Всеукраїнський конкурс на кращу студентську наукову роботу

2018/2019 навчального року

Шифр: «Decentralization»

Тема роботи: «**Система підтримки обчислень у децентралізованих мережах**»

Секція: «Комп’ютерна інженерія»

АНОТАЦІЯ

наукової роботи під шифром “Decentralization”.

Наукова робота: 43 сторінки, 10 рисунків, 4 таблиці, 9 лістингів, 22 джерела.

В роботі проведено аналіз існуючих децентралізованих систем, здатних підтримувати роботу сервісних додатків, наведено етапи реалізації системи з підтримки обчислень у децентралізованих мережах та етапи розробки протоколу передачі і обробки інформації для децентралізованих мереж різного призначення.

Метою роботи *є* *проведення досліджень та реалізація системи підтримки обчислень у децентралізованих мережах, шляхом розробки протоколу децентралізованої мережі з підтримкою роботи сервісних додатків*.

Наукова новизна роботи полягає в тому, що *розроблено новий метод запуску додатків в децентралізованій системі*, *розроблено модель системи та удосконалено процес підтримки обчислень за рахунок розробки протоколу децентралізованої мережі, яка здатна підтримувати роботу сервісних додатків*, *завдяки чому досягається можливість виконання додатків, зберігання та передача інформації децентралізовано.*

Практична цінність результатів роботи полягає в тому, що *розроблено систему, здатну виконувати оригінальний код але децентралізовано і в рамках системи, яка дозволяє використовувати практично будь-яку мову програмування, позбавлена недоліків інших технологій за рахунок виконання програм в спеціальному ізольованому середовищі але не у віртуальній машині. Це може бути корисним у багатьох сферах: у інтернеті речей для розширення функціоналу пристроїв без зайвих витрат; банківській та юридичній сферах при проведенні операцій третьою, незалежною стороною або проведення трудомістких обчислень без необхідності створення власного кластеру пристроїв*.

Робота виконана в рамках проведення кафедрою наукових досліджень з питань вдосконалення децентралізованих систем підтримки обчислень.

Результати досліджень впроваджено у навчальний процес, опубліковано у двох фахових наукових журналах, доповідались на двох конференціях, одна з яких міжнародна, та опубліковані у вигляді 4 тез доповідей у збірниках конференцій.

ДЕЦЕНТРАЛІЗАЦІЯ, ОБРОБКА ІНФОРМАЦІЇ, АЛГОРИТМ, ШИФРУВАННЯ, ХЕШУВАННЯ, ІЗОЛЬОВАНЕ СЕРЕДОВИЩЕ, РОЗПОДІЛЕННА БАЗА ДАНИХ.

ЗМІСТ

ВСТУП 4
1 ПОСТАНОВКА ЗАВДАННЯ 5
2 АНАЛІЗ АНАЛОГІВ, ЩО ІСНУЮТЬ 7
2.1 Децентралізовані мережі 7
2.2 Методи та протоколи безпечної передачі 9
2.3 Робота з готовими рішеннями-бібліотеками 10
3 ВИБІР РІШЕНЬ ДЛЯ ПОБУДОВИ СИСТЕМИ 10
3.1 Аналіз та вибір алгоритму хешування 10
3.2 Алгоритм шифрування 12
3.3 Вибір протоколу передачі інформації 15
3.4 Аспекти вибору мови програмування додатків 15
4 ОСНОВНІ ЕТАПИ РЕАЛІЗАЦІЇ СИСТЕМИ 16
4.1 Модель децентралізованої мережі забезпечення роботи сервісних додатків 16
4.2 Загальний опис протоколу 18
4.3 Формат адрес у мережі, дані та структура пакетів 20
4.4 Модулі мережі, повідомлень та команд 21
4.5 Модулі шифрування, хешування, серіалізації та розширень 23
4.6 Модуль пам’яті 26
4.7 Модуль та алгоритм маршрутизації 27
4.8 Модуль та алгоритм роботи додатків 29
5 ТЕСТУВАННЯ СИСТЕМИ 32
ВИСНОВКИ 36
ПЕРЕЛІК ПОСИЛАНЬ 37
ДОДАТОК А 39
ДОДАТОК Б 40
ДОДАТОК В 43
ДОДАТОК Г 44

ВСТУП

У сучасному світі неможливо представити роботу комп’ютерів, ноутбуків, планшетів та смартфонів без підключення до мережі. Ще складніше уявити виконання багатьох задач без використання звичних сервісів електронної пошти, хмарного сховища, меседжерів, тощо. Розвиток комп’ютерної інженерії та інформаційних технологій досяг рівня, що дозволяє будь-який предмет підключити до мережі. Тому питання доступності інформації, підвищення швидкодії її отримання та використання, підвищення ефективності та якості проведення обчислень є дуже актуальними на даний час. Разом з цим збільшується кількість трафіку і даних, які треба зберігати чи обробляти та з розвитком технологій все гостріше постають питання подальшого розвитку мереж та забезпечення їх безпеки.

Вирішенням усіх вище зазначених проблем є децентралізація мереж та додатків. Тому, на даний час, проводиться багато досліджень у цьому напрямку та вже є практичні впровадження на цю тему [1].

Мета наукової роботи - проведення досліджень та реалізація системи підтримки обчислень у децентралізованих мережах, шляхом розробки протоколу децентралізованої мережі з підтримкою роботи сервісних додатків.

Об’єктом дослідження є система підтримки обчислень у децентралізованих мережах.

Предметом дослідження є моделі, методи та інструментальні засоби підтримки ефективної взаємодії кінцевих пристроїв у децентралізованому середовищі.

Основним завданнями роботи є проведення досліджень та на їх основі розробка моделей, методів та засобів підтримки ефективної взаємодії кінцевих пристроїв у децентралізованій мережі, з можливістю передавати, зберігати, обробляти дані як з використанням сервісних додатків, так і без них.

1 ПОСТАНОВКА ЗАВДАННЯ

Із розвитком Інтернет майже всі додатки працюють як клієнт-серверні сервісів. Вони дозволяють обмінюватись повідомленнями, передавати файли, отримувати новини та публікувати їх. Сервіси взаємодіють як з користувачами так і з іншими сервісами, що часто використовують зловмисники при викраденні інформації. Окрім публічної інформації зловмисники крадуть данні банківських карток, приватну інформацію, файли та інше, тому захист даних стає пріоритетним питанням для компаній, які забезпечують надання сервісів.

Іншим питанням Інтернету є стрімкий розвиток інтернету речей. За останніми прогнозами у наступні роки кількість пристроїв, підключених до мережі Інтернет, збільшиться у декілька разів. Рішення проблеми – використання IPv6 замість IPv4, може допомогти але лише на невеликий період часу. Згідно з темпами росту інтернету речей, діапазон адрес IPv6 може скінчитись набагато раніше [2].

Таким чином питання, які треба вирішити для надійної роботи мережі Інтернет є такі:

– розробка нових протоколів передачі інформації, замість IPv6;

– перехід на децентралізовану модель роботи сервісів [2-3].

Заміна сервісів їх децентралізованими аналогами дозволить вирішити наступні завдання та питання:

– підвищення надійності самих сервісів;

– збільшення швидкості роботи сервісів;

– здійснення переходу на новий протокол передачі інформації.

При розробці нових мереж зручно будувати їх на основі вже існуючих мереж. Мережа, що побудована поверху вже існуючої, називається оверлейною. Оверлейні мережі дозволяють додавати нові властивості існуючим мережам і таким чином значно розширити кількість потенційних абонентів.

Під час розробки програмного забезпечення (ПЗ), необхідно провести дослідження та вирішити наступні задачі:

– реалізувати децентралізовану систему з підтримки роботи сервісів;

– провести тестування децентралізованої системи на працездатність;

– на основі тесту порівняти розроблену систему із вже існуючими аналогами.

На основі перелічених задач отримуємо наступні функціональні вимоги до системи (табл. 1.1).

Таблиця 1.1 – Функціональні вимоги

Вимога до ПЗ	Опис вимоги
Передача даних у рамках системи	Користувачі повинні мати змогу передавати данні один одному.
Зберігання інформації	Користувачі повинні мати змогу зберігати данні у системі та отримувати їх у будь-який час.
Виконання додатків у рамках системи	Система повинна надавати можливість для запуску додатків з якими взаємодіють користувачі.
Децентралізована робота системи	Система повинна передавати, зберігати та обробляти данні децентралізовано.

Система реалізується як протокол передачі та обробки інформації для децентралізованих мереж різного призначення. Для роботи з мережею користувач використовує стороннє програмне забезпечення або розроблене власноруч, на основі чіткого алгоритму роботи окремих частин мережі.

Основними завданнями роботи є:

– проведення досліджень та на їх основі побудова моделей та алгоритмів роботи окремих модулів системи;

– розробка децентралізованої мережі, яка забезпечить децентралізований обмін даними; децентралізоване зберігання інформації та запуск сервісів у мережі;

– розробка протоколу передачі та обробки інформації для децентралізованих мереж різного призначення;

– реалізація децентралізованої системи з підтримки роботи сервісів та проведення її тестування на працездатність.

2 АНАЛІЗ АНАЛОГІВ, ЩО ІСНУЮТЬ

В останній час набули популярності різні децентралізовані додатки та мережі, наприклад мережі ZigBee, Manet та додатки Ethereum, EOS, Bitcoin, IPFS. Але найбільш гучними є криптовалюти. До децентралізованих систем, що дозволяють розробляти власні додатки та виконувати їх у рамках систем є Ethereum або EOS. Вони мають коміркову топологію і це збільшує їх надійність. Кожна окрема децентралізована мережа або додаток створює власну адресацію та є оверлейними, це може збільшувати безпеку передачі даних оскільки вони можуть передаватися іншими шляхами після встановлення або відновлення з'єднання [4-5].

Оскільки системи є децентралізованими, кожен вузол мережі вважається ненадійним. Для передачі, зберігання та обробки інформації використовуються різноманітні криптографічні алгоритми.

Майже всі сучасні децентралізовані мережі та додатки мають відкритий вихідний код. Це збільшує довіру до мережі, збільшує кількість пристроїв на яких працює ця мережа або додаток. Завдяки наявності вихідного коду можна легко дізнатися про складові додатків, їх алгоритми та допоміжне програмне забезпечення. Додатково до вихідних кодів децентралізовані мережі та додатки мають, у вільному доступі, інформацію про алгоритми роботи. Все це дозволяє значно підвищувати безпеку зберігання та передачі інформації за рахунок контролю роботи додатків.

Більша частина популярних децентралізованих мереж, які дозволяють створювати децентралізовані сервіси, не є окремими додатками. Зазвичай вони є описом роботи додатка та алгоритмів, які використовуються. Реалізувати додаток може будь-хто, тому існує багато прикладів однакових мережі. Важливо зазначити, що різні реалізації працюють один з одним як єдиний додаток.

2.1 Децентралізовані мережі

На даний час існують децентралізовані мережі які підтримують виконання власних додатків. Найбільш популярними є Manet, ZigBee, Ethereum, EOS. Manet та ZigBee є фізичними децентралізованими мережами, а Ethereum та EOS – оверлейні мережі з підтримкою роботи децентралізованих додатків.

Manet є бездротовою Ad-Hoc мережею (динамічною і децентралізованою). ZigBee - децентралізована мережа з Mesh топологією. Обидві мережі є самоорганізованими. Кожна з них має власний стек технологій та протоколів. Наразі ZigBee поширені у інтернеті речей через свою простоту. Вони використовують малу кількість електроенергії, що підвищує тривалість роботи пристроїв.

Ethereum та EOS є оверлейними мережами. Вони використовують звичайні IP мережі і працюють на великій кількості пристроїв. Обидві мережі підтримують роботу додатків. Мережа EOS побудована насамперед для підтримки роботи децентралізованих додатків. Їх головний недолік – використання власної мови програмування та обмеження для роботи додатків. Роль додатків у цих мережах виконують смарт-контракти, що мають власну адресу у мережі та можуть оперувати лише даними та коштами, які надають користувачі. Це і є головна проблема – вони орієнтовані на розрахунки, а не на виконання трудомістких задач. Смарт-контракти працюють у середині віртуальних машин, не мають доступу до заліза та мережі і не можуть запускатись самостійно. Усі ці недоліки унеможливлюють використання цих мереж при створенні звичайних додатків. Для того, щоб децентралізовані додатки могли замінити централізовані, мережа не повинна їх обмежувати, або обмежувати значно менше. Ethereum та EOS не здатні вносити записи до баз даних, робити запити до сторонніх ресурсів або обладнання. Це є суттєвим недоліком існуючих мереж, що робить їх непридатними для повноцінного використання [2,6].

2.2 Методи та протоколи безпечної передачі

Поширені у сучасному світі мережі, для захисту інформації, широко використовують криптографію та прості алгоритми разом з поширеними та безпечними криптографічними алгоритмами, що робить їх досить захищеними і простими. За допомогою хешів мережі перевіряють данні, а шифрування використовується під час передачі інформації [7].

Існує багато алгоритмів шифрування та хешування. Різні мережі використовують різні алгоритми. Найбільш поширеними є алгоритми шифрування RSA, AES та Eliptic Curve, з великою обчислювальною складністю яких пов'язана висока крипостійкість. До найпопулярніших алгоритмів хешування у децентралізованих мережах можна віднести SHA1, MD5.

Алгоритми хешування дозволяють отримати відносно невеликі «ідентифікатори» будь-якого набору байтів. Під час хешування створюється фіксований набір байт. Але невелика зміна вхідної послідовності змінює кінцевий хеш дуже сильно. Алгоритми хешування характеризуються кількістю колізій – коли хеш від різного набору байт є однаковим.

Алгоритми шифрування використовуються для безпечної передачі даних у рамках мережі. Оскільки довіри у середині мереж не існує, данні треба надійно захищати. Крім цільового призначення, криптовалюти використовують ключи шифрування такі як ідентифікатор користувача у мережі. Існують симетричні та асиметричні алгоритми шифрування. У децентралізованих оверлейних мережах часто використовуються обидва типи алгоритмів оскільки асиметричні алгоритми можуть не шифрувати великий обсяг даних. У такому випадки використовуються і симетричні і асиметричні алгоритми одночасно.

Оскільки більша частина децентралізованих мереж оверлейні, вони використовують вже існуючі протоколи для передачі інформації. І цими протоколами, зазвичай, є два транспортних протоколи за допомогою яких данні передаються у відкритій мережі Інтернет: Transmission Control Protocol (TCP) та User Datagram Protocol (UDP).

TCP є надійним і гарантує передачу інформації у мережі, але кінцевий розмір даних, які можна ним передати, є меншим порівняно з протоколом UDP. UDP не гарантує передачу даних але здатен передати більше даних за один і той же проміжок часу. Різні децентралізовані мережі використовують різні протоколи. Деякі додатки та мережи підтримують роботу використовуючи обидва протоколи.

2.3 Робота з готовими рішеннями-бібліотеками

Майже всі існуючі мережі не використовують власні алгоритми шифрування, хешування, тощо, а використовують вже існуючі алгоритми, надійність яких доведена вченими. Такий підхід підвищує надійність готового продукту та зменшує вірогідність допустити помилку при розробці нових алгоритмів шифрування, що досить суттєво впливає на роботу мережі в цілому. Тому для реалізації мереж використовують існуючі бібліотеки, методи та функції.

При розробці додатків для нової платформи є ймовірність того, що та чи інша бібліотека не існує для якоїсь мови програмування, платформи чи операційної системи. У такому випадку розробнику треба провести дослідження та додатково реалізувати необхідні алгоритми.

З вище зазначеного стає зрозуміло, що є багато технологій децентралізованих мереж та сервісних додатків різного призначення. Кожен з яких використовує власні методи та алгоритми при реалізації специфічних завдань [2-4]. Алгоритми обираються виходячи з вимог конкретної мережі або додатка. Але майже всі існуючі рішення не здатні задовільнити потреби нових сервісів, що постійно реалізуються в мережі. Тому в даній роботі заплановано провести дослідження та запропонувати рішення більшості проблем шляхом розробки децентралізованої системи підтримки роботи сервісних додатків.

3 ВИБІР РІШЕНЬ ДЛЯ ПОБУДОВИ СИСТЕМИ

3.1 Аналіз та вибір алгоритму хешування

Алгоритми хешування на даний час є дуже поширеними та мають декілька реалізацій (MD5, SHA, RIPEMD та ін). Основними характеристиками алгоритмів хешування є: розрядність; складність обчислення; криптографічна стійкість; швидкість обчислення та кількість колізій. Хеш MD5 містить 128 біт та зазвичай записується як послідовність довжиною у тридцять дві шістнадцяткові цифри [7]. Наразі алгоритм MD5 не є крипостійким [8-9].

Алгоритм RIPEMD має багато версій залежно від довжини хешу: 128, 160, 256 та 320 біт. Наявність різних версій, що відрізняються довжиною хешу, зумовлено зростанням обчислювальних можливостей комп'ютерів [10]. SHA об'єднує різні алгоритми, такі як SHA-1, SHA-2, SHA-3, є дуже популярним та широко використовуються у сучасному світі. Алгоритми SHA характеризуються тим, що внесення навіть невеликих змін у вхідному рядку значно сильно змінюють хеш.

Алгоритм SHA-1 генерує хеш розміром 160 біт [11] та реалізує хеш-функцію, побудовану з використанням функції стиснення. Входом цієї функції є блок повідомлення довжиною 512 біт і вихід попереднього блоку повідомлення. Виходом є значення усіх хеш-блоків до цього моменту. Іншими словами хеш блоку Mі дорівнює $h_{\text{i}} = f(M_{\text{i}}, h_{\text{i}-1})$. У 2017-му році дослідники з компанії Google успішну атаку та опублікували два різних файли, які видають однакові хеші. Реалізація атаки на звичайному комп'ютері займає приблизно 110 років [12].

Під назвою SHA-2 існує ціле сімейство алгоритмів: SHA-224, SHA-256, SHA-384, SHA-512. Початкове повідомлення після доповнення розбивається на блоки з 16 слів. Алгоритм пропускає кожен блок повідомлення через цикли з різною кількістю ітерацій. Кожну ітерацію 2 слова перетворюються, а функцію перетворення задають інші слова. Результати обробки кожного блоку додаються, а сума дорівнює хеш-функції. Алгоритм є більш захищеним ніж SHA-1 але має його вразливості, у 2008-му році дослідники отримали колізії, при використанні усіченої версії [13]. Алгоритм SHA-2 використовує децентралізована система Bitcoin, а інші протоколи використовують його окремі алгоритми: IPSec, DSA, DNSSEC. Щоб

уникнути потенційних проблем SHA-3 створена без використання вже існуючих напрацювань.

Алгоритм SHA-3 (Keccak) розроблено групою співавтора алгоритму AES. Цей алгоритм побудовано за принципом криптографічної губки. Як і SHA-2, алгоритм Keccak має багато версій залежно від довжини отриманого хешу. Завдяки випадковим перестановкам алгоритм не має колізій. На даний момент алгоритм SHA-3 є не досить дослідженим але вже зараз він один із найперспективніших [14].

Оглянувши різні алгоритми хешування обрано сімейство алгоритмів SHA-2 оскільки вони є захищеними, достатньо дослідженими, побудовані на основі структури Меркла-Демґарда і мають велику швидкість роботи.

3.2 Алгоритм шифрування

На даний момент існує велика кількість алгоритмів шифрування, до яких відносяться AES, RSA, 3DES, ГОСТ 28147-89 та багато інших. Вони є симетричні та асиметричні. Симетричні алгоритми шифрування використовують один ключ для шифрування інформації, у той час як асиметричні мають пару ключів. Асиметричне шифрування є більш безпечним, але воно накладає обмеження на вхідні данні для шифрування. Як зазначено раніше, різні децентралізовані мережі використовують різні алгоритми шифрування. Часто можна зустріти використання двох та більше алгоритмів для передачі та зберігання інформації. На даний момент безпечними та популярними симетричними алгоритмами шифрування є AES, 3DES, Twofish. До асиметричних відносяться RSA, Діффі-Гелльмана, DSA, ECC та ECDH.

Симетричні алгоритми для шифрування та дешифрування інформації використовують один й той самий ключ. Це робить роботу з алгоритмом та процес шифрування простим та надійним, але ключ може бути незахищеним під час передачі між двома абонентами. AES, або Rijndael, є симетричним алгоритмом шифрування даних. У 2002-му році алгоритм оголошено стандартом шифрування. Розмір блоку - 128 біт, розмір ключа може приймати значення 128, 192 або 256 біт

[15]. На сьогодні цей алгоритм є найбільш розповсюдженим. Алгоритм 3DES є один із варіантів алгоритму DES. Ключ має розмір у 168 біт. Розмір одного блоку дорівнює 64 біт [16]. Цей алгоритм суттєво підвищує безпеку порівняно з DES, але значно знижує швидкість роботи. На цей алгоритм є теоретична атака, але виконати її на даний момент неможливо. Алгоритм Twofish розроблено у 1998-му році. Ключ – 128, 192 або 256 біт. Один блок дорівнює 128 біт [17]. Цей алгоритм має меншу швидкість роботи аніж Rijndael. Найстійкіший серед інших кандидатів.

Асинхронні алгоритми використовують два ключі для шифрування та дешифрування інформації – закритий та відкритий. За допомогою відкритого ключа можна зашифрувати інформацію та не можливо її дешифрувати. Для того, щоб дешифрувати повідомлення, необхідно володіти приватним ключем. Це робить процес передачі ключа безпечним оскільки при перехоплені відкритого ключа дешифрувати повідомлення залишається неможливим. Приватний ключ не передається та зберігається на пристрої.

Опис алгоритму RSA було опубліковано у 1977-му році. У 1997-му році співробітник центру урядового зв’язку Великої Британії описав схожий алгоритм, але він не був розкритий до 1997-го року. У якості відкритого та закритого ключів використовуються дуже великі прості числа. Цей алгоритм є досить захищеним, хоча існують різні теоретичні та практичні види атак. RSA є повільнішим за симетричні алгоритми. Ключ може бути 512, 1024, 2048, 4096 біт [18]. Чим вище розмір ключа тим надійніше будуть зашифровані дані, але швидкість шифрування буде меншою. Оскільки швидкість та надійність шифрування відносно низька порівняно з симетричними алгоритмами, типовим є використання RSA коли він передає лише ключ для симетричного алгоритму, а далі дані шифруються іншими алгоритмами.

Діффі-Геллмана не є алгоритмом шифрування як таким. Він призначений для генерації спільного ключа двома пристроями, що не мають жодної інформації один про одного. Це дозволяє створювати спільний ключ для, наприклад, симетричного алгоритму шифрування при використанні незахищеної середи

передачі інформації [19]. Алгоритм Діффі-Геллмана є приблизно так само безпечним як і RSA.

Алгоритм DSA не призначений для шифрування даних. Його цільове призначення – створення електронного підпису. За допомогою DSA лише один користувач може створити підпис документа, але перевірити його може будь-хто. Разом з алгоритмом SHA-1 є стандартом підпису документів [20].

Еліптична криптографія є окремим розділом криптографії. Він вивчає асиметричні алгоритми шифрування, які побудовані на еліптичних кривих над кінцевими полями (ECC). Алгоритми з використанням еліптичних кривих не використовують розв’язання якоїсь математичної задачі для створення ключа. Алгоритми шифрування з використанням еліптичної кривої використовується у багатьох популярних системах: TLS, PGP, SSH, багато криптовалют та децентралізованих мереж. Алгоритм є дуже складними для розуміння, але вони є значно захищеними ніж інші асиметричні алгоритми. Алгоритм дозволяє досягти рівня безпеки RSA із значно меншою довжиною ключа ті більшою швидкістю роботи [21]. Дуже часто алгоритм ECC не використовується сам по собі у децентралізованих системах. Зазвичай він використовуються разом із алгоритмом Діффі-Гелльмана. Таке поєднання отримало назву ECDH. Зв'язок цих двох алгоритмів дозволяю дуже швидко створити пару ключі незалежно один від одного, обмінятися публічними ключами та згенерувати спільний ключ за допомогою алгоритму Діффі-Гелльмана. Як результат вхідні данні для генерації спільного ключа є дуже стійкими, а генерація спільного ключа без його передачі дозволяє уникнути його перехоплення [22].

Асиметричні алгоритми мають дуже великий недолік – розмір даних, які вони можуть зашифрувати, не може перевищувати довжину ключа.

Оглянувши існуючі алгоритми шифрування вирішено використовувати два алгоритми шифрування – ECDH для генерації спільного ключа та AES для подальшого шифрування даних. Таке поєднання дає велику стійкість системи, оскільки: ключі шифрування не передаються через незахищені лінії зв’язку; алгоритм AES є найбільш захищений на даний момент; вхідні дані для генерації

спільного ключа є стійкими та обчислюються дуже швидко; існує можливість шифрування великих обсягів даних.

3.3 Вибір протоколу передачі інформації

Існує два протоколи для передачі інформації – TCP та UDP. Обидва протоколи працюють на транспортному рівні моделі OSI. Кожен протокол має свої особливості та переваги. Інколи додатки підтримують роботу використовуючи обидва протоколи.

Перевагами TCP є: надійна доставка сегментів; упорядкування сегментів при отриманні; робота з сесіями; контроль за швидкістю передачі.

Перевагою UDP є велика швидкість передачі відносно TCP. Протокол UDP використовується тоді, коли необхідно швидко отримувати дані. При використанні UDP логічний зв’язок не створюється, отже будь-хто може надсилати дані додатку.

Обидва протоколи адресують дані додаткам за допомогою портів. Одна програма може прослуховувати багато портів. На одному інтерфейсі лише один додаток може прослуховувати окремий порт. Прослуховування одного порту декількома додатками неможливе. Додатки можуть прослуховувати порти TCP та UDP окремо, або не прослуховувати один зовсім.

Оглянувши обидва протоколи було вирішено використовувати протокол TCP через його надійність, що дуже важливо у децентралізованих мережах.

3.4 Аспекти вибору мови програмування додатків

Існуючі децентралізовані мережі, які підтримують роботу сервісних додатків, мають власні мови програмування. Найпопулярнішою є Solidity.

Мова програмування Solidity є інтерпретованою та розроблена для криптовалюти Ethereum. Вона є Тюрінг-повною, об’єктно-орієнтованою, підтримую функції, змінні, цикли та умовні оператори. Для цієї мови вже створено безліч додатків. Виконання програм на мові програмування Solidity потребую

спеціальної валюти – газу. Додатки, які написано цією мовою, не мають доступу до мережі або заліза. Вони можуть оперувати даними, які їм надав користувач. Одна копія програми може працювати з безліччю користувачів. Кожен користувач буде працювати із своїм власним набором даних, але різні користувачі можуть взаємодіяти один з одним. На даний момент ця мова є найбільш популярною для розробки децентралізованих додатків. Хоча Solidity було розроблено для роботи у мережі Ethereum, стороні мережі також підтримують роботу цих додатків, але ізольовано від Ethereum та інших мереж. Програми, написані мовою Solidity, працюють у спеціальній віртуальній машині, що накладає багато обмежень на швидкість роботи та функціонал.

Мова Solidity вже є розповсюдженою, але вона не надає багато можливостей для програмістів. Додатки, написані цією мовою, відносно віжко налагоджувати. Вони не можуть виконувати багато функції, на які здатні звичайні програми: не існує потоків, роботи з файлами, роботи з мережею та обладнанням. Використання традиційних мов програмування дозволить вирішити цю проблему. До традиційних мов програмування відносяться Java, C++, C#, Python та багато інших.

Для кожної з перелічених мов існує безліч бібліотек та готових рішень багатьох задач. Їх швидкість роботи значно вище ніж у Solidity, особливо у мови C++. Кожна з перелічених мов має багато прихильників, вони є більш легшими для вивчення та мають багато додаткових утиліт для перевірки та оптимізації сирцевого коду. Додатково всі перелічені мови мають доступ до заліза і мережі, дозволяють створювати складні додатки, на зо не здатна мова Solidity. Враховуючи вищесказане зрозуміло що слід використовувати одну з традиційних мов– C++, Java, C# чи Python.

4 ОСНОВНІ ЕТАПИ РЕАЛІЗАЦІЇ СИСТЕМИ

4.1 Модель децентралізованої мережі забезпечення роботи сервісних додатків

Основною задачею є розробка децентралізованої мережі, яка підтримує: децентралізований обмін даними; децентралізоване зберігання інформації та запуск сервісів у децентралізованій мережі.

За результатами попереднього аналізу та проведених досліджень виявлена необхідність у децентралізованій мережі, яка здатна підтримувати роботу сервісних додатків. В результаті першого етапу її реалізації розроблено та наведено модель децентралізованої системи, загальними елементами якої є фізичні елементи мережі та логічні зв’язки між ними (рис. 4.1).

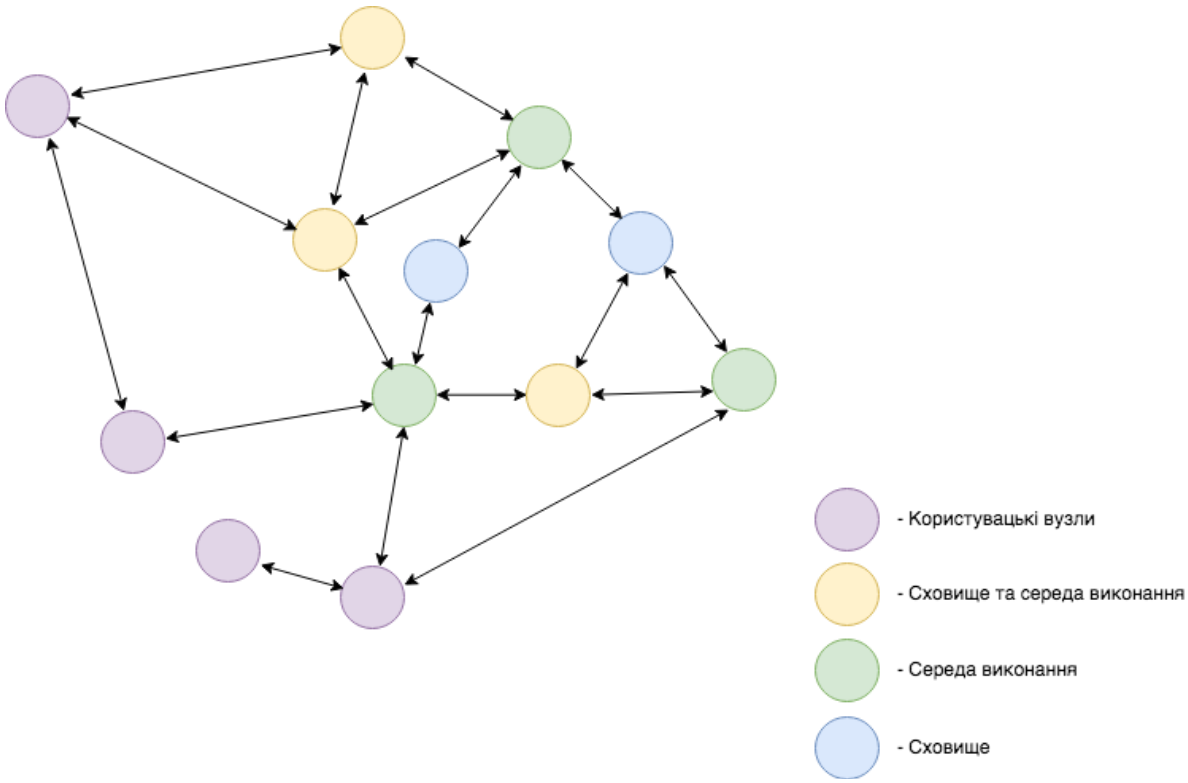

Рисунок 4.1 – Модель децентралізованої системи

Як видно на рисунку, в реальній системі існує декілька типів вузлів: користувацький, сховище, середа виконання та вузли, що поєднують одночасно декілька функцій. Окрім того усі вузли системи є транзитними, тому будь-хто може підключитись до системи через ці вузли. Зазначимо, що система у кожен момент часу може бути як централізованою, так і децентралізованою або розподіленою, інформація може зберігатися або на всіх пристроях, або лише на окремих, а

додатки виконуватись на будь-якому з вузлів системи або лише на спеціальних вузлах, що надають їм таку можливість. При певних умовах система є частково централізованою, оскільки при малої кількості пристроїв є ймовірність існування тільки окремих вузлів для сховища та середовища виконання. При великій кількості пристроїв система частіше всього є розподіленою, а під час роботи додатків вона є частково децентралізованою тому, що є вірогідність існування одного або декількох центральних елементів, які керують додатками. При наявності великої кількості пристроїв у мережах різних типів, система є повністю розподіленою під час пересилання та зберігання даних.

4.2 Загальний опис протоколу

Протокол описує оверлейну децентралізовану мережу, яка здатна передавати, зберігати та обробляти дані користувачів. Передача виконується децентралізовано без прямого встановлення зв’язку між клієнтами. Дані зберігаються розподіллено та публічно. Обробка даних виконується за рахунок підтримки роботи сервісних додатків у рамках мережі. Протокол описує взаємодію різних окремих модулів мережі. Кожен модуль відповідає за певні функції в рамках мережі. Такий підхід дозволяє швидко змінювати складові частини додатка, які реалізує клієнт мережі, не впливаючи на роботу інших модулів. Усього протокол описує дев’ять модулів: мережі; повідомлень; хешування; шифрування; команд; пам’яті; маршрутизації; серіалізації; додатків та розширень (рис.4.2). Додаток, що реалізує протокол, може використовувати не всі модулі, а лише необхідні для його роботи. Кожен модуль виконує одну дію і це зменшує кількість помилок та допомагає тестувати окремі частини мережі. Далі детально розглянемо усі модулі.

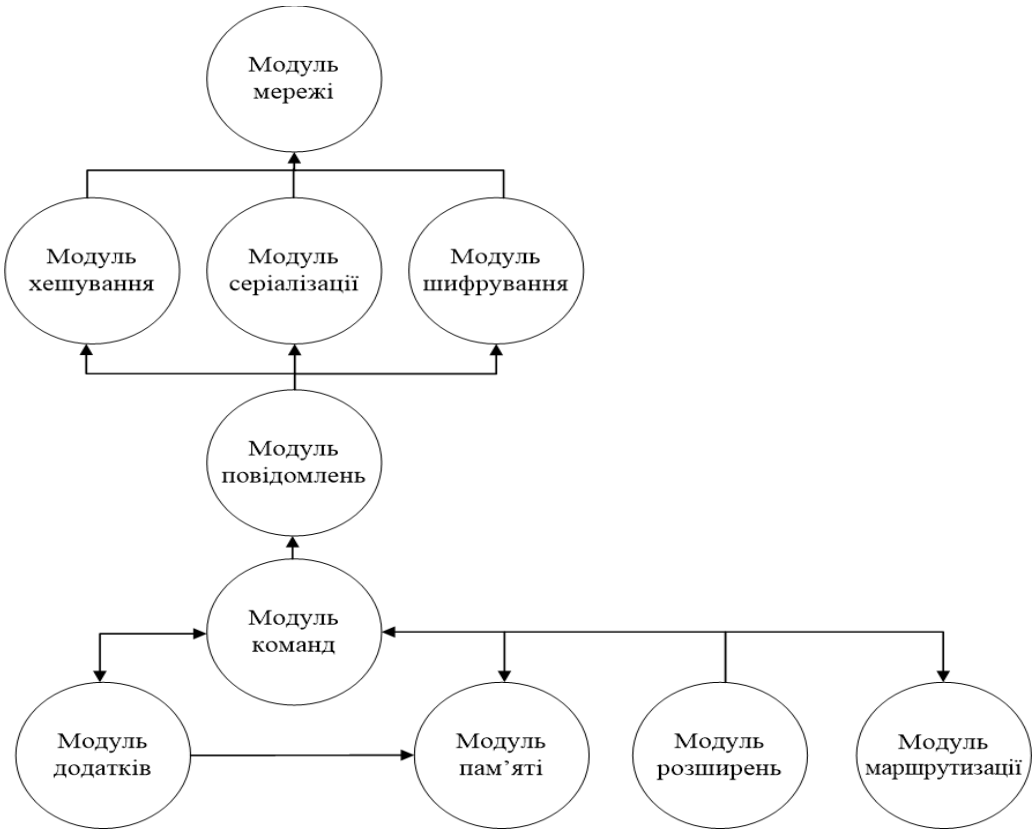

Рисунок 4.2 – Взаємозв'язок модулів протоколу

Додатково протокол описує структуру пакетів у рамках мережі, алгоритми та формат адрес і даних, що передаються. Додаткове ПЗ, необхідне для роботи мережі, обирається програмістом, який інтегрує мережу у свій додаток. Протокол описує взаємодію двох типів пристроїв у мережі: кінцеві та транзитні. Транзитним – є будь-який вузол, що не є відправником або отримувачем повідомлення. Один фізичний пристрій може одночасно бути як транзитним вузлом, так і кінцевим, оскільки кожен додаток, що використовує мережу є незалежною одиницею та має власну адресу.

Сервісні додатки та данні виконуються на транзитних вузлах. Транзитні вузли передають дані та не мають доступу до вмісту даних. Будь-який транзитний вузол може отримати будь-яку інформацію, що зберігається у мережі. Не всі рівні моделі використовуються під час роботи протоколу. Деякі додатки можуть обмежуватись певним необхідним рівнем, наприклад транзитні вузли можуть реалізовувати рівні до модуля повідомлень включно, оскільки інші рівні в їх випадку не використовуються. Додатково кожен з модулів є незалежним і може

використовуватись з будь-якого іншого модуля, наприклад модуль команд може шифрувати та дешифрувати дані напряму без звертання до модуля повідомлень.

4.3 Формат адрес у мережі, дані та структура пакетів

Адресом в мережі є довільна послідовність символів. Будь-який рядок може бути адресом. Завдяки цьому не існує обмеження на кількість вузлів у мережі для їх адресації. При вичерпанні вузлів достатньо додати декілька символів без необхідності зміни всього протоколу. Під час розробки протоколу вирішено використовувати у якості адреси публічний ключ алгоритму ECDH, що перетворено у рядок за допомогою системи числення з основою 64. Використання такого адресу значно спрошує обмін ключами та робить його дуже безпечним, оскільки ключ попередньо передавати не треба. Адреса описує не фізичний пристрій, а додаток. Один пристрій може мати декілька адрес. Дізнатися інформацію щодо фізичного розташування пристрою за допомогою адреси не можливо. Для зміни адреси додатку необхідно лише згенерувати нову пару ключів.

Дані у мережі передаються у текстовому вигляді. Перед передачею даних вони серіалізуються. Як алгоритм серіалізації обрано формат даних JSON. З цим форматом можна працювати у будь-якій мові програмування оскільки він є текстовим та простим при дослідженні. Формат JSON широко використовується у сучасних додатках. Для передачі інформації використовуються пакети. Пакет має два рівні: транзитний та клієнтський. Транзитний рівень є відносно публічним і будь-хто може отримати до нього доступ. Клієнтський рівень призначений для отримувача пакета і передається у зашифрованому виді. Транзитний пакет зберігає інформацію щодо маршруту пакета та інформацію про самі дані. Структура транзитного рівня наведена у таблиці 4.1.

Таблиця 4.1 – Структура транзитного рівня пакета

Назва поля	Розмір (байт)	Опис
Розмір даних	2	Довжина полю даних

Ланцюг вузлів	Кількість вузлів*40	Містить ланцюг вузлів, через які слід переслати повідомлення. Формат ланцюга описано окремо
Відправник	40	Адреса відправника
Дані		Вміст пакета

Зчитавши транзитний рівень пакета вузол знає призначений він йому або ні та куди його слід переслати далі. У полі «дані» зберігається клієнтський рівень у зашифрованому виді. Розшифрувати його може лише отримувач даних. Структура клієнтського рівня наведена у таблиці 4.2.

Таблиця 4.2 – Структура клієнтського рівня

Назва поля	Розмір (байт)	Опис
Дані		Містить самі дані, які пересилаються
ID пакета	40	Ідентифікатор пакета
Номер пакета	4	Номер пакета
Кількість пакетів	4	Загальна кількість пакетів
Контрольная сумма	4	Контрольная сума цього пакета

Поле «дані» клієнтського пакета містить будь-що у текстовому виді. Контрольна сума вираховується для пакету, що передається. За допомогою її додаток перевіряє цілісність пакету. При передачі великого об’єму даних, пакет фрагментується. У такому випадку загальна кількість пакетів зберігається у полі «Кількість пакетів» клієнтського рівня, а номер конкретної частини сегмента у полі «номер пакета» цього ж рівня. Контрольна сума вираховується для окремого сегмента. Ідентифікатор пакета не є унікальним для сегментів, але є унікальним для оригінального не сегментованого повідомлення. Завдяки цим трьом параметрам додаток об’єднує отриманні сегменти у одне ціле повідомлення.

4.4 Модулі мережі, повідомлень та команд

Модуль мережі відповідає за встановлення з’єднання між клієнтами. Цей модуль оперує з’єднанням, передачею та отриманням інформації за допомогою протоколу TCP транспортного рівня моделі OSI. Цей модуль будує певний абстрактний шар між децентралізованою мережею та існуючою фізичною мережею. За допомогою модуля мережі створюється оверлейна мережа, яка теоретично здатна працювати з будь-яким фізичним з’єднанням та протоколом. Ця структура здатна охопити якомога більше пристроїв, що необхідно для безпечної роботи будь-якої децентралізованої мережі або додатка.

Модуль мережі є найнижчим модулем. Через нього інші модулі отримують та передають інформацію. Модуль мережі отримує дані, які необхідно надіслати, у текстовому виді та передає їх необхідному вузлу. Модуль працює лише з тими вузлами, що підключені до нього. Приклад модуля мережі наведено у додатку А.

Модуль команд призначено для створення додатків, які використовують мережу. При використанні команди один клієнт може запросити щось у іншого. Завдяки командам розширюється можливості мережі та додаються нові функції. Команди не є сервісами у мережі, вони є їх заміною у певних випадках.

Типовим використанням команди є передача спеціалізованого повідомлення, запит на підключення, додавання або отримання даних, тощо. За допомогою команд підтримується працездатність мережі. Будь-який додаток може додати власні команди до вже існуючих. При отриманні команди, додаток перевіряє, чи може її виконати. Якщо виконати її неможливо (команда не зареєстрована), додаток її відкидає. Інакше команда виконується.

Командою є спеціальний об’єкт. Дані команди зберігаються у цьому об’єкті. Команда є надбудовою над повідомленням та не відрізняється від повідомлення зовні. Вона відрізняється лише тим, що може виконувати будь-які дії, відсилати відповіді та реагувати на них. Інакше кажучи метод дозволяє створювати двонаправлену взаємодію. Приклад методу наведено у лістингу 4.1. Цей метод отримує повідомлення та надсилає відповідь. Команда приймає рядок. Відповідь містить вхідний рядок великими літерами та поточну дату та час.

Лістинг 4.1 – Приклад реалізації

```
import command
import simple_package
import simple_response_package
import datetime

class SimpleCommand(command.Command):

    def handle(self, package: simple_package.SimplePackage):
        response = simple_response_package.SimpleResponsePackage()
        response.date = datetime.datetime.now()
        response.message = package.message.capitalize()
        self.send_response(response)
```

Модуль повідомлень формує, передає та отримує нові повідомлення (пакети). Він працює з модулями мережі, шифрування, хешування та серіалізації. Цей модуль розбиває дані на сегменти, формує пакети та передає їх модулю мережі для передачі. Також він отримує дані від модуля мережі. Після отримання даних з мережі, він десеріалізує їх та зчитує транзитний рівень. Якщо додаток є вузлом, якому відправлено дані, то він виконує необхідні дії для обробки пакета (рис. 4.1). Якщо вузол є транзитним, то він формує новий ланцюг та надсилає наступному вузлу. Якщо поточний вузол не є кінцевим або транзитним то пакет видаляється.

4.5 Модулі шифрування, хешування, серіалізації та розширень

Модуль шифрування виконує наступні дії: шифрує дані; дешифрує дані; генерує спільний ключ. Він працює з наступними алгоритмами шифрування: Elliptic Curve (ECC), Elliptic Curve Diffie-Hellman (ECDH) та AES. Для публічного ключа відправника та власного приватного ключа цей модуль генерує спільний ключ з метою подальшого використання у алгоритмі AES. Це дозволяє згенерувати ключ будь-коли та не потребує передачі додаткових даних через мережу. Відкритий ключ використовується у якості адреси додатка у мережі, тому для зміни адреси достатньо згенерувати нову пару ключів. Втративши публічний або приватний ключ додаток не зможе отримувати або відправляти дані іншим учасникам мережі. Приклад коду модуля шифрування наведено у додатку Б, а алгоритм шифрування на рисунку 4.3.

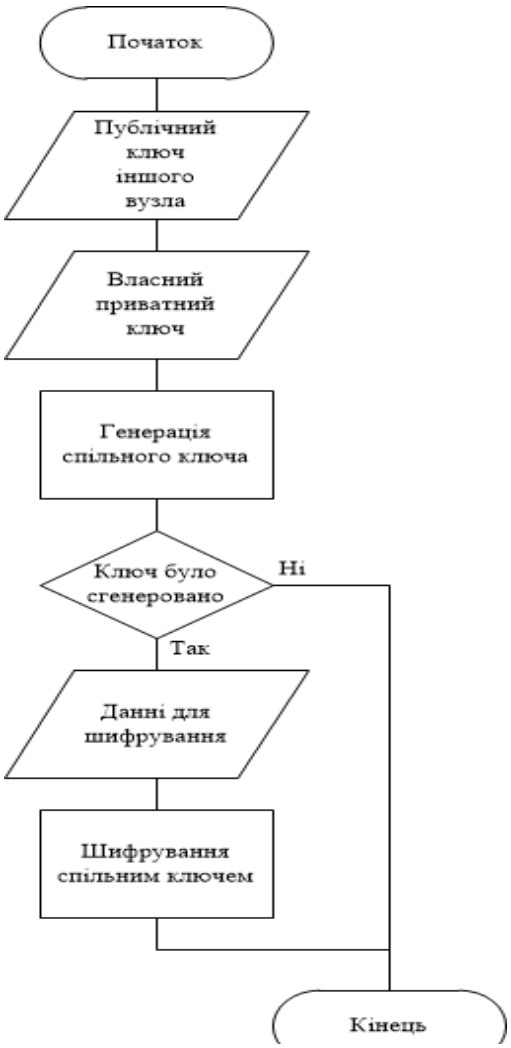

Рисунок 4.3 – Алгоритм шифрування даних

Модуль хешування виконує дві операції: хешує дані та перевіряє хеш. Для створення хешу використовується алгоритм SHA-256. Для перевірки хешу модуль генерує новий хеш для даних, які необхідно перевірити, та перевіряє новий хеш з отриманим для перевірки. Приклад коду модуля хешування приведено у додатку В.

Модуль серіалізації (лістинг 4.2) перетворює структури даних у рядки у форматі JSON. Також цей модуль десеріалізує дані з рядка в об’єкти конкретної мови програмування. Оскільки усі структури даних різні у різних мовах програмування, для уніфікації об’єкти, крім масивів, повинні мати ідентифікатор. За допомогою цього ідентифікатора мова програмування може коректно перетворити отриманні дані у об’єкт. Оскільки додаток розроблює будь-хто, він може містити нестандартні повідомлення та структури даних, або зберігати їх у

іншому вигляді. Тому вирішено передавати дані без прив’язки до мов програмування, а додати ідентифікатори.

Лістинг 4.2 – Модуль серіалізації

```
import sys, os
import json
from serialization import Encoder
sys.path.insert(0, os.path.dirname(os.getcwd()))
class Serializer(object):
    __instance = None
    @staticmethod
    def get_instance():
        if Serializer.__instance is None:
            Serializer.__instance = Serializer()
        return Serializer.__instance
    def __init__(self):
        self.types = {
            # messages types
        }
        self.encoder = Encoder.Encoder()
    def serialize(self, obj):
        return json.dumps(obj, default=self.encoder)
    def deserialize(self, message):
        return json.loads(message, default=self.encoder)
    def get_type(self, obj):
        return obj.__class__.__bases__[0].__name__
    def add_type(self, obj, tag):
        obj_type = self.get_type(obj)
        self.types[obj_type] = tag
```

Модуль розширень (лістинг 4.3) додає нові типи даних, що підлягають серіалізації та команди. Без цього клієнти можуть не знати як правильно перетворити серіалізовану послідовність у об’єкти. Модуль розширень зв’язує додаткові ідентифікатори з типами даних, яким вони відповідають.

Лістинг 4.3 – Модуль розширень

```
from serialization import Serializer
class Extensions(object):
    __instance = None
    @staticmethod
    def get_instance():
        if Extensions.__instance is None:
            Extensions.__instance = Extensions()
        return Extensions.__instance
    def __init__(self):
        self.serializer = Serializer.Serializer()
    def add_type(self, obj, tag):
              self.serializer.add_type(obj, tag)
```

4.6 Модуль пам’яті

Модуль пам’яті виконує роль розподіленої бази даних. Цей модуль відповідає за зберігання даних, що зберігаються у мережі. Ці дані не можуть бути змінено чи видалено. Дані зберігаються у виді розподіленої хеш-таблиці. Кожен запис має власний ідентифікатор – хеш. Знаючи хеш можна отримати дані, які з ним асоційовано. Втративши хеш, дані буде «втрачено». Отримати список усіх даних неможливо. У пам’яті мережі може зберігатися будь-що: текстова інформація, додатки, файли. Формат запису у розподіленій таблиці має наступний вид (табл. 4.3).

Таблиця 4.3 – Структура рядка розподіленої бази даних

Назва колонки	Розмір (байт)	Призначення
Ідентифікатор	40	Ідентифікатор запису, хеш даних
Розмір даних	4	Розмір самих даних

Дані не зберігаються у таблиці як такі. Таблиця лише містить інформацію про дані, що існують на цьому пристрої. Самі дані знаходяться на носії та мають унікальне ім’я. Один і той самий запис на різних пристроях має різну назву.

Оскільки пам’ять на носієві не є нескінченною, додаток їх видаляє. Оскільки видалення даних у мережі неможливе, додаток видаляє їх локально. Перед видаленням додаток «запитує» у мережі, чи може він видалити дані. Після успішної відповіді, додаток має можливість видалити запис зі своєї копії таблиці та з носія. У разі негативної відповіді, додаток повинен спочатку передати дані комусь іншому, а вже потім видалити їх. Якщо передати дані нікому, додаток зберігає ці дані й надалі.

Додаток може обмежувати обсяг даних, які він зберігає. Перед тим як прийняти дані, додаток дає свою згоду на це. Алгоритм роботи модуля даних досить простий. Послідовність дій для отримання даних, записаних у спільній пам’яті: модуль перевіряє, чи зберігаються необхідні дані на локальному комп’ютері; якщо дані зберігаються на комп’ютері, вони повертаються тому, хто їх

запитав; якщо дані не зберігаються, вузол запитує їх у будь-якого найближчого сусіднього вузла доти, поки вони не будуть знайдені та передає їх тому, хто їх запитав.

Модуль працює з пам’яттю так, що він не знає, чи запитав дані локальний комп’ютер чи ні, отже алгоритм отримання інформації працює так само під час пошуку на сусідніх вузлах, а знайдені дані передаються мережею. Алгоритм запису даних набагато простіший. Спочатку вузол, який хоче записати дані, визначає, чи є вони загальнодоступними. Якщо дані не загальнодоступні, вузол повинен їх зашифрувати та зберегти необхідний ключ. Після цього визначається ідентифікатор даних. Ідентифікатором є хеш, який містить наступні дані: власне дані; випадкове число; відбиток часу та адреса вузла, що записує дані. Такий набір гарантує генерацію різного хешу навіть для одних й тих самих даних.

Після визначення даних, типу даних та їх ідентифікатора, вузол записує їх локально. Якщо записати дані локально неможливо, наприклад вичерпано місце відведене під пам’ять, локальний запис пропускається. Далі дані та їх ідентифікатор надсилаються сусіднім вузлам. Сусідні вузли перевіряють, чи є цей ідентифікатор у їх пам’яті чи ні. Якщо міститься – операція на даному вузлі переривається. Інакше вузол перевіряє чи може записати дані, записує їх якщо може та надсилає своїм сусідам. Далі процедура повторюється для усіх вузлів і дані після публікації будуть на усіх вузлах, які можуть зберігати дані. Видалення даних дещо складніше. Оскільки вузол не може видаляти останню копію даних, він запитує у сусідніх вузлів чи може він видалити. Ті перевіряють з сусідами. Якщо на двох рівнях не знайдено копії даних, видаляти їх не можна. Інакше – вузол може видалити локальну копію.

4.7 Модуль та алгоритм маршрутизації

Модуль маршрутизації – один з найважливіших у мережі. Саме він відповідає за побудову маршруту від відправника до отримувача. Основне і єдине призначення модуля – побудова маршруту. Маршрут визначає відправник повідомлення, тому він може бути будь-яким і відмінним від попереднього

маршруту. Завдяки шифруванню маршрут є надійно захищеним і доступним лише певним вузлам. Маршрут зберігається на транзитному рівні тому є загальнодоступним, проте щоб отримати інформацію щодо вузлів необхідно мати певні ключі шифрування. Кожен сегмент ланцюга шифрується окремо і доступний лише вузлу, для якого він зашифрований.

За основу маршрутизації в роботі обрано алгоритм цибулевої маршрутизації з однією відмінністю – шифрується не увесь пакет, а лише ланцюг адрес.

Такий підхід дозволяє безпечно та неочікувано передавати інформацію, оскільки маршрут передачі заздалегідь визначено клієнтом, а сам маршрут надійно зашифровано. Оскільки адресою вузла може бути будь-що, то після дешифрування зловмисник не знає чи правильно дані розшифровано. Модуль маршрутизації будує адресу у прямому напрямку. Побудувати зворотній ланцюг не будучи відправником неможливо. Для відправки будується новий ланцюг, що може сильно відрізнятися.

Як зазначено раніше, за основу взято цибулеву маршрутизацію. Після отримання списку вузлів, які треба додати до ланцюга, вони шифруються. Адреса приймача є останньою у ланцюзі. Сегменти ланцюга шифруються починаючи з кінця попереднім сегментом, тобто для кожного сегмента використовується окремий ключ. Чим ближче сегмент до кінця, тим більше разів його зашифровано, таким чином кількість шифрувань можна вирахувати за формулою 4.1.

$$N_e = i - 1 \tag{4.1}$$

де N_e – кількість шифрувань; i – номер поточного сегмента ланцюга.

Після отримання вузлом пакета та перевірки, чи є він транзитним, вузол декодує своїм ключем усі інші сегменти ланцюга, видаляє свій сегмент та передає пакет далі. Таким чином усі сегменти ланцюга не будуть знати його справжню довжину, що збільшує анонімність оскільки не знаючи ланцюга відновити маршрут, хоча і віртуальний, неможливо. Алгоритм побудови ланцюгу наведено у додатку В, а приклад коду з пошуку отримувача у лістингу 4.4.

Лістинг 4.4 – Метод пошуку отримувача

```
import command
```

```
        import find_reeciver.RequestPacket
        import find_reeciver.ResponsePacket
        from storages import ConnectionsStorage

        class FindReceiverCommand(command.Command):
            def __init__(self):
        self.connections =
ConnectionsStorage.ConnectionsStorage.get_instance()
            def handle(self, package: find_reeciver.RequestPacket):
                response = find_reeciver.ResponsePacket()
                if self.connections.is_connected(package.address):
                    response.is_found = False
                    response.connections = self.connections.get_addresses()
                else:
                    response.is_found = True
                self.send_response(response)
            def on_response(self, packet: find_reeciver.ResponsePacket):
                # process response
                pass
```

4.8 Модуль та алгоритм роботи додатків

Модуль додатків є найвищим модулем у мережі. Саме цей модуль відповідає за роботу сервісних додатків. Він виконує наступні функції: створення запита на виконання додатка; запуск додатка; комунікація між додатком та вузлом; комунікація між додатком та клієнтом; завершення роботи додатка. Додатки зберігаються у розподіленій пам’яті мережі у спеціальному форматі. Уся програма займає один файл, що містить наступні дані: властивості додатка; необхідне ПЗ додатка; скомпільована версія додатка; додаткові ресурси (якщо потрібні).

Оскільки запуск сторонніх додатків є небезпечною процедурою, додатки виконуються у спеціальному ізольованому просторі. Такий підхід відрізняється від віртуалізації оскільки не потребує додаткових ресурсів, тому швидкодія додатка вища порівняно з віртуалізацією. До того ж існує доступ до заліза та можна швидко конфігурувати, запускати та вбивати додатки без внесення змін у основну систему. Такий підхід широко використовується у сучасних сервісних додатках та дозволяє використовувати будь-яку мову програмування оскільки операційна системи виконання може бути відмінною від операційної системи вузла.

Для запуску додатка один із вузлів повинен опублікувати запит, який містить ідентифікатор необхідного додатка. Цей ідентифікатор є ідентифікатором розташування додатка у розподіленій пам’яті. Вузол, на якому опубліковано запит, копіює додаток на свій носій та зчитує системну інформацію. Після цього цей вузол робить широкомовне повідомлення, яке містить необхідні вимоги до вузла. Це повідомлення надсилається за допомогою команд. Вузли, які можуть виконати додаток, відсилають свою згоду. Після отримання згоди, вузол, який опублікував запит, закриває його та наступні згоди відхиляються. На основі цього запиту створюється роутер – віртуальна одиниця системи яка зв’язує додаток та користувача. Цей роутер відіграє центральну роль у роботі додатка: через нього пересилаються дані додатку та клієнта, виконуються команди. Роутер працює доти доки додаток виконує роботу. По завершенню роботи додатка роутер знищується. Модель роботи системи під час виконання додатку зображена на рисунках 4.4 та 4.5.

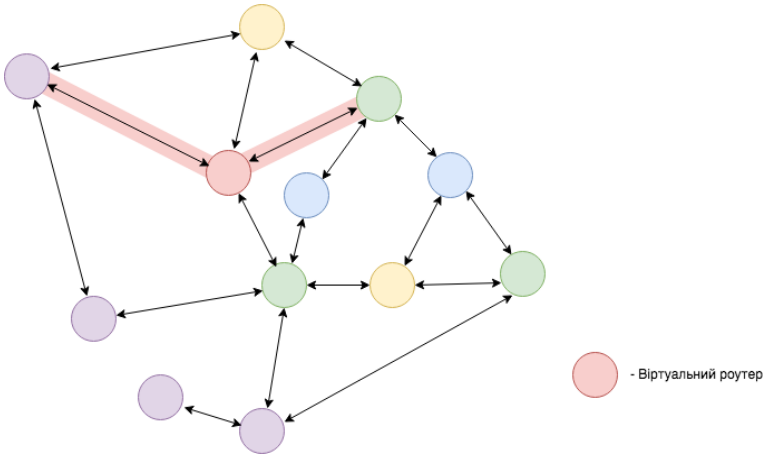

Рисунок 4.4 – Модель системи з віртуальним каналом (1)

Алгоритми запуску додатку, надсилання даних додатку та публікація додатку наведено у додатку Г (рис Г.1 - Г.3). Алгоритм передачі інформації від додатка користувачу виконується у зворотному напрямку в алгоритмі передачі від користувача додатку.

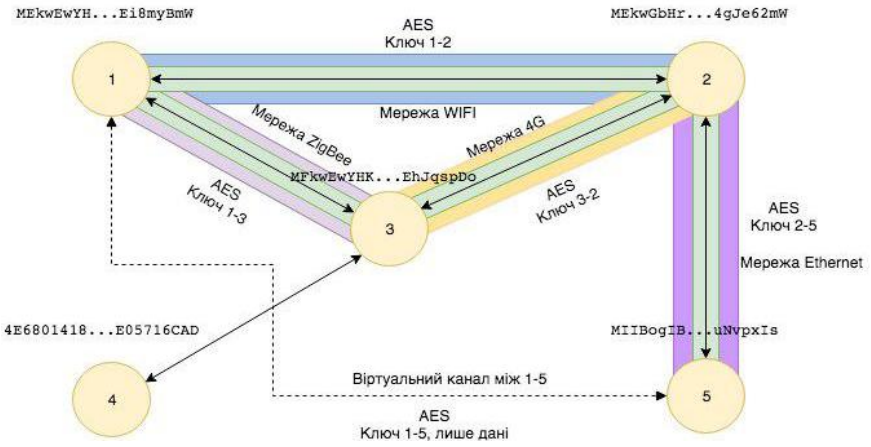

Рисунок 4.5 – Модель системи з віртуальним каналом (2)

Як бачимо, додатки працюють за децентралізованою схемою. Є підтримка роботи двох чи більше сервісі одночасно, але усі вони залежать від одного роутера. При виході вузла, що виконує роль роутера для певного сеансу виконання додатка, робота додатка переривається але додаток може зберегти тимчасові дані або надіслати їх користувачу, який запитав виконання сервісу. Передачі виконуються не використовуючи роутер і не відносяться до сеансу виконання, тому клієнт і додаток повинні самі передбачити такий розвиток подій.

Для взаємодії додатка та мережі, додаток взаємодіє з вузлом, на якому він запущений. Додаток може надсилати дані користувачу, читати та записувати їх у децентралізовану пам’ять. Користувач може виконувати команди, які містяться у додатку. Оскільки доступ до мережі є небезпечним через те, що додатки можуть використовуватись зловмисниками, додаток має значні обмеження щодо роботи з мережею. Він не має доступу до мережі безпосередньо, оскільки у ізольованому середовище це повинно бути заблоковано. Для того, щоб виконати якийсь запит до мережі, додаток повинен довести право володінням цим ресурсом. Для цього під час публікації додатка необхідно вказати до яких інтернет ресурсів повинен мати доступ додаток. Вузол, який публікує додаток, повинен перевірити володіння ресурсом за допомогою зчитування попередньо заданої послідовності символів з попередньо вказаного файлу. Якщо файл прочитати не вдалось, або послідовність інша, заявка на публікацію відхиляється. При успішному підтверджені, додаток

публікується разом з переліком підтверджених ресурсів. Для того, щоб зробити запит, додаток надсилає запит вузлу, на якому працює. Вузол перевіряє, чи є ресурс підтвердженим. Якщо підтверджено – запит додається у чергу запитів до ресурсів. Кожну секунду вузол може виконати обмежену кількість запитів, тому запити виконуються у режимі FIFO. Після виконання запиту, вузол надсилає його додатку.

5 ТЕСТУВАННЯ СИСТЕМИ

Для тестування створено додаток, що виконує множення матриць, працює в одному потоці та написаний на мові програмування Java. За сценарієм тестування, по запиту користувача, отримали матрицю, яка є результатом множення двох вхідних матриць розміром 200x200. Тестування проводилось на локальному та віддаленому вузлах. На локальному - додаток запускався у звичайному середовищі операційної системи, а на віддаленому – у ізольованому середовищі. В процесі тестування виконано 50 множень матриці. Елементи матриці – числа довжиною вісім байт. Результат тестування (рис.5.1) та код додатку (лістинг 5.1) наведено далі.

Лістинг 5.1 – Множення матриць

```
package com.company;
import java.util.Random;
public class Main {
    private final Integer MATRIX_MULTIPLIER_ROUNDS = 100;
    private Main() {
    }
    public static void main(String[] args) {
        Main main = new Main();
        main.run();
    }
    private void run() {
        int round = 50;
        while (round < MATRIX_MULTIPLIER_ROUNDS) {
            int matrixSize = 200;
            long matrixA[][] = new long[matrixSize][matrixSize];
            long matrixB[][] = new long[matrixSize][matrixSize];
            long matrixC[][] = new long[matrixSize][matrixSize];
            Random rand = new Random();
            for (int i = 0; i < matrixSize; i++) {
                for (int j = 0; j < matrixSize; j++) {
                    matrixA[i][j] = rand.nextLong();
                    matrixB[i][j] = rand.nextLong();
```

```
            }
        }
        long startedAt = System.currentTimeMillis();
        for (int i = 0; i < matrixSize; i++) {
          for (int j = 0; j < matrixSize; j++) {
            for (int k = 0; k < matrixSize; k++) {
                    matrixC[i][j] += matrixA[i][k] + matrixB[k][j];
                } } }
        long endedAt = System.currentTimeMillis();
        System.out.println(endedAt - startedAt);
        round++;
    } } }
```

Для побудови графіка використано частину замірів. Для тестування використовувався неоптимальний алгоритм оскільки на цьому можна наглядно побачити різницю у швидкодії додатків.

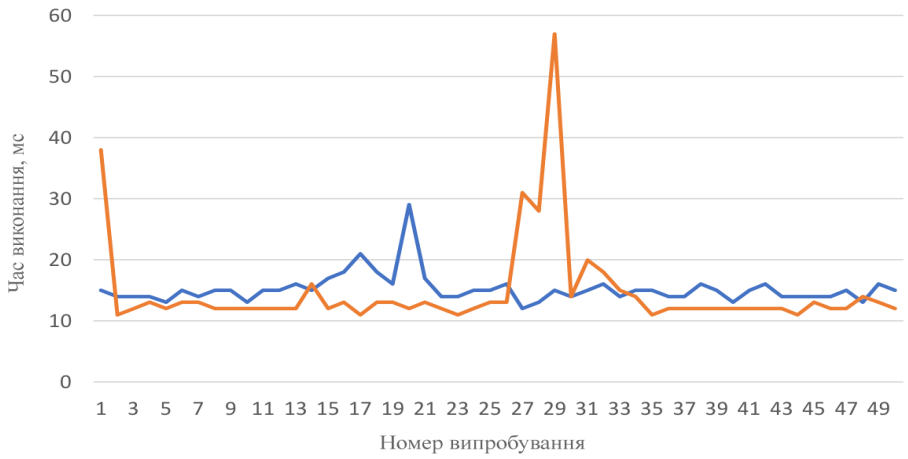

Рисунок 5.1 – Порівняння часу множення у звичайному та ізольованому середовищах.

На рисунку безперервна лінія відноситься до порівняння множення у звичайному середовищі, а пунктирна лінія – у ізольованому середовищі.

Параметри комп’ютера, на якому виконувалось тестування: процесор: Intel Core i5 4278U; об’єм оперативної пам’яті: 8 ГБ 1600 МГц DDR3.

Операційна система вузла має останні оновлення. Відмінними є операційні системи, у яких були запущені додатки. Ізольоване середовище створюється на основі будь-якого дистрибутива Linux, тому обрана операційна системи Alpine, у той час як на локальному комп’ютері використовувалась macOS 10.13.6.

Провести тестування додатка у існуючих системах з підтримкою роботи децентралізованих сервісів неможливе тому, що ці сервіси та мови програмування не надають можливостей для тестування швидкодії додатків.

Під час виконання тестування вираховувався час на саме множення матриць. Час на пересилання даних та запуск додатка не враховується. За результатами тестування видно, що додатки у ізольованому середовищі можуть працювати значно швидше ніж у звичайному середовищі. Це пов’язано з тим, що під час виконання додатка його ніщо не «турбую» оскільки у ізольованому середовищі не працюються будь-які інші ємні додатки, отже додаток не переривається на інші операції.

Швидкість передачі інформації у системі значно падає через деякі причини: максимальна швидкість передачі не може перевищувати мінімальної швидкості між будь-яким сегментом ланцюга передачі; існує додатковий час на побудову ланцюга; дані, що передаються системою, шифруються; передаються додаткові дані разом з корисним навантаженням.

Під час тестування використовувався ланцюг з чотирьох сегментів два з яких є проміжними. Отримано наступні дані: середній розмір додаткових даних – 222 байта; розмір шифрованих даних – 256 байт. Варто зазначити, що через використання шифрування, мінімальний обсяг даних, що передано, складає близько 159 байт навіть при пересиланні декількох байт даних. Таким чином при використанні стандартного значення MTU у мережі Fast Ethernet (1500) та протоколу TCP замість передачі 1476 байт інформації можливо передати не більше 1000 байт (обмеження системи). Разом з шифруванням та додатковою інформацією обсяг інформації складає 1745 байт. Таким чином падіння швидкості передачі додатково складає приблизно 30 відсотків. Зазначимо, що під час передачі невеликих об’ємів інформації швидкість передачі є значно меншою, оскільки більше частину даних, що пересилаються, складають некорисні дані.

Відносно низька швидкість передачі компенсується швидкістю виконання додатків. Порівняти загальну швидкість роботи з існуючими системами неможливо

через їх принцип роботи, а саме через їх строге дотримання графіку обробки даних та затримки під час надсилання та отримання інформації.

ВИСНОВКИ

У зв’язку із стрімким розвитком технологій та розширенням кількості пристроїв, яким необхідне підключення до мережі, звичайні мережі скоро не зможуть обслуговувати користувачів. Через стрімкий розвиток інтернету речей звичне зберігання даних становиться небезпечним. З кожним днем вимоги щодо швидкості отримання даних та безпеки їх зберігання оновлюються – дані необхідно отримувати все швидше, а захищати їх більш надійніше. Виходом з усіх цих питань є децентралізація мережі.

В даній роботі вдосконалено процес роботи сервісних додатків у децентралізованих системах, створено модель децентралізованої мережі, а також розроблено протокол системи підтримки обчислень у децентралізованих мережах. Були розроблені методи взаємодії частин системи та компонентів додатків. Проведено тестування мережі на працездатність, за результатами якого наведено порівняно з існуючими системами.

Розроблена система може використовуватись як типовий приклад у багатьох сферах людської діяльності: у юридичній та фінансовій сферах при автоматизації підпису договорів, як третя незалежна сторона; у медицині в якості формування єдиної бази даних пацієнтів, з подальшою розробкою модуля «віртуальний лікар»; у інтернеті речей, як платформу роботи пристроїв та їх додатків, використовуючи прозору взаємодію різних частин системи між собою; у військовій сфері, як платформу для автоматизації машин та безпечної координації дій військових частин, використовуючи відкриті канали зв’язку. Ця система може використовуватись у кожній сфері де існує потреба у обробці, зберіганні чи передачі даних, особливо коли мережна платформа є децентралізованою.

В ході подальших досліджень планується вдосконалення розроблених методів та доопрацьовування протоколу з метою підтримки більшої кількості мережних та сервісних функцій для збільшення кількості сценаріїв використання.

ПЕРЕЛІК ПОСИЛАНЬ

1. IETF Tools (1999), “Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations”, available at: http://tools.ietf.org/html/rfc2501 (Accessed 25 November 2013).

2. Рудьковський О.Р. Децентралізація технологій / О.Р.Рудьковський, Г.Г.Киричек // матеріали IX Міжн. наук.-практ. конференції «Сучасні проблеми і досягнення в галузі радіотехніки, телекомунікацій та інформаційних технологій», 03–05 жовтня 2018 року, Запоріжжя. – 2018. – С.100-101.

3. Рудьковський О.Р. Децентралізовані системи та мережі / О.Р. Рудьковський, Г.Г. Киричек // Тези доповідей наук.-практ. Конф., Запоріжжя, 16–20 квітня 2018 р. / Редкол. : В. В. Наумик. – Запоріжжя : ЗНТУ, 2018. – С.1002-1003.

4. Тимошенко В.С. Mesh-мережі – розвиток безпровідних мереж / В.С. Тимошенко, Г.Г. Киричек // Тези доповідей наук.-практ. Конф., Запоріжжя, 16–20 квітня 2018 р. / Редкол. : В. В. Наумик. – Запоріжжя : ЗНТУ, 2018. – С.1004-1005.

5. Оверлейная сеть [Електронний ресурс] – Режим доступу: https://traditio.wiki/Оверлейная_сеть.

6. A Next-Generation Smart Contract and Decentralized Application Platform [Електронний ресурс]. – 2014. – Режим доступу: https://github.com/ethereum/wiki/wiki/White-Paper.

7. Rivest R. The MD5 Message-Digest Algorithm [Електронний ресурс] / Rivest. – 1992. – Режим доступу: https://www.ietf.org/rfc/rfc1321.txt.

8. Wang X. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD [Електронний ресурс] / X. Wang, D. Feng, X. Lai. – 2004. – Режим доступу: https://eprint.iacr.org/2004/199.pdf.

9. Klima V. Tunnels in Hash Functions: MD5 Collisions Within a Minute [Електронний ресурс] / V. Klima. – 2006. – Режим доступу: https://eprint.iacr.org/2006/ 105.pdf.

10. Bosselaers A. The hash function RIPEMD-160 [Електронний ресурс] / A.Bosselaers.–Режим доступу: http://homes.esat.kuleuven.be/~bosselae/ripemd160.html.

11. Eastlake D. US Secure Hash Algorithm 1 (SHA1) [Електронний ресурс] / D. Eastlake, P. Jones – Режим доступу: https://tools.ietf.org/html/rfc3174.

12. Announcing the first SHA1 collision[Електронний ресурс]. – 2017. – Режим доступу: https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html.

13. Eastlake D. US Secure Hash Algorithms [Електронний ресурс] / D.Eastlake, T.Hansen. – 2006. – Режим доступу: https://tools.ietf.org/html/rfc4634.

14. Почему Keccak настолько крут и почему его выбрали в качестве нового SHA-3 [Електронний ресурс]. – 2013. – Режим доступу: https://habr.com/post/168707/.

15. Announcing the advanced encryption standard (AES) [Електронний ресурс]. – 2001. – Режим доступу: http://www.impic.org/papers/Aes-192-256.pdf.

16. Karn P. The ESP Triple DES Transform [Електронний ресурс] / P. Karn, P. Metzger, W. Simpson. – Режим доступу: https://tools.ietf.org/html/rfc1851.

17. Ship M. Cryptanalysis of Twofish (II) [Електронний ресурс] / M. Ship, L. Yiqun – Режим доступу: https://www.schneier.com/twofish-analysis-shiho.pdf.

18. Rivest R. Ryptographic communications system and method [Електронний ресурс] / R.Rivest, A.Shamir, L.Ademan, 1977 – Режим доступу: https://patentimages.storage.googleapis.com/49/43/9c/b155bf231090f6/ US4405829.pdf.

19. Rescorla E. Diffie-Hellman Key Agreement Method [Електронний ресурс] / Rescorla. – 1999. – Режим доступу: https://tools.ietf.org/html/rfc2631.

20. Digital signature standard [Електронний ресурс]. – 1994. – Режим доступу: http://www.umich.edu/~x509/ssleay/fip186/fip186.htm.

21. Corbellini A. Elliptic Curve Cryptography: a gentle introduction [Електронний ресурс] / Andrea Corbellini. – 2015. – Режим доступу: http://andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/.

22. Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography [Електронний ресурс]. – 2007. – Режим доступу: https://csrc.nist.gov/ publications/detail/sp/800-56a/revised/archive/2007-03-14.

ДОДАТОК А

Лістинг А.1 – Модуль мережі

```
import socket
import sys
import traceback
from threading import Thread
import sys, os, re
#
sys.path.insert(0, os.path.dirname(os.getcwd()))
#
from serialization import Serializer
from packages import Package, Date
from instructions import Commands
from encryption import ECAlgorithm
from storages import ConnectionsStorage
from routing import Router


class Node():
    def __init__(self):
        self.host = "0.0.0.0"
        self.port = 6666
        self.pack = Package.Package()
        self.com = Commands.Commands()
        self.dat = Data.Data()
        self.ser = Serializer.Serializer()
        self.pubkey = ECAlgorithm.ECAlgorithm()
        self.router = Router.Router()
    def run(self):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,1)
        try:
            self.sock.bind((self.host, self.port))
        except:
            print("Bind failed. Error : " + str(sys.exc_info()))
            self.sock.exit()
        self.sock.listen(5)       # queue up to 5 requests
        while True:
            connection, address = self.sock.accept()
            ip, port = str(address[0]), str(address[1])
            print("Connected with " + ip + ":" + port)
            try:
                Thread(target=self.client_thread, args=(connection, ip,
port)).start()
            except:
                print("Thread did not start.")
```

```
                traceback.print_exc()
        self.sock.close()
    def client_thread(self, connection, ip, port):
        is_active = True
        while is_active:
            client_input = self.receive_input(connection)
            chain_element = self.pack.package['node_chain'][0]
            if chain_element == self.pubkey.getPublicKey():
                self.ser.deserialize(self.pack)
                if self.pack.__class__.__bases__[0].__name__ is “Command”
                    self.com.handle(self.pack, connection)
            else:
                self.router.sendNext(chain_element, self.pack)
    def receive_input(self, connection):
        client_input = connection.recv(8)
        client_input_size = int(client_input)
        client_input = connection.recv(client_input_size)
        client_input_size = sys.getsizeof(client_input)
        decoded_input = client_input.decode("utf8")
        result = self.prepare(decoded_input)
        return result
    def prepare(self, input_str):
        self.pack.unserialize(input_str)
        return self.pack
```

ДОДАТОК Б

Лістинг Б.1 – Код шифрування при використанні алгоритму AES

```
import base64
from Crypto import Random
from Crypto.Cipher import AES
class AESAlgorithm():
    def setMyPublickKey(self, PublicKey):
        self.publicKey = PublicKey
    def setSharedKey(self, sharedKey):
        self.sh_KEY =  sharedKey
    def encrypt(self, message):
        raw = self._pad(message)
        iv = Random.new().read(AES.block_size)
        cipher = AES.new(self.sh_KEY, AES.MODE_CBC, iv)
        return base64.b64encode(iv + cipher.encrypt(raw))
    def decrypt(self, message):
        enc = base64.b64decode(message)
        iv = enc[:AES.block_size]
        cipher = AES.new(self.key, AES.MODE_CBC, iv)
        return self._unpad(cipher.decrypt(enc[AES.block_size:]))
.decode('utf-8')
    def _pad(self, s):
        return s + (self.bs - len(s) % self.bs) * chr(self.bs - len(s) %
self.bs)
```

Лістинг Б.2 - Приклад коду для генерації пари ключів за алгоритмом Elliptic Curve

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives.kdf.hkdf import HKDF

class ECAlgorithm(object):
    def __init__(self, arg):
        self.parameters = dh.generate_parameters(
        generator=2,
        key_size=256,
        backend=default_backend())
    def generateKeyPair(self):
        self.private_key = self.parameters.generate_private_key()
        self.public_key =
self.parameters.generate_private_key().public_key()
    def savePrivateKey(self):
        f = open('myprivatekey.pem','wt')
        f.write(self.private_key)
        f.close()
    def savePublicKey(self):
        f = open('myPublickey.pem','wt')
        f.write(self.public_key)
        f.close()
    def saveKeyPair(self):
        self.savePrivateKey()
        self.savePublicKey()
    def getPublicKey(self):
        return bytes(self.public_key, 'utf-8')
    def getPrivateKey(self):
        return bytes(self.private_key, 'utf-8')
```

Лістинг Б.3 – Приклад коду для генерації спільного ключа

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.kdf import x963kdf

class ECDHAlgorithm():
    def __init__(self):
        self.pub_key = None
        self.priv_key = None
    def setPublickKey(self, publick_key):
        self.pub_key = publick_key
    def setPrivateKey(self, private_key):
        self.priv_key = private_key

    def getSharedKey(self):
        shared_key = self.priv_key.exchange(ec.ECDH(), self.pub_key)
        xkdf = x963kdf.X963KDF(
            algorithm=hashes.SHA256(),
            length=32,
```

```
    sharedinfo='',
    backend=default_backend()
)
derived_key = xkdf.derive(shared_key)
return derived_key
```

ДОДАТОК В

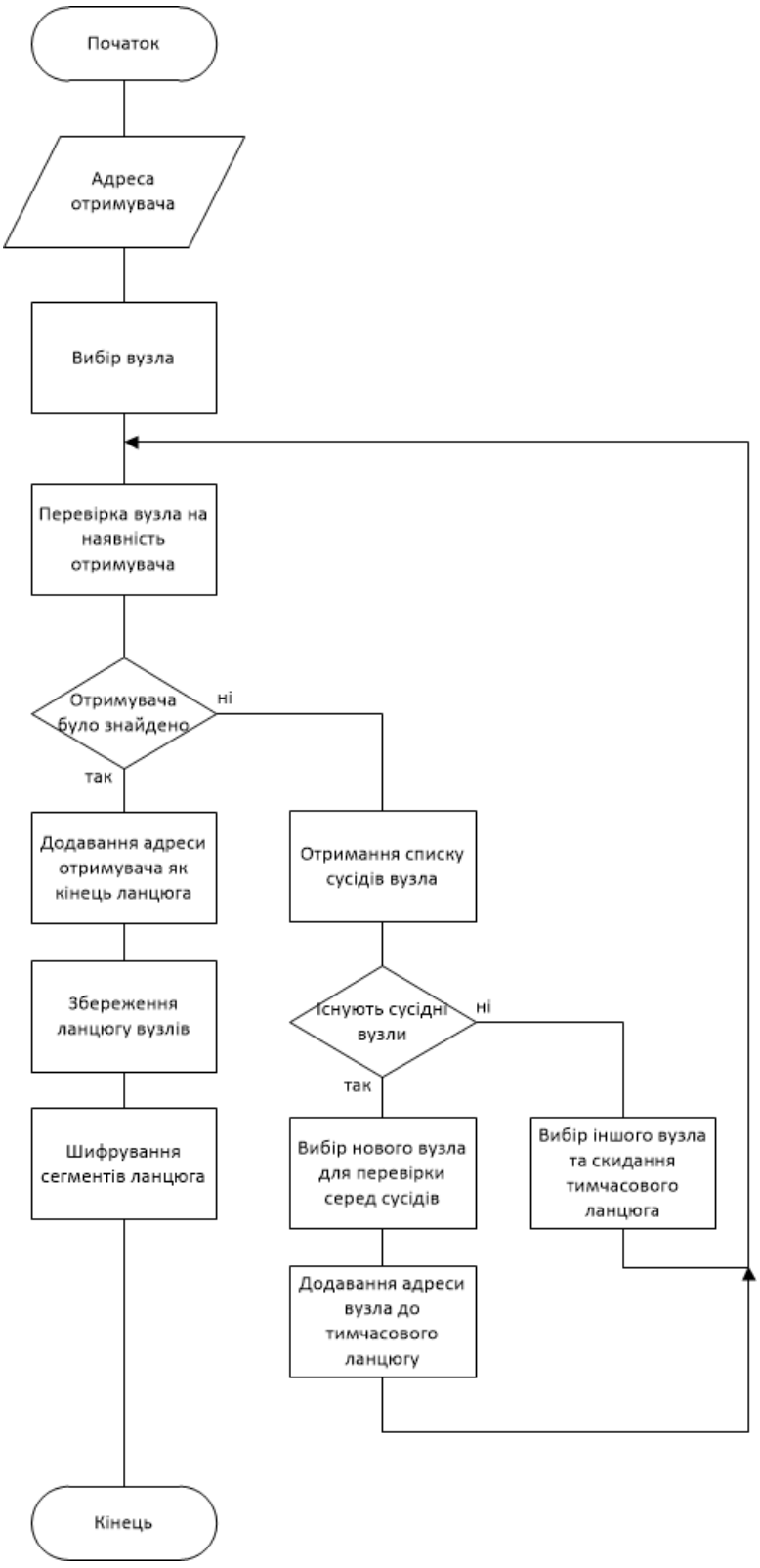

Рисунок В.1 – Алгоритм побудови ланцюгу вузлів для передачі інформації

ДОДАТОК Г

Рисунок Г.1 – Алгоритм запуску додатка

Рисунок Г.2 – Передача даних додатку

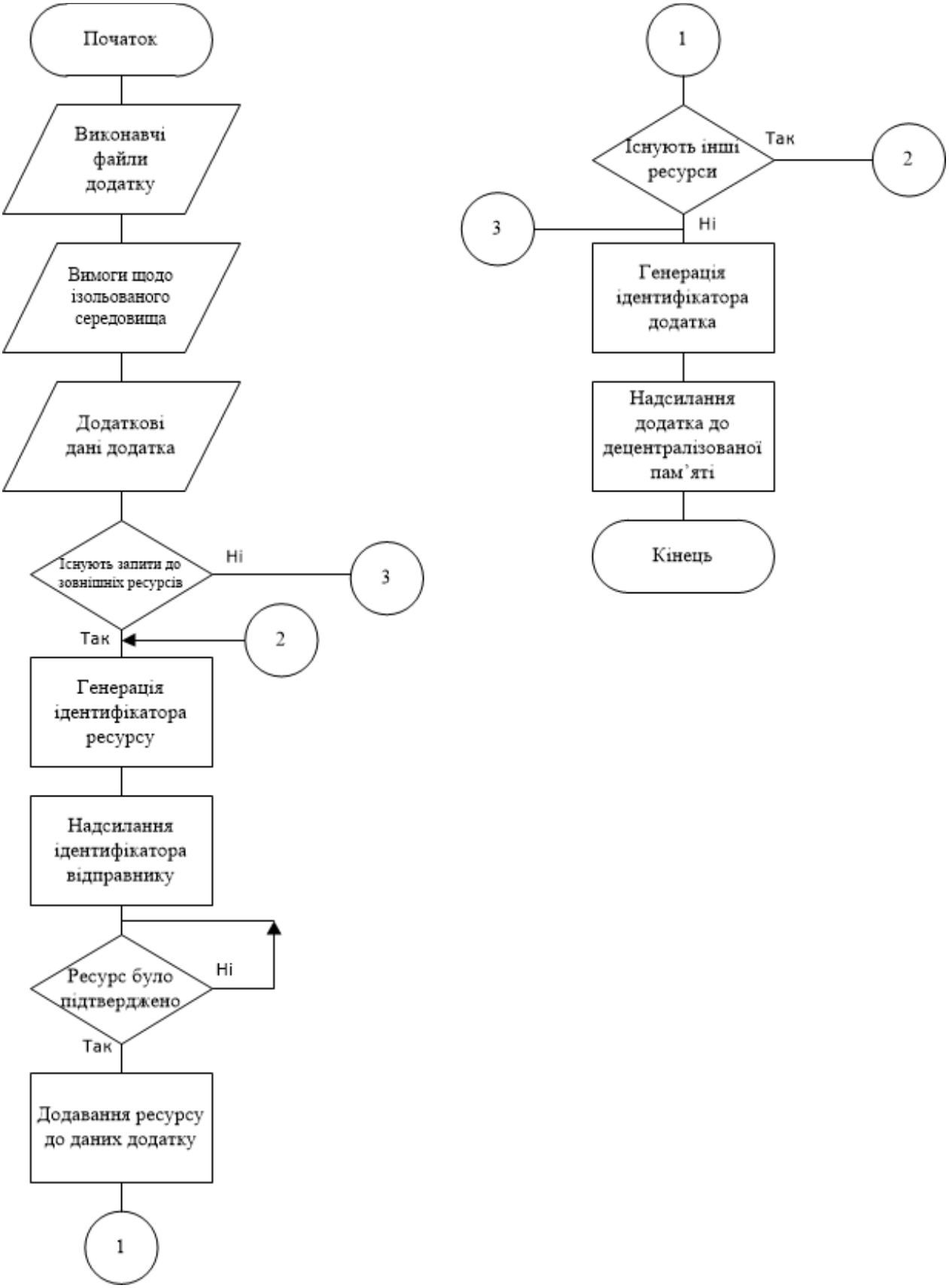

Рисунок Г.3 – Алгоритм публікації додатку