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»
Колотило О.А.

Керівник Жолткевич Г.М.

Рецензент:

ХАРКІВ - 2023

ЗМІСТ

ВСТУП

Галузь розподілених систем є над важливою у сучасному світі. Не дивлячись на те, що на перший погляд ми не дуже часто стикаємось із такого роду системами, вони оточують нас всюди та вже є невід'ємною частиною нашого життя. Яскравими прикладом такої системи є мережа інтернет [1], яка розгалужена по всьому світу. Без інтернету тепер складно уявити світ.

Людство завжди хотіло убезпечити себе, зробити так щоб державні рішення були грунтовні та державна система була стійкою до рішень окремої взятої людини. Саме через це нині майже всі інститути розвинених держав є розподіленими та колективними органами [2], від виборчої та судової системи до військових міністерств. Розподіленість дозволяє зробити систему стійкою гнучкою та надійною.

Формально кажучи розподілені системи (distributed systems) [3]- це системи, які складаються з декількох комп'ютерів, що співпрацюють між собою, щоб виконувати спільні завдання. Тому верификація та тестування розподіленої системи є надваждивою частиною розробки такого програмного забезпечення. Є два принципово різних типи верифікації програми: динамічний та статичний аналіз. Статичний аналіз програмного коду передбачає огляд коду програми без її запуску, а динамічний оцінює досягнення поставлених задач за різних умов безпосередньо під час виконання програми.

Метою даної роботи є побудова уніфікованої математичної моделі для динамічного аналізу алгоритмів для розподілених систем.

Для досягнення поставленої мети необхідно зробити наступне:

1. Провести аналіз існуючих матеріалів, статей, книг за темою дослідження.
2. Обґрунтувати обрані математичні моделі розподіленої системи.
3. Зазначити список вимог до програмної моделі та розробити архітектуру системи.
4. Дослідити вимоги та архітектуру, відповідно до чого обрати програмні інструменти.
5. За допомогою обраних програмних інструментів та на основі математичної моделі й розробленої програмної архітектури імплементувати програмну модель.
6. Обрати алгоритм для тестування програмного прототипу.
7. Доповісти про результати роботи.

Новизна роботи полягає в використанні підходів математичної моделі розподілених обчислень Ненсі Лінч та програмна реалізація на їх основі для практичного застосування.

Практичне значення цієї роботи полягає в можливості динамічного аналізу розподілених алгоритмів на звичайному комп’ютері, з рівнем абстракції, що дозволяє легко та ефективно моделювати різноманітні конфігурації розподілених систем. Також розроблену програму можна використати у навчальних цілях. За допомогою графічного інтерфейсу можна демонструвати роботу компонентів розподіленої системи та принцип роботи відомих алгоритмів.

РОЗДІЛ 1

ДЕЯКІ ВІДОМОСТІ З ТЕОРІЇ РОЗПОДІЛЕНИХ ОБЧИСЛЕНЬ, ДИНАМІЧНОГО ТА СТАТИЧНОГО АНАЛІЗІВ СИСТЕМ

1.1 Концепція розподілених обчислень, основні поняття

За останні десятиліття обчислювальна техніка швидко вдосконалювалася, темпи зростання її можливостей були експоненційними, що досить добре описувалося законом Мура [4]. Закон Мура - це спостереження, сформульоване Гордоном Муром, співзасновником компанії Intel, в 1965 році, та є емпіричним принципом в розробці комп'ютерної техніки. Закон стверджує, що кількість транзисторів на мікрочіпі подвоюється приблизно кожні два роки, в той же час зменшуючи їх розмір і збільшуючи продуктивність.

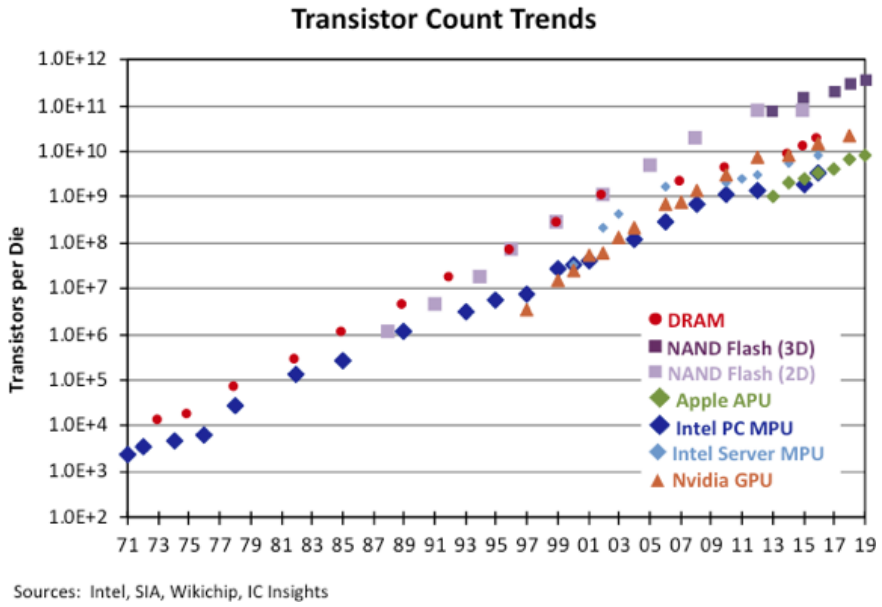

Рис 1.1. Графік числа транзисторів на мікрочіпах з 1971 по 2019 роки

Проте експоненційне зростання відбулося і в об’ємах обчислень, у темпах навіть переверщуючих розвиток мікрочіпів. Тепер багатьом компаніям потрібні обчислювальні потужності для роботи за моделями машинного навчання, для хостингу високодоступних сервісів та багато іншого. Така ситуація призводить до того, що компаніям потрібно

збільшувати витрати на “залізо”, тому набули популярності хмарні обчислювальні сервіси. Вони за низьку ціну забезпечують масштабованість ресурсів для обчислень, відмовостійкість та високодоступність. Ці сервіси - мережа комп’ютерів, що працює над спільною задачею. Такі мережі є розподіленими системами.

Розподілені обчислення - це підхід до обчислення, в якому різні комп'ютери або системи співпрацюють разом для вирішення обчислювальних завдань [5]. Обробка даних та виконання обчислень розподіляється між різними вузлами, які можуть бути розташовані на різних фізичних машинах або мережевих вузлах. Основна ідея розподілених обчислень полягає в тому, щоб використовувати різні ресурси, такі як потужність обчислення, пам'ять, мережеве з'єднання та інші, для вирішення складних обчислювальних задач. Це дозволяє отримати високий рівень продуктивності та ефективності, а також забезпечити більш високу надійність та доступність системи, оскільки випадкові збої на одному вузлі не можуть повністю зупинити роботу системи в цілому.

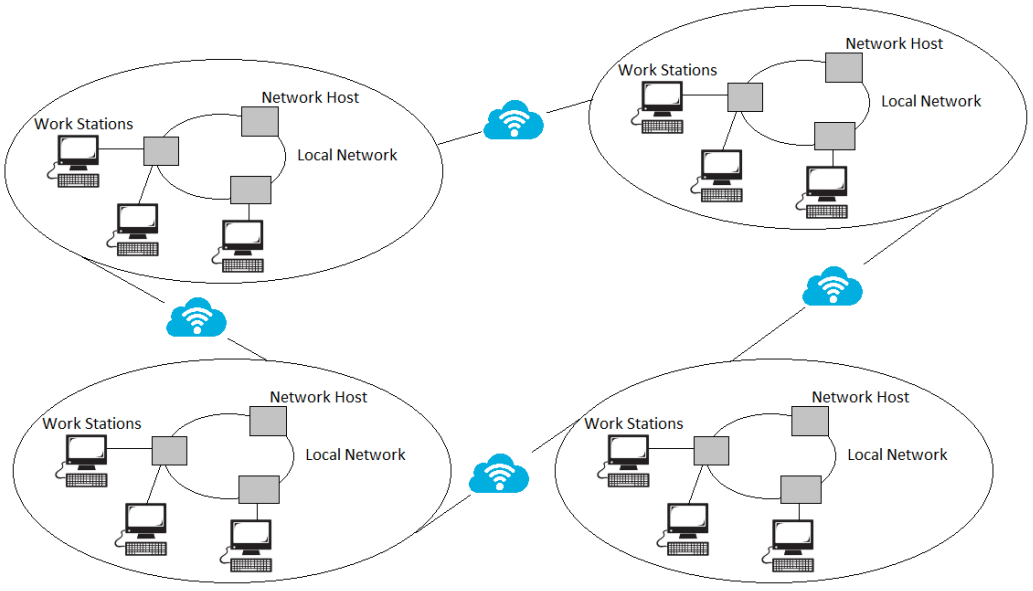

Рис 1.2. Приклад типової сучасної системи для розподілених обчислень

Однією з ключових переваг розподілених обчислень є можливість масштабування. За допомогою розподіленого підходу, системи можуть бути розширені вгору або вниз, швидко адаптуючись до змін в обсягах даних чи обчислювальних ресурсів. Це дозволяє ефективно

використовувати ресурси та забезпечувати високий рівень продуктивності, навіть при збільшенні розмірів задач або даних.

Формальне визначення розподіленої системи: **розподілена система** - це мережа з N вузлів або процесів, що відрізняються один від одного, за ідентифікатором або іншим набором характеристик, поєднання яких робить вузол унікальним [6]. Комунікація між вузлами відбувається через визначені канали комунікації. Якщо зображувати мережу розподіленої системи як граф, то канал комунікації між процесом А та Б - ребро між вершинами відповідним А та Б. Граф мережі в нашій моделі системи розподіленої системи має бути зв'язним. Граф не має обмеженості на орієнтованість.

Перейдемо до важливих понять розподілених обчислень.

Процесс або вузол. Процесс може зберігати будь-яку апріорну інформацію щодо себе та ту, що отримав в ході роботи розподіленої системи через **канали комунікації**. Процесс може обробляти три типи подій:

1. Внутрішні - впливають тільки на стан даного процесу (маніпуляції зі змінними, виклики внутрішніх функцій і.т.д).
2. Відправлення - передає деяку інформацію між двома процессами
3. Отримання - завжди є наслідком відповідної події відправлення.

Процес називають ініціатором за умови, що він може розпочати роботу самостійно (незалежно від інших процесів). Залежно від типу алгоритму таких ініціаторів може бути більше одного.

Головна задача **каналів комунікації** - асинхронна передача повідомлень. Канали повинні мати наступні властивості:

1. З обмеженням FIFO або без (повідомлення можуть переганяти один одного)
2. Кожне повідомлення має бути доставлене за кінцевий проміжок часу
3. Повідомлення не дублюються
4. Повідомлення не пропадають

Треба відзначити, що для деяких моделей розподілених обчислень можна знехтувати якимись з пунктів.

Конфігурація розподіленої системи - глобальний стан системи, що змінюється через переходи між множиною допустимих станів. Конфігурація складається з локальних станів процесів та повідомлень в каналах комунікації.

Формально поведінка будь-якої системи визначається так:

• C - множина всіх можливих станів системи,

• бінарне відношення переходу станів $\rightarrow$ на C

• $I \subseteq C$ початкова конфігурація системи.

Конфігурація є термінальною якщо для неї немає переходу до будь-якої конфігурації: $\gamma \rightarrow \delta$ для жодного $\delta \in C$. Виконання розподіленої системи є послідовністю $\gamma_0\, \gamma_1\, \gamma_2 \cdots$ 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 $\gamma_k = \delta$ [7].

Комунікацію розподіленої *системи* можна умовно поділити на два типи.

- **Black box**, де вузли не можуть коммуніковати між собою на пряму та можуть не мати інформації про сусідні вузли та топологію, а тільки за допомоги системи комунікації. А в свою чергу система комунікації вже передає повідомлення вузлам, яким вважає за потрібне
- У **White box**, на відміну від **black box**, вузли мають інформацію про своїх сусідів та мають можливість самостійно вибирати адресата повідомлень.

1.2. Аналіз програм та систем. Цілі та підходи.

Задачею програмного та системного аналізів на сам перед є аналіз поведінки та інших властивостей коду програми або системи [8]. Такого роду аналіз має знаходити вразливості, баги та інші аномалії в роботі системи. Деякі типи програмного аналізу фокусується на знаходженні поганих практик програмування або “bad code smells”. Багато всесвітньо відомих компанії навіть використовують рішення для знаходження та автоматичного виправлення проблем. Загалом програмний аналіз поділяють на дві категорії за джерелом аналізу: статичний - аналізує код без виконання, а динамічний поведінку під час безпосереднього виконання. Статичний та динамічний аналізи є важливими інструментами для забезпечення якості програмного забезпечення [9][10]. Обидва підходи мають свої переваги та недоліки.

Основними перевагами статичного аналізу є:

Перевірка на відповідність стандартам: статичний аналіз може допомогти виявити помилки, які порушують стандарти програмування, такі як стиль коду, використання змінних, розміщення блоків коду та ін.

Виявлення потенційних помилок: статичний аналіз може допомогти виявити помилки, які можуть виникнути при виконанні програми,

наприклад, недостатні перевірки на помилки вводу/виводу даних, можливі помилки при роботі з пам'яттю та ін.

Покращення продуктивності: статичний аналіз може допомогти знайти частини коду, які можуть бути оптимізовані для покращення продуктивності.

Недоліком статичного аналізу є те, що він не може виявити помилки, які виникають під час виконання програми, наприклад, помилки вводу/виводу даних, помилки в роботі з мережевими протоколами та ін.

Перші інструменти статичного аналізу були розроблені на початку 1970-х років. Вони були розроблені для виявлення помилок у програмах на мові асемблера. Протягом наступних десятиліть інструменти статичного аналізу розвивалися та ставили більш складними, здатними виявляти більш широкий спектр помилок у програмах.

У 1980-х роках з'явилися інструменти статичного аналізу для мов вищого рівня, таких як C та C++. Ці інструменти були спрямовані на виявлення таких помилок як: невірне використання указателів, некоректні операції з пам'яттю та інші помилки, які можуть загрожувати безпеці програмного забезпечення.

У 1990-х роках інструменти статичного аналізу стають все більш популярними, та активно використовуються при розробці тогочасними гігантами в сфері розробки програмного забезпечення, в таких галузях, як медицина, автомобільна промисловість, аерокосмічна промисловість тощо.

У 2000-х роках інструменти статичного аналізу продовжили розвиватися та стали ще більш потужними. Вони були пристосовані до виявлення ширшого спектру помилок, включаючи потенційні помилки в

безпеці, а також були покращені з точки зору продуктивності та використання.

На сьогоднішній день, статичний аналіз є невід'ємною частиною розробки софту та забезпечення його безпеки й надійності. Особливо важливим стало використання статичного аналізу в критичних сферах, таких як медицина, авіаційна та автомобільна промисловість, фінансові послуги та інші. У цих галузях невірна робота програмного забезпечення може призвести до небезпечних ситуацій, таких як аварії транспорту, помилки в медичних діагнозах, порушення безпеки фінансових даних та інші.

Варто також відзначити популярні інструменти статичного аналізу:

Інструмент PVS-Studio для статичного аналізу коду мовами C, C++, C# та Java.

ESLint - інструмент для статичного аналізу JavaScript-коду.

Clang Static Analyzer - інструмент статичного аналізу для мов C та C++.

Окреме місце серед сучасних інструментів статичного аналізу займає мова Coq [11]. Coq - це формальна мова для написання математичних визначень, виконавчих алгоритмів та теорем разом із середовищем для напів інтерактивної розробки машинно-перевірених доказів. Один з основних принципів Coq полягає в тому, що програми не тільки виконуються, але їх коректність може бути формально доведена. Це забезпечує високу надійність та безпеку софту, що є особливо важливим у критичних сферах. Coq використовується для формального доведення коректності компіляторів, алгоритмів шифрування, оптимізації та інші. За допомогою нього можна проводити математичні доведення, перевіряти властивості програм та проводити формальну верифікацію систем.

Один з прикладів використання Coq для статичного аналізу - це проект CompCert, який розробляється з метою створення компілятора для мови C, який генерує надійний та безпечний машинний код. У рамках проекту використовуються формальні доведення на Coq для того, щоб довести коректність оптимізацій та трансляцій мови C.

У цілому, використання мови програмування Coq в статичному аналізі дозволяє забезпечити високу надійність та безпеку програмного забезпечення за рахунок формальної верифікації програм та алгоритмів.

Існує кілька альтернатив мові Coq, які також використовуються для доведення теорем та верифікації програмного забезпечення. Деякі з них:

Isabelle/HOL - інтерактивна система доведення теорем, що базується на логіці вищого порядку (HOL). Вона дозволяє верифікувати програмне забезпечення та формалізовувати математичні теорії.

Agda - це функціональна мова програмування з вбудованою системою доведення теорем. Вона базується на залежних типах і дозволяє верифікувати програмне забезпечення та формалізовувати математичні теорії.

Lean - це мова програмування з вбудованою системою доведення теорем. Вона базується на залежних типах і дозволяє верифікувати програмне забезпечення та формалізовувати математичні теорії.

Динамічний аналіз - це метод аналізу програмного забезпечення, який виконується під час виконання програми. Цей метод дозволяє виявити помилки та дефекти програмного забезпечення, які можуть бути пропущені при статичному аналізі. Він доповнює статичний аналіз, дозволяючи зібрати додаткову інформацію про програму та виявити раніше не помічені недоліки.

Для динамічного аналізу зазвичай використовуються спеціальні інструменти, які називаються "профайлерами". Профайлер - це програмний інструмент, який відстежує виконання програми та збирає дані про час виконання, кількість викликів функцій, кількість використаних ресурсів та інші характеристики. Ці дані потім можуть бути використані для виявлення проблем в програмі та їх виправлення. Динамічний аналіз використовується в різних галузях програмування, включаючи веб-розробку, мобільну розробку та *embedded* розробку.

Основні види динамічного аналізу:

- Тестування програми [12]. Цей метод полягає в запуску програми з різними вхідними даними та вивченні результатів виконання. Тестування може бути автоматизоване за допомогою спеціальних тестових фреймворків.
- Профілювання програми [13]. Цей метод полягає відстеженні виконання програми та зборі статистики про її роботу. Інформація, зібрана профайлером, може бути використана для виявлення узких місць в програмі та їх оптимізації.
- Дебаггінг програми [14]. Цей метод полягає в ручному відстеженні виконання програми з метою виявлення та виправлення помилок.

Серед **переваг** динамічного аналізу відзначають:

- Можливість виявити помилки, які не можуть бути виявлені статичним аналізом.
- Можливість зібрати велику кількість інформації про роботу програми.

Обмеженість динамічного аналізу полягає у тому, що він може зібрати інформацію лише про ті помилки, які з'являються під час

виконання програми з конкретним вхідним набором даних. Також динамічний аналіз може бути дуже часо та ресурсомістким. Під час виконання програми профайлером збирається багато даних, що може збільшити час виконання програми та використання системних ресурсів. Динамічний аналіз також наразі має проблеми зі збором інформації про певні аспекти програми, наприклад, якщо програма містить багато асинхронних запитів до бази даних, профілювання цієї програми може бути складним.

Історія динамічного аналізу пов'язана з історією програмування взагалі. Перші програми, які були написані, складалися з малої кількості інструкцій, і тому їх було досить легко перевірити вручну. З часом, коли розмір програм став більшим, стало складніше виявляти помилки в програмах, і саме тоді почали з'являтися інструменти динамічного аналізу.

У 1950-х роках були розроблені перші електронні комп'ютери, і це прискорило розвиток динамічного аналізу. Першим інструментом динамічного аналізу був "debugger", який дозволяв відстежувати виконання програми та знаходити помилки. У 1960-х роках були розроблені перші мови програмування високого рівня, такі як COBOL та FORTRAN, і з'явилася потреба в інструментах динамічного аналізу для цих мов.

У 1970-х роках почали розроблятися інструменти для автоматизації процесу тестування програмного забезпечення, такі як автоматичні генератори тестових наборів та інструменти для автоматичного виявлення помилок в програмах. У 1980-х роках почали розроблятися інструменти динамічного аналізу для віддаленого виконання програм на віддалених комп'ютерах та віртуальних машинах.

У 1990-х роках почали з'являтися інструменти динамічного аналізу, які були спрямовані на виявлення помилок, пов'язаних зі збільшенням кількості потоків виконання та багатопоточної обробки даних. У цей період також були розроблені інструменти динамічного аналізу для веб-розробки, такі як інструменти для аналізу веб-сторінок та інструменти для тестування веб-додатків.

З 2000-х років динамічний аналіз став все більш популярним, оскільки з'явилися нові інструменти та технології, такі як мобільні пристрої, хмарні обчислення, штучний інтелект та машинне навчання, які потребували нових методів динамічного аналізу.

Сьогодні динамічний аналіз використовується в багатьох галузях програмування, включаючи веб-розробку, мобільні додатки, штучний інтелект, машинне навчання та безпеку програмного забезпечення.

Ось кілька прикладів інструментів для динамічного аналізу:

- Інструменти для профілювання коду, такі як Valgrind та Intel VTune. Вони дозволяють знайти більшість проблем, пов'язаних з пам'яттю, такі як витоки пам'яті, неправильна робота з пам'яттю та інші.
- Інструменти для пошуку помилок, такі як AddressSanitizer (ASan) та UndefinedBehaviorSanitizer (UBSan). ASan знайде витоки пам'яті, UBSan - помилки, що можуть призвести до несправних поведінок, таких як ділення на нуль, переповнення буфера тощо.
- Інструменти для тестування програмного забезпечення, такі як Selenium, PyTest та JUnit. Вони дозволяють автоматизувати процес тестування та забезпечують повний охоплення коду тестами.
- Інструменти для аналізу безпеки, такі як Burp Suite та OWASP ZAP. Вони використовуються для виявлення потенційних вразливостей та вразливих місць у веб-додатках.

- Інструменти для моніторингу, такі як Nagios та Zabbix. Вони дозволяють відслідковувати різні метрики роботи системи та повідомляти про можливі проблеми.

1.3. Відкриті проблеми в галузі розподілених систем

Розподілені системи мають ряд типових проблем. Деякі з найбільш поширених проблем розподілених систем містять:

- Керування вузлами: Кожен вузол у розподіленій системі має свої власні ресурси та можливості, і керування ними може бути вкрай складним.
- Комунікація: У розподіленій системі вузли зазвичай зв'язані мережею, і комунікація між ними може бути проблемою, особливо якщо мережа перенасичена або неправильно сконфігурована.
- Консистентність: Якщо вузли розподіленої системи працюють незалежно, то інформація може бути розбіжною між ними, що може призвести до проблем з консистентністю.
- Безпека: Розподілені системи піддаються різним видам атак, таким як відмови в обслуговуванні, зловживання доступом або злом.
- Масштабованість: Розподілені системи можуть мати проблеми з масштабуванням, особливо якщо вони повинні працювати з великою кількістю даних або великою кількістю користувачів.
- Резервне копіювання та відновлення: У разі виникнення проблем з розподіленою системою може знадобитися резервне копіювання та відновлення даних, що може бути складним завданням.
- Управління транзакціями: Якщо розподілена система забезпечує транзакційні послуги, то керування транзакціями може бути складним завданням, особливо якщо транзакції здійснюються між різними вузлами.

Наочно можна бачити, ці проблеми в базах даних та підходи для їх вирішення в СУБД [15]. Ці декілька підходів:

- **Блокування (Locking)**. Цей підхід заснований на використанні блокування ресурсів бази даних, щоб забезпечити синхронізацію доступу до них. Коли користувач звертається до ресурсу, він блокує його, щоб інші користувачі не могли змінити цей ресурс параллельно. Коли користувач закінчує роботу з ресурсом, він знімає блокування, аби інші могли продовжити роботу з ним. Це може призвести до блокування ресурсів на довгий час, якщо багато користувачів звертаються до них одночасно (дедлоки та starvation).
- **Керування версіями (Versioning)** забезпечує синхронізацію бази даних за допомогою версіонування. Кожен раз, коли користувач змінює дані, створюється нова версія даних, що дозволяє користувачам працювати зі старими версіями, поки інші користувачі не завантажили нові версії. Цей підхід дозволяє уникнути блокування ресурсів, але може збільшити обсяг даних у базі даних.
- **Транзакції (Transactions).** Бази даних синхронізують за допомогою транзакцій. Цей підхід дозволяє уникнути конфліктів при зміні даних, але може зменшити продуктивність системи, особливо якщо багато користувачі
- **Реплікація (Replication)** копіює дані з одного сервера на інший за допомогою чого підтримує синхронізацію. Кожен сервер містить копію бази даних, і якщо дані змінюються на одному сервері, вони автоматично оновлюються на іншому сервері. Цей підхід дозволяє забезпечити доступність даних навіть у разі відмови одного з серверів, але може призвести до конфліктів даних, які не вдасться вирішити автоматично.
- **Шардування (Sharding)** поділяє дані на декілька частин (шардів) та зберігає їх на різних серверах, за рахунок чого підтримується

консистентність БД. Кожен сервер містить лише певну частину даних, і якщо користувач звертається до даних, система визначає, на якому сервері необхідні дані знаходяться та повертає їх користувачу. Цей підхід дозволяє забезпечити високу доступність та продуктивність бази даних, але може збільшити складність розробки та утримання системи.

Також наразі дуже важливими є **системи оркестрації** [16]. Такі системи - це інструменти для керування та автоматизації ПЗ, які розгортаються у розподілених середовищах, що є окремим випадком розподіленої системи. Основна мета систем оркестрації полягає в тому, щоб забезпечити простір для розгортання ПЗ у швидкому та безпечному режимі, а також підтримувати відповідність цього ПЗ стандартам безпеки та керування ресурсами. На практиці вони допомагають забезпечити ефективну роботу розподілених систем, які стають все більш складними та потужними з плином часу. Вони дозволяють забезпечувати високу доступність, безпеку та продуктивність систем, що є важливим для багатьох компаній та організацій.

Великою відкритою проблемою є те, що існуючі рішення більшою мірою вирішують перелічені раніше проблеми для вузького набору програм, але не дають належних інструментів до розробки та тестування роботи розподілених алгоритмів та софту для розподілених систем на моделях з різними конфігураціями системи, що уповільнює вирішення проблем розподілених систем в інших галузях.

1.4. Уточнення формулювання задачі

В попередніх пунктах ми навели основні поняття в розподілених системах, також зазначили принципи на яких базуються підходи до верифікації будь-яких програм. Пропонується використати цю

методологію, а саме статичний та динамічний аналізи для створення моделі розподілених систем, та перевірити зазначену модель шляхом тестування на відомих та математично-доведених розподілених алгоритмів.

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

В основі нашої будь-якої системи повинна лежати модель, для нашої системи була обрана математична модель розподілених обчислень запропонована Н. Лінч [7]. Ця модель заснована на новій концепції I/O (input-output) Automata та переходах між можливими станами системи.

I/O автомат для розподіленої системи це компонент, що має можливість взаємодіяти з іншими компонентами системи [7]. В сутності є окремим випадком кінцевого автомату, де переходи між станами відбуваються після події input та output. Як вже було зазначено по класифікуються як вхідні, вихідні або внутрішні.

Процес в асинхронній розподіленій системі підпадає під визначення I/O автомата.

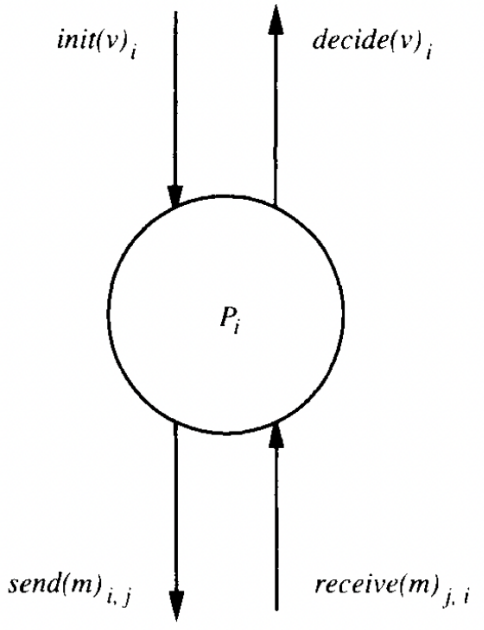

Рис 2.1. I/O автоматон процесу P_i

Автомат позначений P_i, вхідні та вихідні стрілки позначають вхідні та вихідні події відповідно. Внутрішні дії не відображаються. Зображений

автомат отримує вхідні дані у формі $init(v)_i$ ззовні 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.

Поведінку FIFO каналу можна також змоделювати за допомогою I/O автомату, де вхідні події є подіями типу відправки повідомлення send (m), а вихідні події — події отримання повідомлення іншим автоматом — receive (m)

Рис 2.2. I/O автомат каналу фіфо

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. Підпис *S* — це трійка, що складається з трьох непересічних наборів дій: вхідні події, in(S), вихідні події, out(S), і внутрішні події, int(S). Ми визначаємо зовнішні події, ext(S), як in(S) U out(S); локально керовані події, local(S), як out(S) U int(S); і acts(S) — це всі події 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) $\subseteq$ 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') $\in$ trans(A)
- tasks (A), розділення задач, яке є відношенням еквівалентності на local(sig(A)), що має якнайбільш лічильну кількість класів еквівалентності

Треба відзначити, що в роботі Ж. Теля також пропонується математична модель розподілених обчислень, проте, на мій погляд, на відміну від моделі Ненсі Лінч, є занадто абстрактною, аби використовувати її в якості основи для програмної моделі [6].

Запропонована Н. Лінч модель теж має свої недоліки в контексті програмної імплементації. Так ця модель не була розроблена з урахуванням призупинення роботи системи або втрати повідомлення. Призупинення роботи необхідно для дослідження стану системи, а втрата повідомлень є невід'ємними характеристикої реальних каналів передачі даних. Якщо поведінки призупинення роботи системи можна досягти, за допомогою введення нових зовнішніх події pause/resume та станів автоматів pause/resume, що, за умов відповідної програмної імплементації, допоможе зберігати стан та підтримувати систему в консистентному вигляді, то от врату повідомлень модель автомату зовсім не підтримує. Більше того, якщо сторонній користувач хоче розширити поведінку, то не слід змушувати його імплементувати саме кінцевий-автомат.

Отже, систему ми побудуємо, беручи за основу математичну модель лінч, та наступними характеристиками:

- Повідомлення, що відправляються автоматом-процесом, рано чи пізно дійдуть до одержувача.
- Система комунікації з точки зору процесів - white box, де з канали знають тільки про наявних сусідів, а з точки зору реалізації - black box, де комунікаційна система виступає в ролі координатора між процесами.
- Канали - односпрямовані

- Процеси можуть бути зупинені, а їх стан законсервований (збережений)

Ми будемо досліджувати розподілену систему, яка зовні буде виглядати як, наприклад, на рисунку 2.3, а всередині як на рисунку

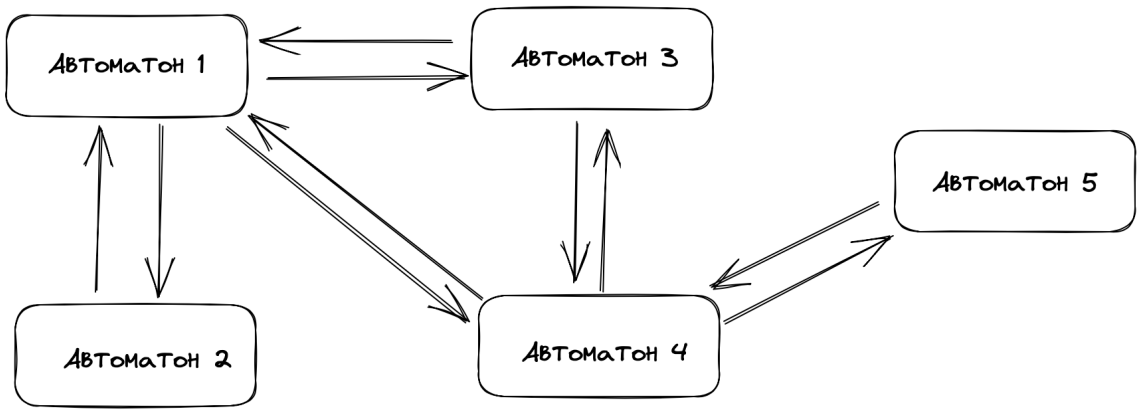

Рис 2.3 Приклад розподіленої системи

Побудова реальної розподіленої систем є складною так затратною задачею, отже ми маємо створити симуляцію розподіленої системи, з широким інструментарієм для дослідження роботи та налаштування характеристик компонентів системи. Тестування системи відбуватиметься на загальновідомих розподілених алгоритмах

У наступному розділі описуються деталі побудови розподіленої системи.

2.2 Архітектура програмного забезпечення моделі для динамічного аналізу розподілених алгоритмів

Під час розробки програмного модуля ми керувались принципами Clean architecture, SOLID та Design Patterns. Нижче описуються сенс цих принципів.

Clean Architecture (Чиста архітектура) - це концепція проектування програмного забезпечення, яка ставить на перший план питання

організації коду та розділення його на логічні шари, щоб забезпечити чистоту, зрозумілість та легкість розширення [17].

Основні принципи Clean Architecture включають:

- **Розділення на логічні шари.** Clean Architecture ставить задачу розділити код на шари з різними відповідальностями. Такий підхід дозволяє знизити залежність між різними частинами програми, забезпечуючи читабельність та легкість розширення.
- **Залежність від абстракцій, а не від конкретних реалізацій**. Цей принцип стверджує, що код має залежати від абстракцій, а не від конкретних реалізацій. Це дозволятиме змінювати реалізацію без впливу на інші частини коду.
- **Чистота та зрозумілість.** Код має бути чистим, зрозумілим та легким для розуміння. Це досягається шляхом застосування принципів SOLID, правильного використання шаблонів проектування та інших технік.
- **Незалежність від зовнішніх фреймворків.** Clean Architecture ставить задачу зменшити залежність від зовнішніх фреймворків та бібліотек. Це дозволяє підтримувати код на більш високому рівні абстракції та зменшує ризик втрати здатності до розширення.
- **Тестування.** Принцип тестування є одним з важливих принципів Clean Architecture. Його ідея полягає в тому, що тестування має бути легким та швидким процесом, який можна виконувати в автоматичному режимі. Це допомагає забезпечити стабільність та якість програмного забезпечення, підтримуючи його в готовому до випуску стані.

Принцип тестування Clean Architecture описується наступними кроками:

1. Використання інтерфейсів: класи мають реалізовувати інтерфейси, які можуть бути легко підмінені на моки (mock objects) під час тестування. Це дозволяє тестувати класи незалежно від інших компонентів.
2. Використання залежностей від зовнішніх компонентів: класи мають використовувати залежності від зовнішніх компонентів через інтерфейси, які можна легко замінити під час тестування. Це дозволяє тестувати класи незалежно від реальних зовнішніх компонентів.
3. Розділення логіки тестування та бізнес-логіки: класи треба мати окремі методи для логіки тестування та бізнес-логіки. Це дозволяє забезпечити, що тестування не перешкоджає бізнес-логіці та що тестові методи не містять непотрібного коду.
4. Використання автоматичних тестів: тести мають бути автоматичними та запускатися під час збірки програми. Це дозволяє швидко виявляти та виправляти типові помилки, підтримуючи високу якість програмного забезпечення.

SOLID - це абревіатура, що складається з п'яти принципів, які допомагають забезпечити гнучкість, розширюваність та підтримуваність програмного забезпечення, тобто принципи, що допомагають на практиці дотримуватись Clean Architecture. Кожна літера SOLID описує окремий принцип [18].

- **Single Responsibility Principle**: клас повинен мати лише одну відповідальність. Це означає, що клас має виконувати лише одну справу, а всі його методи повинні бути спрямовані на

досягнення цієї мети. Якщо клас виконує більше, ніж одну відповідальність, то зміни в одній частині можуть негативно вплинути на інші частини класу.

- **Open/Closed Principle**: клас повинен бути відкритий для розширення, але закритий для модифікації. Це означає, що клас повинен дозволяти додавати нову функціональність, не змінюючи існуючий код. Таким чином, зміни в існуючому коді не повинні впливати на код, який використовує цей клас.
- **Liskov Substitution Principle**: об'єкти підкласу повинні бути замінюваними за базовий тип. Це означає, що об'єкти підкласу повинні вести себе так само, як об'єкти базового класу, і не повинні змінювати інтерфейс класу-батька. Цей принцип допомагає забезпечити правильну ієрархію класів і зменшити кількість помилок в коді.
- **Interface Segregation Principle**: клієнти не повинні залежати від інтерфейсів, які вони не використовують. Це означає, що інтерфейси повинні бути розділені на менші, спеціалізовані інтерфейси, щоб клієнти могли використовувати тільки ту функціональність, яка їм потрібна, і не залежали від зайвого коду. Цей принцип допомагає забезпечити гнучкість та повторне використання коду.
- **Dependency Inversion Principle**: високорівневі модулі не повинні залежати від низькорівневих модулів, а обидва типи модулів повинні залежати від абстракцій. Це означає, що код повинен залежати від абстракцій, а не від конкретних реалізацій. Цей принцип допомагає забезпечити більшу гнучкість та зручність у зміні коду, зменшення залежностей між класами та підвищення тестової зручності.

Керуючись цими принципами для задачі були створені діаграми бізнес - прецедентів [29]:

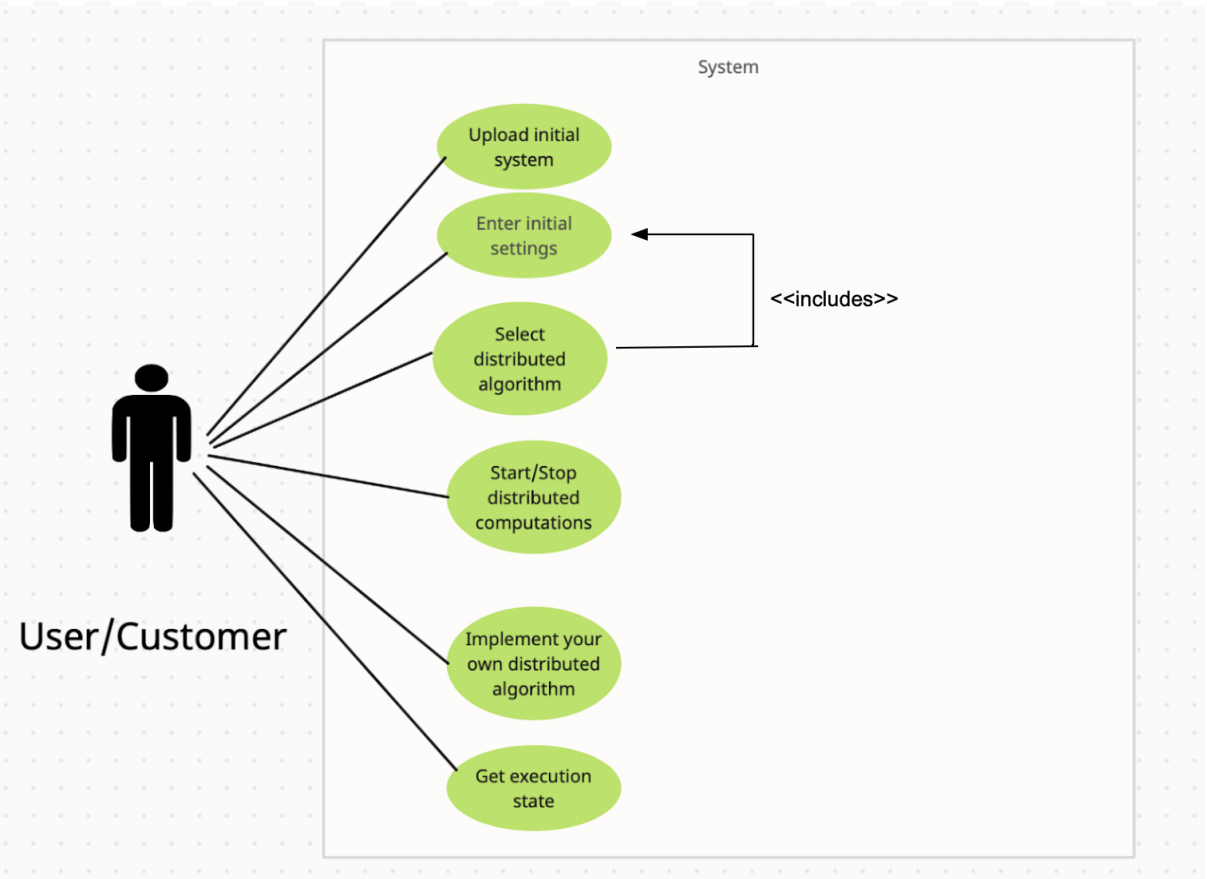

Рис 2.4. Діаграма бізнес-прецедентів

Опис бізнес-прецедентів:

→ Upload initial system

1. Короткий опис:
   Користувач завантажує граф-репрезентацію розподіленої системи
2. Передумова:
   Модуль запущений.
3. Перебіг подій:

Дія користувача	Відклик системи

Натиснути на вкладку “file” і обрати “upload graph from file”	Система дістає дані з файлу та перевіряє їх коректність. В інтерфейсі користувача мальовується меню та графічна репрезентація даних з файлу. Система зберігає граф.

4. Альтернативный перебіг подій:

   Якщо дані у файлі некорректні, (не той формат файлу, є вершини, що ні з ким не поєднані тощо) з’являється повідомлення про помилку.
5. Післяумова

Поява вікна з графічною репрезентацією розподіленою системи, система зберегла та показала поточну версію графу, з’явилося меню для взаємодії.

→ Enter initial settings

1. Короткий опис:
   Вибір початкової конфігурації розподіленої системи через налаштування
2. Передумова:
   Модуль запущено та виконано прецедент “Upload initial system”.
3. Перебіг подій:

Дія користувача	Відклик системи
Задає необхідні початкові дані: розподілений алгоритм, вірогідність доставки повідомлення, ініціатор алгоритму	Налаштування відображаються в інтерфейсі користувача. Налаштування зберігаються в системі.

4. Альтернативный перебіг подій:

5. Післяумова:

Вікно з графічною репрезентацією розподіленою системи, система зберегла та показала поточну версію графу, меню налаштувань містить поточні збережені налаштування.

→ Start algorithm

1. Короткий опис:
   Запуск алгоритму з процеса-ініціатора
2. Передумова:
   Модуль запущено, виконано прецедент “Enter initial settings”.
3. Перебіг подій:

Дія користувача	Відклик системи
З обраними початковими умовами, натискає “Start”	Система розпочинає алгоритм, з обраним алгоритмом у прецеденті “Enter initial settings”. В інтерфейсі користувача малюються повідомлення, що виникають під час комунікації між процесами в даному алгоритмі.

4. Альтернативный перебіг подій:

5. Післяумова:
   Вікно з графічною репрезентацією розподіленою системи, меню налаштувань містить поточні збережені налаштування, у реальному часі малюються повідомлення з каналів між процесами, алгоритм працює у бекграунді.

→ Stop algorithm

1. Короткий опис:
   Алгоритм ставиться на паузу
2. Передумова:
   Модуль запущено, виконано прецедент “Start algorithm”
3. Перебіг подій:

Дія користувача	Відклик системи
Натискає кнопку “Stop”	Всі процеси в моделі системи призупиняються, репрезентація повідомлень у графічному інтерфейсі припиняють переміщуватися. Текст

	кнопки змінюється зі “Stop” на “Resume”

4. Альтернативный перебіг подій:
   “Start algorithm” не відбувся, натискання на “Stop” ігнорується.
5. Післяумова:

Вікно з графічною репрезентацією розподіленою системи, меню налаштувань містить поточні збережені налаштування, робота алгоритму призупинена, візуалізації повідомлень між процесами припиняють рух. Текст кнопки замінений з “Stop” на “Resume”.

→ Resume algorithm

1. Короткий опис:
   Алгоритм знімається з паузи
2. Передумова:
   Модуль запущено, виконано прецедент “Stop algorithm”
3. Перебіг подій:

Дія користувача	Відклик системи
Натискає “Resume”	Всі процеси в моделі системи поновлюють роботу, репрезентація повідомлень у графічному інтерфейсі починають переміщуватися. Текст кнопки змінюється зі “Resume” на “Stop”.

4. Альтернативный перебіг подій:

5. Післяумова:

Вікно з графічною репрезентацією розподіленою системи, меню налаштувань містить поточні збережені налаштування, робота алгоритму поновлена, візуалізації повідомлень між процесами поновлюють рух. Текст кнопки замінений з “Resume” на “Stop”.

→ Exit

1. Короткий опис:
   Завершити програму

Передумова:
   Модуль запущено.

2. Перебіг подій:

Дія користувача	Відклик системи
Натискає на кнопку закриття вікна.	Програма завершується.

3. Альтернативный перебіг подій:

4. Післяумова:
   Програма закривається, використані під час роботи ресурси звільняються.

→ Get execution state

Короткий опис:

1. Отримується інформація про події, що виникли під час виконання алгоритму у розподіленій системі.
2. Передумова:
   Виконано прецедент “Start algorithm”
3. Перебіг подій:

Дія користувача	Відклик системи
Натискає на кнопку “Get execution state”	Система повертає перелік подій розподіленої системи в хронологічному порядку і виводить у текстовому форматі на екран. Розподілена система виконує прецедент “Stop algorithm”.

4. Альтернативный перебіг подій:

5. Післяумова:

Алгоритм стає на паузу, нове вікно з переліком подій розподіленої системи.

Користуючись цією моделлю прецедентів, були розроблені наступні sequence діаграми [19], що відповідають ланцюгам викликів класів та методів у системі:

1. Прецедент “Get execution state”

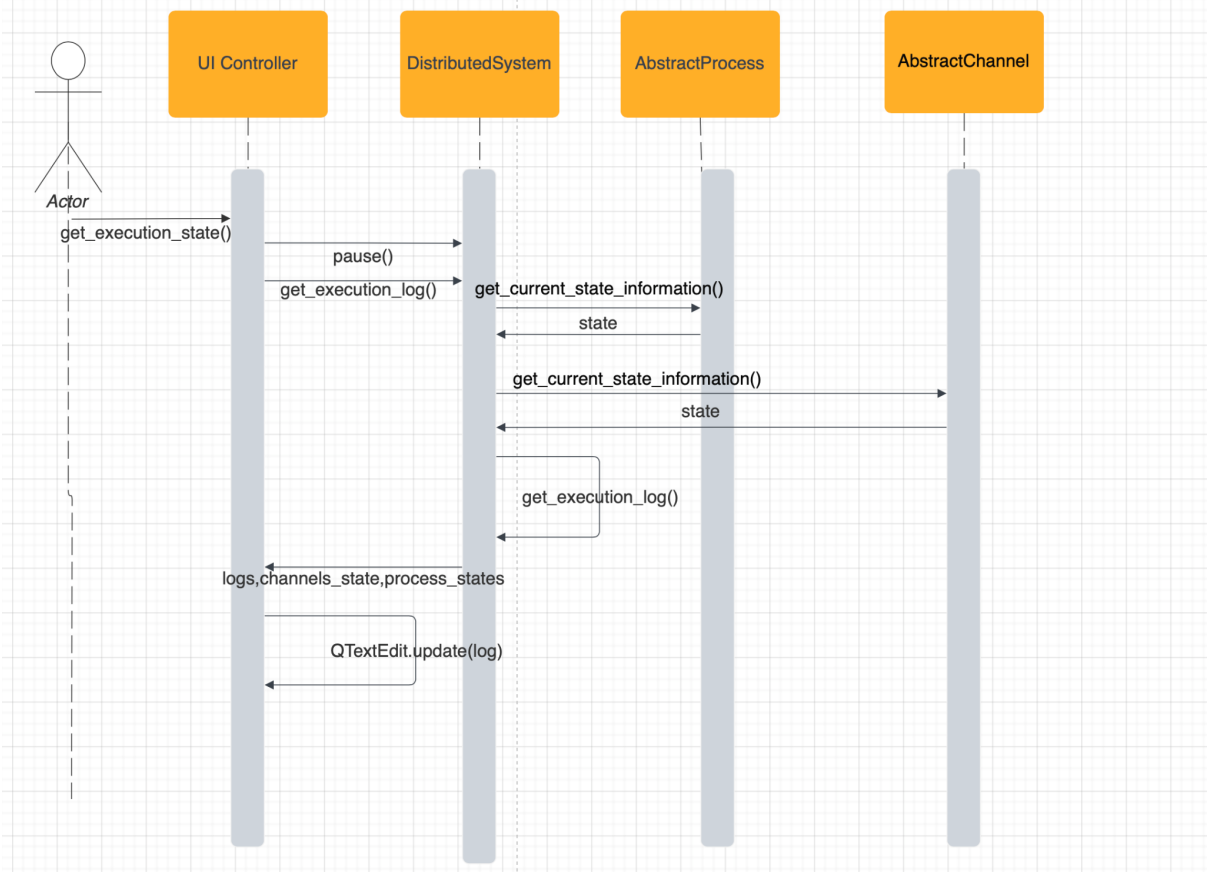

Рис 2. Сіквенс-діаграма “Get execution state”

2. Прецедент “Enter initial settings”

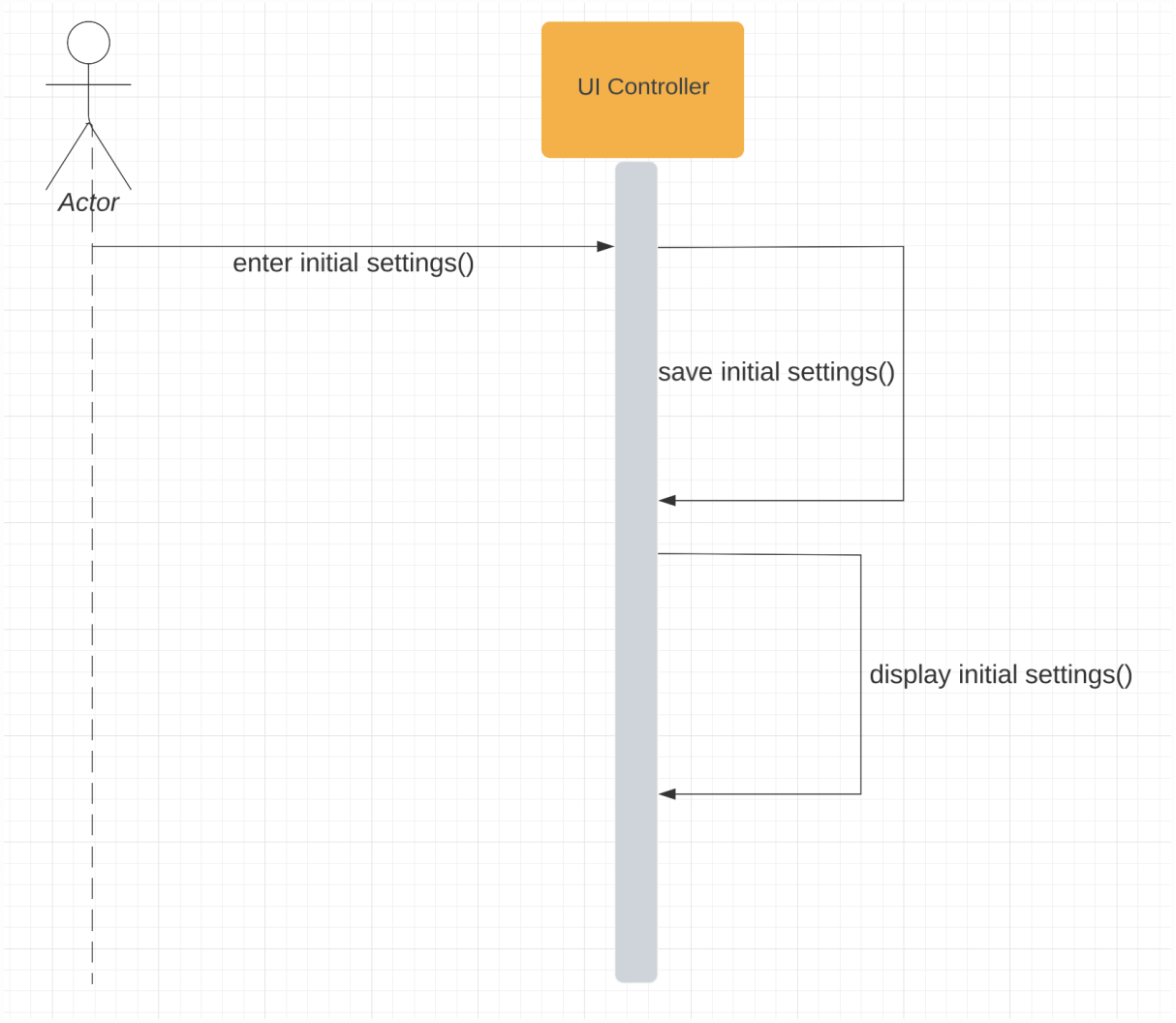

Рис 2. Сіквенс-діаграма “Enter initial settings”

3. Прецедент “Start algorithm”. Через великий розмір розбита на частини: зліва направо, згори донизу.

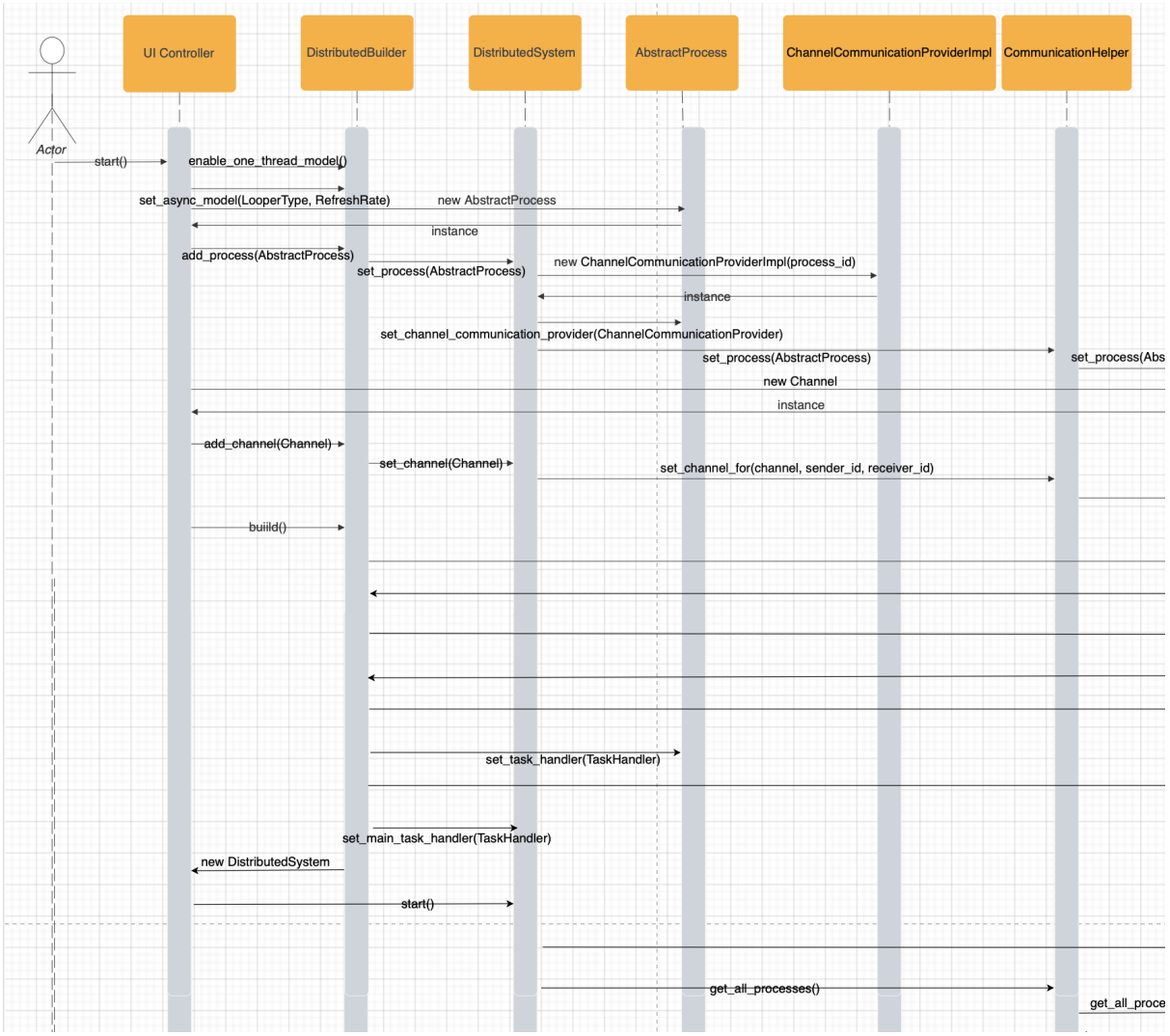

Рис 2.5. Сіквенс-діаграма “Start algorithm”. Лівий верхній кут.

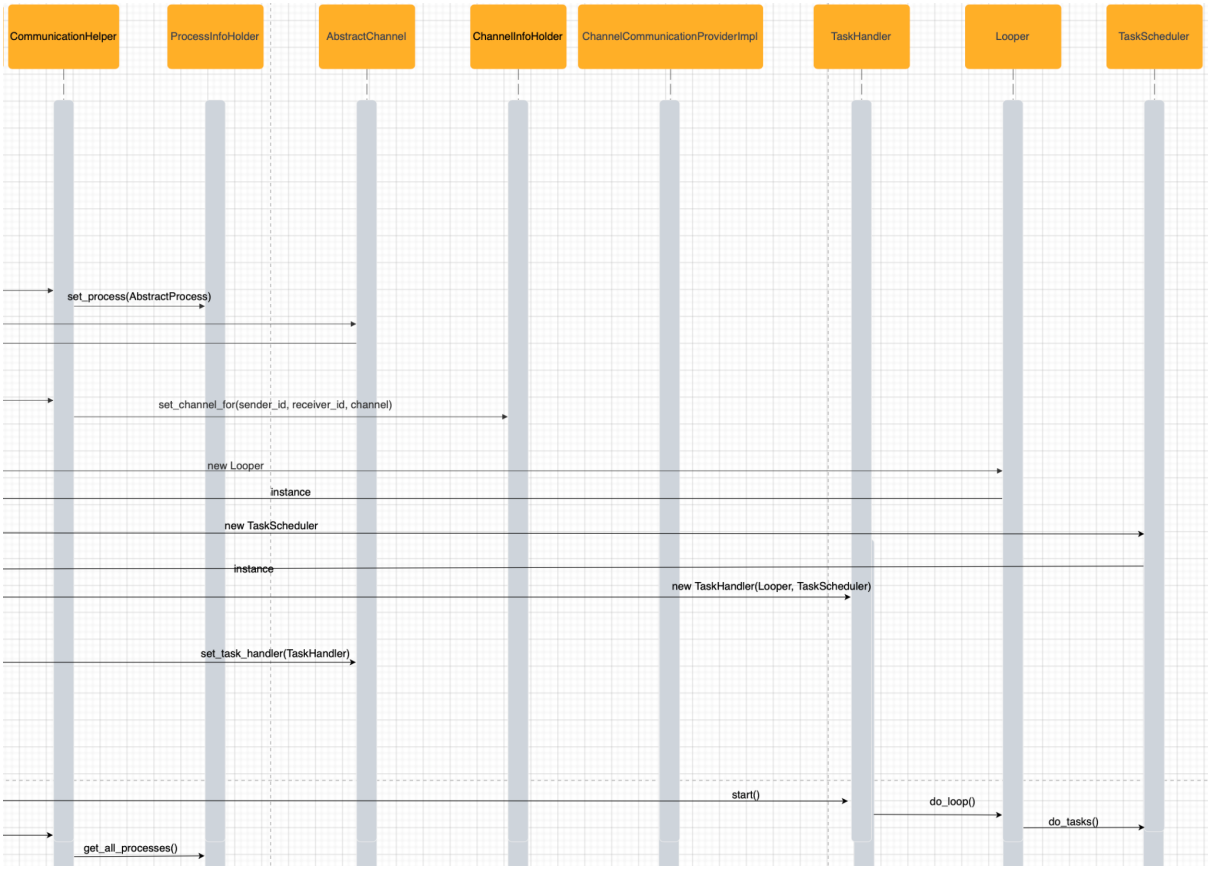

Рис 2.6. Сіквенс-діаграма “Start algorithm”. Правий верхній кут.

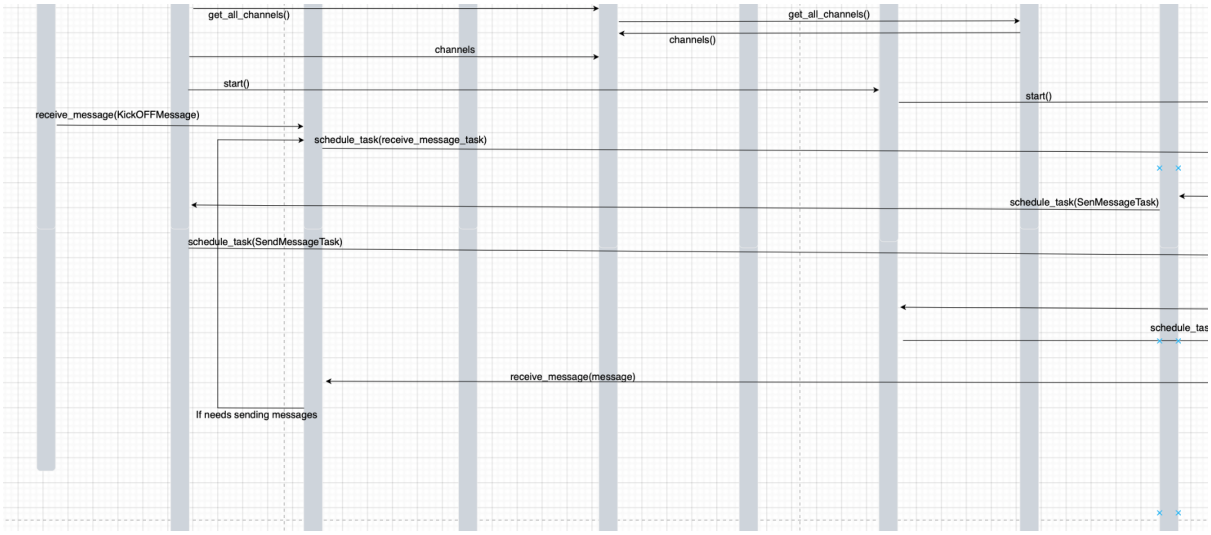

Рис 2.7. Сіквенс-діаграма “Start algorithm”. Лівий нижній кут.

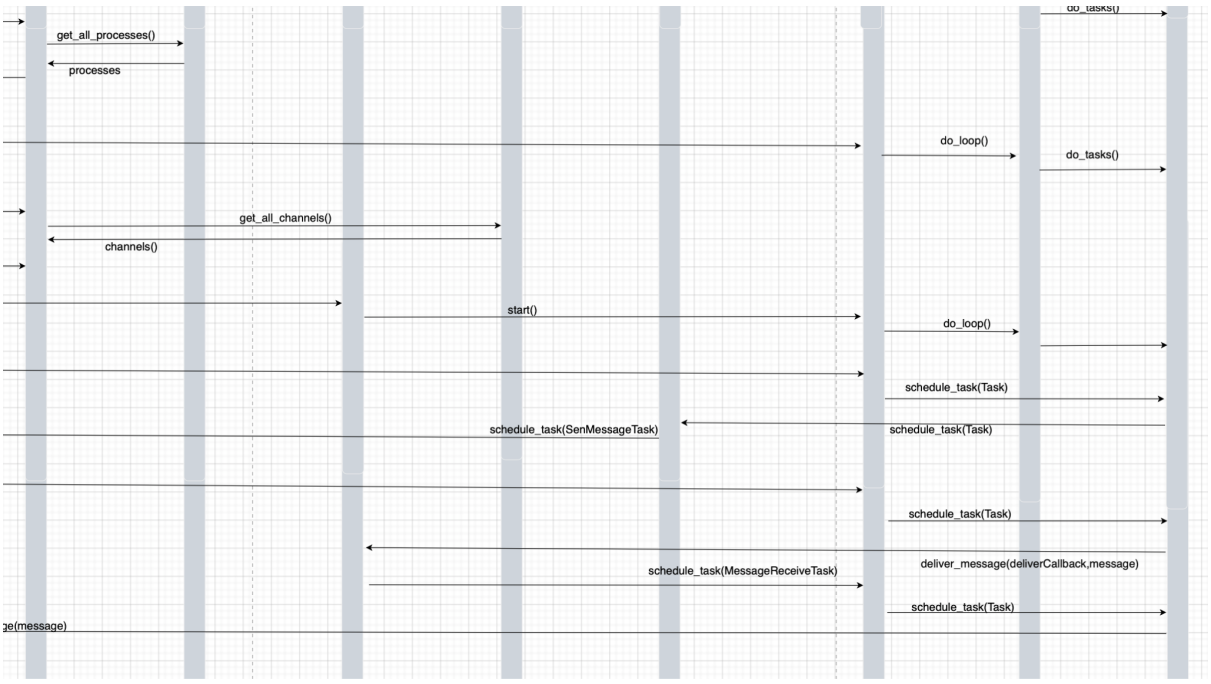

Рис 2.8. Сіквенс-діаграма “Start algorithm”. Правий нижній кут.

4. Прецедент “Pause/resume algorithm”. Через великий розмір діаграма розбита на дві частини: зліва направо.

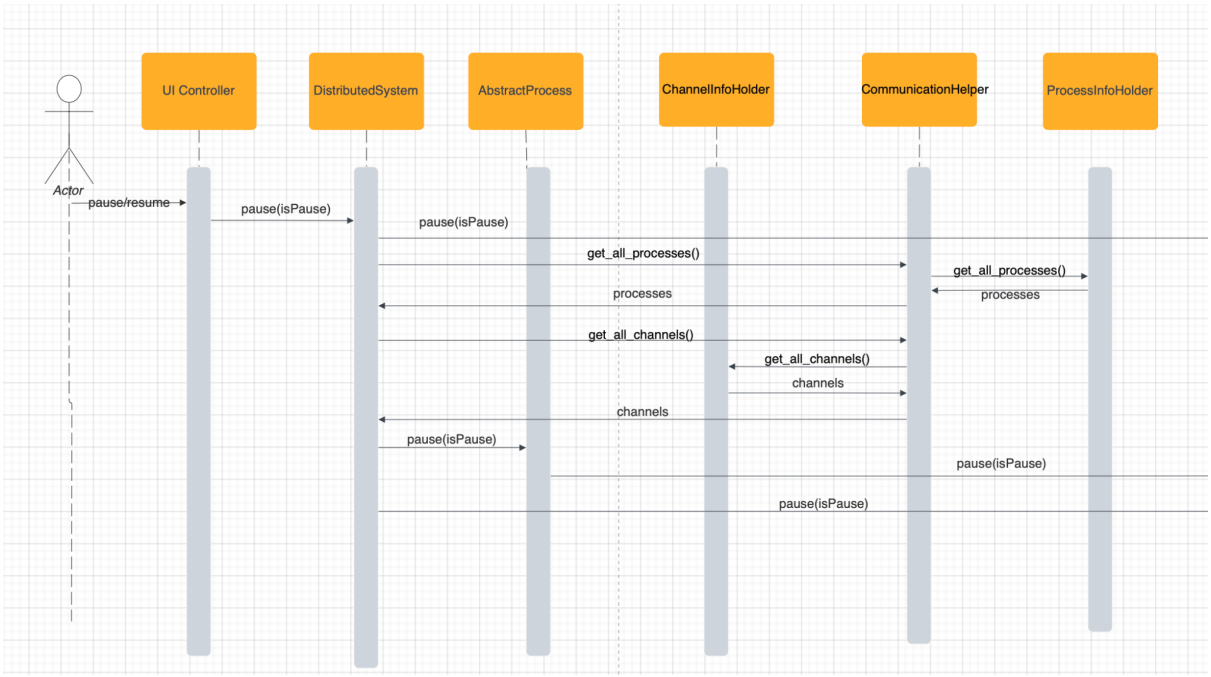

Рис 2.9. Сіквенс-діаграма “Stop/resume algorithm”. Лівий кут.

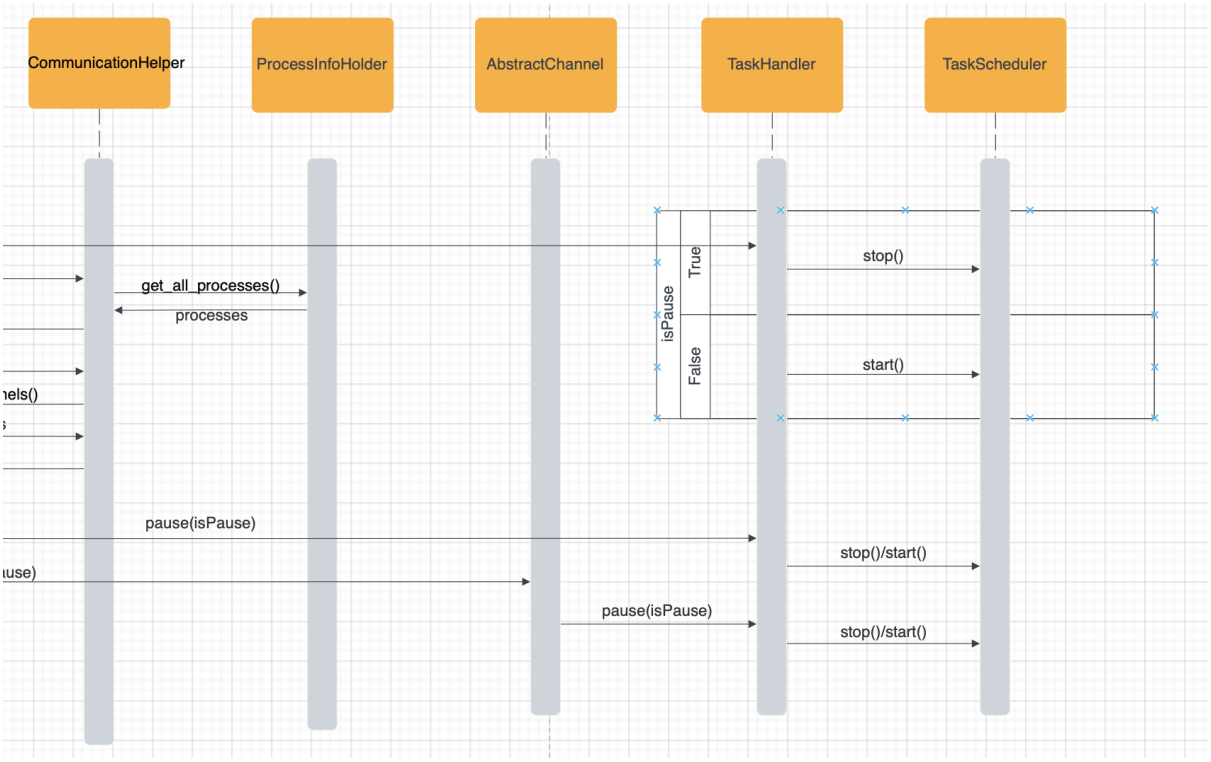

Рис 2.10. Сіквенс-діаграма “Stop/resume algorithm”. Правий кут.

На основі бізнес-прецедентів та сіквенс діаграм, була створена діаграма класів системи [28]

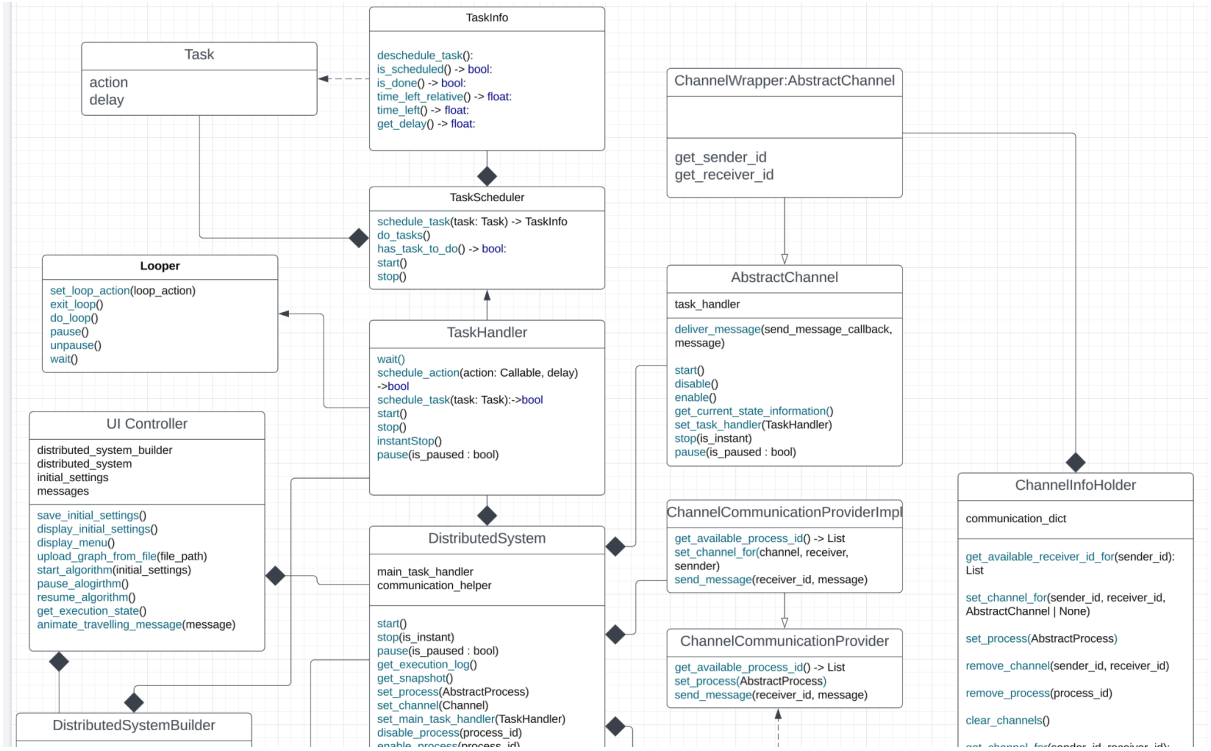

Рис 2.11.Діаграма класів. Верх.

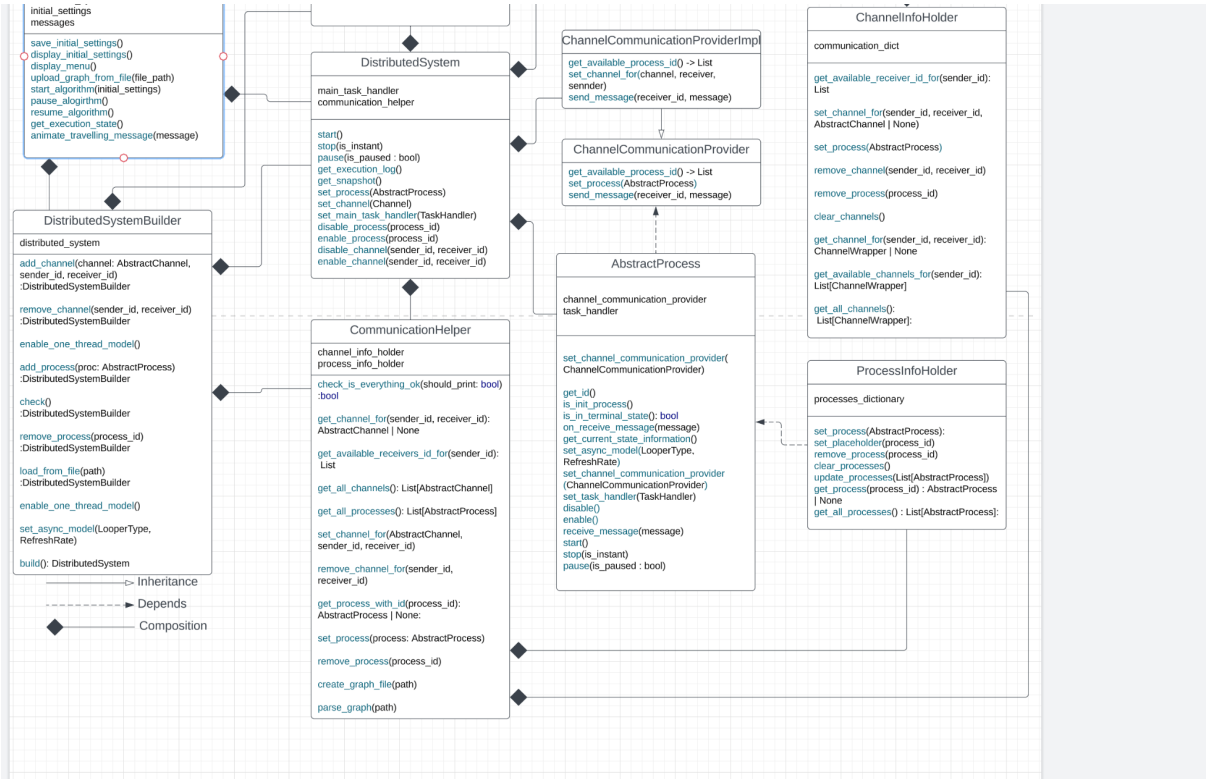

Рис 2.12.Діаграма класів. Низ.

Класс UI Controller забезпечує взаємодію між інтерфейсом користувача та програмною моделлю розподіленої системи через DistributedSystemBuilder, DistributedSystem. В його обов’язки також входить збереження налаштувань системи.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ТА ДОСЛІДЖЕННЯ ПРОТОТИПУ

3.1 Стороннє програмне забезпечення та вимоги модуля

Для написання модуля була обрана мова Python версії 3.10, ПЗ використовує нові можливості для специфікації типів, тому запуск на інтерпритаторах меншої версії не рекомендується [20].

Також використовуються наступні програмні інструменти: **PyQT5** - це набір Python-бібліотек для розробки графічних інтерфейсів (GUI) на основі бібліотеки Qt [21]. Qt є потужною і популярною багатоплатформеною бібліотекою для створення GUI. PyQT5 дозволяє розробникам створювати професійні інтерфейси користувача для десктопних застосунків на платформі Windows, macOS, Linux та інших операційних системах.

Серед причин вибору саме цієї бібліотеки є:

- Простота використання: PyQT5 має легкий синтаксис, зрозумілий для розробників Python, що дозволяє швидко створювати інтерфейси користувача.
- Багатоплатформеність: Завдяки використанню Qt, PyQT5 дозволяє розробляти крос-платформні десктопні застосунки, які можуть працювати на різних операційних системах без необхідності відновлення коду.
- Гнучкість: PyQT5 надає розробникам гнучкість в налаштуванні вигляду та поведінки віджетів, таких як кольори, шрифти, розміри, стилі, анімації, реакції на події та інше. Розробники можуть створювати власні кастомні віджети, розширювати функціонал вбудованих віджетів, а також використовувати сторонні розширення.

- Велика спільнота та документація: PyQT5 має велику активну спільноту розробників, яка забезпечує підтримку, відповідає на питання, надає приклади коду та розв'язання проблем. Крім того, документація PyQT5 є детальною та зрозумілою, що робить процес вивчення та використання бібліотеки досить легким.
- Підтримка подій та сигналів: PyQT5 має потужну систему подій та сигналів, що дозволяє розробникам реагувати на події віджетів та взаємодіяти з ними, що є особливо важливим при комунікації між різними потоками.

Також були використані модулі із набору стандартних інструментів python:

- **typing, enum**, що пропонує додаткову специфікацію типів, чим допомагає уникати багатьох типових помилок при розробці слабко типізованою мовою
- **random**, модуль для отримання випадкових даних. Дозволить моделювати випадкові події.
- **time,** цей модуль надає різноманітні функції, пов'язані з часом. Зручний при моделюванні у симуляціях в реальному часі.
- **threading** - стандартний інструментарій асинхронності [22]
- **multiprocessing** - інструментарій для створення нових інстансів програми на новому процесі (причини використання будуть пояснені у наступних пунктах) [22].

3.2 Пояснення щодо деталей імплементації програмної моделі

Усі процеси в моделі розподіленої системи повинні виконувати свою роботу паралельно або псевдо-паралельно. Для того, аби реалізувати симулювання асинхронного виконання задач був розроблений класс **TaskHandler**. Щоб правильно симулювати асинхронність **TaskHandler** використовує допоміжні класи **TaskScheduler** та **Looper.** Почнемо з того, що таке **Looper:** лупер є безкінечним потоком на якому періодично виконуються деякі задачі. В **Looper** є три реалізації, що базуються на різних способах досягнення асинхронного виконання:

1. ThreadLooper. Асинхронність досягається за рахунок стандартної реалізації багатопоточності python
2. QThreadLooper. Асинхронність досягається за рахунок стандартної реалізації багатопоточності в бібліотеці PyQt5
3. ProcessLooper. Асинхронність досягається за рахунок створення нових процесів на рівні системи.

Чому нам не достатньо однієї лиш реалізації через стандартні потоки мови пітон ми пояснимо в пункті 3.3 .

TaskScheduler відповідає за виконання завдань за деяким графіком або принципом, відповідно до конкретної імплементації TaskScheduler’а. Користувач може додавати завдання до виконання в TaskScheduler’а та керувати його станом (завершити роботу, поставити на паузу і.т.д).

Наразі є дві реалізації **TaskScheduler:**

1. TimingTaskScheduler - симулює виконання роботи в реальному часі, (із реальними затримками). Ця імпелементація зручна для відображення подій симуляції в графічному інтерфейсі.
2. TaskScheduler - симулює виконання роботи, зберігаючи тільки послідовність виконання кожної задачі (ігнорує затримки між виконанням).

В своїй суті TaskHandler є медіатором між **TaskScheduler** та **Looper**, що дозволяє користувачам через простий інтерфейс виконувати задачі асинхронно.

Класс **AbstractProcess** - базовий клас для моделювання поведінки процесу в розподіленій системі. Кожен процес має імплементувати розподілений алгоритм в методі `on_receive_message(message)`. Користувачу програмного забезпечення пропонується використовувати готові процеси з реалізацією набору розподілених алгоритмів, або розробити свій процесс в методі `on_receive_message()`. Кожен процес вже є асинхронним і всі свої обчислення проводить на окремому потоці.

Канали передачі повідомлень - важлива складова системи, і їх програмна модель повинна давати максимум можливостей для симуляції достовірної поведінки. **AbstractChannel** являє собою базовий клас для симуляції каналу. Кожен екземпляр класу працює асинхронно і незалежно від всіх інших компонентів системи. **SimpleDelayChannel** - проста імплементація яка симулює передачу повідомлення з випадковою затримкою. Користувачі можуть також розробити свою реалізацію **AbstractChannel** із біль специфічною поведінкою, як от втрата повідомлення, його зміна тощо.

Головною частиною моделі розподіленої системи є клас **DistributedSystem**. В його задачі входить сама симуляція роботи системи, координація взаємодії між всіма іншими частинами розподіленої системи.

Граф розподіленої системи задається на етапі побудови інстансу **DistributedSystem**, у цьому нам допоможе **DistributedSystemBuilder**, у якому є встановлюється список доступних процесів, каналів передачі повідомлень.

3.3 Проблема GIL - multithreading vs multiprocessing. Апробація на алгоритмах.

Для інтеграції симуляції моделі розподілених обчислень в графічний інтерфейс, симуляція повинна проходити на окремому потоці, аби програма була завжди доступною для користувача. Мова пайтон має свої особливості в реалізація багатопоточності, що спричинило проблеми на етапі інтеграції програмної моделі.

GIL (Global Interpreter Lock) - це мьютекс, який дозволяє використовувати інтерпритатор пітону одночасно тільки одному потоку. Це означає, що незважаючи на використання інструментів багатопоточності, аплікації тільки один поток може виконувати команди. Така реалізація багатопоточності є вузьким місцем для ефективного виконання асинхронних програм.

Незважаючи на ці проблеми Python досі використовує GIL. Хоча GIL не є складовою частиною абстрактного Python, цей компонент найпоширенішої реалізації Python - CPython, яка підтримується Фондом розробки Python. Механізми керування пам'яттю в CPython не є потокобезпечними. Python використовує reference counting для керування пам'яттю. Це означає, що об'єкти, створені в Python, мають reference count, яка відстежує кількість посилань, що вказують на об'єкт. Коли цей лічильник досягає нуля, пам'ять, зайнята об'єктом, вивільняється. Саме тому GIL використовується для послідовного доступу до об'єктів та пам'яті, щоб запобігти гонкам потоків. Якби в CPython не було GIL, він мусив би обробляти конкуренцію та гонки потоків іншим способом.

Деякі мови уникають необхідності в наявності GIL для безпечного менеджменту пам’ятью в багатопотоковому середовищі за допомогою інших підходів, як garbage collection. З іншого боку, це

означає, що цим мовам часто доводиться компенсувати втрату переваг однопотокової продуктивності, яку надає GIL, шляхом додавання інших можливостей для покращення продуктивності, таких як компілятори Just-In-Time (JIT).

Наразі єдина перешкода до видалення GIL - це зворотна сумісність з існуючими розширеннями на C, які сильно спираються обмеження, що надає GIL. Треба відзначити, що існує декілька реалізацій пітон без GIL.

Через це, за поєднання розподіленої системи написаною мовою пайтон, графічний інтерфейс буде суттєво тормозити. Тому треба знайти рішення для уникнення цих проблем.

Наразі є 2 способи досягти реальної багатопоточності на базі пітону:

- **Альтернативні інтерпретатори**. Python має кілька реалізацій інтерпретаторів. CPython, Jython, IronPython та PyPy, написані в мовах програмування C, Java, C# та Python відповідно, є найпопулярнішими [24][25]. GIL існує лише в оригінальній реалізації Python - CPython. Проте вони можуть бути несумісними з усіма бібліотеками використаними в проекті.
- **Multiprocessing**. Найпопулярніший спосіб - використання багатопроцесорного підходу, де використовується кілька процесів системного рівня замість потоків. Кожен процес Python отримує свій власний інтерпретатор Python та простір пам'яті, тому проблеми з GIL не виникають. Модуль multiprocessing дозволяє легко створювати процеси. Проте через використання нових системних процесів комунікація між ними сильно ускладнюється, у порівнянні з стандартними threading.

Через те що в нашій моделі більшість часу сторонній потік використовується для симуляції затримок, а не виконання важких обчислень, було прийнято рішення виконувати необхідні дії тільки через певні проміжки часу. Незважаючи на це, в програмному модулі були реалізований в тому числі multiprocessing підхід.

3.4 Тестування прототипу

В даному розділі описується тестування ПЗ [26]. Це робиться для того, щоби переконатися в працездатності модуля та продемонструвати його работу. Тестування будет проходити за наступними етапами:

- Перевірка візуалізації топології мережі на декількох прикладах
- Перевірка роботи симуляції роботи розподіленої системи на основі відомого алгоритму eха.

Алгоритм ехо є централізованим хвильовим алгоритмом для ненаправлених мереж. Він лежить в основі кількох розподілених алгоритмів [27]. Принцип роботи алгоритму такий: ініціатор починає відправляти повідомлення всім своїм сусідам. Ці повідомлення поширюються у всіх напрямках та відскакують від “кутів” мережі до ініціатора. Це досягається таким чином: коли процес не ініціатор отримує повідомлення вперше, він зберігає відправника як parent і відправляє повідомлення всім сусідам, крім нього. Коли процес отримав повідомлення від усіх своїх сусідів, він відправляє повідомлення своєму parent. Нарешті, коли ініціатор отримав повідомлення від усіх своїх сусідів, він приймає рішення.

Етап візуалізації топології мережі. Під час цього етапу ми будемо зчитувати з файлів топологію мережі у вигляді матриці суміжності та візуалізувати її.

На рисунку 3.1 ми бачимо, що існуючі канали в графі відповідають матриці суміжності з файлу: число процесів 6, кожний процес поєднаний один з одним.

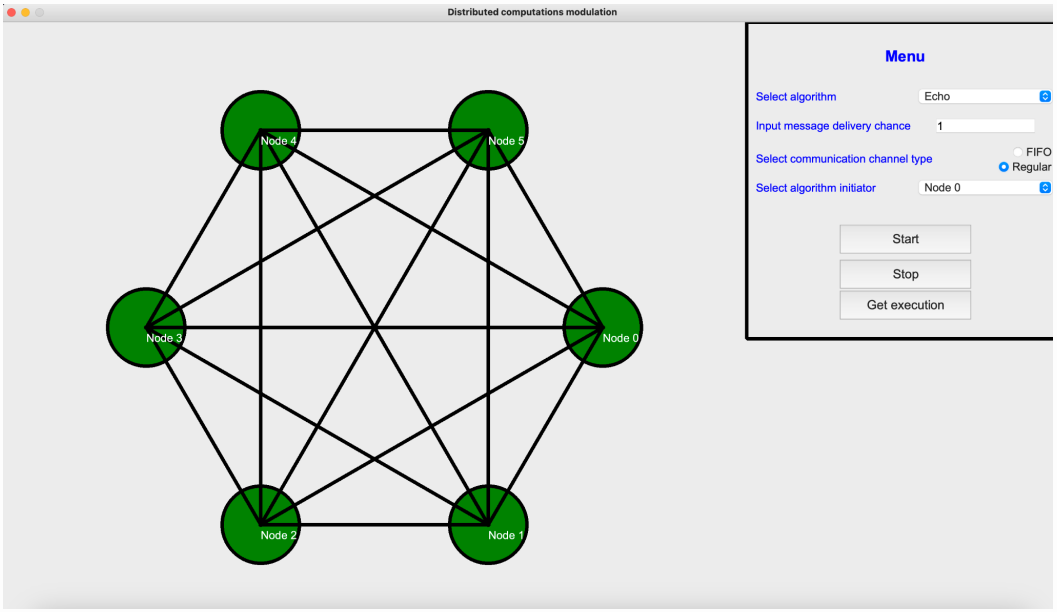

Рис 3.1. Розподілена система з шістьма процесами

На рисунку 3.2 також продемонстрована справна робота модуля: число процесів 3, граф без каналу між нульовим процесом і другим.

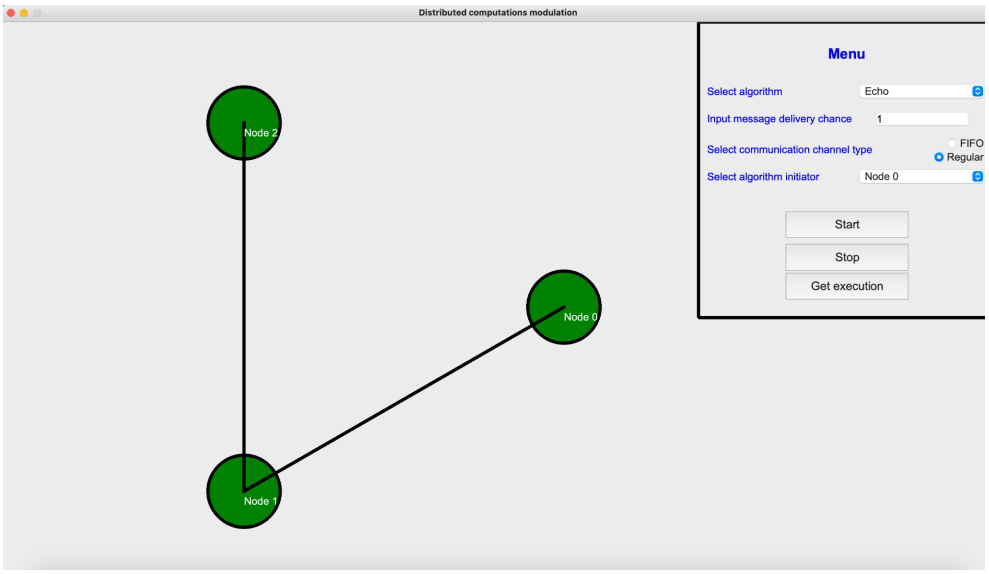

Рис 3.2. Розподілена система з трьома процесами

Етап перевірки роботи симуляції розподіленої системи на алгоритмі exa. В першому випадку тестування проходить на мережі відповідній повному графу з трьох вершин.

На скріншоті 3.3 ми можемо побачити візуалізацію мережі, обраний розподілений алгоритм: exo-алгоритм; процес ініціатор- Node0 та налаштування каналів передачі повідомлень.

Рис 3.3. Налаштування системи перед виконанням алгоритму

На зображеннях 3.4, 3.5, 3.6 маємо змогу спостерігати етапи роботи програми, історію передачі повідомлень всередині системи та стан процесів в залежності від отриманих повідомлень. Тепер нескладно

помітити, що процеси змінюють стани відповідно до алгоритму, тобто симуляция на цьому прикладі є успішною.

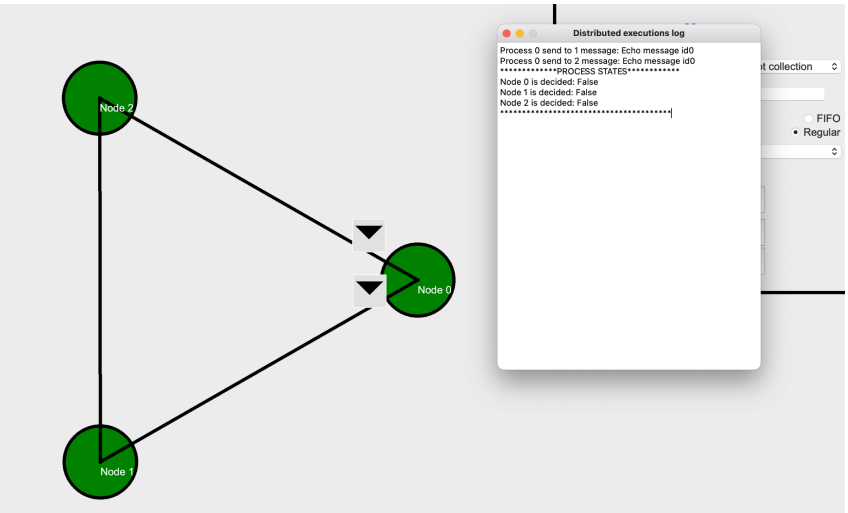

Рис 3.4. Початок виконання алгоритму exa на системі з 3-ма процесами

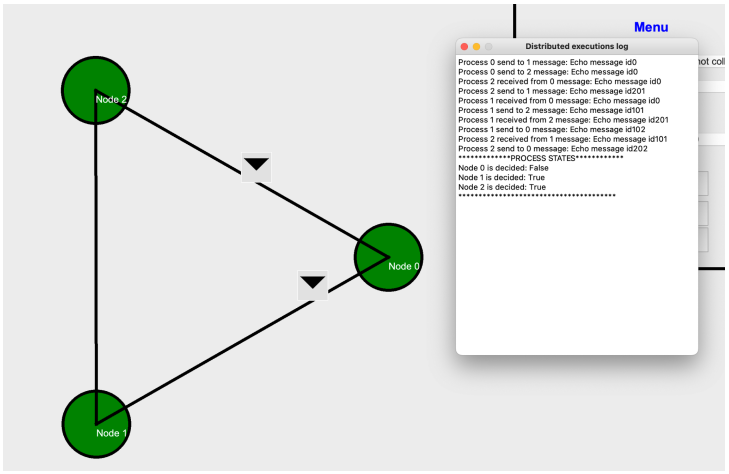

Рис 3.5. Середина виконання алгоритму exa на системі з 3-ма процесами

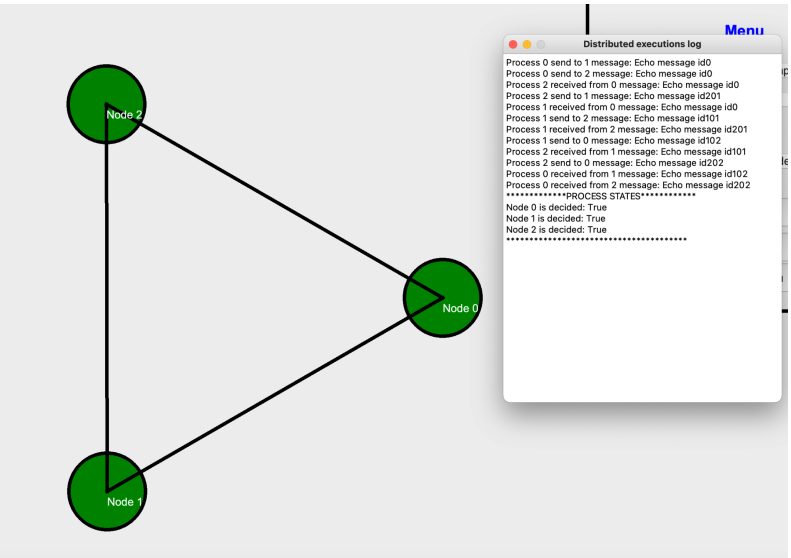

Рис 3.6. Завершення виконання алгоритму на системі з 3-ма процесами

Тепер перевіримо роботу в випадку мережі з 6 процесів.На скріншотах 3.7, 3.8 відображено стан системи на різних етапах виконання.

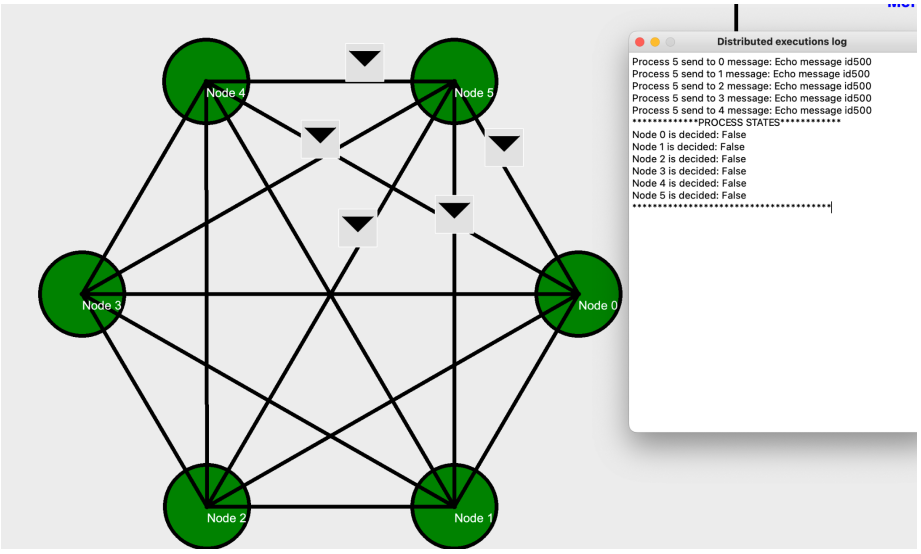

Рис 3.7. Початок виконання алгоритму на системі з 6-ма процесами

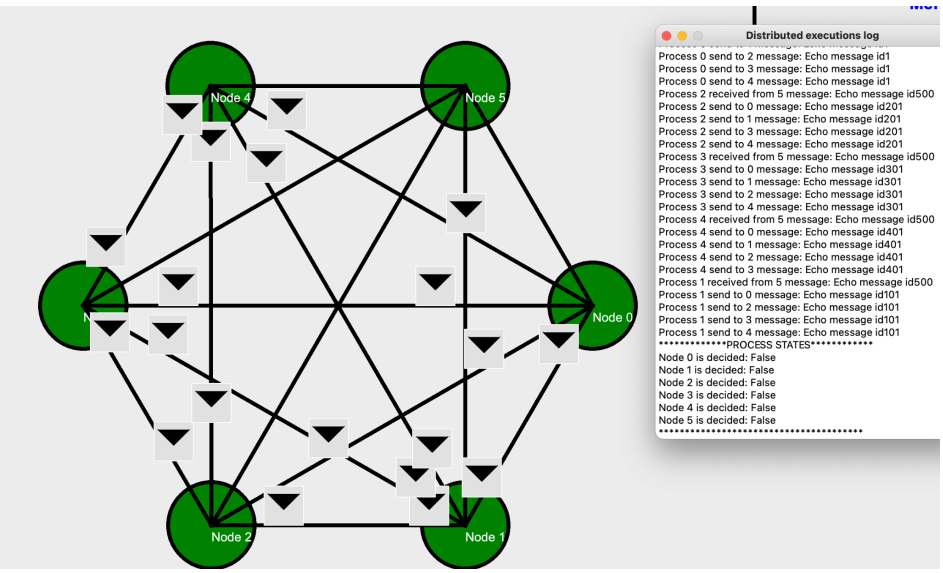

Рис 3.8. Система посеред виконання алгоритму exa

Слідкуючи за роботою системи на рисунку 3.9 та попередніх зображень можно сказати, що в даному випадку модуль також досяг очікуваних результатів.

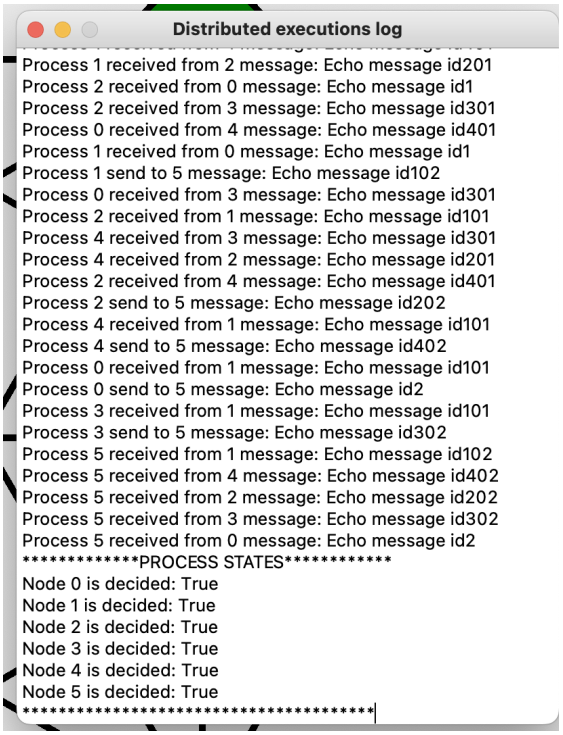

Рис 3.9. Події під час виконання розподіленої системи. Кінцеві стани процесів

Підсумовуючи результати тестування, можна сказати, що ядро симуляції розподіленої системи працює справно та згідно з очікуваннями, графічний інтерфейс ПЗ теж виконує свою задачу.

ВИСНОВКИ

Нами був розроблений програмний засіб для динамічного аналізу роботи розподілених алгоритмів. Всередині була реалізована програмна модель розподіленої системи, яка забезпечує інструментами для моделювання широкого спектру розподілених систем з різними початковими обмеженнями, що надає можливість перевірки та імплементації майже всіх наявних розподілених алгоритмів. Імплементована програмна модель каналів дозволяє створювати канали з різними характеристиками:

- з/без FIFO обмеженням
- з/без ймовірності загубити або пошкодити повідомлення,
- з/без випадкової затримки
- Можливість відключати канал за вимогою.

Програмна модель процесів також пропонує необхідний інструментарій для втілення широкого спектру розподілених алгоритмів.

На сам перед ядро програмної моделі надає зовнішнім користувачам вичерпний набір функцій для керування та відстежування роботи системи та її різних компонентів. Серед таких функцій є:

- Зупинка та поновлення роботи системних складових, зокрема каналів передачі повідомлень та процесів
- Збір поточних станів каналів, процесів та історію подій в системі
- Відключення/ввімкнення “на ходу” процесів і каналів
- Внесення змін до конфігурації системи (нові канали або процеси)

Також важливою частиною написаної програми є графічний інтерфейс, що робить роботу з розподіленою системою доступною, навіть, для користувачів загалом мало-знайомих з темою розподілених систем та алгоритмів. Програмний модуль створює в реальному часі графічну інтерпретацію роботи розподіленої системи.

Робота програми була протестована на загально-відомому ехо-алгоритмі.

У подальшому можно доповнити функціонал модулю статичним аналізом.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Hafner, Katie, and Matthew Lyon. *Where wizards stay up late the origins of the internet*. New York: Simon & Schuster Paperbacks, 2006.
2. Everdell, William R. The end of kings: A history of republics and Republicans. University of Chicago Press, 2000.
3. Burns, Brendan. *Designing distributed systems: Patterns and paradigms for scalable, Reliable Services*. Sebastopol, CA: O'Reilly Media, 2018.
4. Moore, G.E. "Cramming More Components onto Integrated Circuits." *Proceedings of the IEEE* 86, no. 1 (1998): 82–85. https://doi.org/10.1109/jproc.1998.658762.
5. Ghosh, Sukumar. *Distributed systems: An algorithmic approach*. Boca Raton: Chapman & Hall/CRC, 2007.
6. Tel, G. Introduction to Distributed Algorithms. 2nd edition. Cambridge University Press. 2012. https://doi.org/10.1017/CBO9781139168724
7. Lynch, N. A. Distributed Algorithms. Publisher: Morgan Kaufmann Publishers Inc., San Francisco, United States. 1996. 904 p.
8. Nielson, F., Nielson, H. R., & Hankin, C. (2015). Principles of program analysis. Springer.
9. Wichmann, B. A.; Canning, A. A.; Clutterbuck, D. L.; Winsbarrow, L. A.; Ward, N. J.; Marsh, D. W. R. (Mar 1995). "Industrial Perspective on Static Analysis" (PDF). *Software Engineering Journal*. 10 (2): 69–75.
10. "What Is Dynamic Analysis?" TotalView by Perforce. Режим доступу: https://totalview.io/blog/what-dynamic-analysis (дата звернення: 19.03.2023).

11. Bertot, Yves, and Pierre Castéran. *Interactive theorem proving and program development: Coq'art: The Calculus of Inductive Constructions*. Berlin: Springer, 2004.

12. Huizinga, Dorota, and Adam Kolawa. *Automated defect prevention: Best practices in software management*. Hoboken, NJ: Wiley-Interscience, 2007.

13. Altvater, Alexandra. "What Is Code Profiling? Learn the 3 Types of Code Profilers." Stackify, May 1, 2023. https://stackify.com/what-is-code-profiling/ .

14. "What Is Debugging?" Amazon. Режим доступу: https://aws.amazon.com/what-is/debugging/ (дата звернення: 19.03.2023). Назва з екрану.

15. Lang, Niklas. "Database Basics: Acid Transactions." Medium, October 24, 2022. https://towardsdatascience.com/database-basics-acid-transactions-bf4d38bd8e26

16. Erl, Thomas. *Service-oriented architecture: Concepts, technology, and Design*. Upper Saddle River etc.: Prentice Hall Professional Technical Reference, 2011.

17. Martin, Robert C. *Clean architecture a craftsman's guide to software structure and Design*. Boston: Prentice Hall, 2018.

18. Martin, Robert C. Agile Software Development: Principles, patterns, and practices. Pearson Education, 2003.

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

20. "Python Release Python 3.10.0." Python.org Режим доступу: https://www.python.org/downloads/release/python-3100/ (дата звернення: 19.03.2023). Назва з екрану.

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

22. Gorelick, Micha, and Ian Ozsvald. *High performance python: Practical performance programming for humans*. Beijing: O'Reilly, 2020.

23. Understanding the python gil - dabeaz. http://dabeaz.com/python/UnderstandingGIL.pdf (дата звернення: 19.03.2023). Назва з екрану.

24. "Python 3.11.3 Documentation." 3.11.3 Documentation. https://docs.python.org/ (дата звернення: 19.03.2023). Назва з екрану.

25. What's Jpython? Режим доступу: https://www.jython.org/ (дата звернення: 19.03.2023). Назва з екрану.

26. Craig, Rick David, and Stefan P. Jaskiel. *Systematic software testing*. Boston: Artech House, 2007.

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

28. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1. , Object Management Group , Object Management Group (2011), p. 507.

29. What is Use Case Diagram? Режим доступу: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/ (дата звернення: 19.03.2023). Назва з екрану.