

АНОТАЦІЯ

Розроблено сервіс виявлення вузлів мережі за допомогою мови C# у Visual Studio 2022 .NET Framework. Проект, що надає змогу виявляти комп'ютери локальної мережі за використанням оптимізованого пулу TCP-з'єднань та групової розсилки UDP-повідомлень. Також він застосовує протокол Kademlia, що дає можливість оновлювати зібрані дані та швидко й безпечно скачувати файли з набору вузлів.

Для представлення роботи цього сервісу, його було інтегровано з сервісом обміну файлами Light Upon Cloud, реалізовано консольний і Docker-compose проекти. В застосунку обмінювання файлами було виправлено його інсталяцію, а також додані коректування до синхронізації, логінування й потокобезпечності.

ANNOTATION

It was developed a Discovery Service in Visual Studio 2022 .NET Framework that allows clients to define local network nodes using an optimized pool of TCP connections and multicast of UDP messages. This project uses the Kademlia protocol to update collected data, and quickly and safely download files from set of nodes.

To present the work of this service, it was integrated with the Light Upon Cloud file sharing service, console and Docker-compose projects were implemented. An installation of the file exchange application was fixed and also added adjustments to a file synchronization, logging in and thread safety.

ЗМІСТ

Вступ.....	5
Розділ I. Огляд вимог, протоколів і технологій розробки сервісу	7
1.1. Вибір сервісу обміну файлами. Вимоги до сервісу виявлення	7
1.2. Вибір платформи сервісу й середовища розробки	8
1.3. Мережеві протоколи	8
1.3.1. IP	10
1.3.2. TCP	11
1.3.3. UDP.....	15
1.3.4. Протокол оновлення та пошуку інформації в мережі.....	19
1.4. Опис конкурентних конструкцій та шаблонів мови C#.....	24
1.4.1. Захоплення контексту потоку.....	24
1.4.2. Оновлення значення однієї змінної кількома потоками.....	27
1.4.3. Асинхронний шаблон на основі завдань	29
1.4.4. Модель асинхронного програмування	29
1.4.5. Постачальник/Споживач шаблон.....	29
1.5. Вибір пакетів для розробки сервісу	31
1.6. Вибір сервісу для емуляції кількох вузлів на одному пристрої.....	34
Розділ II. Фізична структура сервісу виявлення	35
Розділ III. Реалізація сервісу виявлення	42
3.1. Конфігурації проекту.....	42
3.2. Схема класів	44

3.3. Ініціалізація сервісу й налаштування брандмауера.....	49
3.4. Журналювання роботи сервісу	50
3.5. Запуск сервісу виявлення.....	51
3.5.1. Фільтрація мережевих інтерфейсів та IP-адрес	52
3.5.2. Відправка групової розсилки.....	56
3.6. Слухання TCP-з'єднань й отримання серверних повідомлень	58
3.7. Реалізація обробки повідомлень.....	61
3.8. Алгоритм виявлення вузлів	65
3.9. Протокол оновлення та пошуку інформації в мережі.....	66
3.10. Пул клієнтських TCP-з'єднань	70
3.10.1. Особливості реалізації.....	71
3.10.2. Взяття сокету з пулу.....	75
3.10.3. Повернення сокету в пул.....	77
3.10.4. Фонове відновлення з'єднань	80
3.11. Скачування файлів з локальної мережі	82
3.12. Зупинка сервісу виявлення	86
Розділ IV. Інтеграція сервісу виявлення з сервісом обміну файлами	88
4.1. Демонстрація роботи сервісу обміну файлами до інтеграції сервісу виявлення	88
4.2. Виправлення сервісу обміну файлами.....	93
4.3. Фасад сервісу виявлення, його запуск.....	96
4.4. Інтеграція скачування з локальної мережі	98
4.5. Відміна скачування	104

4.6. Зупинка сервісу виявлення в сервісі обміну файлами	105
Розділ V. Тестування сервісу виявлення й інтеграції	106
5.1. Модульні тести.....	106
5.2. Функціональні тести.....	110
5.3. Відомі проблеми сервісу й способи його покращення	125
Висновки	127
Джерела	130

ВСТУП

- Мета роботи

Розробити оптимізований й водночас безпечний сервіс виявлення й скачування файлів з локальної мережі (далі – сервіс виявлення) й інтегрувати його у вибраний додаток обміну файлами. Ознайомитись з особливостями реалізації бізнес-проектів, для яких важлива швидкість виконання функцій та які націлені на безпечну роботу в локальних мережах.

- Актуальність роботи

Вибраний додаток для інтеграції є досить дешевим, порівняно з іншими, й у в майбутньому буде у відкритому доступі, що відокремлює його від аналогів й в результаті буде знахідкою для клієнтів й розробників. Цей документ надасть їм змогу отримати детальні відомості про конкурентне програмування й розробку сервісу виявлення.

- Теоретико-методологічна основа розробки

Проблематика розробки оптимізованих додатків у світі програмування обговорюється досить активно, зокрема Стівеном Клірі [1], Джозефом та Беном Албахарі [2]. Тема виявлення, збору та оновлення інформації про вузли мережі детально розглянута такими фахівцями, як Петар Маймунков [3], Давид Мазьєр [3], Марком Кліфтоном [4] та іншими [5].

- Об'єкт розробки

Розробка оптимізованого та безпечного сервісу виявлення вузлів локальної мережі та його інтеграція з додатком обміну файлами.

- Предмет розробки

Особливості реалізації та безпечного виявлення вузлів локальної мережі, оновлення про них інформації та скачування з них файлів. Визначення деталей інтеграції з сервісом обміну файлами.

- Гіпотеза

Сервіс виявлення здатен визначати вузли локальної мережі, навіть за відсутності Інтернету, оптимізовано оновлювати про них інформацію та скачувати з них файли. Це повинно зменшити навантаження на сервер додатку обміну файлами.

- Завдання

До виконання в межах розробки були поставлені наступні завдання:

- 1) розробити безпечне й оптимізоване виявлення вузлів мережі;
- 2) реалізувати періодичне оновлення інформації про них;
- 3) додати функціональність оптимізованого скачування з них файлів;
- 4) розробити проект, що надаватиме змогу тестувати сервіс виявлення при різноманітній кількості вузлів мережі, що його використовують;
- 5) інтегрувати розроблений сервіс в додаток обміну файлами;
- 6) додати правки в останньому до інсталяції, синхронізації файлів й папок, логінування та потокобезпечності.

- Структура роботи

Спершу розглядаються теоретичні відомості та пакети для подальшого кращого розуміння реалізації та інтеграції сервісу виявлення. Після цього представляється фізична структура даного проекту, що надає змогу зрозуміти схему класів, що представляється в розділі розробки цієї бібліотеки, де описані найважливіші відповідні пункти. Далі описаний розділ сервісу виявлення інтеграції з додатком обміну файлами, де спершу показується найважливіші функції останнього, приклад його роботи та які конкретні пункти були виправлені мною. Після цього описано реалізація інтеграції. В наступному розділі «Тестування сервісу виявлення й інтеграції» представлено створення проекту Docker-compose [6], модульні й функціональні тести, а також висновки після аналізу тестів й коду програми.

РОЗДІЛ І. ОГЛЯД ВИМОГ, ПРОТОКОЛІВ І ТЕХНОЛОГІЙ РОЗРОБКИ СЕРВІСУ

Оглянути усі класи розробленого сервісу виявлення або ж запропонувати зміни, як це робили інші GitHub-користувачі, можна за допомогою наступного репозиторію [70].

1.1. Вибір сервісу обміну файлами. Вимоги до сервісу виявлення

Сервіс обміну файлами (його назва - Light Upon Cloud (LUC)) [7], куди інтегрується сервіс виявлення (Discovery Service (DS)), насправді є першочерговим проектом, а сервіс скачування з локальної мережі - лише його частиною. Тому вибору, як такого, між сервісами обміну файлами й не було. Розглянемо спершу особливості й вимоги додатку обміну файлами до DS перш, ніж заглибитись детальніше в останній.

LUC є бізнес-проектом й призначений для використання в різноманітних офісах, працівники якого достатньо навантажують центральний процесор та використовують пам'ять.

З огляду на вище сказане, були створені такі вимоги до мого сервісу:

- 1) ефективно виявляти вузли мережі, тобто коли в цьому є потреба, й робити це безпечно та швидко;
- 2) взаємодіяти в один і той самий час між парою вузлів мінімальною кількістю потоків;
- 3) безпечно обмінюватись повідомленнями. Тобто вони повинні проходити ряд перевірок, перед тим, як формувати на них відповідь;
- 4) оптимально оновлювати інформацію про вузли мережі;
- 5) з високою швидкістю скачувати файли та зберігати їх цілісність.

Також серед наявних сервісів виявлення на GitHub [8], не було помічено, що хоча б один з них використовує пул TCP-з'єднань, що задовільняє вимоги пункту номер 2, або ж підтримує скачування файлів в такому вигляді, як це

потрібно додатку LUC (використання ADS [9] й містимість в запитах версії файлу, префіксу й групи (детальніше див. підрозділ «Інтеграція скачування з локальної мережі»)).

1.2. Вибір платформи сервісу й середовища розробки

Оскільки основним напрямком мого професійного розвитку є мова C# [10] й платформа .NET [11], які зараз активно розвиваються й представляють усі необхідні можливості для створення додатків, які можуть взаємодіяти по мережі й використовувати різні відповідні протоколи. Найпопулярнішим й найзручнішим середовищем розробки для них є Visual Studio (VS) 2022 [12]. Через вище вказані причини було вирішено використовувати саме їх.

Оскільки вибраний сервіс обміну файлами фокусується лише на операційній системі Windows, то було вибрано для першої версії DS платформу .NET Framework версії 4.8 [13], тобто найновішу. Саме цю версію також використовує LUC. Звісно, завжди можна оновити ці сервіси до найновішої версії .NET, тобто 6.0 станом на сьогоднішній день, але це може привести до нових проблем (наприклад, відсутність підтримки встановлених пакетів), тому дані проекти, для початку, розроблялись саме для Windows.

1.3. Мережеві протоколи

Мережевий протокол – набір правил й домовленостей щодо зв'язку між мережевими пристроями, включаючи способи, якими обладнання можуть ідентифікувати та встановлювати зв'язки між собою [22]. Існують протоколи, які також включають підтвердження надісланого пакету та стиснення даних для високопродуктивного та надійного мережевого зв'язку.

Попередньо варто розглянути такі поняття для кращого розуміння опису протоколів:

- 1) OSI (Open Systems Interconnection - взаємодія відкритих систем) - семирівнева модель, кожен рівень якої виконує певні функції [34].

Кожен нижчий рівень цієї моделі надає вищому деякі послуги для транспортування інформації. При цьому важливою є взаємодія однакових рівнів відправника й отримувача, яка забезпечує передавання даних між кінцевими точками.

- 2) Вузол мережі - будь-який фізичний пристрій в мережі, що здатен надсилати, отримувати та/або передавати інформацію [28]. ПК - найбільш загальний вузол і його часто називають Інтернет-вузол або комп'ютерний вузол.
- 3) Кінцева точка (endpoint) – будь-який пристрій, що під'єднаний до комп'ютерної мережі [35]. В той же час вузол мережі може мати кілька кінцевих точок. Наприклад, якщо інтернет-провайдер й операційна система (ОС) пристрою підтримують і IPv4, і IPv6 (див. нижче підпункт «IP»), то він має змогу обмінюватись даними за допомогою цих 2-х кінцевих точок. DS підтримує обидві версії протоколів.
- 4) DNS (Domain Name System) – система доменних імен для перетворення імені хоста в IP-адресу [26].
- 5) Хост – пристрій, що під'єднаний до мережі [36]. На відміну від кінцевої точки, одне обладнання не може мати кілька хостів.
- 6) Сокет – програмний інтерфейс для забезпечення обміну даних між процесами [24]. Опирається на вбудовані можливості операційної системи. Мережеві протоколи використовують його для комунікації в мережі.
- 7) Vlinker - діагностичний сканер, бортовий комп'ютер, реєстратор і монітор продуктивності в реальному часі.
- 8) MAC-адреса (Media Access Control — управління доступом до посередників) – ідентифікатор, що присвоюється кожній одиниці

мережевого обладнання і надає змогу єдиним чином визначати кожну точку підключення і доставляти дані для правильної їх передачі й надання послуг [14].

- 9) Октет – це вісім двійкових цифр (шістнадцятковий формат) або десяткове число в діапазоні 0-255 [16].

Були використані, наступні мережеві протоколи, оскільки саме вони надають можливість у локальних мережах виявляти вузли (UDP) або ж безпечно передавати дані (TCP).

1.3.1. IP

IP (Internet Protocol – Інтернет протокол) - набір правил для кожної сторони обміну даними для забезпечення відповідних потоків [24]. IP-адреса - унікальний ідентифікатор мережевого рівня, схожий на поштову адресу, що пов'язана з активністю вузла (користувача) Інтернету [27]. Існує 2 версії цього протоколу:

- 1) IP версії 4 (IPv4) – IP, що використовує 32-бітні(4 байтні) адреси, які обмежують адресний простір близько 4 мільярдів (4 294 967 296) доступними унікальними адресами [37]. Це перша версія протоколу, що набула досить широкого розповсюдження й до сих пір активно використовується.
- 2) IP версії 6 (IPv6) – нова версія IP, що застосовує 128 бітні (16 байтні адреси), в якій [38]:
 - автоконфігурація,
 - покращене з'єднання в мережах P2P (peer-to-peer - усі вузли можуть бути як серверами, так і клієнтами),
 - вища швидкість,
 - підвищена ефективність маршрутизації.

Незважаючи на те, що її розробка розпочалася ще 1998 року, вона почала замінювати протокол IPv4 лише в 2017 році [38].

1.3.2. TCP

TCP (Transmission Control Protocol – протокол керування передачею) – протокол, орієнтований на роботу з підключеннями; передає дані у вигляді потоків байтів [17]. Вони, тобто дані, передаються пакетами (TCP-сегментами), що в свою чергу складаються з набору даних й заголовків TCP. Цей протокол вважається надійним, оскільки в ньому застосовується відправлення підтверджень, щоб гарантувати, що прийняті дані не є зміненими після їх надсилання. Також він використовує контрольні суми розміром від 128-ми до 143-х байт.

Контрольна сума - певне значення, обчислене на основі деякого набору даних зі застосуванням деякого алгоритму, що використовується для визначення цілісності даних при їх збереженні чи передачі [18].

Ця сума використовується як раз для перевірки достовірності сегментів обчислюється на основі всього TCP-сегменту включно зі заголовком та важливих полів IP-пакету, а саме:

- IP-адреса хостів відправника й отримувача;
- загального розміру IP-пакету;
- номера протоколу (для TCP –6).

Варто відмітити, що TCP разом з протоколом IP вважаються стрижневими для мережі Інтернет й разом складають відому модель TCP/IP [20]. Процес встановлення зв'язку між пристроями в мережі Інтернет відбувається відповідно до цієї моделі (урізана версія абстрактної мережевої моделі для комунікацій і розробки мережевих протоколів OSI). Прикладний рівень - це верхня частина стека моделей TCP/IP, звідки мережеві програми, такі як веб-браузери на стороні клієнта, встановлюють з'єднання із сервером.

Модель OSI		
Дані	Рівень	
Дані	п.о.р 7. Прикладний доступ до мережевих служб	[показати]
Дані	п.о.р 6. Представлення представлення і кодування даних	[показати]
Дані	п.о.р 5. Сеансовий керування сеансом зв'язку	[показати]
Блоки	п.о.р 4. Транспортний безпечне та надійне з'єднання «точка - точка»	[показати]
Пакети	п.о.р 3. Мережевий визначення маршруту та логічних адрес	[показати]
Кадри	п.о.р 2. Канальний MAC та LLC (фізична адресація)	[показати]
Біти	п.о.р 1. Фізичний кабель, сигнали, бінарна передача	[показати]

Рис. 1.3.2.1. Модель OSI

З прикладного рівня інформація передається на транспортний рівень, де розташовані 2 важливі протоколи – TCP та UDP (User Datagram Protocol), з яких TCP частіше використовується, оскільки забезпечує надійність встановленого з'єднання.

TCP забезпечує надійний зв'язок за допомогою позитивного підтвердження повторної передачі (PAR – Positive Acknowledgement with Retransmission). Блок даних протоколу (PDU – Protocol Data Unit) транспортного рівня називається сегментом. Тепер пристрій, що використовує PAR, повторно відправляє блок даних, доки не отримає підтвердження. Якщо блок даних, отриманий на стороні одержувача, пошкоджений (він перевіряє дані за допомогою контрольної суми транспортного рівня), одержувач відкидає сегмент. Таким чином, відправник повинен повторно відправити одиницю даних, для якої не було отримано позитивне підтвердження.

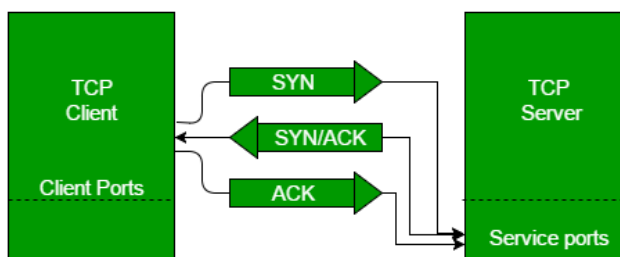


Рис. 1.3.2.2. TCP комунікація

З наведеного вище механізму можна зрозуміти, що для встановлення надійного з'єднання TCP між відправником (клієнтом) і одержувачем (сервером) необхідно обмінятися трьома сегментами:

- 1) SYN(Synchronize Sequence Number – синхронізація номерів послідовності): на першому етапі клієнт хоче встановити з'єднання з сервером, тому він посилає сегмент із SYN, який інформує сервер про те, що клієнт збирається розпочати спілкування і з яким порядковим номером починаються сегменти.
- 2) SYN + ACK(Acknowledgement - підтвердження): сервер відповідає на запит клієнта встановленими бітами сигналу SYN-ACK. ACK вказує про отримання сегменту, а SYN означає, з якого порядкового номера він почне сегменти.
- 3) ACK: у заключній частині клієнт підтверджує відповідь сервера і обидва встановлюють надійне з'єднання, з яким вони почнуть фактичну передачу даних. ACK надсилається одержувачем також при отриманні даних, що теж немає у UDP. Тому важливо при використанні TCP вказувати у властивість `Socket.SendTimeout` певне значення, при UDP немає сенсу цього робити, оскільки відправник немає змогу дізнатись чи дані були доставлені. В C# ця властивість й `Socket.ReceiveTimeout` використовуються лише при викликах асинхронних методів сокета відправлення чи отримання даних

відповідно; для синхронних – очікування буде завжди ідентичним, як без встановлення цих методів доступу: поки не відправляться/отримаються дані відповідно.

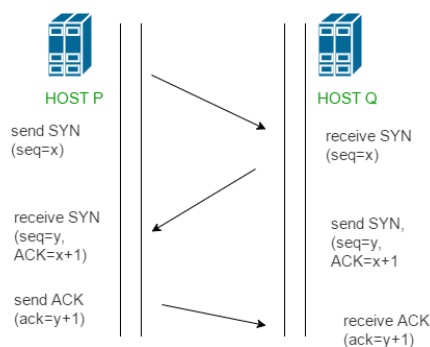


Рис. 1.3.2.3. Детальний опис TCP-комунікація між двома хостами

Як бачимо, встановлення TCP-з'єднань є досить нешвидким процесом, тому було в DS реалізовано пул TCP-з'єднань для збільшення швидкої сервісу (див. відповідний підрозділ).

В .NET для роботи з цим протоколом призначені класи `TcpClient` й `TcpListener`. Ці класи будуться поверх класу `Socket` й ініціалізують відповідний об'єкт наступним чином.

```
var tcpSocket = new Socket(addressFamily, SocketType.Stream, ProtocolType.Tcp);
```

Змінна `addressFamily` приймає будь-яке допустиме значення з перерахування `AddressFamily`, яке вказує на адресну схему. Цю схему буде використовувати сокет для перевірки правильності адреси кінцевих точок (приймачів чи відправників).

Як можна зрозуміти, TCP використовує механізм слухання з'єднань, тобто отримання під'єднаних до кінцевих точок сокетів, за допомогою яких далі можна відправити відповідь.

1.3.3. UDP

UDP (User Datagram Protocol - протокол дейтаграм користувача) – протокол, що без встановлення з'єднання надає можливість швидко передавати дані. Також цей протокол на відміну від TCP немає сигналів квітуння (установка перемикача в положення, що відповідає отриманню певного сигналу). Всі посиальні пакети, що відповідають цьому протоколу називаються дейтаграмами.

Дейтаграма (англ. datagram), також датаграма — блок інформації, посланий як пакет мережевого рівня через передавальне середовище без попереднього встановлення з'єднання і створення віртуального каналу [23]. Максимальний розмір такий пакетів диктується мережею MTU (Maximum transmission unit - максимальний блок передачі).

Недоліком цього протоколу є відсутність гарантії про доставку датаграми кінцевій адресі. Проте його використовується, наприклад, для таких цілей:

- виявлення вузлів в мережі. Окрім того, що можна надсилати окремій кінцевій точці, також є здатність виконувати цю дію для усіх підключених пристроїв мережі за допомогою broadcast (широкопсмугової розсилки) або ж конкретній групі (multicast (див. нижче)).
- Потоківі дані (наприклад, дзвінки, відеозв'язок, керування віддаленими ПК і т. д.), де потрібно дуже швидко надсилати невеликі пакети, втрата яких буде менш відчутною, ніж зависання через постійні перевірки TCP.
- запит до сервера DNS, щоб отримати двійковий еквівалент домену, який використовується для веб-сайту.

В .NET за роботу з UDP відповідає клас UDPClient, який є так званою обгоркою класу Socket.

```
var udpClient = new UdpClient( addressFamily );
```

Для змінної-перечислення `addressFamily` допускається лише значення `AddressFamily.InterNetwork` (IPv4) або ж `AddressFamily.InterNetworkV6` (IPv6), щоб не отримати виняток при створенні даного об'єкту, оскільки UDP підтримує лише дані версії IP-протоколу.

Для створення відповідного об'єкту, який діятиме на основі UDP, в мові C# потрібно передати в конструктор класу `Socket` наступні параметри.

```
var udpSocket = new Socket( addressFamily, SocketType.Dgram, ProtocolType.Udp );
```

`Dgram` – скорочення від `datagram`.

Це найосновніше, що потрібно знайти про UDP-протокол для розуміння роботи DS. Після того, як було вибрано даний протокол, потрібно вирішити, що використовувати для виявлення вузлів. Варіантів всього може бути 2: `broadcast` й `multicast`.

`Broadcast` (широкосмутова розсилка) в комп'ютерних мережах – це відправка даних до всіх приймачів у мережі [25]. Ця відправка не є хорошим вибором для розпізнавання вузлів, які теж використовують DS, оскільки, очевидно, що в мережі можуть бути не тільки вузли, де працює цей сервіс. А коли це стосується великих офісів, як це є з сервісами обміну файлами, то це стає не менш проблемніше, оскільки в мережі можуть перебувати зломисники й коли DS відправить `broadcast`, то можуть без усіляких перехоплень зрозуміти у якому вигляді варто надсилати UDP дейтаграми. З огляду на ці причини було вибрано `multicast`.

`Multicast` (групова розсилка) - доставка даних одночасно у пункти призначення, що підписані на відповідний потік [66]. Отже, `multicast` вирішує проблему, коли сервер намагається почати сесії з множиною потрібних кінцевих точок (так званих отримувачів). `Multicast`-пакети маршрутизуються роутером так само, як `unicast`(індивідуальна адреса, використовується для передачі пакета до

конкретного інтерфейсу хоста), але лише з підписниками відповідної групи за допомогою наступних шарів протоколів :

- 1) Вибраний мережевий протокол (наприклад, UDP), який визначає набір правил й домовленостей зв'язку й обміну даними.
- 2) IGMP (Internet Group Management Protocol – протокол керування групами Інтернету) – протокол управління груповою (multicast) передачею даних в мережах базованих на протоколі IP [59].
- 3) PIM (Protocol Independent Multicast – незалежна від протоколу групова маршрутизація) – множина багатоадресних протоколів маршрутизації для IP-мереж [60].

Сервер й отримувач в деяких реалізаціях PIM зустрічаються в так званій rendezvous point (пункт зустрічі). Звідси вони визначають свій наступний власний шлях. Після цього, наприклад, в Cisco deployment, роутер буде найкоротший шлях між кінцевими точками, дані яких зустрілись. Проте, звісно, це є лише прикладом: маршрутизатори можуть працювати по-різному. Даний алгоритм надає змогу ефективно зібрати пакет даних з серверу й надати його іншим кінцевим точкам, що насправді є найбільш важливим в multicast.

Multicast потік встановлює свій шлях на роутері. Кінцева точка певного вузла мережі може підписатись на отримання цих даних. Наприклад, для IPv4 слухача повідомлень на всіх мережевих інтерфейсах, тобто де IP-адреса рівна 0.0.0.0, підпис на multicast-потік може виглядати так.

```
var multicastOption = new MulticastOption( DsConstants.MulticastAddressIpv4,
IPAddress.Any );
SocketOptionName socketOptionName = SocketOptionName.AddMembership;
Socket.SetSocketOption( SocketOptionLevel.Ipv4, socketOptionName, multicastOption );
```

Або ж наступним чином.

```
UdpClient.JoinMulticastGroup( DsConstants.MulticastAddressIpv4, IPAddress.Any );
```

Цей метод містить ідентичну логіку, як попередній варіант, але є більш читабельним, проте це повинен бути об'єкт типу UdpClient, а не Socket. Для того,

щоб надсилати дані певній групі, потрібно не вказувати 2-го параметру конструктора класу `MulticastOption`.

Коли надсилаються дані на multicast IP-адрес(в цьому випадку на `DsConstants.MulticastAddressIpv4`), то в маршрутизаторах це спричиняє перемиканню switch й надсилання до всіх отримувачів, навіть до тих що не приєднані до multicast групи(`DsConstants.MulticastAddressIpv4`), проте останні відмовлятимуться від цих даних.

Переніс даних маршрутизатором до кінцевих точок, відноситься до протоколу IGMP. Якщо зануритись глибше, то будь-яка multicast MAC-адреса визначається за форматом: 01:00:5E:x:x:x (останні 3 октети беруться з IP multicast й переводяться у MAC-формат). Наприклад, для 239.255.255.250 в MAC-форматі - 01:00:5E:7f:ff:fa.

Multicast MAC-адреси не містяться в MAC-таблиці switch-а. За замовчуванням, коли switch отримує frame даних, то він не знає їх призначення, тому відправляє їх на кожен порт того ж самого Vlinker. Через це всі отримують multicast дані, хто приєднані до відповідного маршрутизатора. Через це був розроблений IGMP Snooping(вистежування) при якому, той хто підписується на multicast потік, надсилає IGMP запит на приєднання (join request), вказуючи маршрутизатору, що він хоче слухати певну групову розсилку. Після цього перемикач може асоціювати цільову MAC-адресу з коректним портом.

Оскільки multicast адреси не маршрутизуються, мережевий стек просто вибирає перший інтерфейс у таблиці маршрутизації з багатоадресним маршрутом. Щоб змінити цю поведінку, опція `MulticastInterface` мови C# може бути використана для встановлення локального інтерфейсу, на який надсилатиметься весь вихідний багатоадресний трафік (тільки для цього сокету). Це робиться шляхом перетворення 4-байтової адреси IPv4 (або 16-байтової адреси IPv6) у масив байтів.

```
multicastSocket.SetSocketOption(
    socketOptionLevel,
    SocketOptionName.MulticastInterface,
    interfaceArray
);
```

Об'єкт `socketOptionLevel` може приймати або значення `SocketOptionLevel.IPv4`, або `SocketOptionLevel.IPv6` для правильної зміни відповідної поведінки.

Це є базова викладка про multicast, оскільки PIM може працювати в різних режимах (`dense` (щільний), `sparse` (рідкісний), `source specific` (конкретне джерело) multicast-y).

1.3.4. Протокол оновлення та пошуку інформації в мережі

Для цих цілей було лише частково використано протокол Kademlia, оскільки LUC потребує високої швидкості скачування файлів і якомога меншої навантаження на вузли мережі. Серед проектів у відкритому доступі, що реалізують цей протокол, було вибрано Clifton.Kademlia, оскільки на основі нього було написано окрему книгу [4]. Також вона представляє досить читабельний код, на відміну від Alethic.Kademlia (єдиної альтернативи на мові C#) [39].

Kademlia, згідно зі статтею, опублікованою в 2015 році Ксін Ші Цай та Люком Девойром, є "стандартним алгоритмом пошуку дефакто для мереж P2P в Інтернеті" [29]. Тобто в мережі, де усі вузли є рівні між собою й кожен може надавати певні сервіси іншим. Kademlia - це специфікація протоколу для децентралізації роботи однорангової мережі, ефективного зберігання та пошуку даних у мережі [4]. Однорангова мережа має кілька позитивних сторін.

- Вона відмовостійка, тобто якщо один або кілька вузлів виходять з мережі, дані, що зберігаються на інших підключених пристроях, повинні бути доступні.
- Вона децентралізована, тобто дані зберігаються не на центральному сервері, а на великій кількості однорангових серверах.

- Складні механізми баз даних не потрібні - дані, що зберігаються в мережі P2P, зазвичай зберігаються у вигляді пар ключ-значення, що надає змогу брати участь у мережі навіть пристроям IoT з обмеженим об'ємом пам'яті.

Kademlia була розроблена Петаром Маймунковим та Давидом Мазьєром у 2002 році [4]. Вона визначає структуру мережі та обмін інформацією через пошук вузлів.

Петар Маймунков щодо походження назви цього протоколу, каже: "Це турецьке слово, що означає "удачлива людина" і, що найважливіше, це назва гірської вершини в Болгарії" [3].

Як вказано в резюме до специфікації Kademlia, завдяки новій метричній топології на основі XOR (виключене «АБО»), Kademlia є першою одноранговою системою, що поєднує в собі узгодженість та продуктивність, маршрутизацію з мінімальною затримкою та симетричну односпрямовану топологію [3].

- Термінологія Kademlia

Варто описати наступні терміни цього протоколу для кращого подальшого розуміння розробки й Kademlia.

- 1) Оверлейна мережа - це мережа, де кожен вузол зберігає (зазвичай неповний) список інших вузлів, що перебувають в мережі.
- 2) Вузол (також відомий як контакт) – рівноправний пристрій мережі. Єдиною його задачею є виконання серверних операцій при отриманні RPC-запиту (див. нижче). Саме тому він містить сховища ключей-значень (BigInteger ID й рядкове значення) для тримання тих значень, які потребує зберегти на поточному вузлі клієнт, що відправив StoreRequest. В DS ці сховища на даних час не застосовуються, оскільки під час інтеграції в сервіс обміну файлами не було знайдено жодної в них потреби. Проте під час аналізу роботи, який відбувся після інтеграції, стало зрозуміло, що вони зможуть пригодитись при

оптимізації скачування (детальніше див. підрозділ «Відомі проблеми сервісу й способи його покращення»). В той же час користувач цього сервісу має змогу їх використовувати, оскільки клас `KadLocalTrp` є публічним й містить метод `Store` (див. додаток 1).

- 3) Ідентифікатор (`Kademlia ID (KID)`) - 160-бітне значення, отримане з `SHA1`-хешу (`Secure Hash Algorithm 1`) деякого ключа або згенероване випадковим чином. Він дає змогу єдиним чином визначити кожен вузол або контакт, а також `ID` запиту. В `Kademlia` для цього використовується випадкове позитивне цілочисельне значення, яке максимум може займати 20 байт (для порівняння `Int64` займає, як можна догадатись, лише 8). В `C#` таке значення надає змогу отримати структуру (значимий тип, значення якого зберігається в стеку адресного простору) – `BigInteger`. Це незмінний тип, що представляє довільно велике ціле число, значення якого теоретично немає верхніх і нижніх меж [30]. Незмінність об'єкту означає, що для отримання нового значення середовище `CLR` (`Common Language Runtime` – загальномовне виконуюче середовище) створює новий об'єкт й присвоює йому потрібне значення. Через цю властивість, а також неіснування верхніх й нижніх меж значень об'єкту, при виконанні будь-якої операції, в результаті якої значення `BigInteger` стає занадто великим, може виникнути виняток `OutOfMemoryException`. Це є однією з причин, чому `opensource`-проектом `Clifton.Kademlia` було створено окремий клас `ID`. Він також надає можливості для отримання випадкових значень типу `BigInteger`. Саме цей сервіс використовується в `DS` для інтеграції відповідного протоколу. В `DS` цей посилальний тип перейменований на `KademliaId` через існування класу `MachineId` (потрібне для присвоювання постійного `ID` вузлам).

- 4) K-Bucket – колекція контактів. Їх не може бути більше, ніж деяке значення K (за замовчуванням воно рівне 20; в DS ідентичне значення) вузлів (або контактів), інколи просто називають - "бакет". Кожен вузол належить певному бакету, діапазон ідентифікаторів якого охоплює ID вузла. Спочатку діапазон ID - це весь спектр від $0 \leq id \leq 2^{160} - 1$.
- 5) BucketList – бінарне дерево K-бакетів. Його основними завданнями є:
 - 5.1) надати бакет, який охоплює вказане KID;
 - 5.2) вказати, чи існує контакт в якому-небудь бакеті;
 - 5.3) спробувати додати контакт в бакет, який охоплює його KID.
- 6) Процедура обміну контактами – періодично (в DS це кожні 30 хвилин) відбувається оновлення K-бакетів шляхом відправки кожному його контакту запиту FindNode. У відповіді містяться інформація про близькі вузли (їх може бути максимум 20 у відповідному повідомленні) відносно випадкового значення KID, що лежить в межах K-бакету. Ця інформація вузлів додається до BucketList.
- 7) Ключ-значення: рівноправні вузли зберігають значення на основі 160-бітних хешованих ключів SHA1. Кожний запис складається з пари ключ-значення.
- 8) Kademlia RPC. Загалом RPC(Remote Procedure Call – виклик віддалених процедур) – комунікаційний протокол, що надає змогу програмі, яка запущена на одному комп'ютері звертатись до функцій програми, що виконуватиметься на іншому вузлі, так ніби це є локальна функція [31]. В Kademlia їх передбачається 4:
 - 8.1) Ping – перевіряє, чи віддалений вузол є онлайн. Як і у всіх Kademlia запитах, містить випадкове ID (проте в даному випадку тільки дане значення обов'язково повинно бути), якщо інший пристрій відповідає таким самим значенням, то він онлайн й

розуміє протокол. В DS використовується лише коли отримується FindNodeRequest від невідомого контакту. Якщо він правильно відповідає на PingRequest, то додається у певний бакет.

8.2) Store – зберігає на іншому пристрої певне значення у вигляді рядка. Найчастіше ця операція використовується для збереження контрольної суми певного файлу. Як було сказано, вище даний протокол в DS не використовується, але підтримується.

8.3) FindValue – спроба отримання значення за певним ключем з віддаленого вузла. Якщо за надісланим ключем в жодному сховищі немає ключа, то надсилаються близькі контакти поблизу цього ключа.

8.4) FindNode – отримання/відновлення інформації про віддалений вузол та його близьких контактів. Протокол FindNode в Kademlia використовується у 2-х випадках:

8.4.1) користувач мережі видає цей RPC для контактів, про які він знає, оновлюючи свій список "близьких" вузлів;

8.4.2) для виявлення інших бакетів в мережі.

9) Маршрутизатор - керує колекцією k-bucket-ів, а також визначає, у яких вузлах зберігається значення ключа.

10) Відстань/близькість - відстань між вузлом та ключем – це обчислення XOR ідентифікатора вузла та ключа. Найбільш важливою особливістю Kademlia є використання цього обчислення XOR для визначення відстані/близькості між ідентифікаторами.

11) Bootstrap - метод, який реєструє надану цій процедурі інформацію про вузол (він повинен бути перевіреним) та ініціалізує наш список бакетів найближчими контактами цього реєр.

Як бачимо, Kademlia є досить складним протоколом, проте ця складність виправдана, оскільки надає змогу оптимізовано скачувати файли з локальної мережі.

1.4. Опис конкурентних конструкцій та шаблонів мови C#

Для початку варто описати наступні поняття:

- 1) Потік - це програмна одиниця, на яку впливає потік управління (виклик функції, цикл, goto і т.д.), оскільки ці інструкції працюють із покажчиком інструкцій, який належить конкретному потоку [56]. Потоки часто плануються відповідно до деякої схеми пріоритетів [56] (хоча можна спроектувати систему з одним потоком на ядро процесора, у цьому випадку кожен потік завжди працює і планування не потрібно).
- 2) Конкурентність – виконання одразу кількох дій в один й той самий час [1].
- 3) Асинхронність – різновид конкурентності, який використовує наперед обдуману секцію коду або зворотні виклики для запобігання створення зайвих потоків [1].
- 4) Завдання в конкурентному програмуванні – секція коду, що представляє певну операцію, яка виконується або збирається це робити. Рекомендується, щоб вона була ні короткою, ні довготривалою для оптимізації виконання [1].

1.4.1. Захоплення контексту потоку

Контекст потоку включає всю інформацію, необхідну потоку для безперешкодного відновлення виконання, включаючи набір реєстрів процесора та стек потоку. Декілька потоків можуть виконуватися в контексті одного процесу [40]. Усі потоки процесу поширюють однаковий його віртуальний адресний простір, але будь-який потік може не побачити оновлення простору

іншим (щоб це уникнути використовують ключове слово `volatile`, замки, клас `Interlocked` та інші способи синхронізації залежно від ситуації). Потік може виконувати будь-яку частину програмного коду, включаючи частини, що виконуються на даний момент іншим потоком, проте локальні змінні методу, який він виконує, є приватними для нього.

Коли асинхронний метод відновлює роботу після `await` (див підрозділ 1.4.3. Асинхронний шаблон на основі завдань), то за замовчуванням він продовжує виконання в тому ж контексті. Це може спричинити проблеми зі швидкістю, особливо, якщо це був UI-контекст, в якому відновлює роботу велика кількість асинхронних методів. Виникає питання: «Скільки продовжень в UI-потоці перевищує допустимий поріг?» Однозначної й простої відповіді на це немає, але Люціан Віщик з Microsoft рекомендує: біля сотні в секунду – нормально, але біля тисячі – це вже забагато [1].

Найкраще для кожного асинхронного методу, якщо він не повинен відновлюватись на контексті, в якому він почав свою роботу, використовувати метод `ConfigureAwait`:

```
private async Task ResumeOnSameContextAsync()
{
    await Task.Delay( millisecondsDelay: 1000 );
    //here is the same context as before previous row
}

private async Task ResumeOnDifferentContextAsync()
{
    await Task.Delay( millisecondsDelay: 1000 ).ConfigureAwait(
        continueOnCapturedContext: false );
    //here is the different context than before previous row
}
```

Ніяких незручностей це не створить. Значення спільних й локальних даних потоку перебуватимуть у потрібному стані (або оновлені, якщо викликаний метод або інший потік їх змінив або ідентичними, як до виклику функції), якщо потокобезпечно змінюються (ця ремарка стосується лише рідкісних випадків оновлення спільних даних). Усі асинхронні методи повинні повертати якесь значення, оскільки по-іншому не буде можливості очікувати їх завершення без

обхідних шляхів, тому будь-який асинхронний метод з правильною сигнатурою (повертаюче значення, назва методу, послідовність типів параметрів) також може називатись функцією. З огляду на вище сказане, можна дійти висновку, що будь-який асинхронний метод в бібліотечному й консольному (не володіє особливими властивостями, як UI-додаток) проектах варто використовувати з `ConfigureAwait` з параметром `false` для збільшення швидкодії.

Якщо існує асинхронна функція з частинами, які вимагають контексту й частинами, які вільні від нього, то варто розглянути можливість розбивання на два або більше асинхронних методів. Такий підхід допомагає організувати код по рівням:

```
private async void button1_Click( Object sender, EventArgs EventArgs )
{
    button1.Enabled = false;
    try
    {
        //can't use ConfigureAwait here, because
        //we need to continue on UI-context
        await ExecuteManyAsyncMethodsAsync();
    }
    finally
    {
        //we are back on the UI-context for this method
        button1.Enabled = true;
        await ResumeOnSameContextAsync().ConfigureAwait(
            continueOnCapturedContext: false );
        //here we are on different context
    }
}

private async Task ExecuteManyAsyncMethodsAsync()
{
    await ResumeOnSameContextAsync().ConfigureAwait(continueOnCapturedContext: false);
    await Task.Delay( millisecondsDelay: 1000 ).ConfigureAwait( false );
    await ResumeOnDifferentContextAsync().ConfigureAwait( false );
    await Task.Delay( 1000 ).ConfigureAwait( false );
}
```

Тобто замість того, щоб викликати багато асинхронних методів без `ConfigureAwait` перед зміною `button1`, було створено й викликано окремий метод, що це робить. Якщо викликати `ConfigureAwait` для `ExecuteManyAsyncMethodsAsync`, то ми отримаємо `InvalidOperationException`

через спробу зміни `button1` на не UI-контексті (те ж саме може статись при роботі зі запитами/відповідями ASP.NET).

Також важливо пам'ятати, що кожен асинхронний метод має свій контекст, що надає зміст використовувати даний підхід.

1.4.2. Оновлення значення однієї змінної кількома потоками

При зміні спільних змінних кількома потоками, повинні відбуватись взаємовиключення, тобто редагування значень має виконуватись в будь-який конкретний момент часу лише одним потоком, інакше це може призвести до перевпорядкування інструкцій й в результаті змінна може набути якогось довільного значення.

Для вирішення цієї задачі може бути використаний один з наступних підходів.

1) Замок

Один з варіантів синхронізації (спільна робота двох або більше потоків, яка гарантує, що кожен потік досягне відомої точки роботи стосовно інших потоків, перш ніж продовжити роботу [43]) в C# реалізується за допомогою механізму замків [44]. Вони насправді створюють чимале навантаження на процесор (інші вбудовані в C# варіанти синхронізації теж, але меншу), тому бажано реалізовувати логіку програми так, щоб максимально уникати синхронізацію потоків, особливо в програмах, що чутливі до швидкості виконання та до пам'яті.

Замки варто використовувати лише для блокування виконання кількох операцій або однієї, яка не може бути виконана за допомогою класу `Interlocked` [45].

2) Volatile

Член певного типу з модифікатором `volatile` (непостійний) насправді не є повністю потокобезпечним. Його варто використовувати у випадках, коли відповідний об'єкт застосовується лише для читання певними потоками, а

іншими – лише для запису, який є атомарною операцією (неперервною) [41]. Якщо об'єкт не підтримує використання модифікатора `volatile`, то можна скористатись класом `Volatile` [42].

В однопроцесорній системі волатильні операції читання та запису гарантують, що значення зчитується або записується в пам'ять, а не кешується (наприклад, у регістрі процесора). Таким чином, можна використовувати ці операції для синхронізації доступу до поля, яке можна оновити іншим потоком.

У багатопроцесорній системі через оптимізацію продуктивності компілятора або процесора звичайні операції з пам'яттю можуть виглядати переупорядкованими, коли кілька процесорів працюють з однією пам'яттю. Волатильні операції з пам'яттю запобігають деяким видам переупорядкування стосовно операції. Волатильна операція запису запобігає переупорядкування ранніх операцій пам'яті, щоб вони відбувалися після волатильного запису. Волатильна операція читання запобігає переупорядкування пізніших операцій пам'яті до волатильного читання. Ці операції можуть включати бар'єри пам'яті на деяких процесорах, що може вплинути на продуктивність.

У багатопроцесорній системі операція волатильного читання не гарантує отримання останнього значення, записаного в цю ділянку пам'яті будь-яким процесором. Аналогічно, операція запису не гарантує, що записане значення негайно видно іншим процесорам.

3) Interlocked

Цей статичний клас представляє атомарні операції для спільних змінних [35]. Методи цього класу надають змогу захиститись від проблем, які можуть виникати, коли планувальник (об'єкт, що вирішує, де повинен виконуватись той чи інший код [30]) переключає контексти, коли потік оновлює ресурс, доступ до якого відбувається іншими потоками або коли 2 шляхи виконання працюють

одночасно на різних процесорах. Варто також відмітити, що члени даного класу не генерують винятки.

1.4.3. Асинхронний шаблон на основі завдань

В .NET цей шаблон (його більш поширена назва – Task Based Asynchronous Pattern (TAP)) – рекомендований патерн асинхронного проектування для розробки додатків [46]. Він базується на Task і Task<TResult> типах, які використовуються для представлень асинхронних операцій, можливості отримання процесу їх виконання й очікування результату. Для останнього варто використовувати ключове слово await, який доступний лише в тому випадку, якщо метод має модифікатор async (для вказівки компілятору, що ця функція чи процедура може виконуватись асинхронно).

1.4.4. Модель асинхронного програмування

В цьому застарілому шаблоні (його більш поширена назва – Asynchronous Programming Model (APM)) використовуються пари методів з іменами Begin<Операція> і End<Операція>, а також об'єктом типу IAsyncResult, що представляє операцію асинхронну. В класі Socket є багато методів, що відповідають цьому патерну. Через це було створено клас AsyncSocket, який має зручні у використанні та оптимізовані функції та процедури (див. додаток 2). Для огляду використання цієї моделі також дивіться цей клас, оскільки в інших частинах коду було уникнуто його використання.

1.4.5. Постачальник/Споживач шаблон

Цей патерн доволі часто використовується в конкурентному програмуванні. Його сенс полягає в тому, що один чи кілька потоків (producers, виробники, постачальники) надають дані, і при цьому паралельно певна кількість потоків (споживачі) отримують їх та можуть виконувати певні дії з ними. В сучасних мовах програмування високого рівня немає сенсу писати реалізацію

цього патерну власноруч. В .NET існує множина класів, які імплементують Постачальник/Споживач шаблон. В сервісі виявлення було використано 3 з них: `BlockingCollection`, `ConcurrentBag` й `ActionBlock`. Також в C# є окремий інтерфейс `IProducerConsumerCollection`, який представляє методи, які повинні бути реалізовані для зручного й ефективного створення відповідної колекції. Перевагою таких колекцій в .NET є те, що вони практично ніколи не використовують замки. `BlockingCollection` має відношення композиції до `IProducerConsumerCollection`, `ConcurrentBag` - реалізації, `ActionBlock` – зовсім не має зв'язку з цим інтерфейсом, оскільки має дещо іншу логіку реалізації й використання.

- `BlockingCollection`

Ця потокобезпечна колекція є так званою «обгорткою» будь-якої колекції, яка реалізує `IProducerConsumerCollection`, і надає змогу виконувати наступні операції [2].

- 1) Вказати, що додавання елементів більше ніколи не відбудеться (метод `CompleteAdding`);
- 2) додати новий елемент (методи `Add`, `TryAdd` (останній повертає `false`, якщо не вдалося додати (розмір колекції досяг свого ліміту)));
- 3) взяти елемент з «обгорнутої» колекції. Якщо немає жодного, то виникає синхронне блокування споживача, доки не буде вказано, що додавання будь-якого елементу більше не відбудеться, або доки якийсь елемент не буде доданий;
- 4) послідовно отримувати всі елементи, поки не буде викликаний метод `CompleteAdding`. Для цього використовується функція `GetConsumingEnumerable`.

`BlockingCollection` також дозволяє обмежити загальний розмір колекції, блокуючи виробника при перевищенні цього розміру.

- ConcurrentBag

Ця потокобезпечна колекція зберігає неупорядковану колекцію об'єктів (з дозволеними дублікатами). ConcurrentBag підходить для ситуацій, коли неважливо який елемент отримується при виклику TryTake чи TryPeek. Надає можливість потокобезпечно:

- 1) додавати елемент (метод Add);
- 2) одержати елемент (метод TryPeek);
- 3) брати, тобто отримати й видалити об'єкти з колекції (метод TryTake).

Ці дві останні функції повертають false, якщо немає більше елементів;

- 4) приводити елементи до непосортованого масиву.

- ActionBlock

Інтерфейс у ActionBlock дуже схожий до інтерфейсу BlockingCollection, але перший не представляє можливості взяти наданий елемент, він автоматично буде оброблений методом, яким був ініціалізований ActionBlock. Також даний клас має властивість Completion за допомогою якої можна очікувати завершення усіх поставлених даних в блок.

Приклади використання ConcurrentBag й ActionBlock див. у підрозділі «Фонове відновлення з'єднань», BlockingCollection - у додатку 2.

1.5. Вибір пакетів для розробки сервісу

Рішення щодо пакетів (у випадку платформи .NET – NuGet-пакетів [15]) є важливою частиною розробки, оскільки деякі з них не є у відкритому доступі, що сповільнює визначення швидкодії методів, унеможлиблює розуміння їхньої реалізації й внесення змін або, навіть, виправлення певних частин.

- Вибір пакету для створення постійного ID вузла мережі

1) Xam.Plugin.DeviceInfo

Найпопулярніший пакет (2,03 млн. установлень) для визначення ID пристрою, що належить до ОС Windows, iOS чи Android. Надає можливість також створити ідентифікатор для застосунку. Суттєвим недоліком цього пакету є відсутність підтримки Linux.

2) DeviceId

Це другий за популярністю пакет, що підтримує усі ОС комп'ютерів. Надає змогу створити ідентифікатор базуючись на ID процесору, серійному номері материнської плати, назві пристрою, користувача ОС і т.д.

Оскільки однією з опцій розширення можливостей DS є підтримка усіх комп'ютерів, то, звісно, використовується пакет DeviceId.

- Вибір пакету для збереження інформації роботи сервісу

Під збереження інформації мається на увазі логування. Усього є 3 наступні найпопулярніші варіанти, які посортовано в порядку спадання за цим показником: Microsoft.Extensions.Logging, Microsoft.IdentityModel.Logging й Serilog. Було вибрано саме останній, оскільки він підтримує запис в Sentry, а також надає можливість виконати логування на якийсь свій сервер з ідентичною ціллю, як робить ця служба.

Sentry - служба, яка відслідковує, усуває збої в режимі реального часу та містить API для надсилання подій з кількох мов у різних програмах [52]. Вона раніше використовувалась в LUC, під час інтеграції DS, але згодом було вирішено перейти на логування на сервер обміну файлами.

- Вибір пакетів для частих потреб при роботі з асинхронним шаблоном на основі завдань

Найвідомішою бібліотекою для різноманітних частих потреб при роботі з конкурентним кодом є Nito.AsyncEx. Вона надає такі можливості:

- 1) Синхронізація потоків для виконання асинхронного коду. Проблема в тому, що в блоці коду ключового слова `lock` не можна використовувати `await`, оскільки це може призвести до `deadlock` (тупикової ситуації) [1]. Саме через це варто використовувати клас `Nito.AsyncEx.AsyncLock`. Приклад, використання в класі `NetworkEventInvoker` (див. додаток 3):

```
internal async Task SendQueryAsync( IoBehavior ioBehavior )
{
    if ( m_udpSenders != null )
    {
        if ( ioBehavior == IoBehavior.Asynchronous )
        {
            using ( await m_asyncLock.LockAsync().ConfigureAwait(
                continueOnCapturedContext: false ) )
            {
                await SendUdpMessagesAsync( ioBehavior ).ConfigureAwait(
false );
            }
        }
        else if ( ioBehavior == IoBehavior.Synchronous )
        {
            using ( m_asyncLock.Lock() )
            {
                await SendUdpMessagesAsync( ioBehavior ).ConfigureAwait(
false );
            }
        }
        else
        {
            throw new ArgumentException( message: $"Has incorrect value:
{ioBehavior}", paramName: nameof( ioBehavior ) );
        }
    }
}
```

Як бачимо цей метод підтримує як асинхронне очікування звільнення замку, так і синхронне. Конструкція `using` гарантує звільнення замку, що б не відбувалось у відповідному блоці коду.

- 2) Синхронне очікування завершення виклику події з асинхронними обробниками

Для цієї задачі існує клас `AsyncContext`, який є потокобезпечним й забезпечує контекст для асинхронних операцій, і його `static` методи-перегрузки під назвою `Run`. Наприклад,

```
AsyncContext.Run( action: () => receiveEvent?.Invoke( this, eventArgs ) );
```

- Вибір пакету для паралельної обробки даних й асинхронного очікування завершення

Для більшості синхронних викликів варто здебільшого використовувати вбудований в .NET статичний клас `Parallel`, який дає змогу паралельно виконувати різноманітні дії. Проте в нього є значний недолік: для використання асинхронних операцій потрібно сторонніми способами (наприклад, об'єктом типу `SemaphoreSlim` [53]) очікувати на завершення всіх потоків, використовуваних під час виконання методу, що значно сповільнює виконання. Саме тому часто доводиться застосовувати якийсь пакет для забезпечення таких дій. За рекомендаціями у розробницьких форумах, використано реалізований компанією Microsoft пакет `System.Threading.Tasks.Dataflow` й саме він застосовується для оптимізованої роботи пулу TCP-з'єднань й скачування файлів. Його також можна успішно використовувати для синхронних методів, якщо поєднувати з пакетом `Nito.AsyncEx`.

1.6. Вибір сервісу для емуляції кількох вузлів на одному пристрої

Для цього прекрасно підходить `Docker compose` - інструмент, розроблений для визначення та спільного використання багатоконтейнерних програм [47]. Було вибрано саме цей інструмент, оскільки знання `Docker` в майбутньому завжди пригодяться, а також через те, що `VS2022` підтримує `Docker-Compose` без усіляких обхідних шляхів (наприклад, установлень розширень `VS`) й дозволяє легко створювати контейнери.

`Docker` контейнер - це абстракція на рівні додатків, яка упаковує код і його залежності разом [48]. Декілька контейнерів можуть працювати на одній машині та спільно використовувати ядро ОС з іншими контейнерами, кожен з яких працює як ізольований процес у просторі користувача.

`Docker` зображення – приватна файлова система для контейнера [49]. Вона забезпечує всі файли й код, що потребує контейнер.

РОЗДІЛ II. ФІЗИЧНА СТРУКТУРА СЕРВІСУ ВИЯВЛЕННЯ

Корінь репозиторію DS [70] включає в себе наступний зміст.

Ім'я	Дата змінення	Тип	Розмір
Installer	03.06.2022 11:03	Папка файлів	
WpfClient	31.05.2022 17:57	Папка файлів	
.gitignore	26.05.2022 13:16	Файл GITIGNORE	5 КБ

Рис. 2.1. Зміст кореневої папки репозиторію DS

Файл .gitignore вказує на файли й папки, які повинні бути проігноровані при синхронізації з віддаленим репозиторієм. В папці Installer розміщені файли для інсталяції проекту, його компіляції, а також іконки проекту й файли для оновлення до новішої версії. Лише змінено структуру папки Installer та виправлено компілятор, тобто файл Installer.iss (див. про це підрозділ «Виправлення сервісу обміну файлами»).

Ім'я	Дата змінення	Тип	Розмір
Icons	03.06.2022 11:03	Папка файлів	
Updater	28.05.2022 17:46	Папка файлів	
Installer.iss	02.04.2022 23:55	Inno Setup Script	5 КБ
Light Upon Cloud Installer.exe	26.05.2022 13:16	Застосунок	7 193 КБ
LUC_certificate.p12	07.02.2022 11:13	X.509 Certificate	7 КБ
LUC_certificate_with_password.p12	07.02.2022 11:13	X.509 Certificate	7 КБ

Рис. 2.2. Файли та підпапки директорії «Installer»

Папка «WpfClient» містить усі проекти LUC, основною платформою якого є WPF. Ця директорія має наступний вигляд.

Ім'я	Дата змінення	Тип	Розмір
<u>bin</u>	31.05.2022 10:48	Папка файлів	
LightClientLibrary	28.05.2022 17:32	Папка файлів	
LUC.ApiClient	30.05.2022 18:00	Папка файлів	
LUC.Common.PrismEvents	28.05.2022 17:33	Папка файлів	
<u>LUC.DiscoveryService</u>	02.06.2022 21:22	Папка файлів	
<u>LUC.DiscoveryService.Test</u>	31.05.2022 13:11	Папка файлів	
LUC.DubstackAdsUtility	28.05.2022 17:33	Папка файлів	
LUC.Globalization	28.05.2022 17:33	Папка файлів	
LUC.IntegrationTests	28.05.2022 17:33	Папка файлів	
LUC.Interfaces	28.05.2022 17:33	Папка файлів	
LUC.Services.Implementation	01.06.2022 14:04	Папка файлів	
<u>LUC.UnitTests</u>	30.05.2022 14:52	Папка файлів	
LUC.ViewModels	28.05.2022 17:34	Папка файлів	
LUC.WpfClient	03.06.2022 10:09	Папка файлів	
NotifyIconWpf	28.05.2022 17:34	Папка файлів	
<u>obj</u>	01.06.2022 19:31	Папка файлів	
<u>.editorconfig</u>	26.05.2022 13:16	Файл EDITORCON...	13 КБ
<u>docker-compose.dcproj</u>	31.05.2022 9:36	Файл DCPROJ	1 КБ
Reporting.WpfClient.sln	03.06.2022 10:43	Visual Studio Solut...	51 КБ
Reporting.WpfClient.sln.DotSettings	26.05.2022 13:16	Файл DOTSETTIN...	3 КБ
<u>docker-compose.yml</u>	31.05.2022 17:57	Исходный файл Y...	1 КБ
TestPlaylist1.playlist	07.02.2022 11:13	Файл PLAYLIST	2 КБ

Рис. 2.3. Зміст папки «WpfClient» й
виділені об'єкти, які були додані мною

Коротко оглянемо підкреслені елементи.

- bin – містить елементи для побудови й запуску проекту DS.Test контейнерами Docker-compose. Загалом папки з такою назвою зазвичай завжди містяться потрібні об'єкти файлової системи для виконання збірок.
- LUC.DiscoveryService й LUC.DiscoveryService.Test – директорії з проектами DS й DS.Test відповідно.
- LUC.UnitTests – папка зі збіркою модульних тестів LUC. Були реалізовані лише деякі тести для класу Downloader з проекту LUC.ApiClient й ServerObjectDescription - з LUC.Interfaces (детальніше див. підрозділ «Модульні тести»).
- obj – містить налаштування та деякі результати роботи (згенеровані автоматично) Docker-compose.

- `.editorconfig` – надає можливість покращити використання єдиного стилю в усьому проекті. Реалізація цього файлу погоджувалась з розробником LUC.
- `docker-compose.dcpj` – файл з конфігураціями проекту `docker-compose`.
- `docker-compose.yml` - вказує особливості створення Docker-контейнерів.

Загалом після інтеграції DS з LUC, рішення [71] у VS має наступний вигляд.

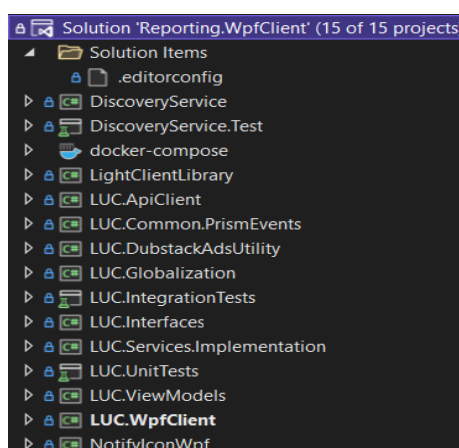


Рис. 2.4. Рішення проектів після інтеграції
сервісу виявлення з сервісом обміну файлами

Тепер розглянемо детальніше фізичну структуру проекту `DiscoveryService`.

Ім'я	Дата змінення	Тип	Розмір
bin	24.05.2022 23:05	Папка файлів	
CodingData	01.06.2022 14:04	Папка файлів	
Common	03.06.2022 18:31	Папка файлів	
Kademlia	02.06.2022 11:41	Папка файлів	
Messages	03.06.2022 18:14	Папка файлів	
NetworkEventHandlers	01.06.2022 8:20	Папка файлів	
obj	03.06.2022 14:18	Папка файлів	
Properties	25.05.2022 20:50	Папка файлів	
DiscoveryService.cs	03.06.2022 15:33	C# Source File	23 КБ
DiscoveryService.csproj	03.06.2022 18:14	C# Project File	16 КБ
DiscoveryServiceFacade.cs	03.06.2022 15:33	C# Source File	3 КБ
NetworkEventInvoker.cs	03.06.2022 15:56	C# Source File	36 КБ
TcpListenersCollection.cs	03.06.2022 18:14	C# Source File	6 КБ
TcpServer.cs	02.06.2022 15:01	C# Source File	29 КБ
TcpSession.cs	25.05.2022 20:50	C# Source File	28 КБ
UdpListenersCollection.cs	03.06.2022 18:14	C# Source File	8 КБ
UdpSendersCollection.cs	03.06.2022 15:56	C# Source File	9 КБ

Рис. 2.5. Фізична структура папки `WpfClient\LUC.DiscoveryService`

- Тут розташовані файли з розширенням .cs, тобто ті, що містять типи даних мови C#. Опишемо коротко тих, що не описані в інших розділах.

- 1) TcpServer – виконує відповідну функцію. Взятий з opensource [72], проте було додано деякі правки відповідно до потреб DS (отримання TCP-сесії з повідомленням(-и)).
- 2) TcpSession – взятий з аналогічного проекту; є обгорткою TCP Socket.

Також тут розташований файл з розширенням .csproj, який зберігає налаштування проекту та посилання на всі ресурси, що використовуються у збірці [62].

- Директорія obj – містить об'єктні або проміжні файли. По суті, це фрагменти, які будуть об'єднані для створення кінцевого виконуваного файлу. Компілятор генерує один об'єктний файл для кожного вихідного файлу, і ці файли розміщуються в папці obj.
- Директорія bin, як зазначалося вище, містить файли виконання проекту, тобто це різноманітні .dll-файли, .pdb [73] та інші.
- Properties – містить інформацію про властивості збірки (її назва, опис, конфігурація і т.д.).
- CodingData – містить класи для кодування та декодування бінарних даних. Ця та нижче описані папки проекту DS можна також повністю переглянути в вікні «Оглядач рішень» VS, оскільки вони містять лише файли посилальних типів.

Ім'я	Дата змінення	Тип	Розмір
Binary.cs	25.05.2022 20:50	C# Source File	1 КБ
Buffer.cs	25.05.2022 20:50	C# Source File	7 КБ
WireReader.cs	01.06.2022 14:04	C# Source File	11 КБ
WireWriter.cs	01.06.2022 14:04	C# Source File	12 КБ

Рис. 2.6. Склад папки «CodingData»

- Common – містить різноманітні методи розширення до класів .NET й пакетів, а також власноруч реалізовані типи даних й інтерфейси, що часто використовуються в проєкті.

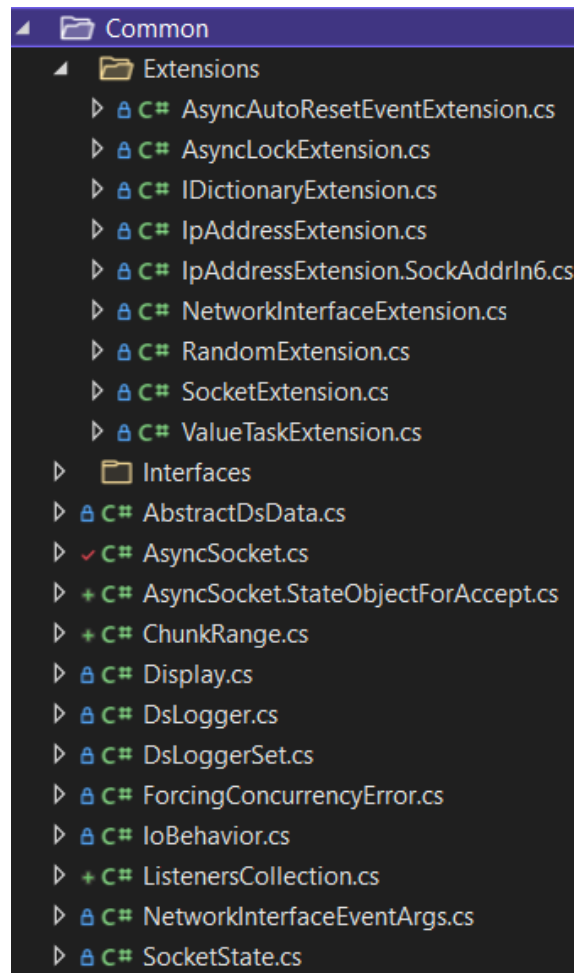


Рис. 2.7. Зміст папки «Common»

- Kademlia – містить усі типи даних для реалізації відповідного протоколу та його оптимізації, які були взяті з opensource Clifton.Kademlia.

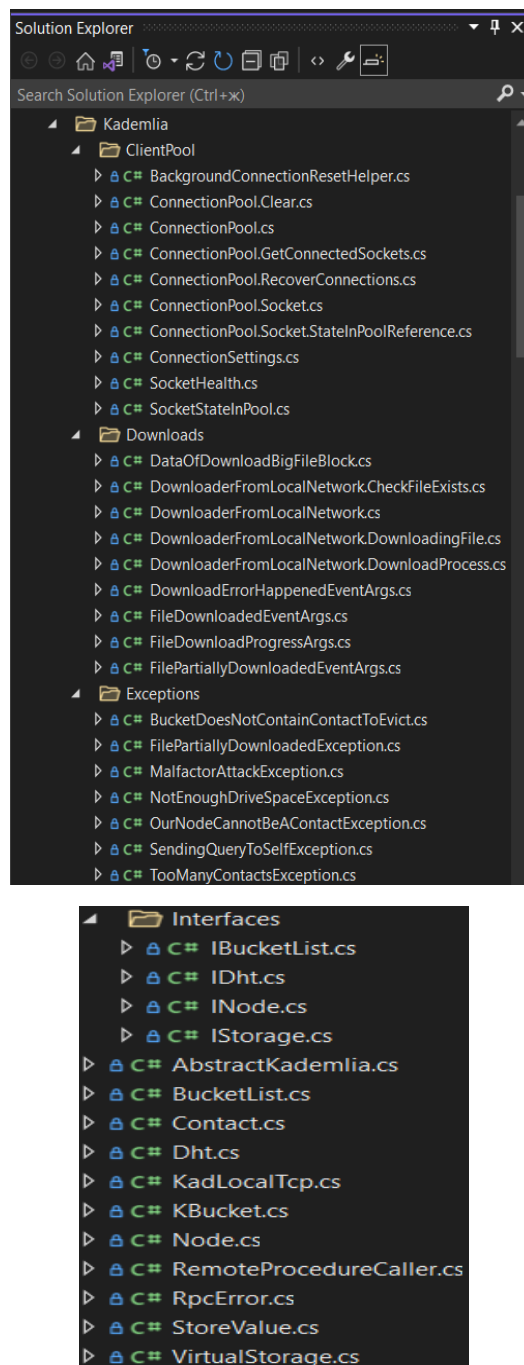


Рис. 2.8. Склад папки «Kademlia»

- **Messages** – папка з усіма класами повідомлень, подій пов'язаних з ними, а також перечислення `MessageOperation`, яке вказує на вид повідомлення, й клас `RecentMessages`, який зберігає ID повідомлень, що отримані в певному інтервалі (детальніше див. «Реалізація обробки повідомлень»).

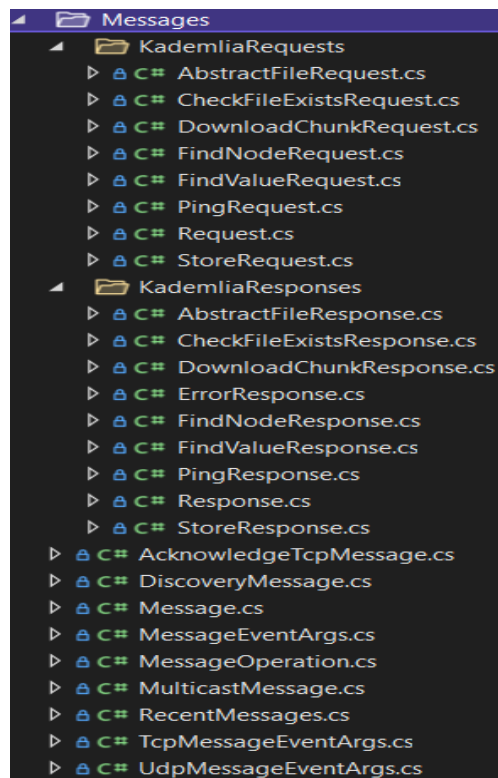


Рис. 2.9. Зміст папки «Messages»

- NetworkEventHandlers – містить класи, що обробляють усі TCP-повідомлення.

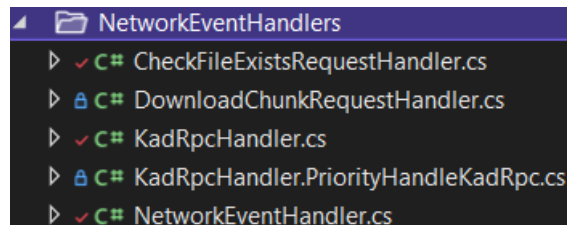


Рис. 2.10. Склад директорії «NetworkEventHandlers»

Отже, сервіс виявлення має досить велику кількість класів. Це пов'язане з тим, що було більшість класів реалізовані за принципами SOLID [74] й сервіс потребує високого рівня безпеки та оптимальності навантаження на вузли мережі. Іншою причиною є наявність деяких типів даних з opensource проектів, а не установлення їх, як окремих пакетів.

РОЗДІЛ III. РЕАЛІЗАЦІЯ СЕРВІСУ ВИЯВЛЕННЯ

Насамперед варто зазначити, що багато принципів з книги [69] (особливо стосовно написання чистого коду й оптимізації) використовувались при розробці DS.

3.1. Конфігурації проекту

- Налаштування виконання збірки сервісу виявлення

Всього збірка DS містить наступні налаштування.

Таблиця 1

Конфігурації проекту DS

Конфігурація збірки	Призначення	Оголошені символи директиви препроцесора та їх використання в DS	
Debug	Для відлагодження проекту	DEBUG	Для визначення, чи варто логувати певну інформацію в консоль чи файли
		TRACE	-
Release	Оптимізувати роботу проекту шляхом відсутності опцій відлагодження. Використовується здебільшого клієнтами додатків.	TRACE	-

Продовження табл. 1.

1	2	3
ReceiveUdpFromOurself	<p>Для відлагодження проекту з дозволом отримання власних UDP повідомлень.</p> <p>Зазвичай використовується, коли лише 1 вузол в мережі використовує DS.</p>	<p>Оголошує аналогічні символи директиви, як режим Debug.</p> <p>Але на додаток також RECEIVE_UDP_FROM_OURSELF – для дозволу отримання від самого себе UDP-повідомлень. Це також спричиняє отримання TCP (див. підрозділ «Алгоритм виявлення вузлів»).</p>
ConnectionPoolTest	<p>Для відлагодження правильності роботи пулу з'єднань.</p>	<p>Оголошує аналогічні символи директиви, як режим ReceiveUdpFromOurself, оскільки здебільшого використовується при існуванні 1 вузла, що застосовує DS. На додаток також декларує CONNECTION_POOL_TEST для додаткових логів, що пов'язані з пулом з'єднань й киданням винятків у випадку неправильної його роботи.</p>

Тобто ці налаштування здебільшого впливають на деякі логування, кидання винятків й отриманні повідомлень, що при Release-режимі відфільтровуються.

- Порти обміну повідомленнями

Для сервісу виявлення було вибрано порт 17500, оскільки саме його використовує Dropbox – один з найпопулярніших сервісів обміну файлами [51], через велику вартість якого й розробляється LUC. Звісно, клієнти не будуть використовувати одночасно ці 2 сервіси, а використання цього порту гарантує обхід помилок, з якими вже стикались розробники Dropbox. Усього передбачається, що в DS підтримуватиме наступний діапазон портів для обміну файлами: 17500-17510.

- Вибір адрес групової розсилки

Для IPv4 було вибрано 239.255.0.251, для IPv6 - FF02::D, оскільки вони належить до діапазону допустимих IP-адрес для multicast й не було знайдено мною жодного сервісу, який б ці адреси використовував.

- Налаштування файлів-маніфестів

В обох проектах – DS.Test й LUC додані файли з розширенням .manifest у папках Properties. Це потрібно для того, щоб вимагати адміністративні права, щоб DS міг налаштувати брандмауер. Тому у відповідні файли слід прописати наступний рядок.

```
<requestedExecutionLevel level="requireAdministrator" uiAccess="false" />
```

Він повинен бути вказаний в тезі <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v2">. Таким чином при кожному запуску DS.Test й LUC вимагаються адміністративні права.

3.2. Схема класів

Перед тим, як розглядати деталі реалізації сервісу виявлення, оглянемо його схему для кращого подальшого розуміння.

Її побудувати можна завдяки інструменту VS – Class Designer. Для цього варто відкрити елемент стрічки меню «Вигляд» → «Представлення класів». В результаті отримаємо наступне вікно.

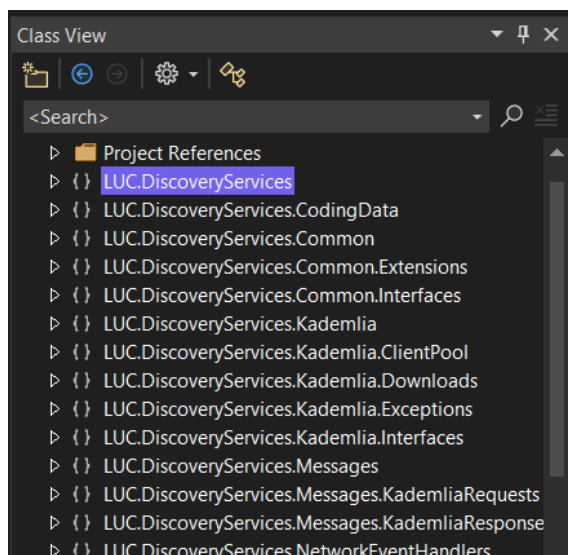


Рис. 3.2.1. Вікно «Представлення класів»

Як бачимо, тут представлено всі простори імен проекту DS. Щоб переглянути діаграму одного з них, потрібно натиснути ПКМ (правою кнопкою миші) на відповідну назву простору й ЛКМ обрати опцію «Оглянути діаграму класів».

Спершу виділимо інтерфейси для кращого подальшого розуміння їх реалізації.

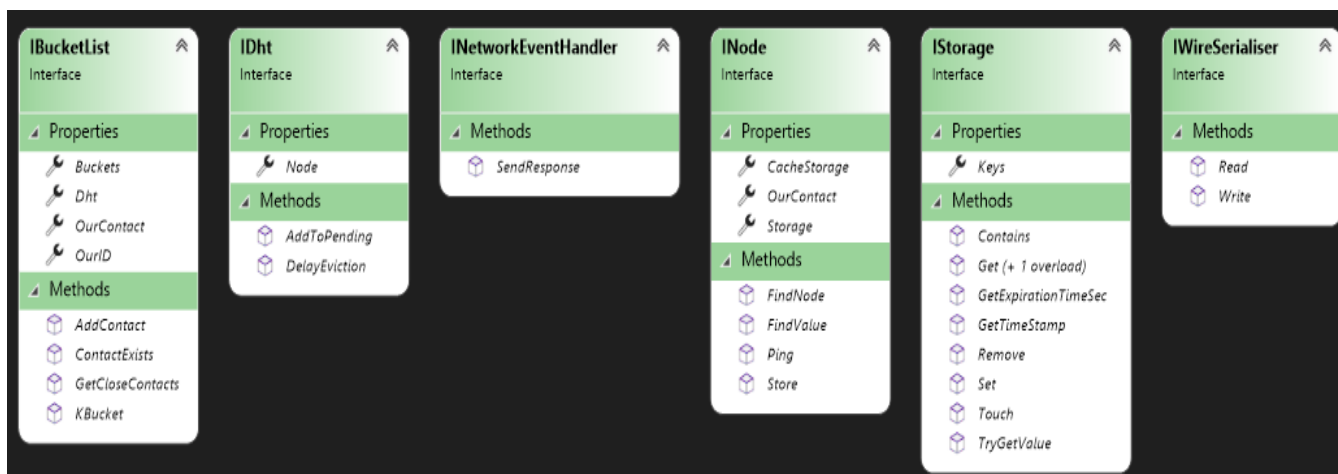


Рис. 3.2.2. Інтерфейси проекту DS

Набір класів, які реалізують ці інтерфейси, але не успадковують інші власноруч реалізовані типи представлені у наступному зображенні.



Рис. 3.2.3. Діаграма класів, які не успадковують інші власноруч реалізовані типи

Як бачимо, таких класів є досить багато, оскільки наслідування варто використовувати, лише коли це не суперечить принципам SOLID.

Також існують наступні типи перелічень.

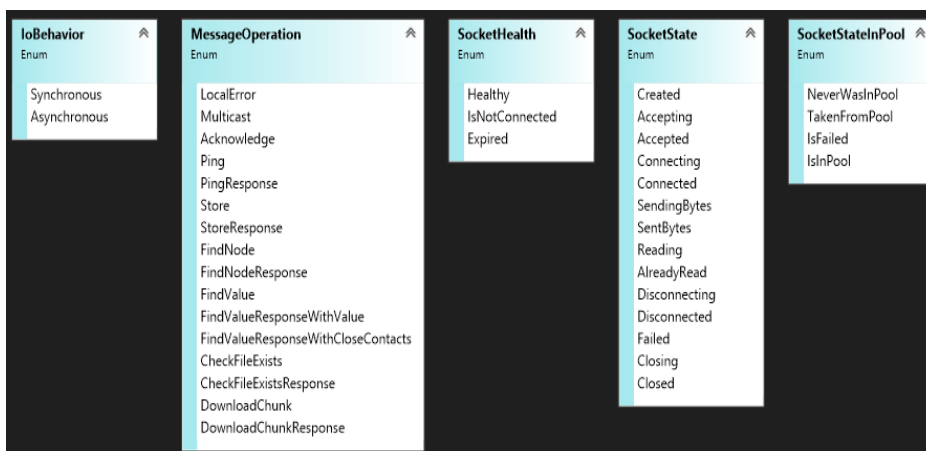


Рис. 3.2.4. Типи перелічень з усіма допустимими значеннями

Тепер оглянемо ієрархію повідомлень. В її основі лежать наступні типи.

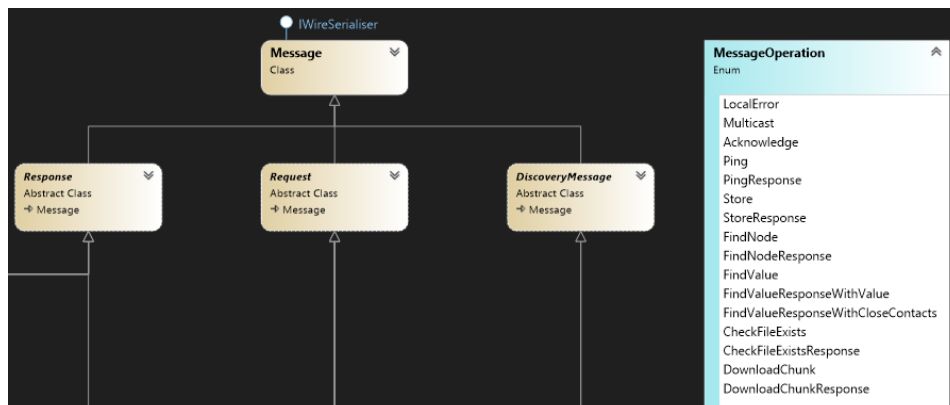


Рис. 3.2.5. Основні типи об'єктів повідомлень

MessageOperation вказує на вид повідомлення й завжди його значення міститься у першому байті пакету.

Клас DiscoveryMessage – абстрактний тип, що містить усі властивості, що потрібні для виявлення вузлів.

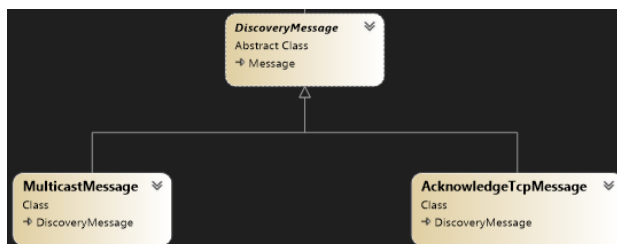


Рис. 3.2.6. Тип DiscoveryMessage та його прямі нащадки

Клас Request – володіє властивостями й методами, що використовують усі інші типи Kademlia-запитів.

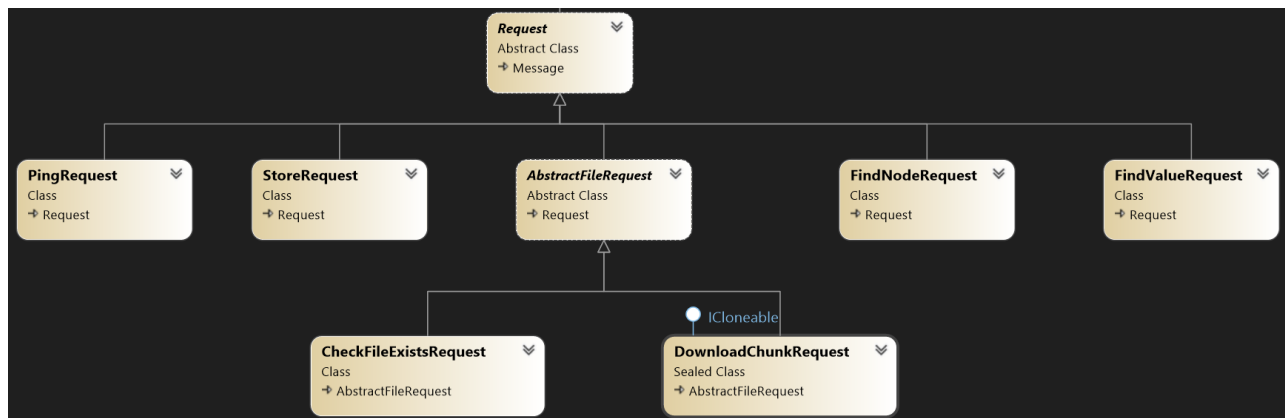


Рис. 3.2.7. Посилальний тип Request та його потомки

DownloadChunkRequest успадковує тип ICloneable, оскільки на кожен вузол при скачуванні потрібен окремий запит (детальніше див. «Скачування з локальної мережі»).

Клас Response – абстрактний клас для усіх видів Kademlia відповідей.

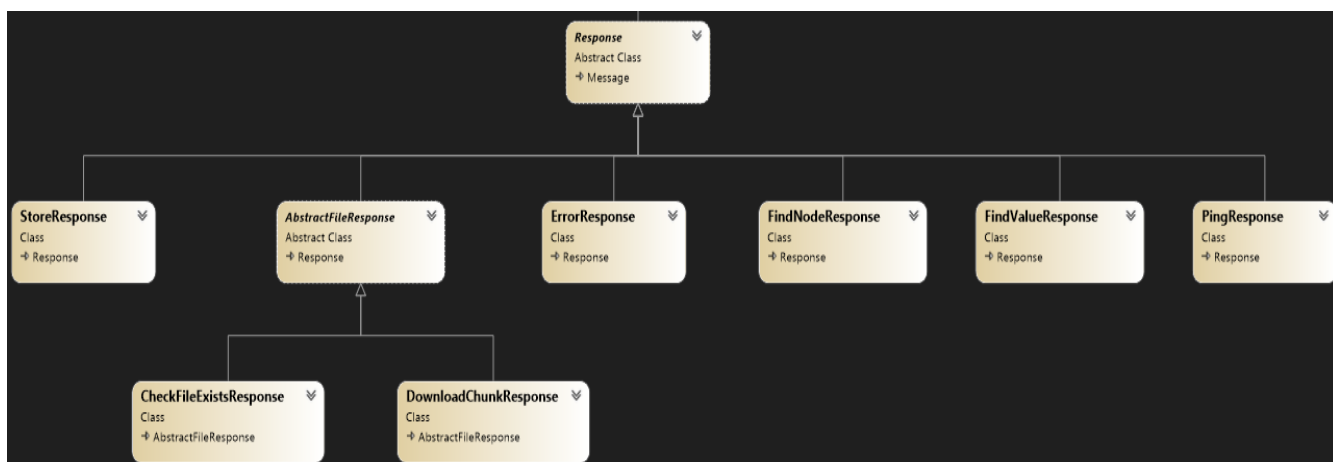


Рис. 3.2.8. Kademlia-відповіді на відповідні запити

Інші ієрархії класів представлені у наступному зображенні.

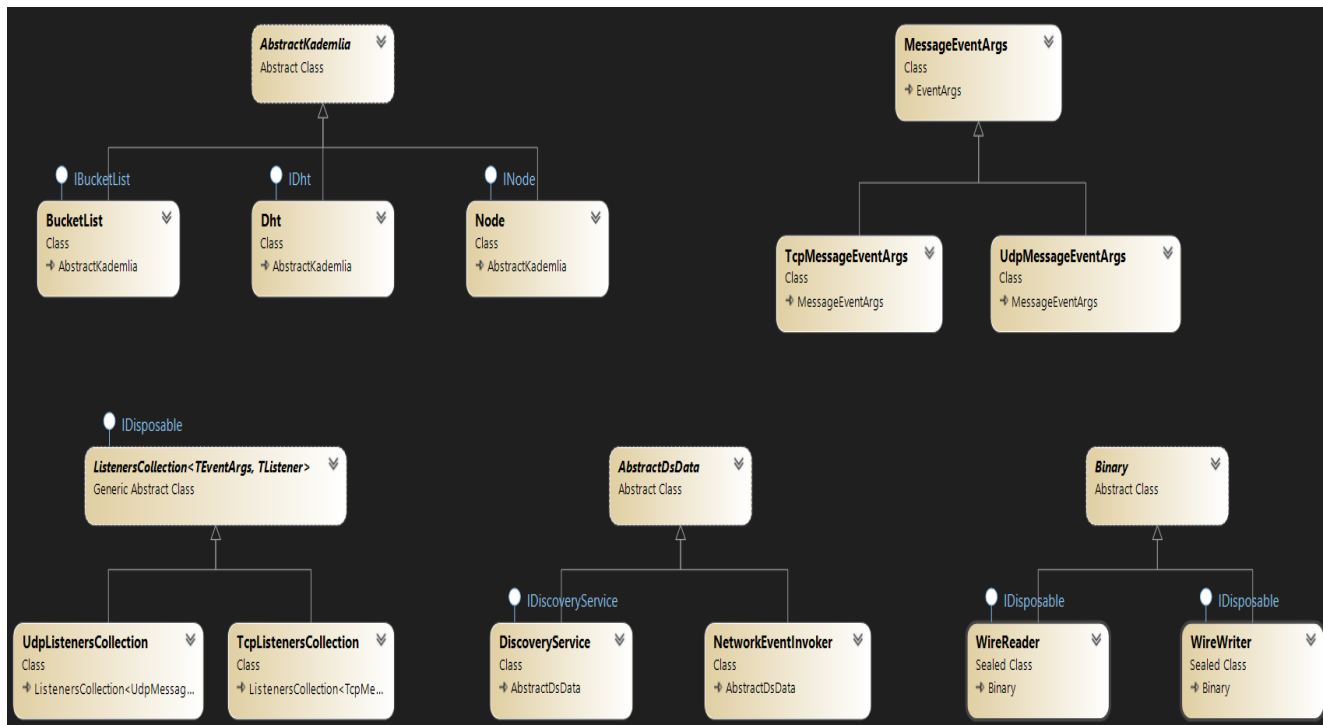


Рис. 3.2.9. Інші ієрархії класів

Отже, складену ієрархію класів мають лише об'єкти повідомлень, але це надає змогу легко додавати нові дані до них, проте дещо ускладнює рефакторинг. Інші успадкування надають змогу уникнути дублювання коду й не використовувати різноманітні патерни (наприклад, Абстрактну Фабрику [75]).

- Використання подій

Варто також наголосити, що проект DS побудований на подіях, що дає змогу легко розширювати класи не тільки розробнику, але й тому хто використовує сервіс (без порушення принципу інкапсуляції). Ось увесь набір подій:

- 1) знайдено новий мережевий інтерфейс;
- 2) отримано певне повідомлення (слухачі повідомлень просто зчитують їх, не визначають цілісність та не проводять певні перевірки);
- 3) отримано перевірене повідомлення сервісу виявлення (на кожен тип повідомлення існує окрема подія);
- 4) файл повністю скачаний;
- 5) файл частково скачаний;
- 6) під час скачування виникла помилка;
- 7) сервіс виявлення зупинено.

Також така реалізація спричиняє спрощення тестування сервісу.

3.3. Ініціалізація сервісу й налаштування брандмауера

DS реалізовує модифіковану версію патерну Одинак [50], згідно з якою створюється новий екземпляр поточного типу лише в тому випадку, якщо було задано параметри ініціалізації, що спричиняють потребу у конструюванні нового об'єкту. Для DS цим параметром є версія протоколу. За замовчуванням вона рівна 1.

Варто відмітити, що DS під час його ініціалізації також додає виконуваний файл до списку винятків Firewall-у для того, щоб мати можливість на будь-яких

портах обмінюватись повідомленнями в локальній мережі (без цього користувач сам вибиратиме, де дозволяти обмінюватись повідомленнями). Реалізацію FirewallHelper було частково взято зі зовнішнього ресурсу, проте модифіковано, оскільки вона не підтримувала дозвіл в обміні в локальній мережі. Реалізацію FirewallHelper див. в додатку 4.

3.4. Журналювання роботи сервісу

Для спрощення зміни журналювання (логування) було реалізовано static клас DsLoggerSet, який містить набір властивостей, що відповідають різноманітним логерам:

- консольному логеру (реалізований мною в LUC);
- логеру LUC;
- логеру за замовчуванням. На даний час він використовується при всіх логуваннях й ініціалізується конструктором класу DsLogger.

Розглянемо детальніше останнього. DsLogger успадковує клас ConsoleLogger (реалізовує інтерфейс ILoggingService), оскільки також логує певну інформацію в консоль. При його ініціалізації спершу створюється постфікс імен файлів, куди будуть логуватись повідомлення в залежності від їхнього типу, після чого отримується об'єкт-налаштування сервісу обміну файлами:

```
String logFileNamePostfix = $"DS-{DateTime.Today.ToString( format: "d" )}.Replace(
oldChar: '/', newChar: '.' )}";
SettingsService = AppSettings.ExportedValue<ISettingsService>();
```

Далі отримується конфігурація логера LUC й додається логування на сервер з вказівкою ОС поточного вузла, якщо виникла критична помилка.

```
LoggerConfiguration loggerConfig = LoggerConfigExtension.BaseLucLoggerConfig(
logFileNamePostfix, SettingsService.IsLogToTxtFile );
```

```
loggerConfig = loggerConfig.WriteTo.Sink( logEventSink: new LoggerToServer(
OsVersionHelper.Version() ), restrictedToMinimumLevel: LogEventLevel.Fatal )
```

Врешті-решт створюється логер на основі цих налаштувань:

```
m_configuredDsLogger = loggerConfig.CreateLogger();
```

Клас DsLogger перевизначає усі методи ConsoleLogger додаючи також виклик методів раніше сконфігурованого логеру. Наприклад, логування суттєвої помилки виглядає наступним чином:

```
public override void LogCriticalError( Exception exception )
{
    base.LogCriticalError( exception );

    m_configuredDsLogger.Fatal( exception, exception.Message );
    if ( exception.InnerException != null )
    {
        m_configuredDsLogger.Fatal( exception.InnerException,
exception.InnerException.Message );
    }
}
```

Як бачимо, логер DS на даний час записує інформацію в такі джерела: файли, консоль та віддалений сервер LUC-у. Клас DsLoggerSet надає можливість легко переключатись між способом логування, просто змінивши ініціалізацію властивості DsLoggerSet.DefaultLogger.

3.5. Запуск сервісу виявлення

Для запуску DS існує метод Start (див. додаток 5), який виконує ініціалізацію NetworkEventInvoker, додає до його подій обробники (детальніше про останнє див. «Реалізація обробки повідомлень»). Варто відзначити, що при новому знаходженні мережевого інтерфейсу (зазвичай стається лише у випадку під'єднання до мережі) відбувається оновлення пулу з'єднань, точніше повне закриття з'єднань й сокетів (їх не можна перевикористати, оскільки ОС закриває їх при від'єднанні від мережі), та створення нових.

```
//puts in events of NetworkEventInvoker sendings response
m_networkEventHandler = new NetworkEventHandler( this, NetworkEventInvoker,
CurrentUserProvider );

NetworkEventInvoker.QueryReceived += async ( invokerEvent, eventArgs ) =>
    await SendTcpMessageAsync( invokerEvent, eventArgs ).ConfigureAwait(
continueOnCapturedContext: false );
NetworkEventInvoker.AnswerReceived += NetworkEventInvoker.Bootstrap;
NetworkEventInvoker.NetworkInterfaceDiscovered += UpdateConnectionPoolAsync;

NetworkEventInvoker.Start();

m_distributedHashTable = NetworkEventInvoker.DistributedHashTable( ProtocolVersion );
```

Після цього в методі DS.Start, починається фонові періодична розсилка UDP multicast повідомлень, доки не буде знайдено хоча б 1 вузол мережі, що використовує DS.

```
if ( !m_isDsTest )
{
    TimeSpan intervalFindNewServices;
    #if CONNECTION_POOL_TEST
        intervalFindNewServices = TimeSpan.FromSeconds( value: 5 );
    #elif DEBUG
        intervalFindNewServices = TimeSpan.FromMinutes( 1 );
    #else
        intervalFindNewServices = TimeSpan.FromMinutes( 3 );
    #endif

    m_tryFindAndUpdateNodesTimer = new Timer( TryFindNewServicesTick, state: this,
        dueTime: TimeSpan.Zero, intervalFindNewServices );
}
```

Ця розсилка виконується періодично лише, якщо проект відмінний від DiscoveryService.Test. Це зроблено для зручності тестування.

Для визначення, як часто варто виконувати періодичну розсилку, використовуються символи директиви препроцесора. DS. При Debug режимі надсилається групова розсилка кожену хвилину (практичним шляхом було визначено, що це оптимальний час для надсилання самому собі повідомлень чи виявленні нових вузлів), при тестуванні пулу з'єднань - кожні 5 секунд, при реальній роботі додатку – кожні 3 хвилини.

3.5.1. Фільтрація мережевих інтерфейсів та IP-адрес

Варто особливу увагу приділити класу NetworkEventInvoker, оскільки саме він ініціалізує TCP- й UDP-сервери в нестатичному методі NetworkEventInvoker.Start, який в свою чергу викликає FindNetworkInterfaces (також викликається при зміні IP-адрес мережевого інтерфейсу). В останньому спершу визначаються мережеві інтерфейси, пакети яких можуть доходити до інших вузлів мережі:

```
IEnumerable<NetworkInterface> allTransmittableNetworkInterfaces =
    AllTransmittableNetworkInterfaces();
List<NetworkInterface> currentNics = InterfacesAfterFilter(
    allTransmittableNetworkInterfaces ).ToList();
```

В об'єкт `allTransmittableNetworkInterfaces` фільтруються за допомогою функції `NetworkEventInvoker.AllTransmittableNetworkInterfaces`, яка, в свою чергу, викликає метод-розширення `NetworkInterfaceExtension.TransmittableNetworkInterfaces`:

```
internal static class NetworkInterfaceExtension
{
    public static IEnumerable<NetworkInterface> TransmittableNetworkInterfaces( this
IEnumerable<NetworkInterface> networkInterfaces ) =>
        networkInterfaces.Where( nic =>
            ( nic.OperationalStatus == OperationalStatus.Up ) &&
            ( nic.NetworkInterfaceType != NetworkInterfaceType.Loopback ) );
}
```

Loopback інтерфейси (відносяться до маршрутизації назад до джерела; використовуються здебільшого для тестування передавання; спроможні до надсилання UDP-multicast іншим вузлам) не можуть брати участь в TCP-комунікації.

Метод `NetworkEventInvoker.InterfacesAfterFilter` має наступний вигляд:

```
private IEnumerable<NetworkInterface> InterfacesAfterFilter(
IEnumerable<NetworkInterface> interfaces ) =>
    m_networkInterfacesFilter?.Invoke( interfaces ) ?? interfaces;
```

Тут відфільтровуються інтерфейси згідно з фільтром, який був наданий клієнтом в метод `DiscoveryService.Start`, який далі передається в конструктор клас `NetworkEventInvoker`.

Після фільтрації інтерфейсів відбувається визначення, яких мережевих інтерфейсів більше немає, а яких нових було виявлено порівняно з минулим визначенням. Це потрібно для того, щоб знати, чи дійсно відбулись якісь зміни в мережі, оскільки вбудована в .NET статична подія `NetworkChange.NetworkAddressChanged` (яка й обробляється методом `NetworkEventInvoker.FindNetworkInterfaces`) спрацьовує на зміну кожної IP-адреси мережевого інтерфейсу. Після цього визначаються IP-адреси мережевих інтерфейсів, які можуть приймати участь в обміні повідомленнями:

```
List<IPAddress> ipAddressesOfFilteredInterfaces =
    ReachableIpAddressesOfInterfaces( m_filteredInterfaces ).ToList();
    ReachableIpAddressesOfInterfaces реалізований наступним чином:
```

```
private IEnumerable<IPAddress> ReachableIpAddressesOfInterfaces(
    IList<NetworkInterface> interfaces ) =>
    IpAddressesOfInterfaces( interfaces, UseIpv4, UseIpv6 ).
    Where( ip => ip.CanBeReachableInCurrentNetwork( interfaces ) );
```

Тут отримуються доступні в обміні пакетами IP-адреси інтерфейсів. Для визначення, чи IP-адреса є такою, використовується метод-розширення `IPAddressExtension.CanBeReachableInCurrentNetwork`. Він спершу визначає індекс мережевого інтерфейсу, який є доступним для обміну з наданою IP-адресою:

```
switch ( destinationAddress.AddressFamily )
{
    case AddressFamily.InterNetwork:
    {
        UInt32 destAddr = BitConverter.ToUInt32( destinationAddress.GetAddressBytes(),
startIndex: 0 );
        codeOfError = (UInt32)GetBestInterface( destAddr, out
interfaceIndex );

        break;
    }

    case AddressFamily.InterNetworkV6:
    {
        var sockaddr = new SockAddrIn6
        {
            ScopeId = (UInt32)destinationAddress.ScopeId,
            Addr = destinationAddress.GetAddressBytes(),
            Family = (UInt16)destinationAddress.AddressFamily
        };

        codeOfError = GetBestInterfaceEx(ref sockaddr, out Int32 ipv6InterfaceIndex);
        interfaceIndex = (UInt32)ipv6InterfaceIndex;

        break;
    }

    default:
    {
        codeOfError = ERROR_INVALID_PARAMETER;
        break;
    }
}
```

Методи `GetBestInterface` (для IPv4) й `GetBestInterfaceEx` (для IPv6) існують в `iphlpapi.dll` (IP Helper API – допоміжні методи для роботи з IP) й імпортуються в файл `IPAddressExtension`. Саме вони повертають індекс мережевого інтерфейсу.

Після цього виконується перебіг (за допомогою циклу `foreach`) крізь усі інтерфейси, що доступні в обміні, і якщо знайдено такий з ідентичним індексом,

то надана IP-адреса може обмінюватись з поточним вузлом пакетами, а також тими, що мають ідентичний мережевий інтерфейс.

Після того, як були визначені доступні в TCP-комунікації IP-адреси, вони оновлюються в контакт поточного вузла, а також у властивості `NetworkEventInvoker.ReachableIpAddresses`.

Далі виконується переініціалізація TCP- й UDP-слухачів повідомлень:

```
m_udpListeners?.Dispose();
m_tcpListenersCollection?.Dispose();

Boolean isConnectedToNetwork = ReachableIpAddresses.Count > 0;
if ( isConnectedToNetwork )
{
    m_udpSenders = new UdpSendersCollection( ReachableIpAddresses, RunningUdpPort );
    InitAllListeners();
}
```

В методі `Dispose` (утилізувати) цих об'єктів викликаються ідентичний методи створених (у випадку з TCP також прийнятих (використовуються для надсилання відповідей)) сокетів. Далі відбувається ініціалізація UDP відправників (див. підрозділ «Відправка групової розсилки»), TCP-слухачів й multicast отримувачів повідомлень (див. наступні підрозділи), якщо є хоча б 1 IP-адреса для обміну повідомленнями в мережі (це також є вказівкою, що пристрій підключений до мережі).

Після цього, якщо було виявлено новий мережевий інтерфейс, тобто, наприклад, коли відбулось підключення до мережі, то викликається відповідна подія:

```
if ( newNics.Any() )
{
    NetworkInterfaceDiscovered?.Invoke( this, new NetworkInterfaceEventArgs
    {
        NetworkInterfaces = newNics
    } );
}
```

Ця подія обробляється не статичним методом `DiscoveryService.UpdateConnectionPoolAsync` (див. підрозділ «Фонове відновлення з'єднань»).

Далі відбувається додання скидування й додання обробки події про зміну IP-адреси до підключеної мережі.

```
// Situation has seen when NetworkAddressChanged is not triggered
// (wifi off, but NIC is not disabled, wifi - on, NIC was not changed
// so no event). Rebinding fixes this.
NetworkChange.NetworkAddressChanged -= OnNetworkAddressChanged;
NetworkChange.NetworkAddressChanged += OnNetworkAddressChanged;
```

Метод `OnNetworkAddressChanged` лише виконує виклик `FindNetworkInterfaces`.

Таким чином DS також працюватиме за відсутності Інтернету, але підключенні до мережі.

3.5.2. Відправка групової розсилки

Розглянемо детальніше метод `UdpSendersCollection.ConfigureMulticastSocket`, який викликається в конструкторі цього класу до кожної переданої IP-адреси. Спершу встановлюється опція сокету `ReuseAddress` (як і у всіх сокетах DS-у, окрім тих, що в пулі TCP-з'єднань), яка надає змогу виконати `Bind` (зв'язування) з локальною IP-точкою (складається з порту й IP-адреси)), за винятком випадків, коли інший сокет активно слухає повідомлення за ідентичною адресою або тою, що слухає на усіх мережевих інтерфейсах:

```
multicastSocket.Client.SetSocketOption(
    SocketOptionLevel.Socket,
    SocketOptionName.ReuseAddress,
    optionValue: true
);
```

Решта налаштувань сокету описані за допомогою коментарів в додатку 6.

Для надсилання повідомлень до групової розсилки, використовується клас `MulticastOption` (для IPv4) або `IPv6MulticastOption` (для IPv6). В конструктор відповідного посилального типу потрібно вказати IP-адресу групи, як це зазначалось у підрозділі «UDP».

Відправка multicast повідомлень відбувається за допомогою методу `DiscoveryService.TryFindAllNodes`, який у свою чергу викликає

NetworkEventInvoker.SendMulticastsAsync, що формує об'єкт типу MulticastMessage й відправляє його за допомогою об'єкту m_udpSenders:

```
var multicastMessage = new MulticastMessage(
    messageId: (UInt32)m_random.Next( minValue: 0, Int32.MaxValue ),
    ProtocolVersion,
    RunningTcpPort,
    MachineId
);
Byte[] packet = multicastMessage.ToArray();
```

```
await m_udpSenders.SendMulticastsAsync( packet, ioBehavior, addressFamily
).ConfigureAwait( continueOnCapturedContext: false );
```

Змінна ioBehavior є типу IoBehavior й може приймати значення або Synchronous, або Asynchronous, тобто синхронне виконання чи асинхронне.

Метод UdpSendersCollection.SendMulticastsAsync відправляє певний пакет даних на кожну кінцеву точку групової розсилки, куди відповідний сокет є доданий:

```
foreach ( KeyValuePair<IPAddress, UdpClient> sender in m_sendersUdp )
{
    IPEndPoint endpoint = sender.Key.AddressFamily == AddressFamily.InterNetwork ?
        DsConstants.MulticastEndpointIpv4 : DsConstants.MulticastEndpointIpv6;
    switch ( ioBehavior )
    {
        case IoBehavior.Asynchronous:
        {
            await sender.Value.SendAsync(
                packet,
                packet.Length,
                endpoint
            ).ConfigureAwait( continueOnCapturedContext: false );
            break;
        }

        case IoBehavior.Synchronous:
        {
            sender.Value.Send( packet, packet.Length, endpoint );
            break;
        }

        default:
        {
            throw new ArgumentException( message: "Incorrect value", paramName: nameof(
ioBehavior ) );
        }
    }
}

#if TEST_WITH_ONE_IP_ADDRESS
    break;
#endif
}
```

Клас `UdpSendersCollection` також надає можливість відправляти лише на 1 групу повідомлення (використовується лише для зручності виконання функціональних тестів). Для цього потрібно розкоментувати перший рядок файлу «`UdpSendersCollection.cs`»:

```
///#define TEST_WITH_ONE_IP_ADDRESS
```

3.6. Слухання TCP-з'єднань й отримання серверних повідомлень

В класі `NetworkEventInvoker` за ініціалізацію усіх слухачів й отримувачів повідомлень поточного вузла відповідає метод `InitAllListeners`:

```
private void InitAllListeners()
{
    InitListeners( ProtocolType.Udp, messageReceivedHandler: HandleMulticastMessage,
out m_udpListeners );
    InitListeners( ProtocolType.Tcp, RaiseSpecificTcpReceivedEvent, out
m_tcpListenersCollection );
}
```

Класи `UdpListenersCollection` й `TcpListenersCollection`, типи яких мають поля `m_udpListeners` й `m_tcpListenersCollection` відповідно, реалізують абстракцію `ListenersCollection`. Вона розроблена таким чином, що можна доволі просто ініціалізувати однаковим чином слухачів.

Розглянемо реалізацію слухання UDP multicast повідомлень. В конструкторі класу `UdpListenersCollection` спершу додається спосіб конвертації з вбудованої в .NET структури `UdpReceiveResult`, яка містить прочитаний буфер даними, IP-адрес та порт відправника, в клас `UdpMessageEventArgs`, який, окрім цих властивостей, надає можливість конвертувати буфер в об'єкт повідомлення:

```
AppSettings.AddNewMap<UdpReceiveResult, UdpMessageEventArgs>();
m_mapper = AppSettings.Mapper;
```

Після цього створюються UDP-отримувачі повідомлень, які слухатимуть на відфільтрованих IP-адресах:

```
udpReceiver = new UdpClient( addressFamily );
udpReceiver.Client.SetSocketOption( SocketOptionLevel.Socket,
SocketOptionName.ReuseAddress, optionValue: true );
udpReceiver.Client.Bind( localEP: new IPEndPoint( listeningAddress, ListeningPort ) );
```

Далі відбувається створення й налаштування сокетів, що будуть слухати повідомлення, за допомогою методу `UdpListenersCollection.ConfigureListeners`. В залежності від адресної схеми (див. кінець підрозділу «TCP»), що визначаються параметри налаштування:

```
case AddressFamily.InterNetwork:
{
    multicastOption = new MulticastOption(
        DsConstants.MulticastAddressIPv4,
        listeningIpAddress
    );
    socketOptionLevel = SocketOptionLevel.IP;

    break;
}

case AddressFamily.InterNetworkV6:
{
    multicastOption = new IPv6MulticastOption(
        DsConstants.MulticastAddressIPv6,
        listeningIpAddress.ScopeId
    );
    socketOptionLevel = SocketOptionLevel.IPv6;

    break;
}

default:
{
    //only IPv4 and IPv6 is supported by UDP
    continue;
}
```

Й ці адреси підписуються на групову розсилку:

```
udpReceiver.Client.SetSocketOption( socketOptionLevel, SocketOptionName.AddMembership,
multicastOption );
```

Варто відмітити, що для слухання повідомлень варто також вказувати 2-ий параметр конструкторів класів `MulticastOption` й `IPv6MulticastOption`, оскільки тут ми встановлюємо ще який саме адрес хоче приймати multicast повідомлення. `ScopeId` – зона мережі, на яку поширюється відповідний адрес. Він рівний 0, коли IP-адрес дорівнює `::ffff:0:0` й надає змогу слухати на всіх мережевих інтерфейсах. Це не було б бажано для DS, оскільки зловмисники можуть надсилати на різноманітні інтерфейси повідомлення й легко перенавантажити вузли.

Ініціалізація TCP-з'єднань виконується в схожій манері, тільки відбувається запуск слухань з'єднань за допомогою методу `Socket.Listen` й немає підписки до `multicast`.

Для запуску слухання повідомлень використовується абстрактний метод `ListenersCollection.StartMessagesReceiving`, який перевизначається у класах-нащадках, які у випадку відсутності попереднього запуску слухання та утилізації об'єкту, в циклі запускають отримання повідомлень (метод `StartNextMessageReceiving`) на кожному попередньо налаштованому сокеті.

Наприклад, для отримання TCP-повідомлень спершу викликається метод `Task.Run`, який додає задачу (певна секція коду, що повинна бути виконана певним потоком) в пул потоків. Після чого запускається окрема задача отримання `TcpSession` (для UDP-отримання в даному місці запускається метод `listener.ReceiveAsync`) з певним повідомленням:

```
Task.Run( async () =>
{
    Task<TcpSession> task = listener.SessionWithMessageAsync();
```

Далі ініціалізується `callback` (зворотній виклик) по завершенню цієї задачі, тобто це той самий метод, що запускає `Task.Run`, так як і для UDP-отримання:

```
//StartNextMessageReceiving call is here in order not to stop find another session
with package when we received TCP message
_ = task.ContinueWith( x => StartNextMessageReceiving( listener ),
    TaskContinuationOptions.OnlyOnRanToCompletion |
    TaskContinuationOptions.RunContinuationsAsynchronously );
```

Це потрібно для того, щоб слухач постійно переглядав на існування нових повідомлень.

Після цього додається ще 1 зворотній виклик, який отримує `TcpSession` (у випадку UDP – отримується об'єкт `UdpReceiveResult`, який містить отриманий буфер й IP-endpoint відправника, після чого викликається подія `MessageReceived`, на чому обробка отриманого повідомлення в цьому класі завершується), тобто результат запущеного завдання, зчитує й встановлює дозвіл на використання сесії іншим потоком. Далі викликається подія про отримання повідомлення й

після цього очікується завершення усього завдання слухання (це стається лише при отриманні події, що отримувач повідомлень є утилізований).

```
//using tasks provides unblocking event calls
_ = task.ContinueWith( async taskReceivingSession =>
{
    TcpMessageEventArgs eventArgs = null;
    TcpSession tcpSession = await taskReceivingSession.ConfigureAwait(
        continueOnCapturedContext: false );

    try
    {
        receiveResult = tcpSession.Receive( DsConstants.ReceiveTimeout );
    }
    finally
    {
        tcpSession?.CanBeUsedByAnotherThread.Set();
    }

    InvokeMessageReceived( listener, receiveResult );
}, TaskContinuationOptions.OnlyOnRanToCompletion |
TaskContinuationOptions.RunContinuationsAsynchronously );

await task.ConfigureAwait( false );
```

Як бачимо, тут налаштування завдання відбувається лише в кінці методу, оскільки в іншому випадку, у нас не буде змоги викликати методи ContinueWith.

3.7. Реалізація обробки повідомлень

Це ще одна важлива частина DS, оскільки без правильної імплементації цього усі відправки запитів не матимуть очікуваної відповіді. Загалом будь-яке отримане повідомлення проходить наступні перевірки.

- А. Чи є отриманий буфер допустимого розміру. MulticastMessage має довжину близько 60 байтів, якщо виконується утиліта DS.Test – 88. ТСП-повідомлення можуть бути різноманітного розміру, але мінімальна довжина вважається 71 байт, максимальна – $1.5 * \text{максимальна довжина частини файлу (2 000 000 байт)}$ в пакеті.
- Б. Відповідність бінарному протоколу поточного повідомлення. Для перетворення об'єкту повідомлення в байти використовується метод Message.ToArray, який спершу серіалізує основний зміст

повідомлення, тобто усі властивості, окрім довжини, потім визначає останню й вказує її після першого байту. Для отримання повідомлення з байтів використовується конструктор, який приймає відповідні значення й встановлює значення властивостей об'єкту за допомогою відповідного методу `Message.Read`. Для кодування та декодування різноманітних властивостей використовуються власно реалізовані класи `WireWriter` та `WireReader` відповідно. Вбудовані в C# класи `BinaryWriter` та `BinaryReader` не дають змоги визначити кінець потоку.

- В. Чи відправник надіслав з діапазону DS-портів. Отримати достовірне значення порту можна лише, якщо воно є в повідомленні. До таких належать: `AcknowledgeTcpMessage`, `FindNodeRequest`, `FindNodeResponse` й `MulticastMessage`.
- Г. Чи ігнорувати нещодавні повідомлення з ідентичним ID(визначається випадковим чином). Зловмисник може отримати інформацію про пакет й навантажувати вузли дублікатами. Тому потрібно виконувати відповідну дію, якщо вибрана опція ігнорувати (за замовчуванням саме так і є). Нещодавними вважаються ті, що отримані в інтервалі 5 секунд, оскільки малоймовірно, що вузли, які не є зловмисниками, за такий час надішлють повідомлення з однаковим ID. У випадку KID це практично неможлива ситуація (див. підрозділ «Модульні тести»).

Почнемо з опрацювання отриманих UDP-дейтаграм. Послідовно воно обробляється наступними методами.

- Функцією `UdpClient.ReceiveAsync` в анонімному методі в `UdpListenersCollection.StartNextMessageReceiving`. Тут викликається подія `UdpListenersCollection.MessageReceived`.
- Обробником цієї події є `NetworkEventInvoker.HandleMulticastMessage`. Спершу він виконує вище описані перевірки, а також чи є надіслане

повідомлення поточного пристрою. Справа в тому, що при відправці UDP multicast також отримуються власні повідомлення, а їх немає сенсу обробляти, якщо DS не є запущений з метою тестування. Якщо перевірки повідомлення успішно пройдені, то в аргументах події встановлюється прочитане за допомогою бінарного протоколу повідомлення й викликається подія `NetworkEventInvoker.QueryReceived`. Вона є в загальному доступі.

- Першим методом у списку виклику цієї події є анонімна процедура, що викликає `DS.SendAcknowledgeTcpMessageAsync`, який бере сокет з TCP-пулу з'єднань й відправляє `AcknowledgeTcpMessage` у вигляді потоку байтів. Після цього відбувається повернення сокет в пул.

Той, хто програмно використовує DS має змогу додавати інші обробники подій, але, звісно, вони завжди будуть виконуватися після вище описаних.

Перейдемо до опрацювання TCP-повідомлень. Воно складається з наступних послідовних пунктів

- Отримання `TcpSession` з потоком байтів в анонімній функції в методі `TcpListenersCollection.StartNextMessageReceiving` за допомогою `TcpServer.SessionWithMessageAsync`;
- Зчитування повідомлення за допомогою методу `TcpServer.ReceiveAsync`.
- Виклику події `TcpListenersCollection.MessageReceived`.
- Обробником цієї події є `NetworkEventInvoker.RaiseSpecificTcpReceivedEvent`. Він виконує усі вище описані перевірки. Якщо вони були успішно пройдені, то в аргументах події встановлюється прочитане за допомогою бінарного протоколу повідомлення й викликається подія що відповідає певному повідомленню. Її

обробляє одна з реалізацій `INetworkEventHandler.SendResponse`. Усі ці події також у відкритому доступі.

Загалом інтерфейс `INetworkEventHandler` реалізують такі класи:

- A. `KadRpcHandler` – виконує Kademlia серверний процедурний виклик при отриманні відповідного Kademlia RPC.
- Б. `CheckFileExistsRequestHandler` – обробляє запит-опитування щодо існування файлу певного розміру й версії. Версія файлу записується у рядковому форматі, визначається за допомогою векторних годинників в LUC [54] й зберігається за використанням ADS.
- В. `DownloadChunkRequestHandler` – обробляє запити отримання певної частини файлу або ж його цілого, якщо його розмір є менший, ніж максимальний розмір чанку (2 МБ). У відповідь також надсилаються розмір та версія файлу, оскільки вони можуть в будь-який момент змінитись (версія файлу змінюється в ADS лише після виконання upload на сервер LUC-у).

Представимо одну з реалізацій вище зазначеного інтерфейсу. Для цього оглянемо клас `KadRpcHandler`, а точніше обробку `PingRequest`. При виклику події `NetworkEventHandler.PingReceived`, викликається метод `KadRpcHandler.SendResponse`, який в залежності від значення першого байту в запиті, тобто типу повідомлення виконує відповідну обробку.

```
case MessageOperation.Ping:
{
    PingRequest request = eventArgs.Message<PingRequest>( whetherReadMessage: false );
    var response = new PingResponse( request.RandomID );

    await HandleKademliaRequestAsync<PingRequest>(
        response,
        eventArgs,
        PriorityHandleKadRequest.FirstSendResponse,
        kadServerOp: ( sendingContact ) =>
        {
            if ( sendingContact != null )
            {
                m_node.Ping( sendingContact );
            }
        }
    );
}
```



```

    }
    ).ConfigureAwait( continueOnCapturedContext: false );
    break;
}

```

Як бачимо, спершу отримується значення запиту, далі формується відповідь й іде обробка отриманого повідомлення (ці кроки стосується кожної обробки Kademlia RPC-запиту). Обробка полягає лише у виконанні серверної операції й відправки відповіді за допомогою прийнятого сокету TcpServer-ом.

Варто також відокремити обробку FindNodeRequest. Окрім сказаних вище кроків також іде перевірка, чи даний контакт існує в DHT. Якщо – ні, то йде відправка PingRequest.

3.8. Алгоритм виявлення вузлів

Після того, як було запущено слухання TCP- й UDP-повідомлень й встановлено групи, до яких належить користувач LUC-у, починаються періодично відправлятися групові розсилки UDP, доки не буде знайдено хоча б 1 вузол. За подальший пошук відповідає протокол Kademlia. Після отримання вузлом UDP-паketу, воно спершу обробляється методом NetworkEventInvoker.HandleMulticastMessage, який виконує перевірки (див. підрозділ «Реалізація обробки повідомлень») одержаної дейтаграми.

Якщо після цих перевірок було визначено, що повідомлення отримано від іншого вузла, який використовує DS, то відбувається виклик DS.SendAcknowledgeTcpMessageAsync. Останній, в свою чергу, відправляє відповідне AcknowledgeTcpMessage у вигляді потоку байтів. Цей об'єкт-повідомлення містить все теж саме, що MulticastMessage за винятком того, що він ще включає KID поточного вузла, а також список ID груп, до яких належить користувач LUC-у. Отримувач AcknowledgeTcpMessage тобто той, хто відправив UDP multicast, виконує Kademlia Bootstrap, в ході якого надсилає FindNodeRequest. Якщо на нього було успішно одержано відповідь, то довершується процедура Bootstrap й вузол, що надіслав останню відповідь

вважається виявленим. Для оптимізації виявлення було вирішено також цим вузлом надсилати PingRequest й після цього спробувати додавати контакт до власної DHT (Distributed Hash Table – розподілена хеш таблиця, що містить увесь набір онлайн вузлів), якщо було отримано правильну відповідь. В інакшому випадку, вузол отримувач UDP-повідомлення зміг би дізнатись про відправника лише через 1 хвилину, якщо він не знайшов жодного вузла (тобто в ході UDP-розсилки), або ж через 20 хвилин (в ході періодичного оновлення K-бакетів поточного вузла).

3.9. Протокол оновлення та пошуку інформації в мережі

Більшість взятих частин з проекту Clifton.Kademlia з відкритим доступом не були змінені, оскільки в інакшому випадку можуть з'являтися проблеми, з якими вже стикались його розробники, або ж одна зміна може понести за собою велику кількість інших коректувань. Так при закритті певним вузлом додатку було вирішено не надсилати UDP multicast про відповідну подію, оскільки це б вносило значні зміни в оновлення K-бакетів.

- Реалізація Kademlia RPC

Clifton.Kademlia представляє клас TcpSubnetProtocol для представлення того, яким приблизно чином повинні відбуватись RPC-виклики, проте цей клас використовує протокол HTTP для взаємодії й певний сервер. Але через те, що DS має свої сервери (UDP й TCP), використовує інші мережеві протоколи й TCP-пул з'єднань, був створений клас KadLocalTcp.

Оглянемо саму реалізацію RPC-викликів. Вона міститься в класі Request, який є базовим для усіх Kademlia-запитів. Спершу отримується копія IP-адрес контакту (для уникнення конкурентних помилок й інкапсуляції відповідного списку в класі, що реалізовує IContact) й відбувається виконання циклу для спроби отримання відповіді від певної кінцевої точки вузла (їх перебір починається з останньої активної IP-адреси):

```

public async ValueTask<(TResponse, RpcError)> ResultAsync<TResponse>( IContact
remoteContact, IoBehavior ioBehavior, UInt16 protocolVersion )
    where TResponse : Response
{
    TResponse response = null;
    RpcError rpcError = null;

    List<IPAddress> clonedListOfIpAddresses = remoteContact.IpAddresses();

    //start from the last active IP-address and go to the oldest
    for ( Int32 numAddress = clonedListOfIpAddresses.Count - 1;
        ( numAddress >= 0 ) && ( response == null );
        numAddress-- )
    {
        var ipEndPoint = new IPEndPoint( clonedListOfIpAddresses[ numAddress
], remoteContact.TcpPort );
        (response, rpcError) = await
s_remoteProcedureCaller.PostAsync<TResponse>( this, ipEndPoint, ioBehavior
).ConfigureAwait( continueOnCapturedContext: false );
    }
    ...
}

```

remoteContact визначає тип RPC-виклику за значенням першого байту. Параметр ioBehavior вказує, яким чином повинен виконуватись метод на поточному вузлі: синхронно чи асинхронно. Якщо вдало виконався RPC-виклик, тобто було отримано відповідь, то вказується остання активна IP-адреса й встановлюється в значення 0 Eviction Count (використовується для визначення, чи варто видаляти контакт з відповідного списку (справджується, коли це значення рівне 3-м). Якщо відповідь не отримано, то йде спроба взаємодіяти з наступною IP-адресою, оскільки попередня могла мати занадто велику кількість запитів.

Розглянемо основне тіло методу RPC-виклику. Спершу визначається, чи може IP-адрес бути доступним в поточній мережі за допомогою реалізованого мною методу-розширення CanBeReachableInCurrent-Network (див. вище «Фільтрація мережевих інтерфейсів та IP-адрес»):

```

//we can change network, so it is better to check IP-address
//and don't wait result from unreachable IP-address
Boolean isSameNetwork = remoteEndPoint.Address.CanBeReachableInCurrentNetwork();
if ( !isSameNetwork )
{
    rpcError = new RpcError
    {
        ErrorMessage = $"{remoteEndPoint} is in different network"
    };
}
else

```

```
{
```

```
...
```

Якщо так, то йде спроба отримати під'єднаний сокет з пулу TCP-з'єднань.

Після чого йде очищення ще доступних даних в сокеті, щоб можна було зчитати потрібне повідомлення:

```
if ( client.Available > 0 )
{
    await CleanExtraBytesAsync( client, ioBehavior ).ConfigureAwait( false );
}
```

Очищення даних відбувається за рахунок їх зчитування:

```
//we use always async method, because ReceiveTimeout works only for async methods
Task<Int32> taskReceive = client.ReceiveAsync(
    buffer: new ArraySegment<Byte>( new Byte[ client.Available ] ),
    SocketFlags.None
);
if ( ioBehavior == IoBehavior.Asynchronous )
{
    await taskReceive.ConfigureAwait( continueOnCapturedContext: false );
}
else if ( ioBehavior == IoBehavior.Synchronous )
{
    AsyncContext.Run( async () => await taskReceive.ConfigureAwait( false ) )
}
```

Якщо за 5 разів не вдалося очистити всі дані, то скоріш за все сокет має з'єднання зі зловмисником й метод зупиняє свою роботу. В іншому випадку йде конвертація запиту в бінарний вигляд та відправка запиту (якщо не вдалося перший раз відправити, то створюється новий сокет з новим з'єднанням й знову йде відправка):

```
Byte[] bytesOfRequest = request.ToArray();
client = await client.DsSendWithAvoidNetworkErrorsAsync(
    bytesOfRequest,
    DsConstants.SendTimeout,
    DsConstants.ConnectTimeout,
    ioBehavior
).ConfigureAwait( false );
```

Далі відбувається логування відправленого запиту, якщо програма запущена в режимі Debug, й очікування на отримання відповіді:

```
Int32 countCheck = 0;
while ( ( client.Available == 0 ) &&
    ( countCheck <= DsConstants.MAX_CHECK_AVAILABLE_DATA ) )
{
    await WaitAsync(
        ioBehavior,
        DsConstants.TimeCheckDataToRead
    ).ConfigureAwait( false );
```

```
countCheck++;
}
```

Якщо за 10 секунд це не відбулось, то виконується скидування з'єднання, оскільки властивість `Socket.Connected` може далі вказувати, що зв'язок ще існує, хоч це може бути не так. TCP-пул з'єднань спробує відновити з'єднання.

Якщо вчасно відбулось зчитування бінарних даних, то на їх основі формується відповідь:

```
if ( countCheck <= DsConstants.MAX_CHECK_AVAILABLE_DATA )
{
    Byte[] bytesOfResponse = await client.DsReceiveAsync(
        ioBehavior,
        DsConstants.ReceiveTimeout
    ).ConfigureAwait( false );

    if ( bytesOfResponse[ 0 ] != (Byte)MessageOperation.LocalError )
    {
        response = (TResponse)Activator.CreateInstance(
            typeof( TResponse ),
            bytesOfResponse
        );
        ...
    }
}
```

Якщо перший байт вказує про те, що щось трапилось не так на віддаленому вузлі, то формується `ErrorResponse` на основі зчитаних даних, а також `RpcError`:

```
var nodeErrorResp = new ErrorResponse( bytesOfResponse );
rpcError = new RpcError
{
    IDMismatchError = request.RandomID != nodeErrorResp.RandomID,
    ErrorMessage = nodeErrorResp.ErrorMessage,
    RemoteError = true
};
```

Згідно з протоколом Kademlia кожен запит містить випадковий KID, у відповідь на який віддалений вузол повинен надіслати ідентичне його значення. за допомогою якого можна визначити, чи знає віддалений вузол протокол.

Далі будь-якому випадку відбувається повернення сокету в пул, якщо він був взятий з нього, й повернення відповіді та `RpcError`:

```
finally
{
    if ( client != null )
    {
        await client.ReturnToPoolAsync(
            DsConstants.ConnectTimeout,
            ioBehavior
        ).ConfigureAwait( false );
    }
}
```

```
return (response, rpcError);
```

Тобто даний метод винятків не кидає. Всі вони обробляються у на основі них створюється RpcError.

- Оновлення KID при перезапуску програми. Причина створення класу MachineId

Кожен контакт має конкретний KID лише на час роботи програми. Тому якщо користувач вимкне додаток, а потім через деякий час знову запустить й матиме ті ж самі IP-адреси (можуть змінитись лише при перезавантаженні маршрутизатора), а він не буде видалений з бакетів Kademlia в інших вузлах, то це призведе до того, що користувач буде обмінюватись TCP-повідомленнями з самим собою через процедуру обміну контактами (див. підрозділ «Протокол оновлення інформації про вузли мережі»), а також до дуплікації контактів в інших вузлах. Саме тому було створено клас MachineId, який надає зашифрований ідентифікатор машини базуючись на ID процесора та серійному номері материнської плати:

```
public static String Create()
{
    var deviceIdBuilder = new DeviceIdBuilder();

    String encodedMachineId = deviceIdBuilder.
        AddProcessorId().
        AddMotherboardSerialNumber().
        ToString();

    return encodedMachineId;
}
```

Цей рядок надсилається у кожному Kademlia-запиті для визначення відправника.

3.10. Пул клієнтських TCP-з'єднань

Це важливий пункт DS, оскільки значно пришвидшує обмін TCP-повідомленнями, особливо коли буде додано SSL/TLS-підтримку. Пул клієнтських TCP-з'єднань — набір закешованих клієнтських TCP-з'єднань з TcpServer-ами віддалених вузлів. Варто відмітити, щоб об'єкти типу Socket варто

використовувати лише, якщо реалізовується пул з'єднань або ж інший спосіб їх підтримки чи зберігання сокетів. Для інших випадків краще завжди зміщувати фокус на класи-обгортки посилаального типу Socket (наприклад, TcpClient) . На даний час ConnectionPool клас підтримує лише TCP, оскільки DS потребує зберігати сокети лише з таким типом протоколу.

3.10.1. Особливості реалізації

Розглянемо детальніше його особливості реалізації, оскільки цей клас є досить складним через потребу високої швидкодії та безпеки роботи DS.

- 1) Реалізація шаблону Одинак й приналежність усіх полів до конкурентних класів.

Це зроблено з метою отримання одного об'єкту pool-у в різних частинах коду й спрощення його можливих майбутніх змін, наприклад, підтримка кількох сокетів для з'єднання з 1 кінцевою точкою.

```
private ConnectionPool( ConnectionSettings connectionSettings )
{
    ConnectionSettings = connectionSettings;
    if (ConnectionSettings.ConnectionReset)
    {
        BackgroundConnectionResetHelper.Start();
    }

    m_cleanSemaphore = new SemaphoreSlim(initialCount: 1);
    m_socketSemaphore = new SemaphoreSlim(connectionSettings.MaximumPoolSize);
    m_lockTakeSocket = new AsyncLock();

    m_sockets = new ConcurrentDictionary<EndPoint, Socket>();
    m_leasedSockets = new ConcurrentDictionary<EndPoint, Socket>();

    //cached tasks is for optimization, otherwise we will create a lot of the same
    ImmutableDictionary<Boolean, ValueTask<Boolean>>.Builder cachedValueTasksBuilder =
        ImmutableDictionary.CreateBuilder<Boolean, ValueTask<Boolean>>();
    cachedValueTasksBuilder.Add(key:true,value:new ValueTask<Boolean>(result:true));
    cachedValueTasksBuilder.Add( false, new ValueTask<Boolean>( false ) );
    m_booleanValueTasks = cachedValueTasksBuilder.ToImmutableDictionary();

    m_cancellationRecover = new CancellationTokensource();
}
public static ConnectionPool Instance
{
    get
    {
        SingletonInitializer.ThreadSafeInit( value: () => new ConnectionPool(
            ConnectionSettings() ), ref s_instance );
    }
}
```

```

        return s_instance;
    }
}
public ConnectionSettings ConnectionSettings { get; }

```

Як бачимо, даний клас містить усі поля, що так чи інакше пов'язані з конкурентністю:

- **ConcurrentDictionary** – потокобезпечний словник, який забезпечує дуже зручний API для роботи з словником. Усі його методи є потокобезпечними, в тому числі надає можливість безпечно виконувати цикл `foreach` з його ключами і значеннями.
- **SemaphoreSlim** - локальний семафор, тобто обмежувач кількості потоків в одному процесі (програмі), які можуть одночасно звертатися до ресурсу чи пулу ресурсів [53].
- **AsyncLock** – замок, що надає можливість синхронно й асинхронно очікувати його звільнення. Детальніше див. «Вибір пакетів для частих потреб при роботі з асинхронним шаблоном на основі завдань».
- **ImmutableDictionary** – незмінний словник. Як вказано в коментарі, використовується для зберігання `ValueTask` з ціллю оптимізації. Загалом останній тип заощаджує пам'ять порівняно з `Task`, але очікування його результату є дещо повільнішим.

2) Фонове відновлення з'єднань

З метою оптимізації при від'єднанні й приєднанні до мережі, клієнту пулу слід запускати скидування усіх з'єднань та встановлення нових, оскільки ОС повинна закривати сокети при від'єднанні.

3) Налаштування ConnectionPool

Клас `ConnectionSettings` містить властивості для налаштування пулу. Ці методи доступу на даний час не можуть змінюватись. Вони вибирались на основі `opensource MySqlConnectionConnector` [32]. На даний час використовуються такі властивості:

- `MaxCountSocketInUse` – максимальна кількість потоків, що можуть одночасно використовувати різні сокети пулу. Було вибрано значення 16 для цієї властивості. 4 – максимальна кількість потоків, що виконують скачування одного файлу (на даний час в LUC максимум 1 файл може скачуватись в 1 момент часу), решта використовується, коли отримуються multicast (для надсилання опису вузла-отримувача) й ще інші – для надсилання Kademlia-запитів. Це значення не варто збільшувати для обмеження навантаження на поточний й інші вузли й за дослідженнями вибране значення підходить для вище вказаних цілей.
- `ConnectionBackgroundReset` – вказує, чи `ConnectionPool` підтримує скидування з'єднань у фоні (див. нижче відповідний підрозділ). На даний час встановлено в `true`.
- `ConnectionTimeout` – максимальна тривалість очікування створення TCP з'єднання (3-way handshake).

4) На кожне віддалене з'єднання з конкретною віддаленою точкою `ConnectionPool` може містити максимум 1 сокет.

Звичайно, якщо сокет застосовується іншим потоком, то можна було б створювати інший відповідний програмний інтерфейс, але DS-у важливо обмежувати навантаження на інші вузли мережі, оскільки, як було вказано вище, сервіс обміну файлами здебільшого використовують ті користувачі, які працюють з великою кількістю інших програм.

5) 1 сокет може використовувати максимум 1 потік

Це найважливіша й найскладніша особливість пулу, оскільки при багатопоточності, якщо 1 й той самий сокет використовуватимуть різні потоки, то це може призвести до випадку, коли кілька потоків зчитуватимуть один й той самий потік даних.

б) Використання власноруч реалізованого класу Socket

Його створено через те, що потрібно зберігати деякі властивості та дані в залежності від його роботи, використання в пулі та за його межами. Він успадковує клас `AsyncSocket`, який також є мною реалізований з метою оптимізованої роботи в мережі, визначення поточного стану сокету (наприклад, відправляє байти на віддалену точку), а також надання можливості відмінити виконання загальноживаних методів та легко змінювати максимальний час створення з'єднання чи виконання інших операцій. Вбудований в С# клас `Socket`, якого власне успадковує посилальний тип `AsyncSocket`, не надає таких можливостей, що є його суттєвим недоліком в сучасних реаліях розробки.

Вказані властивості особливості спричинили велику кількість коду, через що клас `ConnectionPool` був розподілений на кілька файлів.

7) Розбиття класу ConnectionPool на файли.

Мова С# підтримує дану опцію за рахунок ключового слова `partial`. Цей клас міститься у наступних файлах:

- «`ConnectionPool.cs`».

Це найголовніший файл, який містить усі поля загального використання в цьому класі, властивості, а також методи взяття сокету (`SocketAsync`) й повернення його в пул (`ReturnToPoolAsync`).

- «`ConnectionPool.Socket.cs`».

Містить внутрішній клас `Socket` з метою інкапсуляції та вище описаних причин.

- «`ConnectionPool.Socket.StateInPoolReference.cs`».

Тут розміщується внутрішній відносно класу `Socket` закритий клас `StateInPoolReference`. Він потрібен для випадку утилізації старого сокету й створенні нового. Через те, що це посилальний тип, зміна його властивостей призводитиме до змін усіх об'єктів, що посилаються на нього, таким чином

оновлюватиметься стан усіх сокетів (утилізованих та новостворених) в пулі. В залежності від цього стану визначається, чи може потік брати сокет. Цей стан впливає на визначення, як працювати з об'єктами блокувань сокету. Ці змінні в свою чергу визначають доступ до використання сокету.

- «`ConnectionPool.GetConnectedSockets.cs`».

Містить лише приватні методи для створення, перестворення (випадок, коли сокет є утилізований) й взяття сокету з пулу, а також перевірки з'єднання і його відкриття з віддаленою точкою.

- «`ConnectionPool.RecoverConnections.cs`».

Цей файл має 2 публічні методи:

- `TryRecoverAllConnectionsAsync` – асинхронне надання фоновому відновлювачу з'єднань (static класу `BackgroundConnectionResetHelper`) усіх сокетів пулу для відповідних цілей. Якщо сокет використовується іншим потоком, то відбувається очікування на його повернення в пул й лише тоді передання в `BackgroundConnectionResetHelper`.
- `TryCancelRecoverConnections` – відміна попереднього оновлення з'єднань.

Детальніше про цей файл див. підрозділ «Фонове відновлення з'єднань».

- «`ConnectionPool.Clear.cs`».

Міститься лише 1 відкритий метод – `ClearPoolAsync`, який відповідає за закриття усіх сокетів пулу. Даний метод використовується лише при зупинці DS.

3.10.2. Взяття сокету з пулу.

Цей метод є найосновнішим для користувачів пулу, тому розглянемо детальніше його. Ця функція починається з перевірки за допомогою локального семафору, скільки потоків зараз взаємодіють з пулом, тобто використовують певний його сокет або ж очікують на його взяття. Для цього викликається метод `CanSocketBeTakenFromPoolAsync`.

```

private ValueTask<Boolean> CanSocketBeTakenFromPoolAsync(
    IoBehavior ioBehavior,
    TimeSpan waitTimeout )
{
    if ( ioBehavior == IoBehavior.Asynchronous )
    {
        return new ValueTask<Boolean>(
            task: m_socketSemaphore.WaitAsync( waitTimeout ) );
    }
    else
    {
        Boolean successWait = m_socketSemaphore.Wait( waitTimeout );
        return m_booleanValueTasks[ successWait ];
    }
}

```

Було вирішено виконувати кешування ValueTask<Boolean> для оптимізації, оскільки даний метод часто викликається, й представлення, як варто зберігати й використовувати закешовані завдання. Також тут не присвоюється окремій змінній повертаюче значення, оскільки це може призвести до непередбаченої поведінки у випадку асинхронного виконання методу. Якщо даний метод повертає ValueTask зі значенням false, то кидається виняток InvalidOperationException в методі SocketAsync. В іншому випадку відбувається перевірка, чи взятий сокет з потрібним з'єднанням іншим потоком:

```

//we need to try to take socket also here in order to don't wait while
m_lockTakeSocket is released
(Boolean isTaken, Socket desiredSocket) = await TryTakeLeasedSocketAsync(
    remoteEndPoint,
    ioBehavior,
    timeWaitToReturnToPool,
    cancellationToken
).ConfigureAwait( false );

```

Якщо він був успішно взятий, то виконується перевірка з'єднання і відновлення його при потребі (якщо сокет був утилізований, то створюється новий з новим з'єднанням):

```

desiredSocket = await TakenSocketWithRecoveredConnectionAsync(
    remoteEndPoint,
    timeoutToConnect,
    ioBehavior,
    desiredSocket
).ConfigureAwait( false );

```

Якщо сокет не використовувався іншим потоком, то відбувається взаємоблокування взяття усіх незадіяних сокетів з метою відсутності створення

дублікатів (випадок, коли 2 потоки одночасно хочуть взяти сокет з пулу, який не застосовується ніким):

```
//the socket with the same ID can be taken from the pool by another
//thread while current was waiting for releasing m_lockTakeSocket
using ( await m_lockTakeSocket.LockAsync( ioBehavior ).ConfigureAwait( false ) )
{
    desiredSocket = await CreatedOrTakenSocketAsync(
        remoteEndPoint,
        timeoutToConnect,
        timeWaitToReturnToPool,
        ioBehavior
    ).ConfigureAwait( false );
}
```

Після створення нового чи взяття раніше створеного сокету зі з'єднанням звільняється замок (це стається у будь-якому випадку завдяки використанню конструкції using) й повертається потрібний сокет клієнту.

Загалом ConnectionPool обробляє різноманітні ситуації, що пов'язані з сокетами й не надає їх лише у випадках, коли не можливо з'єднатись з віддаленою точкою або занадто багато потоків намагаються взаємодіяти з пулом (якщо уникати цього випадку, то це може призвести до повільної роботи усього пристрою).

3.10.3. Повернення сокету в пул

Цей метод також є дуже важливим, оскільки без повернення сокету в пул, інший потік завжди чекатиме на повернення протягом зазначеного часу очікування.

Спершу перевіряється, чи сокет з відповідним з'єднанням існує в потокобезпечному словнику m_leasedSockets:

```
Boolean wasTakenFromPool = m_leasedSockets.ContainsKey( socket.Id );
```

Якщо ні - значить щось не так в реалізації пулу (протягом останніх місяців такого випадку не траплялось за винятком виклику наступного методу) або ж був викликана функція ConnectionPool.CleanPoolAsync, яка звільняє усі некеровані ресурси й видаляє всі елементи з словників. Якщо сокет є в словнику m_leasedSockets, то перевіряється його загальний стан. За допомогою

властивості `ConnecitonPoolSocket.Health`, яка дає змогу визначити, чи сокет має значення методу доступу `Connected` в `false` (`IsNotConnected`), чи утилізований або зовсім не вдається створити з'єднання(`Expired`), чи ні те, ні інше, тобто під'єднаний (`Healthy`).

```
public SocketHealth Health
{
    get
    {
        SocketHealth socketHealth;

        try
        {
            VerifyConnected();
        }
        catch ( SocketException )
        {
            socketHealth = SocketHealth.IsNotConnected;
            return socketHealth;
        }
        catch ( ObjectDisposedException )
        {
            ;//do nothing, because check whether it is disposed is in ( m_state >=
SocketState.Failed )
        }

        SocketStateInPool stateInPool = StateInPool();
        socketHealth = ( stateInPool == SocketStateInPool.IsFailed ) || ( State >=
SocketState.Failed ) ?
            SocketHealth.Expired :
            SocketHealth.Healthy;

        return socketHealth;
    }
}
```

Якщо сокет не є `Healthy`, то виконується метод `ConnectedSocketAsync`, для отримання відповідного, оскільки потрібно `ConnectionPool` повинен тримати саме під'єднані сокети. Після цього йде перевірка, чи вдалося все-таки встановити з'єднання. Якщо ні, то виконується утилізація сокету й повернення його в пул:

```
SocketStateInPool currentStateInPool;
if ( isConnected )
{
    currentStateInPool = SocketStateInPool.IsInPool;
}
else
{
    socket.DisposeUnmanagedResources();
    currentStateInPool = SocketStateInPool.IsFailed;
}
```

```
}
```

```
InternalReturnSocket( socket, currentStateInPool, out isReturned );
```

Метод `InternalReturnSocket` виконує послідовне повернення в пул для правильного його взяття іншими потоками. Він виглядає наступним чином.

```
private void InternalReturnSocket(
    Socket socket,
    SocketStateInPool newState,
    out Boolean isReturned )
{
    //first we need to add socket to the pool in order to it always
    //was in any dictionary and another threads don't create new socket
    m_sockets.AddOrUpdate(
        socket.Id,
        addValueFactory: socketId => socket,
        updateValueFactory: ( socketId, oldSocket ) => socket
    );

    //socket will be not successfully returned to the pool
    //if it doesn't exist in m_leasedSockets
    isReturned = m_leasedSockets.TryRemove( socket.Id, value: out _ );

    socket.AllowTakeSocket( newState );
#if CONNECTION_POOL_TEST
    if ( isReturned )
    {
        DsLoggerSet.DefaultLogger.LogInfo( logRecord:
            $"Socket with ID {socket.Id} successfully returned in the pool" );
    }
#endif
}
```

В цьому методі важливо, щоб спершу було додано сокет в `m_sockets`, щоб інший потік не створював новий, що б порушило ідею пулу, тобто 1 сокет на 1 з'єднання.

Після виклику методу `InternalReturnSocket` виконується збільшення на 1 лічильника об'єкту `m_socketsSemaphore`, тобто іде вказівка, що потік не використовує якийсь сокет з `ConnectionPool`. Якщо сокет не був повернений в пул, тобто не існував в `m_leasedSockets` під час роботи методу `ReturnToPoolAsync`, то сповіщається відповідне повідомлення. В залежності від конфігурації DS це сповіщення може бути різним (кидання винятку у випадку `ConnectionPoolTest` або ж просто його додання в лог при всіх інших налаштуваннях).

```
ReleaseSocketSemaphore();

if ( !isReturned )
{
    NotifyPoolError( message:
        "Try to return socket which already removed from the pool" );
}
```

3.10.4. Фонове відновлення з'єднань

Як зазначалося вище, для цього був створений окремий файл «ConnectionPool.RecoverConnections.cs» й static клас BackgroundConnectionResetHelper, який був реалізований на основі прикладу opensource MySqlConnectionConnector [32]. Розглянемо детальніше вище вказаний файл.

Відновлення з'єднань у фоні складається з 4 важливих кроків.

- Відміна попереднього процесу скидування

Пристрій може кілька разів перепідключатись до різних мереж за короткий проміжок часу, тому варто відмінити попередній процес відновлення з'єднань й заново його починати (ОС закриває сокети при від'єднанні від мережі).

```
public void CancelRecoverConnections() =>
    m_cancellationRecover.Cancel();
```

- Оновлення параметрів скидування з'єднань

Єдиним способом переведення об'єкту типу CancellationTokenSource з стану відміни в протилежний, є його переініціалізація.

```
private void UpdateRecoveryPars() =>
    Interlocked.Exchange( ref m_cancellationRecover, value: new
        CancellationTokenSource() );
```

Тобто тут відбувається потокобезпечне встановлення нового значення цього поля.

- Відновлення з'єднань сокетів, що зараз використовуються іншими потоками

Для цього існує метод IdsOfRecoveredConnectionsInLeasedSockets. Він використовує шаблон Producer/Consumer для пришвидшення виконання методу: поточний (producer) потік надає сокети, з'єднання яких потрібно оновити, а інші потоки (consumers) очікують на повернення сокету в пул, перевіряють з'єднання

й у випадку його відсутності чи не справності сокету, надають його BackgroundConnectionResetHelper для оновлення з'єднання.

```
Parallel.ForEach( m_leasedSockets, ( leasedSocket ) => recoverSockets.Post(
leasedSocket ) );

//Signals that we will not post more leasedSocket.
//recoverSockets.Completion will never be completed without this call
recoverSockets.Complete();

try
{
    //await completion of adding to
    //BackgroundConnectionResetHelper all leased sockets
    await recoverSockets.Completion.ConfigureAwait( false );
}
catch ( OperationCanceledException )
{
    ;//do nothing
}
```

Сам процес обробки кожного сокету виглядає наступним чином:

```
var socketsWithRecoveredConnections = new ConcurrentBag<EndPoint>();

var recoverLeasedSockets = new ActionBlock<KeyValuePair<EndPoint, Socket>>(
    async socket =>
    {
        //TODO use cancellation token argument
        (Boolean isTaken, Socket takenSocket) = await
            TryTakeLeasedSocketAsync(
                socket.Key,
                IoBehavior.Asynchronous,
                timeWaitToReturnToPool
            ).ConfigureAwait( continueOnCapturedContext: false );

        if ( isTaken )
        {
            BackgroundConnectionResetHelper.AddSocket(takenSocket, cancellationToken);

            socketsWithRecoveredConnections.Add( socket.Key );
        }
    },
    parallelOptions
);
```

Тобто спершу очікується на взяття сокету, що зараз застосовується іншим потоком. Якщо він був успішно взятий (протилежний результат виникає лише у випадку, коли сокету зовсім не було в m_leasedSockets), то він надається класу BackgroundConnectionResetHelper для відновлення з'єднання (він викликає метод Socket.TryRecoverConnectionAsync й зберігає в окремому списку завдань). Після цього в змінну socketsWithRecoveredConnections поміщається ID

віддаленої точки для того, щоб при наступному пунктів вже не відновлювалось відповідне з'єднання.

- Відновлення з'єднань сокетів, що зараз не використовуються

Для виконання цього відбувається ідентичне, як в попередньому пункті, тільки з сокетами, що не застосовуються іншими потоками й у них з'єднання не було відновлене.

3.11.Скачування файлів з локальної мережі

Для даної операції існує клас `DownloaderFromLocalNetwork`. Розглянемо ініціалізацію відповідних об'єктів. Для його ініціалізації існує конструктор наступного вигляду.

```
public DownloaderFromLocalNetwork(
    IDiscoveryService discoveryService,
    IoBehavior ioBehavior,
    Int64 minFreeDriveSpace = 100000000 )
{
    m_downloadingFile = new DownloadingFile();
    m_lockWriteFile = new Object();

    m_discoveryService = discoveryService;
    m_ourContact = discoveryService.OurContact;

    m_currentUserProvider = m_discoveryService.CurrentUserProvider;

    IOBehavior = ioBehavior;

    m_minFreeDriveSpace = minFreeDriveSpace;
}
```

Він ініціалізує поля та властивості поточного об'єкту:

- Змінна `m_downloadingFile` містить тільки методи (з метою забезпечення потокобезпечності), що пов'язані з отриманням інформації про файл, що скачується, й встановленням його атрибутів;
- `m_lockWriteFile` – відповідає за синхронізацію програмних потоків з метою правильного запису у файл, оскільки з кількох вузлів скачується файл, то часто потрібно записувати в різні його частини

байти. Для цього потрібно спершу змінювати поточну позицію потоку файлу;

- `m_discoveryService` – потрібен з метою отримання інформації про віддалені вузли;
- `m_ourContact` – містить дані про поточний вузол. Використовується з метою надсилання деяких з них у запитах (KID та ID пристрою);
- `m_currentUserProvider` – надає змогу поточну інформацію про папку синхронізації;
- `IOBehavior` – визначає спосіб скачування файлів – синхронний чи асинхронний;
- `m_minFreeDriveSpace` – мінімальний вільний простір на диску, що завжди повинен бути. Якщо розмір файлу, що повинен скачатись здатен обмежити вільний простір на розмір більший, ніж ця змінна, то кидатиметься виняток `NotEnoughDriveSpaceException`. За замовчуванням ця змінна рівна 100 МБ, оскільки саме такий розмір часто використовують інші програми, наприклад, Video Inspector [57].

Загалом клас `DownloaderFromLocalNetwork` розділяє файли для скачування на 2 типи: малий файл – його розмір \leq максимальний розмір чанку, великий файл – максимальний розмір чанку $<$ розмір файлу.

При скачуванні будь-якого файлу виконуються наступні пункти.

- Спершу отримуються розпізнані онлайн контакти за допомогою методу `IDiscoveryService.OnlineContacts` й вибираються ті, які належать до групи, яка містить потрібний файл.
- Запуск асинхронного завдання для визначення контактів, що мають файл, за допомогою методу `ContactsWithFile` (див. додаток 7). Під час цього надсилається запит, що містить hex префікс (закодований шлях у шістнадцятковому вигляді від папки синхронізації й певної

групи (на кожен групу, до якої належить користувач LUC існує окрема папка) до скачуваного файлу)), назву файлу й ID групи для покращення безпеки скачування.

- Перевірка вільного простору на диску;
- Отримання потоку файлу для скачування. Він отримується за допомогою методу `FileExtensions.FileStreamForDownload`, який встановлює атрибути файлу до `FileAttributes.Normal` (стандартний файл, що не має особливих атрибутів й може використовуватись лише одним користувачем), якщо попередньо цей файл існував, створює або перезаписує файл за вказаним шляхом й вказує нові атрибути до файлу (тільки схований, щоб користувач не бачив (за умов, якщо в нього не вказано бачити сховані елементи) ще не скачаний файл).
- Встановлення довжини файлу для того, щоб зарезервувати простір на диску й мати можливість скачати повністю файл.
- Після скачування файлу, знімається з нього атрибут «Схований» й переноситься з «LUC temp files» у потрібне місце (якщо такий файл там існує, то він спершу видаляється). Якщо файл не вдалося перемістити (наприклад, користувач відкрив той, який потрібно видалити), то виконується запис в ADS, що він скачаний (для випадку, коли додаток закрився), й додається в `FileChangesQueue` для того, щоб він перемістив його при наступній синхронізації з серверу.

При скачуванні малого файлу почерзі визначених контактів з файлом відбувається спроба його скачати з них, поки це не буде зроблено або ж поки не буде пройдено по всьому списку.

Якщо потрібно скачати великий файл, то окрім вище описаних пунктів виконуються наступні.

- Отримання набору усіх контактів з запитом скачування та властивостями частин файлу (початок; кінець; сумарна кількість байтів, що будуть скачуватись з поточного контакту, увесь розмір), що будуть вказуватись у відповідних запитах;
- встановлення атрибуту `sparse` для файлу, що надає змогу записувати байти в різноманітні частини;
- запуск й очікування завершення паралельної процедури скачування за допомогою класу `ActionBlock`. Максимальна кількість використаних потоків при цьому рівна 4;
- визначення, чи файл повністю скачаний. Якщо – ні, то отримується оновлений новий набір усіх контактів з запитом скачування та властивостями частин файлу (під час нього йде повторне опитування на існування потрібного файлу в контактах, з яких не вдалося визначені частини);
- видалення атрибуту «схований» зі скачаного файлу.

Отже, скачування файлів є доволі оптимізованим й надає можливість скачати потрібний файл, збираючи його послідовно з різноманітних вузлів мережі.

Варто також описати винятки, що метод `DownloaderFromLocalNetwork.DownloadFileAsync` може кинути.

Таблиця 2

Види винятків роботи методу `DownloaderFromLocalNetwork.DownloadFileAsync` та їх опис.

Виняток	Опис
<code>ArgumentNullException</code>	Параметр <code>downloadingFileInfo</code> рівний <code>null</code> .
<code>OperationCanceledException</code>	Якщо <code>downloadingFileInfo.CancellationToken</code> перебуває в стані відміни, тобто його властивість <code>IsCancellationRequested</code> рівна <code>true</code> . Даний виняток може кинутись перед, під час або після процесу скачування.
<code>InvalidOperationException</code>	DS не знайшов жодного онлайн вузла, що належить групі, яка містить скачуваний файл, або жодного що має потрібний файл, або ж немає можливості позначити файл, як <code>sparse</code> (у випадку лише скачування файлу більшого, ніж максимальний розмір однієї частини файлу).
<code>FilePartiallyDownloadedException</code>	Було скачано один чи більше частин файлу, але не має можливості докінчити процес через відсутність вузлів, що мають файл потрібної версії.
<code>IOException</code>	Стався виняток при взаємодії з файловою системою.

3.12. Зупинка сервісу виявлення

Для зупинки сервісу виявлення існує 2 перевантаження методу `DiscoveryService.Stop`.

- А. Приймає булеву змінну `allowReuseService`, яка визначає чи може сервіс бути знову запущеним. Цей метод зупиняє періодичне розсилання multicast повідомлень, слухання TCP й UDP пакетів, викликає подію `ServiceInstanceShutdown` й звільнює усі некеровані дані пулу TCP-з'єднань. Оскільки останній реалізовує шаблон Одинак, то некеровані ресурси можна відновити лише шляхом їх переініціалізації.
- Б. Не приймає жодних параметрів й лише викликає попереднє перевантаження методу зі значенням `false`, оскільки в більшості випадках зупинка сервісу виявлення потрібна лише, коли закривається додаток (як це є, наприклад, в LUC).

РОЗДІЛ IV. ІНТЕГРАЦІЯ СЕРВІСУ ВИЯВЛЕННЯ З СЕРВІСОМ ОБМІНУ ФАЙЛАМИ

Перед тим, як представити інтеграцію, слід ознайомитись детальніше з самим сервісом обміну файлами.

4.1. Демонстрація роботи сервісу обміну файлами до інтеграції сервісу виявлення

Загалом робота LUC полягає у тому, щоб користувачі, які належать до певної групи або груп (наприклад, розробники, бухгалтери) мали однакові файли й папки в директорії синхронізації (вибрана користувачем папка, яка співставляється, узгоджується зі змістом інших користувачів). Це відбувається за рахунок існування окремого віддаленого серверу, який містить усі файли усіх груп.

Для демонстрації роботи застосунку почергово розглянемо найосновніші його властивості.

Після інсталяції LUC і його запуску в користувача з'являється наступне зображення для представлення інформації про цей сервіс:

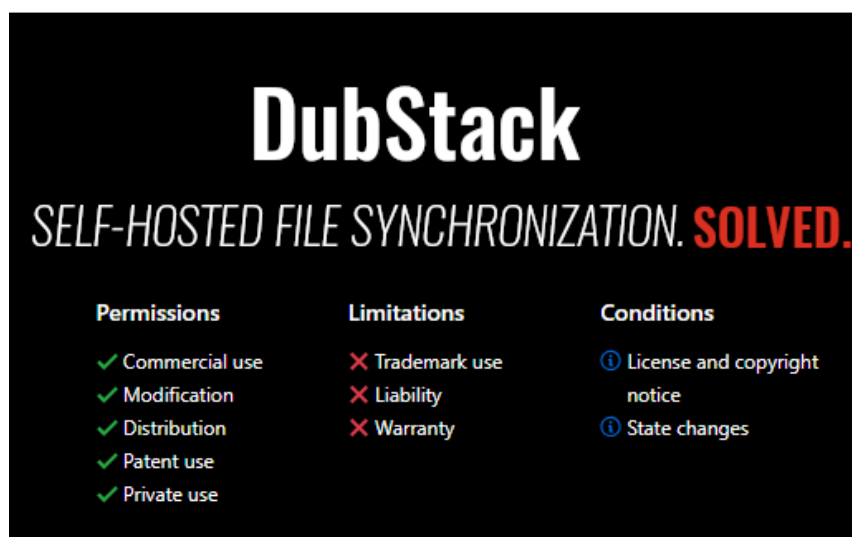


Рис. 4.1.1. Представлення комерційної інформації про сервіс обміну файлами при його запуску.

Після чого з'являється вікно для логінування.



Рис. 4.1.2. Вікно логінування в сервісі обміну файлами

Щоб зареєструватись потрібно перейти на сайт [76]. Коли користувач вже це зробив, він може користуватись даним додатком.

Далі користувачу потрібно вибрати папку синхронізації, тобто ту, в якій буде змінюватись зміст відповідно до серверу й груп, до яких належить користувач.

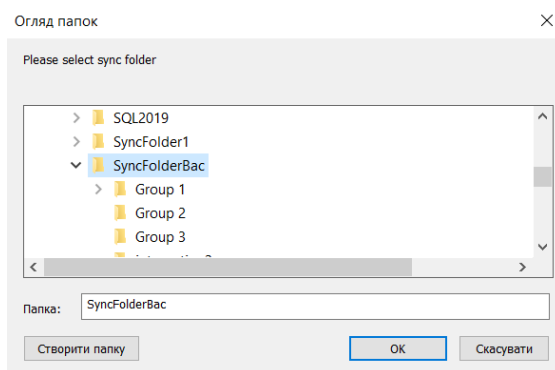


Рис. 4.1.3. Діалогове вікно вибору папки синхронізації

Клієнт має змогу змінити цю папки за допомогою іконки LUC в системному треї, вибравши пункт «Змінити каталог».

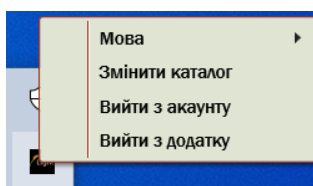


Рис. 4.1.4. Опції сервісу обміну файлами в системному треї

Або ж натиснувши двічі на цю іконку ЛКМ (лівою кнопкою миші).

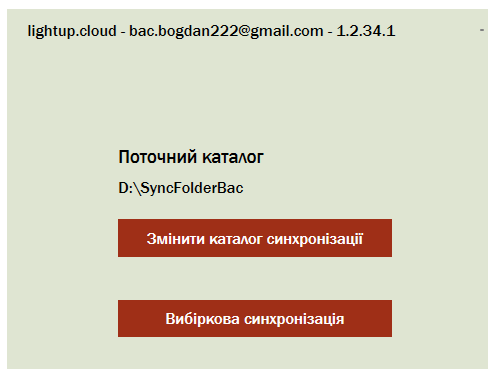


Рис. 4.1.5. Діалогове вікно для зміни папок, що синхронізуються

Вибіркова синхронізація надає можливість обрати папки, які не будуть співставлятись з відповідними директоріями на сервері.

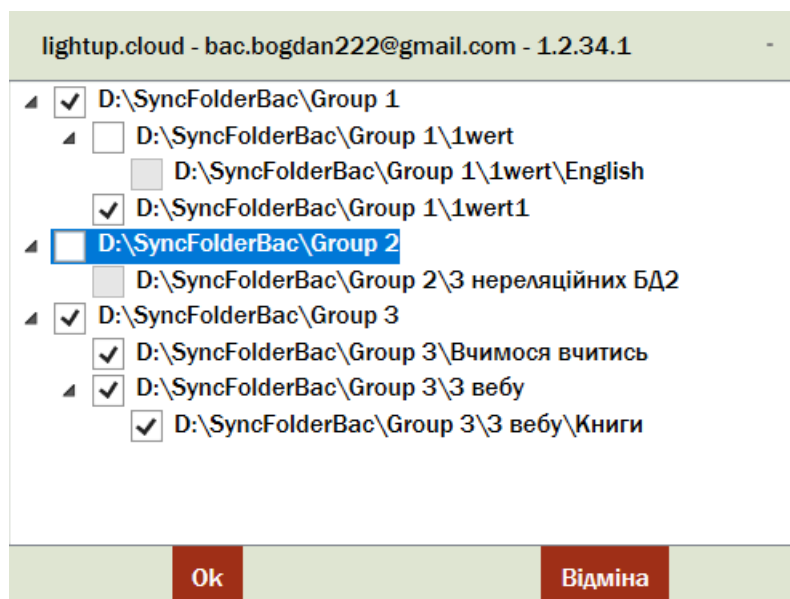


Рис. 4.1.6. Вибіркова синхронізація

Тут вибрано, щоб папки Group1\1wert й Group2 не синхронізувались з сервером.

Варто відмітити, що на кожен окрему групу існує окрема папка, яка створюється відповідно до параметрів акаунту. Лише ці папки синхронізуються у вибраній користувачем директорії співставлень. Під час роботи програми може

змінитись набір груп, до яких належить користувач, тому ці випадки також варто обробляти і в LUC, і в DS.

Наступним кроком роботи вибраного сервісу обміну файлами є повна синхронізація на сервер (тобто порівняння усього змісту папок й файлів) для визначення, які зміни відбулись у папках груп, до яких належить користувач, коли додаток був вимкненим.

```
SyncToServer started...
Syncing to D:\SyncFolderBac\Group 1 ...
Syncing to D:\SyncFolderBac\Group 1\1wert ...
Syncing to D:\SyncFolderBac\Group 1\1wert\English ...
Syncing to D:\SyncFolderBac\Group 1\1wert\FormsCommunicatoinTest ...
Syncing to D:\SyncFolderBac\Group 1\1wert\FormsCommunicatoinTest\.\vs ...
Syncing to D:\SyncFolderBac\Group 1\1wert\FormsCommunicatoinTest\.\vs\FormsCommunicatoinTest ...
Syncing to D:\SyncFolderBac\Group 1\1wert\FormsCommunicatoinTest\.\vs\FormsCommunicatoinTest\v16 ...
Syncing to D:\SyncFolderBac\Group 1\1wert\FormsCommunicatoinTest\.\vs\FormsCommunicatoinTest\v16\Server ...
Syncing to D:\SyncFolderBac\Group 1\1wert\FormsCommunicatoinTest\.\vs\FormsCommunicatoinTest\v16\Server\sqlite3 ...
Syncing to D:\SyncFolderBac\Group 1\1wert\FormsCommunicatoinTest\.\vs\FormsCommunicatoinTest\v16\TestStore ...
Syncing to D:\SyncFolderBac\Group 1\1wert\FormsCommunicatoinTest\.\vs\FormsCommunicatoinTest\v16\TestStore\0 ...
Syncing to D:\SyncFolderBac\Group 1\1wert1 ...
Syncing to D:\SyncFolderBac\Group 2 ...
Syncing to D:\SyncFolderBac\Group 2\3 нереляційних БД2 ...
Syncing to D:\SyncFolderBac\Group 3 ...
Syncing to D:\SyncFolderBac\Group 3\Вчимося вчитись ...
Syncing to D:\SyncFolderBac\Group 3\3 вебу ...
Syncing to D:\SyncFolderBac\Group 3\3 вебу\Книги ...
...finished SyncToServer
```

Рис. 4.1.7. Приклад логів синхронізації на сервер

Як бачимо, застосунок також підтримує перегляд логів за допомогою консолі, проте звичайні користувачі можуть його увімкнути лише, змінивши налаштування додатку в файлі appsettings.json, який розташований у папці C:\Users\<ім'я_користувача_Windows>\AppData\Local\LightUponCloud. Цю зміну потрібно виконати перед запуском програми.

Звісно, користувач також часто змінює файли під час роботи додатку. Для таких випадків існує клас FileChangesQueue, який зберігає, фільтрує події файлової системи при змінах у папці синхронізації, кожну секунду перевіряє список відфільтрованих подій та виконує синхронізацію на сервер. Це відбувається лише за умови, якщо в цей час не виконується синхронізація з серверу. Взагалюму одночасна робота цих видів синхронізації неможлива в LUC, оскільки в іншому випадку це могло б пошкодити цілісності набору

конфліктних файлів, тобто тих, що мають ідентичний шлях, назву та розширення.

```

Syncing from D:\SyncFolderBac\Group 3\Вчимосся вчитись ...
Queue-> Added deleted event: 26.05.2022 14:37:06 D:\SyncFolderBac\Group 1\Debug
Queue-> Added deleted event: 26.05.2022 14:37:06 D:\SyncFolderBac\Group 1\DigitalPictureHistograms
Queue-> Added deleted event: 26.05.2022 14:37:06 D:\SyncFolderBac\Group 1\Group 1
Queue-> Added deleted event: 26.05.2022 14:37:06 D:\SyncFolderBac\Group 1\Group 123231
Queue-> Added deleted event: 26.05.2022 14:37:06 D:\SyncFolderBac\Group 1\HelloApp
Queue-> Added deleted event: 26.05.2022 14:37:06 D:\SyncFolderBac\Group 1\Mobile apps
Queue-> Added deleted event: 26.05.2022 14:37:06 D:\SyncFolderBac\Group 1\NotifyIcon
Queue-> Added deleted event: 26.05.2022 14:37:06 D:\SyncFolderBac\Group 1\TestUpload
Queue-> Added deleted event: 26.05.2022 14:37:06 D:\SyncFolderBac\Group 1\Try
Queue-> Added deleted event: 26.05.2022 14:37:06 D:\SyncFolderBac\Group 1\WebApplication1
Syncing from D:\SyncFolderBac\Group 3\3 вебу ...
Syncing from D:\SyncFolderBac\Group 3\3 вебу\Книги ...
...finished sync from server 11:37:07
API Delete request for 10 items starting from D:\SyncFolderBac\Group 1\Debug

```

Рис. 4.1.8. Обробка змін в папці синхронізації під час
увідповіднення файлів й папок з серверу

Як бачимо, запит на видалення файлів з серверу відбувається лише після завершення синхронізації з серверу.

Після повної синхронізації на сервер запускається відповідна операція з нього. На відміну від попереднього виду, вона виконується періодично (кожну хвилину), а не при зміні файлів.

```

11:43:49 Sync from server started...
Syncing from D:\SyncFolderBac\Group 1 ...
Syncing from D:\SyncFolderBac\Group 1\Iwert ...
Syncing from D:\SyncFolderBac\Group 1\Iwert\English ...
Syncing from D:\SyncFolderBac\Group 1\Iwert\FormsCommunicatoinTest ...
Syncing from D:\SyncFolderBac\Group 1\Iwert\FormsCommunicatoinTest\vs ...
Syncing from D:\SyncFolderBac\Group 1\Iwert\FormsCommunicatoinTest\vs\FormsCommunicatoinTest ...
Syncing from D:\SyncFolderBac\Group 1\Iwert\FormsCommunicatoinTest\vs\FormsCommunicatoinTest\v16 ...
Syncing from D:\SyncFolderBac\Group 1\Iwert\FormsCommunicatoinTest\vs\FormsCommunicatoinTest\v16\Server ...
Syncing from D:\SyncFolderBac\Group 1\Iwert\FormsCommunicatoinTest\vs\FormsCommunicatoinTest\v16\Server\sqliite3 ...
Syncing from D:\SyncFolderBac\Group 1\Iwert\FormsCommunicatoinTest\vs\FormsCommunicatoinTest\v16\TestStore ...
Syncing from D:\SyncFolderBac\Group 1\Iwert\FormsCommunicatoinTest\vs\FormsCommunicatoinTest\v16\TestStore\0 ...
Syncing from D:\SyncFolderBac\Group 1\Iwert1 ...
Syncing from D:\SyncFolderBac\Group 1\Iwert1\Vidzy ...
Syncing from D:\SyncFolderBac\Group 1\Iwert1\Vidzy\vs ...
Syncing from D:\SyncFolderBac\Group 1\Iwert1\Vidzy\vs\Vidzy ...
Syncing from D:\SyncFolderBac\Group 1\Iwert1\Vidzy\vs\Vidzy\v16 ...
Syncing from D:\SyncFolderBac\Group 1\Iwert1\Vidzy\Vidzy ...
Syncing from D:\SyncFolderBac\Group 1\Iwert1\Vidzy\Vidzy\Properties ...
Syncing from D:\SyncFolderBac\Group 1\Iwert1\Vidzy\packages ...
Syncing from D:\SyncFolderBac\Group 1\Iwert1\Vidzy\packages\EntityFramework.6.1.3 ...
Syncing from D:\SyncFolderBac\Group 1\Iwert1\Vidzy\packages\EntityFramework.6.1.3\content ...
Syncing from D:\SyncFolderBac\Group 1\Iwert1\Vidzy\packages\EntityFramework.6.1.3\lib ...
Syncing from D:\SyncFolderBac\Group 1\Iwert1\Vidzy\packages\EntityFramework.6.1.3\lib\net40 ...
Syncing from D:\SyncFolderBac\Group 1\Iwert1\Vidzy\packages\EntityFramework.6.1.3\lib\net45 ...
Syncing from D:\SyncFolderBac\Group 1\Iwert1\Vidzy\packages\EntityFramework.6.1.3\tools ...
Syncing from D:\SyncFolderBac\Group 2 ...
Syncing from D:\SyncFolderBac\Group 2\3 нереляційних БД2 ...
Syncing from D:\SyncFolderBac\Group 3 ...
Syncing from D:\SyncFolderBac\Group 3\Вчимосся вчитись ...
Syncing from D:\SyncFolderBac\Group 3\3 вебу ...
Syncing from D:\SyncFolderBac\Group 3\3 вебу\Книги ...
...finished sync from server 11:43:55

```

Рис. 4.1.9. Приклад логів під час синхронізації з серверу.
Демонстрацію скачування див. у відповідному підрозділі нижче.

4.2. Виправлення сервісу обміну файлами

Дуже важливо було внести коректування в LUC з огляду на те, що синхронізація на початку моєї роботи з ним працювала з великою кількістю помилок. Це унеможливлювало правильне тестування інтеграції скачування з локальної мережі.

Мною були додані такі правки.

- А. Уникнуто використання методів `GetAwaiter`, `GetResult`, `Wait`, `Result._get` в UI-потоці, оскільки вони спричиняють отримання тупикової ситуації, якщо асинхронний метод не завершує своє виконання синхронно (функція може містити модифікатор `async`, але не виконувати жодних дій асинхронно).
- Б. Додано нові функції, процедури та коректування потокобезпечності та попередньо існуючих функцій до класів `CurrentUserProvider` (представляє дані про поточного користувача), `FileChangesQueue` (його опис див. вище).
- В. Скоректовано шаблон Ін'єкція Залежностей [67]. Раніше були значені проблеми з отримання значення певного інтерфейсу, оскільки не було методу, який надавав би змогу отримувати значення з відповідного контейнеру. Це було зроблено за рахунок створення інтерфейсу `IExportValueProvider` й використання патерну Адаптер [77], що надало змогу контейнерам різного типу (в LUC використовується `CompositionContainer` [78], в `DS.Test`, `LUC.UnitTests` та `LUC.IntegrationTests` проектах - `UnityContainer` [79]) реалізовувати даний інтерфейс.
- Г. Виправлено обробку помилок синхронізації, а також її перезапуск. Раніше в LUC під час певних несправностей при роботі з файловою системою або ж через зміну папки синхронізації, більше не

виконувалось узгодження зі змістом на сервері. Основним джерелом помилки було створення не UI-потокком об'єкту типу `DispatherTimer`, який періодично запускає синхронізацію. Достатньо мені було лише зробити ініціалізацію об'єкту 1 раз, тобто саме тоді, коли метод викликає UI-потік та скоректувати блоки `try..catch..finally`.

- Д. Виправлено узгодження неіснуючих папок на сервері при повній синхронізації на нього.
- Е. Вдало скоректовано видалення папок й файлів з серверу під час синхронізації з нього.
- Ж. Виправлено створення конфліктних файлів на сервері. Як було описано вище, вони виникають, коли 2 користувачі працюють з одним й тим самим файлом та змінюють його. Щоб зробити відповідне коректування, потрібно було виправити порівняння файлів на сервері й клієнті, а також забрати зміну `FileUploadResponse.OriginalName` на назву, що не відповідає конфліктній, після завантаження файлу на сервер.
- З. Додано правки до запису й зчитування з ADS. Тепер навіть при незмозі записати інформацію у відповідний потік про скачаний чи завантажений файл на сервер після відповідної операції, це буде зроблено під час порівняння файлів на сервері й локальному пристрої.
- И. Додано можливість логування важливої інформації на сервер. Даний функціонал не є протестований, оскільки відповідна сторінка зараз не працює.

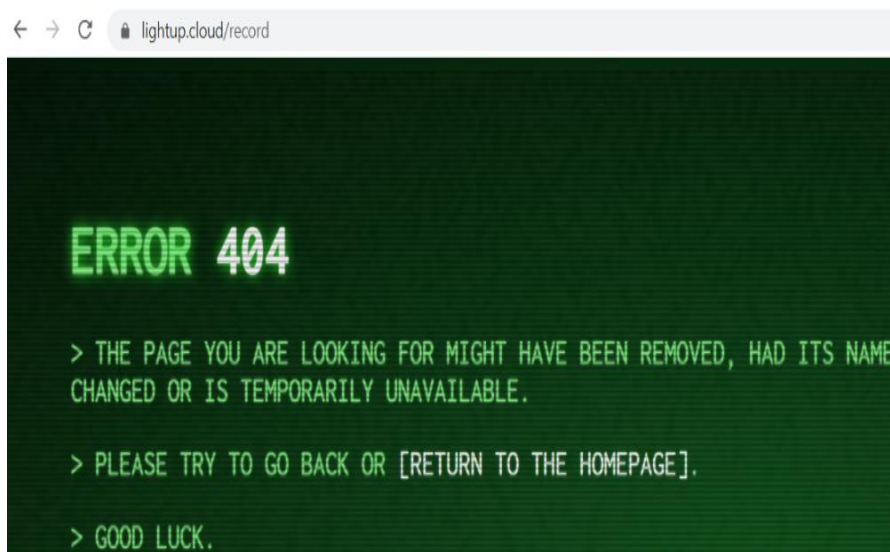


Рис. 4.2.1. Сторінка для логування важливих даних на сервер.

В майбутньому розробники серверу планують це виправити.

- К. Можливість запуску інтеграційних тестів LUC на будь-якому комп'ютері. Раніше була можливість їх запускати лише при наявності диску Е й файлової системи NTFS (New Technology File System — файлова система нової технології, яка використовується в останніх версіях Windows і Windows Server) на ньому (цей сервіс виявлення хіба для неї працює). Ці інтеграційні тести також надають змогу повністю очистити групи, до яких належить користувач.
- Л. Додано файл `.editorconfig`, який попереджає користувачів різними способами (пропозиція, попередження) про недотримання конвенцій коду. Цей файл звісно має деякі обмеження, тому було мною написаний файл «Конвенції в LUC», який можна переглянути за посиланням [80].
- М. Було відрефакторено велику кількість класів.
- Н. Оптимізовано скачування файлів. Див. про це відповідний наступний підрозділ.

О. Додані коректування завантаження файлу на сервер та обробку його відповіді.

П. виправлено встановлювач LUC для клієнтів. Для його створення використовується Inno Setup [58]. Раніше шляхи до файлів додатку були закодовані для конкретного ПК, тому щоб скомпілювати інсталятор потрібно було змінювати усі шляхи до файлів. Це було скоректовано.

Також було додані наступні виправлення до відповідного файлу.

- 1) Надано змогу вибрати будь-яку мову для інсталяції, що підтримується програмою Inno Setup.
- 2) Користувачу не потрібно вибирати жодних параметрів налаштування застосунку, включно з папкою, куди встановлювати (додаток займає близько 25 МБ, тому це не критично й полегшує процес інсталяції).
- 3) Також встановлюється ярлик на робочому столі, оскільки клієнтам LUC потрібно мати швидкий доступ до його запуску (зазвичай вони це будуть робити одразу при ввімкненні ПК, але через те, що ця програма ще не є досконало протестована, то вона не стартує автоматично при запуску Windows).

4.3. Фасад сервісу виявлення, його запуск

Для спрощення ініціалізації DS в LUC, був створений клас `DiscoveryServiceFacade`, який відповідає шаблону Фасад [68]. Він містить 2 відкриті статичні методи.

- 1) `InitWithoutForceToStart` отримує ініціалізований DS з групами, до яких належать залогований користувач. Якщо останнього не існує, то DS не містить ID відповідних груп;
- 2) `FullyInitialized` – викликає попередній метод й, якщо DS не був раніше запущений, то робить це. В інакшому випадку надсилає групову розсилку, але тільки якщо програма працює в режимі відлагодження (потрібно для швидкого оновлення бакетів). Для звичайного режиму це спричинить занадто велику навантаження на вузли мережі. Протокол Kademlia ефективніше оновить відповідну інформацію за рахунок оновлення K-бакетів.

Найчастіше саме вище описані методи по роботі з об'єктом типу `DiscoveryService` використовуються в LUC. Розглянемо, де саме це відбувається. При запуску цього додатку, а точніше при ініціалізації його сервісів за допомогою патерну Ін'єкції Залежностей, викликається метод `InitWithoutForceToStart`.

```
protected override void InitializeModules()
{
    InitDiscoveryService();
    base.InitializeModules();
}

private void InitDiscoveryService()
{
    var currentUserProvider = Container.GetExportedValue<ICurrentUserProvider>();
    var settingsService = Container.GetExportedValue<ISettingsService>();

    IDiscoveryService discoveryService = DiscoveryServiceFacade.InitWithoutForceToStart(
        currentUserProvider,
        settingsService
    );

    Container.ComposeExportedValue( discoveryService );
}
```

Після завершення методу `InitWithoutForceToStart` викликається `Container.ComposeExportedValue`. Він надає змогу в подальшому в будь-якій частині коду отримати значення експорту типу `IDiscoveryService`.

Варто відмітити причину відсутності запуску DS в цьому місці. Вона полягає в тому, що цей сервіс у фоні почне надсилання multicast при тому, що у поточний користувач ще не залогінувався, тобто не належить до жодної групи. В такому випадку усі інші вузли зможуть дізнатись про групи до яких належить поточний користувач, лише під час оновлення Kademlia-бакетів.

При ініціалізації DS також відбувається налаштування брандмауера, а для цього потрібні адміністративні права.

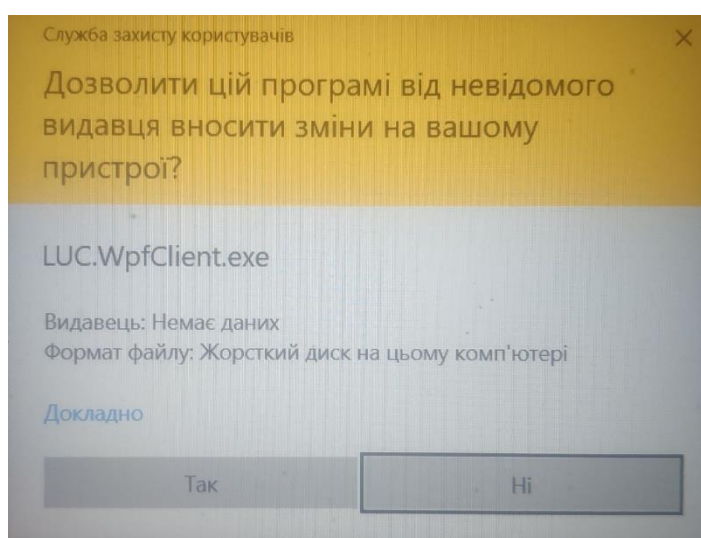


Рис. 4.3.1. Запит адміністративних прав для запуску LUC

Запуск сервісу виявлення, а також оновлення його бакетів відбувається за допомогою методу `DiscoveryServiceFacade.FullyInitialized` одразу після логінування користувача. Групи, до яких належить користувач можуть оновитися в LUC лише при новій авторизації акаунту.

4.4. Інтеграція скачування з локальної мережі

Оглянемо спершу, як працює скачування файлу в LUC без DS й за скільки часу вдається його скачати. Для прикладу завантажимо файл розміром 88 МБ. Для цього перейдемо на сайт [81]. Увійдемо в акаунт й завантажимо файл за допомогою команди “Upload”. В результаті отримаємо наступне.

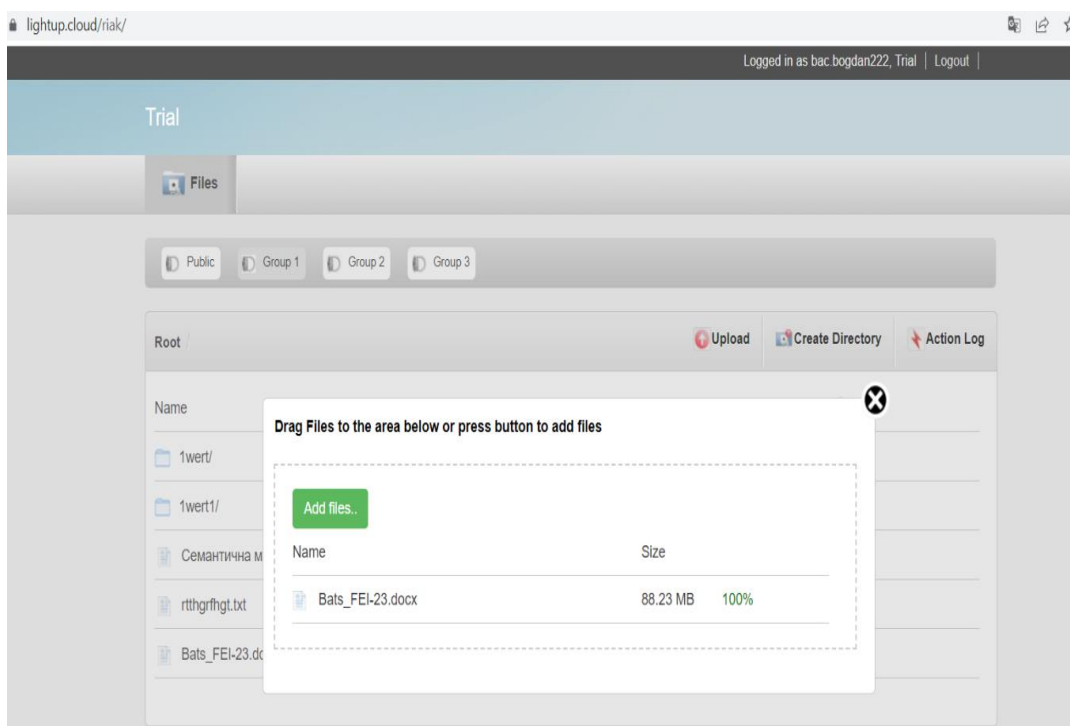


Рис. 4.4.1. Завантаження файлу на віддалений сервер

Логи скачування виглядають наступним чином.

```
Start download Bats_FEI-23.docx with server last modified time 12:36:13
D:\SyncFolderBac\Group 1\Bats_FEI-23.docx doesn't exist before download operation. 12:36:13
Bats_FEI-23.docx.part: downloading 13.28%. Now is 12:36:15
Bats_FEI-23.docx.part: downloading 26.56%. Now is 12:36:17
Bats_FEI-23.docx.part: downloading 39.85%. Now is 12:36:19
Bats_FEI-23.docx.part: downloading 53.13%. Now is 12:36:20
Bats_FEI-23.docx.part: downloading 66.41%. Now is 12:36:22
Bats_FEI-23.docx.part: downloading 79.69%. Now is 12:36:24
Bats_FEI-23.docx.part: downloading 92.98%. Now is 12:36:26
Download process time: 00:00:14.0555379 for file with size 88.2272081375122 MB
D:\SyncFolderBac\Group 1\Bats_FEI-23.docx was downloaded.
IsDownloaded = true => 12:36:28 => D:\SyncFolderBac\Group 1\Bats_FEI-23.docx
```

Рис. 4.4.2. Логи скачування файлу розміром 88 МБ з віддаленого серверу.

Мною було реалізовано так, щоб файли, що наданий момент скачуються, мали розширення .part. Це зроблено з метою вказівки користувачу, що це не є цілісний файл. Ці файли містяться в схованій папці «LUC temp files» в директорії синхронізації для того, щоб уникнути перезапису файлу з ідентичним шляхом, назвою й розширенням (наприклад, торент використовує файли .part).

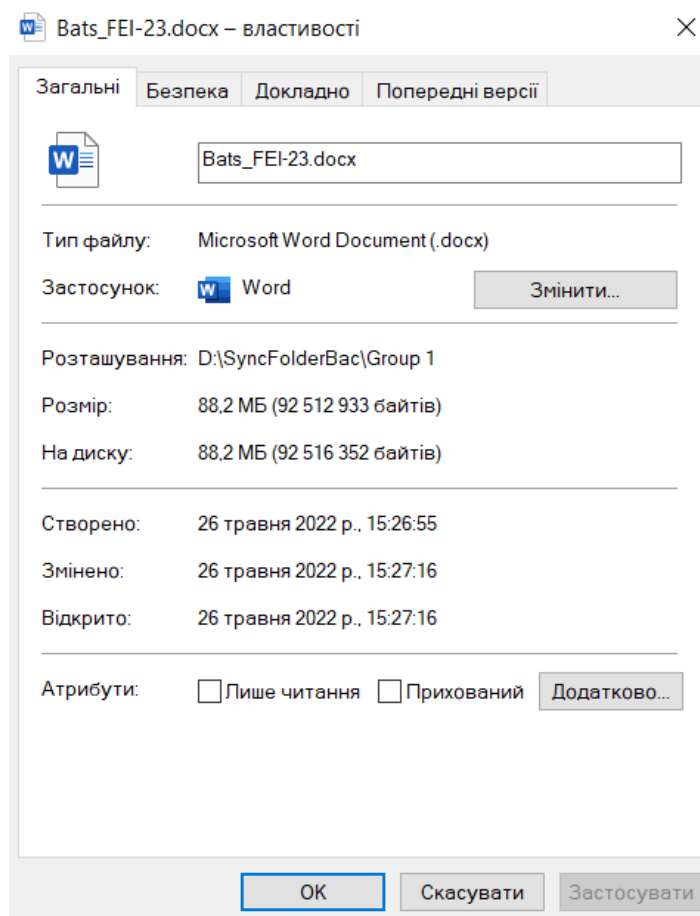


Рис. 4.4.3. Результат скачування у файлові системі.

Як бачимо, процес скачування є досить швидким (14 секунд), проте коли користувачів досить багато, то можна отримати помилку про перевантаження серверу або процес буде тривати занадто довго через відповідні причини. Це і є основна причина інтеграції DS в LUC.

Оглянемо поступово, які зміни були додані мною в скачування файлів в LUC.

А. Додано можливість скачування файлу іншої версії, коли за ідентичним шляхом існує файл, проте він використовується іншим процесом. Це зроблено за рахунок скачування в папку «LUC temp files», збереження відомостей про скачаний файл в ADS, FileChangesQueue і його обробки при наступній синхронізації з серверу.

Б. Створено окремий клас Downloader (див. додаток 8) для дотримання принципу SRP (Single Responsibility Principle – принцип єдиної відповідальності [74]), тобто для відповідальності лише за 1 завдання. В даному випадку – скачування. Раніше уся реалізація скачування була в класі ApiClient, який містить велику кількість методів пов’язаних з взаємодією з сервером.

В. Додано можливість відміни скачування. Детальніше про це див. наступний підрозділ.

Г. Додано створення папки «LUC temp files», куди скачуються файли й надання їх розширення .part (див. про це вище).

Д. Об’єкт типу DownloadingFileInfo тепер містить усю потрібну інформацію про скачуваний файл, окрім узагальнених даних про скачані його частини (див. про це підрозділ «Скачування файлів з локальної мережі»), оскільки це рідко стає в нагоді (лише у випадку, коли було скачано деякі з них з вузлів мережі, але більше немає змоги завершити відповідний процес без використання віддаленого серверу) й погіршує читабельність коду.

Е. Додано метод приведення об’єкта типу ObjectDescriptionModel (створюється на основі відповіді на List-запит) до DownloadingFileInfo.

```
public DownloadingFileInfo ToDownloadingFileInfo(
    String serverBucketName,
    String fullLocalFilePath,
    String fileHexPrefix )
{
```

```
    DownloadingFileInfo downloadingFileInfo =
        AppSettings.Mapper.Map<DownloadingFileInfo>( this );
```

Далі відбувається отримання об’єкту CurrentUserProvider за допомогою постачальника експортних значень й отримання даних про групу, до якої належить користувач.

```
ICurrentUserProvider currentUserProvider =
    AppSettings.ExportedValue<ICurrentUserProvider>();

downloadingFileInfo.BucketId = currentUserProvider.
```

```
LocalBucketIdentifier( serverBucketName );
```

```
downloadingFileInfo.ServerBucketName = serverBucketName;
```

Після цього присвоюється шлях файлу, де спершу він повинен бути скачаний. Цей шлях складається з шляху до папки «LUC temp files», від папки групи до файлу, де повинен вкінці-кінців бути розташований файл, а також з розширення .part. Після цього відбувається присвоєння цільового шляху файлу й зашифрований у 16-ому вигляді шлях від групи (не включно) до файлу. Це закодоване значення називається hex-префіксом файлу.

Ж. Після створення об'єкту типу downloadingFileInfo відбувається спроба запису в ADS про те, що скачування файлу почалось. Це потрібно для того, щоб визначити, чи файл був скачаний, але не переміщений у папку відповідної групи (наприклад, у випадку, коли користувач відкрив файл, що потрібно замінити). Справа в тому, що за допомогою розміру файлу це не можна зробити, оскільки він встановлюється одразу при відкритті потоку файлу.

З. Після запису в ADS відбувається скачування з локальної мережі. При цьому його винятки обробляються наступним чином.

Таблиця 3

Обробки винятків, що кинуті методом
DownloaderFromLocalNetwork.DownloadFileAsync

Виняток	Обробка
InvalidOperationException	Скачування усього файлу з серверу.
FilePartiallyDownloaded-Exception	Виконується перевірка, чи файл є змінений на сервері і якщо так, то відбувається відміна скачування. В іншому випадку відбувається злиття діапазонів частин, що повинні бути скачанні і якщо оптимально їх скачувати по чергово з серверу (що є досить тривалою операцією в порівнянні зі скачуванням усього файлу з нього), то починається відповідна операція. Якщо не є оптимально, то відбувається скачування усього файлу з серверу.
OperationCanceledException	Видалення файлу з «LUC temp files».
IOException	Видалення файлу з «LUC temp files» й сповіщається користувачу про певну помилку при взаємодії з файловою системою

Р. Змінено обробку скачаного файлу. Тепер немає необхідності турбуватись, що користувач може відкрити файл перед записом певної інформації в ADS (див. попередній підрозділ), тому відповідна ситуація не обробляється, що пришвидшує виконання в мінімум 0.5 секунд.

4.5. Відміна скачування

Потреба у зупинці даної функції є у наступних випадках.

- 1) Користувач виходить з акаунту;
- 2) під час синхронізації з серверу. Дана операція запускає кожні 30 секунд (вибрано дане значення базуючись на швидкості скачування та порівняння файлів) виконання List-запитів, тобто отримання інформації про файли на сервері у директорії, що на поточний момент узгоджується. Це потрібно з метою визначення, чи були внесені якісь зміни до файлів на сервері, що зараз скачуються на локальному пристрої. Якщо так, то виконується їхня відміна;
- 3) перейменування папки користувачем, куди скачується файл;
- 4) користувач вибрав іншу папку синхронізації.

Відповідну операцію може зробити або той, хто передав значення в параметр `cancellationToken` (за замовчування воно рівне значенню `CancellationToken.None`) за допомогою екземпляру `CancellationTokenSource` і його методу `Cancel` (для цього потрібно передати в метод `DownloadFileAsync` його властивість `Token`), або будь-який програмний потік LUC-у за допомогою об'єкту типу `ISyncingObjectList`, якого можна отримати за допомогою статичного методу `AppSettings.ExportedValue`. Щоб відмінити скачування за допомогою цього об'єкту, потрібно скористатись його методом `CancelDownloadingAllFilesWhichBelongPath`, який відмінює усі функції скачування, що виконують відповідний процес з файлом, цільовий шлях якого містить шлях, переданий цьому методі. За допомогою методу `TryCancelAllDownloadingFilesWithDifferentVersions` виконується відміна усіх скачувань, в яких версія файлу на сервері відрізняється від тої, що зараз скачується.

4.6. Зупинка сервісу виявлення в сервісі обміну файлами

Потреба у зупинці DS є лише у випадку закриття додатку LUC, оскільки під час його роботи важливо, щоб DS постійно оновлював інформацію про вузли мережі в незалежності від того, чи залогований користувач.

РОЗДІЛ V. ТЕСТУВАННЯ СЕРВІСУ ВІЯВЛЕННЯ Й ІНТЕГРАЦІЇ

5.1. Модульні тести

Оглянемо 1 модульний тест для представлення шаблону, за яким більшість були написані.

1) Приклад модульного тесту

Оглянемо, як часто повторюються випадкові KID й модифікацію створення відповідного значення. Для цього був створений відповідний модульний тест. Він написаний за використанням паралелізму мови C# та шаблону AAA (Arrange-Act-Assert – організувати-виконувати-стверджувати) [82].

- Організація

Спершу визначається кількість KID, що будуть створювати. Оптимальним значенням для досить швидкого виконання є 1 мільйон:

```
class KademliaIdTest
{
    [Test]
    public void RandomId_GenerateLotOfIds_AllShouldBeDifferent()
    {
        Int32 countOfIds = 1000 * 1000;
```

Далі визначаються опції паралелізму: максимальний його ступінь – 1000 *

<кількість_процесорів_поточного_пристрою>:

```
Int32 maxAvailableDegreeOfParallelism = Environment.ProcessorCount * 1000;
```

```
var parallelOptions = new ParallelOptions
{
    MaxDegreeOfParallelism = countOfIds > maxAvailableDegreeOfParallelism ?
        maxAvailableDegreeOfParallelism :
        countOfIds
};
```

Далі ініціалізується потокобезпечний словник, який використовується для зменшення кількості синхронізацій потоків й визначення однакових KademliaId:

```
var ids = new ConcurrentDictionary<KademliaId, Object>();
```

- Виконання

Акт працює паралельно для збільшення швидкості виконання тесту. Для цього використовується статичний клас `Parallel`, який паралельно виконує цикл `for`, в якому створює випадкові `KID`:

```
Parallel.For(
    fromInclusive: 0,
    toExclusive: countOfIds,
    parallelOptions,
    body: ( numId, loopState ) =>
    {
        var rndId = KademliaId.Random();
```

- Ствердження

Якщо в словник не було успішно додано `rndId`, то це означає, що таке значення вже є в ньому. Клас `Assert` використовується лише для зупинення тесту, оскільки його виконання є досить повільним:

```
Boolean isAdded = ids.TryAdd( rndId, value: null );
if(!isAdded)
{
    ids.Count.Should().Be( countOfIds,
        because: "Kademlia protocol generates a lot of IDs and anyone can be for
        contact" );
    loopState.Stop();
}
```

Також зупиняються усі потоки, які запустив метод `Parallel.For`, щоб як можна швидше припинити виконання тесту.

Було досліджено, що в середньому за близько 1000 ітерацій отримуються ідентичні значення.

```
System.AggregateException : One or more errors occurred.
-----> NUnit.Framework.AssertionException : Expected ids.Count to be 1000000 because Kademlia protocol generates a lot of IDs and anyone can be for contact, but found 441.
-----> NUnit.Framework.AssertionException : Expected ids.Count to be 1000000 because Kademlia protocol generates a lot of IDs and anyone can be for contact, but found 440.
-----> NUnit.Framework.AssertionException : Expected ids.Count to be 1000000 because Kademlia protocol generates a lot of IDs and anyone can be for contact, but found 441.
-----> NUnit.Framework.AssertionException : Expected value to be 1000000 because Kademlia protocol generates a lot of IDs and anyone can be for contact, but found 440.
```

Рис. 5.1.1. Результат тестування отримання випадкових
`KademliaId` (реалізація бібліотеки `Clifton.Kademlia`)

```
System.AggregateException : One or more errors occurred.
-----> NUnit.Framework.AssertionException : Expected ids.Count to be 1000000 because Kademlia protocol generates a lot of IDs and anyone can be for contact, but found 8283.
-----> NUnit.Framework.AssertionException : Expected ids.Count to be 1000000 because Kademlia protocol generates a lot of IDs and anyone can be for contact, but found 8282.
-----> NUnit.Framework.AssertionException : Expected ids.Count to be 1000000 because Kademlia protocol generates a lot of IDs and anyone can be for contact, but found 8284.
-----> NUnit.Framework.AssertionException : Expected ids.Count to be 1000000 because Kademlia protocol generates a lot of IDs and anyone can be for contact, but found 8282.
```

Рис. 5.1.2. Інший результат тестування отримання випадкових
`KademliaId` (реалізація бібліотеки `Clifton.Kademlia`)

Саме тому для формування нового рандомного значення тепер використовується GUID (globally unique identifier – глобально унікальний ідентифікатор розміром 16 байт).

Загалом отримання випадкового KID тепер виглядає наступним чином.

```
public static KademliaId Random()
{
    var guid = Guid.NewGuid();
    Byte[] guidBuffer = guid.ToArray();

    Byte[] rndBuffer = new Byte[ DsConstants.KID_LENGTH_BYTES - guidBuffer.Length ];
    s_rndObj.NextBytes( rndBuffer );

    Byte[] idBuffer = guidBuffer.Concat( rndBuffer );

    var randomId = new KademliaId( idBuffer );
    return randomId;
}
```

В Clifton.Kademlia використовувався лише метод `s_rndObj.NextBytes`, який формував одразу ціле значення KID. Тут він використовується для формування лише 4 байт, оскільки `BigInteger` є розміром 20.

При такій зміні цього методу, ідентичні значення KID за всі спроби тестування жодного разу не отримались (результат див. далі).

2) Результати модульних тестів.

Усього тестів проекту сервісу виявлення було написано 29. Їх назви формувались за допомогою наступного шаблону: Метод, що тестується_Сценарій_Очікуваний результат. На основі даних імен можна зрозуміти, всю логіку методу, тому не будемо їх описувати. У нище наведених зображеннях представлено усі тести та час їхнього виконання.

Test	Duration	Traits	Error Message
DiscoveryService.Test (29) <i>Назва проекту</i>	20 sec		
LUC.DiscoveryServices.Test (21)	6,6 sec		
AsyncSocketTest (3) <i>Назва класу, який містить модульні тести</i>	136 ms		
Connect_RemoteEndpointsNull_GetException <i>Назва модульного тесту</i>	127 ms		
ConnectAsync_RemoteEndpointsNull_GetException	5 ms		
SendAsync_WithoutSettingConnection_ThrowSocketException	4 ms		
DiscoveryServiceTest (13)	6,4 sec		
BeforeCreatedInstance_CreateDifferentProtocolVersionAndTryToGetAccordingDs_ThrowsArgumentException	3 ms		
Ctor_PassNullPar_GetException	1 ms		
Instance_CreateOneMoreInstanceOfDsAndCompareMachinelD_TheSameMachinelD	49 ms		
ListenTcp_SendTooBigMessage_ItShoudntBeReceived	3,4 sec		
MachinelD_TwoCallsInstanceMethod_TheSamelD	12 ms		
QueryAllServices_WhenSdlsntStarted_GetException	53 ms		
QueryAllServices_WhenSdlsStartedAndStopped_GetException	306 ms		
SendTcpMess_SendNullMessage_GetException	264 ms		
SendTcpMess_SendNullRemoteEndPoint_GetException	2 ms		
SendTcpMess_SetParInTypeTcpMess_GetException	4 ms		
SendTcpMessageAsync_GetTcpMessage_NotFailed	1,7 sec		
StartAndStop_RoundTrip_WithoutExceptions	599 ms		
SupportedBuckets_AddOneltemUsingCollectionFunc_ShoudntBeAddedToDs	1 ms		
WireReaderWriterTest (5)	82 ms		
IEnumerable_GeneralTest_AreEquivalent	63 ms		
ReadString_BufferOverflow_EndOfStreamException	3 ms		
WireReaderWriter_GeneralTests_Equals	12 ms		
WriteByteLengthPrefixedBytes_TooBigByteArray_ArgumentException	2 ms		
WriteString_TooBigString_ArgumentException	2 ms		

Test	Duration	Traits	Error Message
WriteString_TooBigString_ArgumentException	2 ms		
LUC.DiscoveryServices.Test.InternalTests (4)	8,5 sec		
ConnectionPoolSocketTest (1)	1,2 sec		
StateInPool_FewTasksWantToTakeSocketAndOnlyOneSetstInPool_WaitingOfTakingFromPoolsGreaterThanFrequencySettInPool(auto<Socket	728 ms		
ConnectionPoolTest (2)	509 ms		
ReturnedToPool_TakeSocketFromPoolThenSetStateInPoolinisFailedAndReturnToPool_ShouldBeReturned	5,2 sec		
SocketAsync_FewThreadsWantToTakeSocketAndOnlyOneDoesIt_WaitingOfTakingFromPoolsGreaterThanFrequencySettInPool(auto<Socket>	5,2 sec		
RequestTest (1)	2 sec		
ResultAsync_SendPingRequestSyncLotOfTimes_ValueTasksAreImmediatelyCompleted	2 sec		
LUC.DiscoveryServices.Test.InternalTests.Kademlia (4)	4,9 sec		
BucketListTest (1)	25 ms		
AddContact_AddKContacts_WithoutSplitBuckets	25 ms		
KademliadTest (3)	4,9 sec		
EqualOperator_CreateRandomIdAndCompareWithDefaultBigInteger_ShouldNotBeEqual	1 ms		
EqualOperator_CreateRandomIdAndCompareWithItValue_ShouldBeEqual	< 1 ms		
RandomId_GenerateLotOfIds_AllShouldBeDifferent	4,9 sec		
LUC.IntegrationTests (41)			
LUC.UnitTests (6)	37,5 sec		

Рис. 5.1.3. Результати тестування класів сервісу виявлення

Як бачимо, також існують тести LUC-у. Як інтеграційні, так і модульні. Мною було також написано кілька тестів класів LUC-у.

Test	Duration	Traits	Error Message
LUC.UnitTests (6)	37,5 sec		
LightClientTests (2)			
LUC.UnitTests (4)	37,5 sec		
LucDownloadTest (3)	37,2 sec		
DownloadFileAsync_CancelDownloadBecauseOfFileOnServerAndInFileSystemHaveDifferentVersions_DownloadIsCancelledAndFilesDeleted	6 sec		
DownloadNecessaryChunksFromServer_DownloadAllChunksOfFileFromServer_DownloadedFileHasSameLengthAsCreatedInFileSystem	31,1 sec		
JoinedChunkRanges_GetAllRangesAndRemoveLastAndSecondAndMiddle_CountOfJoinedChunkRangesIsThree	9 ms		
ServerObjectDescriptionTest (1)	345 ms		
CompareFileOnServerAndLocal_VersionIsEmptyFileDoesntExistOnLocalPc_ComparationResultShouldBeDoesntExistLocallyAndDoesntExistOnServer	345 ms		

Рис. 5.1.4. Модульні тести частин LUC, які були змінені мною

Варто виділити, що скачування по чанках з серверу працює дуже повільно, тому його варто використовувати лише коли чанків з локальної мережі є скачано досить багато.

Для представлення роботи DS краще підійдуть функціональні тести.

5.2. Функціональні тести

Передусім потрібно встановити інструменти для створення контейнерів. Це важливо, оскільки не буде можливості додати підтримку Docker, хоч це не описується в документації Microsoft [83].

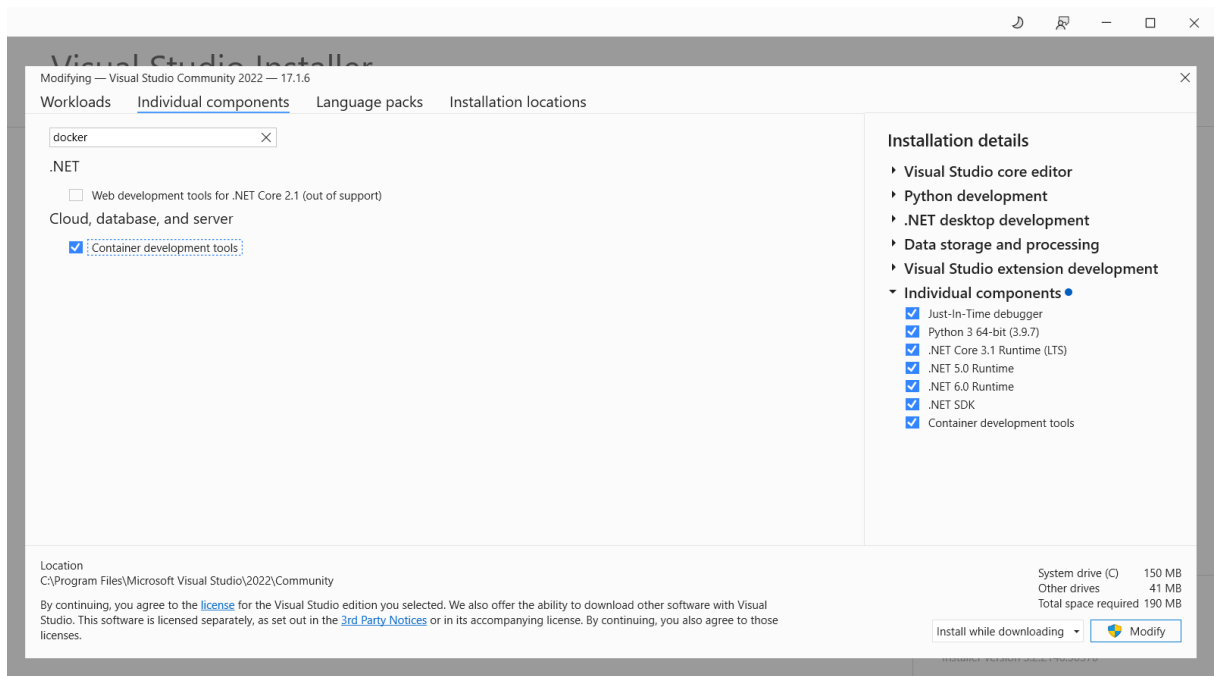


Рис. 5.2.1. Вибір інструменту для додання docker-compose до проекту

Після скачування й установлення «Container development tools», потрібно додати в проект DiscoveryService.Test підтримку «Container Orchestrator», оскільки консольні проекти .NET Framework підтримують тільки оркестр контейнерів. Для цього спершу потрібно запустити Docker Desktop [84] й переключитись на Windows-контейнери [61]. Щоб виконати цей перехід потрібно, щоб був увімкнений Hyper-V [85] й Windows-контейнери, які не підтримується у Windows 10 й 11 Home.

Після того, як Docker Desktop був запущений й був виконаний вище вказаний перехід, потрібно оглядачі рішень у VS2022 натиснути ПКМ (праву кнопку миші) на проект, який контейнери будуть запускати, тобто DiscoveryService.Test. Далі слід вибрати «Додати» → «Підтримка Container Orchestrator». В діалоговому вікні натискаємо «ОК».



Рис. 5.2.2. Діалогове вікно вибору оркестру контейнерів

Як бачимо, тільки Docker Compose підтримується. Після цього в рішення будуть додані наступні елементи.

- Проект docker-compose – збірка, що автоматизує багатопроцесорні розгортання і керування ресурсами додатків в одному файлі - docker-compose.yml, який в свою чергу вказує особливості створення контейнерів.
- Dockerfile – доданий до проекту DiscoveryService.Test файл, який вказує порядок створення та запуску контейнерів
- .dockerignore - файл, що виконує виключення файлів з контексту, схоже, як це робить .gitignore з вилученням даних з git-репозиторію. Це допомагає зробити побудову проекту docker-compose швидшою і менш ресурсозатратною, виключаючи з контексту великі файли або репозиторії, які не використовуються при збиранні.

Після додання цих файлів до рішення виконується скачування сервісів, які необхідні для запуску власних Docker-застосунків. Вони займають близько 10 гігабайт.

Для створення контейнерів, які будуть мати свою мережу потрібно було змінити файл `docker-compose.yml`. В наступному вигляді він створює 3 контейнери й спільну мережу `nat` – аналог мережі `bridge`, тільки перша використовується для Windows-контейнерів.

```
version: '3.4'
services:
  ds.test.node1:
    image: ${DOCKER_REGISTRY-}dstestnode1
    build:
      context: .\LUC.DiscoveryService.Test
      dockerfile: Dockerfile
    networks:
      netForDsTest: {}

  ds.test.node2:
    image: ${DOCKER_REGISTRY-}dstestnode2
    build:
      context: ./LUC.DiscoveryService.Test
      dockerfile: Dockerfile
    networks:
      netForDsTest: {}

  ds.test.node3:
    image: ${DOCKER_REGISTRY-}dstestnode3
    build:
      context: ./LUC.DiscoveryService.Test
      dockerfile: Dockerfile
    networks:
      netForDsTest: {}
networks:
  netForDsTest:
```

Властивість `image` вказує на шлях до зображення Docker. `DOCKER_REGISTRY`-шлях до реєстру Docker – система зберігання та розповсюдження іменованих зображень Docker. Властивість `build` визначає, як буде будуватись зображення, тобто з якого `Dockerfile` й контексту (набір файлів за певною локацією). Властивість `networks` об'єкту сервісу вказує про мережі, до яких буде він під'єднаний. За допомогою окремої властивості `networks` вказується об'єкти мереж, які будуть створюватися.

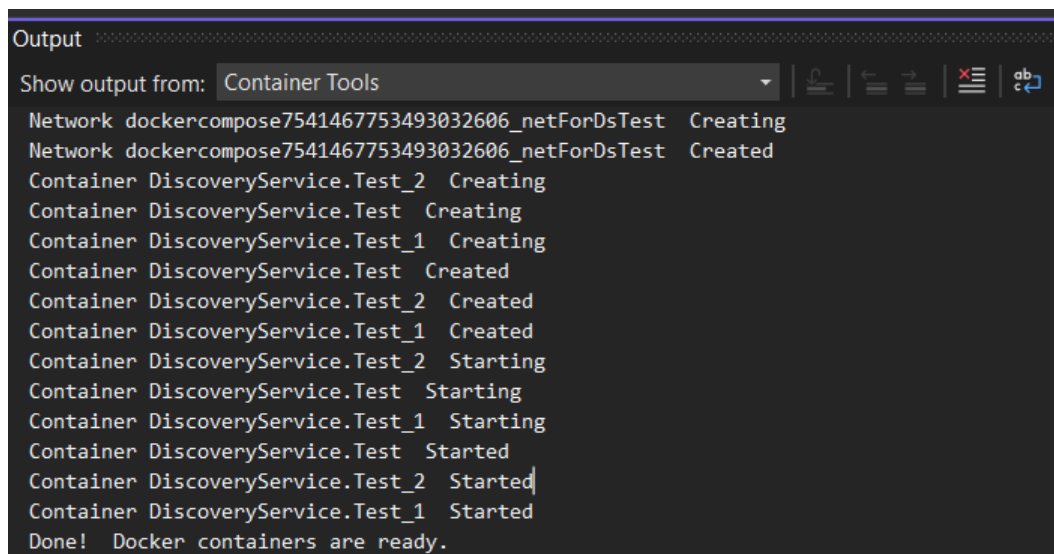
Тепер нам залишилось запустити проект `docker-compose`, щоб протестувати сервіс виявлення. Але часто трапляється, що при спробі вперше це робити, отримується помилка «Status: COPY failed: file not found in build context

or excluded by .dockerignore: stat obj\Docke\r\publish: file does not exist, Code: 1».

Щоб її виправити потрібно виконати або кілька наступних дій, або одну з них.

- 1) Перебудова проекту docker-compose;
- 2) перезапуск Docker Desktop.

Після того, як ця збірка успішно запустилась (тобто були створені мережа netForDsTest й контейнери), потрібно відкрити вікно Контейнери.



```

Output
Show output from: Container Tools
Network dockercompose7541467753493032606_netForDsTest Creating
Network dockercompose7541467753493032606_netForDsTest Created
Container DiscoveryService.Test_2 Creating
Container DiscoveryService.Test Creating
Container DiscoveryService.Test_1 Creating
Container DiscoveryService.Test Created
Container DiscoveryService.Test_2 Created
Container DiscoveryService.Test_1 Created
Container DiscoveryService.Test_2 Starting
Container DiscoveryService.Test Starting
Container DiscoveryService.Test_1 Starting
Container DiscoveryService.Test Started
Container DiscoveryService.Test_2 Started
Container DiscoveryService.Test_1 Started
Done! Docker containers are ready.
  
```

Рис. 5.2.3. Журнал записів створення Docker-мережі й контейнерів

Встановити назву контейнера можна за допомогою властивості container_name. Згодом було переіменовано контейнери за форматом DsTestNode<номер_контейнера>.

Щоб відкрити цей вікно потрібно у VS2022 вибрати Огляд → Інші вікна → Контейнери.

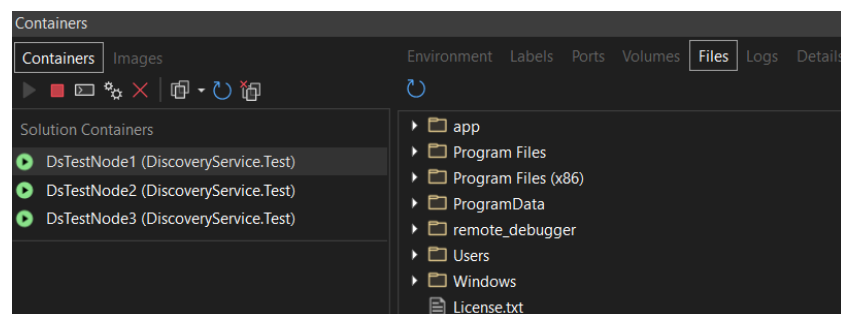



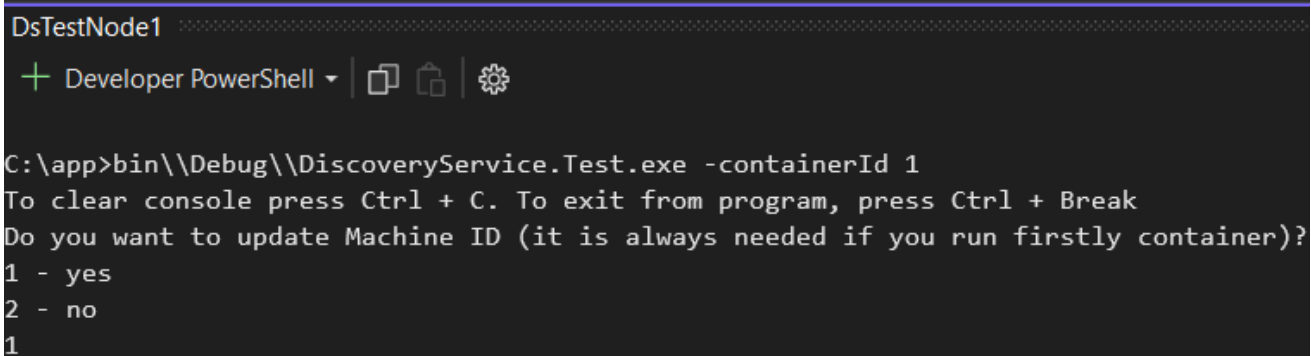
Рис. 5.2.4. Вікно «Контейнери»

Перед запуском тестів, варто очистити всі дані з серверу для того, щоб вузли скачували здебільшого скачували файли, що є на інших пристроях мережі. Це можна зробити за допомогою методу `OnlineOneUserOneActionIntegrationTests.DeleteEverythingFromServer` зі збірки `LUC.IntegrationTests`.

Також потрібно розкоментувати наступний рядок в файлі `DiscoveryService.Test`, щоб не відбувалось налаштування Firewall, оскільки Windows-контейнери його не мають.

```
#define DOES_CONTAINER_USE
```

Тепер запустимо в 3-х контейнерах проект `DS.Test`. Для цього натиснемо по черзі на кожен з них й  (відкрити вікно терміналу). В кожному з них введемо `bin\\Debug\\DiscoveryService.Test.exe -containerId <ID-контейнеру>`. Цей аргумент потрібен для того, щоб в контексті в певному файлі зберігались `MachineId`, що відповідають різним контейнерам, й для можливості тестування `DS`, коли вузол вийшов з мережі. Оскільки ми вперше запускаємо контейнер, спершу потрібно оновити ID вузла. Нехай він відповідатиме останньому символу його назви.



```
DsTestNode1
+ Developer PowerShell ▾ | [ ] [ ] [ ]
C:\app>bin\\Debug\\DiscoveryService.Test.exe -containerId 1
To clear console press Ctrl + C. To exit from program, press Ctrl + Break
Do you want to update Machine ID (it is always needed if you run firstly container)?
1 - yes
2 - no
1
```

Рис. 5.2.5. Запуск `DS.Test` проекту в контейнері

Далі відбувається запуск фонового відновлювача TCP-з'єднань, а також вказується ID вузла. В `DS.Test` ці ідентифікатори формуються на основі серійного номеру материнської плати, ID процесора (як в `LUC`) й GUID.

```
Starting BackgroundConnectionResetHelper worker.
Started BackgroundConnectionResetHelper worker.
Your machine Id: 46a4c615163a54dcafe48014613b76c9---c26d2aa9-12a7-4e83-8148-feb6f01f578f
```

Рис. 5.2.6. Запуск фонового відновлювача
TCP-з'єднань й виведення ID вузла

Після цього виконується логінування користувача `integration1`, який використовується в LUC для його тестування. В методі `ApiClient.LoginAsync` одразу після успішного логінування запускається DS, оскільки саме першому класу потрібно ініціалізувати `Downloader`.

```
DsTestNode1
+ Developer PowerShell |
Logged as 'integration1' at 09:53:49 Wednesday, 01 June 2022
DS (Discovery Service) is starting
Finding network interfaces
Found nic 'Ethernet'.
Found IP fe80::f089:d4e8:fb50:55%4 which is reachable in current network
Found IP 172.22.145.200 which is reachable in current network
Count of Reachable IP-addresses = 2:
    IP = fe80::f089:d4e8:fb50:55%4
    IP = 172.22.145.200

Successfully started listen TCP messages on [fe80::f089:d4e8:fb50:55%4]:17500 by the TcpServer
Successfully started listen TCP messages on 172.22.145.200:17500 by the TcpServer
Successfully started listen UDP messages on [fe80::f089:d4e8:fb50:55%4]:17500 by the UdpClient
Successfully started listen UDP messages on 172.22.145.200:17500 by the UdpClient
DS is started
LoginServiceModel:
    Groups:
        GroupServiceModel:
            Id = integration1;
            Name = integration1;

        GroupServiceModel:
            Id = integration2;
            Name = integration2;

    Id = 03a3a647d7e65013f515b16b1d9225b6;
    Login = integration1;
```

Рис. 5.2.7. Запуск сервісу виявлення

Далі виконується опитування користувача, чи хоче він спостерігати зміни в одній з папок `DownloadTest`, кожна з яких формується на основі останніх 5-ти символів `MachineId` контейнеру. є папкою синхронізації.

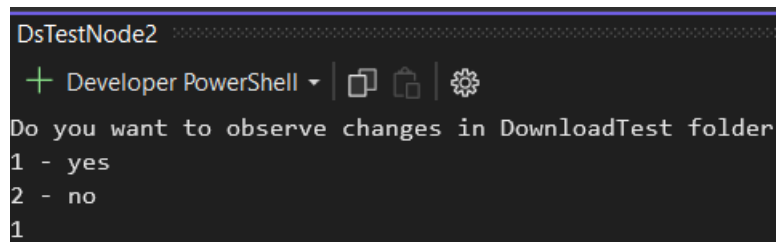


Рис. 5.2.8. Можливість спостереження змін в папці синхронізації

До цих змін на даних час відноситься лише створення файлів й папок для того, щоб можна було завантажити файл на віддалений сервер. Виконання upload є важливим, оскільки визначення, який файл скачувати відбувається саме в залежності від тих файлів, які є на сервері у папках груп, до яких належить integration1, тобто це integration1 й integration2. Цей варіант визначення, який файл скачувати є одним з найкращих, оскільки в такому випадку можна скачувати файли, навіть з тих вузлів, які використовують LUC програму, а не DS.Test. Після цього вказується шлях до файлу, що містить налаштування LUC сервісів, й папка синхронізації поточного вузла (в кожного вона інша з метою одночасного скачування файлів з різних контейнерів). Для DsTestNode1 -, DsTestNode2 -, DsTestNode3 -

```

FilePath = C:\Users\ContainerAdministrator\AppData\Local\LightUponCloud\appsettings.json
Full root folder name is updated to C:\app\bin\Debug\DownloadTest\4a0be

```

Рис. 5.2.9. Шлях до файлу-налаштувань сервісів LUC й до папки синхронізації вузла DsTestNode2

Далі виконується запит до користувача, чи хоче він тестувати кількість допустимих з'єднань, які можуть бути з TcpServer. Ця операція відрізняється від усіх інших й не потребує виявлення жодних вузлів, тому спершу опитується, чи її необхідно виконати. Якщо ні, починає виконуватись періодична розсилка multicast UDP-повідомлень кожні 5 секунд, допоки не буде знайдено хоча б 1 вузол. Сам DS це не робить (визначає, чи це варто робити за допомогою StackTrace й наявності у ньому простору імен LUC.DiscoveryServices.Test),

оскільки в проекті DS.Test дані операції повинні виконуватись за вимогою й не перекривати зміст іншої інформації.

```

DsTestNode1
+ Developer PowerShell
Do you want to test count available connections?
1 - yes
2 - no
2
Started send UDP messages. Their content:
MulticastMessage:
    MachineId = 46a4c615163a54dcafe48014613b76c9---c26d2aa9-12a7-4e83-8148-feb6f01f578f;
    MessageId = 1110669997;
    MessageLength = 85;
    MessageOperation = Multicast;
    ProtocolVersion = 1;
    TcpPort = 17500;

Successfully sent UDP message (85 bytes) by [fe80::f089:d4e8:fb50:55%4]:17500
Successfully sent UDP message (85 bytes) by 172.22.145.200:17500
Finished send UDP messages

```

Рис. 5.2.10. Запит тестування максимальної кількості TCP-з'єднань й періодична розсилка UDP-повідомлень

Саме DsTestNode1 знайшов усі інші, оскільки в них були запущені DS, але не виконувалась періодична розсилка. Примітка: дані повідомлень в записах представляються за допомогою рефлексії.

Наприклад, представимо записи в терміналі контейнеру DsTestNode3.

```

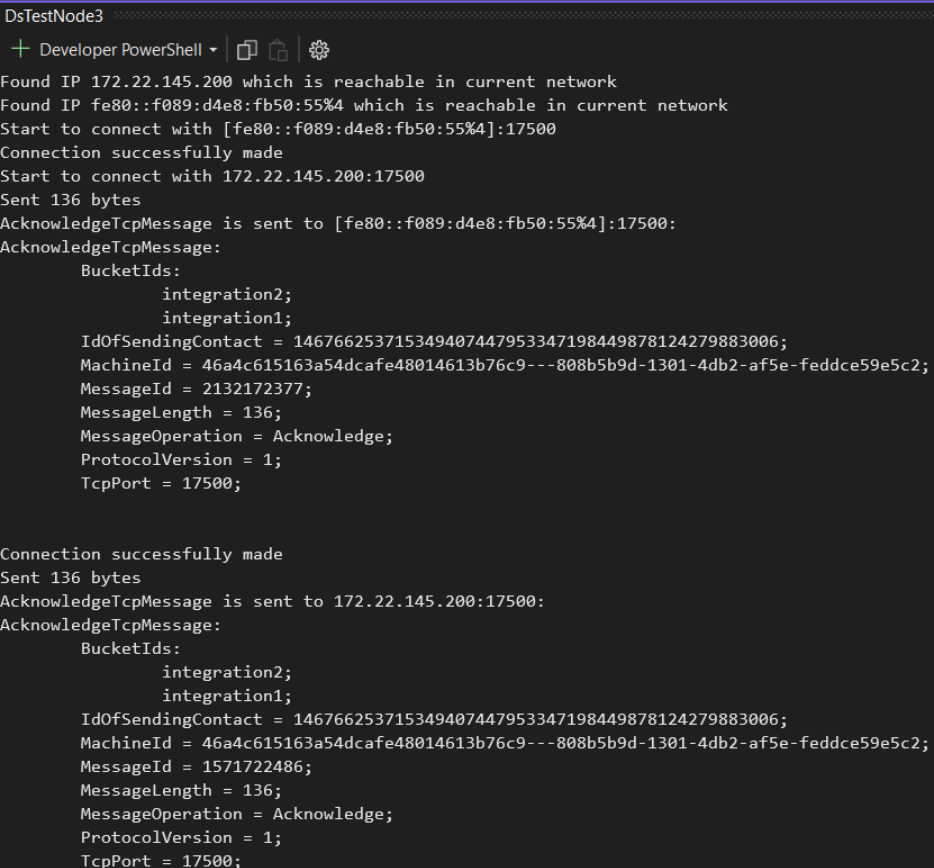
Received UPD message with bytes from RemoteEndPoint = [fe80::f089:d4e8:fb50:55%4]:17500:
MulticastMessage:
    MachineId = 46a4c615163a54dcafe48014613b76c9---c26d2aa9-12a7-4e83-8148-feb6f01f578f;
    MessageId = 1110669997;
    MessageLength = 85;
    MessageOperation = Multicast;
    ProtocolVersion = 1;
    TcpPort = 17500;

Received UPD message with bytes from RemoteEndPoint = 172.22.145.200:17500:
MulticastMessage:
    MachineId = 46a4c615163a54dcafe48014613b76c9---c26d2aa9-12a7-4e83-8148-feb6f01f578f;
    MessageId = 1110669997;
    MessageLength = 85;
    MessageOperation = Multicast;
    ProtocolVersion = 1;
    TcpPort = 17500;

```

Рис. 5.2.11. Отримані UDP-повідомлення вузлами DsTestNode2 й DsTestNode3

Після одержання цих повідомлень виконується з'єднання з кінцевими точками, які надіслали UDP, й відсилання їм AcknowledgeTcpMessage.



```

DsTestNode3
+ Developer PowerShell |
Found IP 172.22.145.200 which is reachable in current network
Found IP fe80::f089:d4e8:fb50:55%4 which is reachable in current network
Start to connect with [fe80::f089:d4e8:fb50:55%4]:17500
Connection successfully made
Start to connect with 172.22.145.200:17500
Sent 136 bytes
AcknowledgeTcpMessage is sent to [fe80::f089:d4e8:fb50:55%4]:17500:
AcknowledgeTcpMessage:
    BucketIds:
        integration2;
        integration1;
    IdOfSendingContact = 146766253715349407447953347198449878124279883006;
    MachineId = 46a4c615163a54dcafe48014613b76c9---808b5b9d-1301-4db2-af5e-feddcce59e5c2;
    MessageId = 2132172377;
    MessageLength = 136;
    MessageOperation = Acknowledge;
    ProtocolVersion = 1;
    TcpPort = 17500;

Connection successfully made
Sent 136 bytes
AcknowledgeTcpMessage is sent to 172.22.145.200:17500:
AcknowledgeTcpMessage:
    BucketIds:
        integration2;
        integration1;
    IdOfSendingContact = 146766253715349407447953347198449878124279883006;
    MachineId = 46a4c615163a54dcafe48014613b76c9---808b5b9d-1301-4db2-af5e-feddcce59e5c2;
    MessageId = 1571722486;
    MessageLength = 136;
    MessageOperation = Acknowledge;
    ProtocolVersion = 1;
    TcpPort = 17500;
  
```

Рис. 5.2.12. Надсилання повідомлення з даними про поточний вузол

При цьому DsTestNode1 приймає сокети, які були створені платформою .NET в результаті успішного TCP-з'єднання, виконує зчитування повідомлень за допомогою них та виконує Kademlia Bootstrap. При цьому він створює вже нові з'єднання для надсилання FindNodeRequest. Звісно, можна було б виконувати надсилання й прийнятими сокетами, але це б спричинило зміну виконання FindNode операції, яка бере сокет з пулу, а не з TcpServer.

```

DsTestNode1
+ Developer PowerShell
Successfully accepted socket by InterNetwork TcpServer
Successfully accepted socket by InterNetworkV6 TcpServer
Successfully accepted socket by InterNetworkV6 TcpServer
Successfully accepted socket by InterNetwork TcpServer
Found TcpSession with 136 available data to read
Found TcpSession with 136 available data to read
Read message with 136 bytes
Read message with 136 bytes
Started to handle 136 bytes
Started to handle 136 bytes
Found IP 172.22.148.101 which is reachable in current network
Found IP 172.22.147.246 which is reachable in current network
Start to connect with 172.22.148.101:17500
Connection successfully made
Start to connect with 172.22.147.246:17500
Connection successfully made
Sent 172 bytes
Sent 173 bytes
Request FindNodeRequest is sent to 172.22.148.101:17500:
FindNodeRequest:
    BucketIds:
        integration2;
        integration1;
    KeyToFindCloseContacts = 704102967522451936099249284595837590793227644869;
    MessageLength = 173;
    MessageOperation = FindNode;
    RandomID = 1302303844875222001160407721227205041451184789599;
    SenderKadId = 704102967522451936099249284595837590793227644869;
    SenderMachineId = 46a4c615163a54dcafe48014613b76c9---c26d2aa9-12a7-4e83-8148-feb6f01f578f;
    TcpPort = 17500;
Request FindNodeRequest is sent to 172.22.147.246:17500:
FindNodeRequest:

```

Рис. 5.2.13. Прийняття сокетів TcpServer-ом від 2-х вузлів й обробка їхніх AcknowledgeTcpMessage

Інші вузли приймають дані запити й надсилають FindNodeResponse.

```

DsTestNode3
+ Developer PowerShell
Successfully accepted socket by InterNetwork TcpServer
Found TcpSession with 173 available data to read
Read message with 173 bytes
Started to handle 173 bytes
Started to handle FindNodeRequest
Started send 63 bytes to 172.22.145.200:49175.
FindNodeResponse:
    BucketIds:
        integration2;
        integration1;
    MessageLength = 63;
    MessageOperation = FindNodeResponse;
    RandomID = 1302303844875222001160407721227205041451184789599;
    TcpPort = 17500;
    CountCloseContacts = 0
Sent 63 bytes to 172.22.145.200:49175
Finished to handle FindNodeRequest
Finished to handle 173 bytes
Started execute Ping and AddContact on contact 46a4c615163a54dcafe48014613b76c9---c26d2aa9-12a7-4e83-8148-feb6f01f578f in
method SendResponse, which doesn't exist before in a Dht

```

Рис. 5.2.14. Відправка FindNodeResponse

```

Read message with 63 bytes
The response is received (63 bytes):
FindNodeResponse:
    BucketIds:
        integration2;
        integration1;
    MessageLength = 63;
    MessageOperation = FindNodeResponse;
    RandomID = 1302303844875222001160407721227205041451184789599;
    TcpPort = 17500;
    CountCloseContacts = 0

```

Рис. 5.2.15. Отримання FindNodeResponse

А оскільки раніше відправника FindNodeRequest раніше не було в жодному К-бакеті, то виконується надсилання йому PingRequest, щоб переконатись що це не зломисник й додати інформацію про нього в певний бакет.

```

Started execute Ping and AddContact on contact 46a4c615163a54dcafe48014613b76c9---c26d2aa9-12a7-4e83-8148-feb6f01f578f in method SendResponse, which doesn't exist before in a Dht
Successfully accepted socket by InterNetworkV6 TcpServer
Found IP 172.22.145.200 which is reachable in current network
Sent 119 bytes
Request PingRequest is sent to 172.22.145.200:17500:
PingRequest:
    MessageLength = 119;
    MessageOperation = Ping;
    RandomID = 393724119457518843744354104530684869247835705595;
    SenderKadId = 146766253715349407447953347198449878124279883006;
    SenderMachineId = 46a4c615163a54dcafe48014613b76c9---808b5b9d-1301-4db2-af5e-feddc59e5c2;

```

Рис. 5.2.16. Відправка PingRequest після отримання FindNodeRequest від невідомого вузла

```

Read message with 26 bytes
The response is received (26 bytes):
PingResponse:
    MessageLength = 26;
    MessageOperation = PingResponse;
    RandomID = 393724119457518843744354104530684869247835705595;

RpcError:
    HasError = False
Finished execute Ping and AddContact on contact 46a4c615163a54dcafe48014613b76c9---c26d2aa9-12a7-4e83-8148-feb6f01f578f in method SendResponse, which doesn't exist before in a Dht

```

Рис. 5.2.17. Отримання PingResponse

Оскільки RpcError.HasError = False, то вузол DsTestNode1 додається до К-бакету у DsTestNode3. Аналогічні операції виконуються й у DsTestNode2. Останні вказані 2 вузли дізнаються про одне одного зарахунок періодичного оновлення К-бакетів, під час якого надсилаються FindNodeRequest до усіх їхніх

контактів й додається уся інформація про контакти, що отримана у властивості `FindNodeResponse.CloseSenderContacts`.

```
DsTestNode2
+ Developer PowerShell |
Request FindNodeRequest is sent to 172.22.145.200:17500:
FindNodeRequest:
    BucketIds:
        integration2;
        integration1;
    KeyToFindCloseContacts = 0;
    MessageLength = 153;
    MessageOperation = FindNode;
    RandomID = 452387639880637211154688391242783607208723876275;
    SenderKadId = 707558109901669009553857793055462556133919997867;
    SenderMachineId = 46a4c615163a54dcafe48014613b76c9---346297b6-a164-4b96-bd64-413c0824a0be;
    TcpPort = 17500;
```

Рис. 5.2.18. Надсилення FindNodeRequest під час періодичного оновлення К-бакетів

```
DsTestNode2
+ Developer PowerShell |
Read message with 331 bytes
The response is received (331 bytes):
FindNodeResponse:
    BucketIds:
        integration2;
        integration1;
    MessageLength = 331;
    MessageOperation = FindNodeResponse;
    RandomID = 452387639880637211154688391242783607208723876275;
    TcpPort = 17500;
    CountCloseContacts = 1
    CloseContacts:
        Contact:
            IpAddressesCount = 2;
            KadId = 146766253715349407447953347198449878124279883006;
            LastActiveIpAddress = fe80::cdb:b8a1:722f:a6ac%4;
            LastSeen = 06/01/2022 10:50:23;
            MachineId = 46a4c615163a54dcafe48014613b76c9---808b5b9d-1301-4db2-af5e-feddce59e5c2;
            TcpPort = 17500;

RpcError:
    HasError = False
```

Рис. 5.2.19. Отримання FindNodeResponse від DsTestNode1

Як можна побачити з властивості `MachineId`, ми отримали інформацію про DsTestNode3. Тепер розглянемо допустимі опції в `DS.Test` для тестування сервісу виявлення.

```

Select an operation:
1 - send multicast
2 - send PingRequest
3 - send StoreRequest
4 - send FindNodeRequest
5 - send FindValueRequest
6 - send AcknowledgeTcpMessage
7 - test count available connections
8 - download random file from another contact(-s)
9 - create file with random bytes

```

Рис. 5.2.20. Допустимі операції в проєкті DS.Test

Розглянемо тепер скачування файлів. Для цього спершу створимо файл з випадковими байтами формату .txt в DsTestNode2. Для цього виберемо операцію під номером 9.

```

DsTestNode2
+ Developer PowerShell
9
Input count MBs: 15
Input path from C:\app\bin\Debug\DownloadTest\4a0be and file name with extension which you want to create: integration1\15MBfromNode2.txt
File C:\app\bin\Debug\DownloadTest\4a0be\integration1\15MBfromNode2.txt successfully created
Do you want to upload on server file integration1\15MBfromNode2.txt. It was created
1 - yes
2 - no
1
API Upload C:\app\bin\Debug\DownloadTest\4a0be\integration1\15MBfromNode2.txt started by user integration1. ID= 03a3a647d7e65013f515b16b1d9225b6... UTC:4:55:04 PM
... finished API Upload C:\app\bin\Debug\DownloadTest\4a0be\integration1\15MBfromNode2.txt UTC:4:55:15 PM
Do you want to find any new contacts?
1 - yes
2 - no

```

Рис. 5.2.21. Створення файлу й завантаження його на віддалений сервер

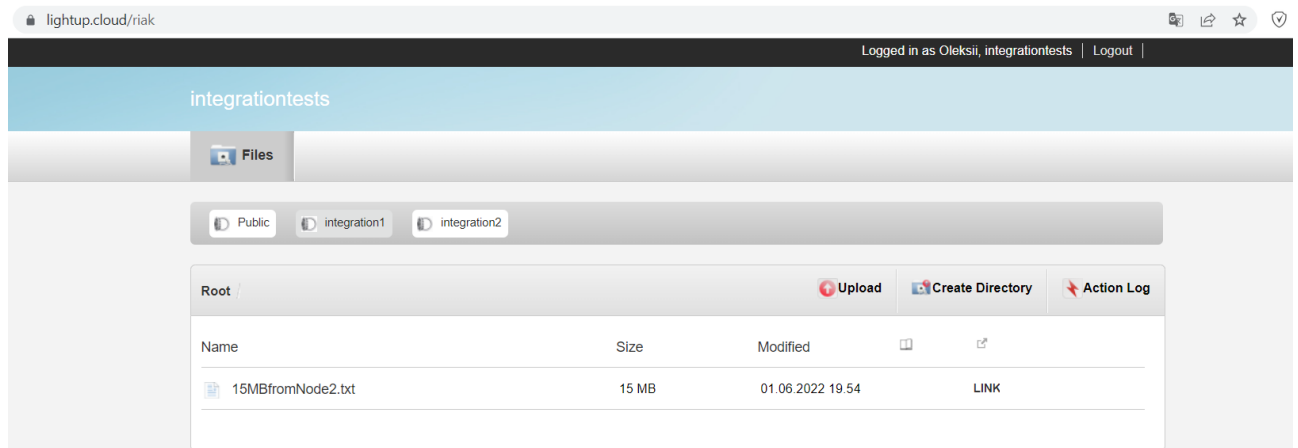
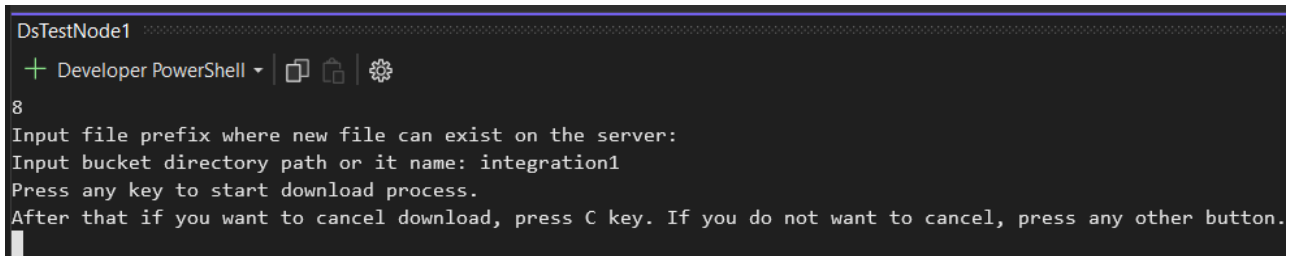


Рис. 5.2.22. Завантажений файл на сервері

Як бачимо також є допустима опція знайти нові контакти. Вона виконує надсилання UDP multicast та очікування їх обробки DS-ом.

Тепер по черзі скачуємо цей файл кожним вузлом. Для цього виберемо опцію під номером 8 й виберемо префікс до файлу відносно назви групи та саму групу. Файл випадковим чином буде вибраний з наявних для скачування. Після цього натиснемо кнопку Enter, щоб розпочати процес. Як бачимо, що при натиску на кнопку C, можна відмінити скачування.



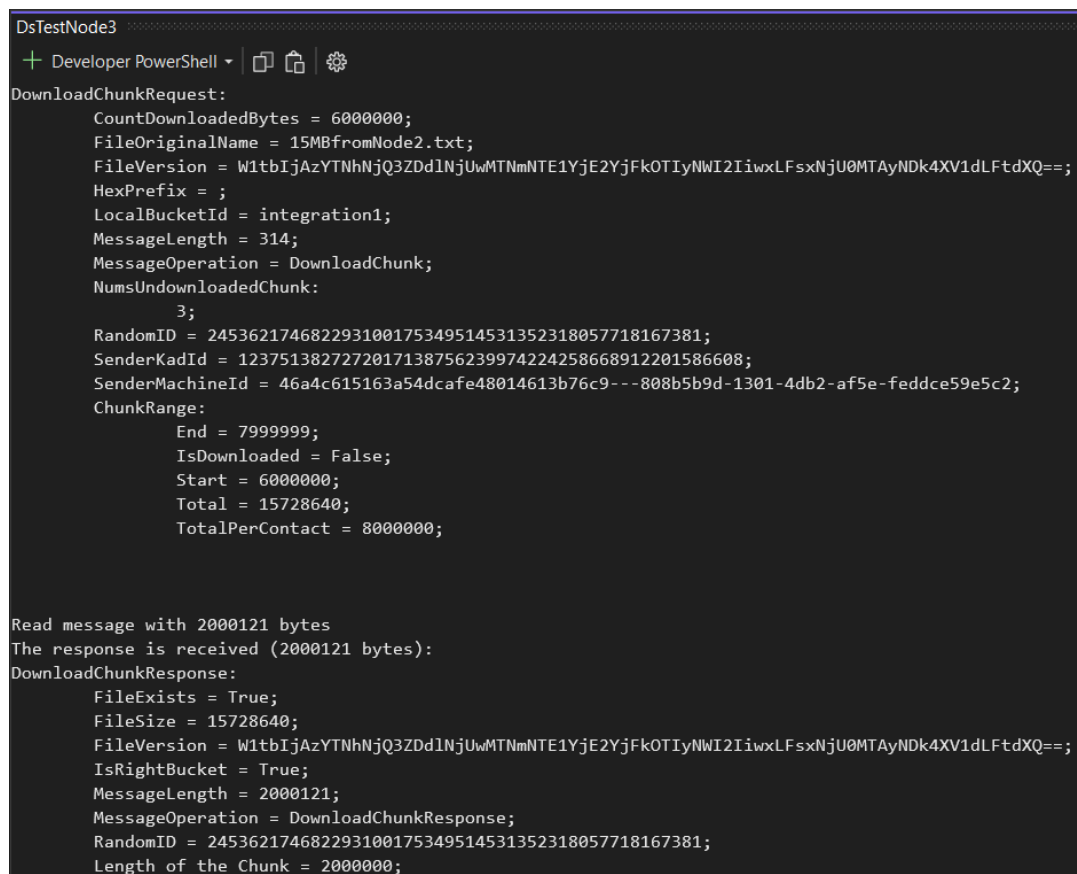
```

DsTestNode1
+ Developer PowerShell | [ ] [ ] [ ]
8
Input file prefix where new file can exist on the server:
Input bucket directory path or it name: integration1
Press any key to start download process.
After that if you want to cancel download, press C key. If you do not want to cancel, press any other button.

```

Рис. 5.2.23. Введення вхідних параметрів скачування файлу

Наприклад, запит скачування чанку й відповіді виглядає наступним чином.



```

DsTestNode3
+ Developer PowerShell | [ ] [ ] [ ]
DownloadChunkRequest:
    CountDownloadedBytes = 6000000;
    FileOriginalName = 15MBfromNode2.txt;
    FileVersion = W1tbIjAzYTNhNjQ3ZDdlNjUwMTNmNTE1YjE2YjFkOTIyNWl2IiwxLFsxNjU0MTAyNDk4XV1dLFtdXQ==;
    HexPrefix = ;
    LocalBucketId = integration1;
    MessageLength = 314;
    MessageOperation = DownloadChunk;
    NumUndownloadedChunk:
        3;
    RandomID = 245362174682293100175349514531352318057718167381;
    SenderKadId = 1237513827272017138756239974224258668912201586608;
    SenderMachineId = 46a4c615163a54dcafe48014613b76c9---808b5b9d-1301-4db2-af5e-feddce59e5c2;
    ChunkRange:
        End = 7999999;
        IsDownloaded = False;
        Start = 6000000;
        Total = 15728640;
        TotalPerContact = 8000000;

Read message with 2000121 bytes
The response is received (2000121 bytes):
DownloadChunkResponse:
    FileExists = True;
    FileSize = 15728640;
    FileVersion = W1tbIjAzYTNhNjQ3ZDdlNjUwMTNmNTE1YjE2YjFkOTIyNWl2IiwxLFsxNjU0MTAyNDk4XV1dLFtdXQ==;
    IsRightBucket = True;
    MessageLength = 2000121;
    MessageOperation = DownloadChunkResponse;
    RandomID = 245362174682293100175349514531352318057718167381;
    Length of the Chunk = 2000000;

```

Рис. 5.2.24. Приклад запиту й відповіді скачування чанку

Як бачимо, запити також зберігають кількість скачаних байтів, що згодом використовується для визначення, чи файл був скачаний. Також варто відмітити, що кожного разу не створюється новий запит, а змінюється лише RandomID й ChunkRange.

```
Download time is 00:00:05.8407490 of file with size 15 MB
Downloaded chunks:
ChunkRange:
  End = 9999999;
  IsDownloaded = True;
  Start = 8000000;
  Total = 15728640;
  TotalPerContact = 7728640;

ChunkRange:
  End = 1999999;
  IsDownloaded = True;
  Start = 0;
  Total = 15728640;
  TotalPerContact = 8000000;
```

Рис. 5.2.25. Час скачування й деякі скачані діапазони частин файлу

Тут представлено скачування частин файлу з 2-х різних вузлів, тому діапазон суттєво відрізняється. Час скачування можна суттєво оптимізувати шляхом вибору конфігурації проекту – Release, оскільки в такому випадку буде значно менше логів.

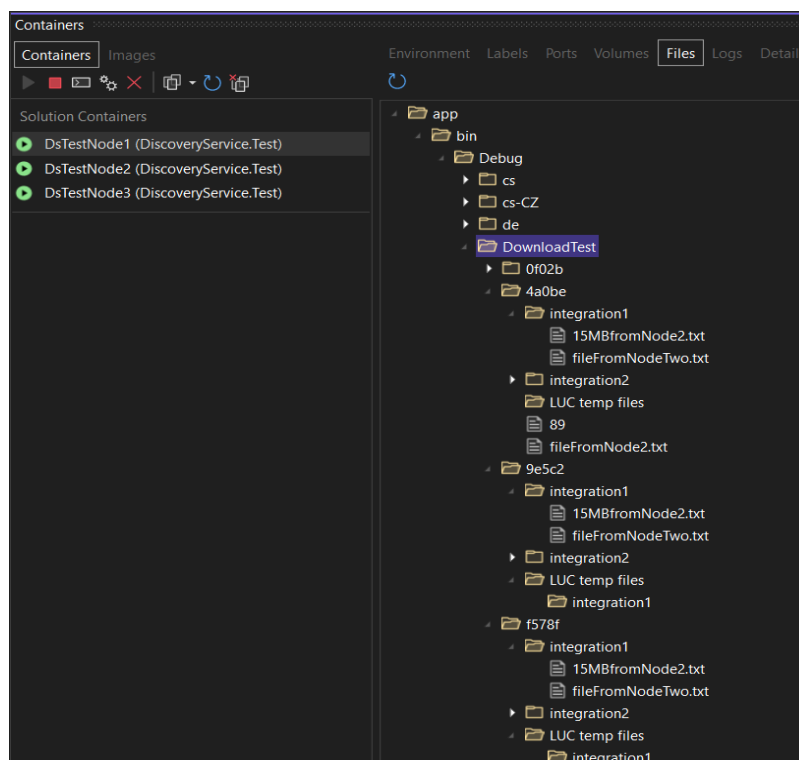


Рис. 5.2.26. Результат скачування файлу «15MBfromNode2.txt»

Папка синхронізації «DownloadTest\f578f» - вузла DsTestNode1, «DownloadTest\4a0be» - DsTestNode2 (він немає жодних папок в LUC temp files, оскільки він створював файли, а не скачував їх), «DownloadTest\9e5c2» - вузла DsTestNode3. Як бачимо, DsTestNode1 й DsTestNode3 успішно скачали файл 15MBfromNode2.txt в папки integration1.

5.3. Відомі проблеми сервісу й способи його покращення

- Додати SSL/TLS-підтримку, щоб слухання TCP-повідомлень не блокували такі додатки, як CrowdStrike Falcon [65]. Це було помічено при тестуванні сервісу на одному з пристроїв, що установив останнього.
- Додати можливість змінювати порт при неможливості слухати повідомлення на поточному.
- Зробити використання методів Task.Run й замків мінімальним, оскільки це сповільнює роботу програми.
- Вимагати привілеїв адміністратора при запуску DS, лише коли в Firewall не було додано дозволу обміну мережевими пакетами відповідному виконуваному файлу.
- Детальніше ознайомитись з Kademlia-протоколом й використати більше його можливостей, зокрема зберігати на усіх вузлах певної групи LUC значення версії файлу, як тільки поточний користувач його створить у відповідній папці й завантажить на віддалений сервер.
- Забрати використання власних замків в класі Contact. Краще використовувати конкурентні колекції.
- З метою покращення безпеки сервісу додати перевірку, чи отримано TCP-повідомлення від вузла з ідентичним MachineId. Такі повідомлення варто ігнорувати, оскільки це спричинить неправильну роботу протоколу Kademlia.

- Змінити класи DS й NetworkEventInvoker таким чином, щоб можна було отримувати кілька об'єктів DS.
- Додати ще рефакторинг згідно з новими знаннями.
- Розробити окрему версію сервісу виявлення, яка буде повністю абстрагована від конкретного сервісу обміну файлами.
- Зробити та опублікувати такі NuGet-пакети:
 - 1) Discovery Service – пакет, де буде міститись попередньо вказана версія проекту;
 - 2) Discovery Service with restricted message exchange – поточна версія DS-у, але з абстрагуванням від LUC-у.
 - 3) Async Socket – містить лише клас AsyncSocket;
 - 4) Connection Pool. One socket per one endpoint – поточна логіка реалізації класів ConnectionPool, ConnectionPool.Socket, ConnectionPool.Settings, BackgroundConnectionResetHelper й перелічень SocketHealth, SocketStateInPool;
 - 5) Connection Pool. Socket on demand – схожа версія поточного посиального типу ConnectionPool, проте якщо іншим потоком використовується сокет, то створюється новий з новим з'єднанням (якщо воно необхідне).

ВИСНОВКИ

В результаті виконання розробки були виконані наступні завдання.

- А. Розробити оптимізоване та безпечне виявлення вузлів мережі, навіть за відсутності Інтернету.
- Б. Реалізувати періодичне оновлення інформації про них.
- В. Додати функціональність оптимізованого скачування з них файлів. Це було досягнуто за рахунок відповідних принципів з наступної книги [69], а також конкурентного програмування, класу AsyncSocket (його перевагою є підтримка відміни операцій і зберігання поточного стану об'єкту в залежності від попередньо виконуваних методів) й власноруч розробленого пулу TCP-з'єднань відповідно до вимог сервісу обміну файлами. Цей пул завжди містить максимум одне з'єднання з однією віддаленою точкою й лише 1 потік може використовувати його. Такої роботи відповідного об'єкту не було знайдено в мережі Інтернет.
- Г. Розробити проект, що надаватиме змогу тестувати сервіс виявлення при будь-якій кількості вузлів мережі, що його використовують.
- Д. Інтегрувати розроблений сервіс в додаток обміну файлами.
- Е. Додати правки в останньому до інсталяції, синхронізації файлів й папок, логінування та потокобезпечності.

Загалом DS й LUC було протестовано в 5-х мережах:

- а) Wi-Fi мережа, мобільний Інтернет в Україні, провідне з'єднання й Docker-мережа nat – DS працює без помилок, LUC відповідно до модульних тестів - так само.
- б) Мобільний Інтернет в Ірландії – відбувається лише UDP-обмін. TCP-з'єднання не вдається встановити, оскільки на одному з ПК увімкнений CrowdStrike Falcon. Щоб це виправити, потрібно додати SSL/TLS-

підтримку, що дещо сповільнить обмін повідомленням, але значно покращить їхню безпеку.

Також були отримані наступні важливі відомості.

1) Особливості конкурентного програмування.

- Варто, як можна менше створювати спільних даних в об'єктах, оскільки в такому випадку потрібно синхронізувати до них доступ. Дотримання принципів функціонального програмування значно спрощує розробку конкурентних додатків та покращує швидкодію.
- Слід для кожного асинхронного методу, якщо він не повинен відновлюватись у вихідному контексті, використовувати метод `ConfigureAwait(false)`. Це покращує оптимізацію коду й актуально для всіх таких функцій в бібліотечному та консольному проектах.
- Перечислення слід реалізовувати типу `(S)Byte` або `(U)Int16`, оскільки в таких випадках не потрібно використовувати синхронізацію потоків для встановлення значень змінним перечислень.
- Будь-які методи, що можуть виконуватись тривалий час повинні підтримувати їхню відміну. В мові C# це завжди можна зробити.
- Тип `ValueTask` варто використовувати, лише, якщо ймовірно метод виконуватиметься синхронно або ж очікування його результату буде лише за допомогою ключового слова `await` (проте це є дещо довше, ніж з `Task`), а програма чутлива до обсягу допустимої пам'яті.

2) Особливості мережевого програмування.

- Для правильної утилізації сокету потрібно викликати метод `Shutdown` й `Dispose`. Якщо в класі-нащадку викликати процедуру-

перевантаження `Dispose(false)`, то некеровані дані не будуть знищені.

- Безпека в мережі – дуже важлива частина роботи відповідних додатків. Важливо правильно налаштовувати брандмауер, використовувати різноманітні мережеві й обміну даними протоколи, перевіряти розмір повідомлень, ігнорувати повідомлення-дублікати й порти віддалених точок, які не відповідають діапазону додатку.
- Для тестування таких додатків чудово підходить `Docker-compose`, який надає змогу легко створити кілька контейнерів на одному ПК. Загалом вміння працювати з `Docker` – важливі навички в сучасному програмуванні, оскільки він надає змогу швидко розробляти, тестувати й розгортати програми.

Отже, я зумів досягнути усіх цілей, які були поставлені, та в процесі цього отримати цінний досвід та знання, які зможу використати у подальшій кар'єрі.

ДЖЕРЕЛА

1. Cleary St. Concurrency in C# Cookbook: Asynchronous, Parallel, and Multithreaded Programming / Cleary St. 2nd ed. : Publication O'Reilly Media, 2019. – 254 p.
2. Albahari J. C# 6.0 in a Nutshell / Albahari J., Albahari B. 2nd ed. : Publication O'Reilly Media, 2015. – 1133 p.
3. Maymounkov P. Kademlia: A Peer-to-peer Information System based on the XOR Metric / Maymounkov P., Mazières D. – New York : Publication New York University. – 13 p.
4. Clifton M. The Kademlia Protocol Succinctly / Clifton M., foreword by Jebaraj D. – Morrisville: Publication Syncfusion Inc., 2018. – 194 p.
5. Kademlia (DHT) - A Practical Guide [Electronic resource] : - Available from : <https://sudonull.com/post/71900-Kademlia-DHT-A-Practical-Guide>.
6. Docker overview [Electronic resource] : - Available from : <https://docs.docker.com/get-started/overview/>.
7. Light Upon Cloud [Электронный ресурс] : - Режим доступа : <https://lightup.cloud/>.
8. GitHub discovery services [Electronic resource] : - Available from : <https://github.com/topics/service-discovery?l=c%23&o=desc&s=stars>.
9. The Abuse of Alternate Data Stream Hasn't Disappeared [Electronic resource] : - Available from : <https://www.deepinstinct.com/blog/the-abuse-of-alternate-data-stream-hasnt-disappeared>.
10. Tour of C# [Electronic resource] : - Available from : <https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/>.
11. What is .NET [Electronic resource] : - Available from : <https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet>.

- 12.Code faster Work smarter Create the future with Visual Studio 2022 [Electronic resource] : - Available from : <https://visualstudio.microsoft.com/ru/vs/>.
- 13.Microsoft .NET Framework 4.8 offline installer for Windows [Electronic resource] : - Available from : <https://support.microsoft.com/en-us/topic/microsoft-net-framework-4-8-offline-installer-for-windows-9d23f658-3b97-68ab-d013-aa3c3e7495e0>.
- 14.Що таке MAC-адреса і для чого вона потрібна [Електронний ресурс] : - Режим доступу : https://my.volia.com/kyiv/uk/faq/article/scho-take-mac-adresa-i-dlya-chogo-vona-potribna?partner=organic_search&utm_source=google&utm_medium=organic.
- 15.What is NuGet [Electronic resource] : - Available from : <https://docs.microsoft.com/en-us/nuget/what-is-nuget>.
- 16.Приклад розрахунку кількості хостів та підмереж на основі IP-адреси та маски [Електронний ресурс] : - Режим доступу : <https://help.keenetic.com/hc/uk/articles/213965829-Приклад-розрахунку-кількості-хостів-та-підмереж-на-основі-IP-адреси-та-маски>.
- 17.Радчук В. Принцип роботи протоколу TCP / Радчук В. - Тернопіль : Видавн. Тернопільський національний технічний університет ім. Івана Пулюя. – 1 с.
- 18.Контрольна сума [Електронний ресурс] : - Режим доступу : https://uk.wikipedia.org/wiki/Контрольна_сума.
- 19.IP-адреса – що це таке, навіщо потрібна та як виглядає. [Електронний ресурс] : - Режим доступу : <https://termin.in.ua/ip-adresa/>.
- 20.TCP [Електронний ресурс] : - Режим доступу : <https://uk.wikipedia.org/wiki/TCP>.

21. Why can't I use the 'await' operator within the body of a lock statement?
[Electronic resource] : - Available from :
<https://stackoverflow.com/q/7612602/13435504>.
22. Що таке мережеві протоколи? [Електронний ресурс] : - Режим доступу :
<https://tebapit.com/що-таке-мережеві-протоколи/>.
23. Датаграма [Електронний ресурс] : - Режим доступу :
<https://uk.wikipedia.org/wiki/Датаграма>.
24. Введение в сети и протоколы [Электронный ресурс] : - Режим доступа :
<https://metanit.com/sharp/net/1.1.php>.
25. Broadcasts [Electronic resource] : - Available from :
<https://www.homenethowto.com/switching/broadcasts/>.
26. Что такое DNS? [Электронный ресурс] : - Режим доступа :
<https://aws.amazon.com/ru/route53/what-is-dns/>.
27. Що таке IP-адреса? [Електронний ресурс] : - Режим доступу :
<https://www.mozilla.org/uk/products/vpn/more/what-is-an-ip-address/>.
28. ЯК: Що таке вузол в комп'ютерній мережі [Електронний ресурс] : - Режим доступу :
<https://uk.go-travels.com/39429-what-is-a-node-4155598-4160332>.
29. The Analysis of Kademlia for Random IDs [Electronic resource] : - Available from :
<http://www.tandfonline.com/doi/abs/10.1080/15427951.2015.1051674?src=recsys&journalCode=uinm20>.
30. BigInteger Struct [Electronic resource] : - Available from :
<https://docs.microsoft.com/en-us/dotnet/api/system.numerics.biginteger?view=net-6.0>.
31. What is Remote Procedure Call? [Electronic resource] : - Available from :
https://www.w3.org/History/1992/nfs_dxcern_mirror/rpc/doc/Introduction/WhatIs.html.

- 32.MySqlConnection [Electronic resource] : - Available from : <https://github.com/mysql-net/MySqlConnection/tree/master>.
- 33.Connection pooling options [Electronic resource] : - Available from : <https://github.com/mysql-net/MySqlConnection/blob/master/docs/content/connection-options.md#connection-pooling-options>.
- 34.Модель OSI [Електронний ресурс] : - Режим доступу : <http://petroonline.ho.ua/OSI.html>.
- 35.What is an endpoint? [Electronic resource] : - Available from : <https://www.cloudflare.com/learning/security/glossary/what-is-endpoint/>.
- 36.Різниця між хостом і сервером [Електронний ресурс] : - Режим доступу : <https://uk.strephonsays.com/difference-between-host-and-server>.
- 37.Що таке IPv4? [Електронний ресурс] : - Режим доступу : <https://uk.education-wiki.com/6575994-what-is-ipv4>.
- 38.Що таке протокол IPv6 і чому ви повинні звернути на нього свою увагу [Електронний ресурс] : - Режим доступу : <https://uk.wizcase.com/blog/що-таке-протокол-ipv6-і-чому-ви-повинні-звер/>.
- 39.Alethic Kademia [Electronic resource] : - Available from : <https://github.com/alethic/Alethic.Kademia/tree/master>.
- 40.Threads and threading [Electronic resource] : - Available from : <https://docs.microsoft.com/en-us/dotnet/standard/threading/threads-and-threading>.
- 41.Volatile vs. Interlocked vs. lock [Electronic resource] : - Available from : <https://stackoverflow.com/questions/154551/volatile-vs-interlocked-vs-lock>.
- 42.Volatile Class [Electronic resource] : - Available from : <https://docs.microsoft.com/en-us/dotnet/api/system.threading.volatile?view=net-6.0>.

- 43.Synchronization techniques among threads [Electronic resource] : - Available from : <https://www.ibm.com/docs/en/i/7.2?topic=techniques-synchronization-among-threads>.
- 44.lock statement (C# reference) [Electronic resource] : - Available from : <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/statements/lock>.
- 45.Interlocked Class [Electronic resource] : - Available from : <https://docs.microsoft.com/en-us/dotnet/api/system.threading.interlocked?view=net-6.0>.
- 46.Task-based asynchronous pattern (TAP) in .NET: Introduction and overview [Electronic resource] : - Available from : <https://docs.microsoft.com/uk-ua/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap>.
- 47.Use Docker Compose [Electronic resource] : - Available from : https://docs.docker.com/get-started/08_using_compose/.
- 48.Use containers to Build, Share and Run your applications [Electronic resource] : - Available from : <https://www.docker.com/resources/what-container/>.
- 49.Docker Tutorial for Beginners [Electronic resource] : - Available from : <https://www.youtube.com/watch?v=pTFZFxd4hOI>.
- 50.Одинак [Электронный ресурс] : - Режим доступа : <https://refactoring.guru/uk/design-patterns/singleton>.
- 51.Больше возможностей для работы с файлами [Электронный ресурс] : - Режим доступа : <https://www.dropbox.com/>.
- 52.Sentry (software company) [Electronic resource] : - Available from : [https://golden.com/wiki/Sentry_\(software_company\)-EAAMVBX](https://golden.com/wiki/Sentry_(software_company)-EAAMVBX).

- 53.SemaphoreSlim Class [Electronic resource] : - Available from :
<https://docs.microsoft.com/en-us/dotnet/api/system.threading.semaphoreslim?view=net-6.0>.
- 54.LUC.DVVSets [Electronic resource] : - Available from :
<https://libraries.io/nuget/LUC.DVVSets>.
- 55.Tasks and Parallelism: The New Wave of Multithreading [Electronic resource]
 : - Available from : <https://www.codemag.com/article/1211071/Tasks-and-Parallelism-The-New-Wave-of-Multithreading>.
- 56.What is a "thread" (really)? [Electronic resource] : - Available from :
<https://stackoverflow.com/questions/5201852/what-is-a-thread-really>.
- 57.VideoInspector [Electronic resource] : - Available from :
<https://www.techspot.com/downloads/480-videoinspector.html#certified>.
- 58.Inno Setup [Electronic resource] : - Available from :
<https://jrsoftware.org/isinfo.php>.
- 59.IGMP [Электронный ресурс] : - Режим доступа :
<https://uk.wikipedia.org/wiki/IGMP>.
- 60.Принципы работы протокола PIM [Электронный ресурс] : - Режим доступа :
<https://habr.com/ru/post/450582/>.
- 61.Windows containers [Electronic resource] : - Available from :
<https://www.techtarget.com/searchwindowsserver/definition/Microsoft-Windows-Containers>.
- 62.Use bridge networks [Electronic resource] : - Available from :
<https://docs.docker.com/network/bridge/>.
- 63.Docker Registry [Electronic resource] : - Available from :
<https://www.aquasec.com/cloud-native-academy/docker-container/docker-registry/>.

- 64.Файл с расширением .csproj [Электронный ресурс] : - Режим доступа : <https://open-file.ru/types/csproj.>
- 65.Компанія CrowdStrike [Электронный ресурс] : - Режим доступа : <https://iitd.com.ua/crowdstrike/>.
- 66.Multicast Explained in 5 Minutes | CCIE Journey for Week 6-12-2020 [Electronic resource] : - Available from : <https://youtu.be/W5oMvrMRM3Q>.
- 67.Dependency injection [Электронный ресурс] : - Режим доступа : <https://habr.com/ru/post/350068/>.
- 68.Фасад [Электронный ресурс] : - Режим доступа : <https://refactoring.guru/uk/design-patterns/facade>.
- 69.McConnell S. Complete Code. Master Class / McConnell S. 2nd ed. : Publication Person Education, 2004. – 916 p.
- 70.DiscoveryService integrated-into-luc [Electronic resource] : - Available from : <https://github.com/bob-byte/DiscoveryService/tree/release/integrated-into-luc>.
- 71.What are solutions and projects in Visual Studio? [Electronic resource] : - Available from : <https://docs.microsoft.com/en-us/visualstudio/ide/solutions-and-projects-in-visual-studio?view=vs-2022>.
- 72.NetCoreServer [Electronic resource] : - Available from : <https://github.com/chronoxor/NetCoreServer>.
- 73.Dll vs pdb file [Electronic resource] : - Available from : <https://bytes.com/topic/c-sharp/answers/247246-dll-vs-pdb-file>.
- 74.S.O.L.I.D. Principles of Object-Oriented Programming in C# [Electronic resource] : - Available from : <https://www.educative.io/blog/solid-principles-oop-c-sharp>.
- 75.Абстрактна фабрика на C# [Электронный ресурс] : - Режим доступа : <https://refactoring.guru/uk/design-patterns/abstract-factory/csharp/example>.

- 76.Sync files business unlimited [Electronic resource] : - Available from : <https://lightup.cloud/offer/sync-files-business-unlimited/>.
- 77.Адаптер на С# [Електронний ресурс] : - Режим доступу : <https://refactoring.guru/uk/design-patterns/adapter/csharp/example>.
- 78.CompositionContainer Class [Electronic resource] : - Available from : <https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.composition.hosting.compositioncontainer?view=dotnet-plat-ext-6.0>.
- 79.UnityContainer Class [Electronic resource] : - Available from : [https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ee650995\(v=pandp.10\)](https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ee650995(v=pandp.10)).
- 80.Конвеції в LUC [Електронний ресурс] : - Режим доступу : <https://docs.google.com/document/d/1glA6max8Be7p0jDsJduGVXWldCBCJQI/edit?usp=sharing&oid=110006529788053704421&rtpof=true&sd=true>.
- 81.LUC Files [Electronic resource] : - Available from : <https://lightup.cloud/riak/>.
- 82.Unit Testing and the Arrange, Act and Assert (AAA) Pattern [Electronic resource] : - Available from : <https://medium.com/@pjbfg/title-testing-code-ocd-and-the-aaa-pattern-df453975ab80>.
- 83.Debug apps in a local Docker container [Electronic resource] : - Available from : <https://docs.microsoft.com/en-us/visualstudio/containers/edit-and-refresh?view=vs-2022>.
- 84.Docker Desktop overview [Electronic resource] : - Available from : <https://docs.docker.com/desktop/>.
- 85.Introduction to Hyper-V on Windows 10 [Electronic resource] : - Available from : <https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/>.