**ДНІПРОВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ОЛЕСЯ ГОНЧАРА**

Кафедра електронних обчислювальних машин

(повна назва кафедри)

**КУРСОВА РОБОТА**

з дисципліни «Системне програмне забезпечення»

(назва дисципліни)

на тему:

Студента ІІI курсу групи КІ-15-2

напряму підготовки 6.050102

«Комп’ютерна інженерія»

(прізвище та ініціали)

Керiвник:

Кількість балів

Національна шкала

Оцінка ECTS

Члени комісії:

(підпис)

(прізвище та ініціали)

(підпис)

(прізвище та ініціали)

(підпис)

(прізвище та ініціали)

м. Дніпро, 2017р.

**Зміст**

[**Теоретична частина** 4](#_Toc501071843)

[**Потоки** 4](#_Toc501071844)

[1.1. Визначення потоку 4](#_Toc501071845)

[1.2. Створення потоку 8](#_Toc501071846)

[1.3. Завершення потоку 9](#_Toc501071847)

[1.4. Зупинка та відновлння потоку 10](#_Toc501071848)

[1.5. Пріоритети потоків 10](#_Toc501071849)

[1.6. Визначення моменту закінчення потоку 11](#_Toc501071850)

[1.7. Передача аргументу потоку 12](#_Toc501071851)

[1.8. Властивість IsBackground 12](#_Toc501071852)

[**Процеси** 13](#_Toc501071853)

[2.1. Визначення процеса 13](#_Toc501071854)

[2.2. Запуск процесу 14](#_Toc501071855)

[2.3. Завершення процесу 14](#_Toc501071856)

[**Синхронізація** 15](#_Toc501071857)

[3.1. Визначення синхронізації 15](#_Toc501071858)

[3.2. М'ютекс 16](#_Toc501071859)

[3.3. Семафор 17](#_Toc501071860)

[3.4. Події 19](#_Toc501071861)

[**Обмін між процесами** 21](#_Toc501071862)

[4.1. Способи передачі даних між процесами 21](#_Toc501071863)

[4.2. Зв'язки між процесами 23](#_Toc501071864)

[4.3. Операції з каналами в .NET Framework 23](#_Toc501071865)

[**Блокчейн** 25](#_Toc501071866)

[5.1. Загальні відомості і основні поняття 25](#_Toc501071867)

[5.2. Майнінг. Proof-of-Work та Proof-of-Stake. 26](#_Toc501071868)

[5.3. Винагорода і генерація нових монет. 26](#_Toc501071869)

[5.4. Синхронізація і вирішення конфліктів. 27](#_Toc501071870)

[**Практична частина** 28](#_Toc501071871)

[1. Блокчейн 28](#_Toc501071872)

[2. Коммунікація 34](#_Toc501071873)

[3. Тестування 42](#_Toc501071874)

[4. Інтерфейс користувача 49](#_Toc501071875)

[5. Приклад синхронізації після операцій у двох блокчейнах у різних процесах 57](#_Toc501071876)

[6. GitHub 57](#_Toc501071877)

[**Висновок** 58](#_Toc501071878)

[**Список використаної літератури** 59](#_Toc501071879)

# **Теоретична частина**

# **Потоки**

## 1.1. Визначення потоку

Потік (thread) являє собою незалежну послідовність інструкцій в програмі. Потоки грають важливу роль як для клієнтських, так і для серверних додатків. Наприклад, під час введення якогось коду C# у вікні редактора Visual Studio проводиться аналіз на предмет різних синтаксичних помилок. Цей аналіз здійснюється окремим фоновим потоком. Те ж саме відбувається і в засобі перевірки орфографії в Microsoft Word. Один потік очікує введення даних користувачем, а інший в цей час виконує у фоновому режимі деякий аналіз. Третій потік може зберігати записуються дані в тимчасовий файл, а четвертий - завантажувати додаткові дані з Інтернету.

У додатку, яке функціонує на сервері, один потік завжди очікує надходження запиту від клієнта і тому називається потоком-слухачем (listener thread). При отриманні запиту він відразу ж пересилає його окремому робочому потоку (worker thread), який далі сам продовжує взаємодіяти з клієнтом. Потік-слухач після цього повинен бути відшкодований до своїх обов'язків по очікуванню надходження наступного запиту від чергового клієнта.

Кожен процес складається з ресурсів, таких як віконні дескриптори, файлові дескриптори і інші об'єкти ядра, має виділену область у віртуальній пам'яті і містить як мінімум один потік. Потоки плануються до виконання операційною системою. У будь-якого потоку є пріоритет, лічильник команд, який вказує на місце в програмі, де відбувається обробка, і стек, в якому зберігаються локальні змінні потоку. Стек у кожного потоку виглядає по-своєму, але пам'ять для програмного коду і купа поділяються серед всіх потоків, які функціонують усередині одного процесу.

Це дозволяє потокам всередині одного процесу швидко взаємодіяти між собою, оскільки всі потоки процесу звертаються до однієї і тієї ж віртуальної пам'яті. Однак це також і ускладнює справу, оскільки дає можливість безлічі потоків змінювати одну і ту ж область пам'яті.

Розрізняють два різновиди багатозадачності:

* на основі процесів
* на основі потоків.

**Процес** відповідає за управління ресурсами, до числа яких належить віртуальна пам'ять і дескриптори Windows, і містить як мінімум один потік. Наявність хоча б одного потоку є обов'язковим для виконання будь-якої програми.

Тому багатозадачність на основі процесів - це засіб, завдяки якому на комп'ютері можуть паралельно виконуватися дві програми і більш.

Так, багатозадачність на основі процесів дозволяє одночасно виконувати програми текстового редактора, електронних таблиць і перегляду вмісту в Інтернеті. При організації багатозадачності на основі процесів програма є найменшою одиницею коду, виконання якої може координувати планувальник завдань.

**Потік** являє собою координовану одиницю виконуваного коду. Своїм походженням цей термін зобов'язаний поняттю "потік виконання". При організації багатозадачності на основі потоків у кожного процесу повинен бути принаймні один потік, хоча їх може бути і більше. Це означає, що в одній програмі одночасно можуть вирішуватися два завдання і більше. Наприклад, текст може форматироваться в редакторі тексту одночасно з його висновком на друк, за умови, що обидва ці дії виконуються в двох окремих потоках.

Відмінності в багатозадачності на основі процесів і потоків можуть бути зведені до наступного:

багатозадачність на основі процесів організується для паралельного виконання програм, а

багатозадачність на основі потоків - для паралельного виконання окремих частин однієї програми.

Головна перевага багатопотокової обробки полягає в тому, що вона дозволяє писати програми, які працюють дуже ефективно завдяки можливості вигідно використовувати час простою, що неминуче виникає в ході виконання більшості програм. Як відомо, більшість пристроїв введення-виведення, будь то пристрої, підключені до мережевих портів, накопичувачі на дисках або клавіатура, працюють набагато повільніше, ніж центральний процесор (ЦП). Тому більшу частину свого часу програмі доводиться очікувати відправки даних на пристрій вводу-виводу або прийому інформації з нього. А завдяки многопоточной обробці програма може вирішувати якусь іншу задачу під час вимушеного простою.

Наприклад, в той час як одна частина програми відправляє файл через з'єднання з Інтернетом, інша її частина може виконувати читання текстової інформації, що вводиться з клавіатури, а третя - здійснювати буферизацию чергового блоку даних, що відправляється.

Потік може перебувати в одному з декількох станів. Потік може бути:

* що виконуються;
* готовим до виконання, як тільки він отримає час і ресурси ЦП;
* призупиненим, тобто тимчасово що не виконується, відновленим в подальшому;
* заблокованим в очікуванні ресурсів для свого виконання;
* завершенним, коли його виконання закінчено і не може бути відновлено.

У середовищі .NET Framework визначені два різновиди потоків:

***пріоритетний і фоновий***.

За замовчуванням створюється потік автоматично стає пріоритетним, але його можна зробити фоновим. Єдина відмінність пріоритетних потоків від фонових полягає в тому, що фоновий потік автоматично завершується, якщо в його процесі зупинені всі пріоритетні потоки.

У зв'язку з організацією багатозадачності на основі потоків виникає потреба в особливого роду режимі, який називається **синхронізацією** і дозволяє координувати виконання потоків цілком певним чином. Для такої синхронізації в C# передбачена окрема підсистема.

Всі процеси складаються хоча б з одного потоку, який зазвичай називають основним, оскільки саме з нього починається виконання програми. З основного потоку можна створити інші потоки.

У мові C# і середовищі .NET Framework підтримуються обидва різновиди багатозадачності: на основі процесів і на основі потоків. Тому засобами C# можна створювати як процеси, так і потоки, а також керувати і тими і іншими. Для того щоб почати новий процес, від програмиста потрібно зовсім небагато зусиль, оскільки кожен попередній процес абсолютно відокремлений від наступного.

Набагато важливішою надається підтримка в C# багатопотокової обробки, завдяки якій спрощується написання високопродуктивних, багатопоточних програм на C# в порівнянні з деякими іншими мовами програмування.

Класи, які підтримують багатопотокове програмування, визначені в просторі імен System.Threading. Тому будь-яка многопоточная програма на C# включає в себе наступний рядок коду:

*using System.Threading;*

Простір імен System.Threading містить різні типи, що дозволяють створювати багатопотокові програми. Мабуть, головним серед них є клас Thread, оскільки він представляє окремий потік. Щоб програмно отримати посилання на потік, що виконується конкретним його примірником, просто викличте статичну властивість Thread.CurrentThread:

*static void ExtractExecutingThread()*

*{ Thread currThread = Thread.CurrentThread; }*

На платформі .NET не існує прямої відповідності "один до одного" між доменами додатків (AppDomain) і потоками. Фактично певний AppDomain може мати кілька потоків, що виконуються в кожен конкретний момент часу. Більш того, конкретний потік не прив'язаний до одного домену додатків протягом свого часу існування. Потоки можуть перетинати кордони доменів додатків, коли це заманеться планувальником Windows і CLR.

Хоча активні потоки можуть перетинати кордони AppDomain, кожен потік в кожен конкретний момент часу може виконуватися тільки всередині одного домену додатків (іншими словами, неможливо, щоб один потік працював в більш ніж одному домені додатків відразу). Щоб програмно отримати доступ до AppDomain, в якому працює поточний потік, треба викликати статичний метод Thread.GetDomain():

*static void ExtractAppDomainHostingThread()*

*{ AppDomain ad = Thread.GetDomain(); }*

Єдиний потік також в будь-який момент може бути переміщений в певний контекст, і він може переміщатися в межах нового контексту за примхою CLR. Для отримання поточного контексту, в якому виконується потік, використовуйте статичну властивість Thread.CurrentContext (яке повертає об'єкт System.Runtime.Remoting.Contexts.Context):

*static void ExtractCurrentThreadContext()*

*{ Context ctx = Thread.CurrentContext; }*

## 1.2. Створення потоку

Для створення потоку досить отримати екземпляр об'єкта типу Thread, тобто класу, визначеного в просторі імен System.Threading. Нижче наведена найпростіша форма конструктора класу Thread:

*public Thread (ThreadStart запуск)*

де *запуск* - це ім'я методу, що викликається з метою почати виконання потоку, a ThreadStart - делегат, певний в середовищі .NET Framework, як показано нижче.

*public delegate void ThreadStart()*

Отже, метод, що вказується в якості точки входу в потік, повинен повертати тип void і не приймати ніяких аргументів.

Новостворений новий потік не почне виконуватися до того часу, поки не буде викликаний його метод Start(), який визначається в класі Thread.

Одного разу почавшись, потік буде виконуватися до того часу, поки не відбудеться повернення з методу, на який вказує запуск. Таким чином, після повернення з цього методу потік автоматично припиняється. Якщо ж спробувати викликати метод Start() для потоку, який вже почався, це призведе до генерування винятку ThreadStateException.

## 1.3. Завершення потоку

Іноді потік корисно перервати до його нормального завершення. Наприклад, отладчику може знадобитися перервати потік, який вийшов з-під контролю. Після переривання потік видаляється з системи і не може бути розпочато знову.

Для переривання потоку до його нормального завершення служить метод Thread.Abort(). Нижче наведена найпростіша форма цього методу.

*public void Abort()*

Метод Abort() створює необхідні умови для генерування винятку ThreadAbortException в тому потоці, для якого він був викликаний. Цей виняток призводить до переривання потоку і може бути перехоплено і в коді програми, але в цьому випадку він автоматично генерується ще раз, щоб зупинити потік. Метод Abort() не завжди здатний зупинити потік негайно, тому якщо потік потрібно зупинити перед тим, як продовжити виконання програми, то після методу Abort() слід відразу ж викликати метод Join(). Крім того, в найрідкісніших випадках методу Abort() взагалі не вдається зупинити потік. Це відбувається, наприклад, в тому випадку, якщо кодовий блок finally входить в нескінченний цикл.

Запит на передчасне переривання може бути перевизначений в самому потоці. Для цього необхідно спочатку перехопити в потоці виключення ThreadAbortException, а потім викликати метод ResetAbort(). Завдяки цьому виключається повторне генерування винятку по завершенні обробника виключення, який перериває цей потік. Нижче наведена форма оголошення методу ResetAbort().

*public static void ResetAbort()*

Виклик методу ResetAbort() може завершитися невдало, якщо в потоці відсутній належний режим надійного скасування передчасного переривання потоку.

## 1.4. Зупинка та відновлння потоку

У початкових версіях середовища .NET Framework потік можна було призупинити викликом методу Thread.Suspend() і відновити викликом методу Thread.Resume(). Але тепер, обидва ці методи вважаються застарілими і не рекомендуються до застосування. Пояснюється це зокрема тим, що користуватися методом Suspend() насправді небезпечно, так як з його допомогою можна призупинити потік, який в даний момент утримує блокування, що перешкоджає його зняттю, а отже, призводить до взаімоблокуванню. Застосування обох методів може стати причиною серйозних ускладнень на рівні системи. Тому для припинення та поновлення потоку слід використовувати інші засоби синхронізації, в тому числі м'ютекс і семафор.

## 1.5. Пріоритети потоків

У кожного потока є свій **пріоритет**, який певною мірою визначає, наскільки часто потік отримує доступ до ЦП. Взагалі кажучи, фонові потоки отримують доступ до ЦП рідше, ніж високопріоритетні. Таким чином, протягом заданого проміжку часу фоновому потоку буде доступно менше часу ЦП, ніж високопріоритетним. Як і слід було очікувати, час ЦП, що отримується потоком, справляє визначальний вплив на характер його виконання і взаємодії з іншими потоками, виконуваними в даний момент в системі.

Слід мати на увазі, що, крім пріоритету, на частоту доступу потоку до ЦП впливають і інші фактори. Так, якщо високопріоритетний потік очікує доступу до деякого ресурсу, наприклад для введення з клавіатури, він блокується, а замість нього виконується фоновий потік. У подібній ситуації фоновий потік може отримувати доступ до ЦП частіше, ніж високопріоритетний потік протягом певного періоду часу. І нарешті, конкретне планування завдань на рівні операційної системи також впливає на час ЦП, що виділяється для потока.

Коли породжений потік починає виконуватися, він отримує пріоритет, який встановлюється за умовчанням. Пріоритет потоку можна змінити за допомогою властивості Priority, що є членом класу Thread. Нижче наведена загальна форма даного властивості:

*public ThreadPriority Priority {get; set;}*

де ThreadPriority позначає перерахування, в якому визначаються наведені нижче значення пріоритетів.

## 1.6. Визначення моменту закінчення потоку

В класі Thread є два інших способа для визначення моменту закінчення потоку. Доступна тільки для читання властивість IsAlive, яка визначається наступним чином.

*public bool IsAlive {get;}*

Властивість IsAlive повертає логічне значення true, якщо потік, для якого воно викликається, як і раніше виконується.

Ще один спосіб відстеження моменту закінчення складається у виклику методу Join(). Нижче наведена його найпростіша форма.

*public void Join()*

Метод Join() очікує поки потік, для якого він був викликаний, не завершиться. Його ім'я відображає принцип очікування до того часу, поки потік, що викликає не приєднається до викликаного методу. Якщо ж даний потік не було розпочато, то генерується виключення ThreadStateException. В інших формах методу Join() можна вказати максимальний період часу, протягом якого слід очікувати завершення зазначеного потоку.

## 1.7. Передача аргументу потоку

Аргумент передається потоку в наступній формі методу Start().

*public void Start (object параметр)*

Об'єкт, що вказується в якості аргументу *параметр*, автоматично передається методу, що виконує роль точки входу в потік. Отже, для того щоб передати аргумент потоку, досить передати його методу Start().

Для застосування параметризованої форми методу Start() буде потрібно наступна форма конструктора класу Thread:

*public Thread (ParameterizedThreadStart запуск)*

де запуск позначає метод, що викликається з метою почати виконання потоку.

В даному випадку ParameterizedThreadStart є делегатом, що декларують в такий спосіб.

*public delegate void ParameterizedThreadStart (object obj)*

Делегат приймає аргумент типу object. Тому для правильного застосування даної форми конструктора класу Thread, у методі в якості точки входу в потік повинен бути параметр типу object.

## 1.8. Властивість IsBackground

В середовищі .NET Framework визначені два різновиди потоків: пріоритетний і фоновий. Єдина відмінність між ними полягає в тому, що процес не завершиться до того часу, поки не закінчиться пріоритетний потік, тоді як фонові потоки завершуються автоматично після закінчення всіх пріоритетних потоків. За замовчуванням створюванний потік стає пріоритетним, але його можна зробити фоновим, використовуючи властивість IsBackground в класі Thread:

*public bool IsBackground {get; set;}*

Для того щоб зробити потік фоновим, досить привласнити логічне значення true властивості IsBackground. А логічне значення false вказує на те, що потік є пріоритетним.

# **Процеси**

## 2.1. Визначення процеса

Багатозадачність на основі потоків найчастіше організовується при програмуванні на С#. Але там, де це доречно, можна організувати і багатозадачність на основі процесів. У цьому випадку замість запуску іншого потоку в одній і тій же програмі одна програма починає виконання іншої. При програмуванні на C# це робиться за допомогою класу Process, визначеного в просторі імен System.Diagnostics.

Єдиний потік також в будь-який момент може бути переміщений в певний контекст, і він може переміщатися в межах нового контексту за примхою CLR. Для отримання поточного контексту, в якому виконується потік, використовуйте статичну властивість Thread.CurrentContext.

У Windows під **процесом** розуміється об'єкт ядра, якому належать системні ресурси, використовувані виконуваним додатком. Тому можна сказати, що в Windows процесом є додаток, що виконується. Виконання кожного процесу починається з первинного потоку. Під час свого виконання процес може створювати інші потоки. Ісполненя процесу закінчується при завершенні роботи всіх його потоків. Кожний процес в операційній системі Windows володіє наступними ресурсами:

* віртуальним адресним простором;
* робочою безліччю сторінок в реальній пам'яті;
* маркером доступу, що містить інформацію для системи безпеки;
* таблицею для зберігання дескрипторів об'єктів ядра.

Крім дескриптора, кожен процес в Windows має свій ідентифікатор, який є унікальним для процесів, що виконуються в системі. Ідентифікатори процесів використовуються, головним чином, службовими програмами, які дозволяють користувачам системи відстежувати роботу процесів.

Іноді процесу потрібно знати свій дескриптор, щоб змінити якісь свої характеристики. Наприклад, процес може змінити свій пріоритет.

Псевдодескриптор процесу відрізняється від справжнього дескриптора процесу тим, що він може використовуватися тільки поточним процесом і не може успадковуватися іншими процесами. Псевдодескриптор процесу не потрібно закривати після його використання. З псевдодескриптором процесу можна отримати справжній дескриптор процесу.

## 2.2. Запуск процесу

Найпростіший спосіб запустити інший процес - скористатися методом Start(), визначених у класі Process. Нижче наведена одна з найпростіших форм цього методу:

public static Process Start (string имя\_файла)

де имя\_файла позначає конкретне ім'я файлу, який повинен виконуватися або ж пов'язаний з виконуваним файлом.

Процес, який створює новий процес, називається батьківським процесом (parent process) по відношенню до створюваного процесу. Новий же процес, який створюється іншим процесом, називається дочірнім процесом (child process) по відношенню до процесу-батькові.

## 2.3. Завершення процесу

Коли створений процес завершується, слід викликати метод Close(), щоб звільнити пам'ять, виділену для цього процесу. Нижче наведена форма оголошення методу Close().

public void Close()

# **Синхронізація**

## 3.1. Визначення синхронізації

Синхронізація процесів - це є досягнення деякого фіксованого порядку між сигналами, якими обмінюються ці процеси. У програмуванні розглядаються паралельні процеси, які є програмами, що виконуються процесором.

Усі потоки, що належать одному процесу, поділяють деякі загальні ресурси - такі, як адресний простір оперативної пам'яті або відкриті файли. Ці ресурси належать всьому процесу, а значить, і кожному його потоку. Отже, кожний поток може працювати з цими ресурсами без будь-яких обмежень. Але ... Якщо один потік ще не закінчив працювати з будь-яким загальним ресурсом, а система переключилася на інший потік, який використовує цей же ресурс, то результат роботи цих потоків може надзвичайно сильно відрізнятися від задуманого. Такі конфлікти можуть виникнути і між потоками, що належать різним процесам. Завжди, коли два або більше потоків використовують будь-якої загальний ресурс, виникає ця проблема.

Саме тому необхідний механізм, що дозволяє потокам погоджувати свою роботу з загальними ресурсами. Цей механізм отримав назву механізму синхронізації потоків.

Цей механізм являє собою набір об'єктів операційної системи, які створюються і управляються програмно, є загальними для всіх потоків в системі (деякі - для потоків, що належать одному процесу) і використовуються для координування доступу до ресурсів. Як ресурси може виступати все, що може бути загальним для двох і більше потоків: файл на диску, порт, запис в базі даних, об'єкт GDI, і навіть глобальна змінна програми (яка може бути доступна з потоків, що належать одному процесу).

Об'єктів синхронізації існує кілька, найважливіші з них:

- взаємовиключення (mutex),

- подія (event),

- семафор (semaphore).

Кожен з цих об'єктів реалізує свій спосіб синхронізації. Також в якості об'єктів синхронізації можуть використовуватися самі процеси і потоки (коли один поток чекає завершення іншого потоку або процесу); а також файли, комунікаційні пристрої, консольне введення і повідомлення про зміну.

Будь-який об'єкт синхронізації може перебувати в так званому сигнальному стані. Для кожного типу об'єктів цей стан має різний зміст. Потоки можуть перевіряти поточний стан об'єкта та / або чекати зміни цього стану і таким чином узгоджувати свої дії. При цьому гарантується, що коли поток працює з об'єктами синхронізації (створює їх, змінює стан) система не перерве його виконання, поки він не завершить цю дію.

## 3.2. М'ютекс

**М'ютекс** є взаємовиключний синхронизирующий об'єкт. Це означає, що він може бути отриманий потоком тільки по черзі. М'ютекс призначений для тих ситуацій, в яких загальний ресурс може бути одночасно використаний тільки в одному потоці. Припустимо, що системний журнал спільно використовується в декількох процесах, але тільки в одному з них дані можуть записуватися в файл цього журналу в будь-який момент часу. Для синхронізації процесів в даній ситуації ідеально підходить м'ютекс.

М'ютекс підтримується в класі System.Threading.Mutex. У нього є кілька конструкторів. Нижче наведені два найбільш уживаних конструктора.

*public Mutex()*

*public Mutex (bool initiallyOwned)*

У першій формі конструктора створюється м'ютекс, яким спочатку ніхто не володіє. А в другій формі вихідним станом мьютекса заволодіває потік, що викликає, якщо параметр initiallyOwned має логічне значення true. В іншому випадку м'ютексом ніхто не володіє.

Для того щоб отримати м'ютекс, в коді програми слід викликати метод WaitOne() для цього мьютекса. Метод WaitOne() успадковується класом Mutex від класу Thread.WaitHandle. Нижче наведена його найпростіша форма.

*public bool WaitOne();*

Метод WaitOne() очікує до того часу, поки не буде отримано м'ютекс, для якого він був викликаний. Отже, цей метод блокує виконання потоку, що викликає до того часу, поки не стане доступним вказаний м'ютекс. Він завжди повертає логічне значення true.

Коли ж в коді більше не потрібно володіти м'ютексом, він звільняється за допомогою виклику методу ReleaseMutex(), форма якого наведена нижче.

*public void ReleaseMutex()*

У цій формі метод ReleaseMutex() звільняє мьютекс, для якого він був викликаний, що дає можливість іншому потоку отримати даний м'ютекс.

Для застосування мьютекса з метою синхронізувати доступ до загального ресурсу згадані вище методи WaitOne() і ReleaseMutex() використовуються так, як показано в наведеному нижче фрагменті коду.

*Mutex myMtx = new Mutex();*

*// ...*

*myMtx.WaitOne(); // очікувати отримання мьютекса*

*// Отримати доступ до загального ресурсу.*

*myMtx.ReleaseMutex(); // звільнити м'ютекс*

При виклику методу WaitOne() виконання відповідного потоку призупиняється до того часу, поки не буде отримано м'ютекс. А при виклику методу ReleaseMutex() м'ютекс звільняється, й потім може бути отриманий іншим потоком. Завдяки такому підходу до синхронізації одночасний доступ до загального ресурсу обмежується тільки одним потоком.

## 3.3. Семафор

Семафор подібний м'ютексу, за винятком того, що він надає одночасний доступ до загального ресурсу не одному, а кільком потокам. Тому семафор придатний для синхронізації цілого ряду ресурсів. Семафор управляє доступом до загального ресурсу, використовуючи для цієї мети лічильник. Якщо значення лічильника більше нуля, то доступ до ресурсу дозволений. А якщо це значення дорівнює нулю, то доступ до ресурсу заборонено. За допомогою лічильника ведеться підрахунок кількості дозволів. Отже, для доступу до ресурсу потік повинен отримати дозвіл від семафора.

Зазвичай потік, якому потрібен доступ до загального ресурсу, намагається отримати дозвіл від семафора. Якщо значення лічильника семафора більше нуля, то потік отримує дозвіл, а лічильник семафора декрементируется. В іншому випадку потік блокується до того часу, поки не отримає дозвіл. Коли ж потоку більше не потрібен доступ до загального ресурсу, він вивільняє дозвіл, а лічильник семафора инкрементируется. Якщо дозволу чекає інший потік, то він отримує його в цей момент. Кількість одночасно дозволених доступів вказується при створенні семафора. Так, якщо створити семафор, який одночасно дозволяє тільки один доступ, то такий семафор буде діяти як м'ютекс.

Семафори особливо корисні в тих випадках, коли загальний ресурс складається з групи або пулу ресурсів. Наприклад, пул ресурсів може складатися з цілого ряду мережевих з'єднань, кожне з яких служить для передачі даних. Потоку, якому потрібно мережеве з'єднання, все одно, яке саме з'єднання він отримає. В даному випадку семафор забезпечує зручний механізм управління доступом до мережевих з'єднань.

Семафор реалізується в класі System.Threading.Semaphore, у якого є декілька конструкторів. Нижче наведена найпростіша форма конструктора даного класу:

*public Semaphore(int initialCount, int maximumCount)*

де initialCount - це початкове значення для лічильника дозволів семафора, тобто кількість доступних дозволів;

maximumCount - максимальне значення даного лічильника, тобто максимальна кількість дозволів, які може дати семафор.

Семафор застосовується таким же чином, як і описаний раніше м'ютекс. З метою отримання доступу до ресурсу в коді програми викликається метод WaitOne() для семафора. Цей метод успадковується класом Semaphore від класу WaitHandle. Метод WaitOne() очікує до того часу, поки не буде отримано семафор, для якого він викликається. Таким чином, він блокує виконання потоку, що викликає до того часу, поки вказаний семафор не надасть дозвіл на доступ до ресурсу.

Якщо коду більше не потрібно володіти семафором, він звільняє його, викликаючи метод Release(). Нижче наведено дві форми цього методу:

*public int Release()*

*public int Release(int releaseCount)*

У першій формі метод Release() вивільняє тільки один дозвіл, а в другій формі - кількість дозволів, які визначаються параметром releaseCount. В обох формах даний метод повертає підраховану кількість дозволів, які існували до вивільнення.

В .NET пропонується два класи з функціональністю семафора:

Semaphore і

SemaphoreSlim.

Клас Semaphore може бути іменований, використовувати ресурси в масштабі всієї системи і забезпечувати синхронізацію між різними процесами. Клас SemaphoreSlim є полегшеною версією класу Semaphore, яка оптимізована для забезпечення більш короткого часу очікування.

## 3.4. Події

Події є ще один ресурс для забезпечення синхронізації в масштабі всієї системи.

Для використання системних подій з керованого коду, .NET Framework пропонує класи ManualResetEvent, AutoResetEvent, ManualResetEventSlim і CountdownEvent, які знаходяться в просторі імен System.Threading. Класи ManualResetEventSlim і CountdownEvent з'явилися у версії .NET 4.

Ці класи є похідними від класу EventWaitHandle, що знаходиться на верхньому рівні ієрархії класів, і застосовуються в тих випадках, коли один потік чекає появи деякої події в іншому потоці. Як тільки така подія з'являється, другий потік повідомляє про нього перший потік, дозволяючи тим самим відновити його виконання.

Нижче наведені конструктори класів ManualResetEvent і AutoResetEvent:

*public ManualResetEvent(bool initialState)*

*public AutoResetEvent(bool initialState)*

Якщо в обох формах параметр initialState має логічне значення true, то про подію спочатку повідомляється. А якщо він має логічне значення false, то про подію спочатку не буде повідомлено.

Для події типу ManualResetEvent порядок застосування наступний. Потік, що очікує деяку подію, викликає метод WaitOne() для подієвого об'єкта, що представляє дану подію. Якщо подієвий об'єкт знаходиться в сигнальному стані, то відбувається негайне повернення з методу WaitOne(). В іншому випадку виконання потоку, що викликає призупиняється до того часу, поки не буде отримано повідомлення про подію. Як тільки подія відбудеться в іншому потоці, цей потік встановить подієвий об'єкт в сигнальний стан, викликавши метод Set(). Тому метод Set() слід розглядати як повідомляє про те, що подія відбулася.

Після установки подієвого об'єкта в сигнальний стан станеться негайне повернення з методу WaitOne(), і перший потік відновить своє виконання. А в результаті виклику методу Reset() подієвий об'єкт повертається в несигнальному стан.

Подія AutoResetEvent відрізняється від події типу ManualResetEvent лише способом установки в початковий стан. Якщо, для події типу ManualResetEvent подієвий об'єкт залишається в сигнальному стані доти, поки не буде викликаний метод Reset(), то для події типу AutoResetEvent подієвий об'єкт автоматично переходить в несигнальному стан, як тільки потік, який чекає цю подію, отримає повідомлення про нього і відновить своє виконання. Тому якщо застосовується подія типу AutoResetEvent, то викликати метод Reset() необов'язково .

Подія ManualResetEventSlim перекладається в сигнальний стан викликом методу Set(), а за допомогою Reset() повертається назад в несигнальному стан. У разі виклику методу Set() при наявності безлічі потоків, які чекають переходу події в сигнальний стан, очікування всіх цих потоків негайно припиняється. У разі, якщо потік просто викликає метод WaitOne(), а подія вже знаходиться в сигнальному стані, потік, який чекав, може відразу ж продовжити роботу.

# **Обмін між процесами**

## 4.1. Способи передачі даних між процесами

Під обміном даними між паралельними процесами розуміється пересилання даних від одного потоку до іншого потоку, припускаючи, що ці потоки виконуються в контекстах різних процесів. Потік, який посилає дані іншому потоку, називається відправником. Потік, який отримує дані від іншого потоку, називається адресатом або одержувачем.

Якщо потоки виконуються в одному процесі, то для обміну даними між ними можна використовувати глобальні змінні і засоби синхронізації потоків. Насправді все складніше в тому випадку, якщо потоки виконуються в різних процесах - потоки не можуть звертатися до загальних змінним і для обміну даними між ними існують спеціальні засоби операційної системи.

Канал даних включає вхідний і вихідний буфери пам'яті, потоки ядра операційної системи і загальну пам'ять, доступ до якої мають обидва потоки ядра. Працює канал передачі даних в такий спосіб:  
- перший потік ядра ОС читає дані з вхідного буфера В1 і записує їх в загальну пам'ять М;  
- другий потік ядра читає дані із загальної пам'яті М і записує їх в буфер В2.

Призначені для користувача потоки Т1 і T2 за допомогою виклику функцій ядра операційної системи мають доступ до буферам В1 і В2 відповідно. Тому пересилання даних з потоку Т1 в потік Т2 відбувається наступним чином:  
 призначений для користувача потік Т1 записує дані в буфер В1, використовуючи спеціальну функцію ядра операційної системи;  
потік К1 ядра операційної системи читає дані з буфера В1 і записує їх в загальну пам'ять М;  
потік K2 ядра операційної системи читає дані із загальної пам'яті М і записує їх в буфер В2.  
призначений для користувача потік T2 читає дані з буфера B2.

T1, T2 - призначені для користувача потоки  
B1, B2 - буфери  
K1, K2 - потоки ядра операційної системи  
М - загальна пам'ять

Обмін даними може бути організований тільки через ланцюжок взаємодіючих потоків, які обмінюються між собою даними через загальну, тільки для них, пам'ять.  
Таким же чином можлива організація каналу передачі даних по мережі. Тільки в цьому випадку загальна пам'ять М, може розглядатися як передає середовище, пристрій якої аналогічно пристрою каналу передачі даних.

Організація каналу передачі даних між процесами програмним чином.  
Для цього потрібно потоки ядра операційної системи і загальну пам'ять, використовувану для обміну даними, замінити файлом. І отримаємо найпростіший канал передачі даних між потоками, що виконуються в контекстах різних процесів. Таким чином, обмін даними між процесами через загальний файл являє собою організацію найпростішого каналу передачі даних між процесами. Іноді ОС може спростити доступ до поділюваного файлу, щоб прискорити обмін даними. Наприклад, ОС Windows для цих цілей можуть відображати або, іншими словами, проектувати файл на адресний простір процесу.

При обміні даними між паралельними процесами розрізняють два способи передачі даних:  
- потоком;  
- повідомленням.  
  
Якщо дані передаються безперервної послідовністю байтів, то така пересилання даних називається передача даних потоком. У цьому випадку загальна пам'ять М, доступна потокам ядра операційної системи, може й не бути, а пересилання даних виконується одним потоком ядра безпосередньо з буфера В1 в буфер В2.  
Якщо дані пересилаються групами байтів, то така група байтів називається повідомленням, а сама пересилання даних називається передачею даних повідомленнями.

## 4.2. Зв'язки між процесами

Перш ніж передавати дані між процесами, потрібно встановити між цими процесами зв'язок. Зв'язок між процесами може встановлюватися як на фізичному (або апаратному), так і на логічному (або програмному) рівнях. З точки зору напрямку передачі даних розрізняють наступні види зв'язків:  
- напівдуплексна зв'язок, т. Е. Дані по зв'язку з цим можуть передаватися тільки в одному напрямку;  
- дуплексний зв'язок, т. Е. Дані по зв'язку з цим можуть передаватися в обох напрямках.

## 4.3. Операції з каналами в .NET Framework

**Анонімні канали.**  
Анонімні канали забезпечують межпроцессное взаємодія на локальному комп'ютері. Анонімні канали використовують менше системних ресурсів, ніж іменовані канали, але їх можливості обмежені. Анонімні канали є односторонніми і не можуть використовуватися для взаємодії з мережі. Вони дозволяють використовувати тільки один екземпляр сервера. Анонімні канали зручно використовувати для організації взаємодії між потоками або між батьківськими і дочірніми процесами - в цьому випадку дескриптор каналу можна просто передати дочірньому процесу при його створенні.  
В .NET Framework анонімні канали реалізуються за допомогою класів AnonymousPipeServerStream і AnonymousPipeClientStream.

**Іменовані канали.**  
Іменовані канали забезпечують межпроцессное взаємодія між сервером і одним або декількома клієнтами. Іменовані канали можуть бути односторонніми або двосторонніми. Вони підтримують зв'язок за допомогою повідомлень і дозволяють кільком клієнтам підключатися до серверного процесу одночасно, використовуючи один і той же ім'я каналу. Іменовані канали також підтримують персональні настройки (уособлення), що дозволяють підключаються процесам використовувати власні дозволу на віддалених серверах.  
В .NET Framework іменовані канали реалізуються за допомогою класів NamedPipeServerStream і NamedPipeClientStream.

Використання анонімних каналів для локального взаємодії між процесами  
Анонімні канали забезпечують межпроцессное взаємодія на локальному комп'ютері. Вони надають менше функціональних можливостей, ніж іменовані канали, але і вимагають менших витрат ресурсів. Можна використовувати анонімні канали для спрощення взаємодії між процесами на локальному комп'ютері. Не можна використовувати анонімні канали для взаємодії з мережі.  
Для реалізації анонімних каналів використовуйте класи AnonymousPipeServerStream і AnonymousPipeClientStream.

Використання іменованих каналів для взаємодії між процесами по мережі  
Іменовані канали забезпечують межпроцессное взаємодія між сервером і одним або декількома клієнтами. Вони надають більше функціональних можливостей, ніж анонімні канали, які забезпечують межпроцессное взаємодія на локальному комп'ютері. Іменовані канали підтримують дуплексний зв'язок по мережі, кілька примірників сервера, взаємодія, засноване на повідомленнях і уособлення клієнта, що дозволяє тим, що підключається процесам використовувати власні набори дозволів на віддалених серверах.  
Для реалізації іменованих каналів використовуйте класи NamedPipeServerStream і NamedPipeClientStream.

# **Блокчейн**

## 5.1. Загальні відомості і основні поняття

Блокчейн можна визначити як розподілену між комп’ютерами базу данних, у якій данні (транзакції) розподілені по зв’язному списку блоків. При цьому гарантується неможливість підробки данних, маючи у владі лише незначну частину комп’ютерів. Але це можливо, якщо контролюється більшість компьютерів.

**Криптовалюта –** система на основі блокчейну, яка використовуються для обміну певною віртуальною валютою. Але потрібно зазначити, що блокчейн не обмежений одними лише криптовалютами.

**Транзакція** – данні, які вводяться у базу данних блокчейн. У базовій криптовалютній реализації блокчейна, у транзакції зберігаеться адресса відправника, отримувача, сумма переказу. При цьому транзакція підписується цифровим підписом (приватний ключ), який належить відправнику транзакції, після чого цей підпис додаеться до транзакції. Завдяки цьому вилучаеться можливість створення переказу від імені іншого користувача, не маючи при цьому його приватного ключа.

У цій курсовій роботі цифровий підпис не був реалізованний.

**Блок –** група транзакцій, окрім яких у базовій реалізації блокчейну повинен враховувати в себе інформацію о минулому блоці, який був останнім до створення цього. При цьому, підробка данних у минулому блоці повинна змінити і наступний блок. Саме для цього використовується результат hash-функції від минулого блоку. Завдяки цьому після створення блоку неможлива зміна данних – будь яка така спроба буде впливати на хеш функції блоків.

**Ланцюг блоків (Blockchain) –** односторонньо зв’язний список із блоків.

**Genesis блок –** перший блок у ланцюгу. Не має транзакцій, але має інформацію о минулому (неіснуючому) блоці – зазвичай це оговорене значення. Додається для того, щоб наступний блок у ланцюгу міг враховувати фактично існуючий блок.

## 5.2. Майнінг. Proof-of-Work та Proof-of-Stake.

**Майнінг** – діяльність задля підтримки розподіленої платформи і створення нових блоків.

Більшість блокчейн систем під час майнінгу використовують алгоритм - Proof-of-Work, за яким блок генерується разом с певним числом (доказ роботи, proof), hash яких повинен відповідати певній умові – наприклад закінчуватись на «000». Якщо хеш не відповідає умові, то це число-доказ інкрементується і хеш підраховується наново. При цьому для перевірки алгоритму достатньо одноразово перерахувати хеш с остаточним числом-доказом і перевірити його на умову.

Цей алгоритм використовується у криптовалюті Bitcoin і в класичному вигляді реалізован у курсовій роботі.

У 2012 році у криптовалюті PeerCoin був вперше використанний інших алгоритм – Proof-of-Stake. Ідея полягає у використанні «частки» (stake) в якості ресурсу, який визначає, який комп’ютер у мережі отримує право майнінгу наступного блоку.

У підході Proof-of-Stake комп’ютери також намагаються знаходити хеш блоків, який повинен відповідати певній умові, але складність в даному випадку розподіляється пропорційно і відповідно до валютного балансу даного комп’ютера. Таким чином, більше шансів згенерувати наступний блок має вузол з великим балансом.

Існують і інші алгоритми майнінгу, а також їх комбінації, але використовуються у більш рідкістних системах.

## 5.3. Винагорода і генерація нових монет.

У більшості кріптовалютних блокчейнах є можливість отримати винагороду за майнінг в формі валюти і комісійних зборів, які визначаються відправниками транзакцій, які додаються до блоку. Обчислення потрібні для забезпечення захисту від повторного витрачання одних і тих же одиниць валюти, а зв'язок майнінг з валютою стимулює людей витрачати свої обчислювальні потужності і підтримувати роботу мереж.

При цьому у більшості кріптовалют нові криптографічні монети генеруються лише у якості винагород за майнінг, і лише до певної їх кількості, після чого винагорода визначається тільки комісійними зборами.

## 5.4. Синхронізація і вирішення конфліктів.

Важливою частиною розподіленної системи є синхронізація.

У класичному блокчейні кожен комп’ютер у мережі може запросити у «сусідів» їх актуальний ланцюг блоків, після чого він аналізується і приймається замість старого у випадку, коли виконуються наступні правила:

1. новий ланцюг повинен бути не менший кількостью блоків за старий
2. всі блоки нового ланцюга повинні пройти перепровірку результатів їх hash функцій, порівнюючи їх з хешем блоку, який зберігається у наступному
3. всі хеши ціх блоків повинні відповідати загальному правилу, яке є зазначенним під час майнінгу (наприклад, закінчуватись на «000»)
4. всі данні у новому ланцюгу повинні виконувати інші правила специфічні від певної реалізації блокчейну

Серед специфічних правил можно виділити умову, коли залишається той ланцюг, для якого першим прийде продовження у вигляді нового блоку від комп’ютера-створювача цього ланцюгу.

У випадку, коли ланцюг блоків був перезаписан новим, що виконав всі умови, всі транзакції, які не існують у новому ланцюгу, перекидуються до списку вільних транзакцій, які будуть частиною наступного блоку.

# **Практична частина**

У курсовій роботі були використані різні підходи з розподіленям функционалу на меньші функціональні блоки, абстрагуванням коду у інтерфейсах, ін’єкцією залежностей де є практична користь для цього.

## Блокчейн

Створенні наступні моделі:

[ProtoContract]

public class Transaction

{

[ProtoMember(1)]

public string Sender { get; }

[ProtoMember(2)]

public string Recipient { get; }

[ProtoMember(3)]

public double Amount { get; }

[ProtoMember(4)]

public DateTime Date { get; }

/// <summary>

/// Приватный конструктор для сериализации

/// </summary>

private Transaction()

{

}

/// <summary>

/// Конструктор транзацкии

/// </summary>

/// <param name="sender">Отправитель</param>

/// <param name="recipient">Получатель</param>

/// <param name="amout">Сумма</param>

/// <param name="date">Дата создания</param>

public Transaction(string sender, string recipient, double amout, DateTime date)

{

Sender = sender;

Recipient = recipient;

Amount = amout;

Date = date;

}

public override string ToString()

{

return $"Transaction: {Amount} from {

(string.IsNullOrEmpty(Sender) ? "mining" : Sender)

} to {Recipient}";

}

}

[ProtoContract]

public class Block

{

[ProtoMember(1)]

public int Index { get; }

[ProtoMember(2)]

public DateTime Date { get; }

[ProtoMember(3)]

private List<Transaction> transaction;

[ProtoIgnore]

public IReadOnlyList<Transaction> Transactions =>

(transaction ?? new List<Transaction>()).AsReadOnly();

[ProtoMember(4)]

public long Proof { get; }

[ProtoMember(5)]

public string PreviousHash { get; }

/// <summary>

/// Приватные конструктор для сериализации

/// </summary>

private Block()

{

}

/// <summary>

/// Конструктор блока

/// </summary>

/// <param name="index">Индекс блока</param>

/// <param name="date">Дата создания</param>

/// <param name="transactions">Транзакции</param>

/// <param name="proof">Доказательство доберия блоку</param>

/// <param name="previousHash">Хэш предыдущего блока</param>

public Block(int index, DateTime date, IEnumerable<Transaction> transactions,

long proof, string previousHash)

{

Index = index;

Date = date;

transaction = transactions?.ToList() ?? new List<Transaction>();

Proof = proof;

PreviousHash = previousHash;

}

public override string ToString()

{

var str = $"Block #{Index} {Date}\r\n";

return str = $"Proof = {Proof}; Previous hash = {PreviousHash}";

}

}

Допоміжний статичний класс для криптографічних функцій

public static class Crypto

{

/// <summary>

/// Хеширует строку

/// </summary>

/// <param name="input">Входящая строка</param>

/// <returns>Захешированная строка</returns>

public static string HashString(string input)

{

var blockBytes = Encoding.UTF8.GetBytes(input);

var sha = SHA256.Create();

return Encoding.UTF8.GetString(sha.ComputeHash(blockBytes));

}

/// <summary>

/// Хеширует блок

/// </summary>

/// <param name="block">Входящий блок</param>

/// <returns>Блок захешированный в строку</returns>

public static string HashBlockInBase64(Block block)

{

using (var stream = new MemoryStream())

{

Serializer.Serialize(stream, block);

var blockBytes = stream.ToArray();

var sha = SHA256.Create();

return Convert.ToBase64String(sha.ComputeHash(blockBytes));

}

}

}

Основний класс блокчейну:

[ProtoContract]

public class Blockchain

{

[ProtoMember(1)]

private List<Block> chain;

[ProtoMember(2)]

private List<Transaction> currentTransactions;

/// <summary>

/// Цепочка блоков, блокчейн

/// </summary>

public IReadOnlyList<Block> Chain => chain ?? new List<Block>();

/// <summary>

/// Текущие транзакции

/// </summary>

public IReadOnlyList<Transaction> CurrentTransactions

=> currentTransactions ?? new List<Transaction>();

public event EventHandler<BlockAddedEventArgs> BlockAdded;

public event EventHandler BlockchainReplaced;

private Blockchain()

{

chain = new List<Block>();

currentTransactions = new List<Transaction>();

}

/// <summary>

/// Создает блокчейн с одним генезис блоком

/// </summary>

/// <returns>Созданный блокчейн</returns>

public static Blockchain CreateNew()

{

var blockchain = new Blockchain();

blockchain.NewBlock(100, "1");

return blockchain;

}

/// <summary>

/// Читает и создает блокчейн из файла

/// </summary>

/// <param name="fileName">Имя файла</param>

/// <returns>Прочитанный блокчейн</returns>

public static Blockchain FromFile(string fileName)

{

if (string.IsNullOrWhiteSpace(fileName))

throw new ArgumentException("File name cannot be null or empty",

nameof(fileName));

using (var file = new FileStream(fileName, FileMode.OpenOrCreate))

return Serializer.Deserialize<Blockchain>(file);

}

/// <summary>

/// Сохраняет текущий блокчейн в файл

/// </summary>

/// <param name="fileName">Имя файла</param>

public void SaveFile(string fileName)

{

if (string.IsNullOrWhiteSpace(fileName))

throw new ArgumentException("File name cannot be null or empty",

nameof(fileName));

using (var file = new FileStream(fileName, FileMode.OpenOrCreate))

Serializer.Serialize(file, this);

}

/// <summary>

/// Проверка блокчейна

/// </summary>

/// <param name="chain">Блокчейн</param>

/// <returns>True если прошел проверку, иначе False</returns>

public bool IsValidChain(IEnumerable<Block> chain)

{

var prevBlock = chain.FirstOrDefault();

if (prevBlock == null)

throw new ArgumentException("Blocks chain cannot be empty",

nameof(chain));

var amountDictionary = new Dictionary<string, double>();

foreach (var block in chain.Skip(1))

{

if (block.PreviousHash != Crypto.HashBlockInBase64(prevBlock)

|| !IsValidProof(prevBlock.Proof, block.Proof))

return false;

foreach (var transaction in block.Transactions)

{

if (!string.IsNullOrEmpty(transaction.Sender))

{

if (amountDictionary.ContainsKey(transaction.Sender))

{

amountDictionary[transaction.Sender] -= transaction.Amount;

if (amountDictionary[transaction.Sender] < 0)

return false;

}

else if (transaction.Amount > 0)

{

return false;

}

}

if (amountDictionary.ContainsKey(transaction.Recipient))

{

amountDictionary[transaction.Recipient] += transaction.Amount;

}

else

{

amountDictionary.Add(transaction.Recipient, transaction.Amount);

}

}

prevBlock = block;

}

return true;

}

/// <summary>

/// Добавляет новую транзакцию к списку транзакций

/// </summary>

/// <param name="sender">Адресс отправителя</param>

/// <param name="recipient">Адресс получателя</param>

/// <param name="amount">Сумма</param>

/// <returns>Индекс блока, хранящего добаленную транзакцию</returns>

public int NewTransaction(string sender, string recipient, double amount)

{

if (GetAccountAmount(sender) < amount)

throw new InvalidOperationException("Sender amount is not enough");

var date = DateTime.Now;

var transaction = new Transaction(sender, recipient, amount, date);

currentTransactions.Add(transaction);

return LastBlock().Index + 1;

}

/// <returns>Последний блок в цепочке</returns>

public Block LastBlock()

{

return chain.Last();

}

/// <summary>

/// Запускает процесс майнинга нового блока

/// </summary>

/// <returns>Новый блок</returns>

public Block Mine(string minerAccount)

{

var lastBlock = LastBlock();

var lastProof = lastBlock.Proof;

var lastHash = Crypto.HashBlockInBase64(lastBlock);

var proof = ProofOfWork(lastProof);

// Оплата за майнинг

var transaction = new Transaction(null, minerAccount, 1, DateTime.Now);

currentTransactions.Add(transaction);

return NewBlock(proof, lastHash);

}

/// <summary>

/// Пытается заменить блокчейн

/// </summary>

/// <param name="recievedChain">Новый блокчейн</param>

/// <returns>True, если новый блок валидный и замена успешна</returns>

public bool TrySetChainIfValid(ICollection<Block> recievedChain)

{

if (recievedChain.Count >= chain.Count

&& IsValidChain(recievedChain))

{

chain = recievedChain.ToList();

BlockchainReplaced?.Invoke(this, EventArgs.Empty);

return true;

}

return false;

}

/// <summary>

/// Возвращает сумму на счету аккаунта

/// </summary>

/// <param name="account">Аккаунт</param>

/// <returns>Сумма на счету</returns>

public double GetAccountAmount(string account)

{

if (string.IsNullOrEmpty(account))

throw new ArgumentException("Account cannot be null or empty",

nameof(account));

var amount = 0d;

foreach (var block in chain)

{

foreach (var transaction in block.Transactions)

{

if (transaction.Recipient == account)

amount += transaction.Amount;

if (transaction.Sender == account)

amount -= transaction.Amount;

}

}

if (amount < 0)

throw new InvalidDataException("Chain is invalid");

return amount;

}

private Block NewBlock(long proof, string previousHash = null)

{

var block = new Block(

chain.Count,

DateTime.Now,

currentTransactions,

proof,

previousHash);

chain.Add(block);

currentTransactions.Clear();

BlockAdded?.Invoke(this, new BlockAddedEventArgs(block, chain));

return block;

}

private long ProofOfWork(long lastProof)

{

var proof = 0L;

while (!IsValidProof(lastProof, proof))

proof++;

return proof;

}

private static bool IsValidProof(long lastProof, long proof)

{

return Crypto.HashString($"{lastProof}{proof}").EndsWith("00");

}

}

## Коммунікація

Для коммунікації між нодами (комп’ютерами або, у виподку курсової, - процессами) були використані наступні інтерфейси:

public interface ICommunication

{

/// <summary>

/// Id сервера

/// </summary>

string ServerId { get; }

/// <summary>

/// Запускает коммуникацию

/// </summary>

void Start();

/// <summary>

/// Остагавливает коммуникацию

/// </summary>

void Stop();

}

public interface ICommunicationClient<T> : ICommunication

{

/// <summary>

/// Id сервера для обратной связи

/// </summary>

string ResponceServerId { get; set; }

/// <summary>

/// Асинхронный метод для отправки сообщения

/// </summary>

/// <param name="message">Сообщение</param>

/// <returns>True если отправка успещна, иначе False</returns>

Task<bool> SendMessageAsync(T message);

}

public interface ICommunicationServer<T> : ICommunication

{

/// <summary>

/// Событие, которое вызывается при получении сообщения

/// </summary>

event EventHandler<MessageReceivedEventArgs<T>> MessageReceivedEvent;

/// <summary>

/// Событие, которое вызывается при подключении клиента

/// </summary>

event EventHandler<ClientConnectedEventArgs> ClientConnectedEvent;

/// <summary>

/// Событие, которое вызывается при отключении клиента

/// </summary>

event EventHandler<ClientDisconnectedEventArgs> ClientDisconnectedEvent;

}

Був створенний інтерфейс-фабрика, який відіграє роль у створювачі обїектів ICommunicatorClient

public interface ICommunicationClientFactory<T>

{

/// <summary>

/// Создает новый клиент, подключаемый к серверу по его Id

/// </summary>

/// <param name="serverId">Id сервера, к которому подключается клиент</param>

/// <returns>Экземпляр клиента</returns>

ICommunicationClient<T> CreateNew(string serverId);

}

За однією із умов курсовою роботи, программа повинна використовувати для коммунікації між кліентами NamedPipeClient(Server)Stream. Тому у окремому проекті були створенні наступні реалізації інтерфейсів-коммунікаторів:

public class PipeClient<T> : ICommunicationClient<T>

{

private NamedPipeClientStream \_pipeClient;

public PipeClient(string serverId)

{

ServerId = serverId;

}

public string ServerId { get; }

public string ResponceServerId { get; set; }

public void Start()

{

if (\_pipeClient == null)

{

\_pipeClient = new NamedPipeClientStream(

".",

ServerId,

PipeDirection.Out,

PipeOptions.Asynchronous);

}

\_pipeClient.Connect();

// Сразу отправляем Id сервера для обратной связи

var buffer = Encoding.UTF8.GetBytes(string.IsNullOrEmpty(ResponceServerId)

? "\0"

: ResponceServerId);

\_pipeClient.Write(buffer, 0, buffer.Length);

}

public void Stop()

{

if (\_pipeClient == null)

return;

try

{

\_pipeClient.WaitForPipeDrain();

}

finally

{

\_pipeClient.Close();

\_pipeClient.Dispose();

\_pipeClient = null;

}

}

public async Task<bool> SendMessageAsync(T message)

{

using (var stream = new MemoryStream())

{

Serializer.Serialize(stream, message);

await \_pipeClient.WriteAsync(stream.ToArray(), 0, (int)stream.Length);

await \_pipeClient.FlushAsync();

}

return true;

}

}

public class PipeClientFactory<T> : ICommunicationClientFactory<T>

{

public ICommunicationClient<T> CreateNew(string serverId)

{

return new PipeClient<T>(serverId);

}

}

internal class InternalPipeServer<T> : ICommunicationServer<T>

{

private readonly NamedPipeServerStream \_pipeServer;

private bool \_isStopping;

class Package

{

public byte[] Buffer = new byte[PipeServer<T>.BufferSize];

public List<byte> Result = new List<byte>();

}

public InternalPipeServer(string pipeName, int maxNumberOfServerInstances)

{

\_pipeServer = new NamedPipeServerStream(

pipeName,

PipeDirection.In,

maxNumberOfServerInstances,

PipeTransmissionMode.Message,

PipeOptions.Asynchronous);

ServerId = pipeName;

}

public event EventHandler<ClientConnectedEventArgs> ClientConnectedEvent;

public event EventHandler<ClientDisconnectedEventArgs> ClientDisconnectedEvent;

public event EventHandler<MessageReceivedEventArgs<T>> MessageReceivedEvent;

public string ServerId { get; }

private string responceClientId;

public void Start()

{

\_pipeServer.WaitForConnectionAsync().ContinueWith(t =>

{

if (!\_isStopping)

{

// сразу читаем id сервера для обратной связи

// он пригодится для передачи в обработчики событий

var buffer = new byte[2048];

var length = \_pipeServer.Read(buffer, 0, buffer.Length);

responceClientId = Encoding.UTF8.GetString(buffer, 0, length);

if (responceClientId == "\0")

responceClientId = null;

ClientConnectedEvent?.Invoke(

this,

new ClientConnectedEventArgs { ClientId = responceClientId });

return ReadAsync(new Package());

}

return Task.CompletedTask;

});

}

public void Stop()

{

\_isStopping = true;

try

{

if (\_pipeServer.IsConnected)

{

\_pipeServer.Disconnect();

}

}

finally

{

\_pipeServer.Close();

\_pipeServer.Dispose();

}

}

private async Task ReadAsync(Package package)

{

// побуферно читаем сообщения, что позволяет не ограничиваться его размером

var readBytes = await \_pipeServer.ReadAsync(

package.Buffer, 0, package.Buffer.Length);

if (readBytes > 0)

{

byte[] result;

if (\_pipeServer.IsMessageComplete)

{

result = new byte[readBytes];

Array.Copy(package.Buffer, 0, result, 0, readBytes);

}

else

{

result = package.Buffer;

}

package.Result.AddRange(result);

// если не дочитали...

if (!\_pipeServer.IsMessageComplete)

{

// ...то читаем следующий буфер

await ReadAsync(package);

}

else

{

using (var stream = new MemoryStream(package.Result.ToArray()))

{

var message = Serializer.Deserialize<T>(stream);

MessageReceivedEvent?.Invoke(this,

new MessageReceivedEventArgs<T>

{

ClientId = responceClientId,

Message = message

});

}

await ReadAsync(new Package());

}

}

// Если прочитано 0 байт, то клиент вероятно отключился

else

{

if (!\_isStopping)

{

Stop();

ClientDisconnectedEvent?.Invoke(

this,

new ClientDisconnectedEventArgs { ClientId = responceClientId });

}

}

}

}

public class PipeServer<T> : ICommunicationServer<T>

{

public const int BufferSize = 2048;

public const int MaxNumberOfServerInstances = 100;

private readonly SynchronizationContext \_synchronizationContext;

private readonly ConcurrentDictionary<string, ICommunicationServer<T>> \_servers;

public PipeServer(string id)

{

ServerId = id;

\_synchronizationContext = AsyncOperationManager.SynchronizationContext;

\_servers = new ConcurrentDictionary<string, ICommunicationServer<T>>();

}

public PipeServer() : this(Guid.NewGuid().ToString())

{

}

public event EventHandler<MessageReceivedEventArgs<T>> MessageReceivedEvent;

public event EventHandler<ClientConnectedEventArgs> ClientConnectedEvent;

public event EventHandler<ClientDisconnectedEventArgs> ClientDisconnectedEvent;

public string ServerId { get; }

public void Start()

{

StartNamedPipeServer();

}

public void Stop()

{

foreach (var server in \_servers.Values)

{

try

{

UnregisterFromServerEvents(server);

server.Stop();

}

catch (Exception)

{

}

}

\_servers.Clear();

}

private void StartNamedPipeServer()

{

// Есть смысл создавать свой InternalPipeServer для каждого клиента,

// но при этом основываясь на одном его имени

// Это позволяет независимо читать данные с множества клиентов

var server = new InternalPipeServer<T>(ServerId, MaxNumberOfServerInstances);

server.ClientConnectedEvent += ClientConnectedHandler;

server.ClientDisconnectedEvent += ClientDisconnectedHandler;

server.MessageReceivedEvent += MessageReceivedHandler;

\_servers[server.ServerId] = server;

server.Start();

}

private void StopNamedPipeServer(string id)

{

if (\_servers.TryRemove(id, out ICommunicationServer<T> removed))

{

UnregisterFromServerEvents(removed);

removed.Stop();

}

}

private void UnregisterFromServerEvents(ICommunicationServer<T> server)

{

server.ClientConnectedEvent -= ClientConnectedHandler;

server.ClientDisconnectedEvent -= ClientDisconnectedHandler;

server.MessageReceivedEvent -= MessageReceivedHandler;

}

private void ClientConnectedHandler(object sender,

ClientConnectedEventArgs eventArgs)

{

\_synchronizationContext.Post(

e => ClientConnectedEvent?.Invoke(this, (ClientConnectedEventArgs)e),

eventArgs);

StartNamedPipeServer();

}

private void ClientDisconnectedHandler(object sender,

ClientDisconnectedEventArgs eventArgs)

{

\_synchronizationContext.Post(

e => ClientDisconnectedEvent?.Invoke(this,

(ClientDisconnectedEventArgs)e),

eventArgs);

StopNamedPipeServer(eventArgs.ClientId);

}

private void MessageReceivedHandler(object sender,

MessageReceivedEventArgs<T> eventArgs)

{

\_synchronizationContext.Post(

e => MessageReceivedEvent?.Invoke(this, (MessageReceivedEventArgs<T>)e),

eventArgs);

}

}

Потрібно зазначити, що були використані окремі PipeServer і InternalPipeServer потрібні.

InternalPipeServer може працювати лише з одним кліентом, при цьому блокуя себе під час очікування повідомлення.

PipeServer у свою чергу є звичайною «прослойкою» між інтерфейсом і декількома InternalPipeServer.

Для роботи з інтерфейсами був створенний класс Communicator, який приймає аргументами конструктора (ін’єекція залежностей) інтерфейс сервера коммунікації та інтерфейс фабрики кліентів коммунікації. При цьому цей класс має посилання на класс Blockchain і має функції для міжнодовою взаємодії блокчейнів:

public class Communicator

{

private readonly ICommunicationServer<List<Block>> server;

private readonly ICommunicationClientFactory<List<Block>> clientFactory;

private readonly List<ICommunicationClient<List<Block>>> nodes;

public Blockchain Blockchain { get; set; }

/// <summary>

/// Конструктор коммуникатора

/// </summary>

/// <param name="server">Сервер</param>

/// <param name="clientFactory">Фаблика клиентов</param>

public Communicator(

ICommunicationServer<List<Block>> server,

ICommunicationClientFactory<List<Block>> clientFactory)

{

this.server = server;

this.clientFactory = clientFactory;

nodes = new List<ICommunicationClient<List<Block>>>();

server.MessageReceivedEvent += Server\_MessageReceivedEvent;

server.ClientConnectedEvent += Server\_ClientConnectedEvent;

server.ClientDisconnectedEvent += Server\_ClientDisconnectedEvent;

server.Start();

}

public Task SyncAsync(bool onlyGet = false)

{

if (Blockchain == null)

throw new InvalidOperationException("Blockchain must be setted");

var sendedList = onlyGet

? new List<Block>()

: Blockchain.Chain.ToList();

return Task

.WhenAll(nodes

.Select(node => node.SendMessageAsync(sendedList)));

}

public void ConnectTo(IEnumerable<string> serversId)

{

foreach (var serverId in serversId)

{

var nodeClient = clientFactory.CreateNew(serverId);

nodeClient.ResponceServerId = server.ServerId;

nodeClient.Start();

nodes.Add(nodeClient);

}

}

public void Close()

{

server.Stop();

foreach (var nodeClient in nodes)

nodeClient.Stop();

}

private void Server\_ClientConnectedEvent(object sender,

ClientConnectedEventArgs e)

{

if (nodes.All(c => c.ServerId != e.ClientId))

{

var nodeClient = clientFactory.CreateNew(e.ClientId);

nodeClient.ResponceServerId = server.ServerId;

nodeClient.Start();

nodes.Add(nodeClient);

}

}

private void Server\_ClientDisconnectedEvent(object sender,

ClientDisconnectedEventArgs e)

{

var node = nodes.FirstOrDefault(n => n.ServerId == e.ClientId);

if (node != null)

{

node.Stop();

nodes.Remove(node);

}

}

private async void Server\_MessageReceivedEvent(object sender,

MessageReceivedEventArgs<List<Block>> e)

{

if (Blockchain == null)

throw new InvalidOperationException("Blockchain must be setted");

var replaced = Blockchain.TrySetChainIfValid(e.Message);

// Если не заменено, то входящяя цепочка или невалидна,

// или меньше существующей

// Есть смысл отправить отправителю свою цепочку для замены

if (!replaced)

{

var nodeClient = nodes.FirstOrDefault(c => c.ServerId == e.ClientId);

await nodeClient.SendMessageAsync(Blockchain.Chain.ToList());

}

}

}

Був створенний допоміжний статичний класс для взаємодії процессів і каналів

public static class ProcessPipeHelper

{

/// <summary>

/// Возвращает Id канала для текущего процесса

/// </summary>

/// <returns>Id канала</returns>

public static string GetCurrentPipeId()

{

var currentProcess = Process.GetCurrentProcess();

return GetPipeIdFromProcessId(currentProcess.Id);

}

/// <summary>

/// Возвращает Id каналов с соседних процесов

/// </summary>

/// <returns>Id каналов соседних процессов</returns>

public static IEnumerable<string> GetNeighborPipesIds()

{

// важно учесть .vshost для процесса с дебагером

var currentProcess = Process.GetCurrentProcess();

var processName = currentProcess.ProcessName.Replace(".vshost", "");

var processNameVsHosted = processName + ".vshost";

return Process

.GetProcessesByName(processName)

.Concat(Process.GetProcessesByName(processNameVsHosted))

.Where(p => p.Id != currentProcess.Id)

.Select(p => GetPipeIdFromProcessId(p.Id));

}

private static string GetPipeIdFromProcessId(int processId)

{

return $"Pipe-{processId}";

}

}

## Тестування

Классам тестування не було приділення багато уваги, але деяку частину було вирішенно реалізувати.

Тестування каналів:

[TestClass]

public class PipeTest

{

[TestMethod, Timeout(5000)]

public void Pipe\_Instantiate\_PipeServer\_IdTest()

{

var server = new PipeServer<string>();

Assert.IsFalse(string.IsNullOrWhiteSpace(server.ServerId), "Server id is null or empty");

}

[TestMethod, Timeout(5000)]

public async Task Pipe\_Client\_ConnectTest()

{

var tcs = new TaskCompletionSource<bool>();

var isConnected = false;

var server = new PipeServer<string>();

var client = new PipeClient<string>(server.ServerId);

server.ClientConnectedEvent += (sender, args) =>

{

isConnected = true;

tcs.SetResult(true);

};

server.Start();

client.Start();

await tcs.Task;

Assert.IsTrue(isConnected, "Client is not connected");

}

[TestMethod, Timeout(5000)]

public async Task Pipe\_Client\_Connect\_ResponceServerIdTest()

{

const string ResponceServerId = "responceServerId";

var tcs = new TaskCompletionSource<bool>();

var id = string.Empty;

var server = new PipeServer<string>();

var client = new PipeClient<string>(server.ServerId)

{

ResponceServerId = ResponceServerId

};

server.ClientConnectedEvent += (sender, args) =>

{

id = args.ClientId;

tcs.SetResult(true);

};

server.Start();

client.Start();

await tcs.Task;

Assert.AreEqual(ResponceServerId, id,

"Responce server id does not match the correct");

}

[TestMethod, Timeout(5000)]

public async Task Pipe\_Client\_Connect\_ResponceServerId\_EmptyTest()

{

const string ResponceServerId = null;

var tcs = new TaskCompletionSource<bool>();

var id = string.Empty;

var server = new PipeServer<string>();

var client = new PipeClient<string>(server.ServerId);

server.ClientConnectedEvent += (sender, args) =>

{

id = args.ClientId;

tcs.SetResult(true);

};

server.Start();

client.Start();

await tcs.Task;

Assert.AreEqual(ResponceServerId, id,

"Responce server id does not match the correct");

}

[TestMethod, Timeout(5000)]

public async Task Pipe\_Client\_DisconnectTest()

{

var tcs = new TaskCompletionSource<bool>();

var isDisconnected = false;

var server = new PipeServer<string>();

var client = new PipeClient<string>(server.ServerId);

server.ClientDisconnectedEvent += (sender, args) =>

{

isDisconnected = true;

tcs.SetResult(true);

};

server.Start();

client.Start();

Assert.IsFalse(isDisconnected, "Client is disconected");

client.Stop();

client = null;

await tcs.Task;

Assert.IsTrue(isDisconnected, "Client is still connected");

}

[TestMethod, Timeout(5000)]

public async Task Pipe\_Client\_SendMessageTest()

{

const string Message = "Client's message";

var tcs = new TaskCompletionSource<string>();

var server = new PipeServer<string>();

var client = new PipeClient<string>(server.ServerId);

server.MessageReceivedEvent += (sender, args) =>

{

tcs.SetResult(args.Message);

};

server.Start();

client.Start();

var success = await client.SendMessageAsync(Message);

Assert.IsTrue(success, "Send message failed");

var message = await tcs.Task;

Assert.AreEqual(Message, message, "Message are not equal");

}

[TestMethod, Timeout(5000)]

public async Task Pipe\_Client\_SendMessage\_LongMessageTest()

{

const int BufferSize = 2048;

var tcs = new TaskCompletionSource<string>();

var server = new PipeServer<string>();

var client = new PipeClient<string>(server.ServerId);

server.MessageReceivedEvent += (sender, args) =>

{

tcs.SetResult(args.Message);

};

server.Start();

client.Start();

var longString = $"[{new string('\*', 10 \* 2048)}]";

Assert.IsTrue(Encoding.UTF8.GetByteCount(longString) > BufferSize,

"Test string is smaller than needed buffer length");

var success = await client.SendMessageAsync(longString);

Assert.IsTrue(success, "Send message failed");

var message = await tcs.Task;

Assert.AreEqual(longString, message, "Messages are not equal");

}

[TestMethod, Timeout(5000)]

public async Task Pipe\_Client\_SendMessage\_MultiMessageTest()

{

const string FirstTestMessage = "Hi from first";

const string SecondTestMessage = "Hi from second";

var autoEvent = new AutoResetEvent(false);

var message = string.Empty;

var server = new PipeServer<string>();

var client = new PipeClient<string>(server.ServerId);

server.MessageReceivedEvent += (sender, args) =>

{

message = args.Message;

autoEvent.Set();

};

server.Start();

client.Start();

var success = await client.SendMessageAsync(FirstTestMessage);

Assert.IsTrue(success, "Send #1 message failed");

autoEvent.WaitOne();

Assert.AreEqual(FirstTestMessage, message, "#1 messages are not equal");

success = await client.SendMessageAsync(SecondTestMessage);

Assert.IsTrue(success, "Send #2 message failed");

autoEvent.WaitOne();

Assert.AreEqual(SecondTestMessage, message, "#2 messages are not equal");

}

[TestMethod, Timeout(10000)]

public async Task Pipe\_Client\_SendMessage\_MessageSpamTest()

{

const int MessageCount = PipeServer<string>.MaxNumberOfServerInstances / 2;

var sendMessages = Enumerable.Range(0, MessageCount).Select(

i => i.ToString()).ToList();

var recievedMessages = new List<string>();

var autoEvent = new AutoResetEvent(false);

var message = string.Empty;

var server = new PipeServer<string>();

var client = new PipeClient<string>(server.ServerId);

server.MessageReceivedEvent += (sender, args) =>

{

recievedMessages.Add(args.Message);

if (recievedMessages.Count >= MessageCount)

autoEvent.Set();

};

server.Start();

client.Start();

await Task.WhenAll(sendMessages.Select(async i =>

{

var success = await client.SendMessageAsync(i);

Assert.IsTrue(success, $"Send #{i} message failed");

}));

autoEvent.WaitOne();

Assert.AreEqual(MessageCount, recievedMessages.Count,

"Not all messages recieved");

CollectionAssert.AreEquivalent(sendMessages, recievedMessages,

"Send and recieved messages are not equal");

}

[TestMethod, Timeout(10000)]

public async Task Pipe\_Client\_SendMessage\_MultiClientSpamTest()

{

const int MessageCount = PipeServer<string>.MaxNumberOfServerInstances / 2;

var sendMessages = Enumerable.Range(0, MessageCount).Select(

i => i.ToString()).ToList();

var recievedMessages = new List<string>();

var autoEvent = new AutoResetEvent(false);

var messages = new List<string>();

var server = new PipeServer<string>();

server.MessageReceivedEvent += (sender, args) =>

{

recievedMessages.Add(args.Message);

if (recievedMessages.Count >= MessageCount)

autoEvent.Set();

};

server.Start();

await Task.WhenAll(sendMessages.Select(async i =>

{

var client = new PipeClient<string>(server.ServerId);

client.Start();

var success = await client.SendMessageAsync(i);

Assert.IsTrue(success, $"Send #{i} message failed");

client.Stop();

}));

autoEvent.WaitOne();

Assert.AreEqual(MessageCount, recievedMessages.Count,

"Not all messages recieved");

CollectionAssert.AreEquivalent(sendMessages, recievedMessages,

"Send and recieved messages are not equal");

}

[TestMethod, Timeout(10000)]

public async Task Pipe\_Client\_SendMessage\_RestartedTest()

{

const string FirstMessage = "First";

const string SecondMessage = "Second";

var message = string.Empty;

var autoEvent = new AutoResetEvent(false);

var server = new PipeServer<string>();

server.MessageReceivedEvent += (sender, args) =>

{

message = args.Message;

autoEvent.Set();

};

server.Start();

var client = new PipeClient<string>(server.ServerId);

client.Start();

await client.SendMessageAsync(FirstMessage);

autoEvent.WaitOne();

Assert.AreEqual(FirstMessage, message, "Message are not equal");

client.Stop();

client.Start();

await client.SendMessageAsync(SecondMessage);

autoEvent.WaitOne();

Assert.AreEqual(SecondMessage, message, "Message are not equal");

}

}

Тестування серіалізації та десеріалізації

[TestClass]

public class BlockTest

{

[TestMethod]

public void Block\_SerializeDeserializeTest()

{

var transactions = new[] {

new Transaction("UserA", "UserB", 999, DateTime.Now) };

var block = new Block(

0, DateTime.Now, transactions, 123, "h34j34h634g5i4h536oi==");

using (var stream = new MemoryStream())

{

Serializer.Serialize(stream, block);

stream.Flush();

stream.Position = 0;

var newBlock = Serializer.Deserialize<Block>(stream);

Assert.IsTrue(newBlock.Index == block.Index,

"Blocks indexes is not equal");

Assert.IsTrue(newBlock.Date == block.Date,

"Blocks date is not equal");

Assert.IsTrue(newBlock.Proof == block.Proof,

"Blocks proof is not equal");

Assert.IsTrue(newBlock.PreviousHash == block.PreviousHash,

"Blocks previous hash is not equal");

for (int index = 0; index < transactions.Length; index++)

{

var transaction = block.Transactions[index];

var newTransaction = newBlock.Transactions[index];

Assert.IsTrue(transaction.Sender == newTransaction.Sender,

$"#{index + 1} transactions sender is not equal");

Assert.IsTrue(transaction.Recipient == newTransaction.Recipient,

$"#{index + 1} transactions recipient is not equal");

Assert.IsTrue(transaction.Amount == newTransaction.Amount,

$"#{index + 1} transactions amount is not equal");

}

}

}

}

[TestClass]

public class TransactionTest

{

[TestMethod]

public void Transaction\_SerializeDeserializeTest()

{

var transaction = new Transaction("UserA", "UserB", 999, DateTime.Now);

using (var stream = new MemoryStream())

{

Serializer.Serialize(stream, transaction);

stream.Flush();

stream.Position = 0;

var newTransaction = Serializer.Deserialize<Transaction>(stream);

Assert.IsTrue(transaction.Sender == newTransaction.Sender,

"Transactions sender is not equal");

Assert.IsTrue(transaction.Recipient == newTransaction.Recipient,

"Transactions recipient is not equal");

Assert.IsTrue(transaction.Amount == newTransaction.Amount,

"Transactions amount is not equal");

Assert.IsTrue(transaction.Date == newTransaction.Date,

"Transactions date is not equal");

}

}

}

Тестування майнінгу нового блоку і перевірки всього ланцюга

[TestClass]

public class BlockchainTest

{

[TestMethod]

public void Blockchain\_MineTest()

{

var blockchain = Blockchain.CreateNew();

blockchain.NewTransaction("Alice", "Bob", 0);

var block = blockchain.Mine("Alice");

var isValid = blockchain.IsValidChain(blockchain.Chain);

Assert.IsTrue(isValid, "Blockchain is invalid");

}

}

## Інтерфейс користувача

У якості інтерфейсу були використані два проекти – консольний та графічний інтерфейс на Windows Forms

Консольний:

static class Program

{

private static Communicator communicator;

private static Blockchain blockchain => communicator.Blockchain;

private static string account;

private delegate bool ConsoleEventDelegate(int eventType);

[DllImport("kernel32.dll", SetLastError = true)]

private static extern bool SetConsoleCtrlHandler(

ConsoleEventDelegate callback, bool add);

private static void Main(string[] args)

{

// Обробка події закриття консолі

SetConsoleCtrlHandler(arg =>

{

if (arg == 2)

communicator.Close();

return false;

}, true);

var currentPipeId = ProcessPipeHelper.GetCurrentPipeId();

Console.Title = currentPipeId;

communicator = new Communicator(

new PipeServer<List<Block>>(currentPipeId),

new PipeClientFactory<List<Block>>())

{

Blockchain = Blockchain.CreateNew()

};

communicator.ConnectTo(ProcessPipeHelper.GetNeighborPipesIds());

AskAndSetAccount();

var active = true;

while (active)

{

var parts = Console.ReadLine().Split(' ');

if (parts.Length == 0)

continue;

switch (parts[0])

{

case "exit":

case "e":

active = false;

break;

case "blocks":

case "b":

PrintBlockchain();

break;

case "transactions":

case "t":

PrintCurrentTransactions();

break;

case "mine":

case "m":

\_ = MineAndPrintNewBlock();

break;

case "add":

case "a":

ReadAndAddNewTransaction(parts.Skip(1).ToArray());

break;

case "sync":

case "s":

SyncBlockchain();

break;

case "help":

case "h":

PrintHelp();

break;

case "amount":

case "am":

PrintAccountAmount(parts.Skip(1).FirstOrDefault());

break;

case "save":

case "sv":

SaveBlockChain(parts.Skip(1).FirstOrDefault());

break;

case "load":

case "ld":

LoadBlockChain(parts.Skip(1).FirstOrDefault());

break;

case "switch":

case "sw":

AskAndSetAccount(parts.Skip(1).FirstOrDefault());

break;

}

}

communicator.Close();

}

private static void AskAndSetAccount(string newAccount = null)

{

if (string.IsNullOrWhiteSpace(newAccount))

{

Console.Write("Input account: ");

newAccount = Console.ReadLine();

}

account = newAccount;

}

private static void PrintAccountAmount(string account)

{

while (string.IsNullOrWhiteSpace(account))

{

Console.Write("Account: ");

account = Console.ReadLine();

}

var amount = blockchain.GetAccountAmount(account);

Console.WriteLine($"{account} has: {amount}");

}

private static void PrintHelp()

{

Console.WriteLine("e, exit - close the node");

Console.WriteLine("a, add [recipient] [amount] - add new transaction");

Console.WriteLine("t, transactions - print current transactions");

Console.WriteLine("b, blocks - print blockschain");

Console.WriteLine("m, mine - mine new block");

Console.WriteLine("s, sync - sync blockchain with neighbor");

Console.WriteLine("sv, save [filename] - save blockchain");

Console.WriteLine("ld, load [filename] - load blockchain");

Console.WriteLine("am, amount [account] - print account amount");

Console.WriteLine("sw, switch [account] - change account");

}

private static void ReadAndAddNewTransaction(string[] parts)

{

var recipient = parts.FirstOrDefault();

while (string.IsNullOrEmpty(recipient))

{

Console.Write("Recipient: ");

recipient = Console.ReadLine();

}

var amount = double.TryParse(parts.Skip(1).FirstOrDefault(), out double temp)

? temp

: 0;

while (amount <= 0)

{

Console.Write("Amount: ");

amount = double.TryParse(Console.ReadLine(), out temp) ? temp : 0;

}

if (blockchain.GetAccountAmount(account) < amount)

{

Console.WriteLine("Account amount is not enough");

return;

}

var index = blockchain.NewTransaction(account, recipient, amount);

Console.WriteLine($"Transaction added to block #{index}");

}

private static Task MineAndPrintNewBlock()

{

return Task.Run(() =>

{

var block = blockchain.Mine(account);

Console.WriteLine(block.ToString());

});

}

private static void PrintBlockchain()

{

Console.WriteLine($"Blocks count = {blockchain.Chain.Count}");

foreach (var block in blockchain.Chain)

{

Console.WriteLine(block.ToString());

if (block.Transactions.Count > 0)

Console.WriteLine(string.Join("\r\n", block.Transactions));

}

Console.WriteLine();

}

private static void PrintCurrentTransactions()

{

Console.WriteLine($"Current transactions {

blockchain.CurrentTransactions.Count}:");

Console.WriteLine(string.Join("\r\n", blockchain.CurrentTransactions));

}

private static async void SyncBlockchain()

{

await communicator.SyncAsync();

Console.WriteLine("Sync messages sended");

}

private static void SaveBlockChain(string fileName)

{

if (string.IsNullOrWhiteSpace(fileName))

fileName = "chain.protobuf";

try

{

blockchain.SaveFile(fileName);

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

Console.WriteLine("Blockchain saved to " + fileName);

}

private static void LoadBlockChain(string fileName)

{

if (string.IsNullOrWhiteSpace(fileName))

fileName = "chain.protobuf";

if (!File.Exists(fileName))

{

Console.WriteLine("No file");

return;

}

try

{

communicator.Blockchain = Blockchain.FromFile(fileName);

Console.WriteLine("Blockchain loaded from " + fileName);

PrintBlockchain();

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

}

private static string GetPipeIdFromProcessId(int processId)

{

return $"Pipe-{processId}";

}

}

Windows Forms:

public partial class MainForm : Form

{

private Communicator communicator;

private Blockchain blockchain;

private string userAccount;

public MainForm()

{

InitializeComponent();

var pipeId = ProcessPipeHelper.GetCurrentPipeId();

blockchain = Blockchain.CreateNew();

communicator = new Communicator(

new PipeServer<List<Block>>(pipeId),

new PipeClientFactory<List<Block>>())

{

Blockchain = blockchain

};

communicator

.ConnectTo(ProcessPipeHelper

.GetNeighborPipesIds());

senderTextBox.Text = userAccount = AskForAccount(true);

UpdateBlocksList();

UpdateTransactionsList();

UpdateUserAmount();

blockchain.BlockchainReplaced += (s, a) =>

{

UpdateBlocksList();

UpdateTransactionsList();

UpdateUserAmount();

};

}

protected override void OnClosing(CancelEventArgs e)

{

communicator.Close();

base.OnClosing(e);

}

private void AccountAmount\_Click(object sender, EventArgs e)

{

var account = AskForAccount(false, userAccount);

if (account == null)

return;

var amount = blockchain.GetAccountAmount(account);

MessageBox.Show(amount + " монет на счету аккаунта \""

+ account + "\"", "Вывод счета");

}

private string AskForAccount(bool requied, string def = "")

{

var account = Interaction.InputBox("Введите аккаунт", "Ввод аккаунта", def);

if (string.IsNullOrWhiteSpace(account))

{

MessageBox.Show("Аккаунт не может быть пустым");

if (requied)

return AskForAccount(requied, def);

return null;

}

return account;

}

private void SaveBlockchainClick(object sender, EventArgs e)

{

if (saveFileDialog.ShowDialog() == DialogResult.OK)

{

try

{

blockchain.SaveFile(saveFileDialog.FileName);

MessageBox.Show("Успешно сохранено");

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

}

private void LoadBlockchainClick(object sender, EventArgs e)

{

if (openFileDialog.ShowDialog() == DialogResult.OK)

{

try

{

communicator.Blockchain = blockchain

= Blockchain.FromFile(openFileDialog.FileName);

UpdateBlocksList();

UpdateTransactionsList();

UpdateUserAmount();

blockchain.BlockchainReplaced += (s, a) =>

{

UpdateBlocksList();

UpdateTransactionsList();

UpdateUserAmount();

};

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

}

private void SwithAccountClick(object sender, EventArgs e)

{

var newAccount = AskForAccount(false, userAccount);

if (!string.IsNullOrEmpty(newAccount))

senderTextBox.Text = userAccount = newAccount;

}

private void ExitClick(object sender, EventArgs e)

{

Close();

}

private async void SyncGetBlocksClick(object sender, EventArgs e)

{

await communicator.SyncAsync(true);

}

private async void SyncBlocksClick(object sender, EventArgs e)

{

await communicator.SyncAsync();

}

private void SendTransactionClick(object sender, EventArgs e)

{

var recipient = recipTextBox.Text;

if (string.IsNullOrEmpty(recipient))

{

MessageBox.Show("Аккаунт получателя не может быть пустым");

return;

}

var amount = (double)amountDropDown.Value;

if (blockchain.GetAccountAmount(userAccount) < amount)

{

MessageBox.Show("Не хватает монет на счету");

return;

}

if (MessageBox.Show("Отправить " + amount + " монет на аккаунт \""

+ recipient + "\"", "Подтверждение",

MessageBoxButtons.OKCancel) != DialogResult.OK)

return;

blockchain.NewTransaction(userAccount, recipient, amount);

UpdateTransactionsList();

UpdateUserAmount();

}

private void UpdateBlocksList()

{

blocksDataGridView.Rows.Clear();

foreach (var block in blockchain.Chain)

{

blocksDataGridView.Rows.Add(new object[]

{

block.Index,

block.Date,

block.Proof,

block.PreviousHash

});

}

}

private void UpdateTransactionsList(int blockIndex = -1)

{

transactionsDataGridView.Rows.Clear();

var transactions = blockIndex >= 0

? blockchain.Chain

.FirstOrDefault(b => b.Index == blockIndex)?

.Transactions ?? new List<Transaction>()

: blockchain.CurrentTransactions;

foreach (var transaction in transactions)

{

transactionsDataGridView.Rows.Add(new object[]

{

transaction.Date,

transaction.Amount,

transaction.Sender,

transaction.Recipient

});

}

}

private void UpdateUserAmount()

{

amountDropDown.Maximum = (decimal)blockchain.GetAccountAmount(userAccount);

}

private void BlocksDataGridViewSelectionChanged(object sender, EventArgs e)

{

if (blocksDataGridView.SelectedCells.Count == 0)

return;

var index = blocksDataGridView

.Rows[blocksDataGridView.SelectedCells[0].RowIndex]

.Cells["Index"].Value as int?;

UpdateTransactionsList(index ?? -1);

}

private async void MineClick(object sender, EventArgs e)

{

await Task.Run(() => blockchain.Mine(userAccount));

MessageBox.Show("Создание блока завершено");

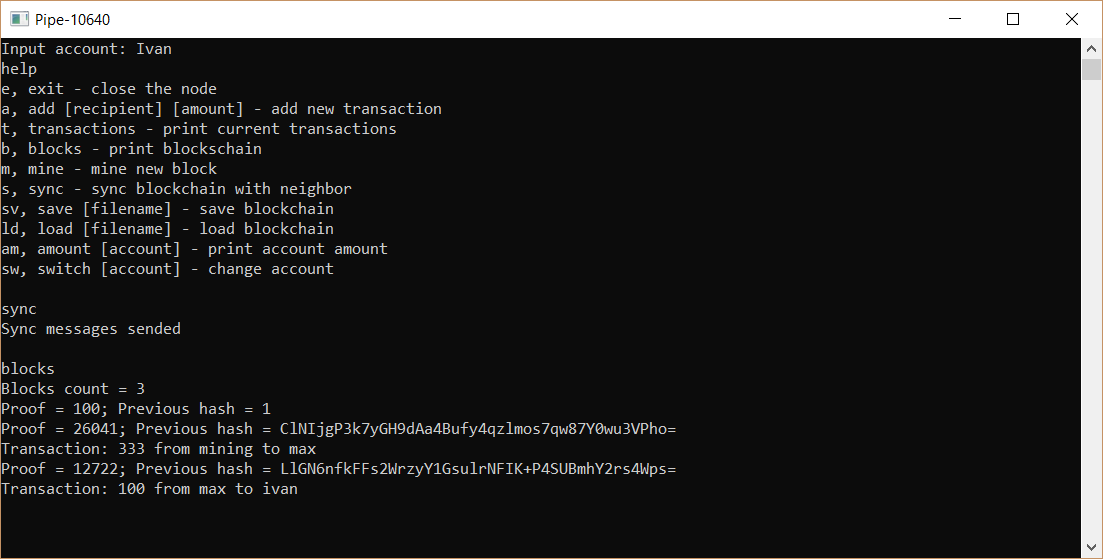
UpdateBlocksList();

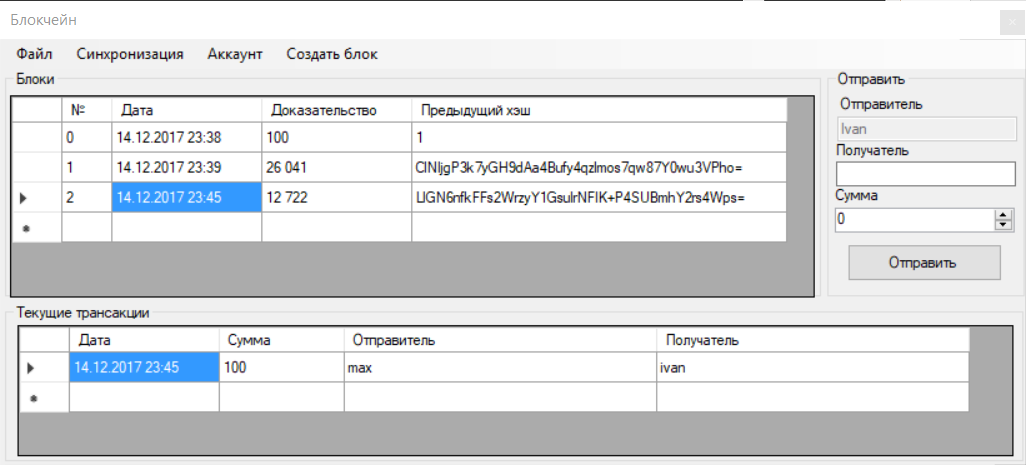
UpdateTransactionsList();

}

}

## Приклад синхронізації після операцій у двох блокчейнах у різних процесах





## GitHub

Код курсової роботи, історія його змін і електронний варіант звіту можно знайти на моій сторінці на сайті GitHub

<https://github.com/Tirraon/Blockchain.Net>

Репозіторій з кодом був відкритий після завершення роботи 15 грудня 2017 року.

# **Висновок**

Вивчені принцип дії та базові алгоритми системи Blockchain шляхом створення мінімальної реалізації криптовалютного блокчейну на мові C# з використанням NamedPipeServerStream та NamedPipeClientStream для коммунікації між нодами мережі блокчейн.

Були використані сучастні засоби створення проектів на мові C# - .Net Core (для проекту тестів) та .Net Standard (для спільних проектів Core і Pipe), які дозволяють зручніше використовувати їх та потенційно дає можливість використовувати на інших ОС. Нажаль, Windows Forms, який був однією з умов роботи, заважає роботі за межами ОС Windows.

Була використана і протестована бібліотека Google Protobuf для бінарної серіалізації та десеріалізації замість класичного BinaryReader/Writer. Окрім незначного приросту швидкості це дає можливість використовувати серіалізовані данні поза межами .Net.

# **Список використаної літератури**

1. [En] Creating a Server Using Named Pipes <https://www.codeproject.com/Articles/864679/Creating-a-Server-Using-Named-Pipes>
2. [En] Proof Of Work <https://en.bitcoin.it/wiki/Proof_of_work>
3. [En] Learn Blockchains by Building One <https://hackernoon.com/learn-blockchains-by-building-one-117428612f46>
4. Bitcoin in a nutshell — Mining <https://habrahabr.ru/post/320178/>
5. Bitcoin in a nutshell — Transaction <https://habrahabr.ru/post/319860/>
6. Майнинг и как он работает: матчасть <https://habrahabr.ru/post/204008/>