Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет __Комп'ютерної інженерії та управління_____
(повна назва)

Кафедра Автоматизації проектування обчислювальної техніки_____
(повна назва)

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

_____другий (магістерський)_____
(рівень вищої освіти)

_____*ГЮІК. 46741Х.001 ПЗ*_____
(позначення документа)

Системи управління реального часу віддаленими об'єктами

(тема)

Виконав: студент 2 курсу, групи СКСм-18-1

_____Євчук К.П._____
(прізвище, ініціали)

Спеціальність *123 _ Комп'ютерна інженерія*

Тип програми ____освітньо-професійна_____
(освітньо-професійна або освітньо-наукова)

Освітня програма_____

*Спеціалізовані комп'ютерні системи*_____
(шифр і назва спеціальності, освітньої прграми)

Керівник роботи __ _ доц. Шкіль О.С.
(підпис) (посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Чумаченко С.В.
(підпис) (прізвище, ініціали)

2020 р.

Харківський національний університет радіоелектроніки

Факультет_____Комп'ютерної інженерії та управління _____

Кафедра_____Автоматизації проектування обчислювальної техніки_____

Рівень вищої освіти____другий (магістерський)_____

Спеціальність_____123_Комп'ютерна інженерія _____
(шифр і назва)

Тип програми _____Освітньо-професійна_____
(освітньо-професійна або освітньо-наукова)

Освітня програма _____Спеціалізовані комп'ютерні системи_____
(повна назва

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
_____ (підпис)
«_____»_____ 20 ___р.

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

студентові_____Євчуку Костянтину Павловичу_____
(прізвище, ім'я, по батькові)

1. Тема роботи (проекту)_ Системи управління реального часу віддаленими об'єктами_____
_____
_____

затверджена наказом по університету від "__"____2020 р. № _Ст_____.____
2. Термін подання студентом роботи до екзаменаційної комісії _____..____
3. Вихідні дані до роботи (проекту)_____
Мережевий Протокол Delay-/Disruption-Tolerant Networking_____
Мова програмування Golang_____
Бібліотека DTN7-GO_____
Середовище розробки Goland IDE_____

_____
4. Перелік питань, що потрібно опрацювати у роботі _____
Методи проектування систем систем реального часу_____
Методи та алгоритми програмування при побудові DTN_____
Система обробки подій_____
Технологічна платформа реалізації_____
Автоматна модель пристрою керування_____
Середовище розробки програмного забезпечення_____

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) _20 слайдів_____
_____
_____
_____
_____
_____

6. Консультанти розділів роботи (проекту)

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

7. Дата видачі завдання 08.01.2020_____

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи (проекту)	Термін виконання етапів проекту (роботи)	Примітка
1	Видача теми проекту, узгодження і затвердження	08.01.2020 -15.01.2020	
2	Аналіз проблемної галузі, постановка задачі, вибір інструментальних засобів	15.01.2020 -05.02.2020	
3	Проектування системи реального часу	05.02.2020 -25.02.2020	
4	Аналіз методів інтеграції протоколу DTN	25.02.2020 -10.03.2020	
5	Розробка сценаріїв програмної реалізації	10.03.2020 - 20.03.2020	
6	Розробка моделі сервісу керування	20.03.2020 -05.04.2020	
7	Розробка програмного забезпечення	05.04.2020 -20.04.2020	
8	Оформлення пояснювальної записки	20.04.2020 -25.04.2020	
9	Захист проекту	25.04.2020 -29.04.2020	

Студент _____
(підпис)

Керівник роботи (проекту) _____ _____доц. Шкіль О.С.___
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка містить 70 сторінок, 27 рисунків, 1 таблицю, 8 джерел за переліком посилань.

УПРАВЛІННЯ ВІДДАЛЕНИМ ОБ’ЄКТОМ, ОБРОБКА ПОДІЙ, СИСТЕМИ РЕАЛЬНОГО ЧАСУ, ПРОТОКОЛ МЕРЕЖІ СТІЙКОЇ ДО РОЗРИВІВ ТА ЗАТРИМОК, DTN, МОВА ПРОГРАМУВАННЯ GOLANG, GOLAND IDE

Метою роботи є розробка методу побудови системи реального часу з використанням технології віддаленої комунікації DTN та паралельного програмування.

Розроблений прототип системи управління віддаленим об’єктом з можливістю збереження та синхронізації даних при втраті зв’язку. Для реалізації програмного забезпечення застосована розподілена архітектура, об’єкти котрої представлені у вигляді програм-сервісів з використанням протоколу DTN для комунікацій між сервісом керування (головним) сервісом та об’єктами.

Програмна реалізація системи здійснена за технологією паралельного програмування. Побудова протоколу мережі зв’язку здійснена за концепцією Delay-Tolerant Networking, яка відповідає критеріям та специфікації Bundle Protocol Version 7.

Сервіс керування реалізований за допомогою мови програмування Golang у середовищі розробки Goland IDE. Конфігурація системи здійснюється за допомогою JSON-файлів, а безпосередньо управління – REST API. Сервіс користується Influx (для статистичних даних) та MySQL (у якості основної) базами даних для збереження.

ABSTRACT

Bachelor’s thesis contains 70 pages, 27 figures, 1 table, 8 sources according to the list of links.

REMOTE OBJECT MANAGEMENT, EVENTS PROCESSING, REAL-TIME SYSTEMS, DELAY & DISRUPTION-TOLERANT NETWORKING, DTN, GOLANG PROGRAMMING LANGUAGE, GOLANG IDE

The purpose of the work is to build a model of the remove object’s management system with the features to save and synchronize the data in case of the connection loss. In order to implement the software of the product there was used a distributed architecture with a set of objects-services that communicate with each other and the main management service by DTN protocol.

The software implementation was done using concurrent programming for handling multiple requests. The build of network communication is done using Delay-Tolerant Networking concept, which follows the Bundle Protocol Version #7 specification.

The management service is implemented using Golang programming language in the GolandIDE. The service’s configuration is done with the help of JSON-formatted files along with the management via REST API. The services uses Influx and MySQL databases as a storage.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ОУ – об’єкт управління;

СК – сервіс керування;

СРЧ – система реального часу;

CLA – Convergence Layer Adapter (адаптер конвергентного рівня);

ПЗ – програмне забезпечення;

HTTP – Hypertext Transfer Protocol (протокол передачі гіпер-текстових документів);

API – Application Programming Interface (прикладний програмний інтерфейс);

REST – Representational state transfer (передача репрезентативного стану);

JSON – JavaScript Object Notation (запис об’єктів JavaScript);

DTN – Delay-Tolerant Networking (протокол мережі толерантної до обривів зв’язку).

ВСТУП

З 1957 року і до сьогодні людство час від часу отримує новини про успішний запуск супутників. “Сьогодні 4 жовтня 1957 року був виведений перший Простий Супутник-1”; “Компанія StarLink вивела на навколоземну орбіту 60 супутників. Це дозволить компанії побудувати мережу супутників для роздачі інтернету” – така новина лунала 24 травня 2019 року. Між цими двома подіями різниця у 62 роки, але є те, що їх об’єднує – неперервна комунікація з пультом керування на Землі.

Загалом у концепції побудови систем керування віддалених об’єктів у реальному часі лежить декілька компонентів:

– безпосередньо віддалений об’єкт, відповідальний за проведення наукових досліджень, обчислювальних процесів;

– пульт (сервіс) керування, відповідальний за прийняття та доставку конфігурації віддалених об’єктів в залежності від отриманих даних.

Мета роботи – розробка методу побудови системи реального часу з використанням технології віддаленої комунікації DTN та паралельного програмування.

Актуальність даної роботи пов’язана зі зростаючою потребою в реалізації систем реального часу у віддаленому контексті, а відповідно і у тестуванні.

Поставлені наступні завдання:

– дослідження протоколів мережі DTN та способи інтеграції в системи реального часу;

– розробка системи управління реального часу віддаленими об’єктами;

– аналіз поведінки системи на збої та розриви у зв’язку;

– розробка алгоритму тестування конфігурації у контексті управління віддаленими об’єктами.

Об’єктом дослідження є моделі систем управління реального часу у віддаленому контексті.

Предметом дослідження є застосування інтеграції DTN протоколу при проектуванні системи управління реального часу за допомогою мови програмування Golang.

Реалізація системи повинна розв’язувати проблему втрати зв’язку між множиною об’єктів у розподіленій системі реального часу зі збереженням та синхронізацією даних. Сферами, що повинна покривати реалізація системи, может бути авіа, космічна, наземна інфракструктура з зовнішними катаклізмами і т.д

Технічне завдання полягає в розробці моделей віддалених об’єктів та сервісу керування, що можуть бути протестовані на втрату зв’язку, помилки та попередження

.

1 ВІДДАЛЕНІ СИСТЕМИ РЕАЛЬНОГО ЧАСУ

1.1 Визначення систем у віддаленому контексті

Апаратне забезпечення комп'ютера вирішує проблеми, повторюючи інструкції, спільно відомі як програмне забезпечення. З іншого боку, програмне забезпечення традиційно розділяється на системні та прикладні програми.

Системні програми складаються з програмного забезпечення, яке має взаємодію з базовими комп'ютерними пристроями, такими як драйвери пристроїв, обробники переривань, планувальники завдань та різні програми, які служать інструментами для розробки або аналізу прикладних програм. Ці програмні засоби включають компілятори, які транслюють мовні програми високого рівня в код збірки; асемблери, які перетворюють код збірки у спеціальний бінарний формат, який називається машинним кодом; і лінкери/локатори, які готують об'єктний код для запуску в середовищі. Операційна система – це колекція системних програм, що управляють фізичними ресурсами комп'ютера. Таким чином, операційна система реального часу є справді важливою системною програмою.

Застосункові програми – це програми, призначені для вирішення специфічних завдань, таких як оптимальне розміщення залу – виклик ліфта у висотному будинку, інерціальне плавання літака та підготовка заробітної плати для певної промислової компанії. Проектний аналіз, дослідження архітектури відіграють важливу роль при розробці системних програм та прикладних програм, призначених для роботи в середовищах реального часу.

Поняття "система" є центральним елементом програмної інженерії, і загалом для всієї інженерії, тож вимагає формалізації.

Коли внутрішні деталі системи не представляють особливого інтересу, функція зображення між вхідними та вихідними компонентами може

розглядатися як чорна скринька з введенням одного або кількох вхідних даних та одним або декількома наборами результатів, що повертаються від системи (рис. 1.1). Крім того, Вернон перераховує п'ять загальних властивостей, які належать до будь-якої "системи" [2]:

– система являє собою набір органічно поєднаних компонентів;

– система кардинально змінюється, якщо компонент додається або залишає її;

– вона має мету;

– вона має ступінь постійності.

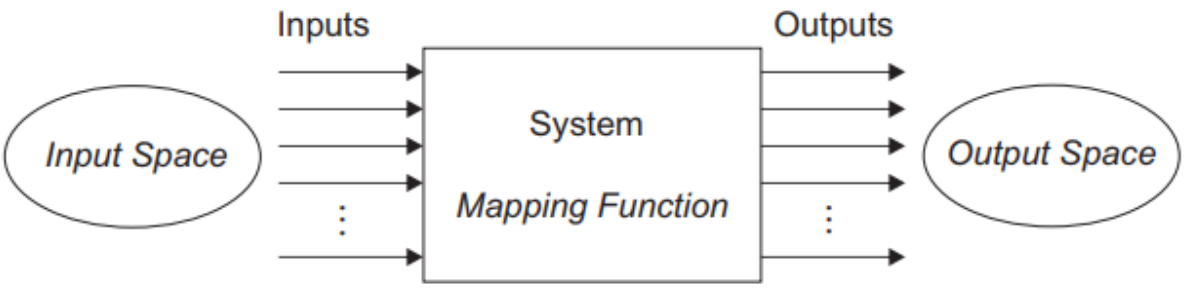

Рисунок 1.1 – Загальна система з вхідними та вихідними даними

Кожна реальна сутність, органічна або синтетична, може бути змодельована як система. В обчислювальних системах входи представляють цифрові дані з апаратних пристроїв або інших програмних систем. Входи часто пов'язані з датчиками, камерами та іншими пристроями, що забезпечують аналогові входи, які перетворюються в цифрові дані або забезпечують прямі цифрові входи. З іншого боку, цифрові виходи комп'ютерних систем можуть бути перетворені в аналогові виходи для керування зовнішніми апаратними пристроями, такими як виконавчі пристрої та дисплеї, або використовуватися безпосередньо без будь-якої конверсії (рис.1.2).

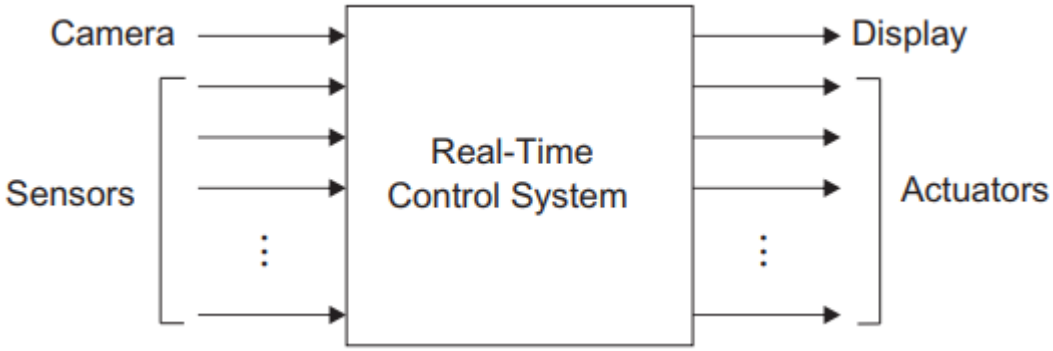

Рисунок 1.2 – Система керування реального часу

1.2 Особливості систем реального часу

Термін "реальний час" широко використовується у багатьох контекстах, як технічних, так і звичайних. Більшість людей думають, що «в реальному часі» означає «відразу» або «миттєво». Однак, словник Random House визначає «реальний час» як додаток, в яких комп'ютер повинен відповідати так швидко, як це вимагається користувачем або процесом, що контролюється користувачем. Ці визначення досить різні, і їх відмінності часто є причиною непорозуміння між комп'ютером, програмним забезпеченням і системними інженерами, а також користувачами систем реального часу.

Розглянемо комп'ютерну систему, в якій дані повинні оброблятися за звичайною швидкістю. Наприклад, для визначення свого положення літак використовує послідовність імпульсів акселерометра. Системи, відмінні від авіонічних, можуть також вимагати швидкого реагування на події, які відбуваються за нерегулярними темпами, наприклад, рівень перегріву в ядерній електростанції. Навіть без визначення терміну «реальний час», мабуть, зрозуміло, що ці події вимагають своєчасної або «реальної» обробки.

Тепер розглянемо ситуацію, коли пасажир підходить до співробітника авіакомпанії, щоб забрати свій посадковий талон на певний рейс з Нью-Йорка до Бостона, який вилітає через п'ять хвилин. Секретар бронювання

вводить відповідну інформацію в комп'ютер, і через кілька секунд друкується посадковий талон. Це система реального часу?

Справді, всі три системи – авіація, атомна електростанція та бронювання авіаліній – в режимі реального часу, тому що вони повинні обробляти інформацію протягом певного інтервалу. Хоча ці приклади можуть забезпечити інтуїтивне визначення системи реального часу, необхідно чітко зрозуміти, коли система є реальним часом і коли не є.

Система реального часу – це система, що повинна обробити запит та сконструювати відповідь або застосувати зміни протягом зазначеного часу, інакше з’являється ризик виникнення послідовності проблем та похибок. Тобто, у таких системах результат чи дія у режимі реального часу має сенс тільки протягом зафіксованого “дедлайну”, або протягом часу. Окрім цього, важливим є як відмітка часу, так і запит, адже результат найчастіше залежить від зовнішньої події. Системи реального часу зустрічаються в різних галузях комп’ютерних програм: у сфері оборони та космічних систем, стрімінгових мультимедійних сервісах (Netflix, Youtube Originals), вбудованих автоматичних системах і т.д.

У системах реального часу правильна поведінка формується не тільки за рахунок логіки обчислень, а також щоб на момент часу результат був сформований. В режимі реального часу система може змінювати свій стан, більше цього реакція на подію може продовжуватися навіть, якщо система керування на паузі, або був втрачений зв’язок. Виходячи з цього, системи реального часу можуть мати розподілену архітектуру.

Система реального часу повинна реагувати на отримані події або “подразники” від сервісу керування (оператора) протягом часових інтервалів в залежності від середовища. Момент (відмітка часу), в котрий був підготовлений результат, називається “дедлайном” або терміном. Якщо результат або відповідь від системи є дійсною після дедлайну, така система класифікується як “некритична система реального часу” або “критична” якщо термін був пропущений і результат не є дійсним.

Рисунок 1.3 – Приклад розподіленої системи реального часу

Серед критичних та некритичних систем є низка характеристик, котрі дозволяють провести чітку порівняльну характеристику та визначити відмінності. По-перше, часові терміни (дедлайни) – зазвичай некритичні системи мають більш гнучкі терміни виконання запиту, але це не означає, що програма онлайн-банкінгу може чекати дані про грошові надходження понад року або двох. Терміни важливі, але по-своєму гнучкі в залежності від галузі застосування. По-друге, наявність переривань – наступна характеристика після дедлайнів, що означає те чи інше залучення від подій. Вони часто з’являються у вигляді сигналів переривань, що виникають внаслідок надходження нових даних або апаратних сигналів.

1.3 Некритичні системи. Компоненти

Некритичні системи мають менш жорсткі обмеження часу та дедлайни. В режимі реального часу такі системи, як правило, використовуються, коли є кілька підключених систем, які необхідно підтримувати, незважаючи на зміни подій та обставин. Ці системи також використовуються при наявності одночасних вимог доступу. Наприклад, програмне забезпечення, яке

використовується для підтримки розкладу поїздок для великих транспортних компаній, часто функціонує в реальному часі. Для такого програмного забезпечення необхідно оновлювати графіки з невеликою затримкою. Однак затримка на кілька секунд, швидше за все, не спричинить хаос. Прикладом некритичної системи в режимі реального часу є перегляд веб-сторінок.

Зазвичай після пошуку за URL-адресою (Уніфікований локатор ресурсів) відповідна веб-сторінка забирається та відображається в середньому за кілька секунд. Однак, коли потрібно кілька хвилин для відображення запитуваної сторінки, ми все ще не вважаємо систему не фукціонуючою, а просто вважаємо, що продуктивність системи знизилася. Інший приклад некритичної системи в режимі реального часу – це запит на обробку завдань щодо бронювання місця в додатку бронювання залізничних квитків. Після того, як буде зроблено запит на бронювання, відповідь має повернутися в середньому за 20 секунд. Відповідь може бути або у формі друкованого квитка, або з вибаченням через відсутність місць. Крім того, можно висловити обмеження щодо завдання на придбання квитків як: принаймні в 95% запитів на бронювання, квиток повинен бути оброблений та надрукований менш ніж за 20 секунд.

1.4 Критичні системи. Особливості

Критичні системи в режимі реального часу – це системи, які можуть створювати свої результати протягом певних заздалегідь визначених часових меж. Вважається, що система вийшла з ладу, коли будь-яке з її критичних завдань у режимі реального часу не дає необхідних результатів до зазначеного часового обмеження (дедлайну). Якщо процес вважається критичним у режимі реального часу, він повинен завершити свою роботу до певного часу. Якщо він не досягне свого терміну, його робота не має значення, і система, для якої він є компонентом, може зіткнутися з відмовою. Однак, коли система вважається некритичною в режимі реального часу, тоді

є певний простір для запізнення. Наприклад, у такій системі затримка процесу може не спричинити збій всієї системи. Натомість це може призвести до зниження звичної якості процесу чи системи. Критичні системи реального часу: вони часто використовуються у вбудованих системах. Розглянемо, наприклад, систему управління автомобільним двигуном. Така система вважається критичною, оскільки пізній процес може призвести до виходу з ладу двигуна. Прикладом системи, що має критичні завдання в режимі реального часу, є робот. Робот циклічно здійснює низку заходів, включаючи спілкування з системою-хостом, реєстрацію всіх завершених заходів, зондування навколишнього середовища для виявлення будь-яких наявних перешкод, відстеження об'єктів, що цікавлять, планування шляху, здійснення наступного руху тощо.

У разі, коли раптом робот стикається з перешкодою, він повинен її виявити раніше і якомога швидше спробувати уникнути зіткнення з нею. Якщо він не зможе швидко на неї відповісти (тобто відповідні завдання не виконані до необхідного часу), то він зіткнеться з перешкодою. Тому виявлення перешкод та реагування на них - критичні завдання в реальному часі. Інша програма, що має критичні завдання в режимі реального часу – протиракетна система. Протиракетна система повинна спочатку виявити всі вхідні ракети, належним чином розташувати протиракетну зброю, а потім здійснити вогонь для знищення вхідної ракети, перш ніж вхідна ракета зможе завдати будь-якої шкоди. Усі ці завдання носять критичний характер у режимі реального часу, і протиракетна система вважатиметься невиконаною, якщо якесь з її завдань не було виконане до відповідних часових обмежень. Зазвичай програми, що мають критичні завдання в режимі реального часу, є надзвичайно важливими. Це означає, що будь-яке невиконання завдання в реальному часі, включаючи невиконання пов'язаних з ними термінів, призведе до тяжких наслідків.

Критичність завдання може варіюватися від надзвичайно критичної до легко критичної. Отже, критичність завдання - це інший вимір, ніж критичні

або некритичні характеристики завдань. Критичність завдання – це міра вартості відмови – чим вища вартість відмови, тим критичнішою є задача.

Задача з більш високим пріоритетом, як кажуть, витісняє нижчий пріоритет, якщо він перериває виконання завдання з нижчим пріоритетом. Системи, які використовують схеми виключення замість кругового планування, називаються попередніми системами. Пріоритети, призначені для кожного переривання, ґрунтуються на важливості та актуальності завдання, пов'язаного з перериванням. Наприклад, система нагляду за атомною електростанцією проектується як система пріоритетів. Хоча, наприклад, належне поводження з діями програмних зловмисників (хакерів) є критичним, ніщо не є більш важливим, ніж обробка попередження про перегрівання ядра. Пріоритетне переривання може бути або фіксованим пріоритетом, або динамічним типом пріоритету. Системи з фіксованим пріоритетом є менш гнучкими, оскільки пріоритети завдань не можуть бути змінені після ініціалізації системи. Системи динамічного пріоритету, з іншого боку, дозволяють коригувати пріоритет завдань під час виконання, щоб задовольнити мінливі потреби в реальному часі. Тим не менш, більшість вбудованих систем реалізуються з фіксованими пріоритетами, оскільки існують обмежені ситуації, в яких системі необхідно налаштувати пріоритети під час виконання. Переважно-пріоритетні системи можуть постраждати від викривлення ресурсів завдяки вищим пріоритетам. Це може призвести до відсутності наявних ресурсів для виконання завдань з нижчим пріоритетом. Потенційна проблема нестачі ресурсів повинна бути ретельно розглянута при розробці переважно-пріоритетних систем.

1.5 Віддалені системи

Яскравим прикладом систем реального часу віддалених об’єктів

можуть бути авіа або космічні програми. Космічні програми можуть бути розділені як ті, що працюють на планеті та на борту. Наземні системи забезпечують наземну та оперативну підтримку космічних місій (наприклад, Міжнародної космічної станції). Наземна підтримка може включати в себе основну базу даних конфігурації програми (наприклад, Міжнародна космічна станція має 2 мільйони компонентів під контролем конфігурації), середовища розробки програмного забезпечення, підтримку моделювання та тестування. Залежно від типу космічних апаратів і вимог місії, наземні програми можуть надавати різні види експлуатаційних послуг. Наприклад, центри супутникового контролю включають такі функції, як супутниковий нагляд та технічний контроль, контроль орбіти, службу корисного навантаження та супутникову експертну оцінку.

Бортові системи відповідають за моніторинг, контроль та збір даних від систем космічних апаратів і корисних навантажень (наприклад, системи візуалізації). Одним з прикладів бортової системи є вбудована програма для супутника. Бортове програмне забезпечення контролює супутник або прилади корисного навантаження, дозволяє спостерігати та контролювати супутник, а також захищає його у разі відмови. Основними функціями супутникових бортових комп'ютерів є:

– моніторинг та управління функціями і станом космічних апаратів;
– планування та підтримка корисного навантаження;
– контроль орбіти;
– обробка даних;
– резервне копіювання системних даних;
– підтримка зв’язку.

У авіа та космічній галузі існує множина критичних компонентів реального часу, вбудованих у системи, які є частиною довгострокових космічних місій або пілотованих перевезень. Прикладами таких програм є:

– автономний космічний апарат (наприклад, космічний корабль

Rosetta буде летіти 10 років до своєї мети, орбіти комети Wirtanen, з багатьма періодами автономної експлуатації) та певні наукові зонди (наприклад, зонд Huygens, запущений у 1997 році, спустився на поверхню Titan у 2004 році);

– космічні системи, які стійкі до дуже жорстких умов (наприклад, Європейська робото-рука);

– автоматичні системи, що проводять рандеву з пілотованими системами (наприклад, автоматизований транспортний засіб);

– космічні транспортні засоби та платформи (наприклад, транспортний засіб для екіпажу);

– запуск транспортних засобів (наприклад, пускові установки Земля-орбіта для космічних кораблів).

Крім того, застосування в повітряних системах (які використовуються для контролю і керування літаком, а також для відображення даних польоту та навігації пілотам) є критичними для безпеки, якщо вплив аварії може перешкодити безпеці польоту і посадки літака. Прекрасним прикладом передової системи реального часу є Mars Exploration Rover NASA (рис.1.4). Це автономна система з екстремальними вимогами до надійності; отримує команди і посилає дані вимірювань по радіозв'язку та виконує наукові завдання за допомогою декількох сенсорів, процесорів і приводів.

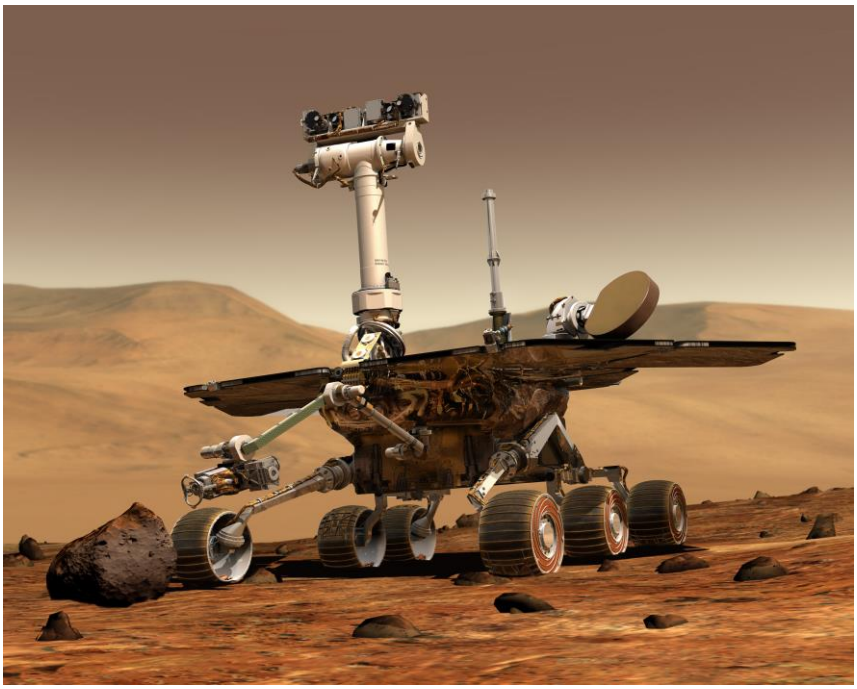

Рисунок 1.4 – Mars Exploration Rover

2 МЕРЕЖЕВИЙ ПРОТОКОЛ ЗВ’ЯЗКУ ВІДДАЛЕНИХ СИСТЕМ

2.1 Визначення та характеристика DTN мережі

Маршрутизація в DTN (delay-tolerant networking), стосується можливостей транспортування або маршрутизації даних від джерела до пункту призначення, що є базисом, який мають мати всі комунікаційні мережі. Мережі толерантні до затримок та обривів (DTN) характеризуються стабільністю до відсутності зв’язку. У середовищах зі складними умовами таки популярним протоколам спеціального маршрутизації, як AODV та DSR або вже відомим протоколам TCP або HTTP (попередня реалізація системи), не вдається встановити маршрути. Це пов'язано з тим, що ці протоколи намагаються спочатку встановити повний маршрут, а потім, після встановлення маршруту, переслати фактичні дані. Однак, коли миттєві кінцеві шляхи встановити важко або неможливо, протоколи маршрутизації повинні скористатися алгоритмом "зберігти і передати", де дані поступово переміщуються та зберігаються по всій мережі з метою, що вони в кінцевому підсумку досягають місця призначення. Загальна методика, яка використовується для максимізації ймовірності успішного передачі повідомлення – це копіювання багатьох копій повідомлення з метою, що вдасться досягти пункту призначення.

Існує багато характеристик протоколів DTN, включаючи маршрутизацію, яку слід враховувати. По-перше, якщо інформація про майбутні контакти буде легко доступною. Наприклад, у міжпланетних комунікаціях багато разів планета чи Місяць є причиною зриву контакту, а велика відстань є причиною затримки зв'язку. Однак, завдяки законам фізики, можна передбачити майбутнє за часом, коли контакти будуть доступні та як довго вони триватимуть. Ці типи контактів відомі як заплановані або передбачувані. З іншого боку, майбутнє місцезнаходження

об'єктів, що знаходяться у мережах, де відновлення після стихійних ситуацій може бути невідомим, наприклад при виникненні надзвичайних ситуацій (останні пожежі в Австралії). Ці типи контактів відомі як переривчасті або опортуністичні контакти.

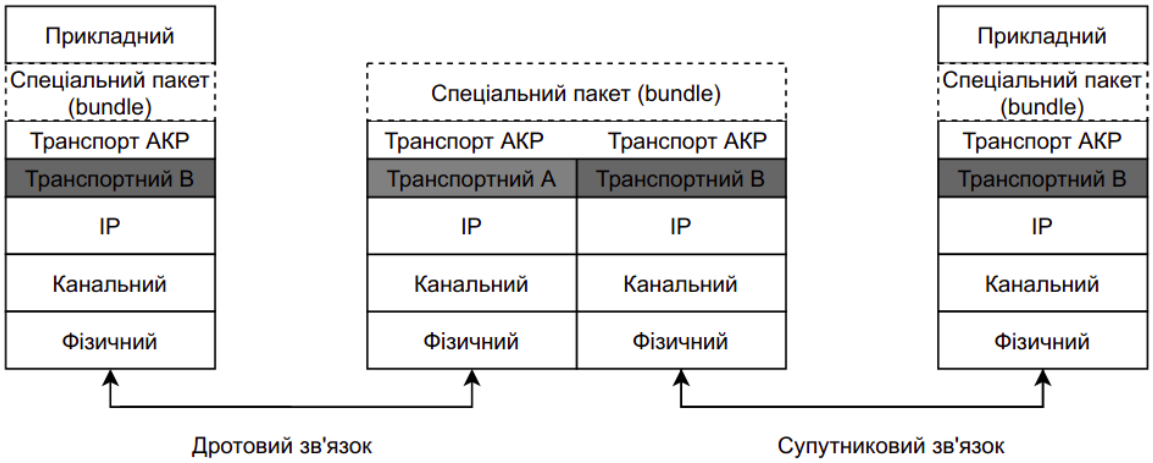

Рисунок 2.1 – Модель DTN мережі

По-друге, питання про те, чи можна експлуатувати мобільність, і якщо так, то які вузли є мобільними. Є три основні випадки, що класифікують рівень мобільності у мережі:

– відсутність мобільних об'єктів. В такому випадку контакти з’являються і зникають виходячи виключно з якості каналу зв'язку між ними. Наприклад, у міжпланетних мережах великі об'єкти в просторі, такі як планети, можуть блокувати комунікаційні вузли протягом заданого періоду часу;

– деякі, але не всі, вузли в мережі є мобільними. Ці вузли, які іноді називають Data Mules, використовуються для їх мобільності. Оскільки вони є основним джерелом перехідного зв'язку між двома не сусідніми вузлами в мережі, важливим питанням маршрутизації є те, як правильно розподілити дані між цими вузлами;

– переважна більшість, якщо не всі, вузли в мережі є мобільними. У цьому випадку протокол маршрутизації, швидше за все, матиме більше опцій під час контактів, і, можливо, не доведеться використовувати кожну з них. Прикладом такого типу мережі є мережа відновлення після аварій, де всі вузли (як правило, люди та транспортні засоби) є мобільними. Другий приклад – транспортна мережа, де автомобілі, вантажні автомобілі та автобуси виступають в якості комунікаційних структур.

Третє питання – наявність мережевих ресурсів. Багато вузлів, таких як мобільні телефони, обмежені щодо місця зберігання, швидкості передачі та часу автономної роботи. Інші, наприклад автобуси в дорозі, можуть бути не такі обмежені. Протоколи маршрутизації можуть використовувати цю інформацію для найкращого визначення способу передачі та зберігання повідомлень, щоб не перевантажувати обмежені ресурси. З 2008 року, наукове співтовариство почало враховувати управління ресурсами, і це все ще є активною сферою досліджень.

Хоча існує багато характеристик протоколів маршрутизації, один з найбільш безпосередніх способів створення таксономії базується на тому, чи він створює, чи ні, репліки повідомлень. Протоколи маршрутизації, які ніколи не копіюють повідомлення, вважаються тими, що працюють на основі переадресації, тоді як протоколи, які копіюють повідомлення, вважаються на основі реплікації.

У кожного підходу є як переваги, так і недоліки, і відповідний підхід до використання залежить від сценарію. Підходи, орієнтовані на переадресацію, як правило, набагато менш марнотратять мережеві ресурси, тому що лише одна копія повідомлення існує на зберіганні в мережі у певний момент часу. Крім того, коли адресат отримує повідомлення, жоден інший вузол не може мати копію. Це виключає необхідність призначення місця надання зворотного зв'язку в мережі (за винятком, можливо, підтвердження, надісланого відправнику), щоб вказати, що непотрібні копії можуть бути видалені. На жаль, підходи на основі переадресації не дозволяють

забезпечити достатню швидкість доставки повідомлень у багатьох DTN. Протоколи на основі реплікації, з іншого боку, дозволяють підвищити швидкість доставки повідомлень, оскільки в мережі існує декілька копій, і лише одна повинна дістатися до пункту призначення. Однак компроміс тут полягає в тому, що ці протоколи можуть витрачати цінні мережеві ресурси. Деякі протоколи, такі як Spray і Wait, намагаються піти на компроміс, обмеживши кількість можливих реплік певного повідомлення.

2.2 Маршрутизація на основі реплікацій

Протоколи на основі реплікації останнім часом привернули багато уваги в науковому співтоваристві, оскільки вони можуть забезпечити значно кращі коефіцієнти доставки повідомлень, ніж протоколи, що базуються на переадресації. Ці типи протоколів маршрутизації дозволяють реплікувати повідомлення; кожну репліку, як і саме оригінальне повідомлення, зазвичай називають копіями повідомлень або репліками повідомлення. Можливі проблеми з маршрутизацією на основі реплікації означають:

– перевантаженість мережі в кластерних районах;

– марні витрати на мережеві ресурси (включаючи пропускну здатність, накопичувач та енергію);

– масштабованість мережі.

Оскільки мережеві ресурси можуть швидко обмежуватися, вирішуючи, які повідомлення передавати першими, а які повідомлення потрібно скинути, відіграють важливу роль у багатьох протоколах маршрутизації.

2.3 Епідемічна маршрутизація

Епідемічна маршрутизація (Epidemic routing) має характер перенасичення, оскільки вузли постійно копіюють та передають повідомлення нововиявленим контактам, які вже не мають копії

повідомлення. У найпростішому випадку епідемічна маршрутизація перенасичена; однак для обмеження кількості передач повідомлень можна використовувати більш складні методи. Епідемічна маршрутизація має свої основи у забезпеченні того, що розподілені бази даних є синхронізованими.

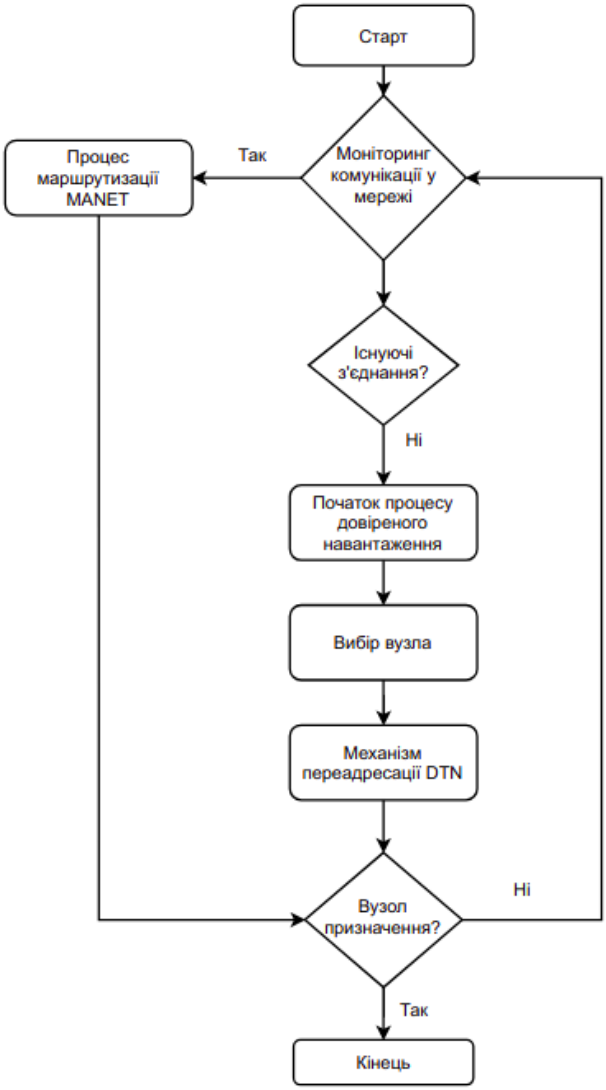

Рисунок 2.2 – Алгоритм визначення вузлу призначення

Епідемічна маршрутизація є особливо «голодною» до ресурсів, оскільки вона навмисно не робить спроб усунути реплікації, що навряд чи поліпшить ймовірність доставки повідомлень. Ця стратегія ефективна, якщо опортуністичні зустрічі між вузлами є чисто випадковими, але в реалістичних ситуаціях зустрічі рідко бувають абсолютно випадковими.

Мули даних (Data Mule) – переважно пов'язані з людиною, рухаються у суспільстві і, відповідно, мають більшу ймовірність зустрітися з певними мулами, ніж інші. Протокол ймовірної маршрутизації з використанням протоколу PRoPHET застосовує алгоритм, який намагається не використовувати випадковість зустрічей у реальному світі, підтримуючи набір ймовірностей для успішної доставки до відомих напрямків у DTN.

Для визначення передбачуваності доставки в кожному мулі використовується адаптивний алгоритм. Мул (Mule) M зберігає передбачуваність доставки P (M, D) для кожного відомого пункту D. Якщо мул не зберігає значення передбачуваності для пункту призначення P (M, D), то він вважається рівним нулю. Передбачуваність доставки, яку використовує кожен мул, перераховується при кожній опортуністичній зустрічі відповідно до трьох правил:

– коли Мул M стикається з іншим мулом E, передбачуваність для E збільшується: P (M, E) новий = P (M, E) старий + (1 - P (M, E) старий) * Lencounter, де Lencounter – константа ініціалізації;

– передбачуваність для всіх напрямків D, окрім E, старий: P (M, D) новий = P (M, D) старий * γK, де γ – константа старіння і K – кількість одиниць часу, що минули з моменту останнього "старіння";

– передбачуваність обмінюється між M і E, а "перехідна" властивість передбачуваності використовується для оновлення напрямків D, для яких E має значення P (E, D) за умови, що M, ймовірно, знову зустріне E: P (M, D) новий = P (M, D) старий + (1 - P (M, D) старий) * P (M, E) * P (E, D) * β, де β – константа масштабування.

Протокол було включено до реалізації, що підтримується дослідницькою групою IRTF DTN, і поточна версія документально зафіксована в RFC 6693. Протокол був випробуваний в реальних ситуаціях під час проекту Sámi Network Connectivity (SNC) і в подальшому розробляється протягом проекту ЄС «N4C».

MaxProp був розроблений в Університеті Массачусетса, Амхерст і

частково фінансувався DARPA та Національним науковим фондом. Оригінальний документ знайдено на конференції IEEE INFOCOM. MaxProp має перевантажений характер, якщо у випадку виявлення контакту всі повідомлення, які не утримуються контактом, будуть намагатися реплікувати та передавати. Алгоритм MaxProp полягає у визначенні того, які повідомлення слід передавати першими, а які повідомлення слід відкинути першими. По суті, MaxProp підтримує впорядковану чергу, засновану на призначенні кожного повідомлення, впорядкованого за оцінкою ймовірності майбутнього перехідного шляху до цього пункту призначення.

Для отримання цих прогнозованих ймовірностей шляху кожен вузол підтримує вектор розміром n-1 (де n – кількість вузлів у мережі), що складається з ймовірності стикання вузла з кожним з інших вузлів у мережі. Кожному з n-1 елементу у векторі спочатку встановлено 1/(|n| -1), тобто вузол однаково ймовірно зустрінеться з будь-яким іншим вузлом. Коли вузол зустрічається з іншим вузлом – “j” елемент його вектора збільшується на 1, і тоді весь вектор нормалізується таким чином, що сума всіх записів додається до 1. Зауважимо, що ця фаза є повністю локальною і не потребує передачі інформації про маршрутизацію між вузлами.

Коли два вузли зустрічаються, вони спочатку обмінюються своїми векторами ймовірності зустрічі вузлів. В ідеалі кожен вузол матиме актуальний вектор від кожного іншого вузла. Маючи під рукою ці n векторів, вузол може потім обчислити найкоротший шлях за допомогою глибинного пошуку, де ваги шляху вказують на ймовірність того, що зв'язок не відбувається. Ці ваги шляху підсумовуються для визначення загальної вартості шляху та обчислюються для всіх можливих шляхах до бажаних пунктів призначення. Шлях з найменшою загальною вагою вибирається як вартість для конкретного пункту призначення. Потім повідомлення впорядковуються за цілями призначення та передаються і видаляються у певному порядку.

У поєднанні з основною маршрутизацією, описаною вище, MaxProp

забезпечує безліч взаємодоповнюючих механізмів, кожен допомагає в цілому співвідношенню доставки повідомлень. По-перше, підтвердження вводяться в мережу вузлами, які успішно отримують повідомлення (і є кінцевим пунктом призначення цього повідомлення). Ці підтвердження є 128-бітовими хешами повідомлення, які завантажуються у мережу, і вказують вузлам видалити зайві копії повідомлення зі своїх буферів. Це допомагає звільнити місце, тому повідомлення не оновлюються так часто. По-друге, пакетам з низьким числом переходів надається більший пріоритет. По-третє, кожне повідомлення підтримує "перехідний список" із зазначенням попередньо відвіданих вузлів, щоб переконатися, що він не перегляне той самий вузол знову.

2.4 Протокол розподілу ресурсів RAPID

RAPID, що є абревіатурою протоколу розподілу ресурсів для маршрутизації DTN, був розроблений в університеті штату Массачусетс, Амхерст. Вперше він був представлений у публікації SIGCOMM 2007, DTN Routing як проблема розподілу ресурсів. Автори RAPID стверджують, що в якості основного припущення – попередні алгоритми маршрутизації DTN випадково впливають на показники ефективності, такі як середня затримка та коефіцієнт доставки повідомлень. Метою RAPID є навмисне здійснення єдиної метрики маршрутизації.

Ядро протоколу RAPID базується на концепції функції утиліти. Функція утиліти призначає значення утиліти Ui кожному пакету «i», що базується на оптимізованій метриці. RAPID спочатку копіює пакети, що локально призводить до найвищого збільшення корисності. Наприклад, припустимо, що показник для оптимізації – це середня затримка. Функція утиліти, визначена для середньої затримки, Ui = - Di, в основному негативне значення середньої затримки. Отже, протокол копіює пакет, що призводить до найбільшого зменшення затримки. RAPID, як і MaxProp, заснований на

перенавантаженні, і тому намагатиметься копіювати всі пакети, якщо дозволяють мережеві ресурси.

Загальний протокол імплементується за чотири кроки:

– ініціалізація: обмін метаданими, щоб допомогти оцінити утиліти пакетів;

– пряма доставка – передаються пакети, призначені для безпосередніх сусідів;

– реплікація – пакети реплікуються на основі граничної утиліти;

– припинення – алгоритм закінчується, коли контакти розриваються або реплікуються всі пакети;

Spray and Wait – протокол маршрутизації, який намагається отримати переваги співвідношення доставки маршрутизації на основі реплікації, а також низькі переваги використання ресурсів маршрутизації на основі переадресації. Spray and Wait був розроблений дослідниками університету Південної Каліфорнії. Вперше він був представлений на конференції ACM SIGCOMM 2005 року під публікацією "Spray and Wait: Ефективна схема маршрутизації мобільних мереж, що перериваються постійно". Spray and Wait досягає ефективності використання ресурсів, встановлюючи чітку верхню межу щодо кількості копій на повідомлення, дозволених у мережі.

Протокол Spray and Wait складається з двох фаз: фази «розпилення» та фази очікування. Коли в системі створюється нове повідомлення, до даного повідомлення додається число L із зазначенням максимально допустимих копій повідомлення в мережі. Під час фази розпилення джерело повідомлення відповідає за "розбризкування" або доставку однієї копії різним "реле" L. Коли ретрансляція отримує копію, вона переходить у фазу очікування, де ретранслятор просто утримує певне повідомлення, поки безпосередньо не зустрінеться призначення.

Існує дві основні версії Spray і Wait: ванільна (комп'ютерне програмне забезпечення, а іноді й інші пов'язані з обчислювальною системою, такі як комп'ютерне обладнання або алгоритми, називаються ванільними, коли вони

не налаштовуються з їх первісної форми, тобто вони використовуються без будь-яких налаштувань або оновлень, застосованих до них. Термін походить від традиційного стандартного аромату морозива, ванілі) і бінарна.

Дві версії ідентичні, за винятком того, як копії L досягають L різних вузлів під час фази розпилення. Найпростіший спосіб досягти цього, відомий як версія ванілі, полягає в тому, щоб джерело передавало єдину копію повідомлення першим L-1 вузлам, з якими воно стикається після створення повідомлення.

Друга версія – бінарна. Тут джерело починається, як і раніше, з копій L. Потім воно передає (L / 2) своїх копій на перший вузол, з яким воно стикається. Кожен з цих вузлів потім передає половину загальної кількості примірників, які вони мають, до майбутніх вузлів, з якими вони зустрічаються, що не мають копій повідомлення. Коли зрештою вузол видає всі свої копії, крім однієї, він переходить у фазу очікування, де чекає можливості прямої передачі з пунктом призначення. Перевага Binary Spray and Wait полягає в тому, що повідомлення розповсюджуються швидше, ніж ванільна версія. Насправді автори доводять [3], що Binary Spray and Wait є оптимальним з точки зору мінімальної очікуваної затримки між усіма схемами Spray and Wait.

Bubble Rap вперше вводить розуміння мобільності людини в дизайн DTN. Вони вивчають соціальні структури між пристроями та використовують їх у розробці алгоритмів переадресації для кишенькових комутованих мереж (PSN). Експерименти зі слідами реального світу виявляють, що взаємодія між людьми неоднорідна як з точки зору вузлів, так і груп чи спільнот. Відповідно до цього висновку, вони пропонують Bubble Rap, алгоритм переадресації на основі соціальних мереж, щоб значно підвищити ефективність пересилання в порівнянні з алгоритмами PROPHET на основі історії та на основі соціальних алгоритмів SimBet. Цей алгоритм також показує, як його можна реалізувати розподіленим способом, що демонструє, що він може застосовуватися у децентралізованому середовищі

PSN.

CafRep – це повністю локалізований адаптивний протокол переадресації та реплікації з контролем перевантажень та уникненням, щоб увімкнути мобільну соціальну структуру у різнорідних DTN. CafRep використовує комбіновані соціальні, буферні та затримані показники для переадресації та реплікації повідомлень про перевантаженість, які максимально збільшують коефіцієнт доставки повідомлень та доступність вузлів, зводячи до мінімуму затримку та швидкість втрати пакетів у періоди підвищення рівня перевантаженості. В основі CafRep – комбінована відносна евристика, орієнтована на корисні програми, яка дозволяє дуже швидко адаптуватися до політики переадресації та реплікації, керуючи виявленням та завантаженням перевантажених частин мережі та адаптацією швидкостей відправки / переадресації на основі прогнозів ресурсів та контактів.

RACOD – маршрутизація за допомогою оптимізації колонії Ant в DTN запроваджує вивчення шляхів за допомогою ACO, а також розумно вирішує, яке повідомлення відкинути, а яке повідомлення передати. У DTN немає точних знань про призначення, і тому потрібно розповсюджувати повідомлення в усіх напрямках для пошуку місця призначення. ACO допомагає у блуканні та ефективній побудові найкоротшого шляху. Протокол використовує полегшені повідомлення під назвою “мураха” для побудови найкоротших шляхів, рух мурашок в ACO можна відобразити за допомогою поширення повідомлень, які копіюються в DTN та шукають їх призначення. Більше того, цей протокол також дає кращу техніку управління буфером, він впроваджує технологію сортування 3-х напрямків, яка допомагає викидати повідомлення старого або шкідливого характеру і, таким чином, зменшує накладні витрати на буфер.

2.5 Маршрутизація на основі експедирування

DTLSR реалізований у впровадженні DTN2 BP стандарті і має на меті

забезпечити прямолінійне розширення маршрутизації стану зв’язку. У DTLSR повідомлення про стан посилань надсилаються як у OLSR, але посилання, які вважаються "битими", не видаляються з графіка негайно. Натомість "биті" посилання вирівнюються за рахунок збільшення їх показників, поки не буде досягнуто деякого максимуму, після чого вони будуть вилучені з графіка. Метою цього є змусити дані продовжувати надходити по шляхах, які раніше підтримувались, сподіваючись, що в майбутньому вони знову будуть підтримуватися.

Протокол SABR – це розширення маршрутизації контактних графіків, яке прагне запропонувати рішення для маршрутизації для широкого спектру сценаріїв, що включають як заплановані, так і виявлені з'єднання. Для запланованого режиму підключення SABR використовує "контактний план", який надається мережевим управлінням, що описує поточний зв’язок та майбутній графік зв'язку. Потім SABR приймає рішення про переадресацію на основі метрики раннього часу прибуття, де пакети маршрутизируються через графік підключення, що залежить від часу. SABR використовує історичну контактну інформацію та сусідське відкриття для вирішення маршрутизації через позапланові посилання. Протокол SABR стандартизується Консультативним комітетом з космічних даних.

Більшість існуючих протоколів маршрутизації та передачі даних для DTN передбачають, що мобільні вузли охоче беруть участь у доставці даних, діляться своїми ресурсами один з одним та дотримуються правил, що лежать в основі мережевих протоколів. Тим не менш, раціональні вузли в реальних сценаріях мають стратегічну взаємодію і можуть проявляти егоїстичну поведінку через різні причини (наприклад, обмеження ресурсів, відсутність інтересу до даних або соціальні переваги). Наприклад, якщо вузол має обмежені ресурси акумулятора або витрати на пропускну здатність мережі, доставлені операторами мобільних мереж, великі, не буде охоче передавати дані для інших, поки не будуть надані відповідні стимули. Тим часом шкідливі вузли можуть різними способами атакувати мережу, щоб порушити

нормальну роботу процесу передачі даних. Наприклад, супротивник може скидати отримані повідомлення, але створювати підроблені метрики маршрутизації або неправдиву інформацію з метою або залучення більшої кількості повідомлень, або зменшення ймовірності її виявлення. Ця проблема стає більш складною, коли деякі зловмисники збитків збільшують свої показники, щоб обманути системи виявлення атак. Однак поводження з некооперативною поведінкою мобільних вузлів у DTN є дуже складним через модель розподіленої мережі та переривчастий доступ вузлів до централізованих органів влади.

2.6 Особливості використання у системі

Протокол DTN використовується у системі, як основний спосіб комунікації та передачі даних. За замовчуванням кожний компонент системи конфігурується як окремий мережевий bundle.

Лістинг 2.1 – Конфігурація об’єкта у мережі DTN

```
UID_LISTEN_HOST=localhost
UID_LISTEN_PORT=3001
UID_NAME=object2001
UID_NETWORK_NAME=bundle1

MAIN_SEND_HOST=localhost
MAIN_SEND_PORT=4000

NEIGHBOUR_SEND_HOST=localhost
NEIGHBOUR_SEND_PORT=3002
NEIGHBOUR_NETWORK_NAME=bundle2
```

Кожний набір даних, помилок або попереджень у системі повинен ретельно фіксуватися, мати окремий алгоритм зберігання та підтримувати функцію резервного копіювання та відновлення. При першому запуску об’єкта починаєтеся процес ініціалізації з’єднання з базою даних, котра виконує функцію зберігання логів (попереджень та помилок).

Лістинг 2.2 – Ініціалізація бази даних зберігання попереджень та помилок

```
func Init() model.LogsDataStore {
	DBHost := "localhost"
	DBPort := "8086"

	db, err := etcd.NewClient(etcd.Config{Addr: fmt.Sprintf("http://%s:%s", DBHost, DBPort)})
	if err != nil {
		log.Fatalf("[ERROR] logs db init failed - (%s)", err.Error())
		return nil
	}

	_, _, err = db.Ping(3 * time.Second)
	if err != nil {
		log.Fatalf("[ERROR] logs db ping failed - (%s)", err.Error())
	}

	return &model.LogsDB{RW: db}
}
```

У ролі бази для зберігання попереджень та помилок системі служить БД ETCD. ETCD – це стабільно розподілене сховище інформації у вигляді ключ-значення, яке забезпечує надійний спосіб зберігання даних, доступ до котрих може отримати будь-яка розподілена система або кластер з обчислювальних машин. ETCD безпечно обробляє помилки та у виникненні аварійної ситуації переключає основний мережевий вузол. Для системи, що розробляється основною особливістю ETCD є можливість безпечно обробляти втрату зв’язку та здійснення резервної копії з наступною синхронізацією.

Кожна помилка, попередження або інший “лог” повинні фіксуватися у єдиному форматі, котрий буде зрозумілим для будь-якого об’єкту у системі. У якості такого формату був обраний схожий до системних помилок у TCP протоколі підхід. Кожна помилка має свій статус-код та опис (рис.2.3). Розглянемо основні помилки, котрі об’єкт може передати, та їх текст опису. “Термін служби закінчився” – такого типу помилка виникає у випадку закінчення терміну служби мережевого bundle. За замовчування термін служби – 24 години, якщо термін не був сконфігурований у процесі ініціалізації об’єкта. “Відправлення через одно-канальний зв’язок” – помилка

виникає, коли об’єкту не вдається знайти об’єкт-сусіда. Така система з початку сконфігурована неправильно та призводить до односторонньої комунікації, що небезпечно для даних при втраті зв’язку. “База даних виснажена” виникає у випадку, коли закінчилася постійна пам’ять для зберігання даних або коли об’єкт намагається дістати та записати однаковий об’єкт. Кожна помилка має специфічний статус-код, що допомагає об’єктам у комунікації.

0	Відсутня додаткова інформація
1	Термін служби закінчився
2	Відправлення через одноканальний зв'язок
3	Відправлення скасоване
4	База даних виснажена
5	Недоступний ідентифікатор кінцевої точки призначення
6	Невідомий маршрут до місця призначення
7	Немає своєчасного контакту з наступним вузлом на маршруті
8	Невідомий блок
9	Ліміт переходів перевищено
10	Моніторинг трафіку (наприклад, звіти про стан)
(інші)	Зарезервовані для подальшого використання

Рисунок 2.3 – Статус коди помилок та попереджень

На додаток класичної для DTN маршрутизації система також використовує потоковий алгоритм (flooding) при передачі обчислювальних даних. Даний протокол дозволяє розв’язати завдання обробки даних, котрі надходять послідовно та у великій кількості, наприклад, кожен об’єкт у розробленій системі, надсилає обчислювальну інформацію кожні 2 секунди. Окрім цього, такий підхід до відправки може бути корисним при використанні машинного навчання, де кожен об’єкт может бути “навченим” за один прохід даних (у такому випадку система може використовувати

метод передбачуваного хешування).

Підсумовуючи, система має реалізований функціонал обробки критичних ситуацій шляхом зберігання та аналізу помилок/попереджень, а потоковий алгоритм дозволяє ретельно обробити кожну ситуацію незважаючи на її походження.

2.7 Технічне завдання

Система повинна містити щонайменшне 3 компонента – 2 моделі віддалених керованих об’єктів та 1 сервіс керування. Компоненти повинні використовувати протокол мережі DTN для комунікацій та потоковий алгоритм.

Функціонал моделі віддаленого керованого об’єкта:

– проведення обчислювальної роботи (у випадку розробленої системи – збирання температури);

– відправлення даних до сервісу керування за інтервалом в секундах;

– при втраті зв’язку зі сервісом керування розпочати комунікацію через об’єкт-сусіда;

– зберігання помилок та попереджень, їх резервні копії.

Важливо зазначити, що кодова база моделі віддаленого об’єкта повинна бути однакова, тобто зі створенням наступного об’єкта повинна змінюватися тільки конфігурація та обчислювальні властивості. Таким чином, моделі будуть мати можливість масштабування до нескінченої кількості.

Функціонал керованого об’єкта:

– зберігання конфігурації моделей віддалених об’єктів;

– зберігання статистичних обчислювальних даних;

– комунікація з об’єктами одно-канально та за допомогою об’єктів-медіаторів (сусідів);

– зберігання помилок та попереджень, їх резервних копій;

– ідентифікація одно-канальних та зв’язків через медіатори.

Система повинна безпечно обробляти ситуації втрати зв’язку (наприклад, коли один із супутників переходить в тінь та втрачає зв’язок зі сервісом керування на Землі). В такому разі, супутник повинен розпочати зв’язок за допомогою супутника-медіатора. Важливим моментом є підтримувати зв’язок якомога довше та не втратити дані з космосу, адже кожний байт даних з космосу – на вагу золота.

3 РЕАЛІЗАЦІЯ СИСТЕМИ УПРАВЛІННЯ ВІДДАЛЕНИМ ОБ’ЄКТОМ У DTN МЕРЕЖІ

3.1 Програмні компоненти реалізації системи

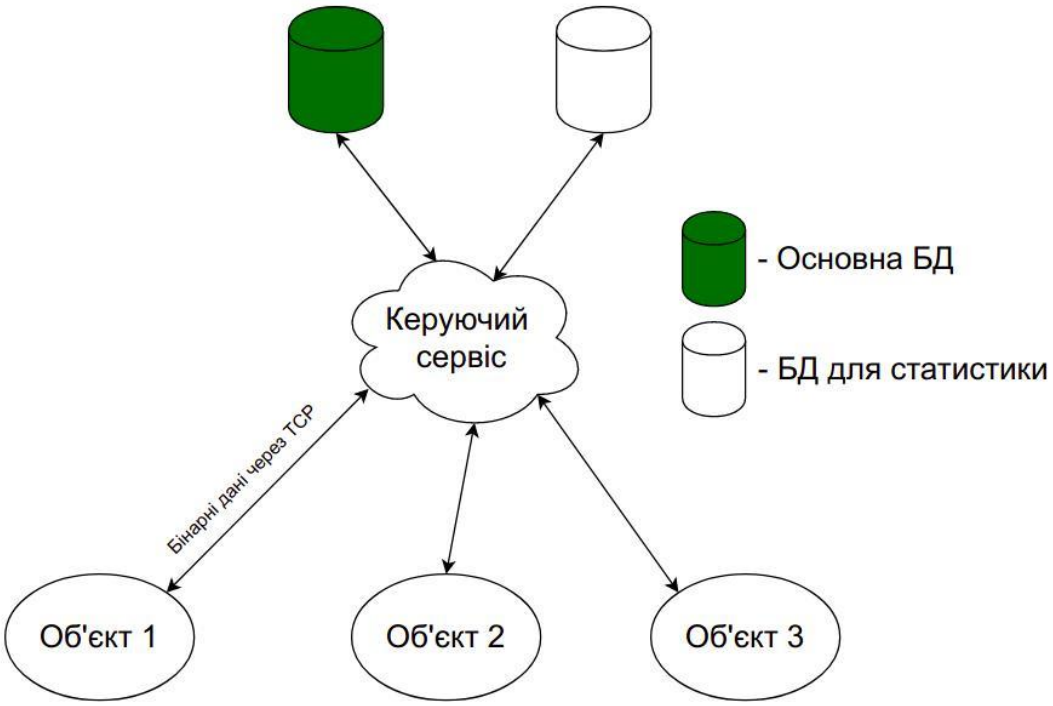

Рисунок 3.1 – Попередня архітектура системи

На рис.3.1 зображена архітектура попередньої реалізації системи реального часу зі сервісом керування та дослідницькими об’єктами, мета котрих була зібрати, проаналізувати та відправити дані до сервісу. Попередній керуючий сервіс мав зв’язок з об’єктами за допомогою TCP протоколу, дані передавалися у чистому (raw) бінарному форматі. Кожна відповідь від сервісу містила в собі спеціальне поле “Action” (Дія), базуючись котрою об’єкт приймав наступне рішення.

Лістинг 3.1 – Класи запросу та відповіді

```
type Request struct {
Type string
Data interface{}
}
type Response struct {
Type   string
Action string
}
```

Реалізація попередньої системи мала ряд переваг та недоліків. План розробки поточної системи мав на меті виправлення недоліків, адже поточна система являється критичною. Серед недоліків попередньої системи важливо відзначити:

– неможливість використання системи у віддаленому контексті через використання TCP протоколу;

– відсутність обробки обривів зв’язку;

– відсутність єдиного протоколу обробки помилок та попереджень;

– відсутність резервного копіювання статистичних даних;

– відсутність зберігання помилок та попереджень об’єктів.

Неможливість використання системи у віддаленому контексті виникла з причини використання TCP протоколу як основного підходу до комунікацій у системі. TCP протокол не має підтримки багато-канального зв’язку та передбачає 2 вузла комунікації. Таким чином, якщо з’являються обриви на лінії, вузли зупиняють обмін даними та “чекають” поки зв’язок буде відновлено, що відкриває наступний недолік – відсутність обробки обривів. Такі недоліки повинні бути виправлені у поточній системі, адже вона має характеристику критичної. З іншого боку – єдиний протокол обробки помилок та попереджень дозволяє користувачам системи швидко та якісно ідентифікувати місце та причину проблеми. Така особливість має значну перевагу у критичних системах, а особливо – у космічному контексті. Говорячи, про космічний контекст, кожне “слово”, кожен набір даних із космосу є дуже важливим для науки та техніки, а тому такий недолік, як

“Резервне копіювання обчислювальних даних” повинен бути виправлений.

Програмні компоненти поточної системи включають (рис.3.2):

– модель віддаленого об’єкта, у космічному контексті – супутник, з назвою *object2000*;

– модель віддаленого об’єкта, у космічному контексті – супутник, з назвою *object2001*;

– сервіс керування – система управління віддалених об’єктів, назва – *air-management*.

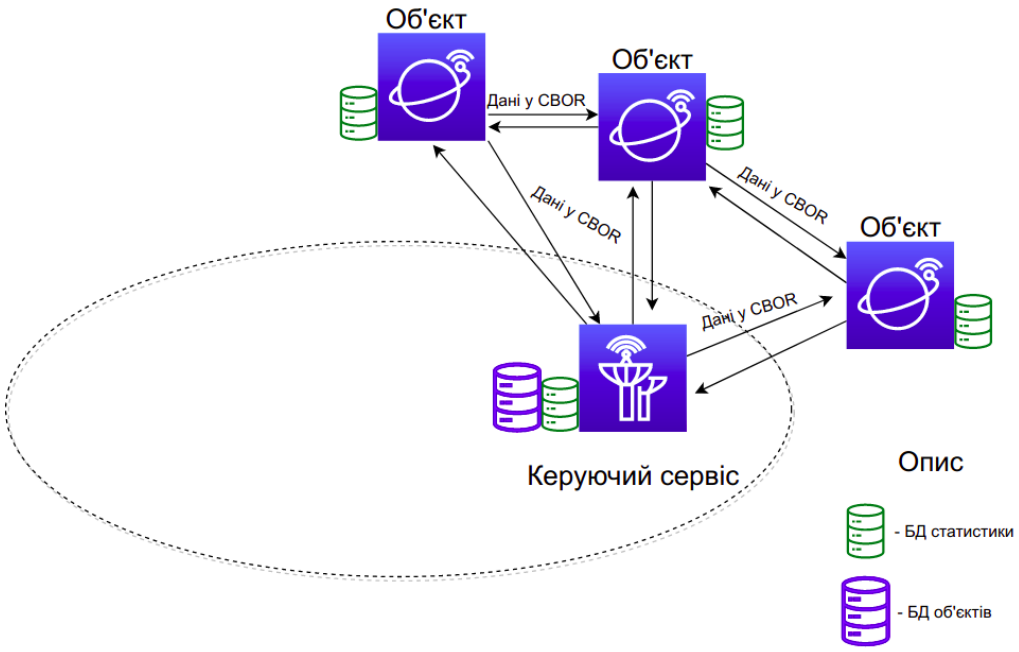

Рисунок 3.2 – Поточна архітектура реалізованої системи

Важливо зазначити, що за технічним завданням моделі віддалених об’єктів повинні мати:

– статистичну базу даних для резервного зберігання обчислювальних результатів;

– базу даних зберігання помилок та попереджень;

У свою чергу сервіс керування повинен мати:

– статистичну базу даних для зберігання даних від усіх об’єктів;

– основну базу даних конфігурацій об’єктів;
– базу даних для помилок та попереджень.

Так як програма складається з множини компонентів, то в якості архітектури були обрані мікросервіси. Мікросервіси – це новий термін на галасливих вулицях розробки ПЗ. І хоча розробники зазвичай досить насторожено ставляться до всіх подібних новинок, конкретно цей термін описує стиль розробки ПЗ, який системні архітектори знаходять все більш і більш привабливим. За останні кілька років було безліч проектів, що використовують цей стиль, і результати були досить позитивними. Настільки, що для більшості розробників ПЗ цей стиль стає основним стилем розробки. На жаль, існує не так багато інформації, яка описує, чому ж є мікросервіси і як застосовувати їх.

Якщо коротко, то архітектурний стиль мікросервісов – це підхід, при якому єдиний додаток будується як набір невеликих сервісів, кожен з яких працює у власному контексті та спілкується з іншими використовуючи прості механізми, як правило HTTP. Ці сервіси побудовані навколо бізнес-потреб і розгортаються незалежно від використання повністю автоматизованого середовища. Існує абсолютний мінімум централізованого управління цими сервісами. Самі по собі ці сервіси можуть бути написані на різних мовах програмування і використовувати різні технології зберігання даних.

Для того, щоб описати стиль мікросервісів, найкраще порівняти його з монолітним (monolithic style) додатком, побудованому як єдине ціле. Enterprise додатки часто включають три основні частини: графічний інтерфейс призначений для користувача (що складається в основному з HTML сторінок та javascript-анімацій), базу даних (як правило реляційну, з множиною таблиць) і сервер. Серверна частина обробляє HTTP запити, виконує доменну логіку, запитує і оновлює дані в БД, заповнює HTML сторінки, які потім відправляються браузеру клієнта. Будь-яка зміна в системі призводить до перекомпіляції і розгортанню нової версії серверної

частини програми.

Монолітний сервер – досить очевидний спосіб побудови подібних систем. Вся логіка з обробки запитів виконується в єдиному процесі, при цьому неможливо скористатися наявними можливостями мови програмування для поділу додатків на класи, функції і namespace-и. Є можливість запускати і тестувати додаток на машині розробника і використовувати стандартний процес розгортання для перевірки змін перед публікацією їх в роботу. Існує можливість масштабувати монолітні додатки горизонтально шляхом запуску декількох фізичних серверів з балансувальником навантаження (load balancer).

Монолітні додатки можуть бути успішними, але все більше людей розчаровуються в них, особливо в світлі того, що все більше додатків розгортаються в хмарних сервісах. Будь-які зміни, навіть самі невеликі, вимагають перекомпіляції і розгортанню всього моноліта. З плином часу, стає важче зберігати хорошу модульну структуру, зміни логіки одного модуля мають тенденцію впливати на код інших модулів. Масштабувати доводиться весь додаток цілком, навіть якщо це потрібно тільки для одного модуля цього додатка (рис.3.3).

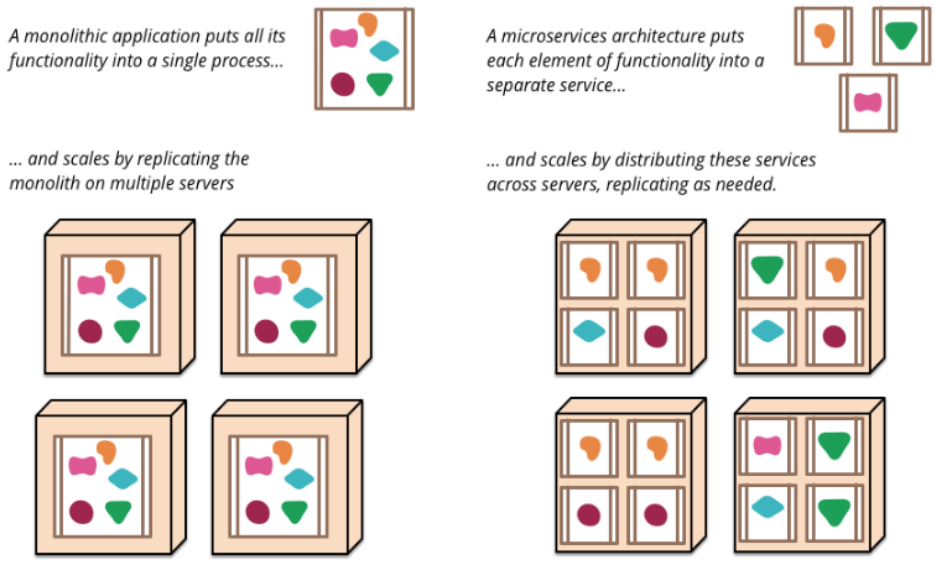

Рисунок 3.3 – Схема масштабування мікросервісів

З вищевказаних вимог можна зробити висновок, що розроблена система буде високонавантаженою. Доказ цьому дуже простий: якщо в системі існують 60 керованих об’єктів які одночасно кожні 5 секунд починають відправляти дані на центральний вузол, то виникає проблема коректного розподілу ресурсів. Тому, було прийнято рішення використовувати технологію паралельного програмування.

Розробка програмного продукту була виконана за допомогою такої мови програмування, як Go (Golang). Для того, щоб зрозуміти суть цієї технології і чому вона була обрана в якості основи для цього проекту було проведено спеціальне дослідження, результати якого описані нижче.

Почнемо з того, що останні роки закон Мура не повністю підтверджується (рис. 3.4). Перший процесор Pentium 4 з тактовою частотою 3.0 ГГц був представлений ще в 2004 році компанією Intel. Сьогодні, Apple Macbook Pro 2020 має тактову частоту 2.9 ГГц. Таким чином, близько десятиліття немає особливого прогресу чистої обчислювальної потужності. Нижче на графіку, є можливість побачити порівняльний графік зростання обчислювальної потужності за часом.

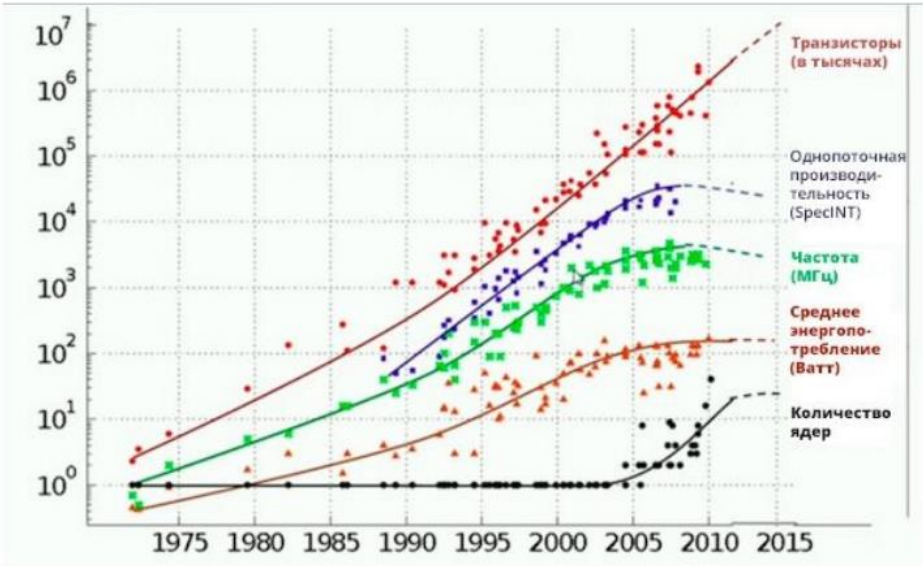

Рисунок 3.4 – Графік результату роботи закону Мура

За графіком можна зрозуміти, що однопоточні продуктивність і частота процесора залишалася стабільною майже десятиліття. Рішення додати більше транзисторів – це не найкраще рішення. Так відбувається тому, що в більш менших масштабах починають проявлятися квантові властивості об'єкта (такі як тонелювання), а додавання більшої кількості транзисторів дійсно може привести до додаткових витрат і кількість транзисторів, яке можна додати за один долар стрімко починає падати.

Тому, для вирішення проблеми вище:

– додано більше ядер – на поточний день є 4-х і 8-ми ядерні процесори;

– започатковано гіперпоточність;

– додано більше кеша в процесор для збільшення продуктивності.

Але рішення вище також мають свої обмеження. Не можна додати більше кеша в процесор, оскільки кеш має фізичні обмеження: чим більше кеш, тим повільніше він стає. Додавання більшої кількості ядер в процесор теж має свою ціну. До того ж, це неможливо робити це до нескінченності. Всі ці багатоядерні процесори можуть запускати безліч потоків одночасно і це надає картині багатозадачність (рис.3.5).

Тобто, не можна покладатися на поліпшення апаратної частини, значить єдиний вихід з цієї ситуації – це більш ефективне програмне забезпечення. Але, на жаль, сучасні мови програмування не так ефективні. Як було зазначено вище, виробники апаратного обладнання додають все більше ядер на процесор, щоб збільшити його продуктивність. Усі дата-центри запускаються на цих процесорах і варто очікувати збільшення кількості ядер в найближчі роки. Більш того, сучасні програми використовують множину мікросервісів для підтримки з'єднань з базою даних, черги повідомлень і кешування. Таким чином, програмне забезпечення та мови програмування повинні легко підтримувати багатозадачність (concurrency) і вони повинні бути масштабованими при

збільшенні кількості ядер.

Рисунок 3.5 – Графік швидкості послідовної роботи системи

Але більшість сучасних мов програмування (такі, як Java, Python) прийшли з однопоточному середовища розробки 90-х. Хоча більшість цих мов підтримують багатопоточність, але реальна проблема пов'язана з одночасним виконанням, блокуванням потоку, станом гонки і тупиковою ситуацією. Ці речі ускладнюють створення багатопотокового додатку на вищеназваних мовами.

Наприклад, створення нового потоку в Java неефективно використовує пам'ять. Оскільки кожен потік споживає приблизно 1 МБ пам'яті з купи (динамічно розподіляє пам'яті. - прим. Перекл.) І в підсумку, якщо запустити тисячі потоків, то вони нададуть колосальний тиск на пам'ять і можуть викликати завершення роботи програми через її нестачу. До того ж, якщо стоїть завдання, щоб два або більше потоки могли спілкуватися між собою, то зробити це буде досить важко.

З іншого боку, Go був випущений в 2009 році, коли багатоядерні процесори були вже доступні. Ось чому Go був створений з урахуванням багатозадачності (concurrency). Go використовує горутіни замість потоків. (рис. 3.6). Вони споживають майже 2кб пам'яті з купи. Тому, є можливість запускати хоч мільйони горутін в будь-який час.

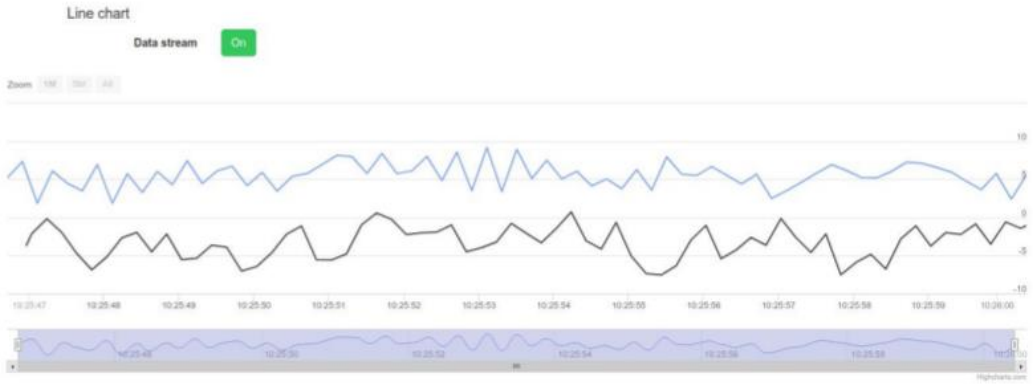

Рисунок 3.6 – Графік швидкості паралельної роботи системи

Горутіни також мають такі переваги:

– використовуються сегментовані стеки – це означає, що горутіни будуть використовувати пам'ять тільки в разі потреби;

– горутіни мають більш швидкий час запуску, ніж потоки;

– горутіни поставляються з вбудованими примітивами для безпечного обміну даними між собою (канали);

– горутіни дозволяють уникнути необхідності вдаватися до блокування м'ютексів при спільному використанні структур даних;

– також, горутіни і потоки ОС не мають зіставлення 1: 1. Одна горутіна може запускатися в безлічі потоків. Горутіни об'єднані (multiplexed) в малу кількість потоків ОС.

3.2 Програмні інструменти

З метою імплементації розподіленої системи була обрана мова програмування Golang. Більш цього, технічне завдання полягає у швидкій конфігурації та єдиній кодовій базі для віддалених об’єктів, Golang у такому контексті є однозначно доцільним.

Golang – компільована, імперативна мова програмування, що має засоби для паралельних обчислень та засоби віддаленого керування

пакунками. Синтаксис мови базується на елементах синтаксису мови програмування C, але має декілька запозичень з Python. Мова програмування Golang розроблялась з оглядкою на багато-поточне програмування та ефективний запуск на багатоядерних системах, що дозволяє безпечно та ефективно виконувати паралельні обчислення та взаємодію між паралельними процесами. Наприклад, модель віддаленого об’єкту виконує процес збирання, зберігання та передачі даних паралельно, що дозволяє ефективно використовувати ресурси та з легкістю імплементувати потоковий алгоритм.

Технічне завдання моделей віддалених об’єктів передбачає використання двох баз даних – статистичної резервної та бази даних помилок та попереджень.

У якості статистичної резервної була розроблена імплентація бази даних InfluxDB. InfluxDB – база даних так званих часових рядів (time-series). Особливостями та перевагами є відсутність зовнішніх залежностей, кодова база на Golang (дозволяє з легкістю адаптувати до реалізації системи), SQL-подібна мова запитів, що має вбудовані функції для роботи з часом, а також структуру даних вимірів (measurements) – у контексті розробленої системи виміри температури, серій (series) та точок даних (points). Точка має декілька пар ключ-значення та мітку часу (timestamp). Таким чином, модель віддаленого об’єкту буде зберігати температуру за міткою часу у якості резерву на випадок обриву зв’язку. Такий процес можна спостерігати у режимі реального часу за допомогою панелі моніторингу, приклад якої зображений на рис.3.3.

Технологія бази даних ETCD була обрана для зберігання помилок та попереджень. За технічним завдання кожний “лог” повинен бути оброблений та збережений ETCD у вигляді ключ-значення. Як було зазначено, ETCD – це стабільно розподілене сховище інформації у вигляді ключ-значення, яке забезпечує надійний спосіб зберігання даних, доступ до котрих може отримати будь-яка розподілена система або кластер з обчислювальних

машин, як зазначено на рис.3.4.

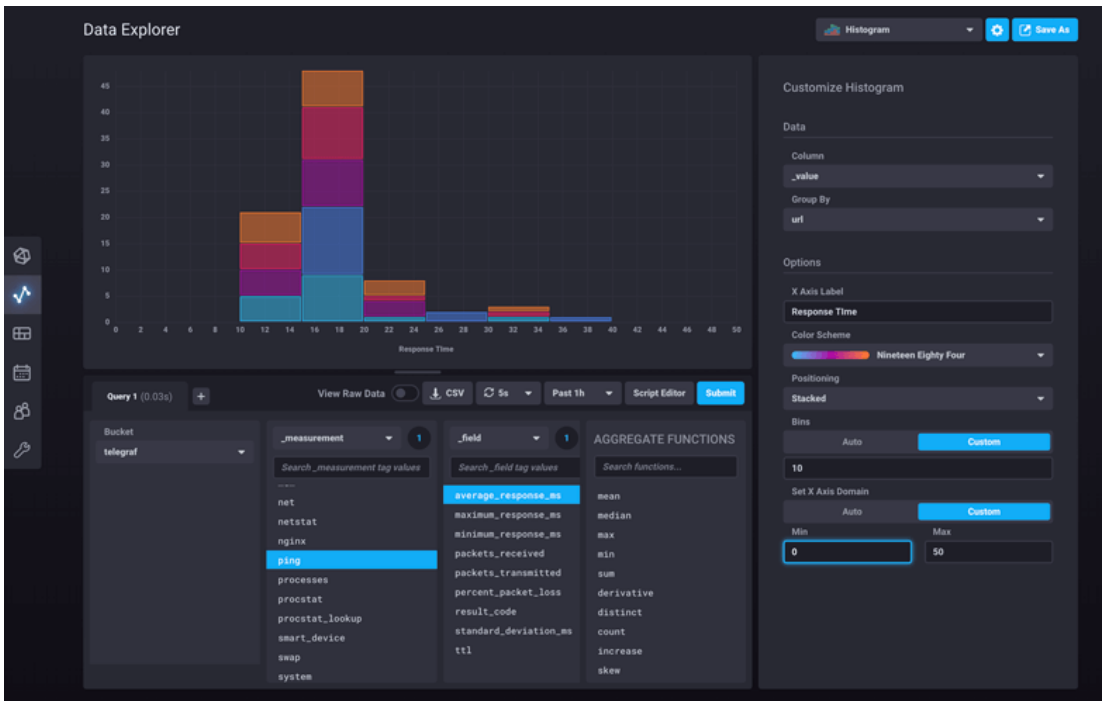

Рисунок 3.7 – Панель моніторингу статистичних даних моделі віддаленого об’єкта в Influx

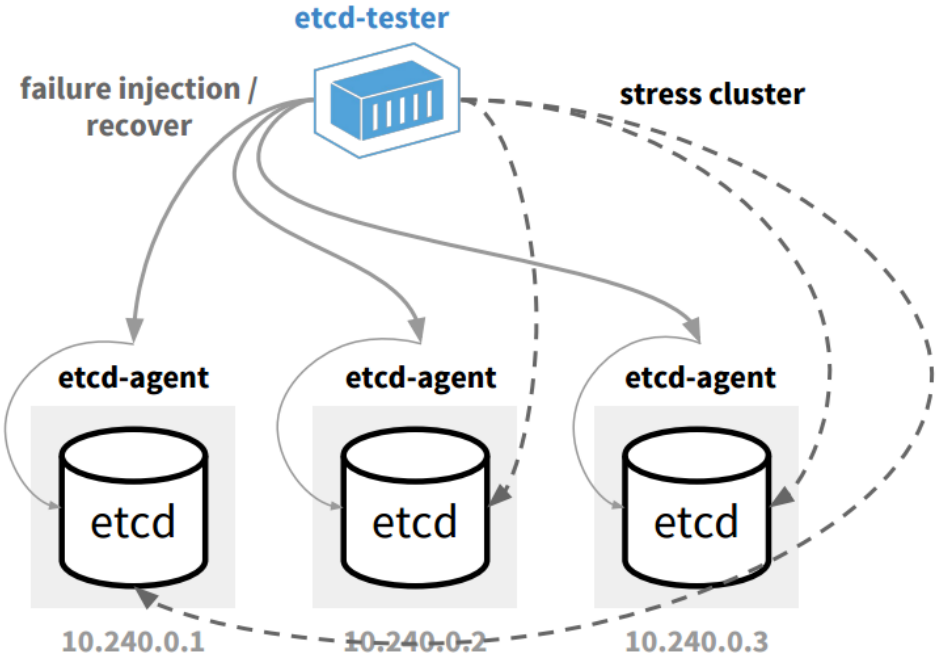

Рисунок 3.8 – Схема аварійного зберігання помилок та попереджень в ETCD

У якості основної бази даних для збереження конфігурацій об’єктів, керуючий сервіс використовує MySQL. Поточна реалізація має декілька таблиць – *objects* та *objects_config.* Для зручного налаштування користувачем системи може бути запущений sql-скрипт, що ініціалізує вищезгадані таблиці.

Лістинг 3.2 – Скрипт ініціалізації основної БД

```
CREATE DATABASE  IF NOT EXISTS `air-manage` /*!40100 DEFAULT CHARACTER SET utf8 */;
USE `air-manage`;

DROP TABLE IF EXISTS `objects`;
CREATE TABLE `objects` (
	`object_id` VARCHAR(255) NOT NULL,
	`name` VARCHAR(255) NOT NULL,
	`network_name` VARCHAR(255) NOT NULL,
	`active` tinyint(1) NOT NULL,
	`mainActive` tinyint(1) NOT NULL,
	`created_at` TIMESTAMP NOT NULL,
	PRIMARY KEY (`object_id`)
);

DROP TABLE IF EXISTS `objects_config`;
CREATE TABLE `objects_data` (
	 `object_id` VARCHAR(255) NOT NULL,
	`mac` VARCHAR(255) NOT NULL,
	`serialNumber` VARCHAR(255) NOT NULL,
	`created_at` TIMESTAMP NOT NULL,
	PRIMARY KEY (`created_at`, `object_id`)
);
```

3.3 Конфігурація системи

Конфігурація моделі віддаленого об’єкту *object2000* відбувається за допомогою .env файлу, що містить налаштування середовища та самого об’єкта.

Лістинг 3.3 – Конфігурація об’єкту object2000 через .env файл

```
UID_LISTEN_HOST=localhost
```

```
UID_LISTEN_PORT=3000
UID_NAME=object2000
UID_NETWORK_NAME=bundle1

MAIN_SEND_HOST=localhost
MAIN_SEND_PORT=4000

NEIGHBOUR_SEND_HOST=localhost
NEIGHBOUR_SEND_PORT=3001
NEIGHBOUR_NETWORK_NAME=bundle2
```

У даному контексті UID – спеціальний ідентифікатор, що допомагає ідентифікувати об’єкт як bundle у DTN мережі. Network name (ім’я в мережі) для object2000 буде bundle1. Конфігураційні параметри з префіксом MAIN позначають ідентифікацію керуючого сервісу, а префікс NEIGHBOUR – об’єкта-сусіда, за допомогою якого буде здійснюватися зв’язок у разі обриву.

Конфігурація наступної моделі віддаленого об’єкту object2001 буде аналогічною до object2000, адже за технічним завданням – об’єкти повинні мати однакові кодові бази, але різну обчислювальну мету.

Лістинг 3.4 – Конфігурація об’єкту object2001 через .env файл

```
UID_LISTEN_HOST=localhost
UID_LISTEN_PORT=3001
UID_NAME=object2001
UID_NETWORK_NAME=bundle2

MAIN_SEND_HOST=localhost
MAIN_SEND_PORT=4000

NEIGHBOUR_SEND_HOST=localhost
NEIGHBOUR_SEND_PORT=3002
NEIGHBOUR_NETWORK_NAME=bundle3
```

Таким чином, object2001 має ім’я в мережі – bundle2, що означає object2001 є медіатор між object2000 та керуючим сервісом. У свою чергу, для object2001 сусідом будет виступати віддалений об’єкт з ім’ям в мережі bundle3.

Для запуску моделей віддалених об’єктів також потребується запуск 2 баз даних на середовищі кожного з об’єктів. Швидким способом запуску є

використання docker-середовища. Docker – спеціальний інструмент для запуску та управління ізольованим Linux-контейнером. У контексті запуску баз даних системи Docker дозволяє не перейматись вмістом контейнера та запустити довільні процеси баз даних і ізолювати їх від логіки сервісів реалізованої системи. Також Docker реалізований за допомогою мови програмування Golang, що сприяє швидкої інтеграції.

Лістинг 3.5 – Запуск Influx за допомогою Docker

```
docker run -p 8086:8086 \
    -v $PWD:/var/lib/influxdb \
    influxdb
```

Лістинг 3.6 – Запуск ETCD за допомогою Docker

```
docker run -p 8086:8086 \
    -v $PWD:/var/lib/etcd \
    etcd
```

Важливо наголосити, що параметр -v $PWD:/var/lib/* є обов’язковим, адже система потребує доступу до постійної пам’яті.

Конфігурація керуючого сервісу відбувається також за допомогою .env файлу, що містить параметри середовища.

Лістинг 3.7 – Конфігурація сервісу керування

```
PORT=5000

MYSQL_USER=test
MYSQL_PASSWORD=password
MYSQL_HOST=localhost
MYSQL_PORT=3306
MYSQL_NAME=air-manage

INFLUX_USER=test
INFLUX_PASSWORD=password
INFLUX_HOST=localhost
INFLUX_PORT=8089
INFLUX_NAME=air-manage
```

```
TRANSPORT_RECEIVE_HOST=localhost
TRANSPORT_RECEIVE_PORT=4000

TRANSPORT_SEND_HOST=localhost
TRANSPORT_SEND_PORT=4040
```

Порт 5000 відкриває доступ до HTTP API, за допомогою котрого здійснюється симуляція обривів зв'язку. Конфігураційні параметри з префіксами MYSQL та INFLUX містять необхідні налаштування для баз даних. У свою чергу, керуючий сервіс має 2 інтерфейси – прийому та відправлення даних, але сконфігурований як 1 bundle (Main). Це дозволяє ідентифікувати керуючий сервіс та заборонити репліки (у майбутньому такі налаштування можуть бути відкинуті для масштабування). Запуск баз даних також може бути здійснений за рахунок інструменту Docker.

Останнім пунктом конфігурації системи є запуск DTN образу, що зможе виконати маршрутизацію та з'єднати кожен з об'єктів у мережі (bundle). Для цього може бути використаний також образ Docker та контейнер. Зазначимо, що для локального запуску та моделювання достатньо контейнеру, але для повного тестування бажано запустити повноцінний механізм маршрутизації на кожному середовищі для першого, другого об'єктів та керуючого сервісу.

Лістинг 3.8 – Запуск DTN маршрутизації за допомогою Docker

```
docker run -p 6000:6000 \
    -v $PWD:/var/lib/dtn7 \
    dtn7-routing
```

3.4 Результат роботи системи

Запуск виконується за допомогою Makefile та розпочинається з керуючого сервісу.

Лістинг 3.9 – Makefile з командами управління запуску керуючого сервісу

```
export CGO_ENABLED=0
SVC_NAME = management
BUILD_DIR=.build
all: test build run
verify:
go vet ./...
test: verify
go test ./...
get-deps:
env GIT_TERMINAL_PROMPT=1 go mod download
build:
go build -ldflags="-s -w" -o $(BUILD_DIR)/$(SVC_NAME) .
run:
env $$(cat $(SVC_NAME).env) $(BUILD_DIR)/$(SVC_NAME)
help: Makefile
@echo " Choose a command run in \033[32m"$(SVC_NAME)"\033[0m:"
@sed -n 's/^##//p' $< | column -t -s ':' |  sed -e 's/^/ /'
```

Спочатку виконується команда “test”, котра запускає всі тести проекту, якщо хоча б один тест не працює – керуючий сервіс не перейде до стадії збірки. Наступна стадія “build” виконує збірку сервісу, а його бінарний файл зберігає у “.build” директорію. Якщо “build” був успішним, наступна стадія – “run”, безпосередньо запуск сервісу. Важливо зазначити, що “run” виконується з експортом згаданих у попередньому розділі конфігураційних .env файлів, таким чином сервіс буде сконфігурований та запущений.

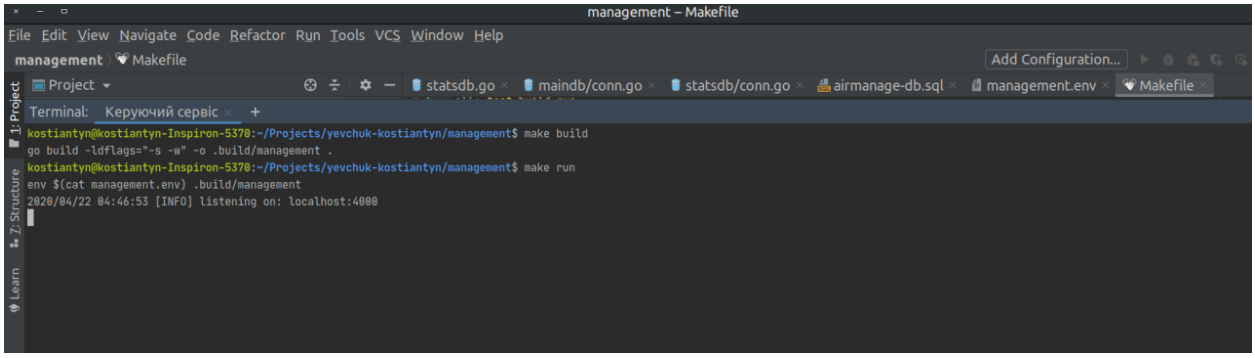

Рисунок 3.9 – Результат запуску сервісу керування

Повідомлення від сервісу на рис.3.9 означають, що сервіс був

запущений з конфігурацією від *management.env* файлу, керуючий сервіс має успішне підключення до баз даних, механізму маршрутизації DTN та може отримувати дані від моделей віддалених об’єктів.

Важливо розуміти, що кожен з процесів відбувається паралельно. Для цього використовується конструкція “go <назва функції>” з мови програмування Golang. Для перевірки успішності запуску паралельних функцій використовується механізм “група очікування”, котрий означає, що програма виконає зупинку, якщо усі паралельні функції закінчать свою роботу.

У випадку сервісу керування є 2 паралельні функції, котрі запускаються одразу після ініціалізації баз даних та DTN – це запуск API, за допомогою якого користувачі можуть управляти сервісом та транспортний модуль, котрий відповідає за комунікацію з моделям віддалених об’єктів через DTN-маршрутизацію.

Лістинг 3.10 – Main-функція сервісу керування

```
func main() {
statsStorage := statsdb.Init()
defer statsStorage.Close()

mainStorage, err := maindb.Init()
if err != nil {
        log.Fatal(err.Error())
}

wg := sync.WaitGroup{}
wg.Add(1)
go service.Run(mainStorage, statsStorage, &wg)
wg.Add(1)
go transport.Start(mainStorage, statsStorage, &wg)

wg.Wait()
}
```

API сервісу керування допомагає користувачу (інженеру або науковцю) запустити симуляцію обривів зв’язку та виконати тести, чи працює система коректно. У поточній імплементації API має 3 команди: створення моделі

віддаленого об’єкта, запуск та стоп симуляції.

Лістинг 3.10 – API сервісу керування

```
func Run(mainStorage model.MainDataStore, statsStorage model.StatsDataStore, wg *sync.WaitGroup)
	defer wg.Done()

	service := service{MainStore: mainStorage, StatsStore: statsStorage}
	r := mux.NewRouter()
	r.HandleFunc("/object", service.AddObject).Methods(http.MethodPost)
	r.HandleFunc("/object/{id}/off", service.SetInactiveState).Methods(http.MethodPatch)
	r.HandleFunc("/object/{id}/on", service.SetActiveState).Methods(http.MethodPatch)

	err := http.ListenAndServe("localhost:5000", r)
	if err != nil {
		log.Printf("[ERROR] http serve failed - (%s)", err.Error())
		return
	}
	return
```

Важливо зауважити, що на початку функції є конструкція defer wg.Done(), котра означає, що як тільки API сервіс виконає свою роботу, функція автоматично відправляє статус “Закінчила роботу” всім паралельним процесам у групі.

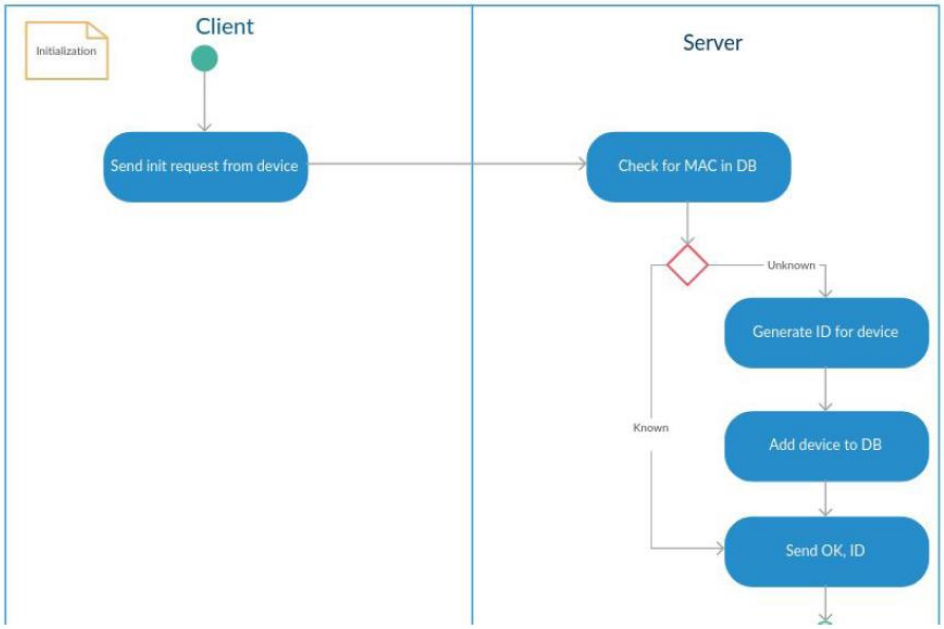

Рисунок 3.10 – Алгоритм створення віддаленого об’єкта у сервісі керування

На рис.3.10 зображений алгоритм, за яким новий об’єкт створюється у керуючому сервісі. Користувач повинен відправити POST запит у вигляді `http POST :5000/object id="object2000" name="object0" status:=true`. Таким чином, в основній базі даних буде створений запис *object2000* і коли модель віддаленого об’єкту розпочне роботу, керуючий сервіс матиме можливість ідентифікувати об’єкт та його конфігурацію.

Лістинг 3.11 – Метод створення об’єкта через API

```
func (s *service) AddObject(w http.ResponseWriter, r *http.Request) {
decoder := json.NewDecoder(r.Body)
var object model.Object
err := decoder.Decode(&object)
if err != nil {
        log.Printf("[ERROR] object decode failed - (%s)", err.Error())
        w.WriteHeader(http.StatusBadRequest)
        return
}
log.Printf("[INFO] adding object to main store - (%s)", object.ID)

object.CreatedAt = time.Now()
err = s.MainStore.CreateObject(&object)
if err != nil {
        log.Printf("[ERROR] saving object failed - (%s)", err.Error())
        w.WriteHeader(http.StatusInternalServerError)
        return
}
log.Printf("[INFO] successfully saved object - (%s)", object.ID)
return
}
```

Наступним кроком є запуск першої моделі віддаленого об’єкта *object2000.* Задля консистенції проекту та ліпшого масштабування також використовується Makefile з набором команд для запуску об’єкту.

Лістинг 3.12 – Makefile для запуску моделі віддаленого об’єкта

```
export CGO_ENABLED=0

SVC_NAME = object
```

```
BUILD_DIR=.build

all: test build run

verify:
go vet ./...

test: verify
go test ./...

get-deps:
env GIT_TERMINAL_PROMPT=1 go mod download

## build: build go service
build:
go build -ldflags="-s -w" -o $(BUILD_DIR)/$(SVC_NAME) .

## run: runs service locally
run:
env $$(cat $(SVC_NAME).env) $(BUILD_DIR)/$(SVC_NAME)

help: Makefile
@echo " Choose a command run in \033[32m"$(SVC_NAME)"\033[0m:"
@sed -n 's/^##//p' $< | column -t -s ':' |  sed -e 's/^/ /'
```

Команда “run” запускає об’єкт, котрий ідентифікував керуючий сервіс у мережі та виконав з’єднання до нього за алгоритмом на рис.3.11.

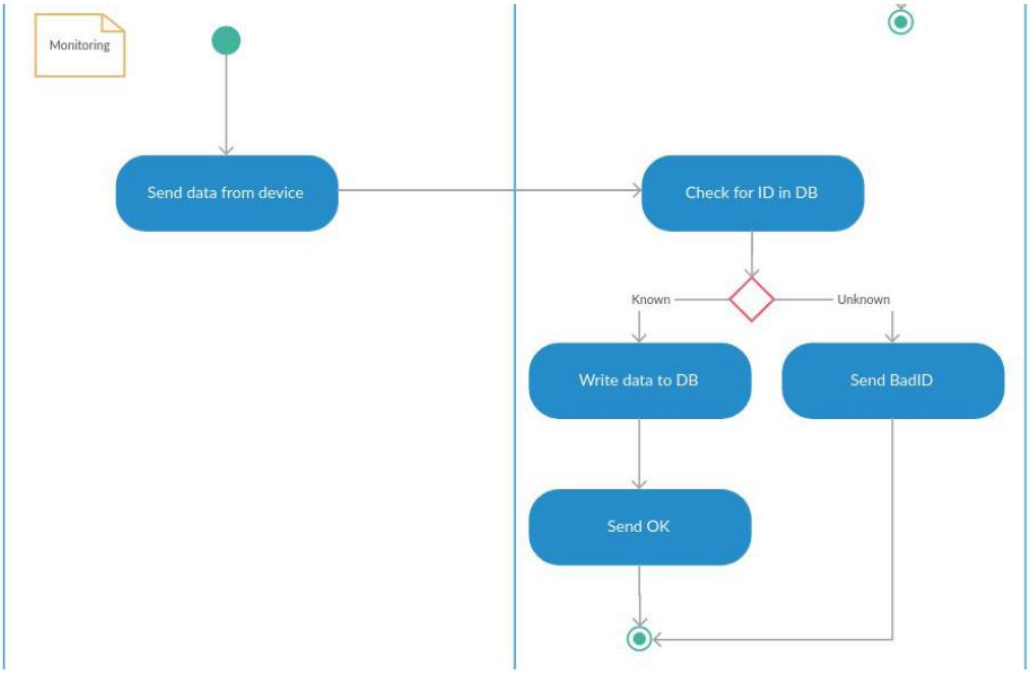

Рисунок 3.11 – Алгоритм з’єднання об’єкта та сервісу керування з передачею обчислювальних даних

Лістинг 3.13 – Функція обчислення та відправлення даних від об’єкта до сервісу керування

```
func Run(wg *sync.WaitGroup) {
defer wg.Done()
var c transport.Config
var uidConfig receiver.UIDConfig

if err := envconfig.Process("MAIN", &c); err != nil {
        log.Printf("[ERROR] parsing main bundle config failed - (%s)", err.Error())
        return
}

if err := envconfig.Process("UID", &uidConfig); err != nil {
        log.Printf("[ERROR] parsing object bundle config failed - (%s)", err.Error())
        return
}
done := make(chan struct{})
errors := make(chan error)
generatedData := make(chan *model.ObjectData)

conn, err := sender.NewConn(&c)
if err != nil {
        log.Println("[ERROR]:", err.Error())
        return
}
go trackClose(done)

wg.Add(1)
go data.Generate(uidConfig, generatedData, errors, done, wg)

wg.Add(1)
go data.Send(conn, generatedData, errors, done, wg)
}
```

Модель віддаленого об’єкта виконує з’єднання зі сервісом керування та починає передавати обчислювальні дані з поточною конфігурацією сервісу. В цей час сервіс керування публікує повідомлення про успішне отримання даних від моделі віддаленого об’єкту object2000 (рис.3.12).

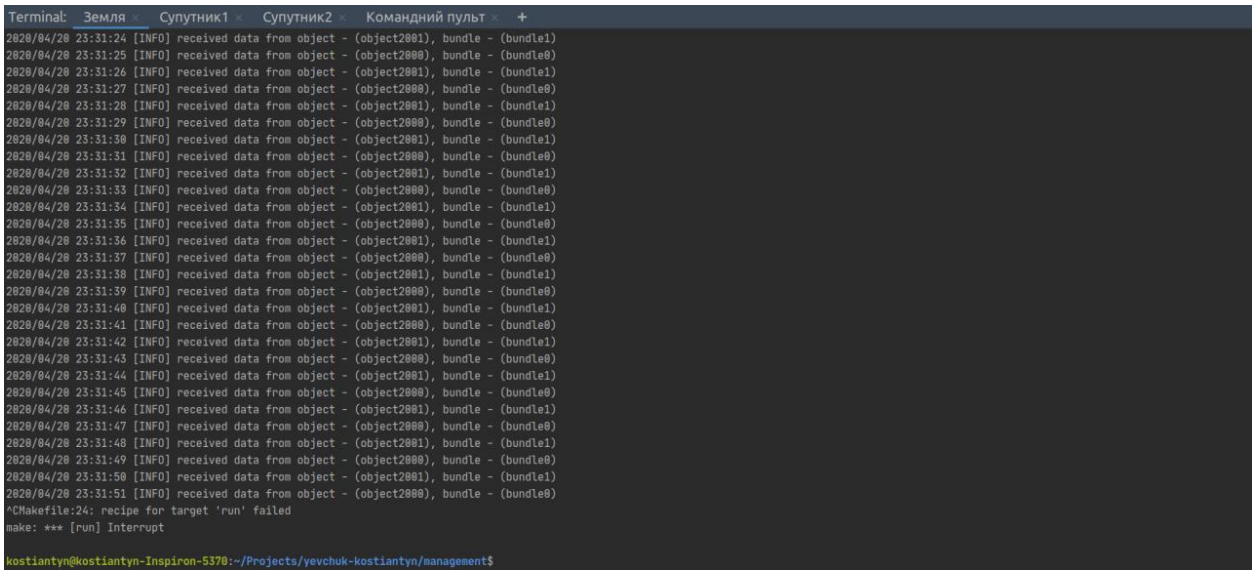

Рисунок 3.12 – Повідомлення керуючого сервісу про успішне отримання даних від *object2000*

Кожне повідомлення має статус-код та мітку часу (timestamp) у сервісі керування та об’єкті (рис.3.13).

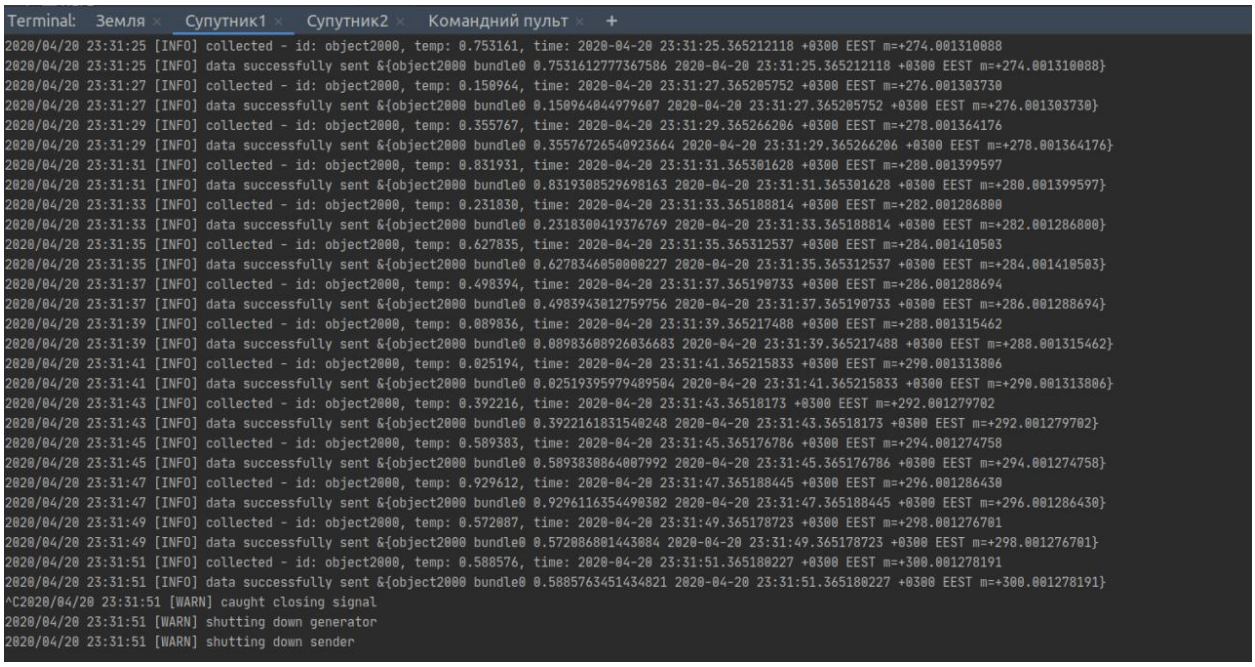

Рисунок 3.13 – Повідомлення моделі віддаленого об’єкта *object2000* про успішне відправлення даних до сервісу керування

Наступник кроком є запуск другої моделі віддаленого об’єкта *object2001*. Для цього потрібно змінити конфігурацію об’єкта в .env файлі та

запустити модель з новим конфігураційним файлом. Зауважимо, що у даному випадку не потрібно робити “build”, достатньо передати новий конфігураційний файл. Це досягається за рахунок єдиної кодової бази та гнучкого конфігурування.

Лістинг 3.14 – Конфігураційний файл моделі віддаленого об’єкта *object2001*

```
UID_LISTEN_HOST=localhost
UID_LISTEN_PORT=3001
UID_NAME=object2001
UID_NETWORK_NAME=bundle1

MAIN_SEND_HOST=localhost
MAIN_SEND_PORT=4000

NEIGHBOUR_SEND_HOST=localhost
NEIGHBOUR_SEND_PORT=3002
NEIGHBOUR_NETWORK_NAME=bundle2
```

Після запуску моделі нового віддаленого об’єкта можно зафіксувати успішні повідомлення про відправлення даних до сервісу керування від об’єкта (рис.3.14) та безпосередньо сервіса керування (рис.3.15).

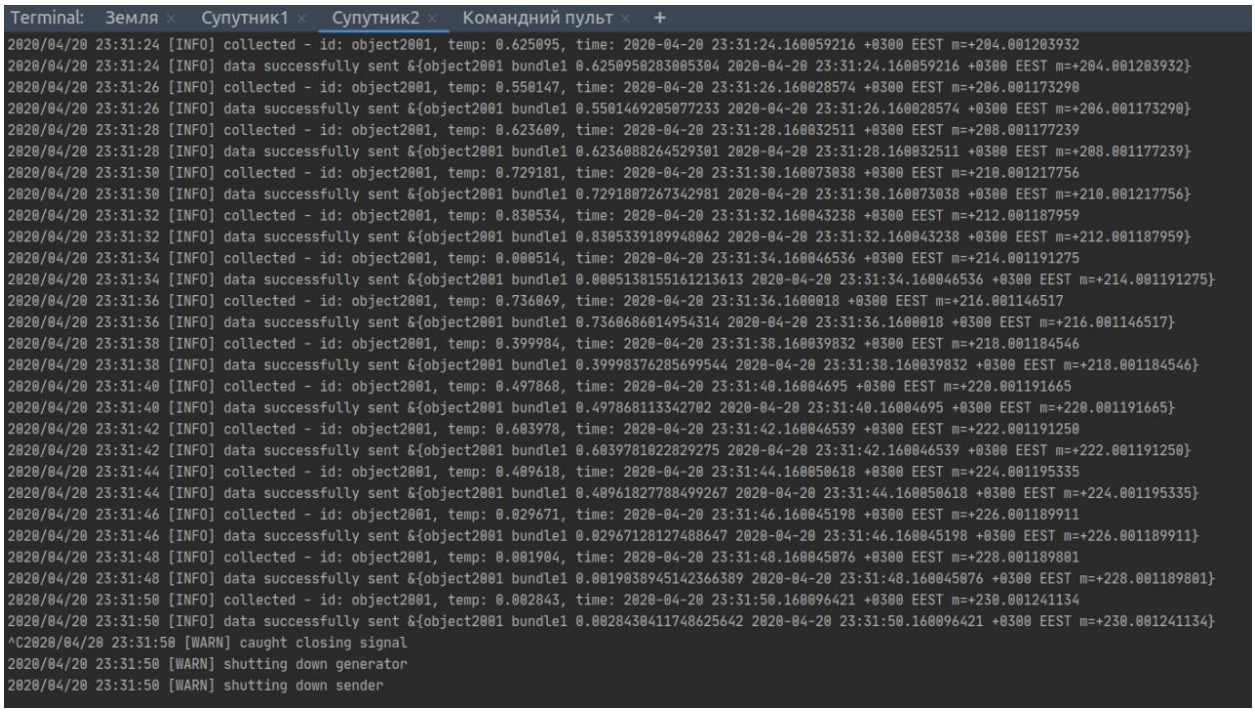

Рисунок 3.14 – Повідомлення об’єкта *object2001* про успішне відправлення даних до сервісу керування

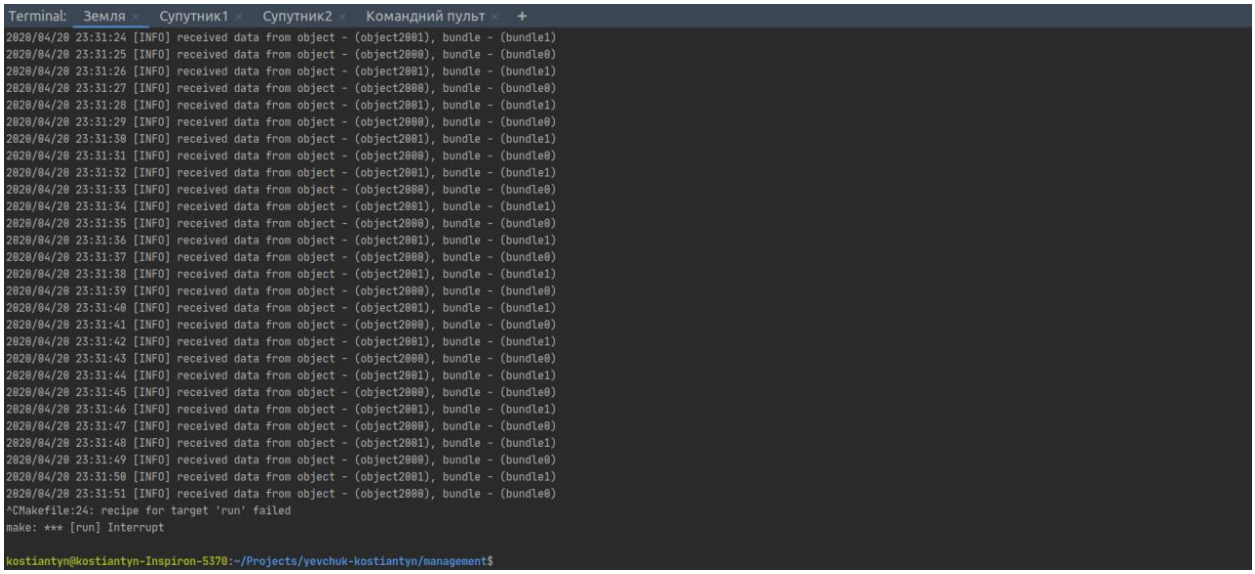

Рисунок 3.15 – Повідомлення керуючого сервісу про успішне отримання даних від *object2000* та *object2001*

На даному етапі обриви зв’язку можуть з’явитися з різних причин, якщо це космічний контекст, то один із об’єктів супутників може вийти в

тінь, таким чином втративши зв’язок з Землею. Важливо зауважити, що в такій ситуації хоча *object2000* фактично втратив зв’язок з керуючим сервісом, все одно завдяки конфігурації та маршрутизації DTN можливо підтримувати зв’язок з керуючим сервісом через *object2001*, котрий у даному випадку окрім виконання обчислювальних операцій, стане об’єктом-медіатором, через який *object2000* та керуючий сервіс зможуть підтримувати зв’язок.

Для моделювання даної ситуації користувач повинен перейти на вкладку “Командний пульт” та відправити команду до інтерфейсу API керуючого сервісу про симуляцію недоступності для *object2000*. (рис.3.16) Приклад команди – “http PATCH :5000/object/object2000/off”.

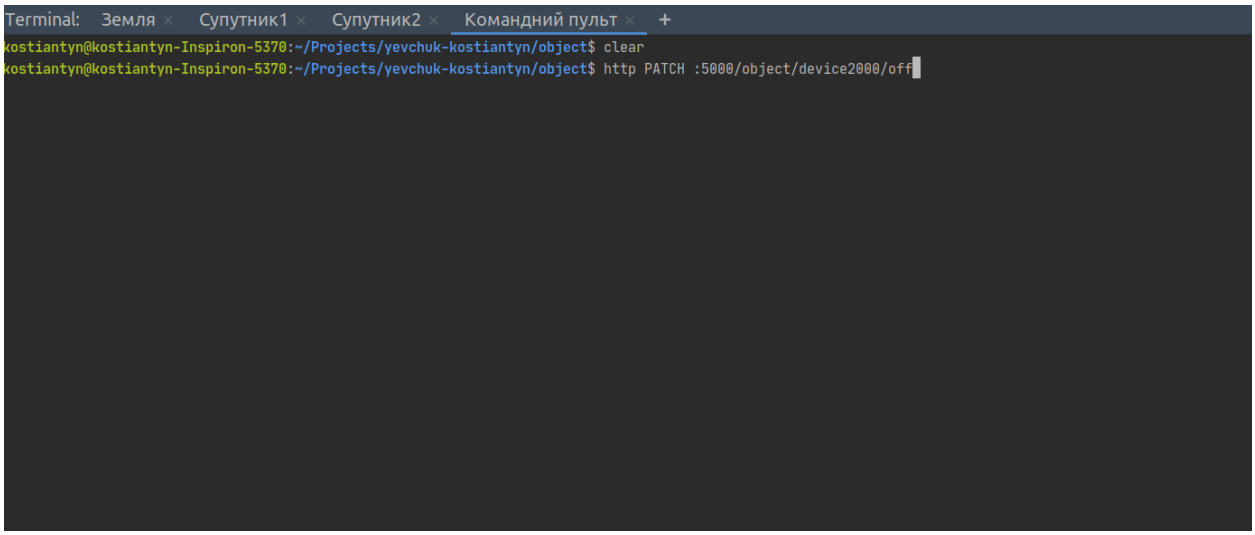

Рисунок 3.16 – Відправлення команди симуляції недоступності для *object2000* з командного пульту керуючого сервісу

Наразі спрацює алгоритм оновлення конфігурації моделі віддаленого об’єкту (рис.3.17) і *object2000* перестане мати зв’язок з керуючим сервісом та почне надсилати дані до *object2001*.

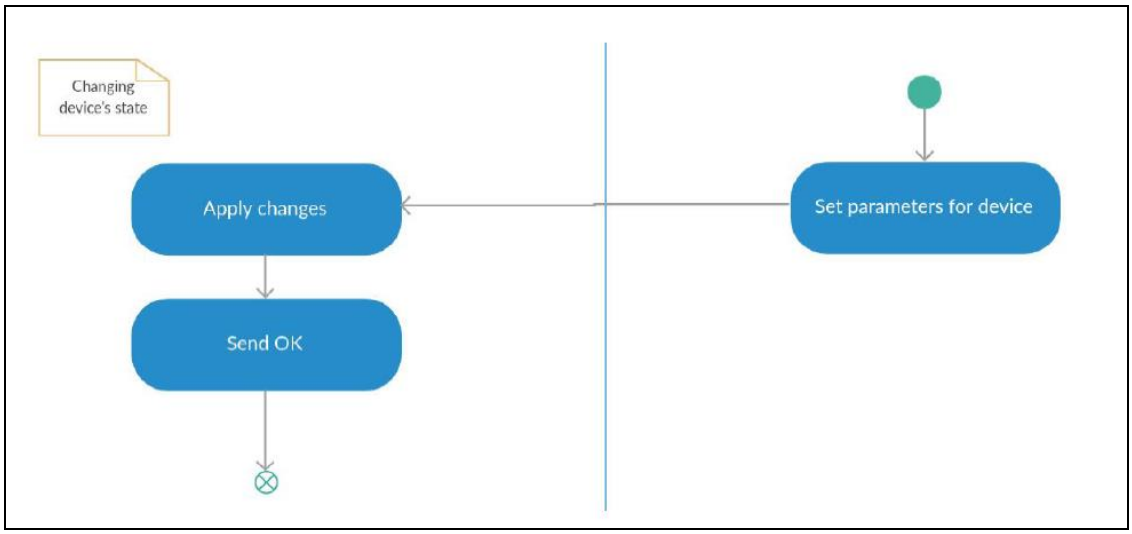

Рисунок 3.17 – Алгоритм зміни конфігурації об’єкту

В цей момент *object2001* починає відсилати повідомлення-попередження, що він також став медіатором (рис.3.18) і успішно доставляє дані до керуючого сервісу (рис.3.19).

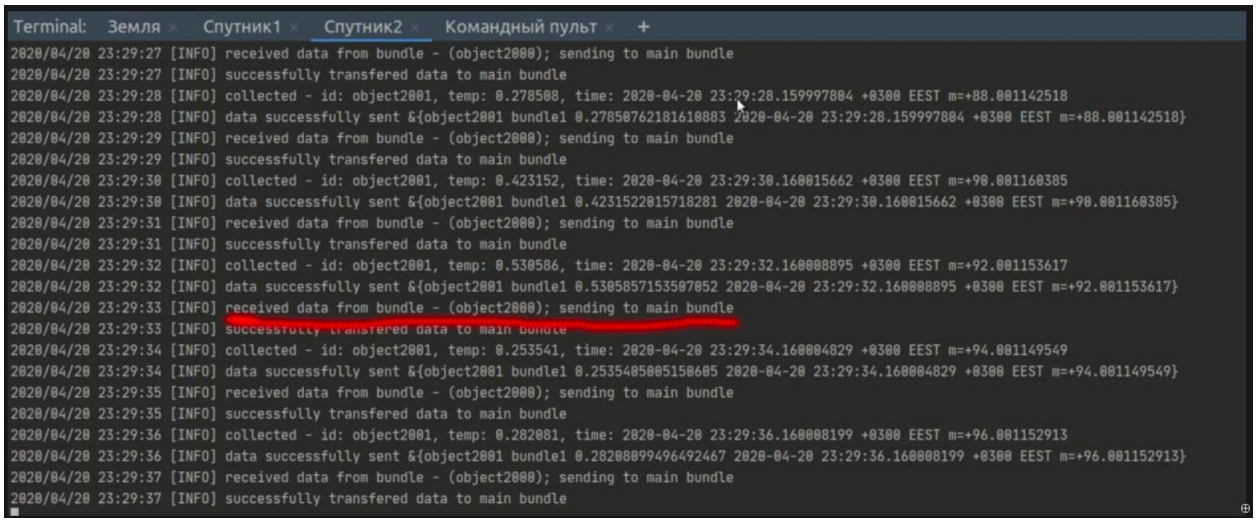

Рисунок 3.18 – Повідомлення від *object2001* про успішне отримання від *object2000* та відправлення даних до керуючого сервісу

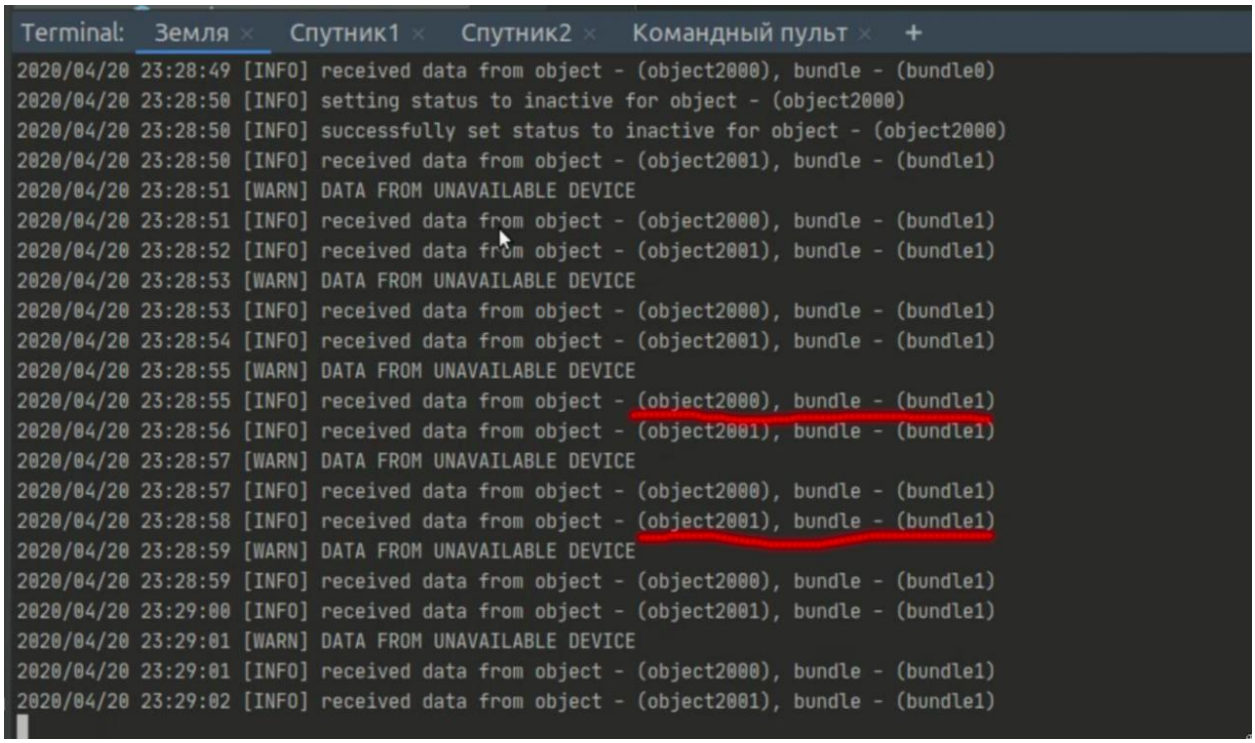

Рисунок 3.19 – Повідомлення від *керуючого сервісу* про успішне отримання даних *object2000* від *object2001*

Таким чином, на рис.3.19 керуючий сервіс повідомляє, що отримує дані недоступного через обрив зв’язку віддаленого об’єкту, але через bundle1, що означає через *object2001*.

Через проміжок часу *object2000* може стати знову доступним, тобто вийти з тіні, якщо у космічному контексті. Для цього користувач має активувати керуючий сервіс для цього об’єкта виконуючи команду “http PATCH :5000/object/object2000/on”. В такому разі, *object2000* перестає використовувати *object2001* у якості медіатора та повертає одно-канальний зв’язок з керуючим сервісом (рис.3.20)

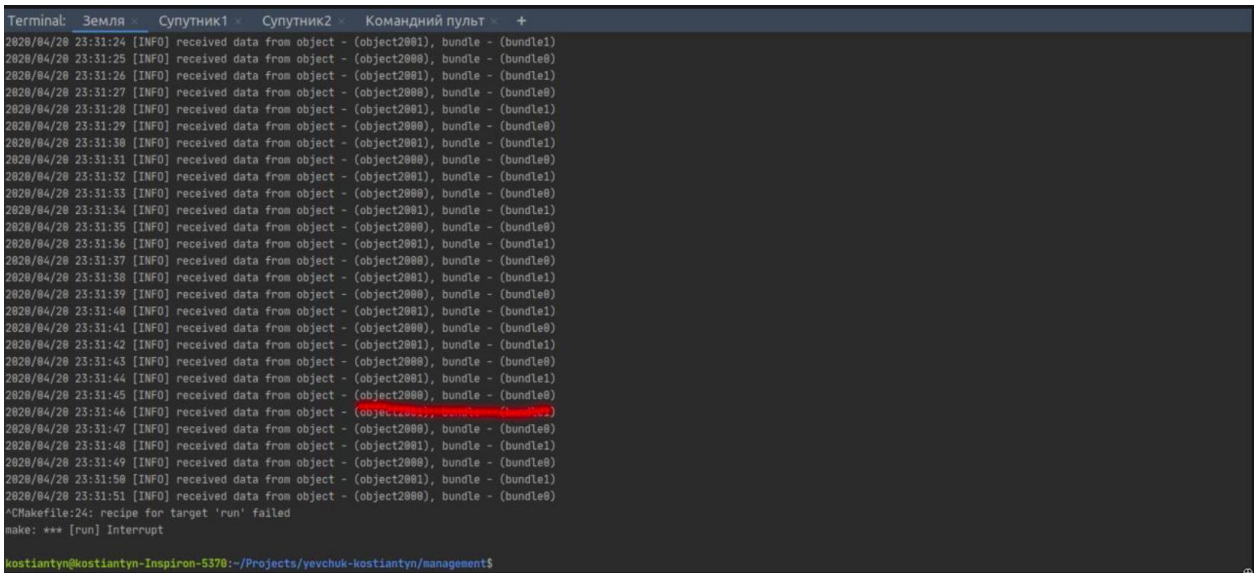

Рисунок 3.20 – Повідомлення від *керуючого сервісу* про успішне отримання даних *object2000* від *object2000*

На рис.3.20 керуючий сервіс продовжує отримувати дані *object2000* від bundle0, тобто від *object2000.*

Підводячи підсумки реалізації системи управління віддаленим об’єктом у мережі DTN, необхідно зазначити, що недоліки попередньої системи були виправлені. Порівняльну характеристику двох систем наведено у табл.3.1.

Таблиця 3.1 – Порівняльна характеристика програмної реалізації системи

	Система мережа TCP (попередня)	Система мережа DTN (поточна)
Тип СРЧ	Некритична	Критична
Розподілена	+	+
Можливість використання у віддаленому контексті	Неможливо, адже TCP підтримує тільки одноканальний тип зв’язку	Підтримує, мережа DTN дозволяє створювати та конфігурувати bundle між собою для багатоканальної ком.
Можливість обробки	Можливо, але дані	Повна підтримка, дані

обривів зв’язку	будуть втрачені	будуть збережені
Єдиний протокол обробки помилок та попереджень	-	+
Резервне копіювання статистичних даних	-	Модель віддаленого об’єкту тепер підтримує окрему статистичну базу даних

ВИСНОВКИ

В ході виконання атестаційної роботи розроблено модель віддаленого об’єкту та керуючий сервіс на основі стандарту побудови протоколу мережі DTN. При проведенні розробки докладно розглянуті алгоритми передачі даних в мережі толерантної до переривань, був обраний найоптимальніший для системи, а саме flooding.

Були проаналізовані особливості підходів до реалізації систем реального часу у віддаленому контексті. Був запропонований та реалізований об’єкт-медіатор, за допомогою якого здійснювався зв’язок, якщо він був втрачений між іншим віддаленим об’єктом та керуючим сервісом.

Був проаналізований підхід до створення резервних копій статистичних даних, зберігання попереджень та помилок, а також синхронізація втрачених даних.

За допомогою мови програмування Golang, середовища GolandIDE баз даних Influx, MySQL, ETCD була виконана імплементація моделей та керуючого сервісу.

Результати тестування підтверджують коректність роботи системи. Статистичні та конфігураційні дані були успішно синхронізовані, а обрив зв’язку між віддаленим об’єктом та керуючим сервісом не мав наслідків.

Реалізація даної системи повністю підтверджує сучасність розглянутих методів, алгоритмів та інструментів побудови СРЧ. Використання таких методів повністю може використовуватися при розробці різних проектів, мета яких протестувати конфігурації об’єктів та чи матиме наслідки система у випадку обривів зв’язку. Поточна система є легко-конфіруваною, кодова база моделі об’єкту одна, тобто для створення нового об’єкту для тестування потрібно тільки змінити конфігураційний файл. Також, кожний із модулів підтримує алгоритми масштабованості, тобто система може мати нескінчену кількість об’єктів та керуючих сервісів.

Наукова новизна роботи полягає у реалізації способу вирішення проблеми втрати зв’язку між об’єктами в системах реального часу. Такі проблеми виникають все частіше і частіше. Одним із прикладів використання системи може бути пожежа поблизу дата-центру з важлими об’єктами та їх даними або втрата зв’язку із супутником та важливими науковими даними. В такому разі, система, що була реалізована в роботі, матиме можливість зберігти та передати важливу інформацію, таким чином зберігти витрачені ресурси. Простий інтерфейс та гнучка конфігурація виконаної роботи дозволить розробникам та науковцям підтвердити або спростувати свої розробки шляхом налаштування поточної реалізації.

Практична цінність роботи полягає в можливості використання поточної системи як сторонньої бібліотеки, адже кожний компонент та модуль системи не залежить один від одного, або має інтерфейс для контакту.

Напрямком подальших досліджень може бути використання лімітів на кількість повідомлень та з’єднань. Зазвичай ресурси віддалених систем обмежені, тож у випадку аварії повідомлення повинні бути обмежені. Таким чином, можна розглянути алгоритми знаходження найефективнішого або об’єкта медіатора, що має велику кількість ресурсів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Орлов С.А. Теория и практика языков программирования: учебник для вузов. Стандарт 3-го поколения. – СПб.: Питер, 2013. – 688 с.

2. Уильямс Э. Параллельное программирование на C++ в действии., 2012. – 672 с.

3. Келеб Д. Go: создавайте надежные и масштабируемые приложения. – O'Reilly Media, 2016. – 62 с.

4. Varghese, S. Рецепты языка программирования Go, 2016. – 496 с.

5. Эндрюс Г. Основы многопоточного, параллельного и распределенного программирования, 2010. – 512 с.

6. Нюмен С. Создание микросервисов. – O'Reilly Media., 2016. – 304 с.

7. Omnipotent Team by Konstantin Yevchuk [Електронний ресурс] / Agilites Company Blog – Режим доступу: www / URL: https://agilites.com/blog-go-3.html – 29.01.2019 г.– Загол. з екрану

8. Bundle Protocol Version 7 [Електронний ресурс] / Tools IETF – Режим доступу: www / URL: https://tools.ietf.org/html/draft-ietf-dtn-bpbis-24 – 09.03.2020 г.– Загол. з екрану