SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE

Fakulta elektrotechniky a informatiky

Evidenčné číslo: FEI-104376-104261

Softvér na prenos súborov

Diplomová práca

2024 **Bc. Tomáš Lizák**

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-100863-104261

SOFTVÉR NA PRENOS SÚBOROV
DIPLOMOVÁ PRÁCA

Študijný program:	Robotika a kybernetika
Študijný odbor:	kybernetika
Školiace pracovisko:	Ústav robotiky a kybernetiky
Vedúci záverečnej práce/školiteľ:	Ing. Juraj Slačka, PhD.

Bratislava 2024 **Bc. Tomáš Lizák**

Slovenská technická univerzita v Bratislave
Ústav robotiky a kybernetiky

Fakulta elektrotechniky a informatiky
Akademický rok: 2023/2024
Evidenčné číslo: FEI-104376-104261

ZADANIE DIPLOMOVEJ PRÁCE

Študent:	**Bc. Tomáš Lizák**
ID študenta:	104261
Študijný program:	robotika a kybernetika
Študijný odbor:	kybernetika
Vedúci práce:	Ing. Juraj Slačka, PhD.
Vedúci pracoviska:	prof. Ing. František Duchoň, PhD.
Miesto vypracovania:	Ústav robotiky a kybernetiky

Názov práce: **Softvér na prenos súborov**

Jazyk, v ktorom sa práca vypracuje: slovenský jazyk

Špecifikácia zadania:

Úlohou študenta v tejto práci je navrhnúť a implementovať softvér na komunikáciu v jazyku C#. Softvér by mal umožňovať prenos súborov medzi používateľmi, pričom by mal vedieť zabezpečiť bezpečnosť a anonymitu prenosu dát. Softvér by mal tiež podporovať voľbu medzi rýchlosťou a bezpečnosťou komunikácie pomocou rôznych protokolov a vrstiev. Softvér by mal mať moderný dizajn, efektívne využívať pamäť aj procesor a byť odolný voči chybám.

Úlohy:

1. Analyzujte existujúce možnosti a riešenia na prenos súborov a porovnajte výhody a nevýhody centralizovaného a decentralizovaného prístupu.
3. Navrhnite architektúru a funkcie vášho softvéru.
4. Naprogramujte váš softvér v jazyku C# s použitím vhodných knižníc a nástrojov, ktoré budú podporovať bezpečnosť a anonymitu prenosu dát pomocou rôznych protokolov a vrstiev.
5. Navrhnite moderný, prehľadný a intuitívny dizajn programu.
6. Vyhodnoťte dosiahnuté výsledky a spíšte záver.

Termín odovzdania diplomovej práce:	10. 05. 2024
Dátum schválenia zadania diplomovej práce:	15. 01. 2024
Zadanie diplomovej práce schválil:	prof. Ing. Jarmila Pavlovičová, PhD. – garantka študijného programu

Poďakovanie

Chcel by som poďakovať vedúcemu diplomovej práce Ing. Jurajovi Slačkovi, PhD. za pomoc, ochotu, a čas, ktorý mi venoval ako pri konzultáciách, tak aj mimo nich.

ANOTÁCIA DIPLOMOVEJ PRÁCE

Slovenská technická univerzita v Bratislave
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný odbor: kybernetika

Študijný program: Robotika a kybernetika

Autor: Bc. Tomáš Lizák

Diplomová práca: Softvér na prenos súborov

Vedúci diplomovej práce: Ing. Juraj Slačka, PhD.

Mesiac, rok odovzdania: Máj, 2024

Kľúčové slová: prenos súborov, decentralizované systémy, blockchain, bezpečnosť dát, SSL/TLS, C#, WPF, SQLite, certifikáty

V dnešnej dobe digitalizácie a masívneho zdieľania dát je bezpečnosť prenosu súborov kľúčová. Táto diplomová práca sa zameriava na porovnanie centralizovaných a decentralizovaných prístupov k prenosu súborov. Centralizované systémy, hoci poskytujú pohodlné riešenia pre užívateľov, sú sprevádzané potenciálnymi bezpečnostnými rizikami a obmedzeniami súkromia. Na druhej strane, decentralizované systémy, využívajúce technológie ako blockchain, ponúkajú zvýšenú bezpečnosť a transparentnosť, čím zabraňujú cenzúre a manipulácii. V rámci práce sme vyvinuli a otestovali prototyp softvéru, ktorý integruje oba prístupy a umožňuje porovnanie ich vlastností, bezpečnosti a používateľskej prívetivosti. Práca analyzuje technologické aspekty, ako sú SSL/TLS protokoly a vývoj vlastného komunikačného protokolu pre prenos dát.

MASTER THESIS ABSTRACT

Slovak University of Technology in Bratislava
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Branch of Study: Cybernetics

Study Programme: Robotics and Cybernetics

Author: Bc. Tomáš Lizák

Master Thesis: File Transfer Software

Supervisor: Ing. Juraj Slačka, PhD.

Year, Month: 2024, May

Keywords: file transfer, decentralized systems, blockchain, data security, SSL/TLS, C#, WPF, SQLite, certificates

In today's era of digitalization and massive data sharing, the security of file transfers is crucial. This thesis focuses on comparing centralized and decentralized approaches to file transfer. While centralized systems provide convenient solutions for users, they are accompanied by potential security risks and privacy limitations. On the other hand, decentralized systems, utilizing technologies such as blockchain, offer enhanced security and transparency, preventing censorship and manipulation. Within this work, we have developed and tested a software prototype that integrates both approaches, allowing for the comparison of their features, security, and user-friendliness. The work analyzes technological aspects such as SSL/TLS protocols and the development of a proprietary communication protocol for data transfer.

Obsah

Zoznam použitých skratiek

GUI	Graphical User Interface
WPF	Windows Presentation Foundation
LTS	Long Term Support
UI	User Interface
IT	Information Technology
ID	Identification
SSL	Secure Socket Layer
TLS	Transport Layer Security
OID	Object Identifier
IETF	Internet Engineering Task Force
CSV	Comma-separated Values
ORM	Object Relational Mapper
SDI	Single Document Interface
P2P	Peer to Peer
SPOF	Single Point of Failure
GUID	Globally Unique Identifier
UUID	Universally Unique Identifier
POW	Proof of Work
POS	Proof of Stake
DPOS	Delegated Proof of Stake
PBFT	Practical Byzantine Fault Tolerance

Úvod

V dnešnej dobe je bezpečnosť pri prenose dát kritickým komponentom v digitálnom svete. Naše každodenné transakcie, komunikácia a interakcie sú neustále cieľom útokov a pokusov o kompromitáciu. Preto je kľúčové zabezpečiť, aby naše systémy, aplikácie a riešenia boli odolné voči takýmto hrozbám. Táto diplomová práca, sa pokúsi preniknúť do hĺbky tejto problematiky a priniesť nový pohľad na centralizované a decentralizované prenosy dát.

V prvej časti práce sa budeme zaoberať základmi tvorby spoľahlivého a vláknovo bezpečného grafického užívateľského rozhrania (GUI).

Ďalšou kľúčovou časťou tejto práce bude preskúmanie rôznych komponentov, ktoré sprevádzajú vývoj robustných aplikácií - od konfiguračných súborov po záznamové systémy. Tieto mechanizmy nám pomáhajú lepšie porozumieť fungovaniu našich aplikácií a sú nevyhnutné pre ich správne a efektívne fungovanie.

V súvislosti s bezpečnostnými otázkami sa zameriame na aktuálne metódy a riešenia v oblasti bezpečnosti, vrátane technológie blockchain a jeho využitia v decentralizovaných systémoch. Blockchain, s jeho schopnosťou poskytnúť transparentný a nemenný záznam transakcií, predstavuje výborný nástroj v boji za bezpečnejšiu a dôveryhodnejšiu digitalizáciu.

Budeme sa tiež podrobne venovať problematike decentralizovaných uzlov, presúvaniu dát a riadeniu toku dát. Keď sa pozrieme, ako rýchlo sa svet okolo nás posúva a stáva sa čoraz viac ovládnutý internetom, je veľmi dobre rozumieť aspoň z časti týmto mechanizmom.

1 Analýza existujúcich riešení a smerovanie našej aplikácie

Pri zameraní na prenos súborov medzi používateľmi, môžeme centralizované a decentralizované prístupy porovnať s dôrazom na ich špecifické vlastnosti a vplyvy na užívateľskú skúsenosť, bezpečnosť a kontrolu.

1.1 Centralizovaný prístup

Centralizované systémy pri prenose dát, sú postavené okolo jedného alebo niekoľkých centrálnych serverov, ktoré iniciujú, spravujú alebo sprostredkúvajú spojenie medzi dvoma užívateľmi. Logika fungovania takéhoto servera sa môže výrazne líšiť podľa konkrétnej implementácie, no základná myšlienka spočíva v tom, že poskytovateľ dát sa pripojí na centrálny server, kde uloží svoj súbor alebo serveru dá informácie o tom aké dáta on ako užívateľ poskytuje. Keď sa neskôr pripojí žiadateľ dát so žiadosťou o dáta, môže si ich stiahnuť priamo z centrálneho servera alebo ak centrálny server vie, kde sa nachádzajú, poskytne tuto informáciu žiadateľovi, aby umožnil spojenie uzla s poskytovateľom.

Výhody centralizovaného prístupu:

- jednoduchšia implementácia oproti decentralizovanému systému,
- jednotlivé uzly nepotrebujú vedieť, kde sa ostatné uzly nachádzajú, stačí im vedieť nadviazať spojenie s centrálnym serverom.

Nevýhody centralizovaného prístupu:

- nutnosť spravovania centrálneho servera a s tým spojené náklady na prevádzku a údržbu,
- centrálny server predstavuje náchylnosť na centrálny bod zlyhania (SPOF), kde zlyhaním centrálneho servera stráca celá sieť schopnosť správneho fungovania,
- centrálna autorita a rozhodovacie procesy sústredene na jednom mieste môžu mať vo výsledku negatívny vplyv na dôležité aspekty, ako je napríklad aj cenzúra,
- veľkým problémom centralizovaných systémov je aj bezpečnosť, keďže centrálna autorita je sústredená na jednom mieste, je ľahko napadnuteľná a náchylná na rôzne typy útokov a pokusov o kompromitáciu.

Medzi niekoľko populárnych platforiem, ktoré sú široko používané v oblasti zdieľania a ukladania dát online, ktoré využívajú centralizované prístupy patria napríklad: Dropbox, Google Drive, Microsoft OneDrive, Amazon S3, ASUS WebStorage a pod.

1.2 Decentralizovaný prístup

Pri decentralizovaných systémoch je vynechaná časť centrálneho servera a logika fungovania takejto siete je upravená tak, aby sa uzly medzi sebou dokázali skontaktovať aj bez pomoci prostredníka, ide o tzv. P2P komunikáciu. To sa v praxi docieľuje napríklad spôsobmi inicializačných uzlov, distribuovaných hašovacích tabuliek, či verejne dostupných zoznamov.

Výhody decentralizovaného prístupu:

- bez potreby vytvorenia a udržiavania centrálneho servera,
- absencia autority a možnosti cenzúry či iných nežiadúcich javov spojených s mocou sústredenou na jednom mieste,
- bezpečnosť a robustnosť decentralizovaných systémov je porovnateľne lepšia ako pri centralizovaných systémoch, nakoľko je samotným následkom absencie autority,
- absencia zraniteľného bodu zlyhania (SPOF) je taktiež ďalším priamym dôsledkom absencie autority, a teda takýto systém sa stáva prakticky nezastaviteľným,
- možnosť nadstavby *blockchain*-u.

Nevýhody decentralizovaného prístupu

- náročnejšia implementácie jednotlivých uzlov,
- P2P siete sa často spájajú s distribúciou autorsky chráneného materiálu, čo môže viesť k právnym problémom pre užívateľov. Toto riziko sa vyskytuje aj pri centralizovanom prístupe pri prenose dát medzi užívateľmi, no centrálna autorita môže byť navrhnutá tak, aby mala prostriedky na riešenie tohto problému. Pri decentralizovanom systéme, ale nie sú možnosti ako s týmto problémom bojovať.

Medzi najpopulárnejšiu decentralizovanú platformu, ktorá zároveň používa P2P komunikáciu patrí *BitTorrent*, za zmienku taktiež stoji aj ich nový *projekt BitTorrent File System*, ktorý poskytuje distribuované úložisko dát na báze *blockchain*-u.

1.3 Plán našej aplikácie

V našom projekte sme sa rozhodli za integráciu dvoch hlavných funkcionalít, ktoré nám umožnia skúmať rozdiely medzi centralizovaným a decentralizovaným prístupom v praxi. Každá z týchto funkcionalít bude slúžiť na demonštráciu a testovanie špecifických výhod a limitácií oboch prístupov.

V rámci centralizovaného prístupu implementujeme funkcionalitu sťahovania súborov. Táto funkcia bude zahŕňať centrálny server, ktorý bude slúžiť ako sprostredkovateľ medzi poskytovateľmi a žiadateľmi súborov. Server bude uchovávať identifikátory dostupných súborov, ktoré používatelia chcú zdieľať a pri žiadosti o sťahovanie bude informovať žiadateľa, kde sa súbor nachádza a ako ho možno stiahnuť.

Pre decentralizovaný prístup navrhneme systém, v ktorom bude *blockchain* slúžiť na transparentné a nemenné ukladanie záznamov o súboroch. Používatelia budú môcť požiadať sieť o uloženie ich súborov, pričom každý záznam o súbore bude zaznamenaný na *blockchain*, zabezpečujúc tak jeho dostupnosť, autentickosť a nezmeniteľnosť.

2 Voľba technológií

Pri vývoji softvéru je jednou z kľúčových rozhodnutí voľba správnej technológie. Vo svete IT je mnoho nástrojov, jazykov a platforiem, z ktorých si môžeme vyberať.

2.1 .NET 8.0

.NET 8.0, je najnovšia iterácia platformy .NET, prináša mnoho vylepšení a optimalizácií v porovnaní s predchádzajúcimi verziami je výkonnejší, flexibilnejší a optimalizovaný pre moderné vývojové postupy. Jednou z obrovských výhod, napríklad oproti .NET Framework, je aj *cross-platform* podpora, to znamená, že náš program sa bude dať spustiť, na platformách: *Windows, macOS, Linux*. Pozitívum je aj LTS, teda dlhá podpora pre .NET 8.0 62[1][2].

2.2 WPF

WPF je výkonný nástroj pre vytváranie desktopových aplikácií s pokročilými grafickými rozhraniami. Jeho možnosti v oblasti zobrazenia, štýlovania a animácií nám umožňujú vytvoriť intuitívne a atraktívne GUI pre koncových používateľov. WPF má, ale aj veľké negatívum, a to podpora len pre platformu *Windows*, v budúcnosti teda možno budeme zvažovať prechod na inú GUI technológiu, napríklad *AvaloniaUI, Uno Platform*, či iné. No pre začiatok sme si zvolili WPF aj z dôvodu, že s ním už máme skúsenosti.

3 Základné prvky programu

Pred tým, ako sa pustíme do tvorby komplexného kódu, musíme si najprv implementovať základy nášho programu.

3.1 UI Vlákno

Podobne ako väčšina GUI technológií, aj WPF má hlavné vlákno, označované ako *UI Vlákno*, ktoré obsluhuje všetky GUI komponenty. To znamená, že zmeny v GUI, ako sú aktualizácie textu, animácie alebo zmeny veľkosti, sa musia vykonávať v rámci tohto vlákna. V prípade, ak sa pokúsime o zmenu GUI komponentu z iného vlákna, aplikácia vyvolá výnimku **System.InvalidOperationException** s hláškou: **'The calling thread cannot access this object because a different thread owns it.'**
A následne sa program ukončí. Tento problém sa dá vyriešiť viacerými spôsobmi.

3.1.1 Dispatcher, Invoke, BeginInvoke

Pri menších projektoch sa často využíva volanie *UI Vlákna* pomocou triedy dispatchera. Dispatcher funguje ako most medzi pracovnými vláknami a *UI vláknom*, umožňuje pracovným vláknam, aby delegovali vykonanie určitých akcií na *UI vlákno*. Môžeme si vybrať ešte medzi funkciami *Invoke* a *BeginInvoke*.

Metóda *Invoke* je synchrónna a čaká, kým *UI vlákno* nevykoná poskytnutú úlohu. Je to užitočné, keď potrebujeme okamžitý výsledok pred pokračovaním v práci v pracovnom vlákne.

Na druhej strane, *BeginInvoke* je asynchrónna a neblokuje volajúce vlákno. Vlákno pokračuje vo svojej práci a *UI vlákno* vykoná úlohu, keď bude môcť. Používa sa najmä, keď nezáleží na okamžitom vykonaní akcie a chceme zachovať plynulosť pracovného vlákna [3]. Príklad použitia:

```
Dispatcher.BeginInvoke(new Action(() =>
{
    GuiKomponent.Vlastnost = novaHodnota;
}));
```

3.1.2 BackgroundWorker

Pre zložitejšie operácie, ktoré vyžadujú neblokujúce pozadie aplikácie a zároveň potrebujú komunikovať s *UI vláknom*, je možné využiť koncept pozadia aplikácie známeho ako *BackgroundWorker*. Toto je špeciálny pracovný komponent v .NET, ktorý zjednodušuje spustenie dlhodobých operácií na pozadí a komunikáciu s *UI vláknom* prostredníctvom udalostí ako **DoWork**, **ProgressChanged** a **RunWorkerCompleted**.

My však pôjdeme nad rámec štandardného *BackgroundWorker*-a a vytvoríme si vlastnú implementáciu tohto pozadia aplikácie, ktorá bude vhodnejšia pre zložitejšie aplikácie. V našom prípade použijeme rozšírený model pozadia s **ConcurrentQueue** a *AutoResetEvent* pre efektívnejšie a bezpečné spracovanie správ medzi vláknami.

Vytvoríme si abstraktnú triedu **BaseWindowForWPF**, ktorá bude fungovať ako základná trieda pre všetky WPF okná a bude poskytovať nasledujúce funkcionality:

- **ConcurrentQueue<BaseMsg>:** vláknovo bezpečná fronta správ, ktorá bude umožňovať vláknam bezpečne pridávať správy typu *BaseMsg* bez rizika kolízie.
- **AutoResetEvent:** Signálny mechanizmus na upovedomenie pracovného vlákna, že nová správa je pripravená na spracovanie.
- **CancellationTokenSource:** Mechanizmus na bezpečné ukončenie pozadia vlákna pri zatváraní okna alebo ukončení aplikácie.
- **HandleMessage metóda:** Bude kontinuálne spracovávať správy z fronty na pozadí a koordinovať vykonávanie akcií v *UI vlákne* pomocou *Dispatcher.BeginInvoke*.
- **HandlingMessageInGuiThread metóda:** Bude zabezpečovať, že akcie spojené so správami budú vykonané na *UI vlákne*, čím sa zabráni výnimkám z dôvodu prístupu z nesprávneho vlákna.

Takýto model pozadia aplikácie je veľmi vhodný pre aplikácie, kde je potrebné vykonávať zložité operácie na pozadí a zároveň udržiavať nezamrznuté užívateľské rozhranie.

Pridanie správy do fronty a jej spracovanie bude vyzerať nasledovne:

```
public void BaseMsgEnque(BaseMsg baseMsg)
{
    _concurrentQueue.Enqueue(baseMsg);
    _autoResetEvent.Set();
}
```

3.2 Správy medzi vláknami

Správy medzi vláknami alebo inak povedane *ThreadMessages*, budú pravé tie **BaseMsg**, ktoré obsluhuje náš *BackgroundWorker*. Koncept je nasledovný: Vytvoríme ID pre každú správu napríklad takto:

```
public static class MsgIds
{
    public static readonly int ClientStateChangeMessage = 1;
    public static readonly int WindowStateSetMessage = 2;
    public static readonly int ClientSocketStateChangeMessage = 3;
    …
    …
}
```

Následne si ešte pripravíme kontrakt logiku na priraďovanie týchto identifikátorov ku konkrétnym objektom správ. Môžeme to buď spraviť hromadne pri štarte programu alebo každá časť programu si tam pridá iba pravé tie, ktoré ona sama potrebuje. A to týmto spôsobom:

```
contract.Add(MsgIds.WindowStateSetMessage, typeof(WindowStateSetMessage));
```

Na takto číselne identifikované správy nám stačí už len vytvoriť mechanizmus, ktorý po prijatí takejto správy pozrie na identifikátor a následne vyvolá požadovanú akciu. Takýto komponent môžeme nazvať *NumberSwitch*. Logika je jednoduchá, tento switch v sebe bude obsahovať objekt *Knižnice*, kde kľúčom bude identifikátor správy, teda číslo a hodnotou bude požadovaná akcia.

Potom určitá časť programu, ktorá bude očakávať konkrétnu správu, bude musieť vykonať registráciu takejto správy, aby ju jej *BackgroundWorker* vedel prijať. Príklad registrácie:

```
numberSwitch
 .Case(contract.GetContractId(typeof(WindowStateSetMessage)),
(WindowStateSetMessage x) => WindowStateSetMessageHandler(x));
```

Takouto registráciou správy, je už všetko pripravené na to, aby naša metóda **HandlingMessageInGuiThread** zo sekcie 2.1.1, automaticky vyvolávala akcie podľa toho, aká správa jej bude doručená, vyzerať bude nasledovne:

```
private void HandlingMessageInGuiThread()
{
   if (_concurrentQueue.TryDequeue(out BaseMsg? baseMsgFromQueue))
   {
       msgSwitch.Switch(baseMsgFromQueue.ai, baseMsgFromQueue);
   }
}
```

3.3 Zaznamenávanie fungovania aplikácie

Implementácia efektívneho záznamového systému je kľúčová pre správu a údržbu akýchkoľvek komplexných softvérových aplikácií, obzvlášť v prípade viac vláknových aplikácií, kde logovanie poskytuje nielen výborný nástroj pre *ladenie* (z anglického slova *debugging),* ale aj nevyhnutný prostriedok pre vyhodnocovanie pádov, nestabilít alebo neobvyklého správania aplikácie. V našej aplikácii sme sa rozhodli implementovať záznamový systém, ktorý bude zapisovať záznamy do CSV súborov.

Náš systém logovania bude postavený na viacerých základných komponentoch:

- **ConcurrentQueue** a **AutoResetEvent**: Použijeme **ConcurrentQueue** pre bezpečné pridávanie záznamov z rôznych vlákien a **AutoResetEvent** na signalizáciu prítomnosti nových záznamov, ktoré majú byť spracované. Táto logika je podobná nami implementovanému *BackgroundWorker-u*.
- Asynchrónne spracovanie záznamov: Záznamy budú spracované asynchrónne v samostatnom vlákne, čo bude znižovať vplyv zaznamenávania na hlavné vlákno aplikácie a zabezpečovať plynulý beh.
- Konfigurácia zaznamenávania: Nastavenia zaznamenávania budú načítané z konfiguračného súboru, čo bude umožňovať ľahkú modifikáciu parametrov ako je veľkosť fronty, umiestnenie záznamov a veľkostný limit súboru so záznamami.
- Automatická kompresia záznamov: Keď súbor so záznamami dosiahne určenú veľkosť, bude automaticky komprimovaný do ZIP formátu, čo zabezpečuje efektívne využívanie diskového priestoru.
- Detailné zaznamenávanie: Každý záznam bude obsahovať časovú pečiatku, názov súboru, metódu, z ktorej bol záznam vytvorený, meno vlákna, úroveň záznamu a samotný záznam. Takto získame komplexný prehľad o aktivite aplikácie.

Príklad použitia zaznamenávania pri zaznamenaní udalosti ako je spustenie pracovného vlákna, príkaz:

```
Log.WriteLog(LogLevel.DEBUG, "WorkerThread starting");
```

Vytvorený záznam:

```
       Time:13:19:05:001
       Line:34
   Filename:BaseWindowForWPF.cs
Method name:HandleMessage
Thread name:MainWindow_WorkerThread
      Level:DEBUG
    Message:WorkerThread starting
```

Vďaka takto zaznamenaným údajom budeme vedieť efektívne filtrovať, vyhľadávať a analyzovať dianie v aplikácii, čo nám umožňuje rýchlejšie identifikovať a riešiť problémy, ako aj optimalizovať výkon.

3.4 Konfiguračný systém

Vývoj akéhokoľvek softvéru si vyžaduje efektívne spravovanie nastavení a konfiguračných parametrov. V našej aplikácii sme sa rozhodli použiť *app.config* súbor pre každý spustiteľný projekt. Hoci v súčasnom prostredí .NET existujú aj iné metódy správy konfigurácii. No *app.config* poskytuje overenú, známu a spoľahlivú cestu, na ktorú sme zvyknutí.

3.4.1 MyConfigManager

Na spracovanie a správu konfiguračných parametrov si vytvoríme vlastný konfiguračný manažér, **MyConfigManager**. Tento manažér bude slúžiť na dynamické načítanie a aktualizáciu konfiguračných nastavení v reálnom čase, týmto docielime jednoduchý prístup k rôznym typom konfiguračných údajov a poskytneme rozhranie pre ich jednoduché spravovanie.

Kľúčové vlastnosti:

- Dynamické Načítanie a Aktualizácie: Systém bude sledovať zmeny v *app.config* súbore pomocou funkcionality *FileSystemWatcher* a bude automaticky načítavať nové nastavenia bez potreby reštartovania aplikácie.
- Podpora Rôznych Typov Údajov: **MyConfigManager** bude podporovať načítanie rôznych typov dát (string, int, bool atď.) a poskytovať metódy pre ich jednoduché a bezpečné získavanie.
- Vláknovo Bezpečný Prístup: S použitím **ConcurrentDictionary** zabezpečíme, že naše konfiguračné nastavenia budú bezpečne spravované aj vo viac vláknovom prostredí.

Naša aplikácia bude využívať tento *app.config* súbor pre definovanie rôznych nastavení, ako napríklad systémové cesty pre sťahovanie a nahrávanie súborov, IP adresy a porty serverov, ako aj nastavenia pre záznamový systém. Tieto nastavenia teda budú ľahko prispôsobiteľné, a to nám umožňuje rýchlo meniť kľúčové parametre aplikácie bez nutnosti zásahu do kódu. Ukážka *app.config* súboru:

```
<configuration>
    <appSettings>
        <!-- Cesty pre sťahovanie a nahrávanie súborov -->
        <add key="DownloadingDirectory" value="C:\Download"/>
        <add key="UploadingDirectory" value="C:\Upload"/>

        <!—— Sieťové nastavenia -->
        <add key="CentralServerIpAddress" value="192.168.36.132"/>
        <add key="CentralServerPort" value="34258"/>
        <add key="UploadingServerPort" value="34259"/>

        <!-- Logovanie -->
        <add key="UseAsynchronousLogging" value="True"/>
        <add key="SizeLimitInMB" value="10"/>
        <add key="EnableLogging" value="True"/>
        <add key="LoggingDirectory" value="C:\Logs"/>
        <add key="BufferSize" value="1"/>
    </appSettings>
</configuration>
```

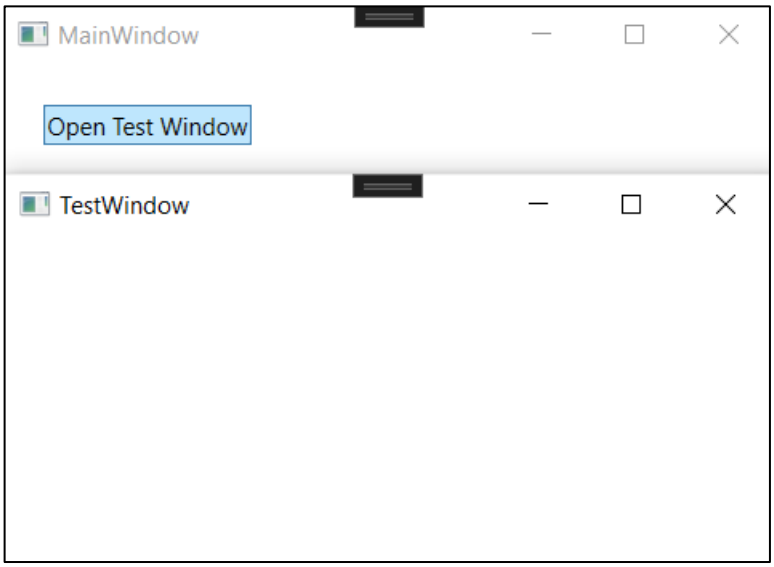
Následne načítanie konfiguračného parametra vieme realizovať napríklad takto:

```
if (!MyConfigManager.TryGetIntConfigValue("UploadingServerPort", out Int32 port))
{
    Log.WriteLog(LogLevel.ERROR, "Invalid server port!");
    return;
}
```

4 Test uvoľňovania zdrojov

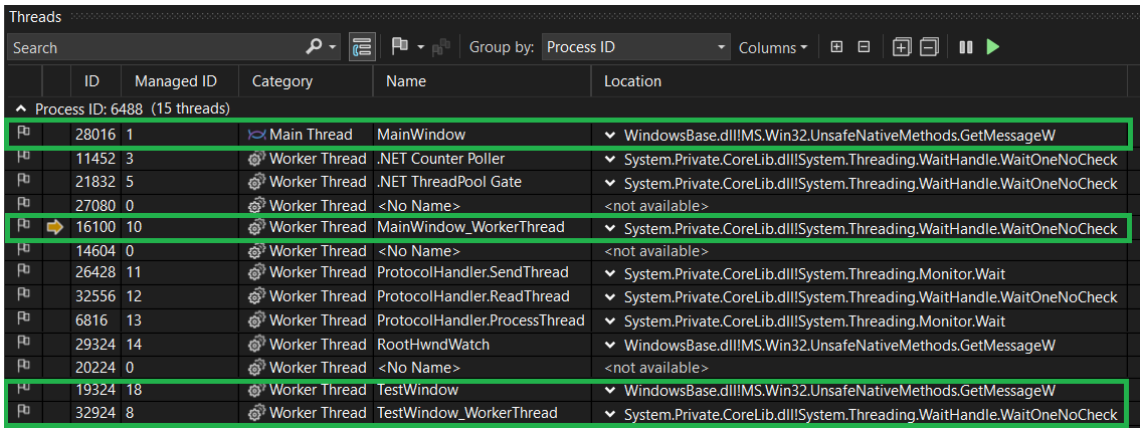
Efektívna správa zdrojov a vlákien je nevyhnutná pre stabilitu a výkon akýchkoľvek moderných aplikácií. Preto sme sa rozhodli otestovať, či naša aplikácia spĺňa tieto dôležité požiadavky prostredníctvom série testov.

Vytvorenie a Otvorenie GUI Okien: Tento krok je dôležitý, pretože nám umožňuje overiť, či aplikácia správne inicializuje a spracováva viacero užívateľských rozhraní. Každé okno by malo mať svoje vlastné *UI vlákno* a *backgroundWorker-a*, čo zabezpečuje, že zdroje sú správne alokované a spravované. Toto je základným stavebným kameňom pre zabezpečenie spoľahlivej viacúčelovej funkcionality aplikácie.



Obr. 1. Hlavné a testovacie okno - Testovanie uvoľňovania zdrojov

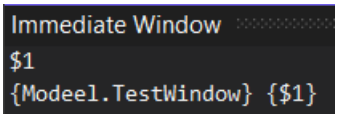
Kontrola Aktívnych Vlákien: Overenie aktívnych vlákien nám poskytuje dôkaz, že naše okná bežia nezávisle a neblokujú hlavné vlákno aplikácie. Toto je kľúčové pre udržanie stavu nezamrznutia aplikácie, najmä pri vykonávaní náročných operácií.



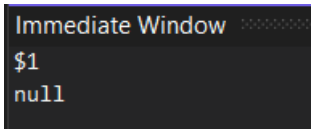
Obr. 2. Aktívne vlákna aplikácie - Testovanie uvoľňovania zdrojov

Zatvorenie Okna a Kontrola Vlákien: Skúmaním, či sa po zatvorení okna správne uvoľnia pridružené vlákna, zabezpečujeme, že aplikácia nezanecháva "visiace" vlákna, ktoré by mohli spôsobiť úniky pamäte a iné problémy s výkonom.

Volanie *Garbage Collectora* a Finálna Kontrola: Tento krok nám potvrdí, že objekty a zdroje sú účinne uvoľnené z pamäte po ich použití. Uvoľnenie zdrojov a čistenie pamäte sú zásadné pre dlhodobú stabilitu a efektívnosť aplikácie.



Obr. 3. Vyhodnotenie objektu podľa ID - Testovanie uvoľňovania zdrojov

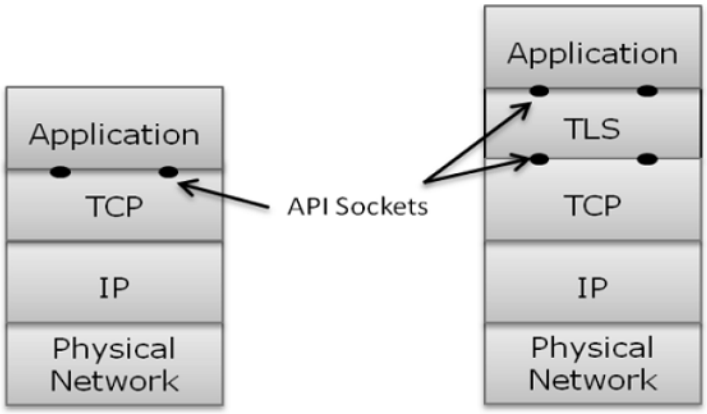


Obr. 4. Vyhodnotenie zatvoreného objektu podľa ID - Testovanie uvoľňovania zdrojov

Výsledkom týchto testov vieme potvrdiť, že naša aplikácia správne spravuje vlákna a zdroje, to nám umožňuje pokračovať vo vývoji aplikácie.

5 SSL/TLS vs čisté TCP/IP, tvorba soketu a testovanie

V dnešnej dobe, keď je bezpečnosť dát a komunikácie kľúčová, sme sa rozhodli vo vývoji našej aplikácie využívať SSL/TLS sokety spolu s tradičnými TCP/IP soketmi. Týmto prístupom necháme užívateľa zvoliť si medzi rýchlosťou čistého TCP/IP protokolu a vylepšenou bezpečnosťou, ktorú ponúka nadstavba SSL/TLS.



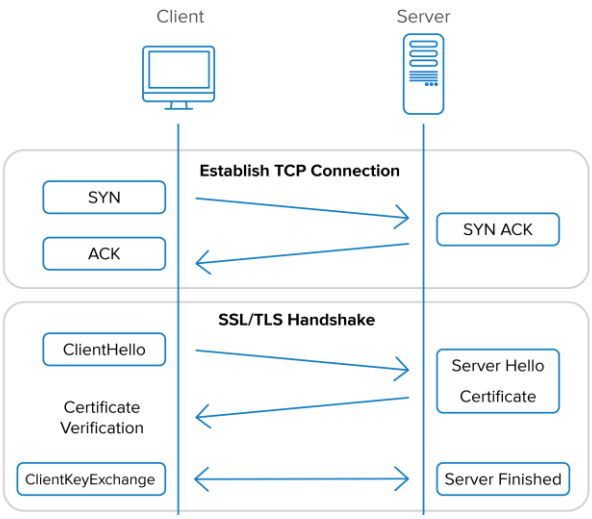
Obr. 5. Diagram protokolových vrstiev čistého TCP/IP a vrstiev rozšírených o TLS [6]

5.1 SSL

SSL bol pôvodný protokol navrhnutý pre zabezpečenie komunikácie na internete. Vytvorila ho spoločnosť Netscape v polovici 90. rokov. SSL prešlo niekoľkými verziami, pričom posledná verzia je SSL 3.0. V praxi sa väčšinou používa už iba TLS, no zaužívane označenie SSL ostalo, a to často krát aj v prípadoch kedy sa hovorí iba čisto o TLS. [5]

5.2 TLS

TLS je nástupcom SSL. Jeho vývoj začal po vydaní SSL 3.0 a bol štandardizovaný organizáciou IETF (Internet Engineering Task Force). Prvá verzia TLS (TLS 1.0) je veľmi podobná SSL 3.0, ale s niekoľkými dôležitými bezpečnostnými a technickými vylepšeniami. My budeme používať TLS 1.2. [5]



Obr. 6. SSL/TLS/TCP diagram [4]

5.3 Certifikáty

Certifikát je digitálny dokument, ktorý obsahuje dvojicu kľúčov využívajúcu asymetrickú kryptografiu, konkrétne verejný a súkromný kľúč. Asymetrická kryptografia, známa tiež ako verejno-súkromná kryptografia, spočíva v použití dvoch rozdielnych kľúčov – jeden na šifrovanie a druhý na dešifrovanie dát. Tento prístup umožňuje bezpečnú výmenu informácií medzi stranami, keďže len držiteľ príslušného súkromného kľúča môže dešifrovať správu, ktorá bola zašifrovaná verejným kľúčom. Certifikáty majú široké spektrum využitia a v našej aplikácii ich využívame predovšetkým na dve kľúčové účely.

Prvým je proces overenia totožnosti užívateľa, to realizujeme požiadavkou na podpísanie (zašifrovanie) určitej správy ich súkromným kľúčom. Ak je následne možné túto správu úspešne dešifrovať príslušným verejným kľúčom, totožnosť užívateľa je považovaná za overenú. Tento mechanizmus poskytuje vysokú úroveň zabezpečenia, keďže len oprávnený držiteľ súkromného kľúča môže vygenerovať platný podpis.

Ďalším využitím certifikátov je SSL/TLS. Certifikáty sú neoddeliteľnou súčasťou SSL/TLS protokolu. Využitie certifikátov v tomto kontexte zabezpečuje šifrovanie dát prenášaných po internete, čo zabraňuje ich odpočúvaniu alebo manipulácii.

5.4 Certifikáty v C#

Na automatickú tvorbu certifikátov v našej aplikácii použijeme knižnicu s názvom *System.Security.Cryptography.X509Certificates*, v tejto knižnici môžeme využiť triedu *CertificateRequest*, ktorá nám vygeneruje certifikát poľa nami zvoleného „hash“ algoritmu a ďalších parametrov ako napríklad OID.

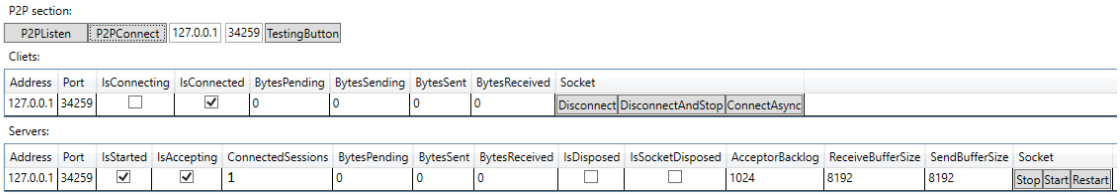
OID sú globálne jedinečné identifikátory priradené objektom, ako sú certifikáty a používajú sa na špecifikáciu aplikácie certifikátu. Pre serverové autentifikačné certifikáty použijeme OID 1.3.6.1.5.5.7.3.1 a pre klientske autentifikačné certifikáty OID 1.3.6.1.5.5.7.3.2. Tieto OID sú štandardizované a určujú použitie certifikátu v kontexte SSL/TLS.

5.5 Tvorba soketu

Pri tvorbe soketov v C# sa budeme spoliehať na štandardné objekty poskytované jazykom C# a .NET. No za účelom dosiahnutia vyššej rýchlosti prenosu dát a zníženia latencie, budeme využívať aj rôzne dostupné nadstavby. Tieto nadstavby budú z veľkej časti tvorené vlastnou implementáciou, ale aj knižnicou(nugetom) *NetCoreServer*, ktorá je špeciálne optimalizovaná pre rýchlosť a asynchrónne spracovanie. V našom prípade nepoužijeme túto knižnicu ako .NET nuget balíček, ale stiahneme si iba potrebné zdrojové kódy z GitHubu, ktoré si následne ešte upravíme podľa našich potrieb. Tieto nadstavby nám umožnia efektívnejšie využívať sieťové zdroje a dosiahnuť tak lepší výkon našej aplikácie.

5.6 Testovanie soketov

Aby sme overili správnosť implementácie soketov, vytvoríme si jednoduché GUI, ktoré nám bude umožňovať dynamicky generovať klientov, serverov a monitorovanie ich interakcie. Tým získame bezprostrednú spätnú väzbu o funkčnosti a správaní soketov v reálnom čase.



Obr. 7. Testovanie Soketov - GUI

S využitím tohto nástroja uskutočníme testy pre štandardné TCP spojenia aj TLS zabezpečené sokety. Vizuálna kontrola v GUI na obrázku 7. naznačuje, že komponenty fungujú správne, avšak pre rigoróznejšiu verifikáciu sme sa rozhodli pre analýzu sieťovej komunikácie prostredníctvom programu *wireshark*.

Time	Source	Destination	Protocol	Length	Info
3.925157	127.0.0.1	127.0.0.1	TCP	44	58646 → 52355 [ACK] Seq=1 Ack=421 Win=10071 Len=0
4.275190	127.0.0.1	127.0.0.1	TCP	47	34259 → 52276 [PSH, ACK] Seq=1 Ack=64 Win=2619648 Len=3
4.275287	127.0.0.1	127.0.0.1	TCP	44	52276 → 34259 [ACK] Seq=64 Ack=4 Win=2619648 Len=0
4.283629	127.0.0.1	127.0.0.1	TCP	59	52276 → 34259 [PSH, ACK] Seq=64 Ack=4 Win=2619648 Len=15
4.283732	127.0.0.1	127.0.0.1	TCP	44	34259 → 52276 [ACK] Seq=4 Ack=79 Win=2619648 Len=0
4.289008	127.0.0.1	127.0.0.1	TCP	65539	34259 → 52276 [ACK] Seq=4 Ack=79 Win=2619648 Len=65495

Obr. 8. Testovanie Soketov – *Wireshark* – TCP *handshake*

Po TCP *handshak*-u, môžeme na obrázku 8. vidieť správu s dĺžkou 65539 bajtov a keďže TCP nie je šifrované, vieme si pozrieť obsah danej správy, ktorú sme posielali pri teste.

```
00000010  7f 00 00 01 7f 00 00 01  85 d3 cc 34 e2 ce c7 66   ........ ...4...f
00000020  98 44 e5 80 50 10 27 f9  a5 03 00 00 62 2e 43 00   ·D··P·'· ····b.C·
00000030  00 00 00 4c 6f 72 65 6d  20 69 70 73 75 6d 20 64   ···Lorem  ipsum d
00000040  6f 6c 6f 72 20 73 69 74  20 61 6d 65 74 2c 20 63   olor sit  amet, c
00000050  6f 6e 73 65 63 74 65 74  75 72 20 61 64 69 70 69   onsectet ur adipi
00000060  73 63 69 6e 67 20 65 6c  69 74 2c 20 73 65 64 20   scing el it, sed
00000070  64 6f 20 65 69 75 73 6d  6f 64 20 74 65 6d 70 6f   do eiusm od tempo
00000080  72 20 69 6e 63 69 64 69  64 75 6e 74 20 75 74 20   r incidi dunt ut
00000090  6c 61 62 6f 72 65 20 65  74 20 64 6f 6c 6f 72 65   labore e t dolore
000000a0  20 6d 61 67 6e 61 20 61  6c 69 71 75 61 2e 20 56    magna a liqua. V
000000b0  65 6c 69 74 20 73 63 65  6c 65 72 69 73 71 75 65   elit sce lerisque
000000c0  20 69 6e 20 64 69 63 74  75 6d 20 6e 6f 6e 20 63    in dict um non c
000000d0  6f 6e 73 65 63 74 65 74  75 72 20 61 20 65 72 61   onsectet ur a era
000000e0  74 2e 20 53 69 74 20 61  6d 65 74 20 6a 75 73 74   t. Sit a met just
000000f0  6f 20 64 6f 6e 65 63 20  65 6e 69 6d 20 64 69 61   o donec  enim dia
00000100  6d 20 76 75 6c 70 75 74  61 74 65 2e 20 49 64 20   m vulput ate. Id
00000110  61 6c 69 71 75 65 74 20  6c 65 63 74 75 73 20 70   aliquet  lectus p
00000120  72 6f 69 6e 20 6e 69 62  68 20 6e 69 73 6c 20 63   roin nib h nisl c
00000130  6f 6e 64 69 6d 65 6e 74  75 6d 20 69 64 20 76 65   ondiment um id ve
00000140  6e 65 6e 61 74 69 73 20  61 2e 20 45 67 65 74 20   nenatis  a. Eget
00000150  67 72 61 76 69 64 61 20  63 75 6d 20 73 6f 63 69   gravida  cum soci
00000160  69 73 20 6e 61 74 6f 71  75 65 20 70 65 6e 61 74   is natoq ue penat
00000170  69 62 75 73 20 65 74 20  6d 61 67 6e 69 73 20 64   ibus et  magnis d
00000180  69 73 2e 20 48 61 62 69  74 61 6e 74 20 6d 6f 72   is. Habi tant mor
00000190  62 69 20 74 72 69 73 74  69 71 75 65 20 73 65 6e   bi trist ique sen
000001a0  65 63 74 75 73 20 65 74  20 6e 65 74 75 73 20 65   ectus et  netus e
000001b0  74 2e 20 49 6e 74 65 72  64 75 6d 20 63 6f 6e 73   t. Inter dum cons
000001c0  65 63 74 65 74 75 72 20  6c 69 62 65 72 6f 20 69   ectetur  libero i
000001d0  64 20 66 61 75 63 69 62  75 73 20 6e 69 73 6c 20   d faucib us nisl
000001e0  74 69 6e 63 69 64 75 6e  74 20 65 67 65 74 20 6e   tincidun t eget n
000001f0  75 6c 6c 61 6d 2e 20 41  6c 69 71 75 61 6d 20 70   ullam. A liquam p
```

Obr.9. Testovanie Soketov – *Wireshark* –TCP správa

Ako vidíme, posielať takéto správy cez internet nemusí byť bezpečné, nakoľko sa k ich obsahom môže ktokoľvek jednoducho dostať.

Time	Source	Destination	Protocol	Length	Info
8.534939	127.0.0.1	127.0.0.1	TCP	56	34259 → 60594 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
8.534969	127.0.0.1	127.0.0.1	TCP	44	60594 → 34259 [ACK] Seq=1 Ack=1 Win=327424 Len=0
8.668195	127.0.0.1	127.0.0.1	TLSv1.2	197	Client Hello
8.668220	127.0.0.1	127.0.0.1	TCP	44	34259 → 60594 [ACK] Seq=1 Ack=154 Win=2619648 Len=0
8.680237	127.0.0.1	127.0.0.1	TLSv1.2	946	Server Hello, Certificate, Server Key Exchange, Server Hello Done
8.680267	127.0.0.1	127.0.0.1	TCP	44	60594 → 34259 [ACK] Seq=154 Ack=903 Win=2619648 Len=0
8.682814	127.0.0.1	127.0.0.1	TLSv1.2	202	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
8.682854	127.0.0.1	127.0.0.1	TCP	44	34259 → 60594 [ACK] Seq=903 Ack=312 Win=2619392 Len=0
8.686471	127.0.0.1	127.0.0.1	TLSv1.2	95	Change Cipher Spec, Encrypted Handshake Message
8.686524	127.0.0.1	127.0.0.1	TCP	44	60594 → 34259 [ACK] Seq=312 Ack=954 Win=2619648 Len=0
8.998520	127.0.0.1	127.0.0.1	TLSv1.2	16428	Application Data

Obr. 10. Testovanie Soketov – *Wireshark* – TLS *handshake*

Na obrázku 10. v kontraste s TCP, TLS *handshake* bol podstatne komplikovanejší, čo koreluje s pridanou vrstvou zabezpečenia. Ďalšou analýzou budeme mať za cieľ zistiť, či

sú dáta prenášané v rámci TLS spojenia prístupné pre potenciálne neoprávnené strany, o tom sa môžeme presvedčiť v správe nasledovanej po *handshak*-u.

Obr. 11. Testovanie Soketov – *Wireshark* – TLS sprava

Ako vidíme na obrázku 11. TLS správa je kompletne zašifrovaná a teda môžeme konštatovať, že je bezpečnejšie posielať takéto správy.

6 Komunikačný protokol

V snahe o dosiahnutie čo najlepšej bezpečnosti a efektivity, sme sa rozhodli implementovať vlastný komunikačný protokol pre posielanie správ medzi klientom a serverom.

Naším cieľom bude vytvoriť protokol, ktorý presne vyhovuje špecifickým požiadavkám a funkčnostiam našej aplikácie. Protokol bude zahŕňať definovanie vlastných správ a mechanizmov na ich spracovanie.

Odosielacia strana komunikácie bude využívať našu triedu **FlagMessagesGenerator** na generovanie rôznych typov správ, ako sú požiadavky na súbory, ich časti a iné správy súvisiace s funkčnosťou aplikácie.

Komunikačná strana, ktorá bude prijímať správu, bude používať na identifikáciu správy nami navrhnutý mechanizmus v triede **BytesFlagSwitch**, ktorý bude identifikovať prijatú správu a podľa preddefinovanej konfigurácie rozhodne o jej prípadnom ďalšom spracovaní triedou **FlagMessageEvaluator**.

6.1 Enum SocketMessageFlag

Začneme vytvorením *enum*-u **SocketMessageFlag**, ktorého funkcionalitu si upravíme podľa našich potrieb. A to tak, že každá *enum* vlastnosť bude obsahovať ešte doplňujúci atribút **StringValue**. Tento atribút budeme využívať na vytvorenie unikátneho identifikátora správy s konkrétnou hlavičkou, ktorú bude následne **FlagMessageEvaluator** rozpoznávať.

Ukážka **SocketMessageFlag**:

```
public enum SocketMessageFlag
{
    // Client => Server
    [StringValue("a.A")]
    FILE_REQUEST,

    [StringValue("a.B")]
    FILE_PART_REQUEST,

    // Server => Client
    [StringValue("b.A")]
    REJECT,

    [StringValue("b.B")]
    ACCEPT,
}
```

6.2 FlagMessagesGenerator

Trieda **FlagMessagesGenerator** sa bude používať na vytvorenie správ, ktoré sú definovane v *enum*-e **SocketMessageFlag**, častokrát sa bude jednať o jednoducho generované správy, ktoré môžu napríklad obsahovať iba hlavičku správy, tak ako je to pri správe **reject**. No správy ako **FilePart** budú o dosť komplexnejšie, nakoľko takáto správa bude obsahovať porovnateľne viac informácii, identifikátor časti súboru, samotné dáta časti súboru a iné.

Príklad jednoduchej správy:

```
public static MethodResult GenerateReject(ISession session)
{
    byte[] request = GenerateMessage(SocketMessageFlag.REJECT);
    bool succes = session.SendAsync(request, 0, request.Length);
    if (succes)
    {
        Log.WriteLog(Log.DEBUG, $"Reject was generated to client: {session.Endpoint}");
    }
    else
    {
        Log.WriteLog(Log.WARNING, $"Unable to send reject to client: {session.Endpoint}");
    }
    return succes ? MethodResult.SUCCES : MethodResult.ERROR;
}
```

6.3 BytesFlagSwitch

Trieda **BytesFlagSwitch** bude komplexná trieda, ktorá bude pozostávať z niekoľko dôležitých časti.

Binding dictionary, ktorej upravíme *equalityComparer*, tak aby vyhovoval našim potrebám. Táto knižnica bude umožňovať identifikovať správy podľa ich hlavičiek a priradiť im príslušné akcie.

Následne si vytvoríme metódu na registráciu sprav, ktoré chceme prijať, jej použitie bude nasledovne:

```
_flagSwitch.Register(SocketMessageFlag.REJECT, OnRejectHandler);
```

Po takomto nastavení nám už stačí každú prijatú správu dať vyhodnotiť do metódy **switch**:

```
protected override void OnReceived(byte[] buffer, long offset, long size)
{
    _flagSwitch.Switch(buffer, offset, size);
}
```

Switch bude v našom prípade veľmi dôležitá metóda, ktorá sa bude starať hneď o niekoľko vecí, ako napríklad kontrola minimálnej dĺžky správy, kontrola hlavičky správy,

ukladanie časti správ do vyrovnávacej pamäte v prípade detekcie neúplnej správy a iné. Nakoniec bude vyvolávať požadovanú akciu priradenú k danej hlavičke.

6.4 FlagMessageEvaluator

FlagMessageEvaluator bude trieda určená na hodnotenie prijatých správ. Jej úlohou bude overovať korektnosť správ a extrahovať z nich informácie, ako napríklad názov súboru alebo jeho veľkosť, a potvrdiť, že sú správy formátované a spracované správne.

7 Prenos súboru

V tejto kapitole sa budeme zaoberať problematikou prenášania súborov prostredníctvom TCP/IP soketu. Ukážeme si typy správ, ktoré budeme medzi užívateľmi posielať a implementujeme reguláciu prenosu, tak aby sme nezahltili prijímateľa.

7.1 Konkretizácia prenášaného súboru

Aby obe strany komunikácie mali jasno v tom, ktorý súbor sa bude prenášať, vytvorili sme nasledovnú logiku: Žiadateľ po pripojení na poskytovateľa odošle žiadosť (správa FILE_REQUEST), v ktorej sú informácie o žiadanom súbore, konkrétne názov súboru a jeho veľkosť. Poskytovateľ žiadosť prijme alebo odmietne (správy REJECT a ACCEPT). V prípade odmietnutia komunikácia končí a obe strany uzatvoria soket. V prípade prijatia, komunikácia pokračuje.

7.2 Dohľaď nad tokom dát

Je nevyhnutné regulovať prenos dát počas prenášania súborov, aby sa predišlo situácii, kde poskytovateľ odosiela dáta rýchlejšie, než je žiadateľ schopný ich efektívne prijímať a spracovať. Bez tejto regulácie by poskytovateľ mohol naraz poslať celý súbor, čo by mohlo preťažiť pamäť alebo zdroje žiadateľa, najmä pri veľkých súboroch, a viesť tak k stratám dát alebo zlyhaniu prenosu. Preto sme implementovali pre prenos nasledovnú logiku.

Žiadateľ si po prijatej správe ACCEPT vytvorí objekt **FileReceiver**, ktorý bude dohliadať na presun daného súboru. Tento **FileReceiver** bude zdieľaný objekt a bude ho môcť vlastniť viacero soketových vlákien naraz, čo znamená, že sťahovanie konkrétneho súboru nebude obmedzené na jedného poskytovateľa daného súboru, ale bude môcť byť sťahovaný z neobmedzeného počtu poskytovateľov súčasne, čím sa môže znížiť potrebný čas na stiahnutie súboru.

7.2.1 FileReceiver

Po svojom vytvorení **FileReceiver** rozdelí súbor na časti, kde každá časť má rovnakú veľkosť s výnimkou poslednej časti, ktorá môže byť menšia v závislosti od celkovej veľkosti súboru. Veľkosť týchto častí si môže užívateľ zvoliť v konfiguračnom súbore. Následne každej časti tohto súboru pridelí jej status, tieto statusy budú nasledovne: WAITING_FOR_ASSIGNMENT = 0, DOWNLOADING = 1 a DOWNLOADED = 2, kde status 0 bude začiatočný status každej časti súboru. Tiež **FileReceiver** vytvorí a uloží

k sťahovanému súboru nový špeciálny súbor, v ktorom bude informácia o počte všetkých častí súboru, ich aktuálnom statuse, veľkosti týchto častí a veľkosti poslednej časti súboru. Tieto informácie sa v špeciálnom súbore budú aktualizovať tak, aby bolo možné pokračovať v sťahovaní aj po reštartovaní programu či zlyhaní sťahovania. Potom sťahovanie prebieha tak, že soketové vlákno vyzve **FileReceiver**, aby mu dal index časti súboru, ktorá ešte nie je stiahnutá a nesťahuje sa. Soketové vlákno následne pošle žiadosť na túto časť poskytovateľovi (správa FILE_PART_REQUEST), táto žiadosť obsahuje informácie o poradovom čísle časti súboru a veľkosti danej časti. Poskytovateľ odpovie zaslaním danej časti súboru (správa FILE_PART), soketové vlákno, ktoré tieto dáta prijme, ich poskytne komponente **FileReceiver**, ktorá sa postará o zapísanie danej časti na jej príslušné miesto. Soketové vlákno následne požiada o ďalšiu časť súboru. Týmto mechanizmom sme dosiahli kontrolu nad tokom dát pri presúvaní súboru, a teda poskytovateľ čaká na žiadosti žiadateľa, aby sme predišli jeho zahlteniu.

8 Offering files

Aby sme uľahčili proces zdieľania súborov, je dôležité, aby sme ako poskytovateľ umožnili žiadateľom zistiť, aké súbory si môžu od nás stiahnuť. S týmto cieľom vytvoríme tzv. „Offering file“. Každý takýto súbor bude reprezentovať jeden stiahnuteľný súbor a bude obsahovať dôležité informácie, ako napríklad názov súboru, jeho veľkosť, a tiež knižnicu poskytovateľov, kde budú uvedené IP adresy, porty, typy soketov, ako aj hodnotenie dôvery poskytovateľov.

Každý poskytovateľ si v konfiguračnom súbore zvolí priečinok na svojom počítači, kam bude môcť vložiť súbory, ktoré chce ponúknuť na stiahnutie. Následne prostredníctvom jednoduchej funkcie náš program automaticky vytvorí pre každý ponúkaný súbor príslušný „Offering file“ vo formáte JSON.

Takýto prístup nielenže zjednodušuje proces vyhľadávania a stiahnutia súborov pre žiadateľov, ale tiež zvýši transparentnosť a dôveru v poskytovateľov tým, že umožňuje hodnotenie ich spoľahlivosti.

Príklad „Offering file“:

```
{

    "FileName":"SampleText.txt",
    "FileSize":2167737,
    "EndpointsAndProperties": {
        "192.168.0.101:34259": {
            "Grade": 0, "TypeOfServerSocket":1
        }
    }
}
```

9 Centrálny server

Ďalším krokom pre zefektívnenie procesu získavania „Offering files“ žiadateľmi je zavedenie centralizovaného systému našej aplikácie. Tento systém bude realizovaný prostredníctvom novej aplikácie, ktorú nazveme „Centrálny server“. Úlohou tohto servera bude fungovať ako sprostredkovateľ „Offering files“ medzi žiadateľmi a poskytovateľmi. Poskytovatelia budú mať možnosť odoslať svoje „Offering files“ na tento centrálny server prostredníctvom správy OFFERING_FILE, žiadatelia budú mať možnosť získať tieto súbory správou OFFERING_FILES_REQUEST

Uloženie týchto „Offering files“ centrálnym serverom si vyžaduje iný prístup než bežné ukladanie súborov poskytovateľmi v jednotlivých JSON súboroch, hlavne kvôli efektívnosti správy veľkého objemu dát. Navyše potrebujeme mechanizmus, ktorým by centrálny server dokázal zlúčiť „Offering files“ v prípade, ak nájde rovnaké súbory ponúkané rôznymi poskytovateľmi. Z týchto dôvodov bude ukladanie realizované do databázy SQLite.

Tento prístup nielenže umožní efektívnejšie spravovanie veľkého množstva „Offering files“, ale tiež zjednoduší proces vyhľadávania a získavania požadovaných súborov pre žiadateľov. Centrálny server tak poskytne robustné riešenie pre správu dostupných zdrojov a ich distribúciu medzi užívateľmi našej aplikácie

9.1 SQLite databáza

SQLite je jednoduchá, samostatná, vysoko spoľahlivá SQL databázová jednotka vytvorená v jazyku C, ktorá nevyžaduje samostatný server na fungovanie a všetky údaje sa uchovávajú v jedinom diskovom súbore. To ju robí ideálnou voľbou pre aplikácie, ktoré potrebujú robustnú databázovú podporu bez komplexnosti a nárokov na zdroje spojených s tradičnými databázovými systémami. SQLite plne podporuje transakčný model ACID (Atomicity, Consistency, Isolation, Durability). [7]

9.1.1 Atomickosť

Atomickosť transakcie znamená, že sa nedá rozdeliť na žiadne menšie časti. Keď je transakcia odoslaná do databázy, musí byť vykonaný buď celý obsah tejto transakcie alebo naopak, nič z tejto transakcie. [7]

9.1.2 Konzistentnosť

Konzistentnosť je základným pilierom transakcií v databáze, ktorý zaisťuje, že všetky operácie udržiavajú dáta v logicky korektnom stave. To znamená, že každá transakcia musí prejsť systémom pravidiel a obmedzení definovaných návrhom databázy, aby sa po jej dokončení zachovala integrita dát. Napriek tomu, že v priebehu transakcie je dovolená dočasná nekonzistencia, konečný stav po ukončení transakcie musí byť konzistentný a odzrkadľovať všetky zmeny presne a úplne. [7]

9.1.3 Izolovanosť

Izolovanosť je jedna z ďalších dôležitých vlastností, ktorá zabezpečuje, že všetky operácie vykonané v rámci jednej transakcie sú oddelené od aktivít ostatných klientov, a to až do momentu, keď je transakcia kompletná. V praxi to znamená, že akýkoľvek prebiehajúci proces úpravy dát je pre ostatné procesy neviditeľný, čím sa predchádza konfliktom a nekonzistenciám. [7]

9.1.4 Trvalosť

Nakoniec, transakcia musí byt trvalá. V prípade, keď je transakcia úspešné odoslaná, musí sa stať permanentnou a nezvrátiteľnou. Po vypnutí procesu, strate napájania, či iných nežiadúcich scenároch, zmeny vykonané transakciou musia byť zachované v databáze. Na druhú stranu, ak databáza stratí napájanie pred úspešným odoslaním transakcie, databáza by znovu po zapnutí nemala obsahovať zmeny spôsobené transakciou. [7]

9.2 SQLite databáza v centrálnom serveri

Na ukládanie „Offering files“ do *SQLite* databázy použijeme dve tabuľky, prvou bude tabuľka *OfferingFiles*, ktorej štruktúra bude nasledovná:

```
CREATE TABLE "OfferingFiles" (
    "OfferingFileId"    TEXT NOT NULL UNIQUE,
    "FileName"    TEXT NOT NULL,
    "FileSize"    INTEGER NOT NULL,
    PRIMARY KEY("OfferingFileId")
);
```

Druhou tabuľkou bude *EndpointsAndProperties* so štruktúrou:

```
CREATE TABLE "EndpointsAndProperties" (
    "Id"     INTEGER NOT NULL UNIQUE,
    "OfferingFileId"       TEXT NOT NULL,
    "Endpoint"      TEXT NOT NULL,
    "Grade"  INTEGER NOT NULL DEFAULT 0,
    "TypeOfServerSocket"   INTEGER NOT NULL DEFAULT 0,
    UNIQUE("OfferingFileId","Endpoint"),
    FOREIGN KEY("OfferingFileId") REFERENCES "OfferingFiles"("OfferingFileId"),
    PRIMARY KEY("Id")
);
```

9.2.1 Dapper

Dapper je objektovo-relačný mapovací nástroj (ORM), ktorý zjednodušuje interakciu medzi .NET aplikáciami a relačnými databázami. Je to malá knižnica, ktorá poskytuje rozhranie pre vykonávanie SQL príkazov. Bol vytvorený tímom v *Stack Overflow,* aby vyhovoval ich potrebám pre rýchlu a efektívnu prácu s dátami, je voľne dostupný ako *open-source* projekt. [8]

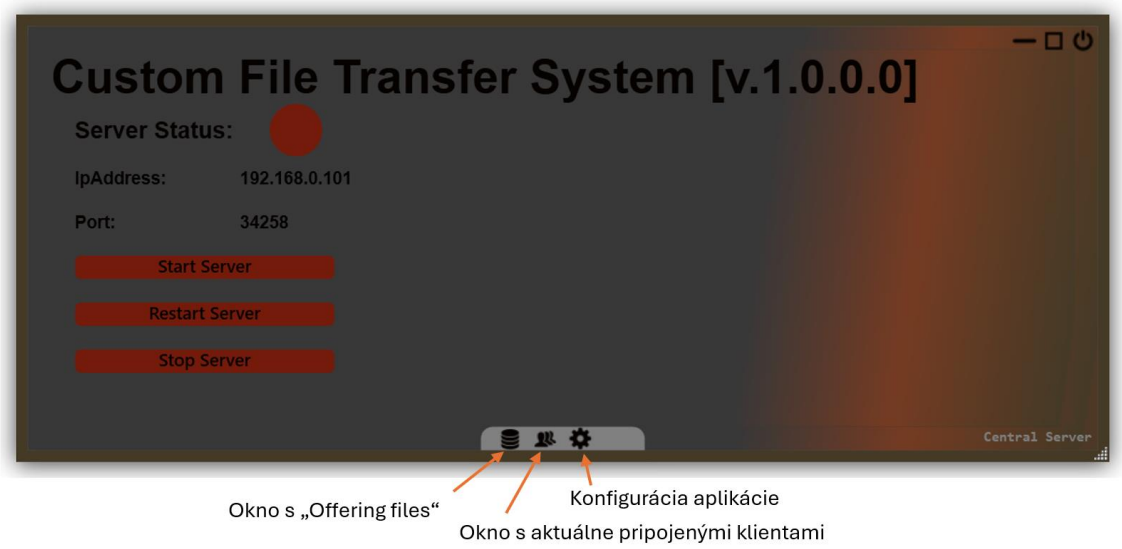
Dapper pridáva minimálnu vrstvu na čisté vykonanie SQL príkazov a priame mapovanie výsledkov na C# objekty. V porovnaní s inými ORM nástrojmi, ako je *Entity Framework*, je *Dapper* navrhnutý tak, aby bol menší a rýchlejší, zatiaľ čo obetuje niektoré abstraktné funkcie. Jeho jednoduchosť a výkon robia z *Dappera* výbornú voľbu pre aplikácie, ktoré potrebujú rýchle operácie s databázou a priamou kontrolou nad SQL príkazmi. [8]

Dapper podporuje asynchrónne operácie, čo umožňuje efektívne využitie zdrojov počas databázových operácií a je kompatibilný so všetkými ADO.NET poskytovateľmi, vrátane *SQLite* [8]

My sme sa teda rozhodli na zápis a čítanie dát z SQLite databázy použiť Dapper na jednoduché a čisté implementácie príkazov INSERT, UPDATE, DELETE a SELECT, ktoré sú optimalizované pre rýchlosť a správu zdrojov, čo je dôležité pre našu aplikáciu.

9.3 Dizajn centrálneho servera

Aplikácia centrálneho servera bude pozostávať z troch okien, hlavného okna, okna s tabuľkou aktuálne pripojených klientov a okna s „Offering files“ tabuľkou vyčítavaných z databázy.



Obr. 12. Dizajn hlavného okna centrálneho servera

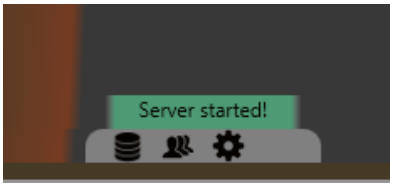
Na hlavnom okne aplikácie obrázku 12. sa nachádzajú tlačidlá na riadenie soketu, taktiež si môžeme všimnúť status signalizujúci stav soketu a v časti konfigurácie aplikácie sa dá nastaviť soketový port, dynamickosť farieb aplikácie a parametre záznamovej komponenty.

Obr.13. Dizajn okna s „Offering files“ tabuľkou

Účel „Offering files“ okna, ktoré vidíme na obrázku 13. je poskytnúť užívateľovi náhľad úložných „Offering files“ priamo v aplikácii bez nutnosti otvárať externý interface databázy.

Obr. 14. Dizajn okna pripojených klientov

Na okne pripojených klientov, ktoré je na obrázku 14. môžeme pozorovať otvorené spojenia aplikácie, pri každom spojení vidíme ID spojenia, adresu a port pripojeného klienta, účel spojenia a nakoniec dostupné operácie, konkrétne možnosť ručného odpojenia klienta od soketu aplikácie.



Obr. 15. Informačná správa pre užívateľa

Na obrázku 15. môžeme pozorovať správu oznamujúcu zapnutie servera, táto správa sa objaví ako reakcia na zapnutie servera tlačidlom štart alebo reštart. Aplikácia týmto spôsobom bude informovať užívateľa o všetkých dôležitých akciách, ktoré budú uskutočnené.

10 Poskytovatelia, žiadatelia a klientska aplikácia

Klientska aplikácia bude slúžiť ako poskytovateľom tak aj žiadateľom. Pri jej vývoji sme sa rozhodli držať zásad dizajnu *Single Document Interface* (SDI). Budeme teda dávať prednosť jednoduchosti a prehľadnosti v našom používateľskom rozhraní a celú aplikáciu udržiavať v jednom hlavnom okne aby sme zabezpečili hladkú interakciu bez nutnosti opustiť hlavný pracovný priestor. Taktiež menej okien znamená menej rušenia a preťaženia informáciami, čo uľahčuje koncentráciu na úlohy.

V aplikácii vytvoríme systém kariet, medzi ktorými sa bude môcť užívateľ prepínať na rôzne riadiace a informačné prvky aplikácie, no spodná časť okna bude statická a budeme tam vedieť nájsť podobne ako v aplikácii centrálneho servera, konfiguráciu aplikácie a informačné správy, tak ako to vidíme na obrázku 15. a naviac skratky na priečinky so sťahovanými a poskytovanými súbormi.

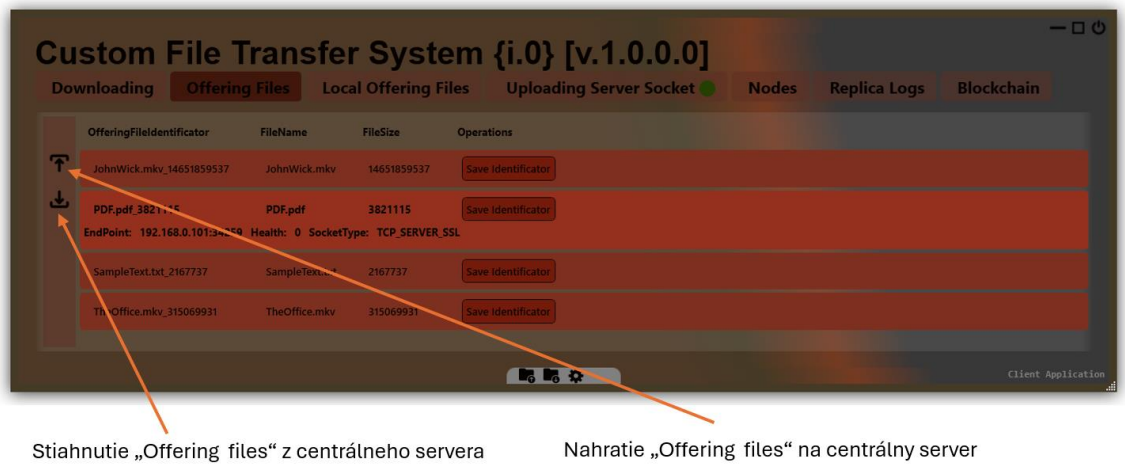
10.1 Karta sťahovania

Karta sťahovania informuje užívateľa o stiahnutých alebo aktuálne sťahovaných súboroch, užívateľ má možnosť po kliknutí na dané sťahovanie vidieť poskytovateľov, od ktorých súbor získava, a taktiež môže pozastaviť označené sťahovanie súboru.

Obr. 16. Klientska karta sťahovania

10.2 Karta „Offering files“

Karta „Offering files“ bude slúžiť na komunikáciu s centrálnym serverom a prácu s „Offering files“.



Obr. 17. Klientska karta „Offering files“

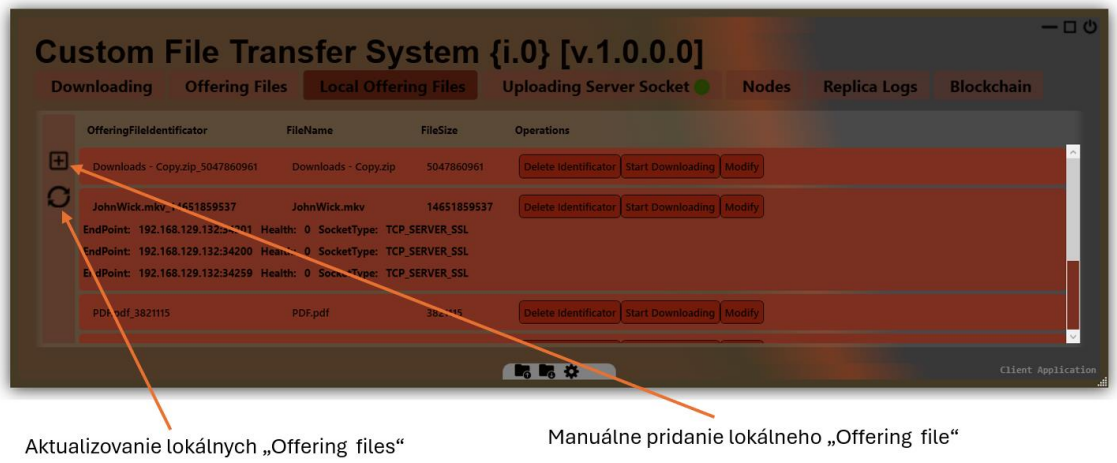
Označene tlačidla z obrázka 17. budú používať na komunikáciu s centrálnym serverom adresu a port, tieto parametre budú určené v konfigurácii aplikácie. V hlavnej tabuľke tohto okna môžeme vidieť „Offering files“, ktoré boli prijaté z centrálneho servera po použití tlačidla stiahnutia, v tabuľke si môžeme všimnúť aj operáciu stiahnutia identifikátora, ktorá stiahne daný „Offering file“ a uloží ho do sťahovaného priečinka. Tam bude môcť byť následne použitý na stiahnutie samotného súboru, na ktorý poukazuje.

V neskoršom vývoji aplikácie by bolo na mieste uvažovať o možnosti filtrovania sťahovaných „Offering files“ z centrálneho servera, či ich delenia na jednotlivé strany po určitom množstve súborov na stranu. Nakoľko by pri vysokom počte týchto súborov mohlo dôjsť k zahlteniu aplikácie alebo neprehľadnosti.

10.3 Karta lokálnych „Offering files“

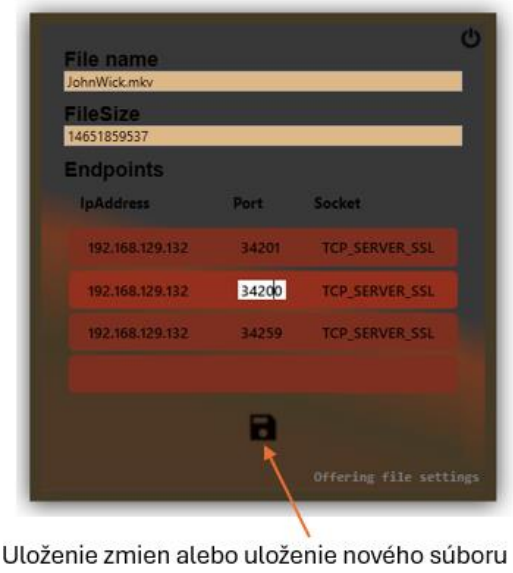
Na karte lokálnych „Offering files“ si môže užívateľ prezerať „Offering files“, ktoré sa nachádzajú v jeho sťahovacom priečinku

Užívateľ ma možnosť zmazať konkrétny „Offering file“, modifikovať ho alebo začať sťahovanie daného súboru. Jednou z možností užívateľa je aj manuálne pridanie nového lokálneho „Offering file“.



Obr. 18. Klientska karta lokálnych „Offering files“

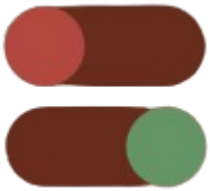
Na účely modifikácie a manuálnej tvorby „Offering files“ súborov si vytvoríme pomocné dialógové okno, ktoré môžeme vidieť na obrázku 19.



Obr. 19. Pomocné dialógové okna na tvorbu a úpravu „Offering files“

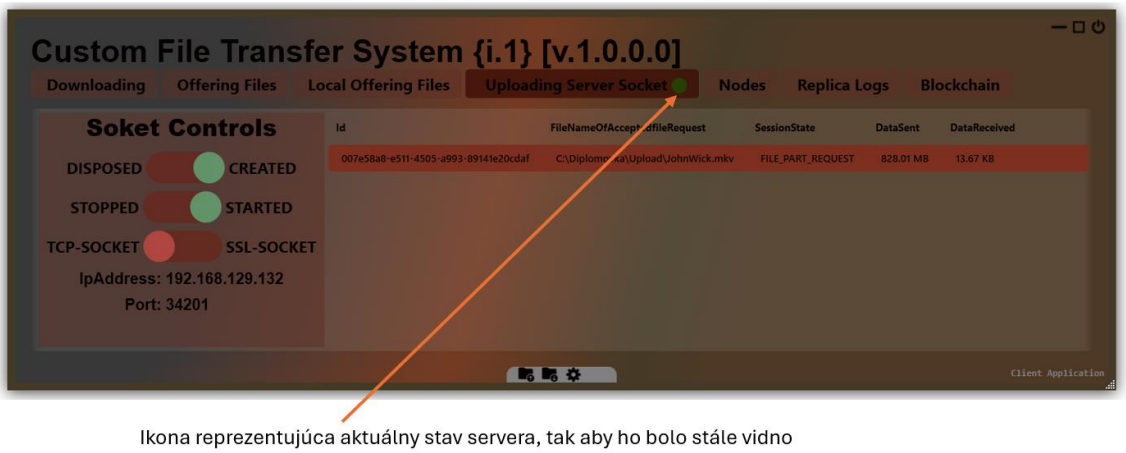
10.4 Karta serverového soketu

Na túto kartu si vytvoríme vlastný komponent prepínacieho tlačidla, ktorý bude meniť farbu podľa svojho stavu (obrázok 20.), taktiež jeho stav nezmeníme už po stlačení naňho, ale až keď dostaneme spätnú väzbu o tom, že korešpondujúca akcia sa naozaj vykonala.



Obr. 20. Vlastné prepínacie tlačidlo

Samotná karta serverového soketu bude pozostávať z dvoch hlavných častí, kontrolnej a informačnej. V kontrolnej časti bude môcť užívateľ riadiť serverový soket, na ktorý sa budú môcť žiadatelia o súbor pripojiť. V informačnej časti užívateľ uvidí pripojených klientov súbor, ktorý od neho sťahujú, množstvo presunutých dát a iné.



Obr. 20. Klientska karta serverového soketu

Port serverového soketu bude nastaviteľný v konfigurácii aplikácie.

Touto kapitolou je prvá časť našej práce s centralizovaním prístupom dokončená.

11 Blockchain, uzly a decentralizácia

V tejto a nasledujúcich kapitolách sa budeme venovať decentralizovanej časti našej aplikácie.

Plán fungovania je nasledovný: Každý užívateľ bude mať možnosť za určitý počet kreditov – ktoré budú v kontexte našej aplikácie poskytované zadarmo – zažiadať o uloženie súboru v sieti. Cena tejto služby bude úmerná veľkosti súboru. Keď užívateľ vyšle takúto žiadosť, ostatní účastníci siete budú môcť rozhodnúť, či si súbor uložia a za odmenu získajú určitý počet kreditov. Užívatelia, ktorí sa rozhodnú zúčastniť na uchovaní súborov, dostanú od žiadateľa súbor v bezpečne zašifrovanej forme. Overovanie integrity týchto súborov bude zabezpečované pravidelnými kontrolami ich existencie, veľkosti a v niektorých prípadoch aj kontrolou celkového *hash*-u súboru. Tento postup zabezpečí, že dáta zostanú nezmenené a bezpečné. V prípade, že sa žiadateľ rozhodne pre vymazanie súboru, všetci, ktorí súbor uchovávajú, by ho mali vymazať a potvrdiť jeho zmazanie. Tento proces bude poskytovať žiadateľom kontrolu nad ich dátami a zároveň zabezpečí, že dáta nebudú uchovávané nad rámec ich požiadaviek. Decentralizovaný prístup, ktorý naša aplikácia bude implementovať, prinesie výhody, ako odolnosť proti cenzúre, distribuovanú bezpečnosť a zvýšenú odolnosť siete vďaka distribuovanej povahe ukladania dát. Každý uzol v sieti bude fungovať ako nezávislý strážca informácií, čím sa znižuje riziko útokov a zlyhaní spojených s centralizovanými dátovými úložiskami. Táto architektúra bude podporovať transparentnosť a dôveru v rámci siete.

11.1 Teória synchronizácie siete

Prvým a veľmi dôležitým krokom pri vytváraní efektívnej decentralizovanej siete bude synchronizácia užívateľov, tak aby všetci jej účastníci mali o sebe navzájom informácie. Aby sme to dosiahli, použijeme koncept inicializačných uzlov. Tieto uzly budú slúžiť ako orientačné body pre novovznikajúce uzly, budú im poskytovať dôležité informácie o existujúcich uzloch a infraštruktúre siete, tým im pomôžu rýchlejšie sa integrovať a efektívne začať komunikovať s ostatnými účastníkmi siete.

Dôkladne navrhnutý systém inicializačných uzlov je nevyhnutný, pretože zabezpečuje, že decentralizovaná sieť zostane koherentná a synchronizovaná aj napriek priebežne sa meniacemu počtu a stave uzlov. Týmto spôsobom bude môcť sieť efektívne rozširovať svoju funkčnosť a zároveň si udržať vysokú úroveň odolnosti a bezpečnosti.

11.2 Uzol ako základný prvok decentralizovanej siete

Ako sme už spomenuli, decentralizované riešenia sa vyznačujú absenciou centrálnej autority, pričom sieť pozostáva z jednotlivých, rovnocenných uzlov. Kritickým aspektom je overovanie totožnosti týchto uzlov, aby sme predišli možnosti, že sa neoprávnené entity budú vydávať za iné uzly a zasahovať do siete, každý uzol v našej sieti bude využívať vlastný certifikát určený výhradne na overovanie identity. Toto overovanie bude prebiehať tak, že každá správa odoslaná uzlom bude podpísaná jeho súkromným kľúčom, zatiaľ čo príjemcovia budú môcť pomocou verejného kľúča overiť platnosť podpisu a tým aj identitu odosielateľa. Bez príslušného certifikátu nebude možné úspešne autentifikovať uzol.

Poďme si teda definovať štruktúru nášho uzla. Každý uzol bude pozostávať s unikátneho identifikátora (v našom prípade GUID), IP adresy, portom a spomínaním verejným kľúčom z certifikátu.

11.2.1 GUID (UUID)

GUID (*Globally Unique Identifier*), známy aj ako UUID (*Universally Unique Identifier*), je štatisticky jedinečný identifikátor používaný v rôznych softvérových aplikáciách.
Tieto identifikátori boli navrhnuté ako časť štandardu, ktorý by umožnil výrazne znížiť riziko duplicity identifikátorov v rozsiahlych distribuovaných systémoch a databázach. Cieľom bolo vytvoriť identifikátory, ktoré by boli unikátne bez ohľadu na to, kde a kedy boli vygenerované. Ide o 128-bitovú hodnotu, ktorá sa typicky zobrazuje ako 32 hexadecimálnych znakov rozdelených pomlčkami na skupiny vo formáte 8-4-4-4-12. Príklad štandardného UUID/GUID môže vyzerať takto: 123e4567-e89b-12d3-a456-426614174000. Unikátnosť týchto identifikátorov vyplýva z obrovskej možnosti kombinácií, kde teoretická pravdepodobnosť kolízie, teda, že dve náhodne vygenerované identifikátory by boli identické, je extrémne nízka (približne $2.94 * 10^{-39}$), nakoľko existuje 2^{128} = 340 282 366 920 938 463 463 374 607 431 768 211 456 možností. V softvérovom inžinierstve sa tieto identifikátory používajú pre identifikáciu objektov a informácií, ktoré potrebujú byť jednoznačne rozpoznateľné v rámci celej systémovej infraštruktúry, často cez viaceré databázy a aplikácie. Toto zahŕňa objekty ako sú používateľské účty, platby, transakcie a mnoho ďalších entít. Hlavnou výhodou je bezpečné generovanie jedinečných identifikátorov bez potreby centralizovanej kontroly, čo je ideálne pre moderné distribuované systémy. [9]

11.3 Realizácia synchronizácie siete

Na zabezpečenie efektívnej synchronizácie v našej decentralizovanej sieti sme vyvinuli nový typ správ, ktoré umožňujú uzlom vymieňať si informácie. Dve hlavné typy správ, ktoré používame, sú NODE_LIST_REQUEST a NODE_LIST_RESPONSE ktorá pozostáva z viacerých častí a končí identifikátorom označujúcim koniec správy.

Každý uzol v sieti udržiava dva typy pamäte pre uzly: krátkodobú a dlhodobú pamäť uzlov. Dlhodobá pamäť je perzistentná a uchováva informácie o všetkých uzloch, ktoré uzol kedy kontaktoval, zatiaľ čo krátkodobá pamäť obsahuje len aktuálne aktívne a dostupné uzly.

Synchronizácia začína, keď uzol rozpošle NODE_LIST_REQUEST všetkým uzlom, ktoré sú v jeho dlhodobej, no nie v krátkodobej pamäti. V tejto žiadosti poskytne informácie aj o sebe, aby mohli príjemcovia správy prípadne pridať uzol do svojej pamäte. Príklad správy NODE_LIST_REQUEST:

```
a.D||{
  "Id": "5f898416-ca1a-46ac-8437-921bf4dc928c",
  "Address": "192.168.94.132",
  "Port": 34259,
  "PublicKey":
"{\"Modulus\":\"zmYU3xWTWBMko70o8RpQ==\",\......"Exponent\":\"AQAB\"}"
}
```

Kde „a.D“ reprezentuje typ správy a zbytok sú informácie o samotnom uzle žiadateľa

Príjemcovia správy NODE_LIST_REQUEST vložia všetky svoje informácie o uzloch do správy NODE_LIST_RESPONSE a odošlú ju späť iniciátorovi, ten si pridá každý uzol, ktorý mu vrátil odpoveď do krátkodobej pamäte, pričom pokračuje v rozosielaní NODE_LIST_REQUEST do nových uzlov, o ktorých sa dozvedel z odpovedí. Tento proces pokračuje až kým nie sú skontaktované všetky uzly. Každý uzol, ktorý obdrží NODE_LIST_REQUEST, následne taktiež začne proces synchronizácie, čím sa docieli reťazová reakcia, ktorej výsledkom bude zosynchronizovaná celá sieť.

Príklad správy NODE_LIST_RESPONSE:

```
b.E||{
  "Id": " 7d0e921d-32c4-4532-b1d7-0ae3c5858ef3",
  "Address": "192.168.0.10",
  "Port": 34201,
  "PublicKey":
"{\"Modulus\":\"dasiFETF456EFse5ffsDFHH",\......"Exponent\":\"XZYA\"}"
}||e.G
```

Kde „b.E“ reprezentuje typ správy, zbytok je informácia o poskytovanom uzle a na konci je „e.G“, ktoré znamená, že toto je posledná časť správy typu „b.E“.

11.3.1 Ochrana pred zneužitím identity

Pri synchronizácii siete je kľúčové zabezpečiť, aby aktualizácia informácií uzlov prebiehala bezpečne a efektívne. Kritickým bodom je identifikácia a autorizácia uzlov, aby sme zabránili neoprávnenému prístupu a zneužitiu identít.

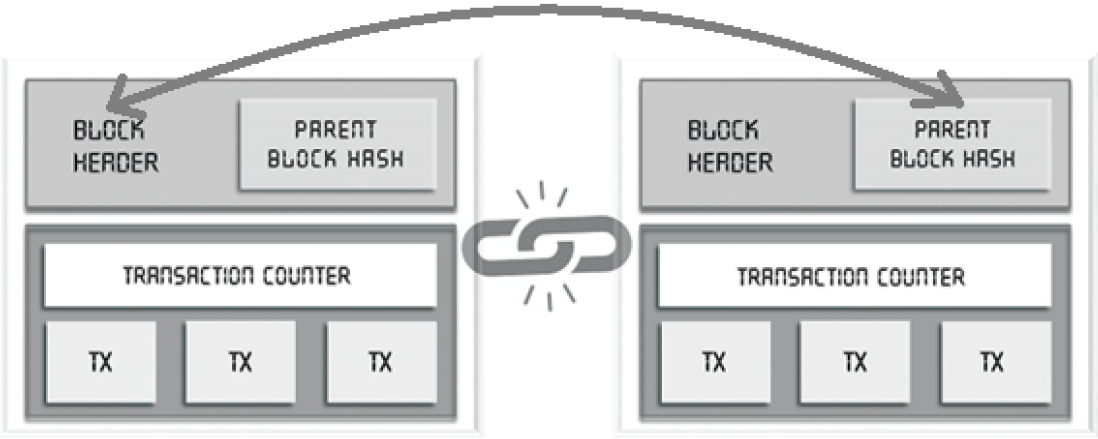
Bezproblémový prípad bude taký, kedy sa do siete pripojí uzol s GUID, ktoré sa v sieti už predtým nachádzalo a bude mať aj rovnaký verejný kľúč ako uzol, o ktorom informácie v sieti už sú. Pri takomto scenári sa uzlu jednoducho len aktualizujú položky adresy a portu, nakoľko vieme s určitosťou povedať, že je to rovnaký uzol, lebo má rovnaký verejný kľúč. Aj v prípade, že by svoj verejný kľúč len sfalšoval a poskytol sieti voľne dostupný verejný kľúč uzla, o ktorého identitu sa usiluje a v skutočnosti by nemal korešpondujúci certifikát so súkromným kľúčom, nijako by nemohol jednať v mene daného uzla, pretože všetky jeho požiadavky by boli zamietnuté sieťou, nakoľko by nedokázal vytvoriť taký podpis, ktorý by sa dal overiť daným verejným kľúčom.

Problémový prípad by bol, ak by sa uzol pokúsil pripojiť s už existujúcim GUID, ale s novým verejným kľúčom, takáto zmena musí byť zamietnutá. To je nevyhnutné na ochranu integrity siete, pretože nový verejný kľúč by umožnil potenciálne zneužitie identity pôvodného uzla. Teda uzol, ktorý stratí prístup k svojmu súkromnému kľúču (certifikátu), nebude môcť svoju identitu overiť a bude musieť začať ako nový uzol s novým GUID.

Tieto opatrenia zabezpečia, že každá zmena v sieti bude riadne overená a schválená, čím sa zvýši celková bezpečnosť a dôveryhodnosť nášho decentralizovaného systému.

11.4 Teória blockchainu – Formy a vývoj

Blockchain predstavuje distribuovanú databázovú technológiu, ktorá zapisuje informácie do blokov spojených do reťazca, čím zabezpečuje ich nemennosť a transparentnosť. Každý blok v reťazci obsahuje dáta o jednotlivých transakciách, kryptografický *hash* predchádzajúceho bloku, časovú pečiatku, čo zaisťuje chronologickú a nezmeniteľnú štruktúru a rôzne ďalšie údaje, ktoré závisia od konkrétnej implementácie a potrieb. [10]



Obr. 21. *Blockchain* bloky [10]

Štruktúra *blockchain*-u zaručuje výhody:

- Bezpečnosť, pretože dáta uložené v blokoch sú chránené kryptografiou a sú prakticky nemenné.
- Decentralizácia, na rozdiel od tradičných databáz, kde dáta kontroluje jedna entita, *blockchain* distribuuje údaje medzi všetky uzly siete, čo znižuje riziko manipulácie.
- Transparentnosť, všetky transakcie na *blockchain*-e sú viditeľné pre každého, kto má prístup k sieti. [10]

Blockchain teda slúži na bezpečné a decentralizované ukladanie záznamov v čase, ktoré sa využíva vo finančníctve, logistike, spravovaní digitálnych identít a ďalších oblastiach. Existuje niekoľko generácii blockchainu. Prvou je *Blockchain 1.0,* táto prvá generácia *blockchain*-u sa zameriava predovšetkým na kryptomeny, ako je Bitcoin. Uviedla koncept digitálnej meny, ktorá umožňuje transakcie bez potreby centrálnej autority, čím sa zásadne líši od tradičných platobných systémov. *Blockchain 2.0,* priniesol technológiu inteligentných kontraktov, ako sú tie, ktoré sa používajú na platforme *Ethereum*. Tieto inteligentné kontrakty umožňujú automatizované vykonávanie zmluvných podmienok bez zásahu vonkajších autorít, čo otvorilo dvere pre širokú škálu aplikácií nad rámec jednoduchých transakcií. *Blockchain 3.0* a jeho vyššie verzie priniesli použitia v mnohých ďalších sektoroch, ako zdravotníctvo, vzdelávanie a vládne systémy, kde zabezpečujú nielen transakcie, ale aj dáta s vysokou citlivosťou. [10]

11.5 Funkcionalita a štruktúra nášho blockchainu

Plán funkcionality našej decentralizovanej siete, spočíva v niekoľkých základný krokoch:

1. Uzol si kúpi, alebo dostane kredity k svojmu kontu pomocou bloku s transakciou ADD_CREDIT. Takýto blok mu bude schválny sieťou iba v prípade overenia správneho prepočtu novej aktuálnej hodnoty kreditov.
2. Uzol si zvolí súbor, ktorý si chce bezpečne uložiť na sieti a vytvorí blok s transakciou ADD_FILE_REQUEST, cena tejto transakcie bude priamo úmerná veľkosti zvoleného súboru. Takýto blok mu bude schválny sieťou iba v prípade overenia dostatočnej výšky kreditov a správneho prepočtu novej aktuálnej hodnoty kreditov.
3. Uzol, ktorý sa rozhodol zrušiť svoju požiadavku uloženia súboru sieťou, vytvorí blok s transakciou REMOVE_FILE_REQUEST. Takýto blok mu bude schválny sieťou iba v prípade overenia vlastníctva súboru a skutočnosti, že k danému súboru existuje blok s transakciou ADD_FILE_REQUEST, ku ktorému ešte nebol schválený blok s transakciou REMOVE_FILE_REQUEST.
4. Uzol, ktorý chce poskytnúť svoje zdroje k uloženiu súboru, vytvorí blok s transakciou ADD_FILE s informáciou konkrétneho súboru, ktorý chce uložiť. Odmenou mu bude množstvo kreditov, priamo úmerne veľkosti súboru. Takýto blok mu bude schválny sieťou iba v prípade overenia správneho prepočtu novej aktuálnej hodnoty kreditov a overenia skutočnosti, že daný uzol už nemá schválený blok s transakciou ADD_FILE ku korešpondujúcemu súboru (výnimkou je, ak medzi predchádzajúcou transakciou ADD_FILE a novou žiadosťou, bol uzlu schválny blok s transakciou REMOVE_FILE), taktiež dôjde k overeniu, či vlastník súboru už nemá schválený blok s transakciou REMOVE_FILE_REQUEST k danému súboru. Po schválení bloku ADD_FILE sa uzol pokúsi skontaktovať vlastníka súboru s úmyslom stiahnutia daného súboru, skontaktovaný uzol si v *blockchain*-e overí správnosť jeho kontaktovania a následne mu súbor poskytne. Uzol so schválenou transakciou ADD_FILE, taktiež bude v pravidelných intervaloch kontrolovať daný súbor na svojom úložisku, jednou z troch náhodne zvolených druhov kontroly. Možnosti kontroly budú nasledovné: kontrola prítomnosti súboru, kontrola prítomnosti + veľkosti súboru a kontrola prítomnosti + veľkosti + *hash*-u súboru. V prípade

nezrovnalostí, bude uzol kontaktovať vlastníka súboru s novou žiadosťou stiahnutia.

5. Ak uzol, ktorý má schválny blok s transakciou ADD_FILE zistil, že došlo k schváleniu bloku s transakciu REMOVE_FILE_REQUEST k korešpondujúcemu súboru, vymaže svoju kópiu súboru a vytvorí blok s transakciou REMOVE_FILE. To môže urobiť aj z vlastnej vôle, v prípade, že sa rozhodne už nevlastniť kópiu daného súboru. Blok s transakciou REMOVE_FILE mu bude schválny sieťou iba v prípade overenia, že jeho posledným blokom k tomuto súboru bol blok s transakciou ADD_FILE.

Podľa nášho plánu bude mať blok v *blockchain*-e túto štruktúru:

- index bloku [celé číslo]
- časová známka [typ *DateTime*]
- *hash* súboru [text – prázdny pri transakciách nesúvisiacich so súbormi]
- veľkosť súboru [celé číslo]
- id súboru [GUID]
- miesta kópii súboru [list s GUID uzlami, ktoré majú mať kópiu súboru]
- *hash* bloku [text]
- *hash* predchádzajúceho bloku [text]
- id uzla, ktorý blok vytvoril [GUID]
- zmena stavu kreditu [desatine číslo]
- nová aktuálna hodnota kreditu [desatine číslo]
- podpis bloku [text]

11.6 Konsenzus

Konsenzné mechanizmy sú základným kameňom *blockchain*-ových technológií, keďže umožňujú decentralizovaným sieťam dosiahnuť dohodu o aktuálnom stave *blockchain*-u bez potreby centrálnej autority. Tieto mechanizmy zabezpečujú, že všetky pridané transakcie sú platné a všetci účastníci siete majú konzistentné informácie. Niektoré konsenzné mechanizmy umožňujú sieti fungovať efektívne aj v prostredí, kde sa môžu nachádzať skomprimované a nefunkčne uzly. Každá transakcia je overená viacerými uzlami, čo zvyšuje transparentnosť a znižuje možnosti podvodu. V dnešnej dobe existuje veľké množstvo typov týchto mechanizmov, tu je niekoľko z nich:

- *Proof of Work* (POW): Tento mechanizmus je používaný napríklad *Bitcoinom*. Vyžaduje od ťažiarov vykonanie výpočtovo náročnej úlohy, čo zabezpečuje, že nové bloky sú pridávané do blockchainu správne a v časových intervaloch.
- *Proof of Stake* (POS): V tomto prípade účastníci siete blokujú časť svojich tokenov ako záruku za správne správanie. Čím viac tokenov je zablokovaných, tým vyššia je pravdepodobnosť, že uzol bude vybraný na vytvorenie nového bloku.
- *Delegated Proof of Stake* (DPOS): Variant POS, kde držitelia tokenov hlasujú za malý počet "delegátov", ktorí potom majú právo pridávať bloky do blockchainu.
- *Practical Byzantine Fault Tolerance* (PBFT): Zameriava sa na minimalizáciu vplyvu chybných uzlov a zabezpečuje, že všetky transakcie sú validované iba správnymi uzlami, čo výrazne zvyšuje odolnosť siete voči Byzantským chybám. Tento mechanizmus je vhodný pre aplikácie vyžadujúce rýchlu validáciu transakcií bez vysokých energetických nákladov typických pre POW.

11.6.1 Practical Byzantine Fault Tolerance

Pre naše požiadavky siete nám najviac vyhovuje PBFT, preto sa naň poďme bližšie pozrieť.

Internetové útoky a softvérové chyby sú čoraz bežnejšie. Rastúca závislosť priemyslu a vlád na online informačných službách robí tieto útoky atraktívnejšími a následky úspešných útokov vážnejšími. Okrem toho počet softvérových chýb narastá v dôsledku zvyšujúcej sa veľkosti a zložitosti softvéru. Keďže tieto útoky a softvérové chyby môžu spôsobiť, že chybné uzly prejavujú byzantské (t. j. nesprávne) správanie, algoritmy tolerantné voči byzantským chybám sú čoraz dôležitejšie. PBFT zabezpečuje funkčnosť a bezpečnosť za predpokladu, že maximálne jedna tretina z celkového počtu replík je súčasne chybná. V kontexte PBFT je replika chápaná ako jeden uzol. [11]

Existuje rozsiahly výskum v oblasti konsenzných dohôd a replikačných techník, ktoré tolerujú byzantské chyby. Avšak väčšina týchto techník, ktoré boli publikované pred PBFT sa zameriava na návrh a demonštráciu teoretickej uskutočniteľnosti, a sú príliš neefektívne na praktické použitie alebo predpokladajú synchrónnu komunikáciu a spoliehajú sa na známe limity oneskorení správ a rýchlosti procesov. Tam patria riešenia ako, *Rampart* a *SecureRing*, síce boli navrhnuté s cieľom byť praktické, ale ich správnosť závisí na predpoklade synchronizácie, čo je nebezpečné v prítomnosti útokov. Útočník môže ohroziť bezpečnosť služby tým, že spomalí funkčné uzly alebo komunikáciu medzi nimi, až kým nebudú označené za chybné a vylúčené zo skupiny replík. Takýto útok

odmietnutia služby je vo všeobecnosti jednoduchší ako získať kontrolu nad uzlom. PBFT nie je zraniteľné voči tomuto typu útokov, pretože práve kvôli bezpečnosti nespolieha na synchrónnu komunikáciu. [11]

Na ochranu pred falšovaním identity, a duplikovanými správami, a na detekciu skorumpovaných správ sa v PBFT využívajú kryptografické techniky. Správy obsahujú digitálne podpisy, kódy pre autentifikáciu správ a výstupy z *hash*-ovacích funkcií odolných proti kolíziám. [11]

V PBFT sa pripúšťajú útoky, ktoré môžu koordinovať chybné uzly, spôsobiť oneskorenia komunikácie alebo spomaliť funkčné uzly tak, aby spôsobili čo najväčšie škody na replikovanej službe. Predpokladá sa, že tieto útoky nemôžu správy funkčných uzlov spomaliť na nekonečne dlho. Tiež sa predpokladá, že útočník (a chybné uzly, ktoré kontroluje) sú obmedzené svojou výpočtovou kapacitou, takže (s veľmi vysokou pravdepodobnosťou) nedokáže obísť spomenuté kryptografické techniky. Napríklad, protivník nemôže vytvoriť platný podpis funkčného uzla, vypočítať informácie sumarizované *hash*-i, ani nájsť dve správy s rovnakým *hash*-om. [11]

11.6.1.1 Algoritmus PBFT

PBFT algoritmus funguje formou replikácie stavového automatu, ktorý je replikovaný naprieč rôznymi uzlami v distribuovanom systéme. Každá replika stavového automatu udržiava stav služby a implementuje operácie služby. Množinu replík označujeme $\mathcal{R}$ a každú repliku identifikujeme pomocou celého čísla v rozsahu $\{0, \dots, |\mathcal{R}| - 1\}$. Pre jednoduchosť predpokladáme, že $|\mathcal{R}| = 3f + 1$, kde f je maximálny počet replík, ktoré môžu byť chybné. Hoci môže existovať viac než $3f + 1$ replík. [11]

Repliky prechádzajú postupnosťou konfigurácií nazývaných pohľady (z angl. views). V každom pohľade je práve jedna replika primárnou a ostatné sú záložné. Pohľady sú číslované postupne. Primárnou replikou pohľadu sa stáva replika p taká, že $p = v \; mod |\mathcal{R}|$, kde v je číslo pohľadu. Algoritmus pripúšťa zmeny pohľadu v prípade, keď sa zdá, že primárna replika zlyhala. [11]

Kroky algoritmu:

- klient pošle požiadavku na vykonanie služby primárnej replike,
- primárna replika rozpošle požiadavku záložným replikám,
- repliky vykonajú požiadavku a pošlú odpoveď klientovi,
- klient čaká na $f + 1$ odpovedí od rôznych replík s rovnakým výsledkom.

Podobne ako všetky techniky replikácie stavového automatu, kladie tento algoritmus dve požiadavky na repliky: musia byť deterministické (t.j. vykonanie operácie v danom stave a so zadanou množinou argumentov musí vždy produkovať rovnaký výsledok) a musia začínať v rovnakom stave. Vzhľadom na tieto dve požiadavky algoritmus zaručuje vlastnosť bezpečnosti tým, že zabezpečí, aby sa všetky funkčné repliky dohodli na celkovom poradí pre vykonanie požiadaviek napriek zlyhaniam. Podrobný postup algoritmu je:

Kontaktovanie primárnej repliky prebieha nasledovne: Klient "c" pošle žiadosť o vykonanie operácie stavového automatu "o" odoslaním správy $\langle REQUEST, o, t, c\rangle_{\sigma_i}$ primárnej replike. Časová pečiatka "t" sa používa na zabezpečenie detekcie duplikácii. Pečiatky "c" reprezentuje poradie odoslaných správ tak, že neskoršie správy budú mať túto hodnotu vyššiu. X_{σ_i} reprezentuje podpísanie správy uzlom "i". [11]

Každá správa odoslaná replikami klientovi obsahuje aktuálne číslo pohľadu, čo umožňuje klientovi sledovať aktuálny pohľad a teda aktuálnu primárnu repliku. Klient odosiela požiadavku na repliku, o ktorej verí, že je aktuálne primárna replika v danom momente. Nasledujú tri fázy algoritmu: pred-príprava (*pre-prepare*), príprava (*prepare*) a potvrdenie (*commit*). [11]

Vo fáze pred-prípravy primárna replika priradí sekvencie číslo "n" požiadavke, rozošle správu pred-prípravy s pripevnenou požiadavkou "m" všetkým záložným replikám a pripojí správu do svojho záznamu. Správa má formu $\langle\langle PRE-PREPARE, v, n, d\rangle_{\sigma_i}, m\rangle$, kde "$v$" označuje pohľad, v ktorom sa správa odosiela, "m" je správa požiadavky klienta a d je *hash* správy "m". [11]

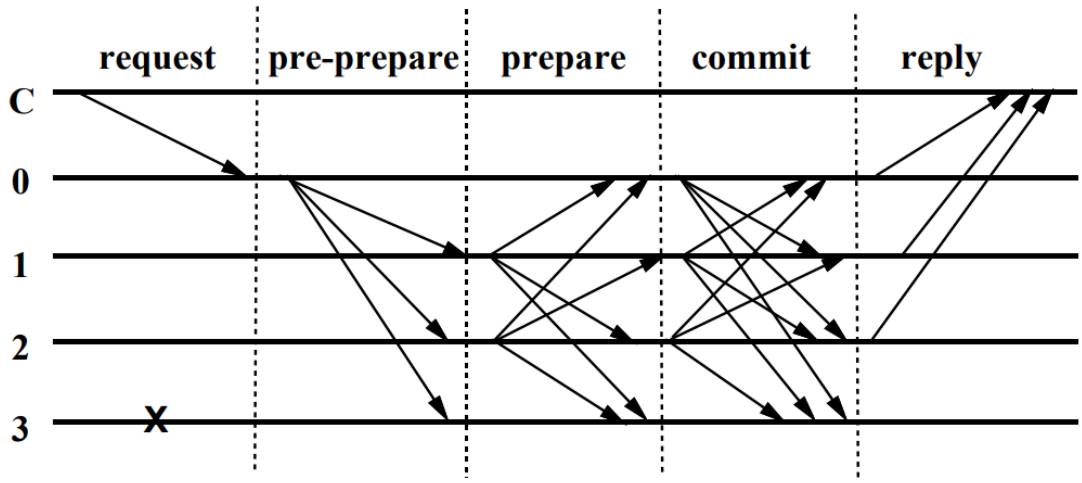
Záložný uzol prijme správu pred-prípravy za predpokladu, že:

- podpisy v požiadavke a správe pred-prípravy sú správne a "d" je *hash* pre "m",
- je v pohľade "v",
- neakceptoval správu pred-prípravy pre pohľad "v" a sekvencie číslo "n", obsahujúcu iný *hash*;
- sekvencie číslo v správe pred-prípravy je medzi nízkym prahovým číslom "h" a vysokým prahovým číslom "H".

Ak záložná replika "i" prijme správu $\langle\langle PRE-PREPARE, v, n, d\rangle_{\sigma_i}, m\rangle\rangle$, vstupuje do fázy prípravy (*prepare*) tým, že rozošle správu $\langle PREPARE, v, n, d, i\rangle_{\sigma_i}$ všetkým ostatným replikám a obe správy pridá do svojho záznamu. V prípade neprijatie nič nerobí.

Replika (vrátane primárnej) prijíma prípravné (*prepare*) správy a pridáva ich do svojho záznamu za predpokladu, že podpisy sú správne, ich číslo pohľadu zodpovedá aktuálnemu pohľadu repliky a ich sekvencie číslo je medzi dolnou hranicou "h" a hornou hranicou "H". Predikát $prepared(m, v, n, i)$ definujeme ako pravdivý práve vtedy, ak replika "i" vložila do svojho zoznamu požiadavku "m", pred-prípravnú (*pre-prepare*) správu pre "m" v pohľade "v" so sekvencie číslom "n" a $2f$ prípravných (*prepare*) správ od rôznych záložných replík, ktoré zodpovedajú pred-prípravnej (*pre-prepare*). Repliky overujú, či prípravné (*prepare*) správy zodpovedajú pred-prípravným (*pre-prepare*) správam tým, že kontrolujú, či majú rovnaké číslo pohľadu, sekvenčné číslo a *hash*. [11]

Replika "i" odosiela správu $\langle COMMIT, v, n, D(m), i \rangle_{\sigma_i}$ ostatným replikám, keď $prepared(m, v, n, i)$ vyhodnotí pravdivo, kde $D(m)$ je *hash* požiadavky "m". To spúšťa fázu potvrdenia (commit). Repliky prijímajú potvrdzovacie (*commit*) správy a vkladajú ich do svojho záznamov za predpokladu, že sú správne podpísané, ich číslo pohľadu zodpovedá aktuálnemu pohľadu repliky a ich sekvencie číslo je medzi dolnou hranicou "h" a hornou hranicou "H". Každá replika uskutoční požiadavku "m", ked prijme $2f + 1$ potvrdení (*commit*). [11]



Obr. 22. Priebeh PBFT konsenzu [11]

Originálne PBFT obsahuje aj veľké množstvo ďalších vlastností a typov správ, ako napríklad $\langle VIEW - CHANGE, v + 1, \dots \rangle_{\sigma_i}$ pri zmene primárnej repliky, ak je podozrenie, že by mohla byť skomprimovaná a podobne. No pre našu implementáciu nie sú potrebné, nakoľko my si tento algoritmus upravíme tak, aby vyhovoval našej aplikácii.

11.6.2 Naša implementácia PBFT

Hlavnou zmenou našej implementácie oproti originálnemu PBFT bude vynechanie funkcionality pohľadu (*view*). Ako sme si vysvetlili v kapitole 11.6.1.1, úlohou pohľadu je, aby pozícia primárnej repliky rotovala v sieti, čo jej zabezpečí bezpečnosť, nakoľko sa útočník nebude môcť zamerať iba na jeden uzol. Nevýhodou implementácie pohľadu je veľmi jednoduchá predvídateľnosť toho, kto bude nasledujúca primárna replika, a teda útočník sa na to môže dopredu pripraviť. Medzi nevýhody by sme taktiež uviedli komplikované implementovanie, nakoľko nejde len o samotné posielanie pohľadu v správach, ale sieť musí vedieť vyriešiť situácie skomprimovanej primárnej repliky, zhodnúť sa na určitých záveroch voči nej, rozposielať správy ako: *view-change* a *new-view*.

Na to, aby sme sa vedeli tomuto vyhnúť, prišli sme s implementáciou, kde bude rotácia primárnej repliky nepredvídateľná a zároveň deterministická. To docielime tak, že na výber primárnej repliky sa vygenerujeme pre každú repliku z krátkodobej pamäti novy *hash*, ten bude pozostávať z *hash*-u z posledného bloku, GUID-u konkrétnej repliky a aktuálneho času v ktorom budú obsiahnuté aj stotiny. Potom zoberieme tieto vygenerované *hash*-e a ten, ktorého hodnota bude abecedne najnižšia sa stane primárnou replikou. Týmto docielime, že každá žiadosť bude v tomto ohľade jedinečná a overiteľná. Taktiež nebude potrebná implementácia zmeny primárnej repliky tak ako je to v originálnom algoritme, pretože ak si žiadateľ bude myslieť, že jeho žiadosť bola správna a bola neprávom odmietnutá primárnou replikou, môže ju jednoducho poslať znova a overí mu ju nová primárna replika.

Do našej verzie algoritmu sme ešte pridali položku, ktorá hovorí o aktuálnej štruktúre siete, ide znova o *hash*, ktorý generujeme tak, že zoradíme všetky uzly z krátkodobej pamäti a z ich spojených GUID vygenerujeme tento *hash*. V prípade, že by akejkoľvek replike prišla správa, v ktorej by bol tento *hash* iný ako jej vlastný, ktorý si vygeneruje, jej reakcia bude závisieť od typu prijatej správy, v prípade *request* správy vráti *error* správu s textovým obsahom *error*-u, v prípade ostatných správ ich úplne odignoruje.

Záznamy o prijatých a odoslaných správach ako aj samotné bloky *blockchain*-u budeme ukladať v *SQLite* databáz aplikácie.

11.6.3 Vizuálna stránka decentralizovanej časti aplikácie

Poďme sa pozrieť, ako sme vyriešili GUI decentralizovanej časti našej aplikácie. Najprv sme si vytvorili kartu uzlov.

11.6.3.1 Karta uzlov

Na karte uzlov, bude môcť užívateľ spravovať svoj zoznam uzlov a ich synchronizáciu.

Obr. 23. Klientska karta uzlov

Popis k obrázku 23.

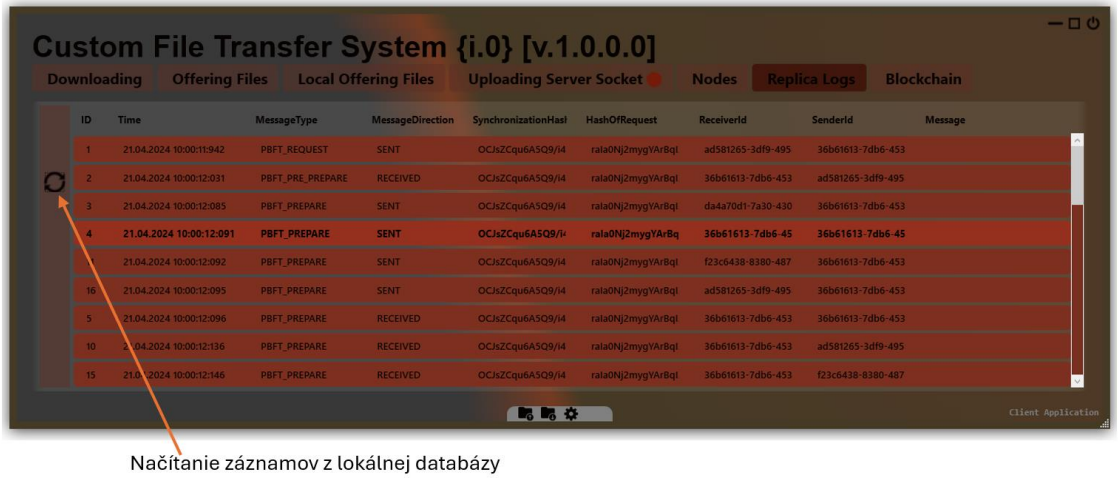
1. Štart synchronizácie.
2. Načítanie uzlov z lokálnej dlhodobej pamäte a označenie „Active“ zelenou pre uzly nachádzajúce sa aj v krátkodobej pamäti.
3. Konfigurácia svojho vlastného uzla.

Obr. 24. Konfigurácia vlastného uzla

4. Manuálne pridanie nového uzla do dlhodobej pamäti pomocou adresy a portu.
5. „Active“ označenie uzlov z krátkodobej pamäte.
6. Možnosť manuálnej úpravy uzlu (iba adresy a portu).
7. Zmazanie uzlu z dlhodobej pamäti.

11.6.3.2 Karta záznamov repliky

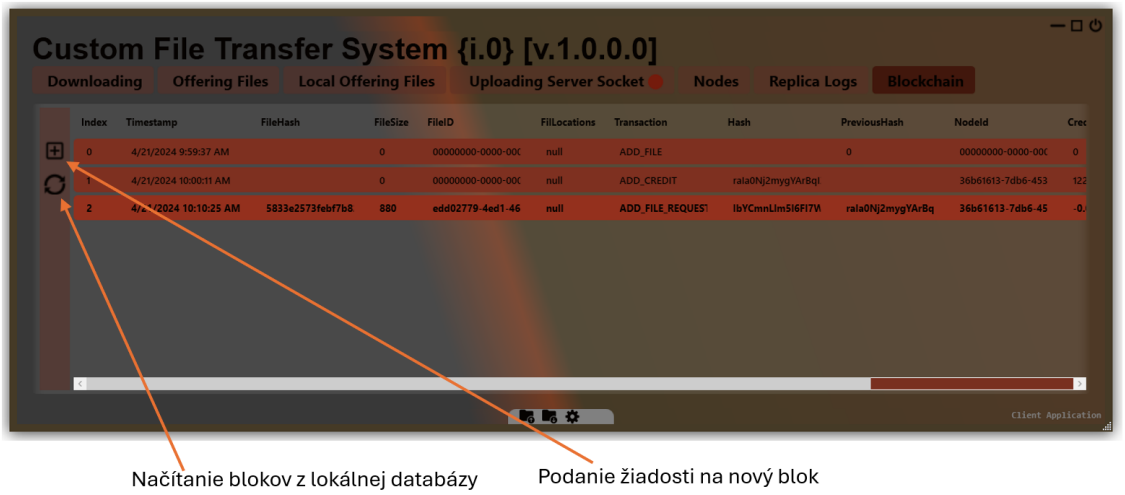
Karta záznamov replík bude slúžiť iba čisto pre informačne účely, užívateľ si na tejto karte bude môcť pozrieť aké správy jeho replika v kontexte konsenzu prijala a aké odoslala.



Obr. 25 Klientska karta záznamov repliky

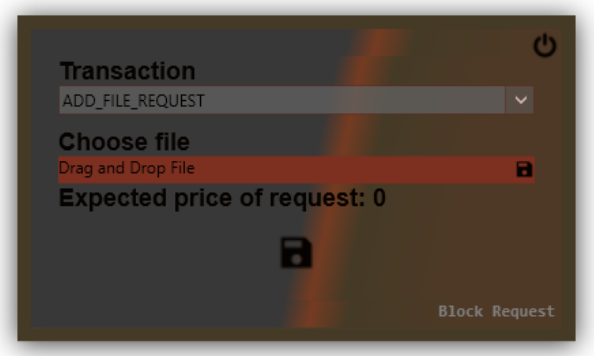
11.6.3.3 Karta blockchainu

Na tejto karte si bude môcť užívateľ zobraziť jednotlivé bloky *blockchain*-u a taktiež podávať nové žiadosti blokov na sieť.



Obr. 26. Klientska karta *blockchain*-u

Pri podávaní novej žiadosti, aplikácia automaticky mení formulár tvorby bloku podľa aktuálnych možnosti, pri bloku s transakciou ADD_FILE aplikácie poskytne všetky možné GUID súborov, ktoré sú pre túto možnosť dostupné, rovnako tak pri transakciách ADD_REQUEST a ADD_REMOVE_REQUEST. Pri bloku s transakciou ADD_FILE_REQUEST dostane užívateľ možnosti zvoliť si svoj súbor pretiahnutím do aplikácie okna alebo dialógom na prehľadávanie súborov.



Obr. 27. Dialóg na tvorbu žiadosti pre nový blok s transakciou ADD_FILE_REQUEST

12 Testovanie decentralizovanej časti

Testovanie decentralizovanej časti našej aplikácie, teda blockchainu a konsenzuálneho mechanizmu, predstavuje dôležitý krok v zabezpečení stability a spoľahlivosti systému. Ako podstatná súčasť vývojového cyklu, testovanie nám umožňuje identifikovať a opraviť problémy skôr, než sa aplikácia dostane do produkčného prostredia.

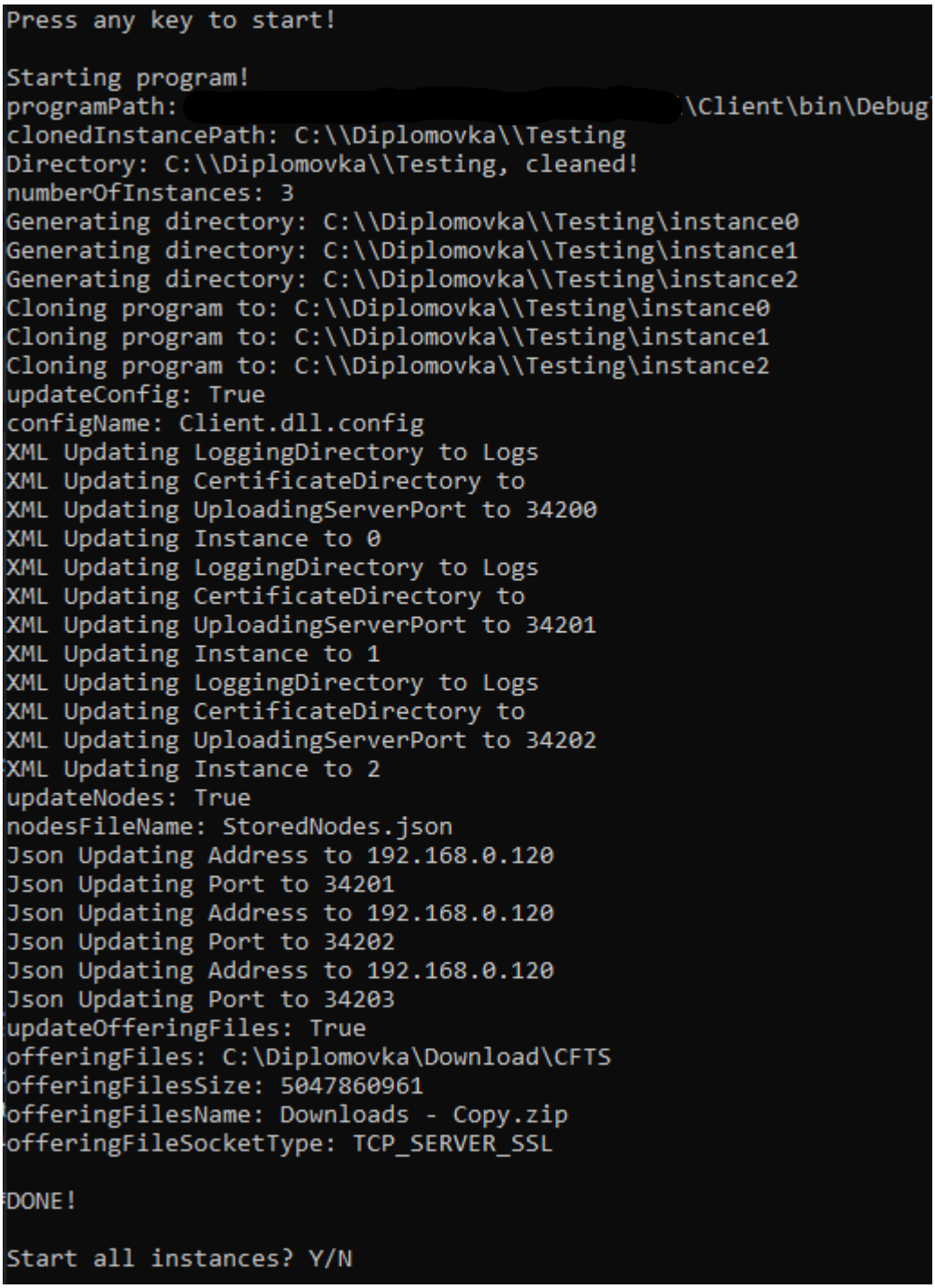
Za týmto účelom si vytvoríme novú aplikácie, ktorá nám umožní vytvoriť a nastaviť viacero inštancií nášho programu. Tieto inštancie budú ihneď po spustení schopné medzi sebou vykonať synchronizáciu siete. Tento prístup nám umožní simulovať reálnu sieťovú infraštruktúru a testovať, ako sa naša aplikácia správa v distribuovanom prostredí.

Každá vzniknutá inštancia bude plnohodnotne reprezentovať funkcionalitu uzla v sieti, čo zahŕňa spracovanie transakcií, udržiavanie konzistencie dát a reakciu na zmeny v sieti.

Testovaciu aplikáciu nastavíme tak, aby používala náš konfiguračný nástroj a následnú konfiguráciu:

```
<add key="programPath" value="C:\Client\bin\Debug"/>

<add key="clonedInstancePath" value="C:\\Diplomovka\\Testing"/>
<add key="numberOfInstances" value="10"/>

<add key="updateConfig" value="true"/>
<add key="configName" value="Client.dll.config"/>

<add key="updateNodes" value="true"/>
<add key="nodesFileName" value="StoredNodes.json"/>

<add key="updateOfferingFiles" value="true"/>
<add key="offeringFiles" value="C:\Diplomovka\Download\CFTS"/>
<add key="offeringFilesName" value="Downloads - Copy.zip"/>
<add key="offeringFilesSize" value="5047860961"/>
<add key="offeringFileSocketType" value="1"/>
```

Tento program bude konzolová aplikácia a po jej spustení môžeme pozorovať výsledok ako na obrázku 28.



Obr. 28. Testovací projekt

13 Inštalácia programu

Je dôležité, čo najviac minimalizovať problémy a nezrovnalosti pri inštalácii programu. Preto sme k obom našim aplikáciám vytvorili „setup“ projekty.

„Setup“ projekty budú základným nástrojom pre distribúciu našej aplikácie. Ich účelom bude zjednodušiť proces inštalácie pre koncového používateľa tým, že poskytnú všetky potrebné súbory, závislosti a konfigurácie v jednom balíku. Tieto projekty zabezpečia, že aplikácia bude nainštalovaná správne, vrátane správneho nastavenia databáz, knižníc a ďalších služieb potrebných pre jej beh.

Záver

Hlavným cieľom tejto diplomovej práce bolo vytvoriť efektívny softvér na prenos súborov, ktorý by vyhovoval súčasným potrebám v oblasti bezpečnosti a operatívnosti. V rámci tohto cieľa sme preskúmali dva rozdielne prístupy: centralizovaný a decentralizovaný. Tieto prístupy boli implementované ako samostatné funkcionality, čím sme umožnili detailné hodnotenie a porovnanie každého systému z hľadiska jeho technických a praktických vlastností.

Centralizovaný model bol navrhnutý na prenášanie súborov medzi klientami pomocou súborových identifikátorov s ohľadom na jednoduchosť používania a efektivitu správy, zatiaľ čo decentralizovaný model využíva technológiu blockchain a nahrávania dát na sieť tak, aby poskytol vysokú úroveň bezpečnosti a odolnosť voči externým zásahom. Táto nezávislá implementácia nám umožnila podrobne analyzovať každý prístup zvlášť a zhodnotiť ich prínosy a možné nevýhody pri aplikáciách v reálnom svete. Proces vývoja a testovania bol plný technických a konceptuálnych výziev, ktoré sme museli prekonať. Tieto skúsenosti nám však poskytli hlboké pochopenie praktickej realizácie komplexných informačných systémov a zároveň ukázali význam pevného plánovania a adaptability. Pri implementácii decentralizovaného modelu sme čelili špecifickým bezpečnostným výzvam, ktoré vyžadovali inovatívne prístupy, šifrovanie a overovanie transakcií.

Vytvorený softvér úspešne splnil hlavný cieľ práce a je schopný efektívne riešiť potreby užívateľov pri prenose súborov. Implementácie v tejto práci môžu poskytnúť cenné informácie pre ďalší vývoj a výskum v oblasti technológií prenosu súborov.

Literatúra

[1] GAURAV, S. 2023, Announcing .NET 8 In *.NET Blog* [online]. 2023, [cit. 2023-12-30]. Dostupné na internete:< https://devblogs.microsoft.com/dotnet/announcing-dotnet-8/>.

[2] STEPHEN, T. 2023, Performance Improvements in .NET 8. In *NET Blog*, [online]. 2023, [cit. 2023-12-30]. Dostupné na internete:< https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-8/>.

[3] ROBERT, C. M. – MICAH, M. 2006, *Agile Principles, Patterns, and Practices in C# 1st Edition.* Pearson, 2006. ISBN 978-0131857254

[4] Establishing a SSL/TLS Session. In *okta Developer* [online]. 2023, [cit. 2023-12-30]. Dostupné na internete: <https://developer.okta.com/books/api-security/tls/how/#tls-how/>

[5] SHAZIA, R. – SHAFIA. ASMA, S. – MADIHA, K. 2014, Performance Analysis of SSL/TLS in *International Journal of Innovative Science, Engineering & Technology*. ISSN 2348 – 7968, 2014, vol 1, p. 524-531.

[6] Network Security – Transport Layer in *Tutorialspoint* [online]. 2024, [cit. 2024-1-12]. Dostupné na internete:< https://www.tutorialspoint.com/network_security/network_security_transport_layer.htm/>

[7] KREIBICH, A. J. 2010, *Using SQLite*. Sebastopol : O'Reilly Media, 2010. ISBN 987-0-596-52118-9, p. 51-52

[8] Dapper - a simple object mapper for .Net. In *GitHub* [online]. 2024, [cit. 2024-4-8]. Dostupné na internete:< https://github.com/DapperLib/Dapper/>

[9] DIVINE, P. 2021, What is a UUID (or GUID)? In *Medium* [online] 2021, [cit 2024-4-13]. Dostupné na internete:<https://pddivine.medium.com/what-is-a-uuid-or-guid-16d0ead25008/>

[10] RISHABH, G. 2023, *Blockchain for Real World Applications,* Hoboken: John Willey & Sons, 2023. ISBN 9781119903734

[11] CASTRO, M – LISKOV, B. 1999, Practical Byzantine Fault Tolerance in *Proceedings of the third symposium on Operating systems design and*

implementation, USENIX Association : United States, ISBN 978-1-880446-39-3, p. 173 – 186.

Prílohy

Príloha A: obsah CD

V priečinku *Diplomová práca* nájdeme sedem pod priečinkov:

• *Centrálny server* – v tomto priečinku sa nachádza aplikácia centrálneho servera pre platformu windows.

• *Centrálny server Setup* – v tomto priečinku vieme nájsť inštalačný súbor pre aplikáciu centrálneho servera.

• *Client* - v tomto priečinku sa nachádza klientska aplikácia pre platformu windows.

• *Client Setup* – v tomto priečinku vieme nájsť inštalačný súbor pre klientsku aplikáciu.

• *Prázdne databázy* – Kópia databázy centrálneho servera pre „Offering files“ a kópia databázy klienta pre *blockchain* a PBFT záznamy.

• *Testovacia aplikácia* – Testovacia aplikácia z kapitoly 12.

• *Zdrojové kódy* – Visual studio projekt s kompletnými zdrojovými kódmi k projektom: CentralServer, Client, Common, ConfigManager, ConsoleTests(Testovacia aplikácia), Logger, SqliteClassLibrary, SslTcpSession, TcpSession, CentralServerSetup, ClientSetup. Tento priečinok taktiež obsahuje súbor: *Dokumentácia.docx*, kde môžeme nájsť dokumentáciu k zdrojovým kódom.