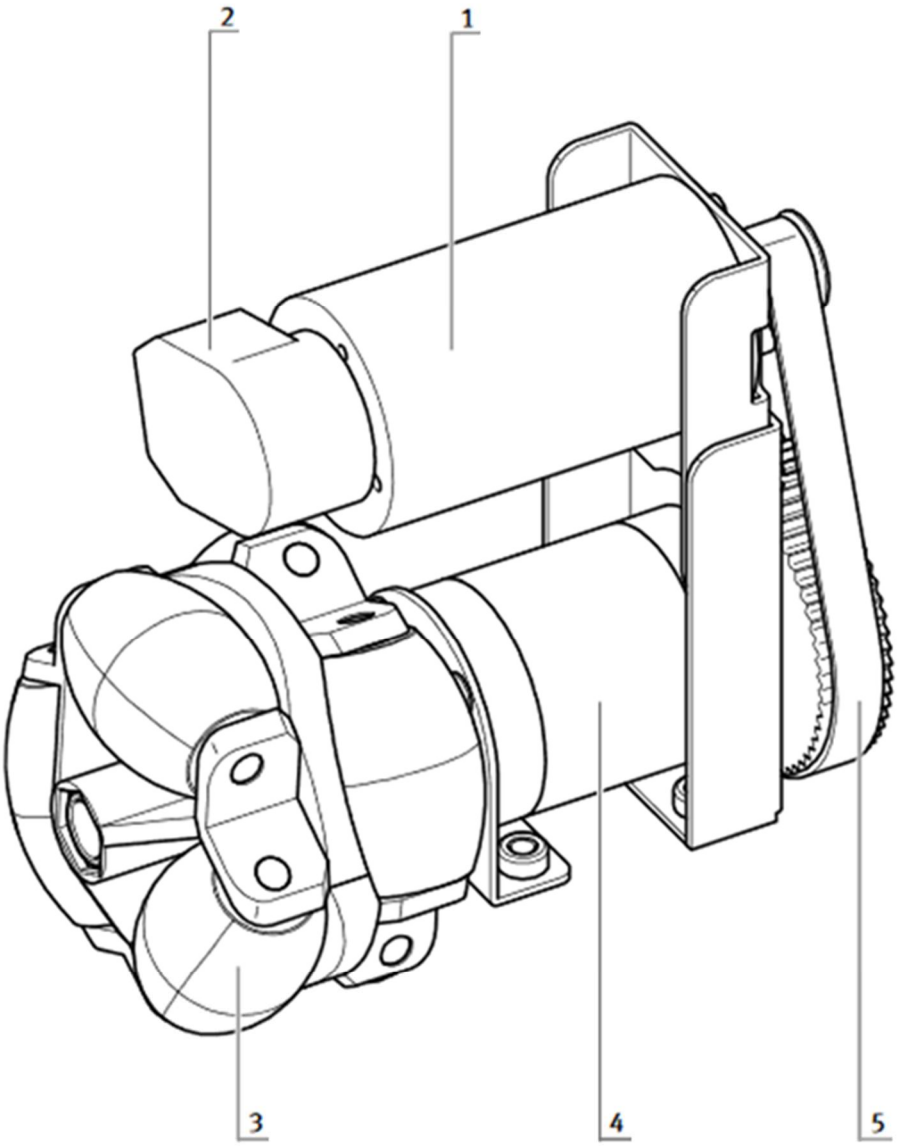

Рисунок 1.5 – Схема модуля руху

Камера робота підтримує роботу відео у форматах:

– VGA, 640x480, 15 fps;

– CIF, 352x288, 30 fps;
– QVGA, 320x240, 30 fps;
– QCIF, 176x144, 30 fps;
– SQCGA, 160x120, 30 fps.

Або статичне зображення у форматах BMP та JPG у максимальному розширенні 1024x768.

Для управління роботом використовується точка доступу Wireless LAN за стандартом IEEE 802.11g та 802.11b. Це дозволяє швидкості 54 *МБ/с* та 11 *МБ/с* на відстані до 100 *м*.

1.3 Опис системи керування роботом Robotino

Керування роботом виконується за допомогою двоповерхового протоколу RPC. RPC, або Remote procedure call, протокол, що дозволяє програмі, запущеній на одному комп'ютері, звертатись до функцій (процедур) програми, що виконується на іншому комп'ютері, подібно до того, як програма звертається до власних локальних функцій [10].

Цей протокол є універсальним протоколом для двостороннього зв’язку 2 вузлів мережі. RPC може бути реалізований різними шляхами, тому не існує єдиного стандарту для цього формату мережевої комунікації.

З урахуванням відсутності документації, як наслідок, формат Robotino RPC, або далі REC RPC, є протоколом без відкритої специфікації та має лише один набір програмних інтерфейсів на базі мови програмування C++.

На рисунках 1.6 та 1.7 зображено стадію ініціалізації та процес надіслання команди для руху робота.

```
REC_RPC_example_circleRobotino RPC Server

..t..<configuration>
 <item enqueued="true" id="0" name="rec_robotino_rpc_process_status"/>
 <item enqueued="true" id="1" name="rec_robotino_rpc_process_output"/>
 <item permanent="true" id="2" name="rec_robotino_rpc_camera0_settings"/>
 <item id="3" name="rec_robotino_rpc_set_camera0_settings"/>
 <item id="4" name="rec_robotino_rpc_set_camera0_control"/>
 <item permanent="true" id="5" name="rec_robotino_rpc_camera0_capabilities"/>
 <item permanent="true" id="6" name="rec_robotino_rpc_camera0_calibration"/>
 <item permanent="true" id="7" name="rec_robotino_rpc_camera1_settings"/>
 <item id="8" name="rec_robotino_rpc_set_camera1_settings"/>
 <item id="9" name="rec_robotino_rpc_set_camera1_control"/>
 <item permanent="true" id="10" name="rec_robotino_rpc_camera1_capabilities"/>
 <item permanent="true" id="11" name="rec_robotino_rpc_camera1_calibration"/>
 <item permanent="true" id="12" name="rec_robotino_rpc_camera2_settings"/>
 <item id="13" name="rec_robotino_rpc_set_camera2_settings"/>
 <item id="14" name="rec_robotino_rpc_set_camera2_control"/>
 <item permanent="true" id="15" name="rec_robotino_rpc_camera2_capabilities"/>
 <item permanent="true" id="16" name="rec_robotino_rpc_camera2_calibration"/>
 <item permanent="true" id="17" name="rec_robotino_rpc_camera3_settings"/>
 <item id="18" name="rec_robotino_rpc_set_camera3_settings"/>
 <item id="19" name="rec_robotino_rpc_set_camera3_control"/>
 <item permanent="true" id="20" name="rec_robotino_rpc_camera3_capabilities"/>
 <item permanent="true" id="21" name="rec_robotino_rpc_camera3_calibration"/>
 <item id="22" name="rec_robotino_rpc_motor0_setpoint"/>
 <item id="23" name="rec_robotino_rpc_motor0_reset_position"/>
 <item permanent="true" id="24" name="rec_robotino_rpc_motor0_mode"/>
 <item id="25" name="rec_robotino_rpc_set_motor0_mode"/>
 <item id="26" name="rec_robotino_rpc_motor1_setpoint"/>
 <item id="27" name="rec_robotino_rpc_motor1_reset_position"/>
 <item permanent="true" id="28" name="rec_robotino_rpc_motor1_mode"/>
 <item id="29" name="rec_robotino_rpc_set_motor1_mode"/>
 <item id="30" name="rec_robotino_rpc_motor2_setpoint"/>
 <item id="31" name="rec_robotino_rpc_motor2_reset_position"/>
 <item permanent="true" id="32" name="rec_robotino_rpc_motor2_mode"/>
 <item id="33" name="rec_robotino_rpc_set_motor2_mode"/>
 <item id="34" name="rec_robotino_rpc_motor3_setpoint"/>
 <item id="35" name="rec_robotino_rpc_motor3_reset_position"/>
 <item permanent="true" id="36" name="rec_robotino_rpc_motor3_mode"/>
 <item id="37" name="rec_robotino_rpc_set_motor3_mode"/>
 <item id="38" name="rec_robotino_rpc_motor_setpoints"/>
 <item permanent="true" id="39" name="rec_robotino_rpc_motor_readings"/>
 <item id="40" name="rec_robotino_rpc_omnidrive"/>
 <item permanent="true" id="41" name="rec_robotino_rpc_odometry"/>
 <item id="42" name="rec_robotino_rpc_set_odometry"/>
 <item id="43" name="rec_robotino_rpc_northstar"/>
 <item permanent="true" id="44" name="rec_robotino_rpc_set_northstar_parameters"/>
 <item id="45" name="rec_robotino_rpc_gyroscope"/>
 <item permanent="true" id="46" name="rec_robotino_rpc_emergency_bumper"/>
 <item id="47" name="rec_robotino_rpc_set_emergency_bumper"/>
 <item enqueued="true" id="48" name="rec_robotino_rpc_display_text"/>
 <item id="49" name="rec_robotino_rpc_display_backlight"/>
 <item id="50" name="rec_robotino_rpc_display_buttons"/>
 <item id="51" name="rec_robotino_rpc_display_vbar"/>
 <item id="52" name="rec_robotino_rpc_display_hbar"/>
 <item id="53" name="rec_robotino_rpc_display_progress"/>
```

Рисунок 1.6 – Приклад мережевого повідомлення REC RPC

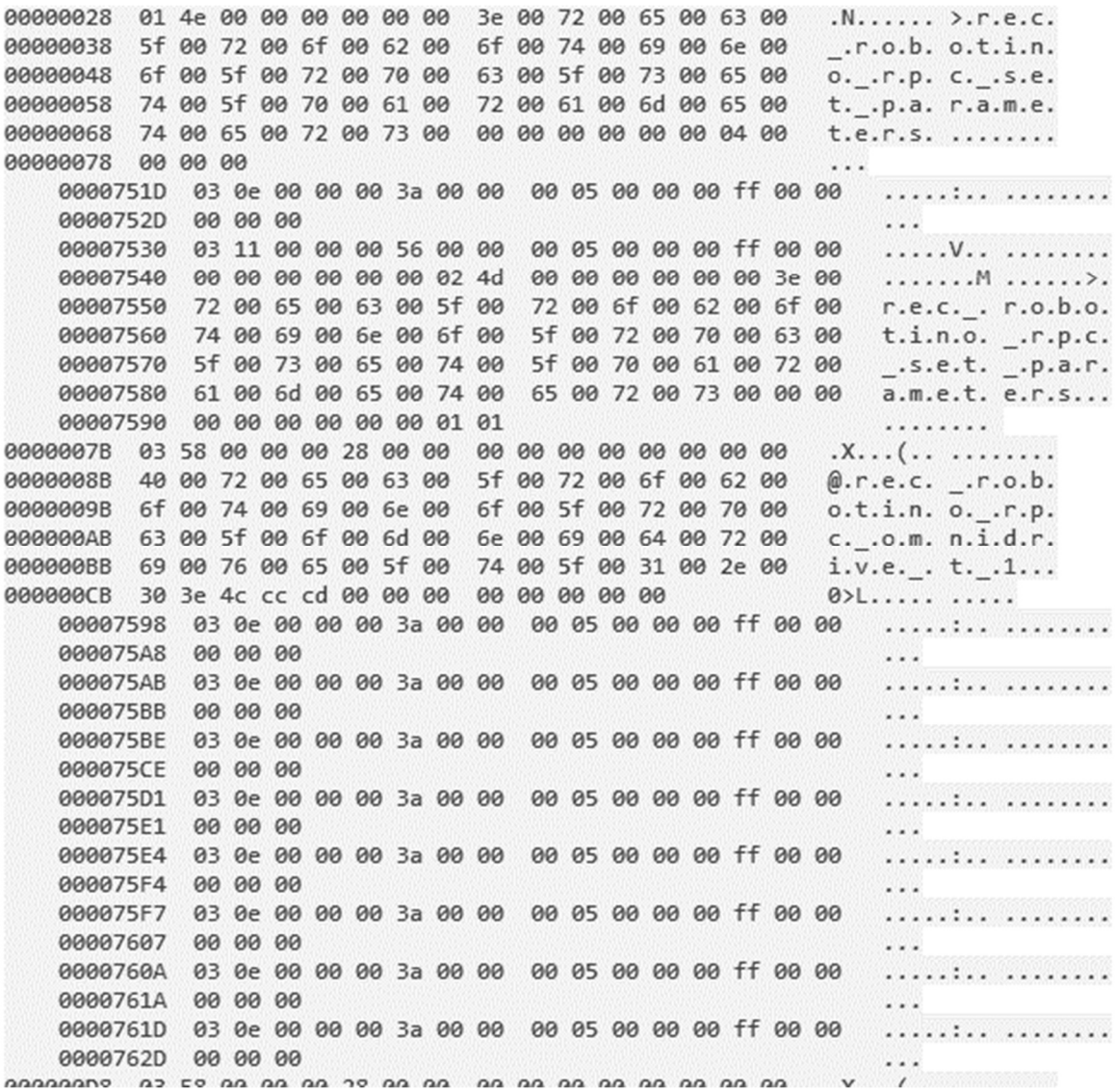

Рисунок 1.7 – Приклад структури повідомлення для управляння двигунами робота

Порядок обміну інформацією між роботом Robotino та клієнтом можна побачити на рисунку 1.8.

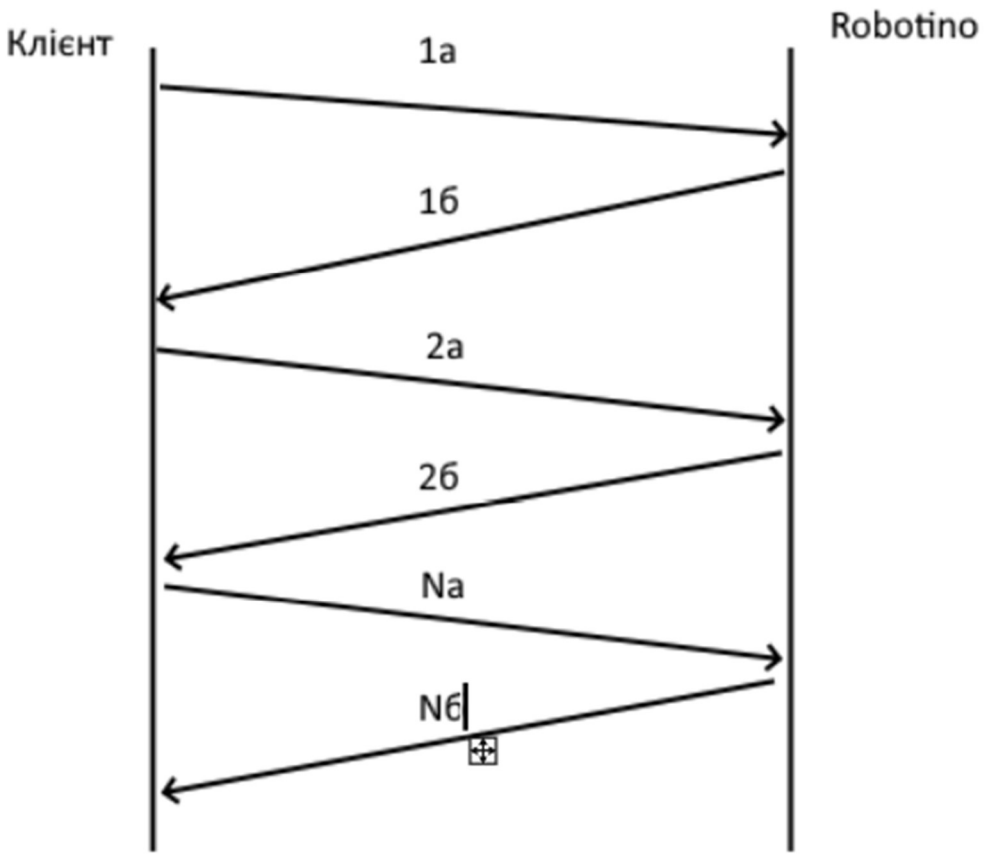

Рисунок 1.8 – Порядок повідомлень RPC

Кроки 1а та 1б є узагальненням TCP handshake [11], що зображено на рисунку 1.9.

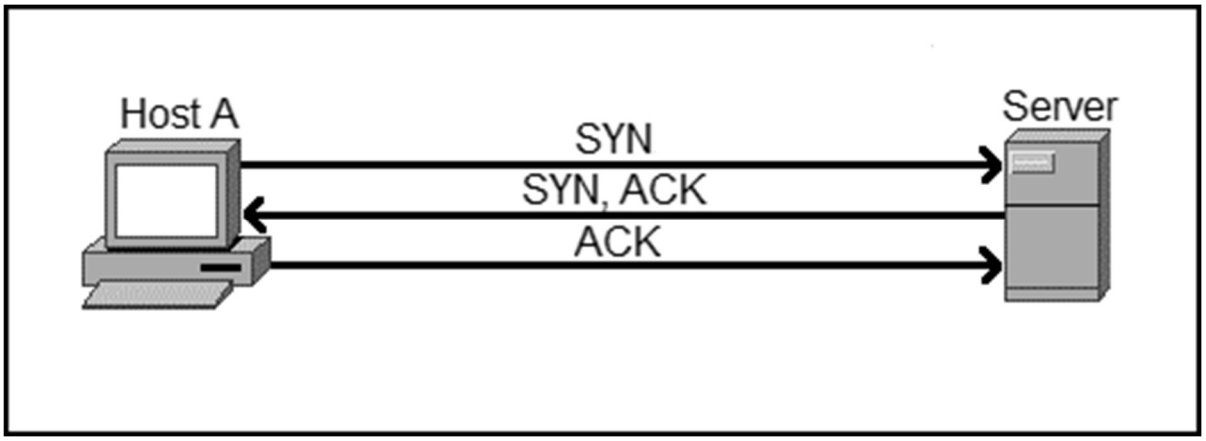

Рисунок 1.9 – TCP handshake

Кроки 2а та 2б є ініціалізацією протоколу спілкування з роботом.

Існує 2 версії цього протоколу:

– версія для Robotino API v1 або Robotino View [12];
– версія для Robotino API v2.

Robotino View є графічним середовищем для розробки та тестування робота Robotino. Це програмне забезпечення має можливість виводу інформації у реальному часі кожного з параметрів, обновлення коду програми без необхідності перекомпіляції коду.

Найважливіших функції, що входять до складу Robotino View:

– програми відображаються як GRAFCETs. Діаграми GRAFCET — це елементи керування робочим процесом, які використовують мову специфікації GRAFCET (GRAphe Fonctionnel de Commande Etapes/Transitions) по системі стандартів DIN EN 60848;
– одночасний контроль більш ніж одного Robotino;
– представлення апаратних компонентів як функціональних блоків;
– двигуни, IO, датчики, камера, одометрія, захват, маніпулятор, вихідна потужність, вхід валу;
– блок функцій для обробки зображень;
– лінійний детектор, пошук колірного діапазону, виявлення маркерів
– функціональні блоки для навігації;
– навігатор положення, навігатор відстані, уникнення перешкод;
– функціональні блоки для обміну даними;
– UDP, клієнт/сервер TCP/IP, OPC;
– створення та інтеграція індивідуалізованих функціональних блоків у C++.

На рисунку 1.10 зображується проект Robotino View, створений завдяки візуальним засобам програмування.

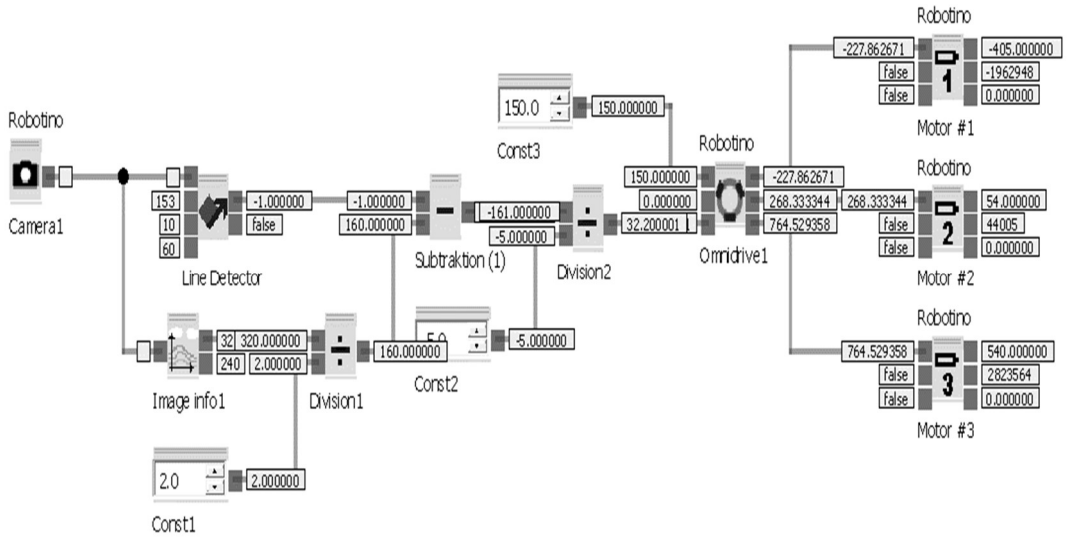

Рисунок 1.10 – Приклад програми у Robotino View

На рисунку 1.11 можна побачити відео у реальному часі з тестового роботу.

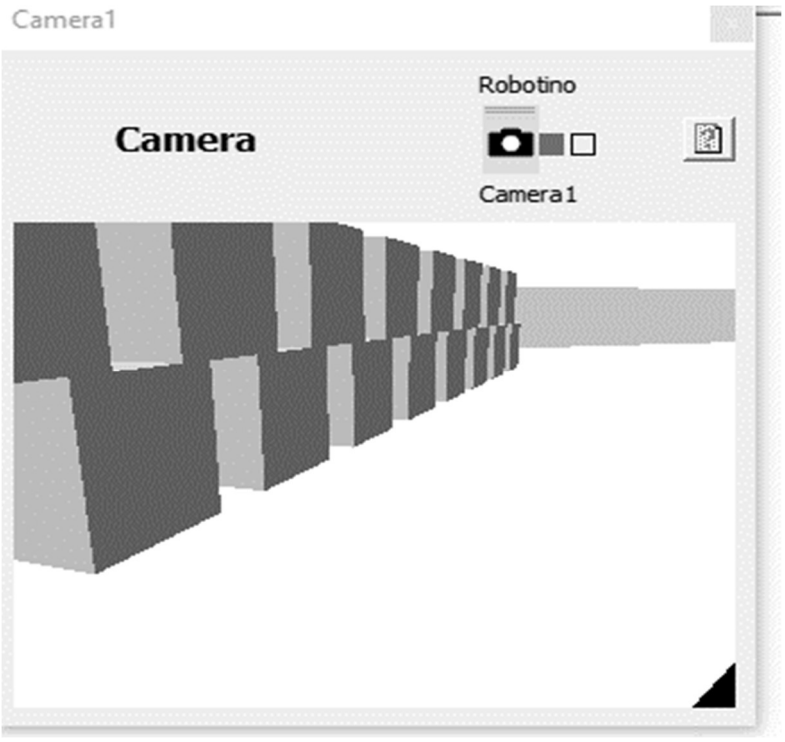

Рисунок 1.11 – Приклад зображення з камери робота

Версія v1 не є публічно доступною та використовується лише у програмному забезпеченні для візуального програмування Robotino View, що не має інтерфейсу для розширення функцій. До того ж Robotino View не підтримує можливість використання загальних мов програмування. Єдиним доступним варіантом є мова візуального програмування, що має закриту реалізацію. Тому для подальшого аналізу розглядається тільки версія v2.

Для управління роботом у середовищі мови програмування C, C++ єдиним оптимальним шляхом є завантаження та встановлення бібліотеки Robotino API v2. У випадку інших мов програмування треба викликати процедури для адаптації викликів функцій. Наприклад, у випадку мови програмування Python – ctypes [13].

Robotino API v2, або як вказано у документації - API2 [14], постачається у зібраному виді для таких систем, як Windows та Ubuntu. Зібраний пакет складається з такого набору файлів: .h файли з описом інтерфейсу бібліотеки, .lib файли з програмною реалізацією бібліотеки, FindRobotinoAPI2 для використання у програмному забезпеченні CMake. Але цей пакет не містить коду з файлами .c та .cpp, або інформації для тестування.pdb.

Головною умовою та складністю підключення до робота є необхідність розташування робота та програми керування в одній мережі. Це можливо за рахунок того, що робот Robotino має повноцінний образ системи Linux та WiFi модуль, що створює локальну мережу WiFi. Як результат клієнт цієї мережі має можливість посилати та приймати повідомлення від робота. Але це накладає обмеження на дальність знаходження пристрою керування та як результат людини-розробника. Також ускладняється контроль багатьох роботів, що насамперед збільшує вірогідність помилки. Структуру комунікації між роботом та клієнтом зображено на рисунку 1.12.

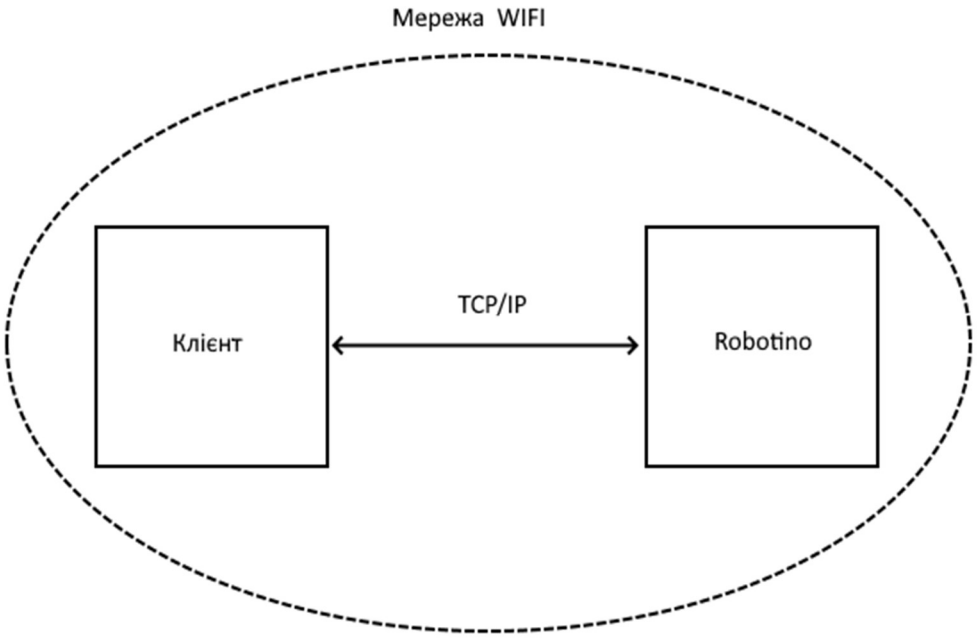

Рисунок 1.12 – Мережева структура зв’язку з роботом

1.4 Постановка мети та задач дослідження

Аналіз системи керування роботом Robotino, незважаючи на гнучке налаштування, потребує подальшого удосконалення. Впровадження змін дозволяє автоматизувати процес оновлення програмного забезпечення керуванням багатьма роботами, завдяки чому можна зменшити час внесення змін до роботів, мінімізувати кількість помилок та розходжень між роботами, працювати віддалено. Все це, як результат, зменшує витрати на обслуговування системи роботів.

Для заданої задачі удосконалення засобів програмного керування мобільним роботом Festo Robotino необхідно:

– реалізувати можливість контролю декількома роботами Robotino, використовуючи ідеї Robotino API v1 (Robotino View);
– дозволити користувачеві проводити оновлення програми керування роботом без прямого контакту з роботом;

– оптимізувати час оновлення програмного забезпечення, потенційно зрівнявши його з часом оновлення коду в Robotino View;
– розробити можливість перегляду значень програми на вимогу користувача, відштовхуючись від ідей Robotino View.

У зв’язку з відсутністю відкритого коду роботу Robotino та Robotino API v1/v2, розв’язання задачі буде відбуватися за допомогою Robotino API v2, як єдиний та стабільний програмний інтерфейс до робота.

Розробка буде проходити для систем Windows та Linux, як медіатори [15] у спілкуванні з роботом Robotino, та платформи Web для інтерфейсу програмного керування роботом.

Схему задачі для реалізації відображено на рисунку 1.13.

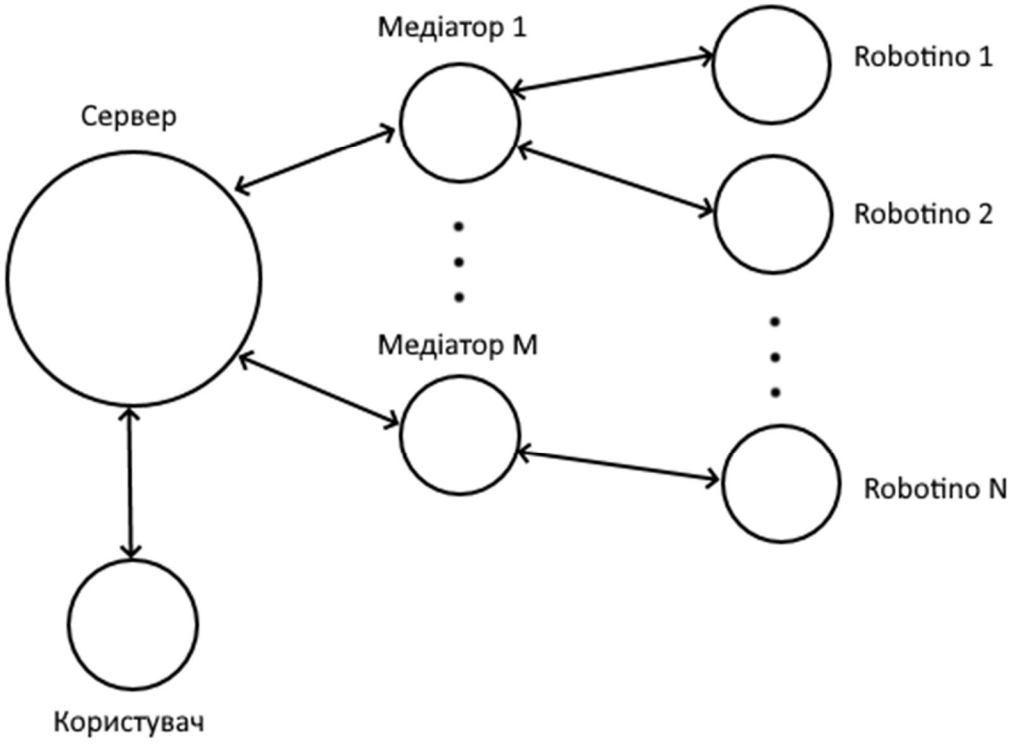

Рисунок 1.13 – Схема задачі для реалізації

2 РОЗРОБКА АРХІТЕКТУРИ ЗАСОБІВ ПРОГРАМНОГО КЕРУВАННЯ МОБІЛЬНИМ РОБОТОМ

2.1 Визначення архітектури системи керування

Враховуючи необхідність управління роботами віддалено, головною частиною системи є сервер, що спрямовує запити клієнта до відповідного робота та готує відповідь на цей запит для подальшого відображення. Це можна зобразити таким чином, як на рисунку 2.1.

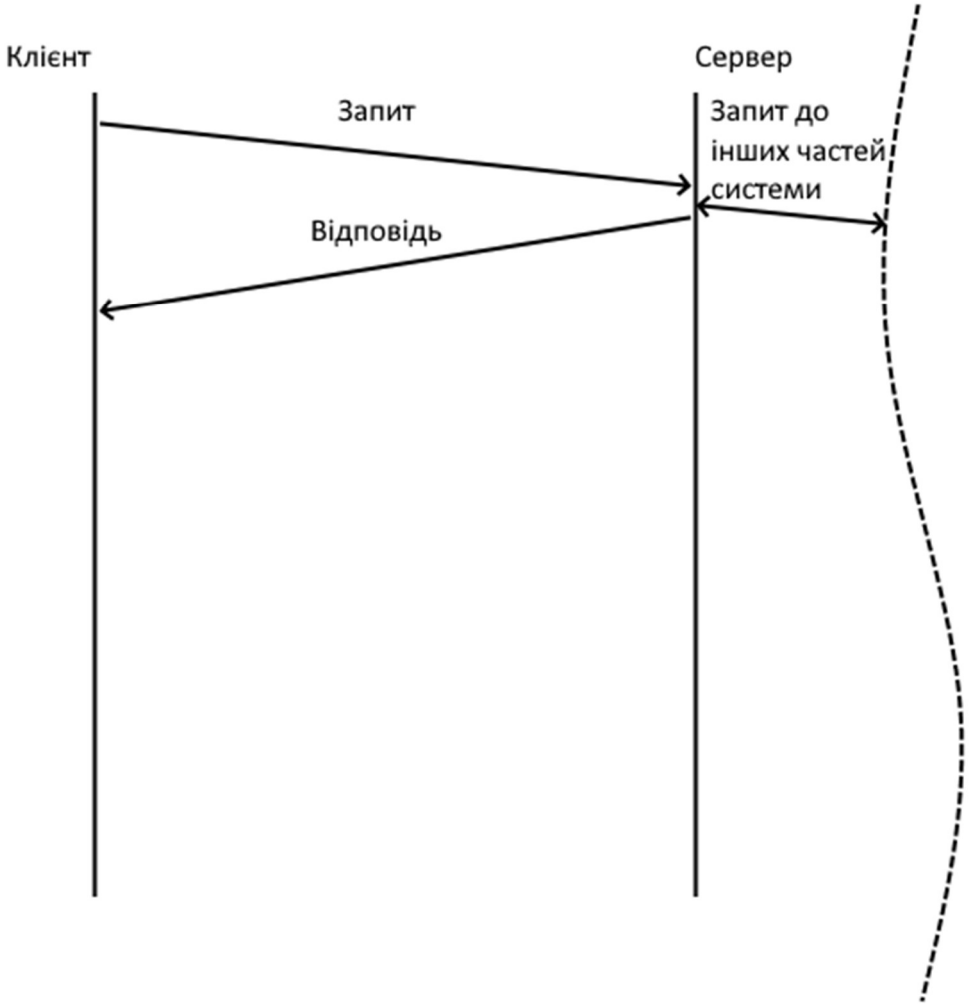

Рисунок 2.1 – Взаємодія клієнта та сервера

Клієнтів може бути декілька, тому усі операції повинні виповнятись атомарно, тобто зміна стану системи не має призводити до невизначеного значення у випадку паралельного запису або паралельного запису та зчитування.

Для збільшенні надійності системи та задля зберігання постійних даних використовується база даних, базова схема, якої є на рисунку 2.2.

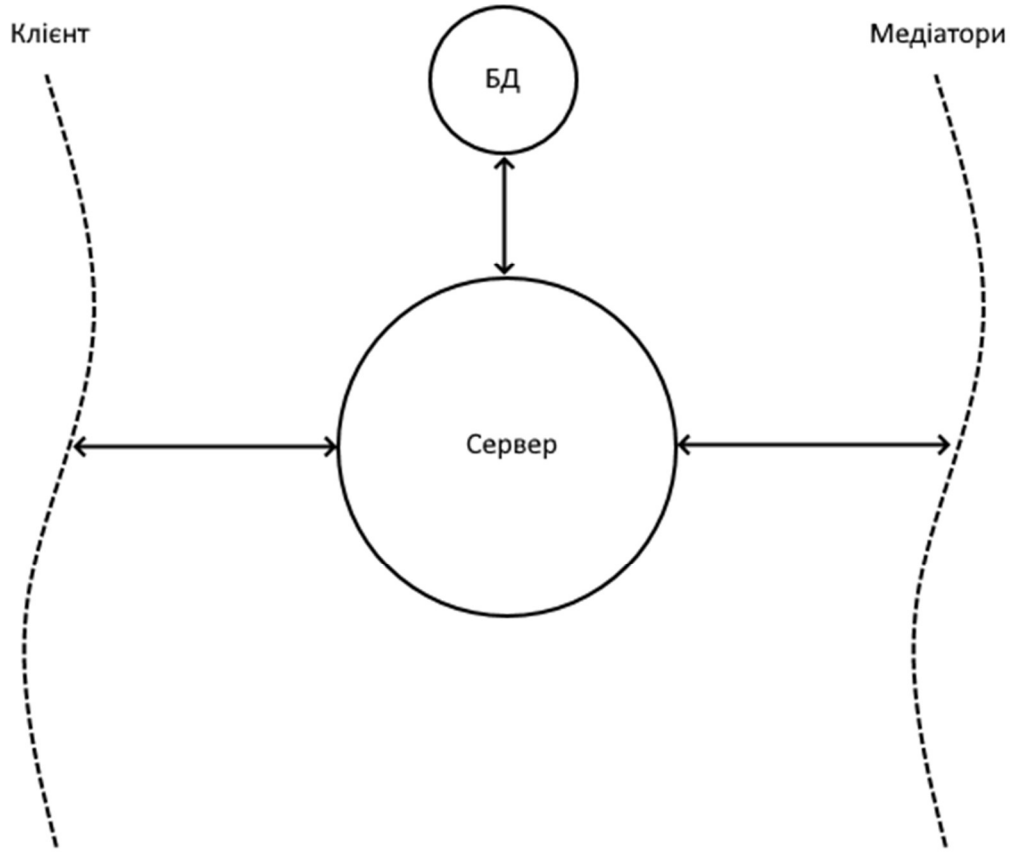

Рисунок 2.2 – Взаємодія серверу з базою даних

Така архітектура серверу є достатньо простою з мінімальною кількістю вузлів, які можуть відказати. Також у випадку надзвичайної ситуації з сервером база даних не буде порушена, що дозволе швидко відновити роботу усієї системи.

Зв’язок між медіаторами та сервером є подібним до зв’язку між клієнтом та сервером. Головною різницею є те, що медіатори не знають про клієнта та ізольовані один від одного. Схема взаємодії зображена на рисунку 2.3.

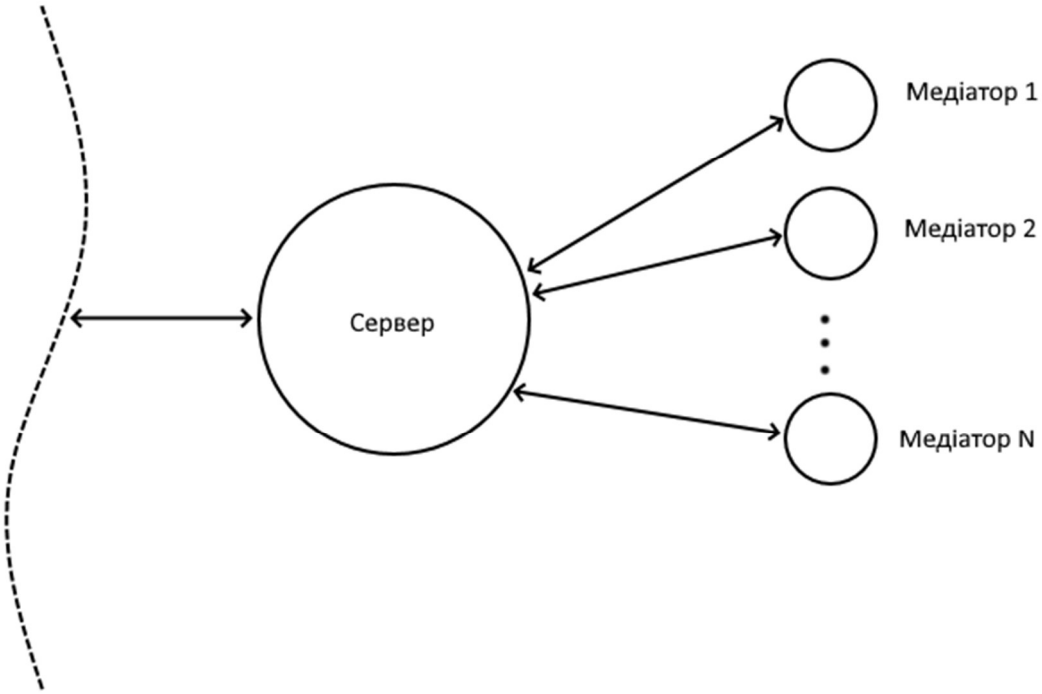

Рисунок 2.3 – Взаємодія серверу з медіаторами

У випадку існування декількох роботів, які знаходяться неподалік один від одного, існує можливість використання одного медіатору для керування ними. Для цього пристрій, який виконує код медіатора повинен мати можливість приєднатися до декількох мереж Wi-Fi. У якості альтернативи використовується відношення один до одного, тобто 1 медіатор – 1 робот.

У будь-якому випадку медіатор містить лише один клієнт сервера, що дозволяє зменшити кількість даних, які пересилаються мережею та спрощує комплексні операції у випадку синхронного виконання команд.

Незважаючи на те що, серверами є роботи Robotino, а медіатор є клієнтом, поведінка зворотна – роботи Robotino підкорюються одержуваним командам. Тому далі буде використовуватися також поняття сервер медіатору для позначення цих зв’язків.

Схему зв’язку медіатора та Robotino зображено на рисунку 2.4.

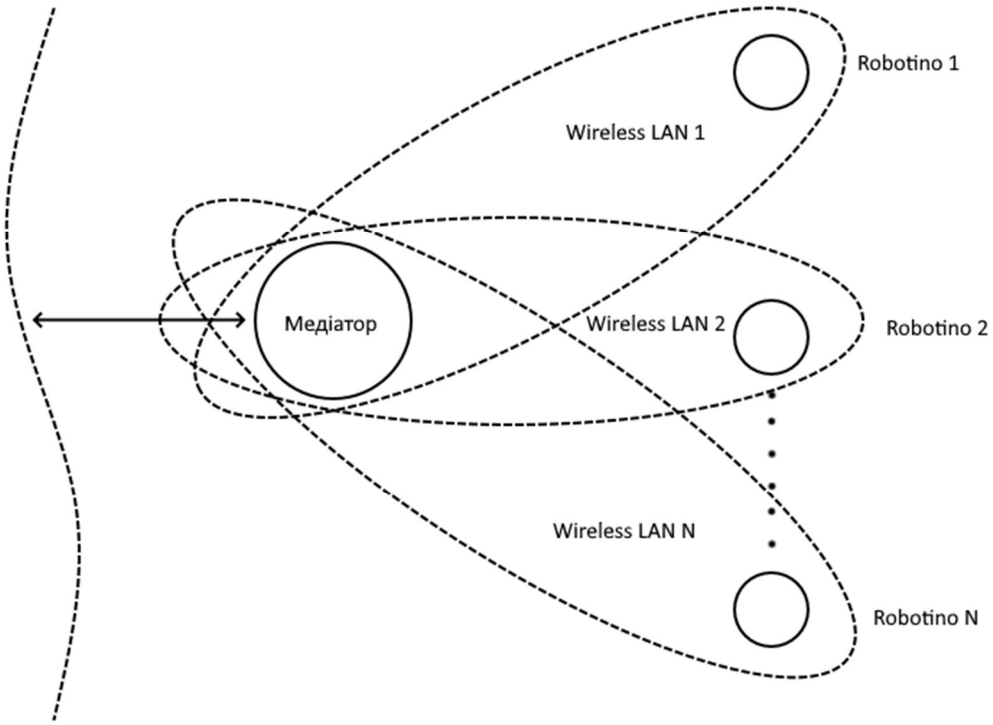

Рисунок 2.4 – Взаємодія медіатору з роботами Robotino

2.2 Розробка клієнт-серверного зв’язку

Програма клієнт потрібна мати такі можливості:

– робота на різних операційних системах;
– можливість здійснювати запити та оброблювати їх у реальному часі;
– можливість зображення інформації та зображення з робота;
– зручне оновлення програмного забезпечення.

Враховуючи перший та четвертий пункт, виконання у браузері є найбільш ефективною відповіддю. Альтернативою є виконання у якості прикладної програми на комп’ютері користувача. Ця альтернатива потребує рекомпіляції програми клієнту для кожної системи та ускладнює можливість оновлення додатку, вимагаючи додаткової підсистеми оновлення та контролю версій.

Для вирішення другої та третьої проблеми застосовується WebAssembly. Цей формат був розроблений для виконання браузерами коду на мовах C, C++, c#, Rust. Та дозволяє переносити код на цих мовах до браузерного середовища,

зберігаючи високу швидкість виконання [16]. Процес розповсюдження WebAssembly описаний на рисунку 2.5.

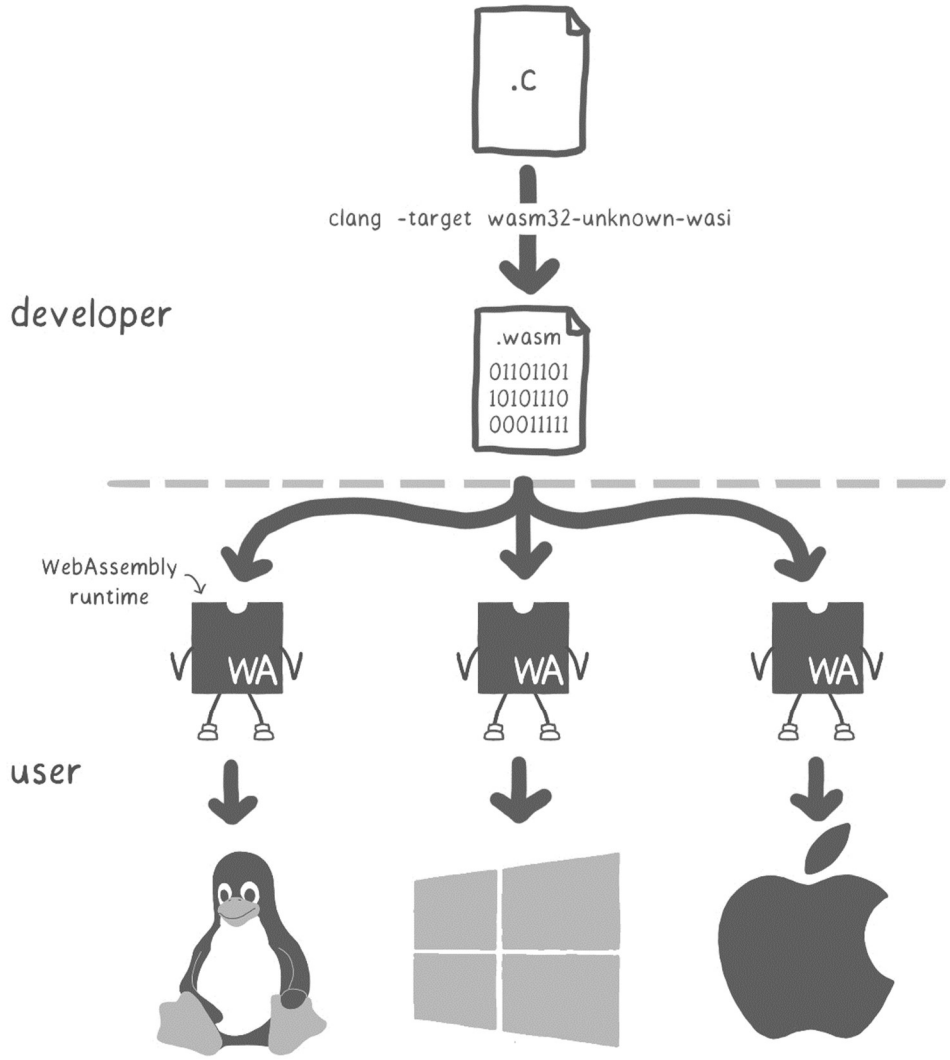

Рисунок 2.5 – Схема розповсюдження WebAssembly [17]

Інтерфейс користувача відображається за допомогою програмного фреймворку Unity.

Unity має багатий функціонал для майбутнього розширення, стабільний програмний інтерфейс та надійну підтримку платформи WebGL, що дозволяє публікувати Unity C# програми у середовищі HTML5/JavaScript [18].

Реалізація розповсюдження WebAssembly в Unity відображена на рисунку 2.6.

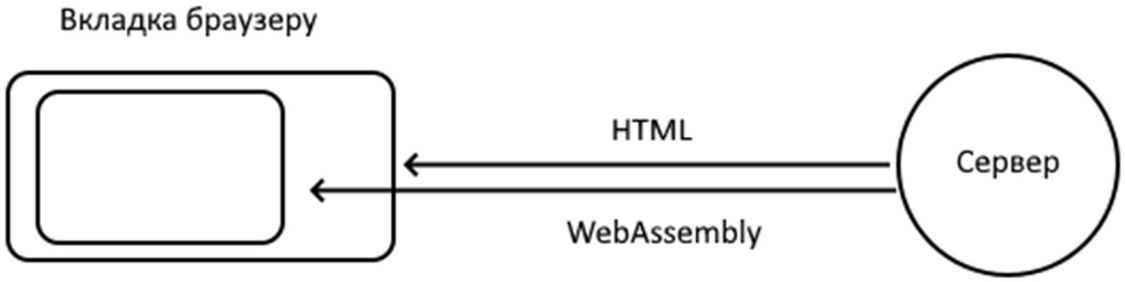

Рисунок 2.6 – Схема розповсюдження WebAssembly

Для зв’язку між сервером та клієнтом є такі варіанти протоколів:

– JSON REST over HTTP;
– JSON/XML RPC over HTTP;
– WebSockets.

Інші варіанти у браузерному середовищі недоступні.

Перші два варіанти не є оптимальними з причини однобічності протоколу HTTP. А за умовами задачі, клієнт може отримувати дані від сервера. Більш того HTTP не підходить для передачі даних у реальному часі.

Тому єдиним варіантом є протокол WebSockets, який побудований поверх HTTP. Цей протокол дозволяє відкрити двобічне постійне мережеве з’єднання між браузером та сервером. Завдяки такій структурі клієнт не має необхідності відправляти запит для отримання події. Також WebSockets підтримує роботу з JSON та бінарними пакетами, що збільшує область використання.

2.3 Розробка серверного середовища

Однією з задач, яка повинен вирішити сервер є створення програми на базі коду, що було надіслано клієнтом та виконання цього коду на роботах Robotino.

Це можна відобразити на рисунку нижче.

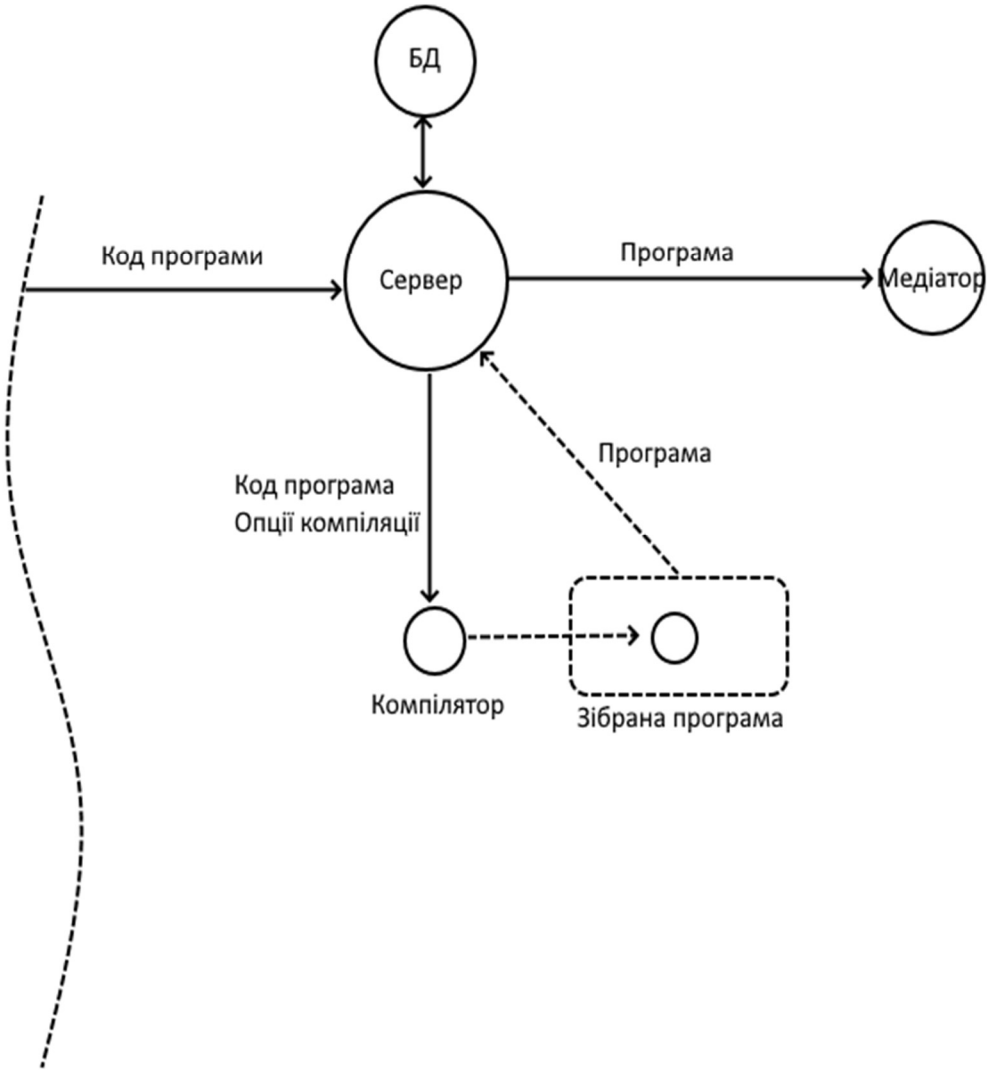

Рисунок 2.7 – Схема компіляції коду програми

Код програми поступає з клієнта. Після чого сервер, використовуючи інформацію з бази даних, приймає рішення для яких медіаторів та систем треба зібрати програму. Прийняв рішення, сервер запускає *N* компіляторів, де *N* – це кількість роботів Robotino у мережі. Останнім кроком є відправлення зібраних програм до медіаторів, які будуть чекати наступних команд.

Це показано схемою на рисунку 2.8.

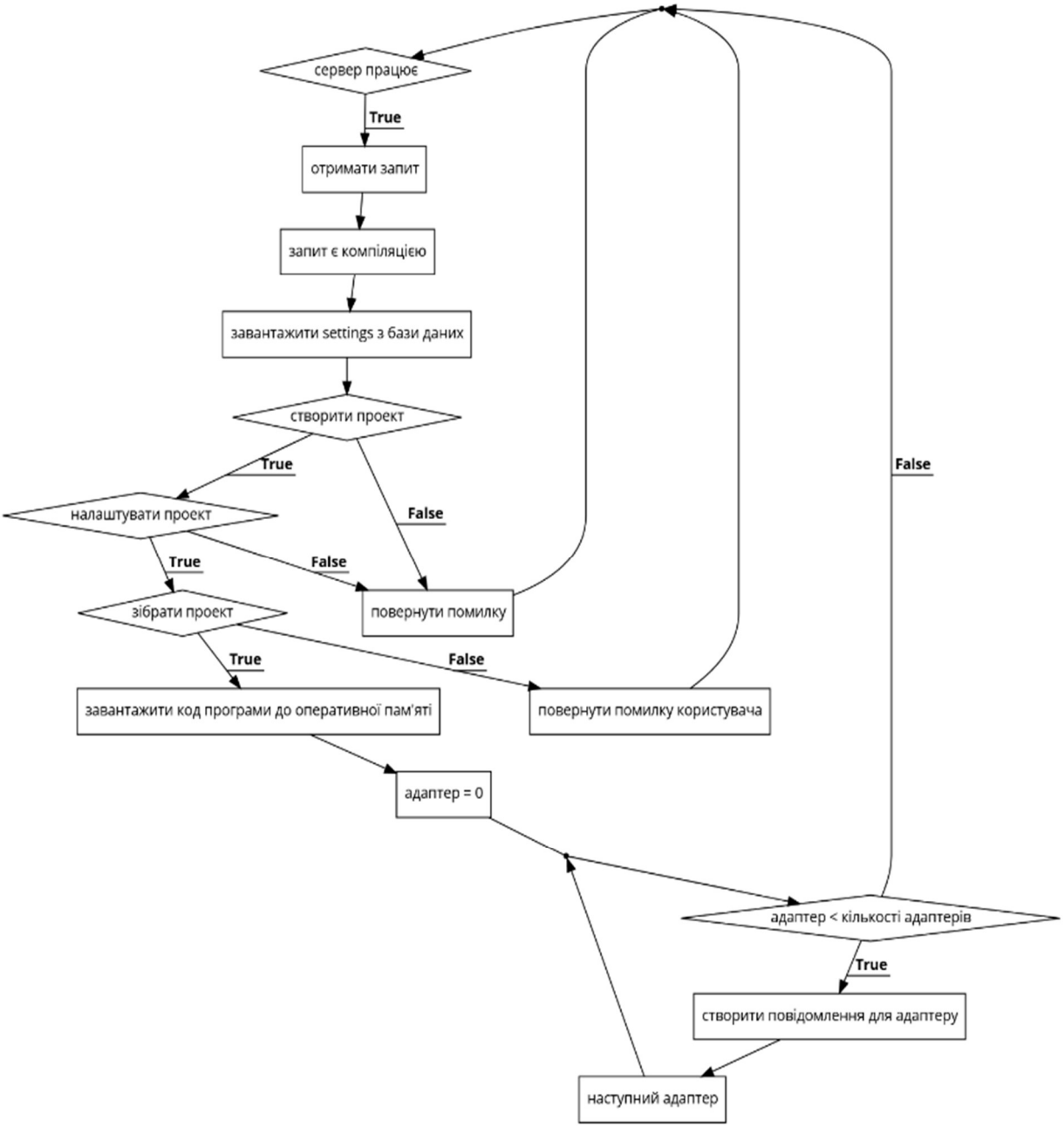

Рисунок 2.8 – Схема процесу компіляції

Процес відправлення повідомлення представлено на рисунку 2.9.

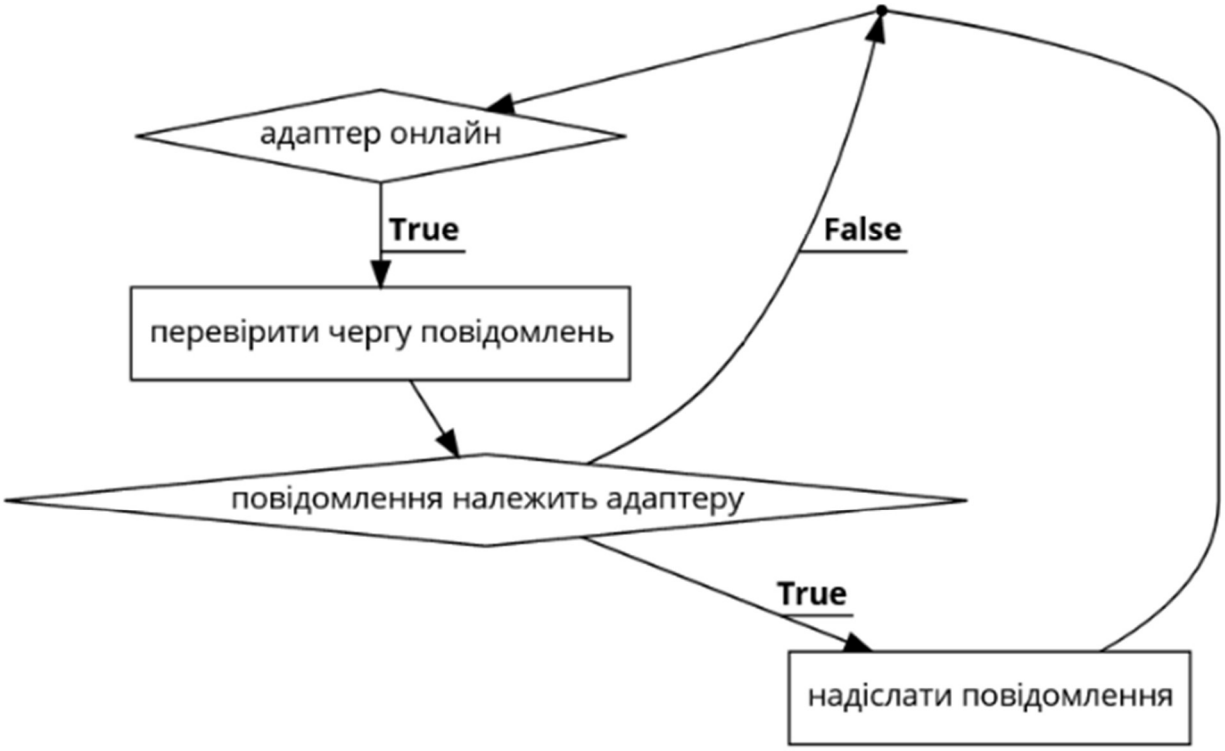

Рисунок 2.9 – Схема процесу надіслання повідомлення

Цей метод дозволяє використовувати такий вид програмування, як метапрограмування [19]. Метапрограмування – це техніка програмування, в якій програма може вважати код інших програм даними. Результатом з цього твердження є можливість змінювати, аналізувати код інших програм. Або створювати новий код, використовуючи код попередніх програм.

Даний підхід вирішує такі проблеми:

– проблему необхідності контролю за циклом життя програми. А саме завершення та перевірку статусу;
– проблему встановлення вхідних даних;
– проблему перевірки значень програми на вимогу користувача.

Розглянемо проблеми окремо.

Проблему необхідності контролю за циклом життя програми можливо вирішити за допомогою додаткового коду, який генерується сервером.

Наприклад, маючи такий псевдокод користувача:

#include <header.h>

```
int main()
        {
        connect(“address”);
        moveAt(0, 0);
}
```

Сервер створює код нижче:

```
#include <internal.h>
#include <header.h>
int main()
        {
        init(“listen address”);
        connect(“address”);
        moveAt(0, 0);
}
```

Незважаючи на невелику зміну в коді, цього достатньо, щоб встановити необхідні обробники та почати слухати команди медіатора. Усі подальші операції проходять асинхронно, не порушуючи код користувача.

Проблема встановлення вхідних даних оминається схожим шляхом.

Для початкового коду:

```
#include <header.h>

#include <const.h>
int main()
        {
        connect(ROBOTINO_ADDRESS);
        moveAt(0, 0);
}
```

Сервер створює файл const.h:

```
#pragma once
#define ROBOTINO_ADDRESS “127.0.0.1:12080”
```

До коду користувача у такому випадку не вноситься жодної зміни.

Проблема перевірки значень програми на вимогу користувача має декілька розв’язків.

Перший має мінімальний вплив на код користувача. Останній має викликати спеціальну функцію, що має схожу сигнатуру до std::ostream. Вказана функції у свою чергу відправляє значення до клієнту, що дозволяє дивитися отримане значення.

Наприклад:

```
#include <header.h>
int main()
        {
        connect(“address”);
        moveAt(0, 0);
        debug() << bumper->isHit();
}
```

Що виведе до клієнтської консолі true або false.

Другий спосіб використовує кодогенерацію. А саме обгорнення типів змінних програми користувача. Надійність цього засобу падає в ростом кількості коду. Це пов’язано ростом кількості складних типів як структури, класи, параметризовані класи та інші.

Тому оптимальним варіантом для цієї проблеми є перший спосіб з покращенням аналізу лог даних на стороні клієнта.

Узагальнюючи рішення вказаних проблем з’являється такий алгоритм генерації програми:

– завантаження коду програми до серверу;
– завантаження списку медіаторів;
– завантаження списку роботів Robotino для кожного медіатору;
– завантаження шаблону проекту для збірки проекту;
– аналіз вхідного коду;
– згенерувати новий код, який є комбінацією вхідного та код ініціалізації;

– створення N проектів, де N – число роботів Robotino;
– для кожного з проектів створити файл з інформацією про цільову машину та робота;
– для кожного з проектів зібрати цей код використовуючи компілятор, на приклад, clang.

Це також відображено схемою на рисунку 2.10.

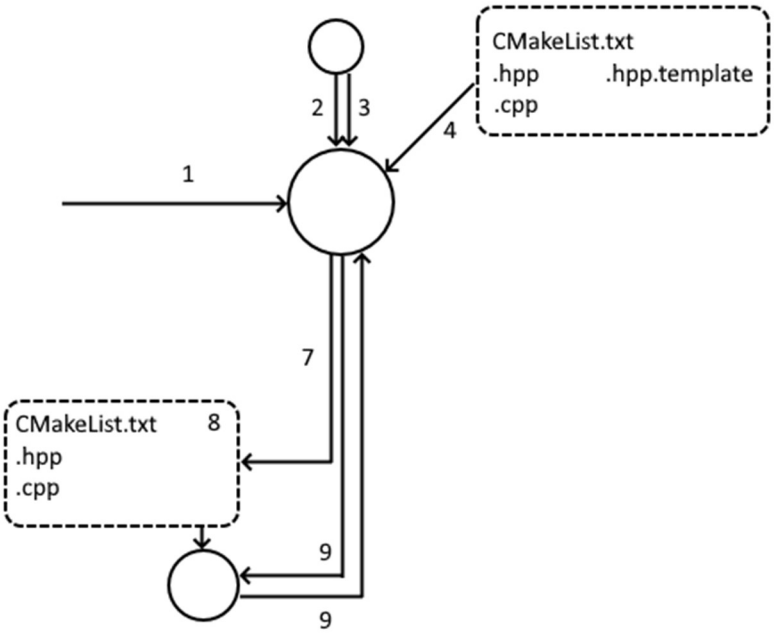

Рисунок 2.10 – Алгоритм генерації програми

2.4 Розробка зберігання даних

Нижче будуть достіджені три найбільш широко використовувані СУБД з відкритим вихідним кодом: SQLite, MySQL і PostgreSQL. Зокрема, будуть розглянуті типи даних, які використовує кожна СУБД, їх переваги та недоліки, а також ситуації, в яких вони найкраще оптимізовані.

MySQL потребує сервер бази даних для взаємодії з клієнтом за допомогою мережі.

MySQL потребує близько до 600 Мб простору для роботи.

MySQL підтримує більшість типів, а саме: TINYINT, SMALLINT,

MEDIUMINT, INT, BIGINT, FLOAT, DOUBLE, DOUBLE PRECISION, REAL, DECIMAL, NUMERIC, DATE, DATETIME, TIMESTAMP, YEAR, CHAR, VARCHAR, TINYBLOB, TINYTEXT, BLOB, TEXT, LONGBLOB, LONGTEXT, ENUM, SET.

MySQL може підтримувати декілька з’єднань одночасно.

У MySQL можливо створити декілька користувачів, які мають унікальні рівні доступу та ролі.

MySQL підтримує масштабування, що дозволяє обробляти великі об’єми даних.

MySQL надає можливість авторизації запиту для блокування доступу невідомим відправником. Авторизація може відбуватися за ім’ям та паролем користувача або за допомогою технології SSH.

Для встановлення MySQL потрібно налаштування декількох конфігураційних файлів..

MySQL є оптимальним у таких ситуаціях, які потребують розподілені операції, високий рівень безпеки, розробку веб-сайтів або інші складні системи. З іншого боку, MySQL не реалізує весь стандарт SQL, що ускладнює міграції між базами даних.

SQLite не потребує серверу та виконується як частина прикладної програми, не маючи можливості підключення до інших програм за допомогою мережі.

SQLite потребує близько до 250 Кб простору.

SQLite підтримує невелику кількість типів: BLOB, NULL, INTEGER, TEXT, REAL.

SQLite може працювати лише з одним з’єднанням.

SQLite не підтримує управління користувачами.

SQLite швидко працює з невеликими об’ємами даних, але швидко уповільнюється зі зростанням кількості даних.

SQLite не підтримує авторизацію. Це дозволяє усім доступ до відкритих даних та можливість змінювати її.

SQLite не потребує конфігураційні файли для своєї роботи.

Тому SQLite є оптимальним рішенням для невеликих програм, програм, що потребують базу даних у альтернативних носіях інформації та тестування. Але не підходить для програм, які потребують декількох користувачів або працюють з великим об’ємом даних на запис.

PostgreSQL є базою даних з відкритим кодом, що має ціль реалізовувати повний стандарт SQL. Порівнюючи з іншими базами даних можливо побачити, що вона має підтримку об’єктно-орієнтованих операцій додатково до реляційних.

PostgreSQL підтримує багато як стандартних, так і нестандартних типів, а саме: BIGINT, BIGSERIAL, BIT, BIT VARYING, BOOLEAN, BOX, BYTEA, CHARACTER, CIDR, CIRCLE, DATE, DOUBLE PRECISION, INET, INTEGER, INTERVAL, LINE, LSEG, MACADDR, MONEY, NUMERIC, PATH, POINT, POLYGON, REAL, SMALLINT, SERIAL, TEXT, TIME (WITH/WITHOUT TIME ZONE), TIMESTAMP (WITH/WITHOUT TIME ZONE), TSQUERY, TSVECTOR, TXID-SNAPSHOT, UUID, XML, JSON [20].

PostgreSQL має велику кількість користувачів, що надають підтримку, зв’язану з базою даних.

PostgreSQL підтримує розширення сторонніми інструментами, що надає можливість управління, проектування з інших додатків.

PostgreSQL легко інтегрується з мовами програмування, що орієнтовані на об’єктну структуру.

PostgreSQL не є оптимальним рішенням у випадку необхідності частих операцій зчитування.

PostgreSQL підходить у випадках потреби надійності структури даних, оскільки він має додаткові примітиви синхронізації та коректності виконання операцій.

PostgreSQL також є оптимальним та єдиним рішенням у разі необхідності складних процедур або дизайну структури.

Маючи вимогу до великої надійності, можна сказати, що PostgreSQL є

найкращим інструментом для зберігання даних, а підтримка реплікації дозволяє масштабувати систему в разі нестачі ресурсів.

Виходячи з опису системи була розроблена наступна структура бази даних, що наведена нижче.

Adapter є таблицею, що зберігає налаштування для конкретного адаптеру для подальшого використання у процесі компіляції або у процесі перегляду значень в програмі клієнту.

Поле id є головним ключем таблиці, що оновлюється автоматично з додаванням нових даних.

robotino_ip та robotino_port необхідні для конфігурації підключення в зібраній програмі.

Для знаходження статусу адаптеру використовується поле online, що має значення TRUE у разі активного та робочого підключення між ним та сервером.

```
CREATE TABLE adapter
(
  id serial NOT NULL,
  robotino_ip text,
  robotino_port bigint,
  online boolean
);
ALTER TABLE adapter ADD CONSTRAINT id
  PRIMARY KEY (id);
```

Таблиця settings зберігає налаштування компіляції програм. Оскільки має бути надана можливість використання різних компіляторів та платформ, є необхідність у таких полях, як toolchain_path, output_name та build_configuration.

toolchain_path зберігає шлях до файлу конфігурації компілятору у CMake. Саме цей файл змінює кінцевий результат компіляції коду.

output_name є важливим, оскільки різні системи мають різний підхід до імені робочого файлу.

build_configuration дозволяє використовувати різний рівень оптимізації для подальшого тестування.

```
CREATE TABLE settings
(
  id serial NOT NULL,
  toolchain_path text,
  output_name text,
  build_configuration text
);
ALTER TABLE settings ADD CONSTRAINT pk_settings
PRIMARY KEY (id);
```

Таблиця user необхідно для перевірку доступу користувача до серверу. Для більшої надійності, прикладний інтерфейс програми має бути недоступним без коректної комбінації імені та паролю.

```
CREATE TABLE user
(
  id serial NOT NULL,
  name text,
  password text
);
ALTER TABLE user ADD CONSTRAINT pk_user
PRIMARY KEY (id);
```

Візуальну структуру бази даних можна побачити на рисунках 2.11, 2.12, 2.13.

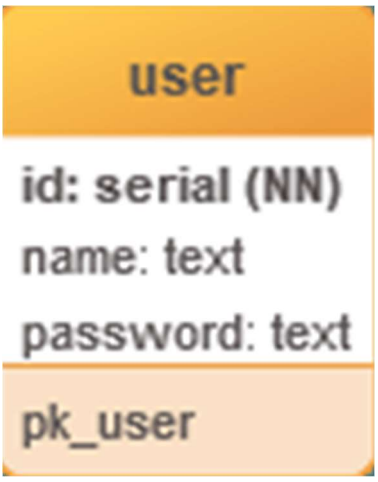

Рисунок 2.11 – Структура таблиці user

Рисунок 2.12 – Структура таблиці adapter

Рисунок 2.13 – Структура таблиці settings

2.5 Розробка взаємодії медіатору та роботу Robotino

Відношення між медіатором та роботом Robotino є один до багатьох. Тобто, внутрішня архітектура серверу медіатора має відрізняти роботів та вміти направляти запити до потрібних зібраних програм. Це відношення відображено на рисунку 2.14.

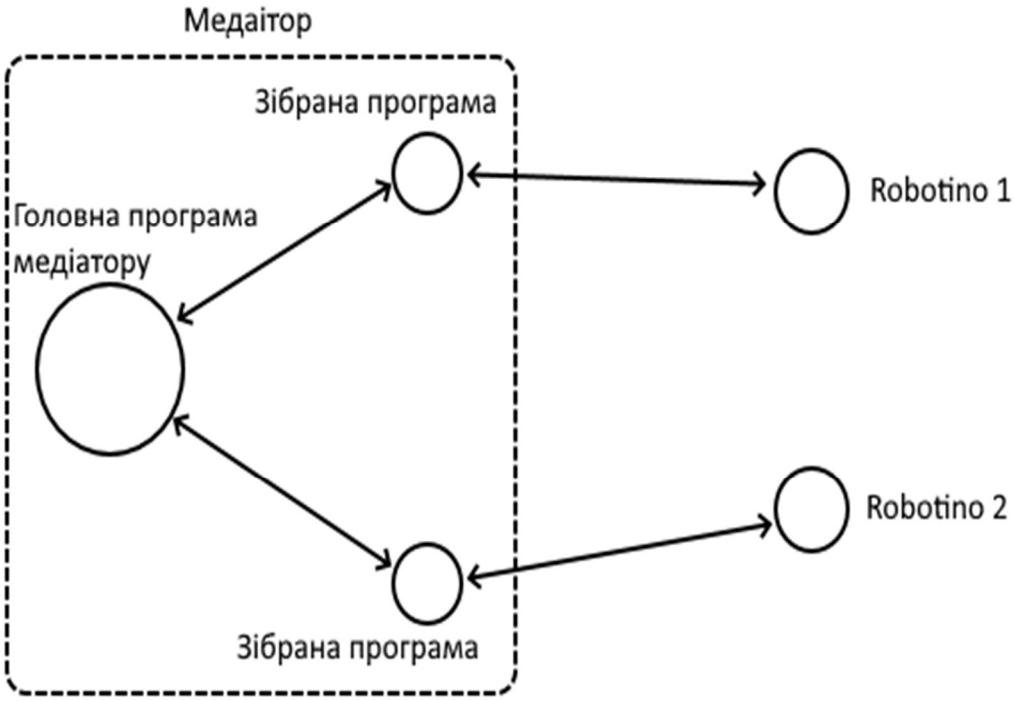

Рисунок 2.14 – Схема взаємодії медіатору з роботами Robotino

Схема процесу обробки запиту від серверу адаптером для компіляції програми клієнту Robotino зображено на рисунку 2.15.

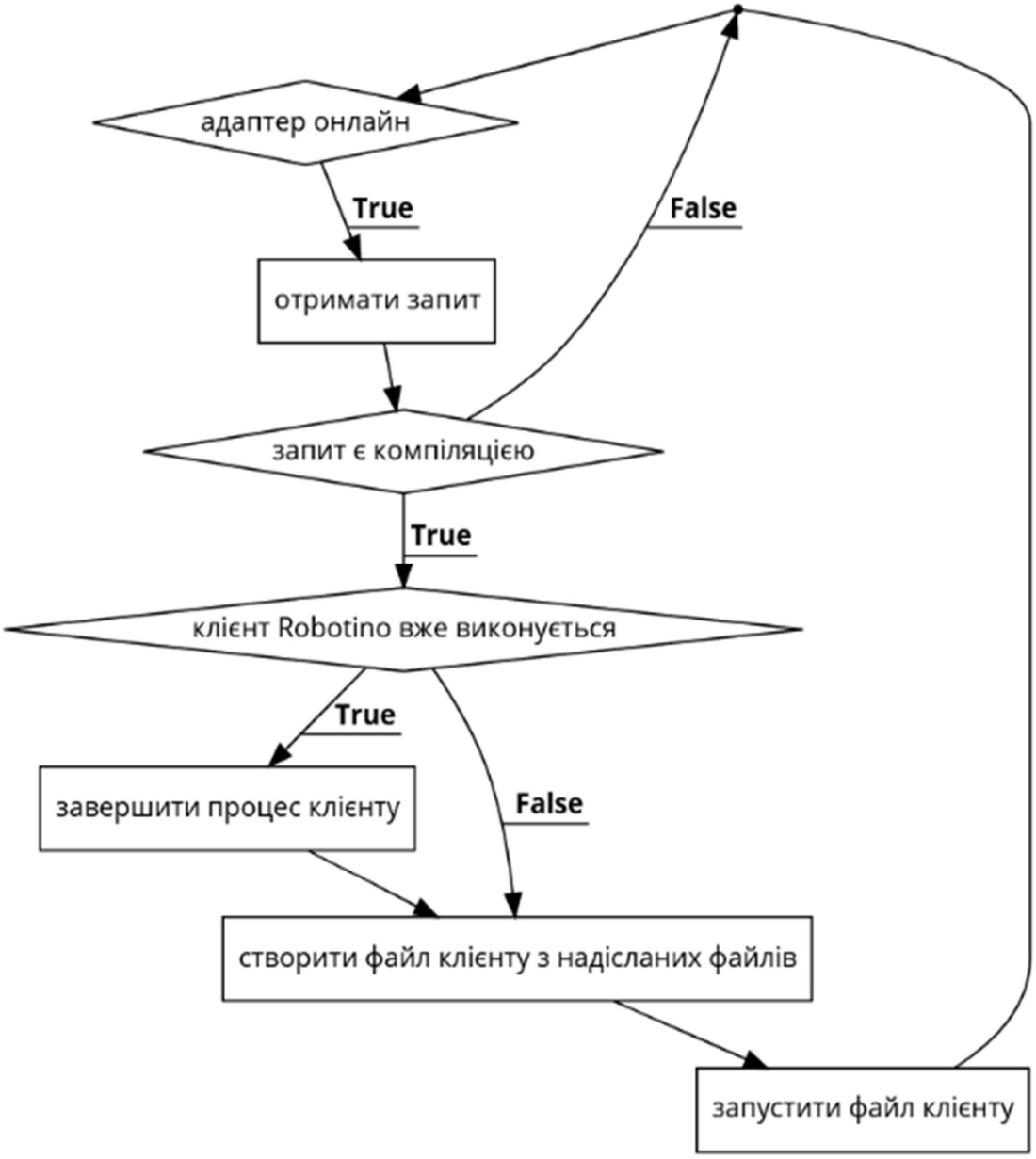

Рисунок 2.15 – Схема процесу обробки запиту адаптеру

Комунікація між усіма вузлами в схемі проходить за протоколом TCP/IP. Robotino API v2 вимагає звичайне TCP з’єднання у зв’язку використанням власного протоколу. Але для виконуваної програми можливі і інші формати. Альтернативою TCP сокетам є WebSockets, які були вибрані для взаємодією з сервером. Одним з аргументів для їх вибору є уніфікація протоколів спілкування частин системи. Зменшуючи різноманіття реалізованих форматів можливо зменшити вірогідність помилку при адаптації одного формату до іншого. Втрата

інформації або її пошкодження можуть трапитися, у випадку принциповій різниці між двома форматами або невідповідності усіх властивостей даних. [21]

Наприклад, у випадку WebSocksets та TCP/IP сокетів перший підтримує групування інформації у пакети довільного розміру, тоді як другий обмежений [22]. З цього випливає необхідність аналізувати довжину потоку для знаходження кордонів пакету на прикладному рівні моделі OSI. Іншим прикладом можна вказати зміну властивостей пакета для диференціації типу даних, як текстова команда або кадр відеопотоку.

Різницю між TCP потоком та WebSocket відображено на рисунках 2.16, 2.17.

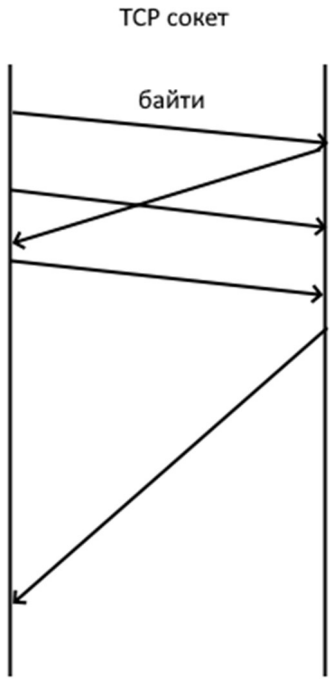

Рисунок 2.16 – Схема обміну інформацією між вузлами TCP сокету

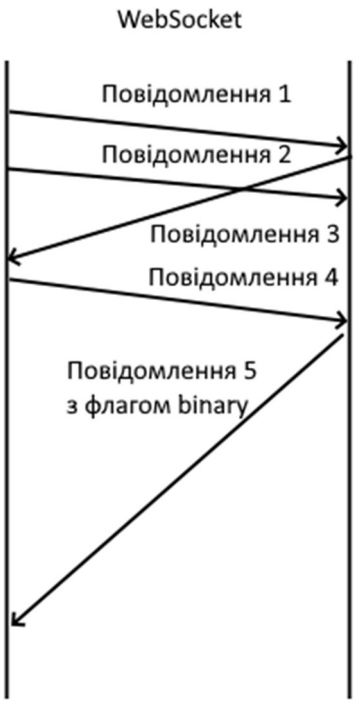

Рисунок 2.17 – Схема обміну інформацією між вузлами WebSocket

2.6 Висновки до другого розділу

У другому розділі були розглянуті можливі варіанти архітектури системи. Зокрема, браузерного клієнта, сервера, адаптера та клієнта Robotino. Було виконано проектування вказаних програм. Також розроблена структура бази даних та взаємодія системи управління базою даних з головним сервером. Використовуючи створені схеми, що описують взаємодію між компонентами системи, буде створено програмне забезпечення у обраних мовах програмування.

Виходячи з завдання було запропоновано використання гібридного зв’язку HTTP та WebSocket. Це має переваги обох технологій, використовуючи простоту та надійність HTTP та універсальність WebSocket.

3 РОЗРОБКА АЛГОРИТМІВ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ КЕРУВАННЯ РОБОТОМ ROBOTINO

3.1 Вибір мови програмування та програмного забезпечення

У зв’язку з тим, що розробка програмного комплексу керування роботом Robotino вимагає використання декількох компонентів у різних середовищах, є необхідність ретельного вибору мови програмування для кожного елемента системи.

Як було вказано бібліотека Robotino API2 є прикладним інтерфейсом програмування для мов C та C++. Використання інших мов можливо, але має свої недоліки та переваги. Для подальшого аналізу проаналізовані такі мови, як C, C++, C#, Java, Python.

Переваги мови програмування C:

– простий програмний інтерфейс доступу до бібліотеки;

– можливість подальшого розширення інших мов за допомогою складових частин мови C;

– легка підтримка доступу до апаратних та системних складових системи;

– використання пам’яті без необхідності додаткової інформації для роботи програми.

Недоліки мови програмування C:

– відсутність підтримки об’єктно-орієнтованого програмування, що ускладнює розробку великих програм;

– помилки виконання можуть привести до падіння програми без можливості відновити робочий стан;

– складний аналіз та пошуки помилок виконання;

– відсутність можливості групування елементів мови у єдиний простір імен;

– відсутність стандартного засобу для роботи з помилками, а саме, генерації помилки та її автоматичної обробки;

– відсутність конструкторів та деструкторів, що ускладнює управління системними ресурсами.

Переваги мови програмування C++:

– легка підтримка доступу до апаратних та системних складових системи;

– можливість використання пам’яті без необхідності додаткової інформації для роботи програми

– об’єктно-орієнтована мова програмування, що робить більш легким створення складних програм;

– підтримка шаблонів для повторного використання коду;

– підтримка мови програмування C;

– наявність стандартного механізму для роботи з помилками.

Недоліки мови програмування C++:

– помилки виконання можуть привести до падіння програми без можливості відновити робочий стан;

– складний аналіз та пошуки помилок виконання.

Переваги мови програмування C#:

– легка інтеграція з сервісами Windows;

– об’єктно-орієнтована мова програмування;

– наявність стандартного механізму для роботи з помилками;

– автоматичне управління пам’яттю програми, що зменшує вірогідність помилок подвійного видалення об’єктів;

– строга перевірка операції конвертації даних.

Недоліки мови програмування C#:

– необхідність додаткових програм для запуску додатку на мові C#;

– взаємодія з апаратним та системним забезпеченням потребує більше кроків, ніж мови C та C++.

Переваги та недоліки мови програмування Java є здебільшого схожими на переваги та недоліки мови C#.

Переваги мови програмування Java:

– компактний код, що виконується без необхідності компіляції;

– наявність стандартного механізму для роботи з помилками;

– швидка розробка програми.

Недоліки мови програмування Java:

– складна інтеграція з іншими мовами програмування з причини відсутності компіляції;

– невелика швидкість виконання;

– помилки коду можна виявити лише у процесі виконання.

Мова програмування C++ є збалансованим рішенням, що має пряму інтеграцію з Robotino API2, підтримує багато функцій мов високого рівня, може працювати з апаратним та системним забезпеченням, є швидким у процесі виконання.

У випадку головного сервера вибором також є мова програмування C++, тому що вона має велику швидкість виконання, наявність бібліотек для реалізації серверної архітектури. Також суттєвим аргументом є необхідність генерації сервером бінарного файлу для взаємодії з роботом Robotino, який використовує мову C++. Це дозволяє використати єдину платформу збірки проектів.

Для реалізації клієнтської програми оптимальним варіантом є програмний комплекс Unity. Він об’єднує переваги мови програмування C# та реалізує виконання у браузерному середовищі. Ще однією перевагою є схожість мов C++ та C#. Це спрощує подальшу інтеграцію та зменшує можливі помилки від різниці ідеологій мов.

3.2 Вибір програмного середовища

У зв’язку з вибором мови C++ для збирання програм необхідні такі додатки:

– CMake;

– Docker;

– Conan.

CMake є програмним забезпеченням для опису та конфігурації проекту [23]. За допомогою цього додатку є можливість створити робочий проект та зібрати його однією командою для різних операційних систем.

Він є стандартним засобом, що широко підтримується багатьма виробниками програмного забезпечення.

Conan є менеджером пакетів. Менеджер пакетів або система управління пакетами являє собою набір програмних засобів, який автоматизує процес установки, оновлення, налаштування і видалення комп'ютерних програм для комп'ютера [24].

Використання пакетів спрощує та пришвидшує процес збірки програм, зменшуючи можливі помилки організації залежностей.

Docker – є програмним забезпеченням для організації створення та контейнерів. Завдяки Docker можливо зменшити вплив програмного середовища та системне середовище та навпаки. Додатковою перевагою є єдина точка налаштування, що запускається єдиною командою [25].

Контейнеризація стає все більш популярною, оскільки контейнери:

– гнучкі: навіть найскладніші програми можна контейнеризувати;

– легкі: контейнери використовують і використовують спільне ядро хоста, роблячи їх набагато ефективнішими з точки зору системних ресурсів, ніж віртуальні машини;

– портативні: можливо створювати локально, розгортати в хмарі та запускати де завгодно;

– слабко зв’язані: контейнери дуже самодостатні та інкапсульовані, що дозволяє замінити або оновити один, не порушуючи роботу інших;

– масштабовані: можливо збільшувати та автоматично розповсюджувати репліки контейнера по центру обробки даних;

– безпечні: контейнери застосовують агресивні обмеження та ізоляцію до процесів без будь-якої конфігурації, необхідної для користувача.

У випадку клієнтської програми використовується мова C# та середовище Unity. Завдяки тому, що Unity автоматично налаштовує залежності, для створення робочої програми необхідно лише мати Unity Editor.

Для серверного середовища, що роздає клієнтську програму потрібно налаштувати файловий сервер. У зв’язку з тим, що роздача файлів не потребує великих навантажень на процесор та пам’ять, а також є необхідність налаштування цього серверу двома доступними виборами є Node.js та Nginx.

Проаналізувавши ці варіанти можна побачити, що Node.js використовує мову програмування JavaScript для налаштування серверу. Це є перевагою, порівнюючи з Nginx, у випадку необхідності подальшого удосконалення особливої поведінки роздачи файлів.

3.3 Побудова середовища проекту

3.3.1 Побудова серверного середовища

У зв’язку з тим, що архітектура серверної частини системи використовує комбінацію таких програмних засобів, як CMake, Conan та Docker, кожен з етапів буде розглянуто окремо.

Головним файлом проекту є CMakeLists.txt, що налаштовує процес створення оптимізованого для конкретної системи робочого файлу. Для подальшого розгляду будуть наведені лише найважливіші команди мови програмування CMake.

Для налаштування імені результуючого файлу та імені проекту для подальшої роботи використана команда project.

```
project(RobotinoServer)
```

Опція CMAKE_EXPORT_COMPILE_COMMANDS є необхідною для поєднання компілятору мови програмування C++ зі стороннім програмним забезпеченням. Це дозволяє знайти використану програму компілятор, опції компіляції та шляхи до системних ресурсів.

set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

Опція CMAKE_CXX_STANDARD змушує програму компілятор використати версію 17 мови програмування C++. З причини використання сервером файлової системи для створення програми-посередника робота Robotino, модуль std::filesystem є необхідним.

set(CMAKE_CXX_STANDARD 17)

Опції CMAKE_ARCHIVE_OUTPUT_DIRECTORY, CMAKE_LIBRARY_OUTPUT_DIRECTORY та CMAKE_RUNTIME_OUTPUT_DIRECTORY є необхідними для подальшого налаштування автоматичної збірки та запуску. Вони дозволяють зафіксувати директорію, у якій буде зібрано результуючий файл. Це у комбінації з опцією PROJECT_NAME, що була створена завдяки команді project дає можливість знайти абсолютний шлях до вихідного файлу.

```
set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY
${CMAKE_BINARY_DIR}/lib)
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY
${CMAKE_BINARY_DIR}/lib)
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY
${CMAKE_BINARY_DIR}/bin)
```

Менеджер пакетів Conan був використаний у наступному фрагменту для підключення залежностей, що були проаналізовані раніше.

Розглянемо кожну з залежностей окрему.

Boost – це бібліотека широкого використання. Найважливішою особливістю якої є модульність. Для серверного проекту був використаний модуль boost.asio, що є інтерфейсом доступу до системного мережевого API.

Restinio – це бібліотека, що реалізує стандарти HTTP та WebSocket для серверного середовища.

nlohmann_json – бібліотека, яка реалізує стандарт JSON.

Spdlog – бібліотека організації вихідної інформації у реальному часі.

Fmt – бібліотека формування інформації, використовуючи вхідні дані.

```
conan_cmake_run(
  REQUIRES
    boost/1.74.0
    restinio/0.6.12@stiffstream/stable
    nlohmann_json/3.9.1
    spdlog/1.8.1
    fmt/7.1.2
  BASIC_SETUP
  BUILD missing
  )
```

Для формування вихідного файлу застосовується команда add_executable. PROJECT_NAME є опцією, що створюється автоматично. Конфігурація цієї команди відбувається за допомогою опцій CMake. Створення опцій HEADERS, SOURCES не було відображено у зв’язку з їх меншою важливістю.

```
add_executable(${PROJECT_NAME}
  ${HEADERS}
  ${SOURCES}
  )
```

Команда target_include_directories налаштовує шляхи підключення файлів з інформацією про структуру залежностей або проекту. У даному випадку був використаний модифікатор PRIVATE, що дає доступ для цих шляхів лише цьому проекту. Це має сенс у зв’язку з тим, що сервер є єдиним проектом у даному рішенні.

```
target_include_directories(${PROJECT_NAME}
  PRIVATE
    include
  )
```

Команда target_link_libraries схожа на команду target_include_directories. Головною різницею є те, що target_link_libraries налаштовує та підключає

реалізацію залежностей проекту. У даному випадку це CONAN_LIBS, що створена менеджером пакетів Conan та stdc++fs або std C++ filesystem.

target_link_libraries(${PROJECT_NAME} ${CONAN_LIBS} -lstdc++fs)

Для запуску конфігурації та компіляції серверу потрібно описати цей процес у файлі Dockerfile, опис якого наведений нижче.

Команда FROM дозволяє вказати базове зображення для контейнеру. У даному випадку, у зв'язку з тим, що використовується мова програмування C++, обране середовище gcc. gcc:latest – це середовище операційної системи Linux зі встановленим компілятором мов C та C++.

FROM gcc:latest as robotinoserverenvironment

Команда RUN виконує інші програми, використовуючи оболонку shell. У випадку нижче встановлюються CMake, Conan (та мова програмування Python як залежність), крос-компілятор Mingw. Mingw використовується самим сервером для побудови клієнта до робота Robitono, який може виконуватись на іншій від Linux операційній системі.

RUN apt-get update && apt-get -y install cmake python3 python3-pip mingw-w64

sudo update-alternatives --config x86_64-w64-mingw32-g++

sudo update-alternatives --config x86_64-w64-mingw32-gcc

RUN pip3 install conan

Друга команда FROM дає можливість розділити будівництво на частини. У цьому випадку першою частиною є завантаження базового середовища та залежностей. Другою же частиною є будівництво серверу з наявного коду.

FROM robotinoserverenvironment AS robotinoserverbuilder

Команда COPY виконує копіювання елемента файлової системи з першого шляху до другого. Крапка позначає директорію, з якою на цей час працює Docker.

COPY . /usr/src/RobotinoServer

Команда WORKDIR змінює робочу директорію, з якою потім будуть працювати наступні команди.

WORKDIR /usr/build/RobotinoServer

Починаючи з рядку нижче виконується робота с проектом CMake, який був описаний раніше.

Першою командою є виклик створення проекту для конкретного компілятора за інформацією з CMakeLists.txt за шляхом першого аргументу команди. Опція CMAKE_BUILD_TYPE вказує на необхідність увімкненої оптимізації.

Другою командою є запуск компілятора за вже створеним проектом. Як і раніше, крапка позначає директорію, з якою працює зараз Docker.

RUN cmake '/usr/src/RobotinoServer' -DCMAKE_BUILD_TYPE=Release

RUN cmake --build .

Команда CMD виконує запуск зібраного серверу та очікує його завершення.

CMD ["./bin/RobotinoServer"]

LABEL є командою для вказання інформації. Це необхідно для контролю зібраним контейнером сервісом Docker.

LABEL Name=RobotinoServer Version=0.0.1

3.3.2 Побудова середовища клієнту Robotino

Клієнт Robotino за складом середовища є подібним до сервера. Але існують деякі важливі винятки, пов’язані з кросс-компіляцією та автоматичним збиранням, які будуть вказані нижче.

Першою різницею є файл CMakeLists.txt, який має додаткові команди.

Команда find_package використовує CMake для пошуку бібліотеки за іменем. Для пошуку бібліотеки конфігуратор CMake має мати доступ до файла, який виконує пошук та підключення вказаної бібліотеки. У випадку Robotino API цей файл поставляється з самою бібліотекою. Тому лише необхідно вказати шлях до директорії, у якої знаходиться цей файл у опції CMAKE_MODULE_PATH.

SET(ROBOTINOAPI2_DIR

"${CMAKE_SOURCE_DIR}/dependencies/RECGmbH/robotino/api2")

```
SET( CMAKE_MODULE_PATH "${CMAKE_MODULE_PATH}"
"${ROBOTINOAPI2_DIR}/cmake" )
find_package( RobotinoAPI2 REQUIRED )
```

Другою різницею є більш ширший список залежностей програми. Бібліотеки mingw32, libgcc, -static-libstdc++, pthread дозволяють зібрати програму на операційній системі Linux для операційної системи Windows додавши функції, які необхідні для роботі на останній.

```
target_link_libraries(${PROJECT_NAME} mingw32 -lrec_robotino_api2 -
static-libgcc -static-libstdc++ -Wl,-Bstatic -lstdc++ -lpthread )
```

Однією з проблем побудови цього клієнта є відсутність бібліотеки Robotino API для кросс-компілятора mingw. У ході аналізу був розроблений процес повторної побудови бінарних файлів бібліотеки у форматі MSVC library до gcc/mingw library. Це потребує наявність інструмента Google lib2a. lib2a аналізує вхідні файли у форматі MS Visual C linker library та створює нові у форматі MinGW linker library [26].

Другою проблемою є неможливість кросс-компіляції у програмі CMake без додаткового налаштування. CMake використовує набір утиліт для компіляції, зв’язування бібліотек та створення архівів та інших завдань для керування збіркою. Доступні набори інструментів визначаються увімкненими мовами. У звичайних збірках CMake автоматично визначає Toolchain для збірки хостів на основі самоаналізу системи та за замовчуванням. У сценаріях кросс-компіляції Toolchain файл може бути вказаний вручну [27].

Для цього був створений файл Toolchain-cross-mingw32-linux.cmake який містить перелік команд та опцій, які будуть проаналізовані нижче.

Опція CMAKE_SYSTEM_NAME вказує ім’я системи для якої виконується побудова додатку.

```
SET(CMAKE_SYSTEM_NAME Windows)
```

Опція COMPILER_PREFIX зберігає префікс імені компілятора, що виконує побудову проекту. Головною різницею між опціями нижче є архітектура

для якої відбувається збирання програми. Варіант i686 позначає архітектуру x86 та x86_64 x86/64 відповідно.

```
#set(COMPILER_PREFIX "i686-w64-mingw32")
set(COMPILER_PREFIX "x86_64-w64-mingw32")
```

Команда find_program у загальному значенні виконує пошук програми за ім’ям. Для робочого Toolchain необхідно мати компілятори мов C, C++ та компілятор ресурсів. Всі вони поставляються з пакетом mingw.

```
find_program(CMAKE_RC_COMPILER NAMES ${COMPILER_PREFIX}-windres)
find_program(CMAKE_C_COMPILER NAMES ${COMPILER_PREFIX}-gcc)
find_program(CMAKE_CXX_COMPILER NAMES ${COMPILER_PREFIX}-g++)
```

Опція CMAKE_FIND_ROOT_PATH_MODE_PROGRAM має бути вимкнена, а CMAKE_FIND_ROOT_PATH_MODE_LIBRARY, CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ввімкнені у зв’язку зі змішаним середовищем компілятора. Частина цього середовища є файлами, які лише мають сенс при використанні в операційній системі Windows, тому вони потребують відокремлення значенням ONLY. Значення NEVER позначає необхідність використання Linux програм у процесі збирання Windows проекту.

```
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
```

3.3.3 Побудова середовища браузерного клієнту

У зв’язку з використанням програми Unity Editor для побудови клієнта, більша частина процесу збирання є інкапсульованою та недоступною для модифікацій.

Додатково, браузерний клієнт потребує файловий сервер для відправки файлів за запитом. Для цього використовується Node.JS.

Метод http.createServer створює HTTP сервер та надає доступ для обробки одного запиту в одному моменті.

```
http.createServer(function (request, response) {
```

Для коректного запуску програми у браузері необхідно позначити заголовок Content-Type, що є підказкою для програми-користувача.

Найбільш важливим випадком є тип файлу .wasm. Для неправильно налаштованого серверу браузер не зможе виконати повторну компіляцію та оптимізацію цього коду.

```
case '.wasm':
return 'application/wasm';
break;
default:
eturn 'application/octet-stream'
break;
```

Іншою проблемою є формати компресії. Вони можуть бути використані для будь-якого типу файла. Це потребує наявність другого заголовку – Content-Encoding. У випадку коректного налаштування браузер має можливість автоматично декодувати цей файл, що значно прискорює процес запуску у порівнянні з декодуванням в JavaScript. [28]

```
switch (extname) {
case '.gz':
var extbasename = path.parse(basename).ext;
contentType = extToContentType(extbasename)
contentEncoding = "gzip";
break;
default:
contentType = extToContentType(extname)
}
```

Найголовнішою частиною цього серверу є завантаження файлів що відбувається методом fs.readFile.

```
fs.readFile(filePath, function(error, content) {
response.writeHead(200, { 'Content-Type': contentType, 'Content-Encoding': contentEncoding });
response.end(content, 'utf-8');
});

}).listen(8325);
```

3.3.4 Побудова середовища адаптера

Конфігурація та побудова адаптеру відбувається за подібним до серверу процесом. Головною відмінністю є відсутність етапу для роботи з Docker.

Побудова проекту виконується прямим викликом програми CMake для платформи MSVC.

3.4 Побудова сервера

Головний сервер складається з декількох частин. Побудова кожної частини буде розглянута окремо.

Точкою старту для першої частини – HTTP серверу є функція restinio::run.

HttpTraits є типом для опису обробника вхідних запитів. Завдяки зображеному методу поведінка серверу фіксується у момент компіляції, що зменшує навантаження при запуску сервера.

```
struct HttpTraits : public restinio::default_single_thread_traits_t {
   using request_handler_t = restinio::router::express_router_t<>;
};
```

Тип restinio::router є маршрутизатором запитів. Він аналізує вхідну інформацію та надіслає її до необхідного обробника або повертає помилку. Деталі будуть розглянуті при описі обробника запиту.

```
restinio::router::express_router_t<> router;
http::handlers::build::createRouter(router);
```

restinio::run виконує запуск сервера з вказаними у HttpTraits параметрами, портом та мережевою адресою для прослуховування.

```
restinio::run(
   restinio::on_thread_pool<HttpTraits>(8)
   .port(HttpServerPort)
   .address(ServerAddress)
   .request_handler(std::move(router)));
```

Кожен з вхідних запитів потрапляє на один із обробників, що підключаються до маршрутизатора. Далі буде продемонстровано один з обробників для побудови клієнта Robotino для конкретного робота.

Метод http_post позначає, що обробник використовує HTTP метод POST для виконання задачі з вхідними даними. Першим аргументом є відносний шлях доступу до обробника. Цей аргумент може використовувати складні іменовані параметри. Другим аргументом є обробник конкретного запиту.

```
router.http_post(
   R"(/build)",
   [](request_handle_t req, router::route_params_t params) ->
request_handling_status_t
   {
      try
      {
```

Вхідна інформація надсилається у форматі JSON в тілі HTTP запиту. Це дозволяє надсилати великі запити, що необхідно у випадку пересилання коду клієнту Robotino.

```
auto data = json::parse(req->body());
```

Оскільки запит може бути поданий багато разів, необхідно кожен раз створювати новий проект для конфігурації. Це мінімізує вірогідність пошкодження проекту або колізії значень. У зв’язки з чим процес компіляції проходить за випадковим шляхом у директорії /tmp.

```
auto buildPath = path("/tmp") / RandomString();
```

```
create_directories(buildPath);
```

Наступним важливим кроком є створення текстової репрезентації проекту, використовуючи вхідну інформацію від користувача. Деталі структури проекту клієнта Robotino будуть розглянуті пізніше.

```
copy(path(TemplateDirectoryPath), buildPath, copy_options::recursive);
std::ofstream appCpp(buildPath / "app.cpp");
appCpp << "#include \"client.hpp\"\n" << data["source"].get<std::string>();
appCpp.close();
```

Після створення текстової версії проекту необхідно створити проект для компілятору mingw. Для цього необхідно викликати CMake, передавши Toolchain файл, що вже був розглянутий.

```
auto result = system(fmt::format("cd \"{}\" && cmake -DCMAKE_TOOLCHAIN_FILE=~/../Toolchain-cross-mingw32-linux.cmake .", buildPath.string()).c_str());
```

Будь яка помилка конфігураціє є внутрішньою помилкою сервера та має призводити до припинення всього процесу компіляції.

```
if (result != 0)
{
   spdlog::error("Configure error: {}", result);
   return req->create_response(status_internal_server_error()).done();
}
spdlog::debug("Building the project...");
```

Останнім кроком створення прикладної програми є запуск компілятора. Цей крок не потребує додаткових опцій з причини їх встановлення на попередньому кроку.

```
result = system(fmt::format("cd \"{}\" && cmake --build .", buildPath.string()).c_str());
if (result != 0)
{
```

Помилки компіляції є помилками користувача, оскільки сервер не може контролювати вхідний код.

```
    spdlog::error("Build error: {}", result);
    return req->create_response(status_bad_request()).done();
}
```

Якщо весь процес закінчився з успіхом, то отриманий файл необхідно завантажити до оперативної пам’яті. Це можливо завдяки налаштованому CMakeLists.txt, в якому вказано ім’я та шлях до очікуваного файлу.

```
std::ifstream ifs(fmt::format("{}/bin/RoboClient.exe", buildPath.string()), std::ios::in | std::ios::binary);
std::vector<uint8_t> exeContent((std::istreambuf_iterator<char>(ifs)), std::istreambuf_iterator<char>());
```

Оскільки обробка HTTP запиту відбувається у окремому потоці, неможливо викликати адаптер з поточного методу. Тому отримана команда надсилається до іншої підсистеми серверу та пересилається до адаптеру за можливістю.

Також, отриманий виконуємий файл не є текстовим, тому для рішення цієї проблеми було реалізовано метод base64_encode, що імплементацією стандарта Base64.

```
adapterpoll::AddMessage("", json{ {"method", "run"}, {"exe", base64_encode(exeContent.data(), exeContent.size()) } }.dump());

return req->create_response()
.set_body(json::object().dump(2))
.done();
}
catch (const std::exception& e)
{
std::cout << "Unexpected build error: " << e.what() << std::endl;
return req->create_response(status_internal_server_error()).done();
```

```
}
});
```

У випадку отримання невідомого методу потрібно повернути помилку.

```
router.non_matched_request_handler(
[](auto req) {
return req->create_response(status_not_found()).connection_close().done();
});
}
```

Робота з WebSocket відбувається за схожим алгоритмом. Але базова архітектура реалізована іншою архітектурою, яка буде розглянута нижче.

Клас WsServer виконує мережеву комунікацію, встановлення та видалення підключень. А також обробку та надіслання запитів через сокет.

Метод Listen є блокуючим, тобто доки сервер працює, цей метод не повертає управління до попередньої функції.

```
void WsServer::Listen()
{
```

Клас io_context є класом, що містить механізми синхронізації потоків. Він є необхідним у випадку роботи та розподілення обов'язків між потоками.

```
boost::asio::io_context ioc{ 1 };
```

Клас acceptor прослуховає вказану комбінацію мережевої адреси та порту та відкриває нові сокети для обробки вхідних запитів.

```
boost::asio::ip::tcp::acceptor acceptor{ ioc,
{boost::asio::ip::make_address(address_), port_} };
  for (;;)
  {
    boost::asio::ip::tcp::socket socket{ ioc };

    acceptor.accept(socket);
```

У випадку наявності нового підключення необхідно виділити ресурси для незалежного виконання сокету та повернення управління для обробки подальших нових підключень.

```
std::thread{ std::bind(
    &WsServer::DoSession,
     this,
     std::move(socket)) }.detach();
  }
}
```

Метод DoSession є головним методом з'єднання. Перед початком обміну інформації необхідно перевірити коректність цього з'єднання.

```
void WsServer::DoSession(boost::asio::ip::tcp::socket& socket)
{
  try
  {
     boost::beast::websocket::stream<boost::asio::ip::tcp::socket>  ws{
std::move(socket) };
```

Метод повинен продовжити тільки лише у випадку, якщо запит був сформований як WebSocket.

```
ws.accept();
```

Та якщо перевірки пройдена сервер мусить підтримувати з'єднання доки інша сторона не завершить його сама.

```
for (;;)
{
```

Вхідні дані можуть поступати декількома пакетами для зменшення мережевої загрузки. Але для обробки повідомлення необхідно з'єднати ці пакети. Тому для накопичення пакетів був обраний буфер multi_buffer, що має ефективно зростати у розмірах та метод read, що очікує у потоку з'єднання кінцевого пакету.

```
boost::beast::multi_buffer buffer;
```

```
ws.read(buffer);
```

Текстові повідомлення очікуються у форматі JSON. Саме цей формат використовується у HTTP сервері, що полегшує інтеграцію між підсистемами.

```
if (ws.got_text())
{
auto json = nlohmann::json::parse(boost::beast::buffers_to_string(buffer.data()));
```

Єдине необхідне поле у надісланому JSON об’єкту є ім’я методу. Саме це дозволяє вирішати, куди потрібно перенаправити повідомлення.

```
const auto method = json["method"];
if (handlers_.count(method) == 0)
{
        spdlog::error("Unknown websocket method '{}'", method);
        continue;
}
handlers_[method]->Process(ws, json);
}
ws.text(ws.got_text());
ws.write(buffer.data());
}
}
catch (boost::system::system_error const& se)
{
if (se.code() != boost::beast::websocket::error::closed)
spdlog::error("Websocket error: {}", se.code().message());

catch (std::exception const& e)
{
spdlog::error("Websocket unknown error: {}", e.what());
}
```

3.5 Побудова адаптера

Як і у випадку з сервером io_context є класом, який забезпечує багатопотокову та асинхронну взаємодію.

net::io_context ioc;

Оскільки для здійснення з’єднання необхідно ідентифікувати видалений сервер за наявною адресою, додатковим кроком є виклик методу resolve класу resolver. Цей клас знаходить IP адресу для прямого з’єднання з вказаним пристроєм.

```
tcp::resolver resolver(ioc);
beast::tcp_stream stream(ioc);
auto const results = resolver.resolve(host, port);
```

Оскільки адаптер має буди постійно на зв’язку з сервером, то у випадку пошкодженого або завершеного з’єднання програма має надіслати повторне з’єднання та відновити обмін повідомленнями.

```
while (true)
{
   stream.connect(results);

   http::request<http::string_body> req(http::verb::get, "/adapterpoll", 11);
   req.set(http::field::host, host);
   req.set(http::field::user_agent, BOOST_BEAST_VERSION_STRING);
   http::write(stream, req);
   beast::flat_buffer buffer;
   http::response<http::dynamic_body> res;
   boost::system::error_code ec;
   http::read(stream, buffer, res, ec);
```

У випадку помилки зчитування можна вважати сокет у непрацюючому стану та відновити його. Єдиною помилкою, що є допустимою можна вказати

закінчення потоку. Вказана помилка вказує лише на неможливість подальшої роботи з сокетом, але повертає інформацію для подальшої роботи з нею.

```
if (ec && ec != http::error::end_of_stream)
{
   std::cout << "Error: " << ec << ". Reconnecting..." << std::endl;
   continue;
}
std::string body = boost::beast::buffers_to_string(res.body().data());
if (body.empty())
{
   std::cout << "Empty payload. Reconnecting..." << std::endl;
   continue;;
}

auto json = nlohmann::json::parse(body);
if (json["method"] == "run")
{
```

Для запуску виконуємого файлу клієнта Robotino потрібно провести процес, зворотній серверному. Тобто, декодувати формат Base64 до зчитаної в оперативну пам’ять програми та записати програму на жорсткий диск.

Останнім шагом є запуск вказаної програми через операційну систему.

```
uto exe = base64_decode(json["exe"].get<std::string>());
std::ofstream ofs("roboclient.exe", std::ios::out | std::ios::binary);
ofs.write(exe.data(), exe.size());
ofs.close();
std::cout << "Launching exe." << std::endl;
system("roboclient.exe");
}
```

3.6 Побудова клієнта Robotino

У зв’язку з тим, що більша частина коду невідома на момент компіляції, але програма потребує початкової ініціалізації, програмний код був розділений на декілька частин.

Точкою входу для коду користувача є функція RobotinoMain. Вона викликається з функції main, що компонується з головного файлу, який буде розглянуто нижче.

```
int main()
{
   std::cout << "Starting Robotino Client..." << std::endl;
```

Оскільки паралельно з кодом користувача необхідно налаштувати додаткове з’єднання для взаємодії з клієнтом був створений який не блокуватиме головний потік, але триматиме програму від завершення.

```
auto thread = std::async(std::launch::async, InitConnection);
```

Наявність блоку управління винятковими ситуаціями є необхідною. Код користувача є довільним, тому будь-яка помилка є можливою. Але не всі помилки необхідно оброблювати, оскільки це може призвести до пошкодження пам’яті. Тому було зроблено висновок про необхідність обробки лише виняткових ситуацій типу exception. Інші типи та системні виняткові ситуації мають призводити до падіння з генерацією відладочної інформації.

```
   try
   {
      RobotinoMain();
   }
   catch (const std::exception& e)
   {
      std::cout << "Client exception: " << e.what() << std::endl;
      return 1;
   }
```

```
    return 0;
}
```

3.7 Тестування розробленого програмного забезпечення

Для перевірки розробленої системи була створена програма, що керує рухом робота Robotino. Далі буде розглянено найважливіші елементи цієї програми.

Програма не потребує підключення бібліотек Robotino API2 з причини їх автоматичного встановлення.

```
int _run = 1;
```

ComId, OmniDriveId, BumperId є унікальними значеннями, що вказують на екземпляри підключення, двигуна та датчику зіткнення.

```
ComId com;
OmniDriveId omniDrive;
BumperId bumper;
```

Функція rotate виконує обертання вектору in (x; y) у двомірному просторі на деяке значення deg.Результатом цієї операції є інший вектор out.

```
void rotate( const float* in, float* out, float deg )
{
        const float pi = 3.14159265358979f;

        float rad = 2 * pi / 360.0f * deg;

        out[0] = (float)( cos( rad ) * in[0] - sin( rad ) * in[1] );
        out[1] = (float)( sin( rad ) * in[0] + cos( rad ) * in[1] );
}
```

Функція drive керує рухом роботу доки екземпляр підключення є дійсним. Рух відбувається за колом. У тестовій програмі вектор напряму виконує повний оберт за 10 секунд.

```
void drive()
{
	const float startVector[2] = {0.2f, 0.0f};
	float dir[2];
	float a = 0.0f;
	unsigned int msecsElapsed = 0;
	while( Com_isConnected( com ) && _run )
	{
		rotate( startVector, dir, a );
		a = 360.0f * msecsElapsed / 10000;

		OmniDrive_setVelocity( omniDrive, dir[0], dir[1], 0 );

		msleep( 50 );
		msecsElapsed += 50;
	}
}
```

Функція RobotinoMain є точкою входу для цієї програми. Точка входу main використовується автоматично створеним файлом на сервері. ROBOTINO_IP є значенням, що передається до цієї підпрограми у процесі компіляції. Це дозволяє використовувати код як шаблон, що спрощує масове оновлення.

```
void RobotinoMain()
{
	com = Com_construct();

	Com_setAddress( com, ROBOTINO_IP );
```

```
    if(Com_connect( com ) == FALSE )
    {
        error( "Error on connect" );
    }
    else
    {
        char addressBuffer[256];
        Com_address( com, addressBuffer, 256 );
        printf( "Connected to %s\n", addressBuffer );
    }

    omniDrive = OmniDrive_construct();
    OmniDrive_setComId( omniDrive, com );

    bumper = Bumper_construct();
    Bumper_setComId( bumper, com );

    drive();

    OmniDrive_destroy( omniDrive );
    Bumper_destroy( bumper );
    Com_destroy( com );
}
```

Результати роботи цієї програми можна побачити на рисунках 3.1 та 3.2

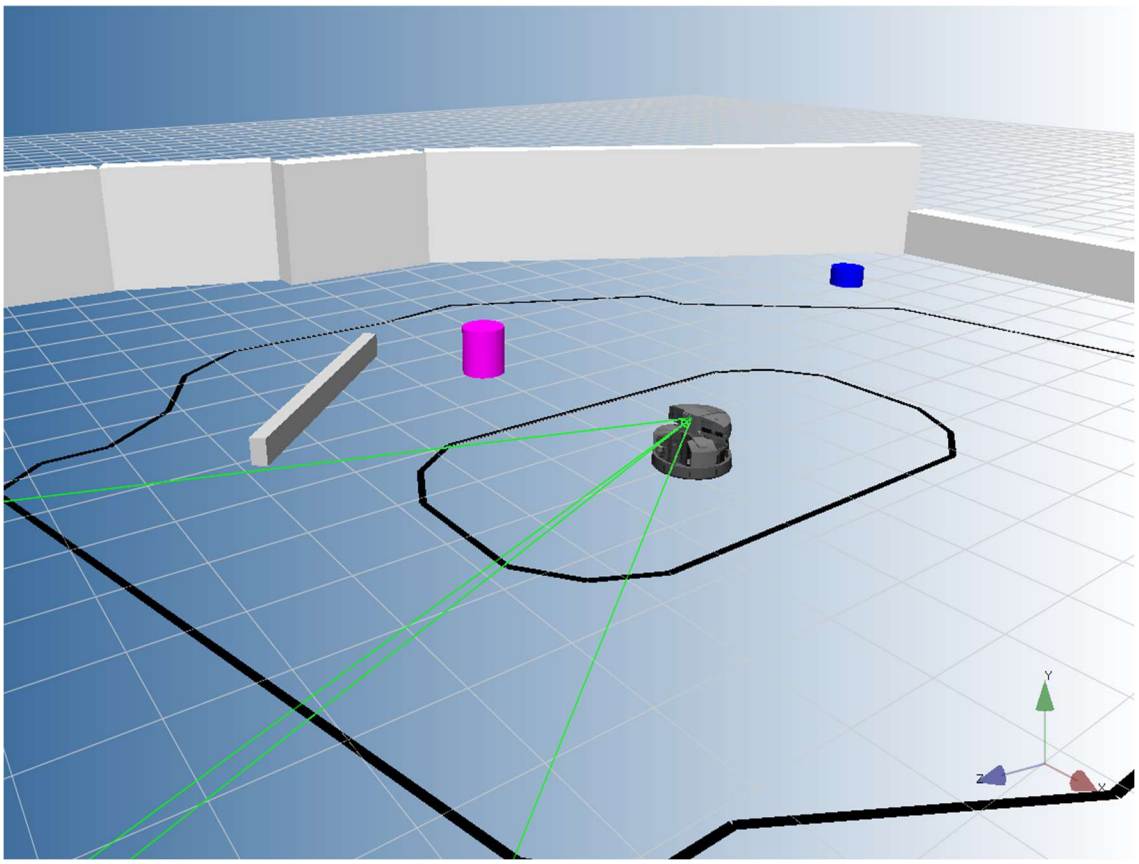

Рисунок 3.1 – Результат роботи програми після запуску

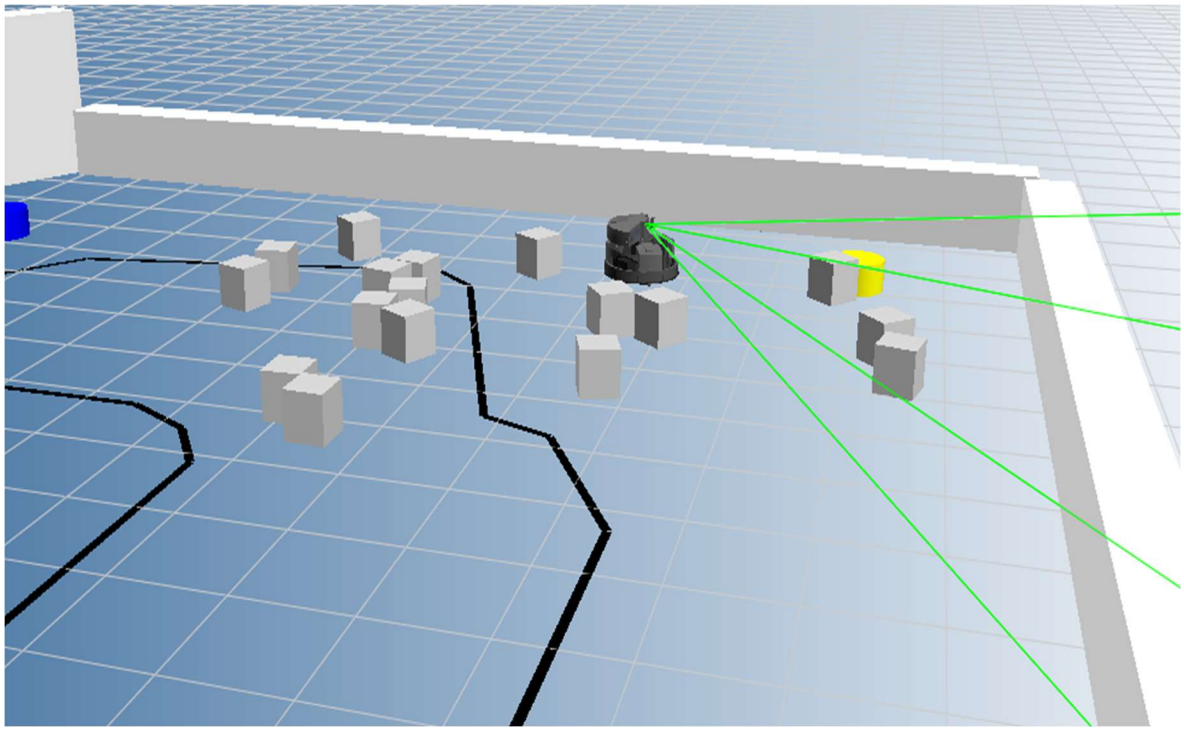

Рисунок 3.2 – Вигляд вікна симулятора в ході роботи програми

3.8 Висновки до третього розділу

У третьому розділі була створена програмна реалізація компонентів системи для управління роботом Robotino. Були запрограмовані такі компоненти, як сервер, клієнт у браузерному середовищі, адаптер та клієнт Robotino. Для виконання розробки були обрані необхідні технології та засоби інтеграції цих технології. Прикладний програмний інтерфейс сервера був реалізований з урахуванням сучасних методів та рекомендацій розробки програмного забезпечення. Для перевірки працездатності розробленої системи була створена тестова програма та продемонстровано результати її роботи.

Створена система управління роботом Robotino дозволяє використання робота не маючи фізичного доступу до нього та без необхідності наявності програмного забезпечення компіляції.

4 ОХОРОНА ПРАЦІ

4.1 Забезпечення безпеки робочого місця

Для розробки програмного забезпечення було використано приміщення, яке має такі особливості: одне робоче місце з персональним комп’ютером та маршрутизатором з доступом до мережі інтернет. Зображення зі схемою місця роботи відображено на рисунку 4.1.

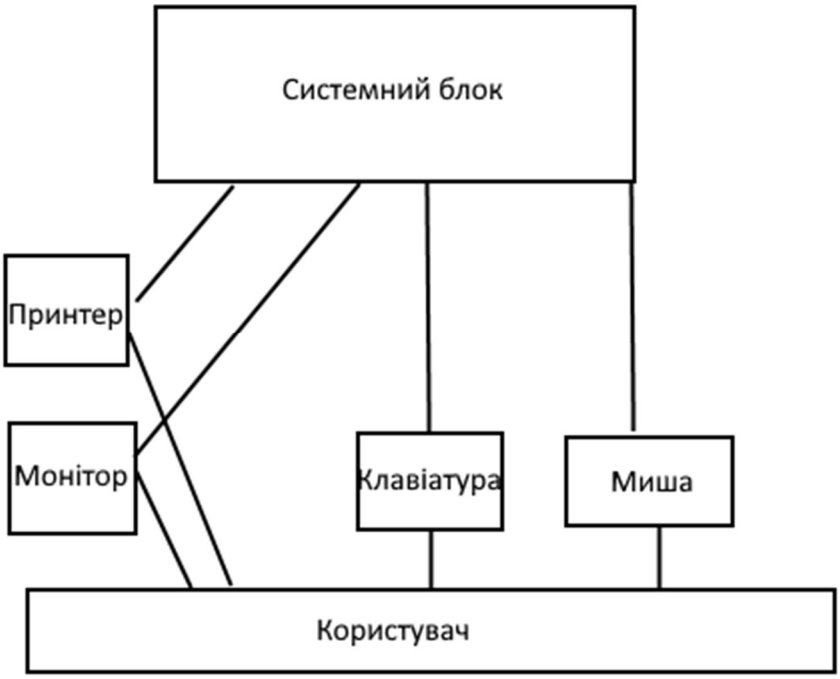

Рисунок 4.1 – Функціональна схема робочого місця

Оскільки приміщення, у якому виконується розробка програмного забезпечення, не має небезпечних пристроїв або умов, тому ця кімната не має підвищеної небезпеки. До додаткових засобів контролю наявні пристрої для захисту від підвищеного струму, перевантаження та розподілений електрощит. Розетки мають захисні пристрої для обмеження прямого доступу та напис 220.

Важливими є інструктажі з техніки безпеки, а саме вступні інструктажі, первинні, цільові.

Для знаходження умов надійного виконання зануленням задач необхідно розрахувати параметри мережі. До цих задач можна віднести підвищену безпеку для людини навіть у випадку аварійного періоду та швидке відключення пошкодженого елементу мережі.

На рисунку 4.2 зображена схема занулення.

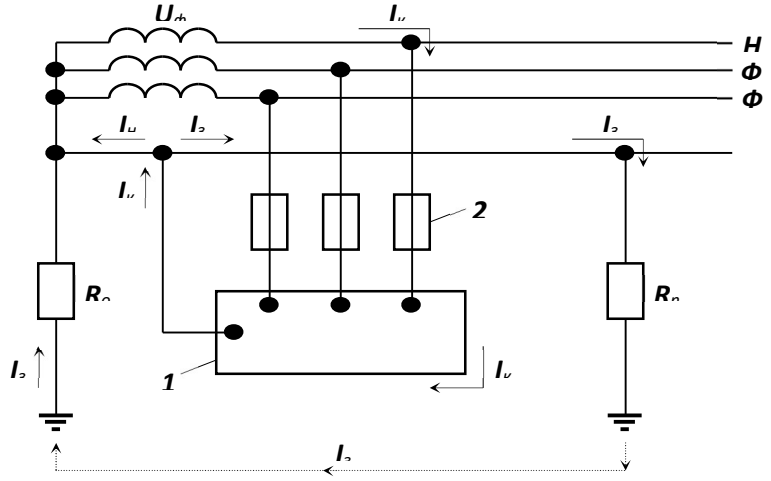

Рисунок 4.2 – Схема занулення в трифазній мережі

Таблиця 4.1 зображає параметри схеми рисунку 4.2.

Таблиця 4.1 – Параметри схеми рисунку 4.2

Параметр	Опис параметру
1	корпус електроустановки;
2	пристрої для захисту від короткого замикання (КЗ)
Ф	провідник фази
НЗ	провідник нулю
R_o	опір заземлення нейтрали
$R_п$	опір повторного заземлення
$I_к$	струм КЗ
$I_н$	струму КЗ, що проходить через провідник нулю

Для розрахунку струму КЗ у випадку замикання фази на корпус обладнання зроблено обчислення нижче.

У випадку нормальної роботи мережі струм обчислюється як:

$$I_{max} = P_н / U_ф, \tag{4.1}$$

де P – потужність електрообладнання, 2 *кВт*;

$U_ф$ – напруга, 220 *В*.

$$I_{max} = 2000 / 220 = 9{,}0\ A.$$

Перетин провідників по економічній щільності струму розраховується за формулою нижче:

$$S_e = I_{max} / J_{ек}, \tag{4.2}$$

де $J_{ек}$ – економічна щільність струму, $А/мм^2$.

Так як кількість робочих годин становить 2520 годин на рік, то $J_{ек} = 2{,}3\ А/мм^2$ для мідного кабелю.

$$S_e = 9{,}09 / 2{,}3 = 3{,}95\ мм^2.$$

Округливши значення перетину до найближчого стандартного значення отримаємо $S = 4\ мм^2$.

Опір фазного проводу довжиною *L* з матеріалу з питомим опором ρ і перетином *S* обчислюється за формулою нижче.

$$R_ф = \frac{l \cdot \rho}{S}, \tag{4.3}$$

де l – відстань до підстанції, 100 *м*;

ρ – питомий опір міді, 0,017 *Ом·мм²/м*;

$$R_ф = \frac{100 \cdot 0{,}017}{4} = 0{,}42 \ \ Ом.$$

Коли перетин фазного провідника має мале значення, то можна допустити $S_н = S_ф$ та $R_н = R_ф$.

У випадку з мідними проводами їх внутрішній індуктивний опір фазного та нульового проводів, зовнішній індуктивний опір контуру малі. Тому ці значення можна проігнорувати.

Активний опір петлі вазначається за формулою:

$$R_п = R_ф + R_н, \tag{4.4}$$

де $R_н$ – активний опір нульового провідника:

$$R_п = 2 \cdot R_ф \tag{4.5}$$

$$R_п = 2 \cdot 0{,}42 = 0{,}84 \ Ом.$$

Виберемо трансформатор потужністю 100 *кВА*, його опір $Z_т / 3 = 0{,}26$ *Ом*.

Дійсний струм короткого замикання при замиканні фази на корпус електроустановки обчислюється як:

$$I_{кз} = U_ф / (Z_{кз} + Z_т / 3), \tag{4.6}$$

де $Z_т$ – повний опір обмоток трансформатора.

$$I_{кз} = U_ф / (Z_{кз} + Z_т / 3) = 200 \ A.$$

Як результат був обраний автомат захисту А31 24Т 380 / 220*В* (50*А*). Цей автомат відключає струм при збільшенні навантаження понад 50*А*.

4.2 Висновки до четвертого розділу

У четвертому розділі було проаналізовано робоче приміщення та вказано використане обладнання. Також було знайдено, що кімната не відноситься до приміщень з високою небезпекою, що зменшує необхідність контролю за електричними пристроями. Тому було описано необхідні правила для забезпечення безпеки у приміщенні та знайдено необхідний автомат захисту від короткого замикання.

ВИСНОВКИ

В даний час розробка моделей та методів взаємодії є невід'ємною частиною під час проектування тої чи іншої системи. Автоматизовані системи зараз використовуються у різних сферах діяльності людини і допомагають покращити точність, якість та швидкість роботи, тому дана атестаційна робота є актуальною.

У першому розділі дослідницької роботи було проаналізовано існуючі роботи та методи керування ними. Було знайдено недоліки методів керування роботом Robotino, що є причиною необхідності розробки вдосконалених методів.

В результаті дослідницької роботи у другому розділі було створено покращені способи керування, що виконують поставлену мету. А саме, ці методи надають можливість віддаленого керування роботом Robotino.

Третій розділ дослідницької роботи складається з розробки цих методів, використовуючи модель робота Robitono. Результатом є створений програмний комплекс, що виконує вказану мету. Також результат був перевірений у моделі робота Robotino Sim.

Під час виконання дослідницької роботи були вирішені такі задачі:

– аналіз існуючих архітектур програмного забезпечення;

– дослідження методологій розробки;

– розробка архітектури програми;

– розробка логічної частини програми;

– розробка графічного інтерфейсу.

Розроблене програмне забезпечення може бути покращене шляхом створення нових компонентів або ускладнення логіки вже існуючих для отримання нових можливостей у керуванні елементами роботу та для отримання статистики.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. ДСТУ 3008: 2015. Інформація та документація. Звіти у сфері науки і техніки. Структура і правила оформлення [Текст] - К.: ДП «УкрНДНЦ» , 2016. - 31 с.

2. Невлюдов, І.Ш. Дипломне проектування для студентів усіх форм навчання спеціальностей 151 «Автоматизація та комп'ютерно-інтегровані технології» [Текст]: навч. посіб. / І.Ш. Невлюдов, А.О. Андрусевич, О.В. Токарєва, Г.В. Пономарьова. – К.: Київ-58, пр. Космонавта Комарова, 1, 2016. – 320 с.

3. Методичні вказівки з «Розробки й оформлення магістерської атестаційної роботи» для студентів другого (магістерського) рівня вищої освіти галузі знань 15 Автоматизація та приладобудування за спеціальністю 151 Автоматизація та комп'ютерно-інтегровані технології освітні програми: «Автоматизоване управління технологічними процесами», «Комп'ютерно-інтегровані технологічні процеси і виробництва», «Комп'ютеризовані та робототехнічні системи» [Текст] / Упоряд. І.Ш. Невлюдов, В.В. Косенко, В.В. Євсєєв. – Харків: ХНУРЕ, 2019. – 55 с.

4. Положення про протидію академічному плагіату в ХНУРЕ [Електронний ресурс] / Режим доступу: www/ URL: https://nure.ua/wp-content/uploads/Main_Docs_NURE/Polozhennya-pro-protidiyu-akademichnomu-plagiatu-v-HNURE----290-vid-28.04.2017.pdf - 29.08.2019р. - Назва з екрану.

5. Nevliudov, I. Intelligent Decision-Making Support for Flexible Integrated manufacturing [Текст] / I. Nevliudov, O. Tsymbal, A. Andrusevitch, V. Gopejenko.– Riga: ISMA, 2020. – 390 p.

6. Nevlyudov, I.S. Acoustic model application to mobile robot guidance [Текст] / Nevlyudov I.S., Tsimbal A.M., Milyutina S.S., Sharkovsky V.Y.– Telecommunications And Radio Engineering, 2012, v 71, No 17, P. 1589-1597.

7. Jungwook Han Autonomous collision detection and avoidance for ARAGON USV: Development and field tests [Електронний ресурс] – Режим доступу: https://onlinelibrary.wiley.com/doi/full/10.1002/rob.21935 – (дата звернення: 10.10.2020). – Назва з екрану.

8. Bronnikov, Artem. Flexible manufacturing tendencies and improvements with visual sensoring [Текст] / Artem Bronnikov, Nevliudov Igor, Oleksandr Tsymbal. – Eskisehir Technical University Journal of Science and Technology. Applied Sciences and Engineering, 2019. Vol. 20, ICONAT issue, P. 77-83.

9. Ralph-Christoph Weber Festo Robotino Manual [Text] / Ralph-Christoph Weber, Markus Bellenberg – Denkendorf, 2010. – P. 16.

10. Олифер, В. Г. Сетевые операционные системы [Текст] / В. Г. Олифер, Н. А. Олифер. – СПб.: Питер, 2009. – 669 с.

11. TCP three-way handshake [Електронний ресурс]. – Режим доступу: https://study-ccna.com/tcp-three-way-handshake. – (дата звернення: 15.10.2020). – Назва з екрану.

12. Robotino® View [Електронний ресурс]. – Режим доступу: https://www.festo-didactic.com/int-en/services/robotino/programming/robotino+view/?fbid=aW50LmVuLjU1Ny4xNy4zNC4xNDI2. – (дата звернення: 16.10.2020). – Назва з екрану.

13. ctypes — A foreign function library for Python [Електронний ресурс]. – Режим доступу: https://docs.python.org/3.3/library/ctypes.html. – (дата звернення: 16.10.2020). – Назва з екрану.

14. API2 [Електронний ресурс]. – Режим доступу: https://wiki.openrobotino.org/index.php?title=API2. – (дата звернення: 16.10.2020). – Назва з екрану.

15. Посредник (шаблон проектирования) handshake [Електронний ресурс]. – Режим доступу: https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%81%D1%80%D0%B5%D0%B4%D0%BD%D0%B8%D0%BA_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%

D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F. – (дата звернення: 17.10.2020). – Назва з екрану.

16. WebAssembly [Електронний ресурс]. – Режим доступу: https://developer.mozilla.org/ru/docs/WebAssembly. – (дата звернення: 19.10.2020). – Назва з екрану.

17. Standardizing WASI: A system interface to run WebAssembly outside the web [Електронний ресурс]. – Режим доступу: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface. – (дата звернення: 19.10.2020). – Назва з екрану.

18. Getting started with WebGL development [Електронний ресурс]. – Режим доступу: https://docs.unity3d.com/2020.1/Documentation/Manual/webgl-gettingstarted.html. – (дата звернення: 19.10.2020). – Назва з екрану.

19. Метапрограммирование [Електронний ресурс]. – Режим доступу: https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%B0%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5. – (дата звернення: 20.10.2020). – Назва з екрану.

20. Chapter 8. Data Types [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/docs/9.5/datatype.html. – (дата звернення: 21.10.2020). – Назва з екрану.

21. Data conversion [Електронний ресурс]. – Режим доступу: https://en.wikipedia.org/wiki/Data_conversion#Lost_and_inexact_data_conversion. – (дата звернення: 21.10.2020). – Назва з екрану.

22. The TCP Maximum Segment Size and Related Topics [Електронний ресурс]. – Режим доступу: https://tools.ietf.org/html/rfc879. – (дата звернення: 21.10.2020). – Назва з екрану.

23. CMake [Електронний ресурс]. – Режим доступу: https://cmake.org. – (дата звернення: 22.10.2020). – Назва з екрану.

24. Менеджер пакетов - Package manager [Електронний ресурс]. – Режим доступу: https://ru.qaz.wiki/wiki/Package_manager. – (дата звернення: 22.10.2020). – Назва з екрану.

25. Orientation and setup [Електронний ресурс]. – Режим доступу: https://docs.docker.com/get-started. – (дата звернення: 22.10.2020). – Назва з екрану.

26. LIB to A converter [Електронний ресурс]. – Режим доступу: https://code.google.com/archive/p/lib2a. – (дата звернення: 22.10.2020). – Назва з екрану.

27. cmake-toolchains(7) [Електронний ресурс]. – Режим доступу: https://cmake.org/cmake/help/latest/manual/cmake-toolchains.7.html. – (дата звернення: 22.10.2020). – Назва з екрану.

28. WebGL: Deploying compressed builds [Електронний ресурс]. – Режим доступу: https://docs.unity3d.com/Manual/webgl-deploying.html. – (дата звернення: 19.10.2020). – Назва з екрану.