МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В.Н.Каразіна

Факультет математики і інформатики

Кафедра теоретичної та прикладної інформатики

Кваліфікаційна робота

магістр

на тему “Уніфікована математична модель для статичного та динамічного аналізу розподілених обчислень”

Виконав: студент 6 курсу, групи МФ-61

спеціальність 122 «Комп’ютерні науки»

освітньо-професійна програма

«Інформатика» Ільїн І.І.

Керівник: д. техн. н.,

проф. Жолткевич Г.М.

Рецензент:

ХАРКІВ - 2023

ЗМІСТ

ВСТУП

Актуальність вивчення розподілених обчислень пов'язана з їх повсякденним впливом на наше життя, адже найбільш розповсюджений приклад системи з розподіленими обчисленнями – мережа інтернет [1]. Ідея розподілених обчислень виникла у 1960-х і 1970-х роках, коли вчені почали досліджувати концепцію мережевих обчислень. Пізніше теорія і практика розподілених обчислень інтенсивно розвивалась і знайшла застосування у бізнесі, науці та сучасній обчислювальній техніці. Сьогодні розподілені системи є повсюдною частиною сучасних обчислень, які використовуються скрізь: від веб-додатків до соціальних мереж і наукового моделювання. Розвиток нових технологій, таких як контейнеризація, мікросервіси та безсерверні обчислення, розширює межі застосування розподілених обчислень.

З іншого боку, для верифікації на тестування програм були винайдені статичний та динамічний аналізи систем. Перші інструменти статичного аналізу були розроблені в 1960-х і 1970-х роках, коли вчені-комп’ютерщики почали досліджувати шляхи підвищення якості та надійності програмного забезпечення. Динамічний аналіз, з іншого боку, з’явився як галузь дослідження у 1980-х і 1990-х роках із розвитком тестування програмного забезпечення та засобів налагодження програмного забезпечення. Важливість статичного та динамічного аналізу полягає в їх здатності виявляти дефекти програмного забезпечення та запобігати їм. Використовуючи ці методи, розробники можуть виявити потенційні проблеми на ранніх стадіях процесу розробки та запобігти їх перетворенню в більш серйозні проблеми пізніше.

Метою даної роботи є побудова уніфікованої математичної моделі для статичного та динамічного аналізу розподілених обчислень.

Для досягнення поставленої мети необхідно було вирішити наступні завдання дослідження:

1. Провести аналіз літературних першоджерел за темою дослідження.

2. Обґрунтувати вибір методів математичного моделювання розподіленої системи.
3. Побудувати модель вимог і архітектуру системи.
4. Проаналізувати розроблену модель вимог та обрати мову реалізації.
5. Реалізувати побудовану математичну модель мовою програмування.
6. Обрати алгоритм, на якому буде апробовано модель. Провести тестування моделі.
7. Провести узагальнення отриманих результатів.

Новизна роботи полягає у подальшому розвитку математичної моделі розподілених обчислень, що запропонувала Ненсі Лінч, і її практичного застосування.

Практичне значення цієї роботи полягає у можливості застосування розробленої програми у навчальних цілях – графічна демонстрації роботи розподіленої системи на вже описаних алгоритмах. Студенти зможуть спробувати власноруч реалізувати розподілений алгоритм з книги у розробленому модулі. Побудований модуль може також використовуватися у науково-дослідницьких цілях для тестування власних ідей алгоритмів на розподілених системах.

РОЗДІЛ 1
ОКРЕМІ ВІДОМОСТІ З ТЕОРІЇ РОЗПОДІЛЕНИХ ОБЧИСЛЕНЬ, ДИНАМІЧНОГО ТА СТАТИЧНОГО АНАЛІЗІВ СИСТЕМ

1.1 Концепція розподілених обчислень, стан, конфігурація та комунікація систем.

Під час розвитку обчислювальної техніки задля зручності, гнучкості, пришвидшення швидкості обчислень, відсутності прив’язки до географічного положення, вчені вирішили розподілити обчислення на декілька машин або процесів, що і дало поштовх для розвитку розподілених систем та обчислень. Під *розподіленою системою* ми розуміємо позначення взаємопов’язаної сукупності автономних комп’ютерів, процесів або процесорів [2, 5]. Останні називаються *вузлами розподіленої системи*. Щоб вважатися *автономними*, вузли мають бути оснащені власним приватним контролем. Щоб вважатися *взаємозв’язаними*, (такі вузли ми будемо досліджувати надалі) вузли повинні мати можливість обмінюватися інформацією.

Оскільки процеси можуть відігравати роль вузлів системи, під це визначення підпадають системи програмного забезпечення, створені як сукупність процесів, що взаємодіють, навіть якщо вони працюють на одній установці апаратного забезпечення. Наприклад, так працюватиме бібліотека multiprocessing у python [6]. Однак частіше розподілена система буде містити хоча б декілька процесорів або машин/вузлів, з’єднаних апаратним забезпеченням зв’язку, тобто кабелем або системою кабелів.

Під *розподіленими обчисленнями* ми розуміємо обчислення та алгоритми, розроблені для роботи на апаратному забезпеченні, що складається з багатьох взаємопов’язаних процесорів [3]. Фрагменти розподіленого алгоритму працюють одночасно і незалежно, кожен з певним обмеженим обсягом інформації. Алгоритми мають працювати правильно, навіть якщо окремі процесори та канали зв'язку працюють на різних швидкостях і навіть

якщо деякі компоненти виходять з ладу [3]. Деякі алгоритми розраховані і на втрату повідомлень (рис. 1.1).

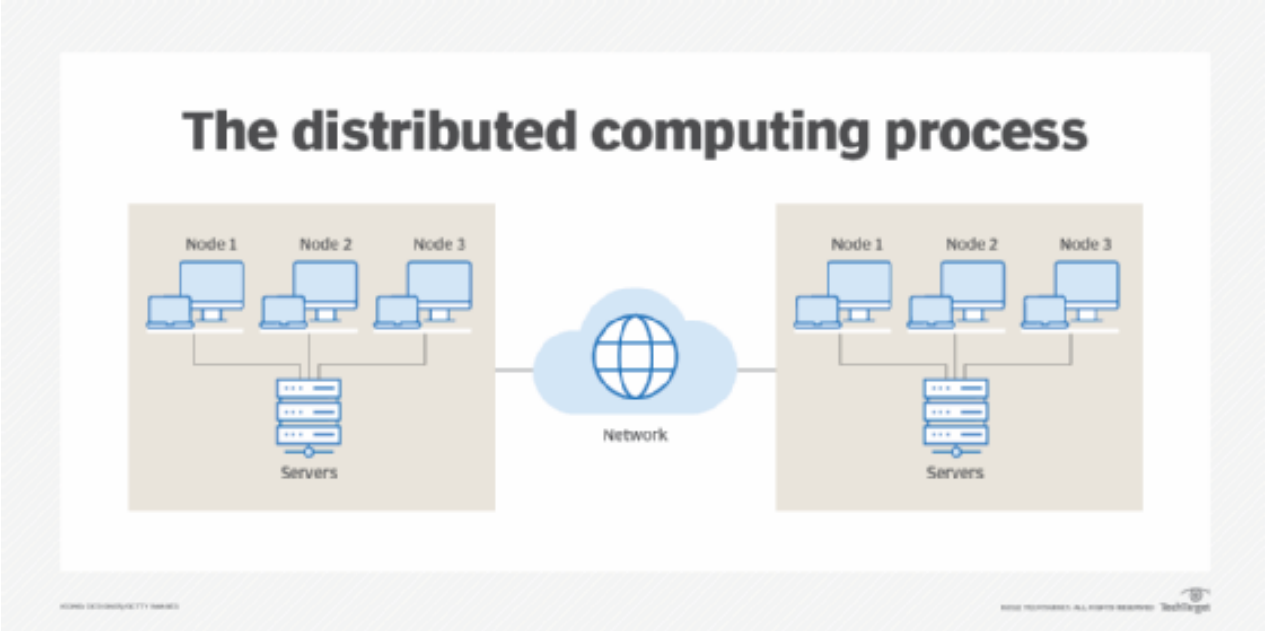


Рис. 1.1. Ілюстрація розподілених обчислень

Розподілений алгоритм, який працює в системі, забезпечує певні специфікації для окремих процесів. Глобальний стан розподіленого алгоритму, який називається *конфігурацією [4]*, розвивається за допомогою переходу станів її компонент. Загальна поведінка розподіленої системи характеризується системою переходів, яка складається з:

- набір C конфігурацій,
- бінарне відношення переходу $\rightarrow$ на C, та
- множина $I \subseteq C$ *початкових конфігурацій*.

Конфігурація γ є *термінальною*, якщо вона не має вихідного переходу: $\gamma \rightarrow \delta$ для жодного $\delta \in C$. Виконання розподіленої системи є послідовністю $\gamma_0\ \gamma_1\ \gamma_2$ конфігурацій, яка або нескінченна, або закінчується термінальною конфігурацією γ_k, такою що:

- $\gamma_0 \in I$,
- $\gamma_i \rightarrow \gamma_{i+1}$ для усіх $i \geq 0$ (в разі скінченного виконання $i < k$).

Конфігурація δ *досяжна*, якщо існує $\gamma_0 \in I$ та послідовність $\gamma_0 \gamma_1 \cdots \gamma_k$ з $\gamma_i \rightarrow \gamma_{i+1}$ для всіх $0 \leq i < k$ та $\gamma_k = \delta$ [4].

Конфігурація розподіленої системи складається з індивідуальних локальних станів її процесів і повідомлень, що передаються по каналах зв'язку. Перехід між конфігураціями розподіленої системи пов'язаний з подією в одному з її процесів. Процес може виконувати внутрішні, або локальні, події надсилати та отримувати події. Внутрішня подія впливає лише на стан процесу, де ця подія виконується. Типовими внутрішніми подіями є читання або запис у локальну змінну. Присвоєння нового значення змінній записується як ←; наприклад, n ← n + l означає, що значення змінної n, яка представляє натуральне число, збільшується на одиницю. Подія надсилання повідомлення породжує відповідну подію отримання того самого повідомлення в іншому процесі. Передбачається, що дві різні події ніколи не відбуваються в один і той самий момент у реальному часі (за винятком синхронного зв'язку [7]).

Процес є *ініціатором*, якщо його перша подія є внутрішньою подією або подією надсилання; тобто ініціатор може почати виконувати події без введення з боку іншого процесу. Алгоритм називається *централізованим*, якщо у нього рівно один ініціатор. *Децентралізований* алгоритм може мати кілька ініціаторів [3].

Знімок виконання розподіленого алгоритму – це конфігурація цього виконання, що складається з локальних станів процесів і повідомлень, що передаються [4]. Миттєві знімки необхідні, щоб зрозуміти стан системи у конкретний час. Крім того, знімки можна використовувати для контрольних

точок для перезапуску після збою розподілених обчислень та нод системи або для дебагу роботи системи.

У централізованому середовищі можна в будь-який момент під час виконання програми запитати стан програми, що складається зі значень локальних змінних, саме так зазвичай і дебажать програми. У розподілених умовах так не вийде через те що процесів декілька, вони обмінюються повідомленнями, є проблема синхронізації годинників [8]. Припустимо, що процес з розподіленої системи хоче зробити знімок конфігурації поточного виконання. Тоді він повинен попросити всі процеси також зробити знімок свого локального стану. Крім того, процеси мають обчислювати стани каналів повідомлень, щоб захопити повідомлення, які були в дорозі на момент моментального знімка. Це важливо, бо інакше ми би загубили повідомлення, що залишилися у каналах зв'язку, хочу вони прямим образом впливають на стан системи. Алгоритм моментального знімка має працювати під час виконання, тобто без заморожування виконання основного алгоритму, для якого робиться знімок. Повідомлення базового алгоритму називаються *базовими повідомленнями*, тоді як повідомлення алгоритму моментального знімка називаються *контрольними повідомленнями [3]*.

Комунікацію розподіленої системи можна розглядати декількома способами. Так, наприклад, вузли мережі можуть знати або не знати о структурі мережі та своїх сусідах, тобто сама комунікація – це *black box*. У цьому випадку вузли системи не спілкуються між собою напряму, не знають топології мережі, вони знають лише про якийсь керуючий механізм мережі, якому вони надсилають повідомлення. А керуючий механізм вже пересилає повідомлення усім іншим вузлам або тим, кому він вважає за потрібне відправити (рис. 1.2).

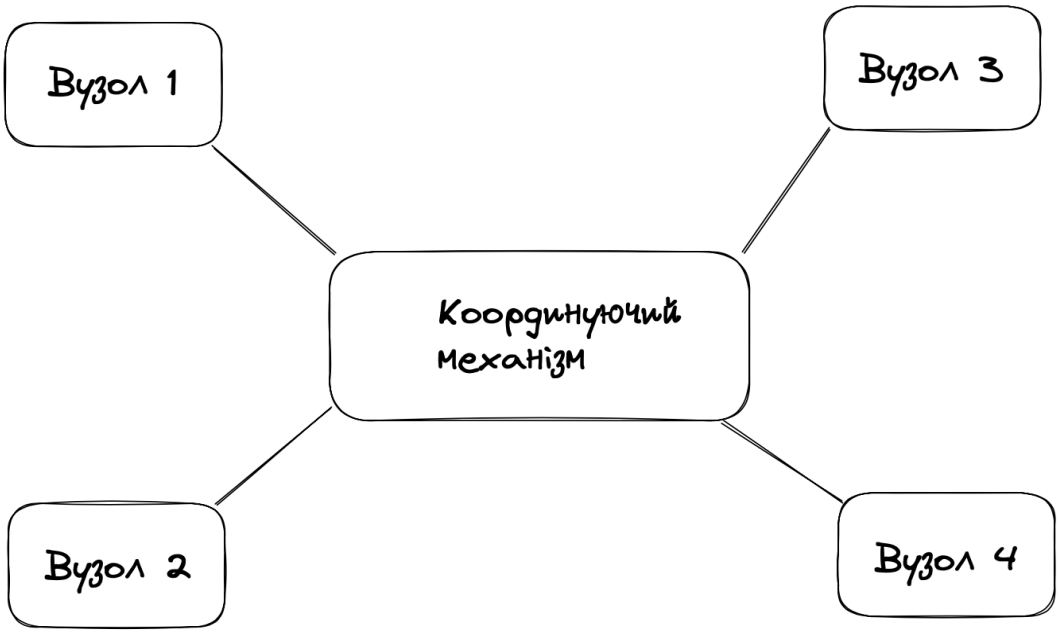
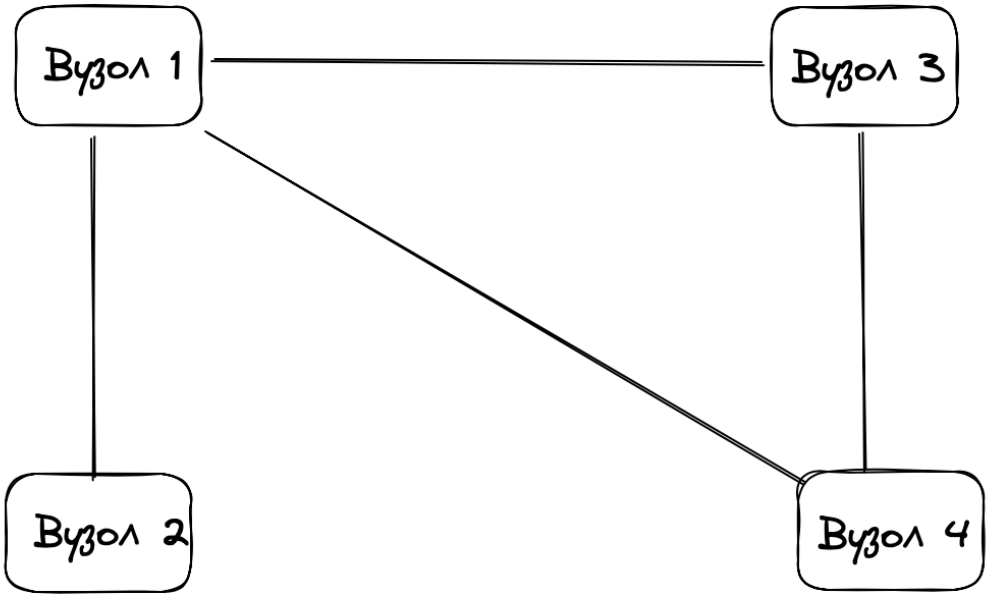


Рис 1.2. *Black box* комунікація мережі

Інший варіант комунікації мережі – *white box*. У цьому випадку, навпаки, вузли системи знають про своїх сусідів та кому вони відправляють повідомлення (рис. 1.3).



Ріс 1.3. *White box* комунікація мережі

У наведених схемах прямокутники позначають вузли системи, а риски між ними – канали зв’язку. Самі канали зв’язку можуть бути

односпрямованими або *двоспрямованими*, в залежності від того, як ходять повідомлення по каналу – в одну чи дві сторони (рис. 1.4).

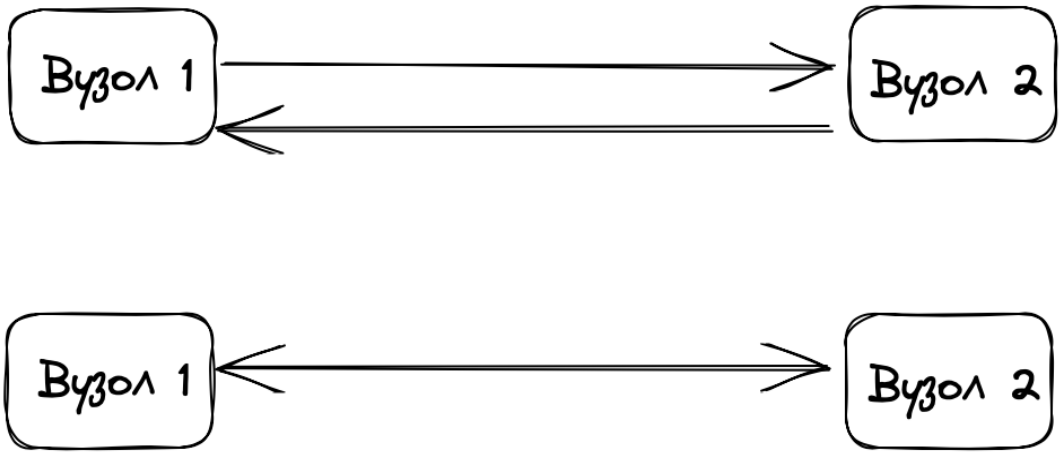


Рис 1.4. *Односпрямований* (зверху) та *двоспрямований* (знизу) канали

Один двоспрямований канал можно замінити двома односпрямованими в різні сторони.

1.2. Динамічний та статичний підхід до аналізу систем

Ми всі хотіли б, щоб програми перевіряли правильність наших програм. На сьогодні більшість людей, які пишуть програмне забезпечення у бізнесі, практикуючих і науковців, вважають, що витрати на офіційну перевірку програм того варті.

Статичний і динамічний аналізи є двома методами, які використовуються в розробці програмного забезпечення для виявлення та запобігання дефектів програмного забезпечення. Обидві методики мають довгу історію, починаючи з ранніх днів обчислювальної техніки.

Статичний аналіз передбачає перевірку вихідного коду програми без її виконання [9]. Цей аналіз зазвичай виконується за допомогою спеціалізованих інструментів, які можуть виявити потенційні проблеми, такі як синтаксичні помилки, логічні помилки та вразливі місця безпеки.

Дослідження механізованого доведення теорем почалося в другій половині 20-го століття, і деякі з найперших практичних робіт стосувалися Nqthm, «доказу теорем Боєра-Мура», який використовувався для підтвердження таких теорем, як правильність повного апаратного забезпечення і програмного стеку [10]. ACL2, наступник Nqthm, знайшов значне впровадження в галузі, наприклад, AMD для перевірки правильності одиниць ділення з плаваючою комою [11].

Приблизно на початку 21 століття прогрес у практичних застосуваннях інтерактивного доведення теорем значно прискорився. У царстві чистої математики Жорж Гонтьє побудував перевірений машиною доказ теореми чотирьох кольорів [12], математичної проблеми, вперше поставленої понад сто років тому, де єдині попередні докази вимагали довіри ad hoc програмне забезпечення для перевірки ключових фактів грубою силою. У сфері верифікації програм Ксав'є Леруа очолював проект CompCert для створення верифікованого компілятора C, достатньо надійного для використання з реальним вбудованим програмним забезпеченням [13].

Кілька відомих формальних розробок було здійснено в Coq – помічнику для перевірки логіки вищого порядку, що дозволяє розробляти комп'ютерні програми відповідно до їхньої формальної специфікації. Логічна мова, яку використовує Coq, є різновидом теорії типів, яка називається *численням індуктивних конструкцій*.

Перша реалізація CoC була розпочата в 1984 році G. Huet і T. Coquand. Він був реалізований мовою була CAML, функціональна мова програмування з сімейства ML, розроблена в INRIA в Рокенкурі. Ядром цієї системи був засіб перевірки доказів для CoC, який розглядався як типізоване λ-числення - Constructive Engine. Цей механізм керувався за допомогою нотації високого рівня, що дозволяло декларувати аксіоми та параметри, визначати математичні типи та об'єкти, а також явну конструкцію доказових об'єктів, закодованих як λ-терми. Механізм секцій, розроблений і реалізований Г. Довеком, дозволив ієрархічний розвиток математичних теорій. Цю мову високого рівня назвали

математичною мовою. Крім того, інтерактивний засіб перевірки теорем дозволяв поетапне конструювання дерев доказів зверху вниз, рекурсивно підцілюючи та повертаючись із глухих кутів. Довідник теорем виконував тактику, написану на CAML, у моді LCF. Був попередньо визначений базовий набір тактик, який користувач міг розширити своїми власними тактиками. Потім систему було розширено для роботи з новим численням з індуктивними типами К. Пауліна з відповідною новою тактикою для доказів індукцією. Було впорядковано новий стандартний набір тактик, а народну мову розширено для виконання тактик. Coq був перенесений на нову реалізацію Caml-light у 1992 році, але саме ядро і принцип праці залишилися [14].

Отже, ідея сучасного статичного аналізу полягає у побудові моделі за допомогою типізованого λ-числення та використання спеціальних тактик для доведення лем, теорем, та властивостей побудованої моделі.

Динамічний аналіз, з іншого боку, з'явився як галузь дослідження у 1980-х - 1990-х роках із розвитком тестування програмного забезпечення та засобів налагодження. Ці інструменти дозволяли розробникам відстежувати поведінку програм під час їх роботи та виявляти такі проблеми, як витоки пам'яті, конкуренцію потоків, вразливі місця безпеки, перевіряти коректність програми [15].

Історію динамічної перевірки можна простежити до ранніх днів обчислювальної техніки, коли програмісти використовували ручні методи для відстеження виконання програм і виявлення помилок. Оскільки обчислювальні системи ставали все більш складними, а розмір і складність програмних додатків збільшувалися, ручні методи стали непрактичними, і дослідники почали розробляти автоматизовані інструменти для допомоги в динамічній перевірці.

Одним із найперших прикладів інструменту динамічної перевірки є дебагер, який був розроблений у 1950-х та 1960-х роках. Дебагери дозволяли програмістам зупиняти виконання програми в певних точках і перевіряти значення змінних та інших структур даних для діагностики помилок.

На сьогодні є кілька типів методів динамічного аналізу, які можна використовувати для перевірки поведінки програм під час виконання. Ці техніки включають:

Дебаг – техніка, яка використовується для діагностики та виправлення помилок у програмному забезпеченні. Дебагери дозволяють програмістам зупиняти виконання програми в певних точках і перевіряти значення змінних та інших структур даних для діагностики помилок. Налагодження часто використовується в поєднанні з іншими методами динамічного аналізу, щоб виявити помилки та переконатися, що програма функціонує правильно [16].

Профілювання – техніка, яка використовується для вимірювання продуктивності програми шляхом відстеження часу, витраченого на виконання кожної функції або блоку коду. Профайлери можуть допомогти виявити вузькі місця продуктивності та інші проблеми, які можуть вплинути на швидкість і ефективність програми [17].

Аналіз пам’яті – техніка, яка використовується для відстеження виділення та звільнення пам’яті в програмі. Інструменти аналізу пам’яті можуть допомогти виявити витоки пам’яті, переповнення буфера та інші пов’язані з пам’яттю помилки, які можуть спричинити збій або несподівану поведінку програми [18].

Фаззинг – техніка, яка використовується для автоматичного створення та виконання великої кількості вхідних даних у програму для виявлення вразливостей безпеки. Інструменти фаззингу можуть допомогти виявити переповнення буфера, вразливості рядків форматування та інші проблеми, пов’язані з безпекою, які можуть спричинити неочікувану поведінку програми [19].

Перевірка під час виконання – техніка, яка використовується для моніторингу поведінки програми під час виконання для виявлення порушень безпеки. Інструменти перевірки під час виконання можуть допомогти виявити вразливі місця безпеки, як-от недійсний доступ до пам’яті, впровадження коду та переповнення буфера, якими можуть скористатися зловмисники [20].

Методи динамічної верифікації є невід'ємною частиною процесу розробки програмного забезпечення, допомагаючи переконатися, що програмні додатки функціонують правильно, працюють ефективно та безпечно.

Проблемою динамічної верифікації є те, що ми не можемо дати гарантію, що програма виконується коректно. Так, вона може пройти усі наші тести, бути без проблем пам'яті, стійкою та продуктивною, але не існує можливості стверджувати, що програма правильно побудови. Ми можемо написати лише кінцевий набір перевірок, наприклад 999 тестів, але де гарантія, що на тисячному тесті програма не зламається? Гарантію коректності програми може дати лише статична верифікація і доведення відповідної теореми. Проте, статична верифікація – це дорога річ, яка займає багато часу, грошей та потребує високого рівня кваліфікації працівника. Треба побудувати правильну модель і заздалегідь люди не знають, чи можливо взагалі доказати певні властивості цієї моделі, які нас цікавлять? З динамічною верифікацію ми швидше можемо зрозуміти, чи схожа програма на робочу. А потім за допомогою статичної верифікації, якщо треба, ми можемо довести її коректність.

Отже, підхід до аналізу програмного забезпечення має поєднувати статичний та динамічний аналізи.

1.3. Невирішені проблеми в галузі розподілених систем

Розподілені обчислення – це галузь, що швидко розвивається, яка передбачає проектування та впровадження систем, які працюють на декількох машинах або вузлах у мережі. Незважаючи на значний прогрес у цій галузі, все ще є кілька невирішених проблем у розподілених обчисленнях, які створюють проблеми для дослідників і практиків.

Деякі з невирішених проблем у розподілених обчисленнях включають:

Консенсус – це проблема досягнення згоди між групою вузлів у розподіленій системі, навіть за наявності збоїв. Проблема є складною, оскільки вузли можуть виходити з ладу або бути ненадійними, а повідомлення можуть затримуватись або втрачатися. Існує кілька запропонованих рішень проблеми консенсусу, наприклад алгоритми Paxos і Raft, але ці алгоритми мають обмеження та не працюють належним чином у всіх сценаріях [21].

Відмовостійкість – це здатність системи продовжувати функціонувати навіть за наявності збоїв. У розподіленій системі збої можуть виникнути на будь-якому рівні, від апаратних збоїв до програмних помилок. Забезпечення відмовостійкості в розподіленій системі є складною проблемою, що вимагає детального проектування та впровадження відмовостійких механізмів [22].

Масштабованість – це здатність системи обробляти зростаючу кількість користувачів або запитів без значного зниження продуктивності. Досягнення масштабованості в розподіленій системі є складним завданням, оскільки система повинна бути розроблена таким чином, щоб розподіляти робоче навантаження між декількома вузлами ефективним і ефективним способом [23].

Безпека – є критичною проблемою в розподілених системах, оскільки вузли можуть бути розташовані в різних фізичних місцях і можуть контролюватися різними організаціями або особами. Забезпечення безпеки розподіленої системи вимагає впровадження захищених протоколів зв’язку, механізмів аутентифікації та механізмів контролю доступу [24].

Узгодженість даних – це проблема забезпечення того, щоб усі вузли в розподіленій системі мали узгоджені представлення даних системи. Ця проблема є складною, оскільки вузли можуть оновлювати дані незалежно один від одного, а для поширення оновлень на всі вузли може знадобитися час. Забезпечення узгодженості даних у розподіленій системі вимагає детального проектування та реалізації механізмів синхронізації [25].

Таким чином, існує чимало актуальних складних проблем у галузі розподілених обчислень, і так як ця галузь відносно нова та стрімко

розвивається, можна бути певним, що список відкритих проблем буде розширюватися.

1.4. Уточнення формулювання задачі

В п.п. 1.1 наведені відомі результати та термінологія у сфері розподілених обчислень, а в п.п. 1.2 – відомі підходи статичної та динамічної верифікації програмного забезпечення. У даній роботі пропонується застосувати методи динамічного та статичного аналізів до побудованої математичної моделі розподілених систем. Протестувати модель, роботу динамічного та статичного аналізів можна на одному з класів відомих розподілених алгоритмів, наприклад, на класі алгоритмів збирання стану системи.

РОЗДІЛ 2
ЗАСТОСУВАННЯ ТЕОРІЇ ДИНАМІЧНОГО ТА СТАТИЧНОГО АНАЛІЗІВ ДО УНІФІКОВАНОЇ МАТЕМАТИЧНОЇ МОДЕЛІ РОЗПОДІЛЕНИХ ОБЧИСЛЕНЬ

2.1. Математична модель, застосування динамічного та статичного аналізів до цієї моделі

Математична модель розподілених обчислень була описана у працях Ж. Теля [2] та Н. Лінч [3]. Ці роботи базуються на схожих ідеях та введених понять множин переходу станів системи, початкових конфігурацій та множини усіх доступних станів у системі.

Ненсі Н. Лінч використовує введене поняття “I/O (input-output) автомат”. Автомат моделює компонент розподіленої системи, який може взаємодіяти з іншими компонентами системи. Це простий тип кінцевого автомата, в якому переходи пов’язані з названими діями. Дії класифікуються як вхідні, вихідні і внутрішні. Входи та виходи використовуються для зв'язку з середовищем автомата, тоді як внутрішні дії видимі лише самому автомату. Передбачається, що вхідні дії не знаходяться під контролем автомата. Вони просто надходять ззовні, тоді як сам автомат визначає, які вихідні та внутрішні дії мають бути виконані.

Прикладом типового I/O автомата є процес в асинхронній розподіленій системі. Автомат P_i, намальований у вигляді кола з вхідними стрілками, позначеними вхідними діями, і вихідними стрілками, позначеними вихідними діями. Внутрішні дії не відображаються. Зображений автомат отримує вхідні дані у формі init(v)_i із зовнішнього світу, які повинні представляти отримання вхідного значення v. Він передає виходи у формі decide (v)_i, які мають представляти рішення v. Щоб прийняти рішення, процес P_i може захотіти спілкуватися з іншими процесами за допомогою системи повідомлень. Його

інтерфейс із системою повідомлень складається з вихідних дій у формі $send(m)_{i,j}$, що представляє процес P_i, який надсилає повідомлення з вмістом m до процесу P_j, і вхідних дій у формі $receive(m)_{j,i}$, які представляє процес P_i, який отримує повідомлення з вмістом m від процесу P_j (рис. 2.1) [3].

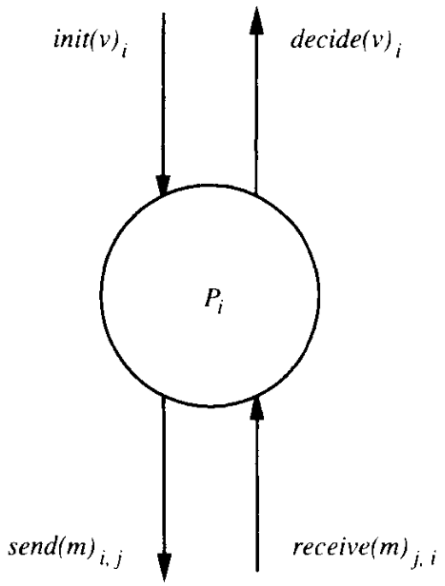


Рис 2.1. I/O автоматон процесу P_i

Цікаво, що FIFO канали у системі також можна представити у вигляді I/O автомату. Його вхідні події є подіями типу відправки повідомлення іншим автоматам – send (m), а вихідні події – події отримання повідомлення іншим автоматом – receive (m) (рис. 2.2).

Рис 2.2. I/O автомат фіфо каналу P

Якщо казати про формальне описання автомата, перше, що вказується, це його «підпис», який є просто описом його входу, виходу та внутрішніх дій. Ми припускаємо універсальний набір дій. Сигнатура *S* – це трійка, що складається з трьох непересічних наборів дій: вхідні дії, in(S), вихідні дії,

out(S), і внутрішні дії, int(S). Ми визначаємо зовнішні дії, ext(S), як in(S) U out(S); локально керовані дії, local(S), як out(S) U int(S); і acts(S) – це всі дії S. Зовнішній підпис, extsig(S), визначається як підпис (in(S), out(S), 0). Ми часто називатимемо зовнішній підпис зовнішнім інтерфейсом.

I/O автомат A, який ми також називаємо просто автоматом, складається з п'яти компонентів:

- sig(A), підпис
- states(A), (не обов'язково скінченний) набір станів
- start(A), непорожня підмножина станів (A), відомих як початкові стани
- trans(A), відношення стан-перехід, де trans(A) ⊆ states(A) x acts(sig(A)) x states(A); це повинно мати властивість, що для кожного стану s і кожної вхідної дії π існує перехід (s, π ,s') ∈ trans(A)
- tasks (A), розділення задач, яке є відношенням еквівалентності на local(sig(A)), що має якнайбільш лічильну кількість класів еквівалентності

Розподілена система у Н. Лінч [3] будується з поєднання різних автоматів (процесів та каналів).

Ж. Тель будує свою математичну модель розподілених обчислень на терміні "перехідна система" [4]. *Перехідною системою* є трійка:

$$S = (C, \rightarrow, I)$$

де C – набір конфігурацій,

$\rightarrow$ бінарне відношення переходів на C,

I – підмножина C початкових конфігурацій.

Іншими словами, → потрібна для визначення переходу конфігурацій системи та зміни її станів. Надалі з цього терміну Ж. Тель вводить поняття розподіленого алгоритму та розширює поняття перехідної системи з процесу на всю розподілену систему.

Проте, підхід Ж. Теля не розглядає окремо канали як сутності. Навпроти, у Н. Лінч поняття I/O автомата можна використати до кожного елементу системи, тобто є уніфікований підхід як до системи в цілому, так і до кожної її частини. Також, підхід Н. Лінч не такий математично складний, що сприяє його практичному застосуванню у програмному забезпеченні, тому надалі ми обрали підхід Н. Лінч і будемо будувати розподілену систему над автоматами.

Проте, є проблема з будуванням програмного забезпечення лише над автоматами. Автомати-процеси та автомати-канали передбачають white-box комунікацію, бо канали під'єднуються до процесів і процеси знають лише про канали, які виходять з нього та входять до нього. Уявімо ситуацію: у програмній симуляції ми вирішили поставити програму на паузу, щоб подивитися, що відбувається у процесів, їх стан, що у каналах та інше. У white box комунікації ми були б вимушені ініціювати паузу на каналах або процесах, бо вони несуть відповідальність за комунікацію. Але немає способа одночасно зробити паузу на всі канали/процеси, так чи інакше ми будемо паузити процеси та канали один за одним. Це може призвести до того, що ми поставили на паузу процес або канал А, потім ми хочемо поставити на паузу процес Б, процес Б встигає відправити повідомлення до каналу А, виключається, що призводить до того, що у нас система не консистентна та стан системи не відповідає дійсності, бо повідомлення просто загубилося.

Щоб такої ситуації не було, ми маємо користуватися спеціальною комунікаційною системою, яку можна зупинити, тоді вся комунікація миттєво зупиниться. Така система також може управляти та працювати з будь-якими типами каналів, фіфо та звичайні, що додає гнучкості нашій моделі.

У своїй праці Н. Лінч припускає, що повідомлення, що відправляються автоматом-процесом, рано чи пізно дійдуть до одержувача. Це деяке обмеження, але розумне, зараз технології достатньо розвинені, щоб воно часто виконувалося. Проте, ми не можемо казати, що в дійсності повідомлення дійдуть з вірогідністю в 100%, наприклад щось може трапитися з каналом фізично - впало дерево, обрізали тощо. Наша комунікаційна система може

працювати з певною, заданою користувачем, вірогідністю доставки повідомлення. Проте, у нашій роботі ми будемо припускати, що повідомлення завжди дійде до адресата.

Отже, систему ми будемо будувати з наступними умовами:

- Повідомлення, що відправляються автоматом-процесом, рано чи пізно дійдуть до одержувача.
- Система комунікації – black box з елементами white box. White box присутня у тому сенсі, що канали знають про своїх сусідів, а black box - у наявності відповідальної комунікаційної системи.
- Канали – односпрямовані. Можна було б обрати двоспрямовані, але їх можна замінити двома однопрямованими, тож ця умова ніяк нас не обмежує.

Отже, ми будемо досліджувати розподілену систему, де автомати знають про своїх безпосередніх сусідів і обмінюються повідомленнями через односпрямовані канали зв’язку (рис. 2.3).

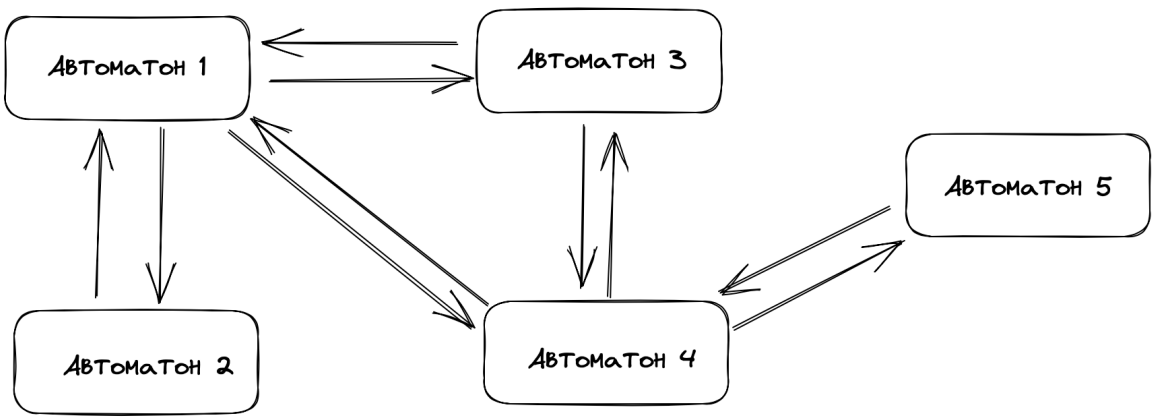


Рис 2.3. Приклад розподіленої системи, що задовольняє вимогам

Так як в дійсності ми не маємо розподіленої системи та її було б важко дослідити у реальному світі, ми маємо створити симуляцію розподіленої системи, де можемо відстежувати процеси, встановлювати різний термін доставки повідомлень, провести статичний та динамічний аналізи та протестувати їх. Статичний аналіз буде проведено за допомогою язика

програмування Coq, а динамічний аналіз за допомогою Python. Для цього ми побудуємо відповідну систему з автоматів на цих двох язиках. Для тестування системи я реалізую деякі алгоритми з класів алгоритмів збирання станів.

У симуляції розподіленої системи ми використаємо потоки, що будуть визначати автомати-процеси. Процеси в моделі ми поділяємо на процеси-ініціатори та звичайні процеси. Так як швидкість доставки повідомлень на локальному комп'ютері висока, що не дасть можливості передивится перебіг програми, то ми додамо спеціальні тіки, що будуть відповідати, через скільки кроків роботи алгоритму/каналу повідомлення буде дійсно доставлене.

У наступному розділі буде більше детально описана програмна реалізація розподіленої системи автоматів.

2.2. Архітектура програмного забезпечення, модель вимог.

Для поставленої задачі була розроблена наступна діаграма бізнес-прецедентів [27]:

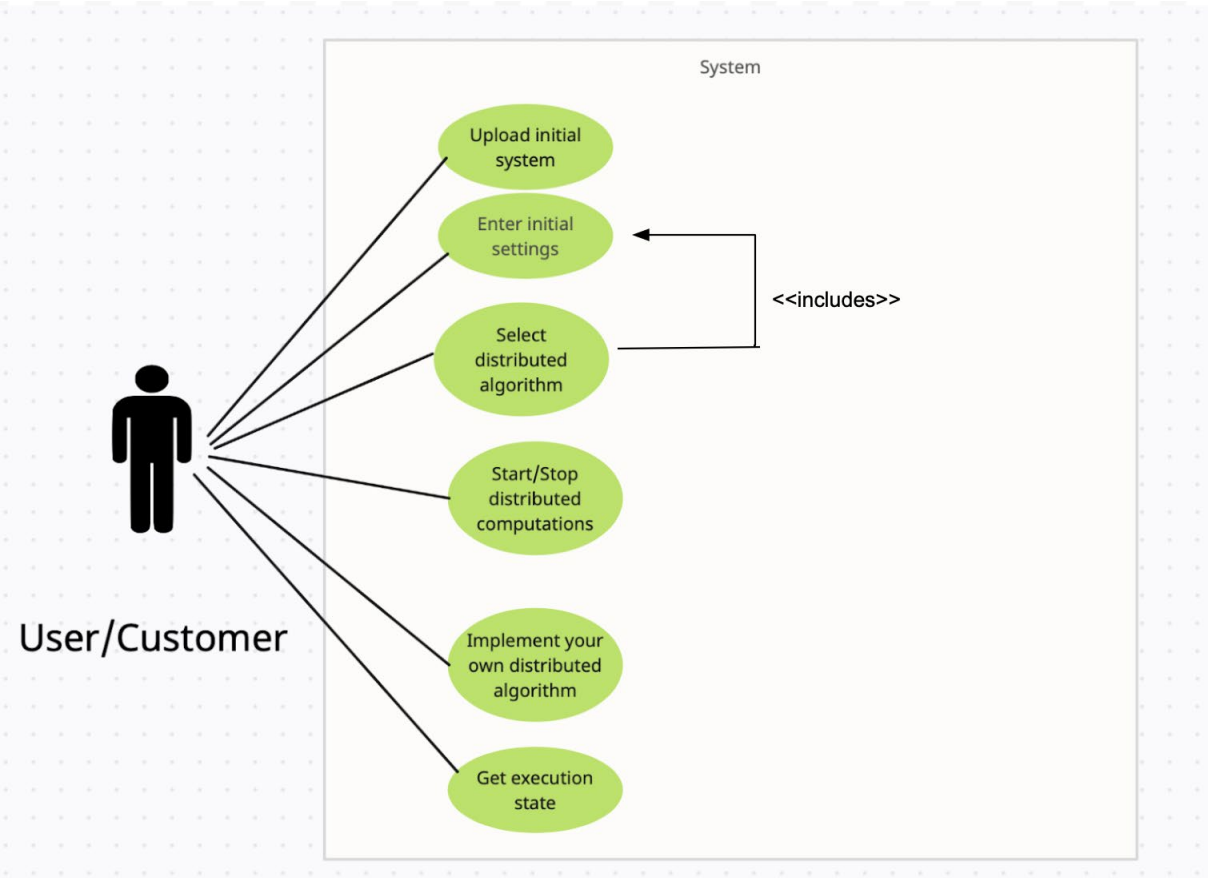


Рис 2.4. Діаграма бізнес-прецедентів

Опис бізнес-прецедентів:

→Upload initial system

1. Короткий опис:

Користувач підгружає граф, що відповідає розподіленій системі

2. Передумова:

Користувач запустив модуль.

3. Перебіг подій (табл. 2.1.):

Таблиця 2.1

Перебіг подій прецеденту Upload initial system

Дія користувача	Відклик системи

Користувач нажав на вкладку “file” і обрав “upload graph from file”	Система парсить файл, перевіряє валідність файлу та графу. У графічному вікні вимальовується меню та граф, що відповідає написаному у файлі. Система зберігає граф.

4. Альтернативный перебіг подій:

Якщо граф у файлі введений невірно, (не той формат файлу, є вершини, що ні з ким не поєднані тощо) відображаються повідомлення про помилку.

5. Післяумова

Відкрите вікно з зображеною розподіленою системою, у системі збережено та відображено поточну версію графу, відкрите меню для взаємодії.

→Enter initial settings

1. Короткий опис:

Введення початкових умов, налаштувань розподіленої системи

2. Передумова:

Користувач запустив модуль та виконано прецедент “Upload initial system”.

3. Перебіг подій (табл. 2.2):

Таблиця 2.2

Перебіг подій прецеденту Enter initial settings

Дія користувача	Відклик системи
Користувач обирає алгоритм, вірогідність доставки повідомлення, ініціатор алгоритму	Введені налаштування відображаються на графічному вікні. Система зберігає введені налаштування.

4. Альтернативный перебіг подій:

5. Післяумова:

Відкрите вікно з зображеною розподіленою системою, меню для взаємодії містить збережені системою введені налаштування.

→Start algorithm

1. Короткий опис:

Ініціація алгоритму з обраного процеса-ініціатора

2. Передумова:

Користувач запустив модуль, виконано прецедент “Enter initial settings”.

3. Перебіг подій (табл. 2.3):

Таблиця 2.3

Перебіг подій прецеденту Start algorithm

Дія користувача	Відклик системи
Користувач обрав початкові умови, натискає “Start”	Система стартує алгоритм, що користувач обрав у прецеденті “Enter initial settings”. У графічному вікні вимальовуються повідомлення, що виникають у процесах запущеного алгоритму та йдуть між процесами.

4. Альтернативный перебіг подій:

5. Післяумова:

Відкрите вікно з зображеною розподіленою системою, меню для взаємодії містить введені налаштування, у реальному часі вимальовуються повідомлення по каналах між процесами, алгоритм працює у бекграунді. Користувач має змогу бачити повідомлення, що генерується алгоритмом.

→Stop algorithm

1. Короткий опис:

Алгоритм ставиться на паузу

2. Передумова:

Користувач запустив модуль, виконано прецедент “Start algorithm”

3. Перебіг подій (табл. 2.4):

Таблиця 2.4

Перебіг подій прецеденту Stop algorithm

Дія користувача	Відклик системи
Користувач натискає “Stop”	Повідомлення, що йдуть між процесами у системі, припиняють вимальовуватися, алгоритм поставлений на паузу. Кнопка змінюється зі “Stop” на “Resume”

4. Альтернативный перебіг подій:

Якщо користувач не виконав прецедент “Start algorithm”, нічого не відбувається.

5. Післяумова:

Відкрите вікно з зображеною розподіленою системою, меню для взаємодії містить введені налаштування, алгоритм поставлений на паузу, повідомлення не вимальовуються, кнопка змінилась зі “Stop” на “Resume”.

→Resume algorithm

1. Короткий опис:

Алгоритм знімається з паузи

2. Передумова:

Користувач запустив модуль, виконано прецедент “Stop algorithm”

3. Перебіг подій (табл. 2.5):

Таблиця 2.5

Перебіг подій прецеденту Resume algorithm

Дія користувача	Відклик системи
Користувач натискає “Resume”	У графічному вікні повідомлення знову починаються вимальовуватися, алгоритм знявся з паузи.

4. Альтернативный перебіг подій:

5. Післяумова:

Відкрите вікно з зображеною розподіленою системою, меню для взаємодії містить введені налаштування, алгоритм знятий з паузи, повідомлення вимальовуються.

→Exit

1. Короткий опис:

Завершити програму

2. Передумова:

Користувач запустив модуль.

3. Перебіг подій (табл. 2.6.):

Таблиця 2.6

Перебіг подій прецеденту Exit

Дія користувача	Відклик системи
Користувач натискає на крестик у лівому верхньому куті екрану.	Програма завершується.

4. Альтернативный перебіг подій:

5. Післяумова:

Програма закінчена, користувач знаходиться у вікні свого девайсу.

→Implement your own distributed algorithm

1. Короткий опис:

Користувач може створити власний розподілений алгоритм і протестувати його на нашій системі.

2. Передумова:

Користувач витягнув наш код, має досвід програмування і середовище програмування.

3. Перебіг подій (табл. 2.7):

Таблиця 2.7

Перебіг подій прецеденту Implement your own distributed algorithm

Дія користувача	Відклик системи
Користувач скачує програмний код і реалізує інтерфейс класу “Process”. Потім він запускає модуль і тестує реалізований алгоритм.	Система містить розподілений алгоритм, який користувач реалізував самостійно.

4. Альтернативный перебіг подій:

5. Післяумова:

Модуль збудований над алгоритмом, що користувач реалізував самостійно.

→Get execution state

Короткий опис:

1. Користувач може отримати журнал виконання алгоритму у розподіленій системі.

2. Передумова:

Виконаний прецедент “Start algorithm”

3. Перебіг подій (табл. 2.8):

Таблиця 2.8

Перебіг подій прецеденту Get execution state

Дія користувача	Відклик системи
Користувач натискає на кнопку “Get execution state”	Система дістає журнал обміну повідомленнями між елементами розподіленої системи і відображає його на екрані. У цей час алгоритм ставиться на паузу.

4. Альтернативный перебіг подій:

Якщо алгоритм і був на паузі, на паузі він і залишається.

5. Післяумова:

Запущений алгоритм на паузі, відкрите нове графічне вікно з журналом обміну повідомленнями.

Користуючись цією моделлю прецедентів, були розроблені наступні sequence діаграми [27], що відповідають ланцюгам викликів класів та методів у системі:

1. Сіквенс-діаграма, що відповідає прецеденту “Get execution state” (рис. 2.5)

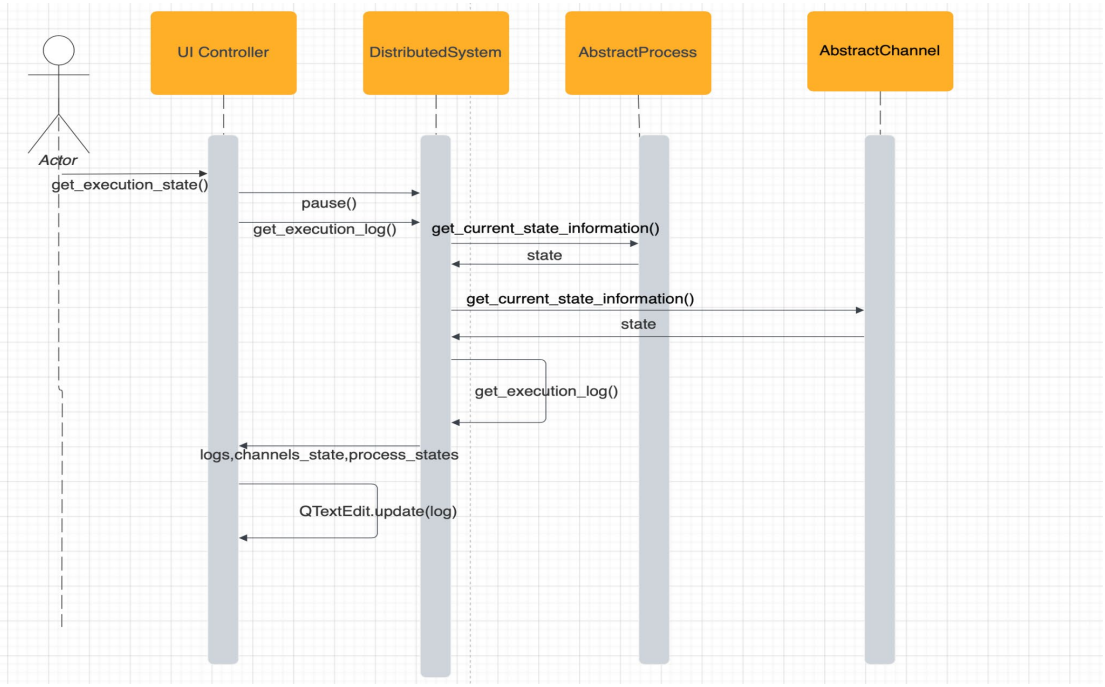


Рис 2.5. Сіквенс-діаграма “Get execution state”

2. Сіквенс-діаграма, що відповідає прецеденту “Enter initial settings” (рис. 2.6)

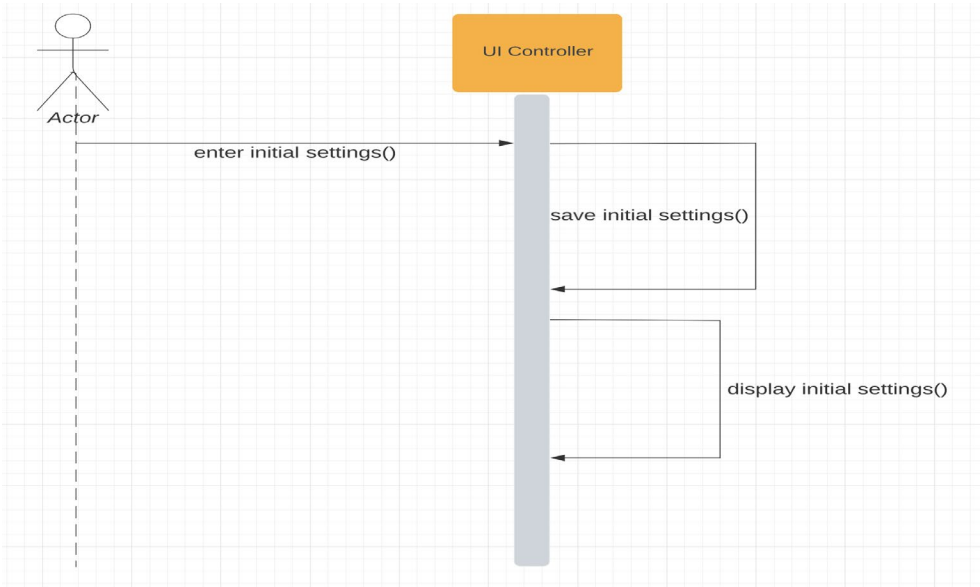


Рис 2.6. Сіквенс-діаграма “Enter initial settings”

3. Сіквенс-діаграма, що відповідає прецеденту “Start algorithm”. Вона велика, тому була розбита на декілька рисунків, зліва направо, згори донизу (рис. 2.7 – 2.10).

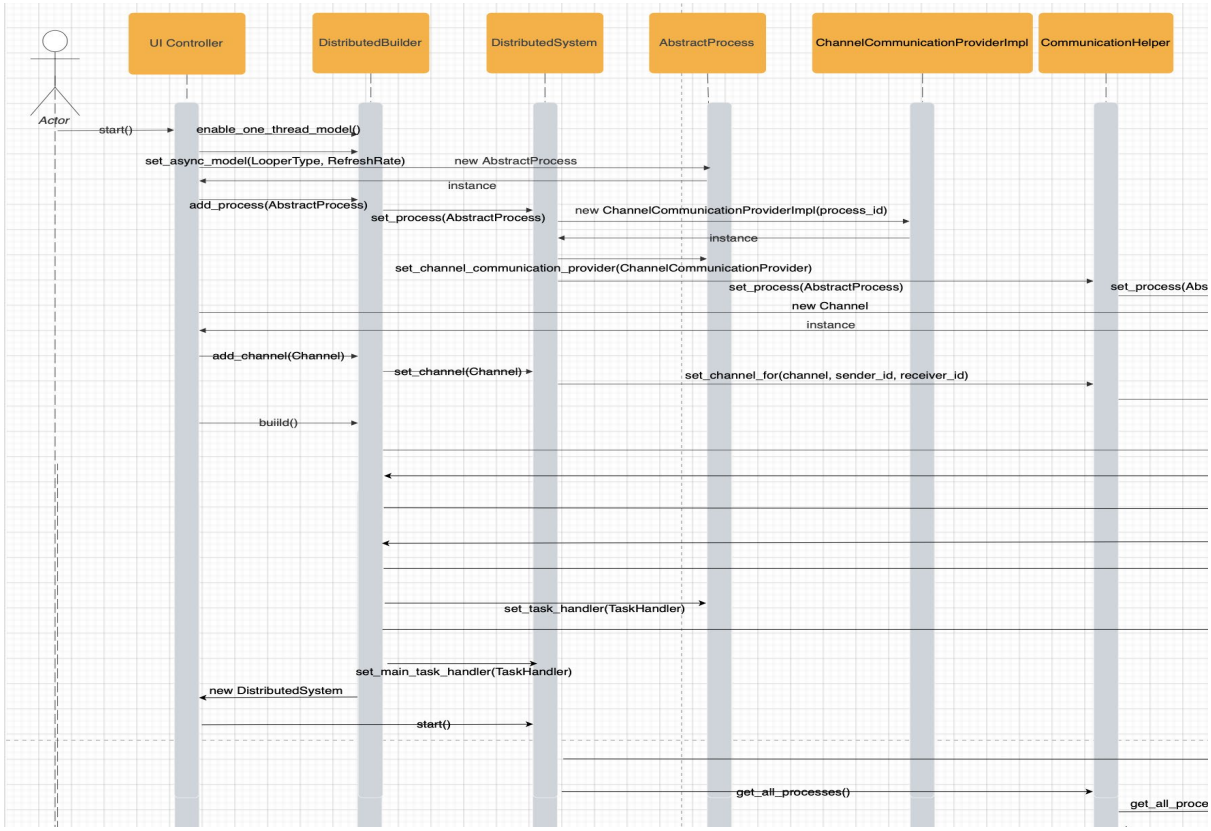


Рис 2.7. Сіквенс-діаграма “Start algorithm”. Лівий верхній кут.

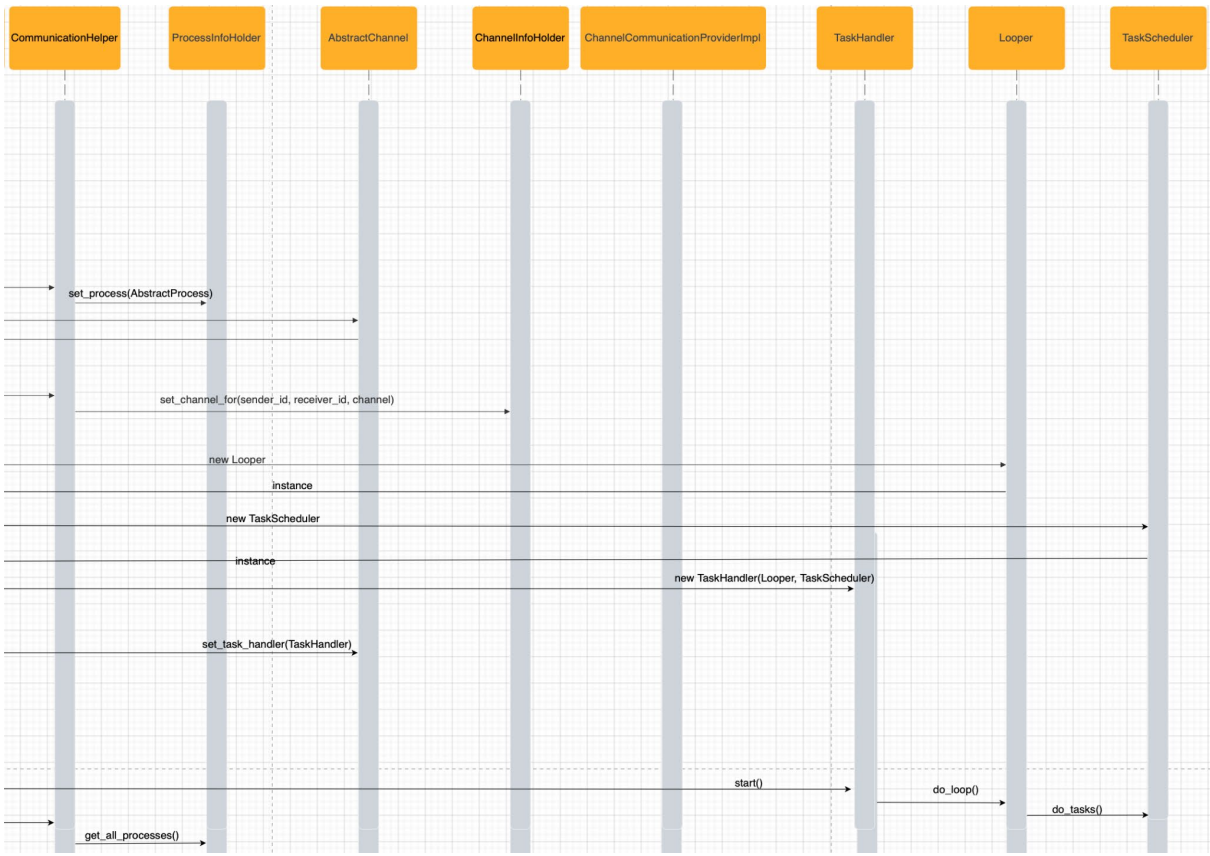


Рис 2.8. Сіквенс-діаграма “Start algorithm”. Правий верхній кут.

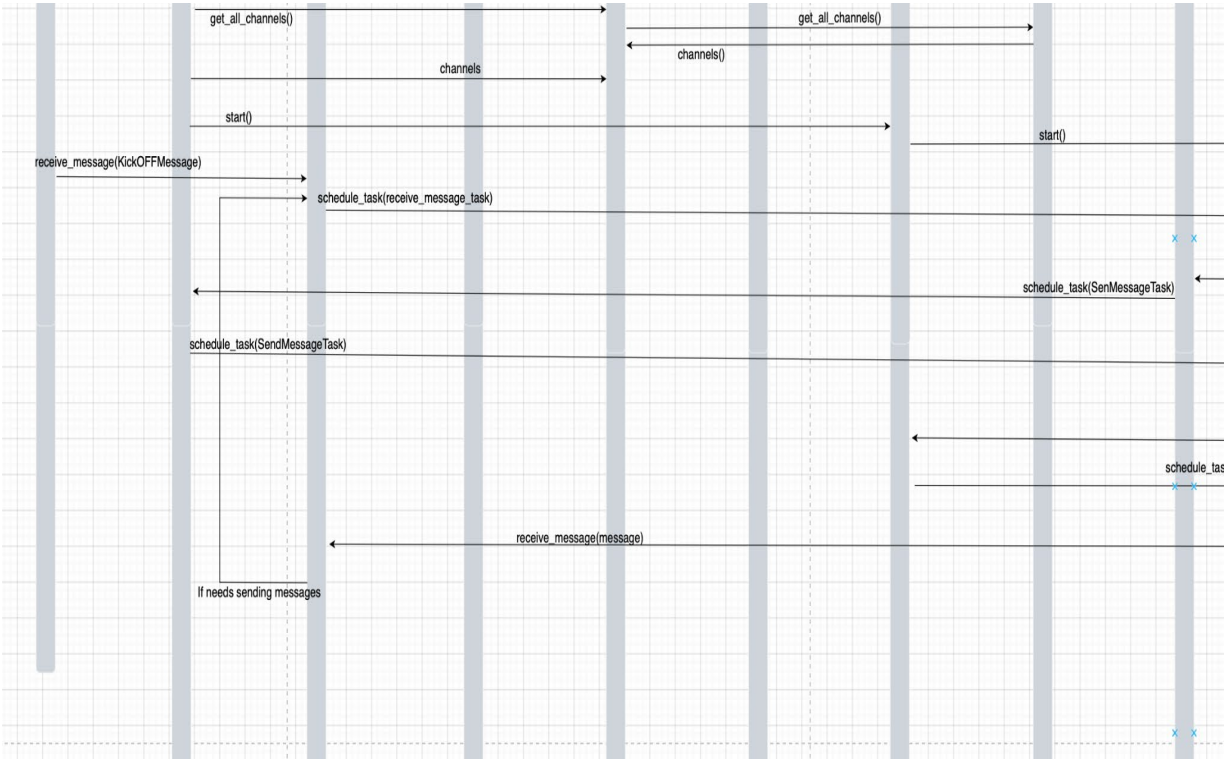


Рис 2.9. Сіквенс-діаграма “Start algorithm”. Лівий нижній кут.

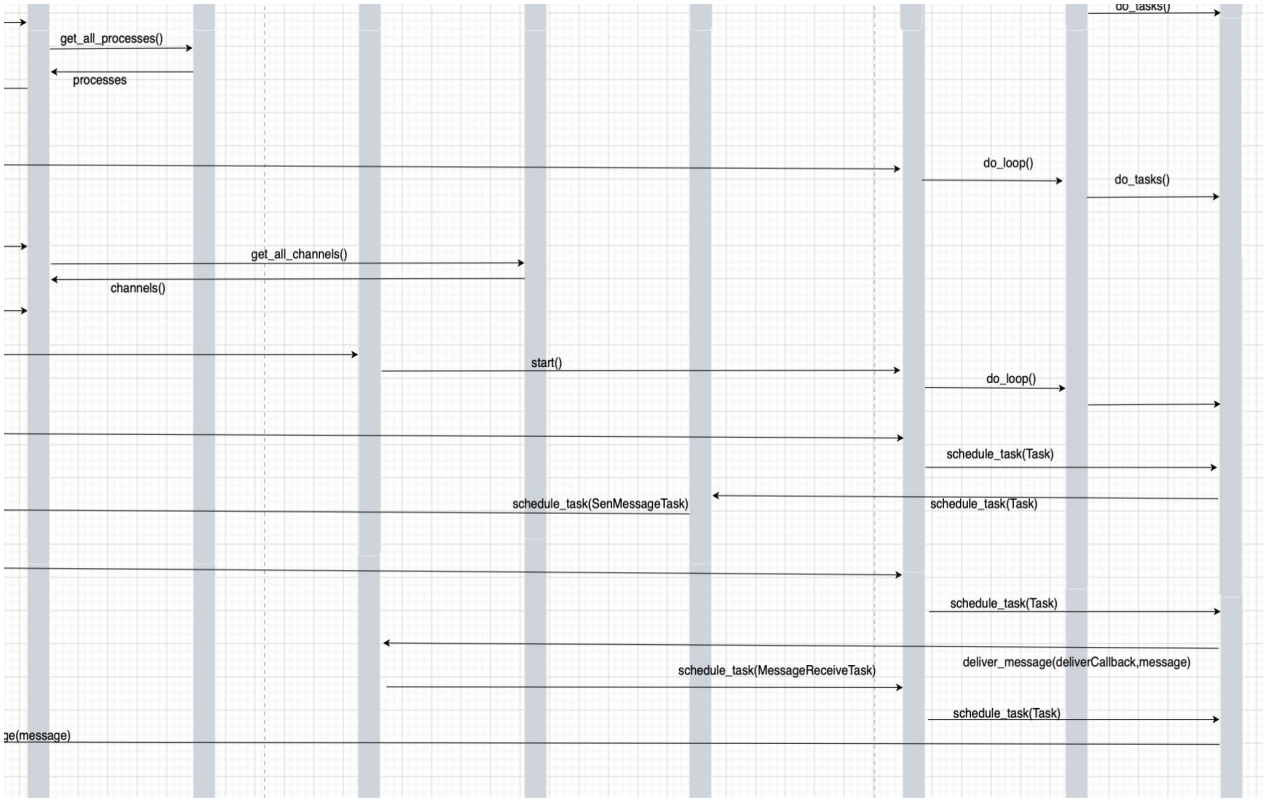


Рис 2.10. Сіквенс-діаграма “Start algorithm”. Правий нижній кут.

4. Сіквенс-діаграма, що відповідає прецедентам “Pause/resume algorithm”. Вона велика, тому була розбита на два рисунки, зліва направо

(рис. 2.11-2.12.).

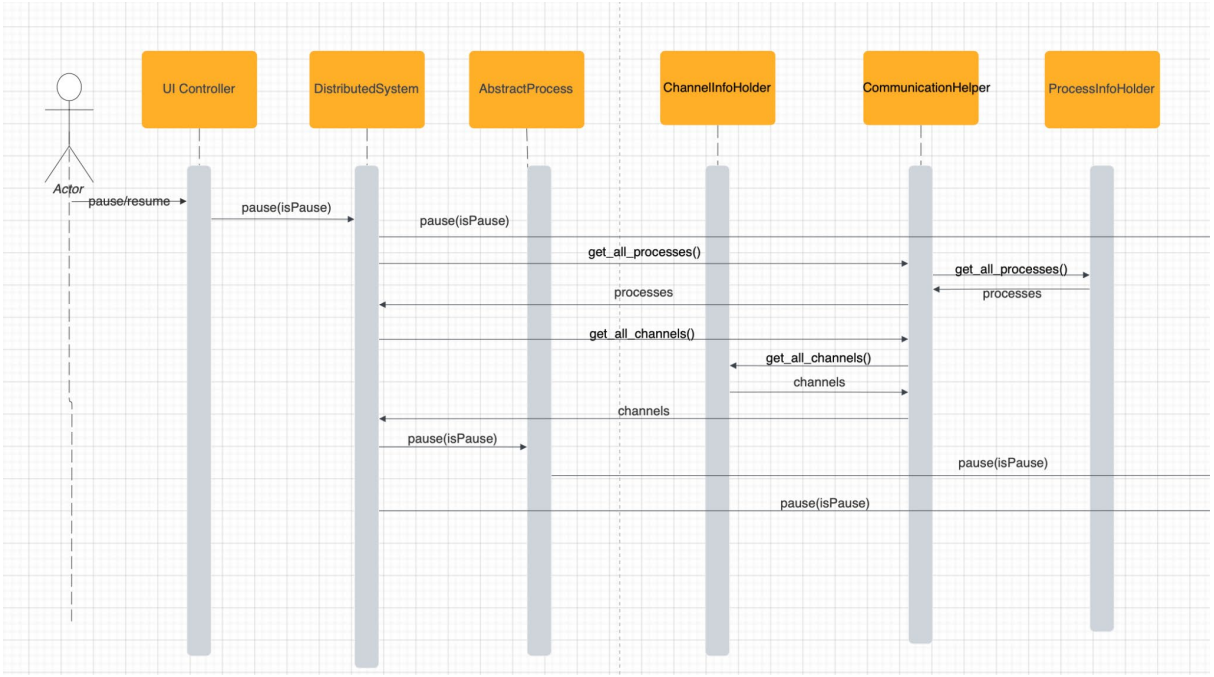


Рис 2.11. Сіквенс-діаграма “Stop/resume algorithm”. Лівий кут.

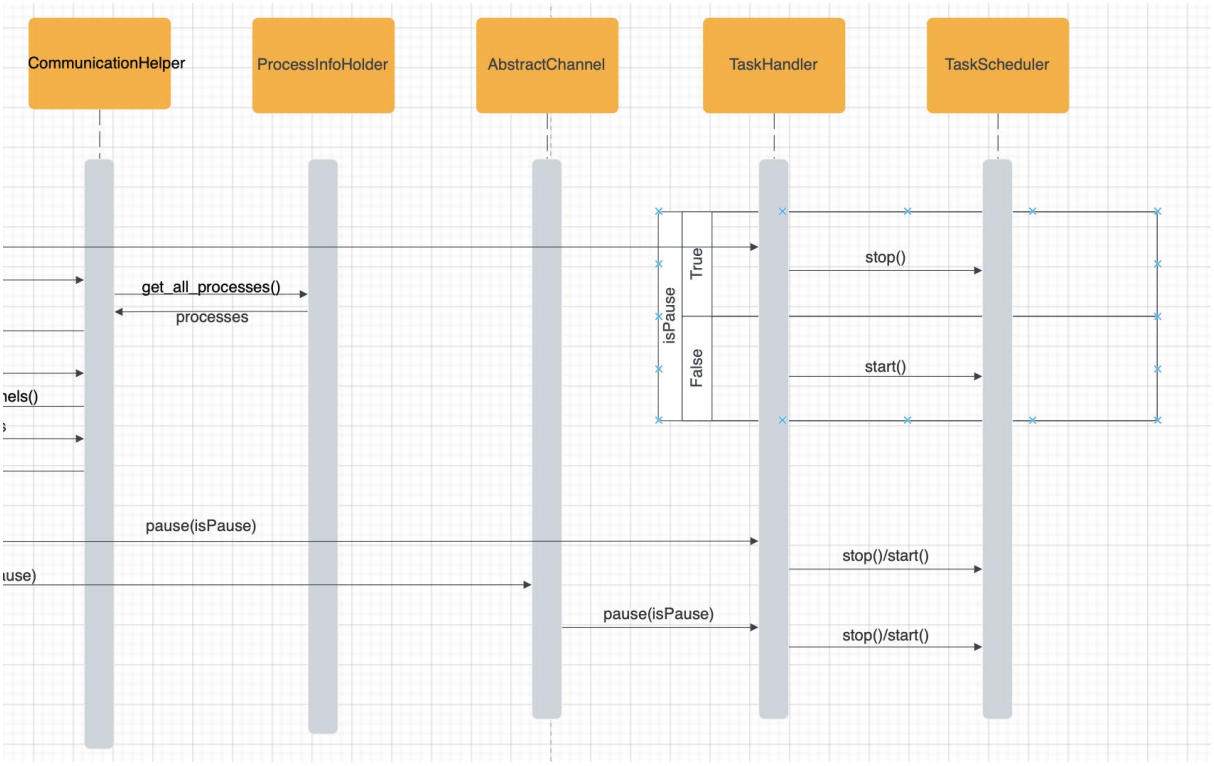


Рис 2.12. Сіквенс-діаграма “Stop/resume algorithm”. Правий кут.

Виходячи з побудованих сіквенс-діаграм, була побудована діаграма класів [28], що відповідає архітектурі системи (рис. 2.13. – 2.14.):

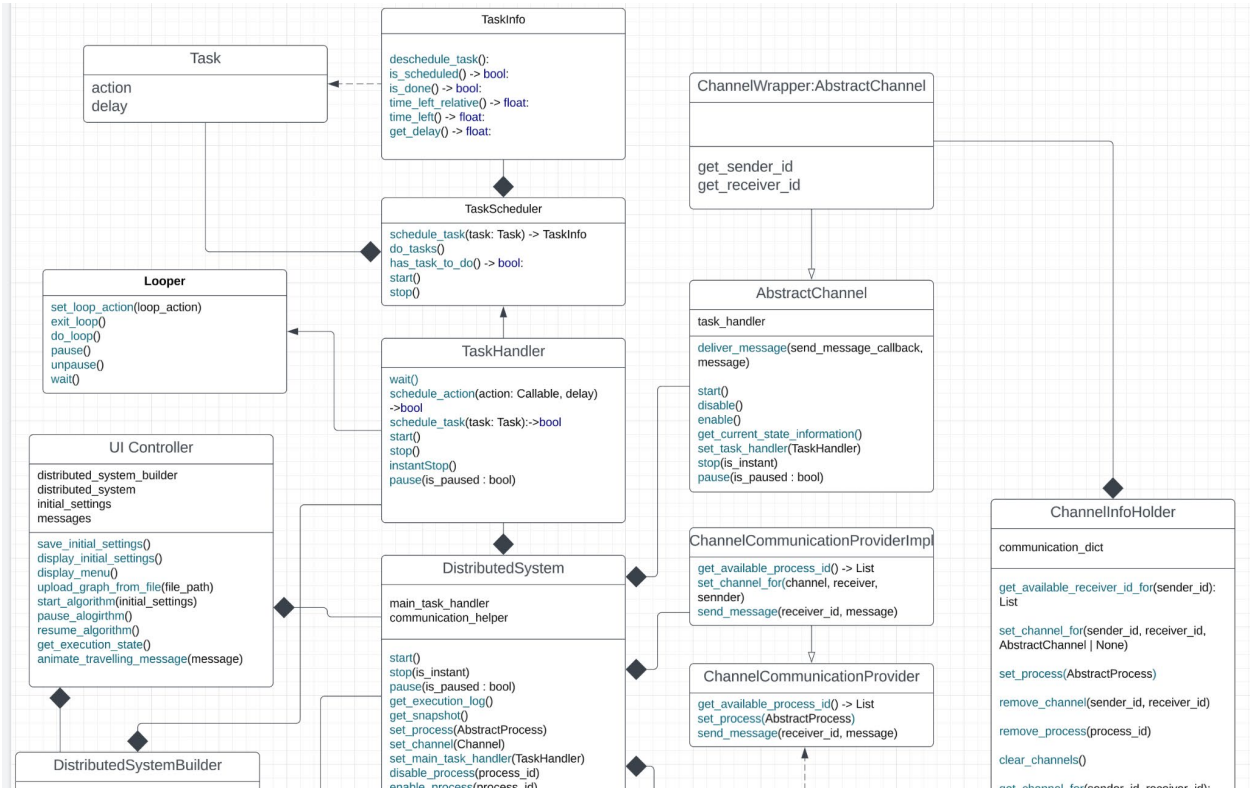


Рис 2.13. Діаграма класів. Верх.

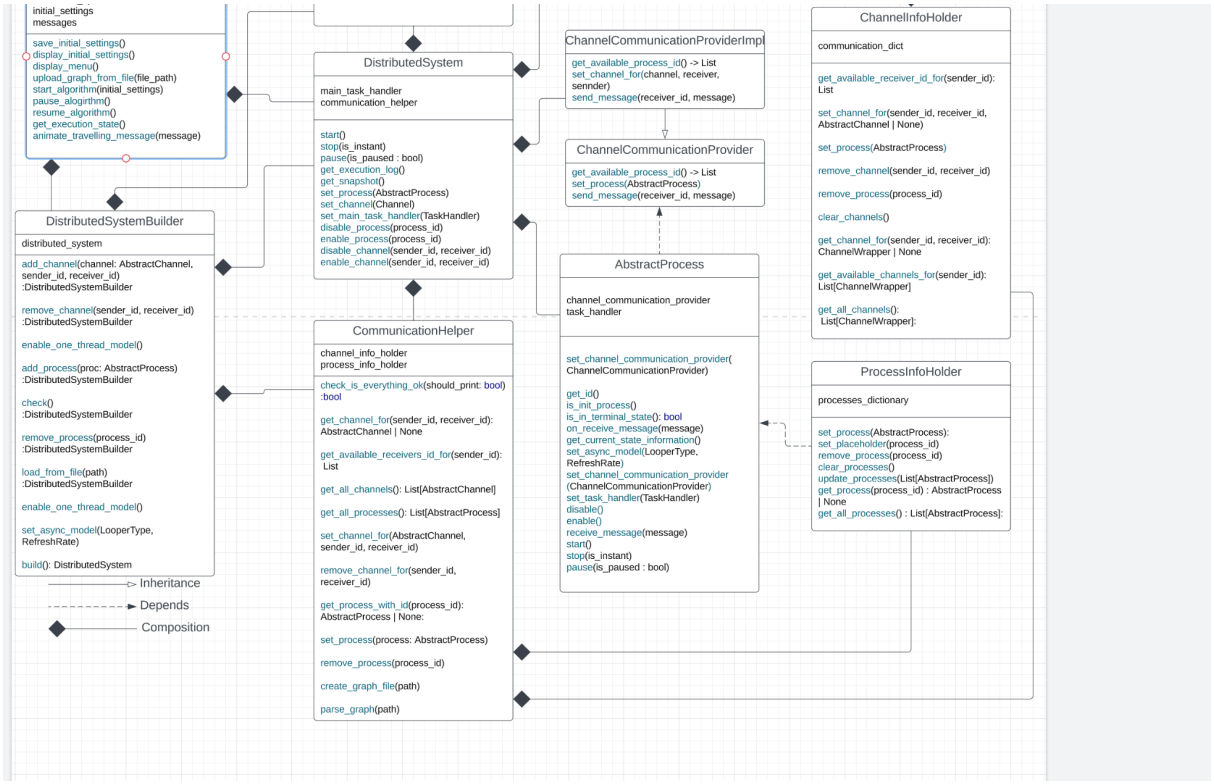


Рис 2.14. Діаграма класів. Низ.

Класом, що напряму взаємодіє з користувачем є UI Controller. Він зберігає initial settings, може отримувати команди та викликати DistributedSystemBuilder, DistributedSystem – головний клас програми, який

створений для симуляції роботи розподіленої системи, взаємодії між всіма її частинами.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ТА ДОСЛІДЖЕННЯ ПОБУДОВАНОГО ПРОТОТИПУ

3.1 Реалізація математичної моделі розподіленої системи для динамічного аналізу

Модуль, що реалізує уніфікованою математичну модель, був розроблений мовою Python версії 3.3, бо вона містить модулі для графічного зображення оброблених даних, модулі для процесів та потоків, інструменти для реалізації GUI [29].

При розробці були використані наступні пакети, які при бажанні можна скачати командою pip install module_name:

PyQT5 [30] – бібліотека, що надає можливість створювати графічний інтерфейс, забезпечує взаємодію користувача і інтерфейсу.

Також були використані власні модулі пітона:

- time [31] – дає можливість дивитися поточний час та проводити операції з часом;
- random [32] – модуль для генерації випадкових чисел;
- typing [33] – модуль для підтримки типізації;
- multiprocessing [6] – для роботи з процесами.

Головна проблема, з якою ми зіштовхнулися – Python GIL [34].

Спочатку ми намагалися на кожний процес у розподіленій системі створювати власний потік, який комунікував з іншими. Це б була справжня розподіленість. Проте, моделювання системи за допомогою потоків – не найкраща стратегія.

По перше, потоки коштують дуже багато ресурсів. Якщо б у нас була розподілена система зі 100 машин, потоки б з’їли усі ресурси операційної системи і вона впала би.

До того ж, потоки не є передбачуваними з точки зору модуляції – їх робота залежить від локів, ресурсів, операційної системи та іншого. Тож не можна вважати їх гарним інструментом для моделювання чогось.

По друге, ми зіштовхнулися з проблемою Python GIL.

Глобальне блокування інтерпретатора (GIL) – це механізм, який використовується CPython, еталонною реалізацією Python, для забезпечення безпеки потоків. GIL дозволяє лише одному потоку виконувати байт-код Python за раз, навіть на багатоядерному ЦП. Це означає, що навіть якщо у вашій програмі Python запущено кілька потоків, лише один потік може виконувати код Python у будь-який момент часу.

GIL необхідний для забезпечення безпеки потоків у Python. Без GIL кілька потоків могли б спробувати отримати доступ до того самого об’єкта Python одночасно, що призвело б до конкуренції та інших проблем. Однак GIL може мати негативний вплив на продуктивність багатопоточних програм Python.

У додатках графічного інтерфейсу GIL може спричинити проблеми, оскільки головний потік програми зазвичай відповідає за обробку подій введення користувача та оновлення графічного інтерфейсу користувача. Якщо основний потік заблоковано тривалим завданням, наприклад вводом-виводом або обчисленням, інтерфейс користувача може перестати реагувати. Це пояснюється тим, що основний потік не може відповідати на події, введені користувачем, поки він заблокований довгостроковим завданням.

Щоб уникнути цієї проблеми, додатки графічного інтерфейсу повинні використовувати методи паралельного виконання, які не блокують основний потік. Одним з підходів є використання робочих потоків або процесів для виконання трудомістких завдань, таких як введення-виведення або обчислення. Це дозволяє основному потоку реагувати на події, введені користувачем, у той час як трудомістке завдання виконується у фоновому режимі.

Інший підхід полягає у використанні методів асинхронного програмування, таких як співпрограми або цикли подій. Ці методи дозволяють виконувати кілька завдань одночасно в одному потоці, не вимагаючи додаткових потоків або процесів. Однак асинхронне програмування може бути складнішим, ніж використання окремих потоків або процесів, і потребує детального проектування, щоб уникнути конкуренції та інших проблем паралельного доступу.

Таким чином, GIL – це механізм, який використовується CPython для забезпечення безпеки потоків, але він може негативно вплинути на продуктивність багатопоточних програм Python, особливо в додатках GUI. Щоб уникнути цієї проблеми, додатки з графічним інтерфейсом користувача повинні використовувати методи паралелізму, які не блокують основний потік, наприклад робочі потоки, процеси або методи асинхронного програмування.

Іншими словами, Python не реалізує реальну багатопоточність, і наш основний потік з графічним інтерфейсом завжди блокувався потоками, що відповідали за процеси в розподіленій системі.

Щоб вирішити цю проблему, ми використали QThread [35], який з періодичністю перевіряє, які події надійшли на відмальовку. Іншим підходом міг би бути multiprocessing за допомогою ProcessLooper, який створює реальні різні асинхронні процеси, які не крадуть ресурси, як у многопоточному середовищі.

3.2 Дослідження побудованого прототипу

Робота розподіленої системи була перевірена на алгоритмі Лаї-Янга [36].

Цей алгоритм є централізованим алгоритмом збору станів системи, який працює на звичайних каналах, не вимагає ФІФО типу каналу. Мета полягає у тому, щоб кожен процес зробив знімок стану, які потім можна

використовувати для дебагінгу та рестарту системи після якоїсь критичної помилки. Складність полягає у тому, щоб випадково не втратити повідомлення, які ще були у каналах. Інакше б була ситуація, коли процес надіслав повідомлення, зробив знімок стану, інший процес зробив знімок стану до отримання повідомлення і таким чином повідомлення у каналі було назавжди втрачене.

Алгоритм працює наступним чином. Поки процес не зробив локальний знімок стану, від додає до всіх повідомлень, що надсилає, тег false; після зробленого знімку стану додає тег true. Якщо процес отримує повідомлення з тегом true, але він ще не зробив знімок локального стану, він робить знімок свого стану перед обробленням цього повідомлення. Процес знає про стан каналу, який просто є усіма повідомлення з тегом false, що він отримав після того, як зробив локальний знімок стану.

Є дві проблеми. По-перше, якщо після локального знімка ініціатор не буде надсилати жодних основних повідомлень, інші процеси можуть ніколи не зробити локальний знімок. По-друге, як процес дізнається, що він може припинити очікування основних повідомлень з тегом false і обчислити стан вхідного каналу? Обидва питання вирішуються спеціальним керуючим повідомленням, яке процес p надсилає в кожен вихідний канал pq після того, як зробив свій локальний знімок. Це керуюче повідомлення інформує q, скільки базових повідомлень з тегом false p було надіслано в канал pq. Якщо q ще не зробив локальний знімок, він робить його після отримання цього керуючого повідомлення.

Отже, процес p завершується після того, як він отримав від кожного каналу:

1) контрольне повідомлення <control_message, k> з k повідомленнями, що були надіслані у канал з тегом false ;

2) k-1 повідомлення з тегом false, що були йому відправлені до того, як процес зробив знімок локального стану.

Таким чином, процес робить знімок локального стану та ми не втрачаємо жодне повідомлення з каналів, бо усі повідомлення з тегом false процеси отримали перед завершенням, а повідомлення з тегом true процеси надішлють заново у випадку перезагрузки, бо повідомлення з тегом true виникають і відправляються вже після снепшоту.

Алгоритм був реалізований мовою Python і протестований у нашій моделі.

Вікно з графом розподіленої системи (рис. 3.1):

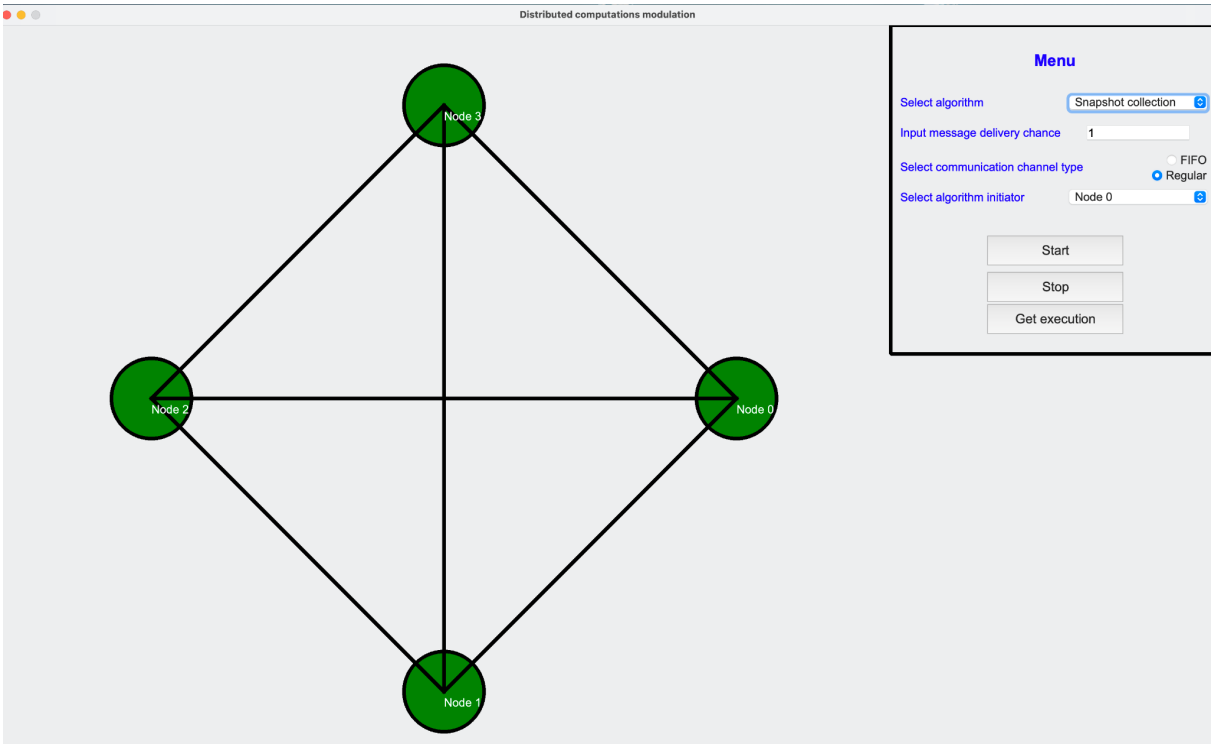


Рис 3.1. Розподілена система повного графа на чотирьох вершинах та меню.

Спочатку процеси обмінюються звичайними повідомлення з тегом false. Кількість таких повідомлень, їх час, кому вони відправляються – випадкові. Це емулює реальну розподілену систему, яка живе своїм життям та робить свої основні обчислення, що не прив’язані до алгоритму збору станів, який є допоміжним. Також, це дає змогу протестувати на різних сценаріях і додає елемент випадковості, який так чи інакше є у кожній розподіленій системі. Варто зазначити, що час доставки повідомлення є також випадковим, як у

житті, та варується від 5ти до 20ти секунд. На рисунку ми бачимо, що повідомлення у каналах є як раз відправленими до збору станів і містять тег false. Через 5 секунд обміну звичайними повідомленнями ініціюється алгоритм збору станів у процесі 2, бо його я обрав ініціатором у меню. Виникаючі повідомлення у процесі 2 і є якраз повідомленнями типу ControlMessage, які містять кількість відправлених повідомлень з тегом false і відправляються усім сусідам ініціатора (рис. 3.2).

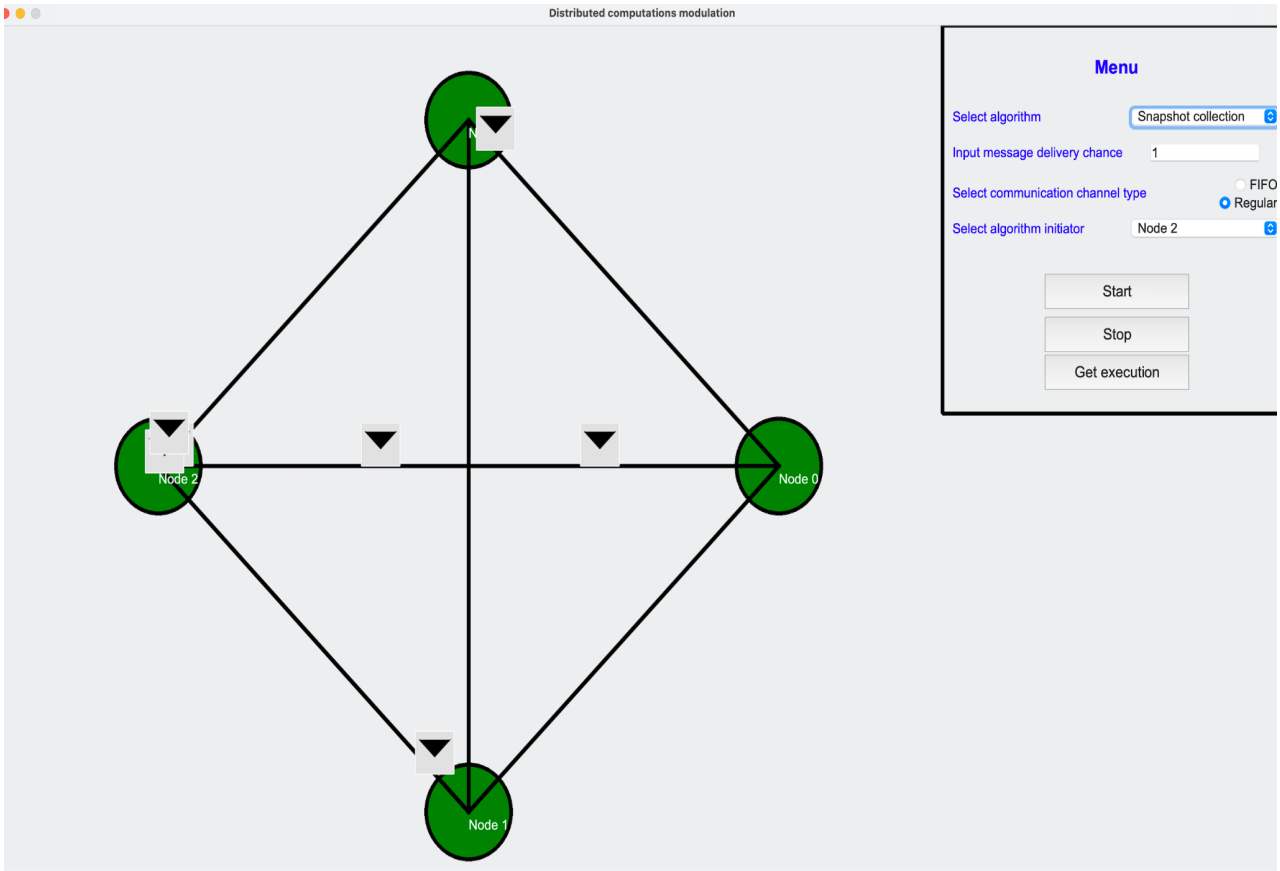


Рис 3.2. Повідомлення запущенного алгоритму Лаї-Янга.

Поки такі повідомлення від процесу 2 надійдуть до сусідів, вони встигнуть відправити ще якусь кількість повідомлень з тегом false.

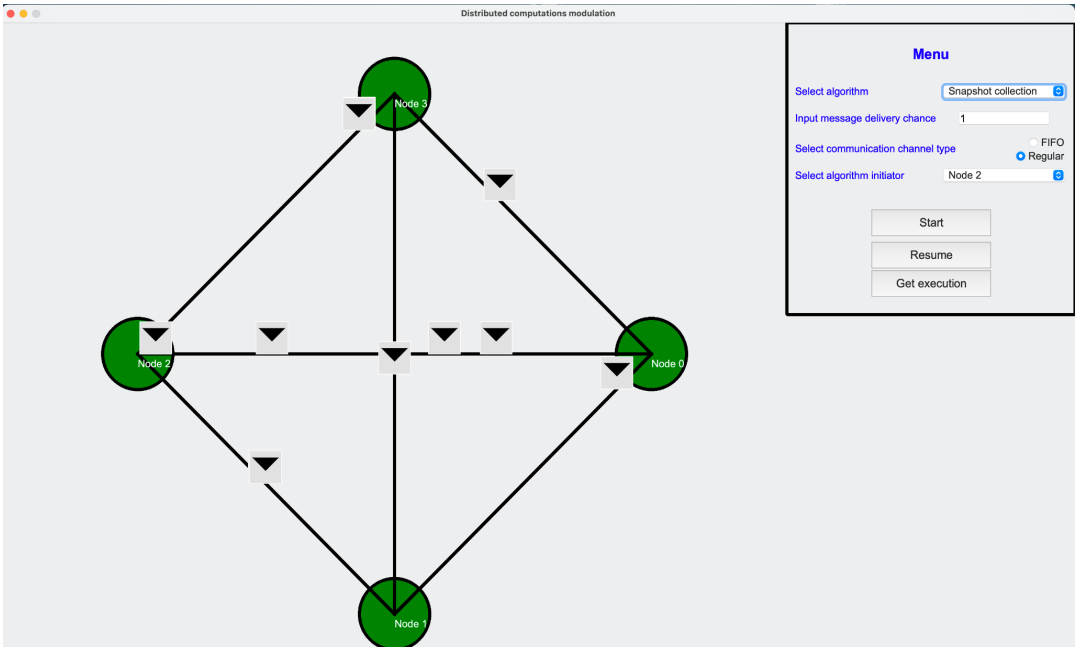


Рис 3.2. Збільшення кількості повідомлень запущенного алгоритму Лаї-Янга.

Тут ми бачимо велику кількість повідомлень через те, що кожен процес надсилав звичайні повідомлення, у яких час доставки повідомлення може бути великим і до цього додались по 3 повідомлення типу ControlMessage, які процеси надсилають своїм сусідам (рис. 3.3):

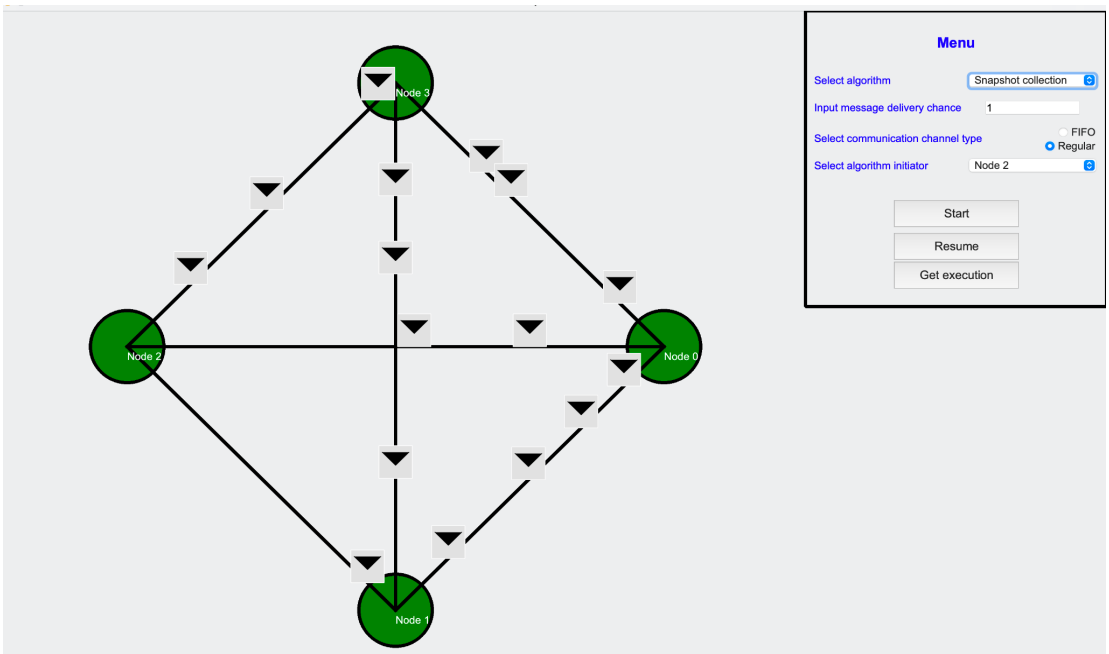


Рис 3.3. Контрольні повідомлення збору станів надійшли до нод 1,3,0 та вони відправили аналогічні повідомлення сусідам.

Далі кількість повідомлень поступово зменшується і алгоритм завершується (рис. 3.4):

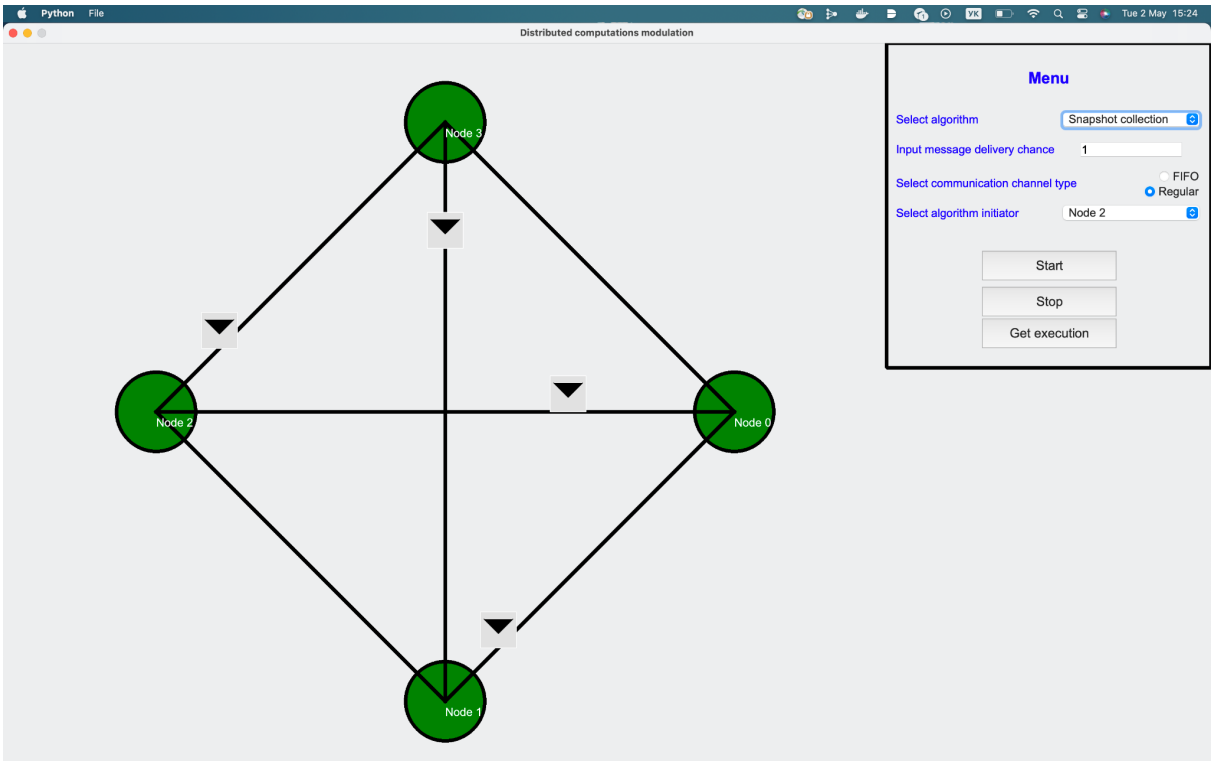


Рис 3.4. Завершення алгоритму.

Після завершення алгоритмів і відсутності повідомлень, я отримав журнал подій у системі [37] (рис. 3.5 – 3.7):

Рис 3.5. Журнал повідомлень “Get execution state”

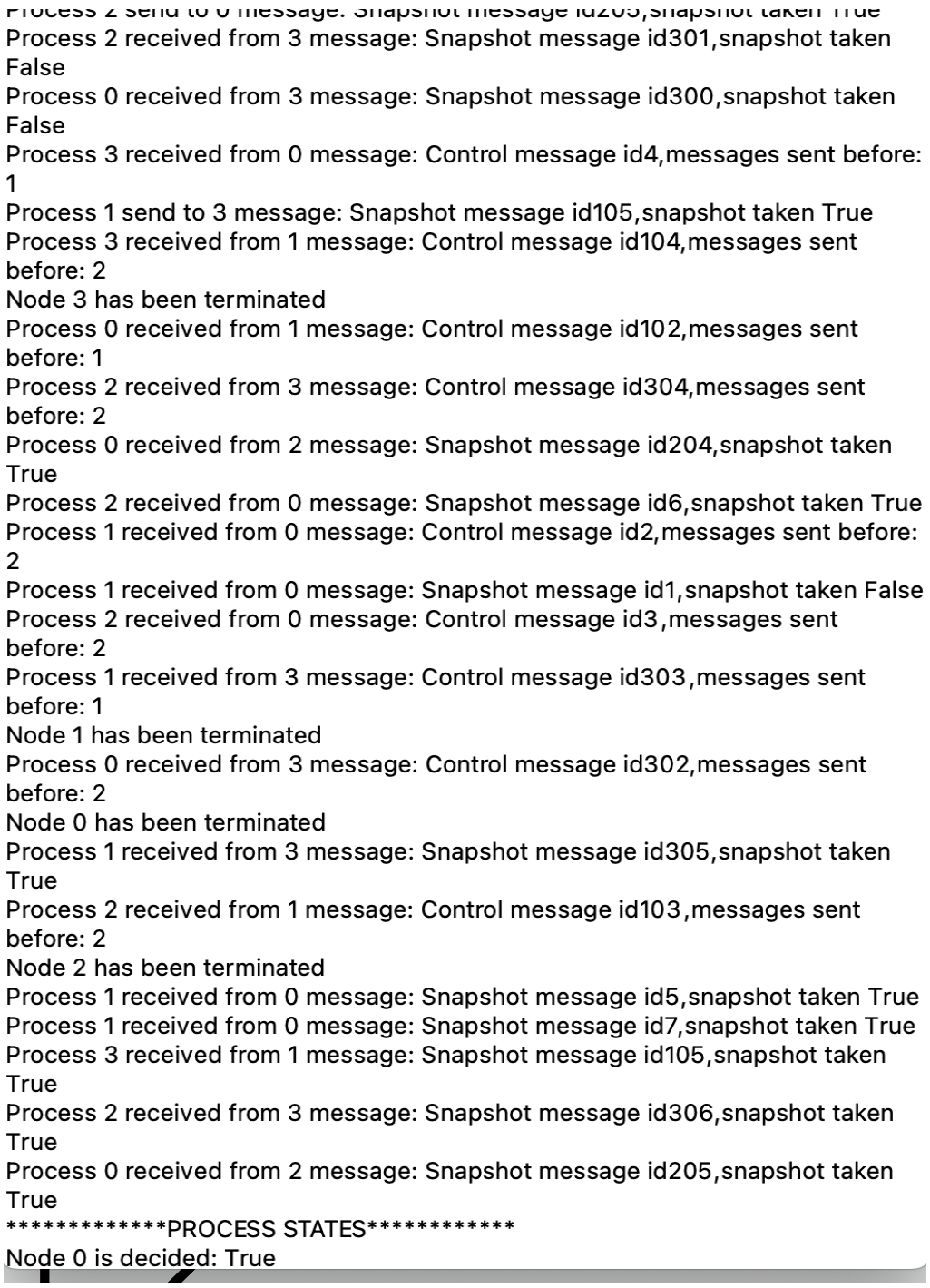


Рис 3.6. Журнал повідомлень “Get execution state”. Продовження.

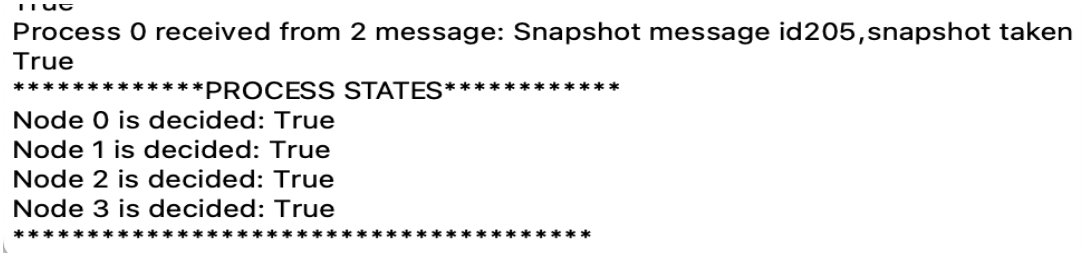


Рис 3.7. Журнал повідомлень “Get execution state”. Поточний стан елементів системи.

Як можна побачити, алгоритм відпрацював і усі процеси завершилися.

Цікавими є останні останні записи у журналі (рис. 3.8):

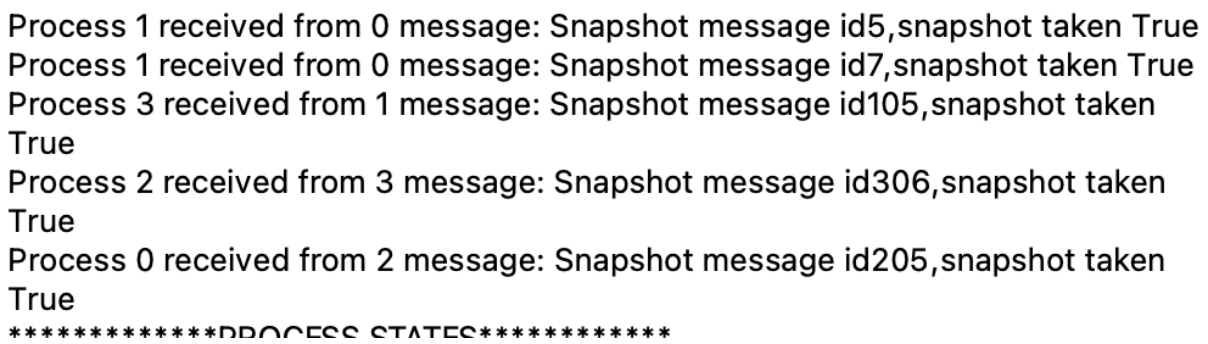


Рис 3.8. Журнал повідомлень “Get execution state”.

Процеси отримують повідомлення з тегом true, але вони вже не важливі.

Усі ці повідомлення містять тег True. Процеси перейшли у terminated state навіть не очікуючі на ці повідомлення, бо вони їм і не потрібні, так як важливі лише повідомлення, що були отримані до того, як інші процеси зробили знімок стану, так як повідомлення з тегом true вони все одно надішлють після рестарту.

Цікавою є також ситуація, відображена на рисунку 3.9. Процеси мають завершитися, коли вони отримали контрольне повідомлення з усіх каналів і усі повідомлення з тегом false з усіх каналів. На рисунку можна бачити, що процеси 2, 0, 3 вже отримали усіх false повідомлення, і тому завершилися як тільки отримали не вистачаючі Control message. А ось node 1 навпаки, спочатку отримала усі Control message, проте вона не мала усі повідомлення з тегом false, і вона була вимушена їх чекати. Як тільки node 1 отримала від 3 останнє повідомлення з тегом false, порівняла кількість отриманих повідомлень від 3 з тегом false і кількість відправлених повідомлень з 3 у отриманому Control message, одразу завершилась.

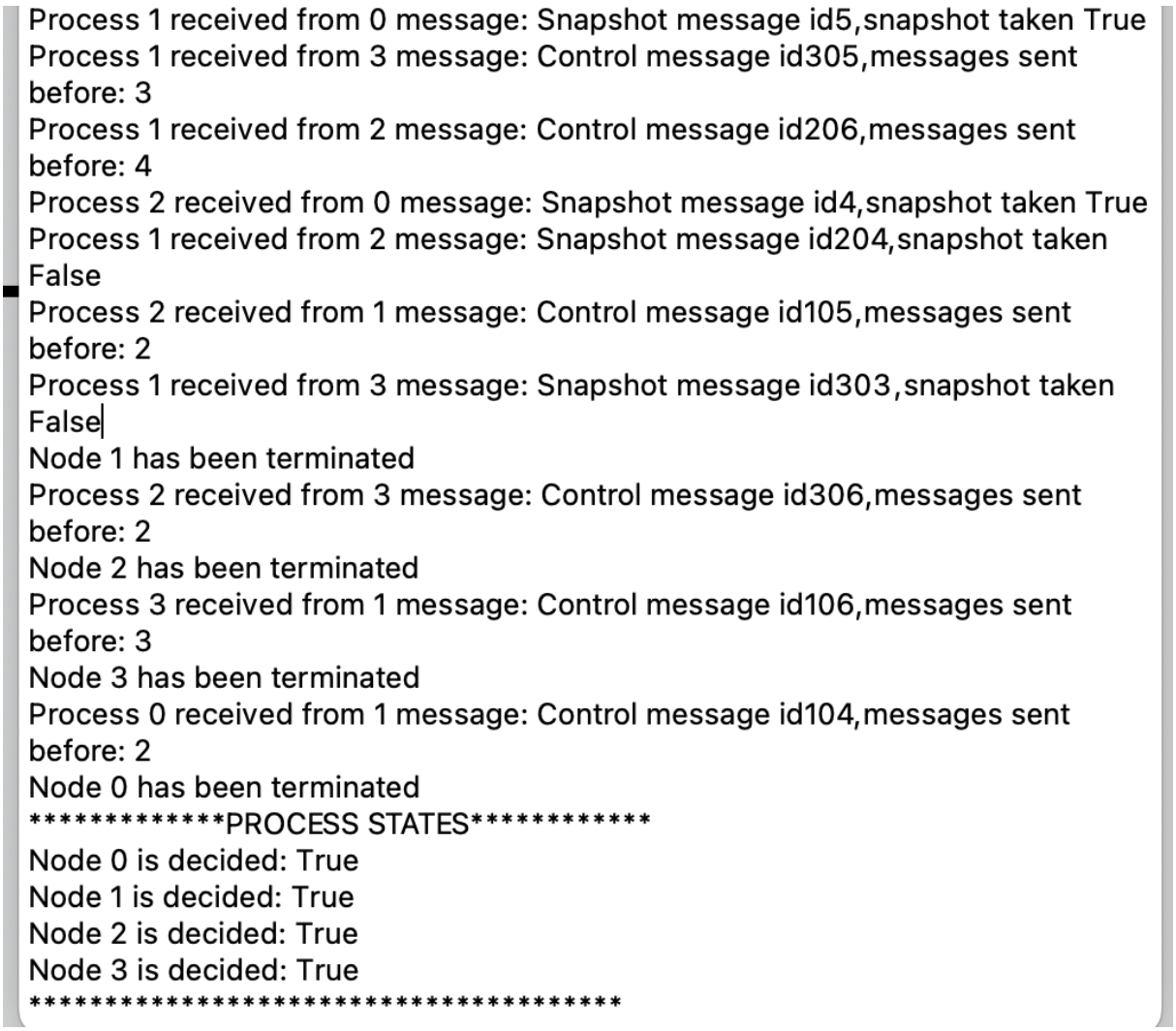


Рис 3.9. Журнал повідомлень “Get execution state”. Node 1 переходить у стан “terminated” після отримання невистачаючого повідомлення з тегом false.

Алгоритм був апробований на різних графах розподіленої системи. Було підраховані кількість отриманих повідомлень з тегом false, порівняно з кількістю відправлених і тим, що записано у Control Message. Результати сходяться і алгоритм працює коректно. Елементи випадковості, такі як кількість повідомлень від процесів, кому вони відправляються, час, коли вони відправляються, час доставки повідомлення, дають змогу протестувати алгоритм на багатьох унікальних випадках, навіть на одному й тому самому графі.

ВИСНОВКИ

Розподілені обчислення виникають повсемісно у нашому житті – інтернет, великі обчислювальні центри і кластери, комп'ютери з багатьма розподіленими процесами. З розвитком розподілених обчислень виникає багато задач, які вимагають збору даних з системи, важливим є вміння їх розв'язувати.

1. Проаналізовано існуючі першоджерела, під час дослідів були розглянуті моделі розподілених обчислень Н. Лінч та Ж. Теля. Був запропонований метод моделювання розподілених обчислень за допомогою автоматів Н. Лінч.

2. Були проаналізовані наявні методи математичного моделювання розподілених систем та обрані шляхи оптимізації наявних математичних моделей для практичного застосування.

3. Була побудована модель вимог для програмного модуля, що був створений для математичного моделювання розподілених систем. Виходячи з моделі вимог, була побудована архітектура програми.

4. З побудованої моделі вимог було вирішено реалізувати їх мовою Python за допомогою сучасних методів розробки програмного забезпечення, зокрема PyQT5, бібліотек пітона для multiprocessing, multithreading, рандомізації різних параметрів розподіленої системи.

5. Наведена математична модель була реалізованою мовою Python версії 3.3, був використаний сторонній модуль PyQT5 для графічної частини програми.

6. Тестування показало, що модуль добре емулює розподілену систему, відомі алгоритми, такі як алгоритм Еха, алгоритм сбору станів Лаї-Янга, працюють справно. Алгоритм Лаї-Янга залежить від багатьох випадкових параметрів, що дає різноманітне тестування на одній і тій самій розподіленій системі. Система розроблена так, що можна легко написати свій алгоритм і протестувати його, змінюючи різні параметри.

7. Побудований модуль може бути застосований у навчальних та науково-дослідницьких цілях. Проте, математична модель розподіленої системи була реалізована лише для динамічного аналізу алгоритмів. У майбутньому хотілося б реалізувати побудовану математичну модель і для статичного аналізу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Cohen-Almagor, R. Internet History. *International Journal of Technoethics*, 2011.Vol. 2(2):45-64. DOI:10.4018/jte.2011040104

2. Tel, G. Introduction to Distributed Algorithms. 2nd edition. Cambridge University Press. 2012. https://doi.org/10.1017/CBO9781139168724

3. Lynch, N. A. Distributed Algorithms. Publisher: Morgan Kaufmann Publishers Inc., San Francisco, United States. 1996. 904 p.

4. Fokink, W. Distributed Algorithms: An Intuitive Approach. MIT Press, 2013. 248 p.

5. What is a Distributed System, and How Does it Work? Режим доступу: Distributed Algorithms: An Intuitive Approach - Wan Fokkink - Google Книги (дата звернення: 22.02.2023). Назва з екрану.

6. Multiprocessing – Process-based parallelism. Режим доступу: multiprocessing – Process-based parallelismPythonhttps://docs.python.org › library › multiprocessing. (дата звернення: 22.02.2023). Назва з екрану.

7. What is synchronous communication? Режим доступу: https://www.techsmith.com/blog/synchronous-vs-asynchronous-communication/#:~:text=Synchronous%20communication%20is%20the%20exchange,also%20examples%20of%20synchronous%20communication. (дата звернення: 22.02.2023). Назва з екрану.

8. Latha, C. A. Clock Synchronization in Distributed systems. *5th International Conference on Industrial & Information Systems*, N I T K, Surathkal, India; 29 July – 1 August, 2010. 6 p.

9. Møller, A., Schwartzbach, M. I. Static Program Analysis. Published in Encyclopedia of Cryptography and Security. 2011. 202 p.

10. Kaufmann M., Pecchiari P. Interaction with the Boyer-Moore Theorem Prover: A Tutorial Study Using the Arithmetic-Geometric Mean Theorem. Technical Report. Computational Logic, Inc. Austin, Texas. 1994. 116 p.

11. Dillinger, P. C. Manolios, P., Vroon D. ACL2s: “The ACL2 Sedan”. *Electronic Notes in Theoretical Computer Science*. 2007. Vol. 174. P. 3–18.

12. Gonthier, G. Formal proof–the four-color theorem. *Notices of the AMS*. 2008. Vol 55 (11). P. 1382-1393.

13. COMPCERT. Режим доступу: https://compcert.org/ (дата звернення: 22.02.2023). Назва з екрану.

14. Early history of Coq. Режим доступу: https://coq.inria.fr/refman/history.html (дата звернення: 22.02.2023). Назва з екрану.

15. What Is Dynamic Analysis? Режим доступу: https://totalview.io/blog/what-dynamic-analysis (дата звернення: 22.02.2023). Назва з екрану.

16. Что такое отладка? Режим доступу: https://aws.amazon.com/what-is/debugging/ (дата звернення: 22.02.2023). Назва з екрану.

17. "What is code profiling? Learn the 3 Types of Code Profilers". *Stackify Developer Tips, Tricks and Resources*. Disqus. 2016.

18. Bradley, J. Memory Analysis. In OS X Incident Response, 2016. P. 143 – 174.

19. Neystadt, J. "Automated Penetration Testing with White-Box Fuzzing". 2008. Microsoft. Retrieved 2009-05-14.

20. Reger, G., Falcone, Y., & Havelund, K. (2013). A Tutorial on Runtime Verification. In *Summer School Marktoberdorf 2012 - Engineering Dependable Software Systems* (Vol. 34). IOS Press.

21. Lamport, L., Shostak, R., Pease, M. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 1982. Vol. 4 (3). P. 382-401.

22. Sari A., Akkaya M. Fault Tolerance Mechanisms in Distributed Systems. International Journal of Communications, Network and System Sciences. 2015. Vol.08 (12). 12 p. 10.4236/ijcns.2015.812042

23. Scalability in Distributed Systems. Режим доступу: https://www.math.unipd.it/~abujari/fis1819/lecSlides/scalability.pdf (дата звернення: 22.02.2023). Назва з екрану.

24. Reiher P. Distributed System Security. In: Operating Systems: Three Easy Pieces. Remzi Arpaci-Dusseau, By (author): Andrea Arpaci-Dusseau. University of Wisconsin-Madison, 2013. 714 p.

25. Data consistency in the latest distrubuted database technology. Режим доступу: https://www.ibm.com/cloud/architecture/articles/hadr-containers/hadr-containers-1-consistency/ (дата звернення: 22.02.2023). Назва з екрану.

26. What is Use Case Diagram? Режим доступу: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/ (дата звернення: 22.02.2023). Назва з екрану.

27. Bell D. Explore the UML sequence diagram. Technical Library, IBM Rational Software, February 2004.

28. Bell D. The UML 2 class diagram. An introduction to structure diagrams in UML 2. Technical Library, IBM Rational Software, September 2004.

29. Python 3.3.0. Режим доступу: https://www.python.org/downloads/release/python-330/ (дата звернення: 22.02.2023). Назва з екрану.

30. The complete PyQt5 tutorial — Create GUI applications with Python. Режим доступу: https://www.pythonguis.com/pyqt5-tutorial/

31. time — Time access and conversions. Режим доступу: https://docs.python.org/3/library/time.html (дата звернення: 22.02.2023). Назва з екрану.

32. random — Generate pseudo-random numbers. Режим доступу: https://docs.python.org/3/library/random.html (дата звернення: 22.02.2023). Назва з екрану.

33. typing — Support for type hints. Режим доступу: https://docs.python.org/3/library/typing.html (дата звернення: 22.02.2023). Назва з екрану.

34. GlobalInterpreterLock. Режим доступу: https://wiki.python.org/moin/GlobalInterpreterLock (дата звернення: 22.02.2023). Назва з екрану.

35. Use PyQt's QThread to Prevent Freezing GUIs. Режим доступу: https://realpython.com/python-pyqt-qthread/ (дата звернення: 22.02.2023). Назва з екрану.

36. Lai T., Yang T. On distributed snapshots. *Information Processing Letters*, 1987. Vol. 25(3). P.153–158.

37. log (log file). Режим доступу: https://www.techtarget.com/whatis/definition/log-log-file (дата звернення: 22.02.2023). Назва з екрану.