**Szegedi Tudományegyetem**

**Informatikai Intézet**

# Videóchat alkalmazás fejlesztése C# alapokon

Szakdolgozat

Készítette:
**Bán János**
programtervező informatikus BSc szakos hallgató

Témavezető:
**Dr. Alexin Zoltán**
adjunktus

Szeged

2024

**Feladatkiírás**

Név: Bán János

Neptun azonosító: SOD4D0

Szak: Programtervező Informatikus Bsc. nappali tagozat

Témavezető: Dr. Alexin Zoltán

Tanszék: Szoftverfejlesztés

Angol cím: Video chat application development based on C#

Téma: Videóchat alkalmazás fejlesztése C# alapokon

Az elmúlt évtized során a videóchat alkalmazások rohamos fejlődésen mentek keresztül, az online kommunikáció egyre fontosabbá vált. A TeamSpeak, a Skype vagy a Discord egy-egy ismert példa ilyen alkalmazásokra. Az emberek könnyen és kényelmesen kommunikálhatnak a családtagokkal, barátaikkal vagy munkatársaikkal. Az elkészülő alkalmazás felhasználóbarát és intuitív felhasználói felületet kínál, ahol a felhasználók könnyedén indíthatnak videóhívásokat egy másik személy felé. Egy alkalmazás egyszerre két személy interaktív beszélgetését biztosítja. Lehetőséget fog biztosítani a videó- és a hangminőség beállítására, valamint felvételre és lejátszásra is. A beállítások menüben a felhasználó ki tudja választani az input és az output eszközöket, ha több elérhető eszköz is van. A programot később tovább is lehet fejleszteni kisebb beépülő modulokkal. Például lehetne benne többféle színbeállítás, illetve mesterséges hátteret lehetne beállítani, egy jpeg (bmp) képet, amivel divatosabbá lehetne tenni a felületet. A mesterséges háttér alkalmazása szükségessé teszi online képfelismerő algoritmus alkalmazását a videóchat alkalmazásban, ami a beszélő sziluettjét felismeri, és háttér előtt a beszélőt jeleníti meg. A projektet C# programozási nyelven készíteném el az AForge.Video nyílt forráskódú USB webkamera kezelő könyvtár segítségével. A projektben a kommunikációt UDP protokollal akarom megvalósítani hiszen a két félnek valahogy el kell érnie egymást. Emellett fontosak lesznek a portok, hiszen ezen keresztül fogadom és küldöm az adatokat. A tűzfalak miatt esélyes hogy egy kis NAT(Network Address Translation) traverzálást is be kell iktatnom hogy el tudjam érni esetlegesen a másik számítógépet. A szakdolgozat tartalmazni fog egy mesterséges háttér algoritmust. Az alábbi ütemterv szerint készítem el.

| Ütemterv |
| --- |

| Időintervallum | Tevékenység |
|---|---|
| 2023. Szeptember 01. – 2023. Szeptember 30. | Szakirodalmak elemzése, kurzusok megtekintése, példák készítése |
| 2023. Október 01. – 2023. Október 30 | A videóchat alkalmazás alapjának kidolgozása és tervezése |
| 2023. November 01. – 2023. November 02. | Projekt konfigurálása és környezet beállítása |
| 2023. November 01. – 2023. November 02. | Felhasználói felület tervezése és implementálása |
| 2023. December 01. – 2023.December 30 | Videó- és hangfunkciók implementálása |
| 2024. Január 01. – 2023. Január 31. | Arcfelismerés implementálása |
| 2024. Február 01. – 2024. Február 29 | Hibajavítás és tesztelés |

| | |
|---|---|
| 2024. Március 01. –<br>2024. Május 01. | Dokumentáció Írása |

**Tartalmi összefoglaló**

- **A téma megnevezése:**

  Videóchat alkalmazás fejlesztése C# alapokon

- **A megadott feladat megfogalmazása:**

  A feladat egy peer-to-peer videóchat alkalmazás készítése, ahol a felhasználók közvetlenül egymással kommunikálhatnak videóhívások útján. Az alkalmazás lehetővé teszi a videóhívások indítását és fogadását, valamint egy egyszerű chat felületet is biztosít a felhasználók számára.

- **A megoldási mód:**

  Az alkalmazás fejlesztése során a C# nyelvet és a .NET keretrendszert használom, valamint a peer-to-peer kommunikációt megvalósító technológiát integrálom a projektbe.

- **Alkalmazott eszközök, módszerek:**

  A fejlesztés során a Visual Studio fejlesztői környezet és a .NET keretrendszer mellett a WPF (Windows Presentation Foundation) keretrendszert használom. A kommunikáció megvalósításához a .NET Core vagy a .NET Framework beépített funkcióit használom, és szükség esetén kiegészítem azokat további könyvtárakkal. A képfeldolgozáshoz az AForge keretrendszert, valamint a hangkezeléshez az NAudio eszközt alkalmazom.

- **Elért eredmények:**

  Az alkalmazás sikeresen elkészült, és számos új funkcióval lett kiegészítve. A videóchat mellett hozzáadtam egy szöveges chat funkciót is, ami lehetővé teszi a felhasználók számára a szöveges üzenetküldést. Emellett a videóchatnél implementáltam egy képkimentési funkciót is, amely lehetővé teszi a felhasználók számára, hogy a beszélgetés során képeket készítsenek és elmentsék azokat. Ezek az új funkciók tovább javítják az alkalmazás funkcionalitását és növelik a felhasználói élményt.

- **Kulcsszavak:**

  AForge, NAudio, WPF, videóchat, C#

# Tartalomjegyzék

# BEVEZETÉS

Az információs technológia fejlődése során az online kommunikáció egyre inkább elterjedté válik, és a videóchat alkalmazások különösen fontos szerepet töltenek be a távoli kommunikációban. Ebben a digitális környezetben az egyszerűen kezelhető és megbízható videóchat platformok nagy értéket képviselnek a felhasználók számára, legyen szó családi, baráti vagy üzleti kapcsolatokról.

A jelen szakdolgozat célja egy olyan videóchat alkalmazás fejlesztése, amely hatékony, megbízható és felhasználóbarát. Az alkalmazás kialakításának motivációja az online kommunikáció során szerzett tapasztalatokból származik, amelyek rávilágítottak az egyszerű és hatékony videóchat megoldások iránti nagy igényre.

Bár számos videóchat alkalmazás elérhető a piacon, sok felhasználó még mindig keresi az olyan platformokat, amelyek könnyen kezelhetőek és megbízhatóak mind a személyes, mind a munkahelyi kommunikáció során. Ebben a kontextusban a szakdolgozat célja egy olyan videóchat alkalmazás létrehozása, amely egyszerűen kezelhető, megbízható és hatékony.

A videóchat alkalmazás fejlesztéséhez a Windows Presentation Foundation (WPF) keretrendszert választottam, amely egy megfelelő eszköz a Windows alapú asztali alkalmazások készítéséhez. A C# programozási nyelvet alkalmaztam, mivel ez a nyelv jól integrálható a WPF keretrendszerrel, és lehetővé teszi a hatékony és rugalmas fejlesztést az alkalmazás funkcionalitásának megvalósításához. Emellett szerettem volna elmélyülni jobban a C# programozási nyelv mélységeiben és minél többet tanulni a nyelvről.

A videóchat alkalmazás fejlesztése során először több alapvető funkciót teszteltem és implementáltam, beleértve a videó- és hangkapcsolat létrehozását, a kliens és szerver közötti kommunikációt, magát a szöveges kommunikációt, a felhasználói felület tervezését és az alapvető kommunikációs protokollok integrálását, ebbe kettő port kezelés tartozik, a TCP és az UDP. Ezek az előzetes lépések segítettek megismerni a WPF keretrendszert és a videóchat alkalmazások különféle technikáit, így elkezdhettem a nehezebb részek implementálását.

# SZOFTVERTERVEZÉS

Az alkalmazásom két fő komponensből áll: a Server és a Kliens. A szerver biztosítja a kommunikációt, kezeli az üzenetváltásokat és a felhasználókat, míg a kliens lehetővé teszi a felhasználók számára a szöveges és hang alapú kommunikációt.

## Elvárt funkciók

**Felhasználói Regisztráció és Bejelentkezés**: Az alkalmazásnak lehetőséget kell biztosítania a felhasználók számára, hogy regisztráljanak és bejelentkezzenek. Az adatokat biztonságosan kell tárolni és kezelni.
**Üzenetváltás**: A felhasználóknak lehetőséget kell teremteni arra, hogy szöveges üzeneteket tudjanak küldeni és fogadni. Az üzeneteket valós időben kell továbbítani a szerveren keresztül.
**Hangüzenet Küldése és Fogadása**: Az alkalmazásnak támogatnia kell a hangüzenetek küldését és fogadását is.
**Adattárolás**: Az üzeneteket és a felhasználói adatokat tartósan kell tárolni, hogy újraindítás esetén is elérhetők legyenek
**Videó Kommunikáció**: A szerveren a felhasználók egymás videókamera képét kell, hogy lássák.

## Megvalósítás módja

A felhasználói regisztráció és bejelentkezés az AES (Advanced Encryption Standard) titkosítás használatával valósul meg, amely biztonságos tárolást biztosít a felhasználói adatok számára. Az AES titkosítja a felhasználónév és jelszót, így csak az alkalmazás megfelelő működéséhez szükséges esetekben kerülnek elküldésre a szervernek.

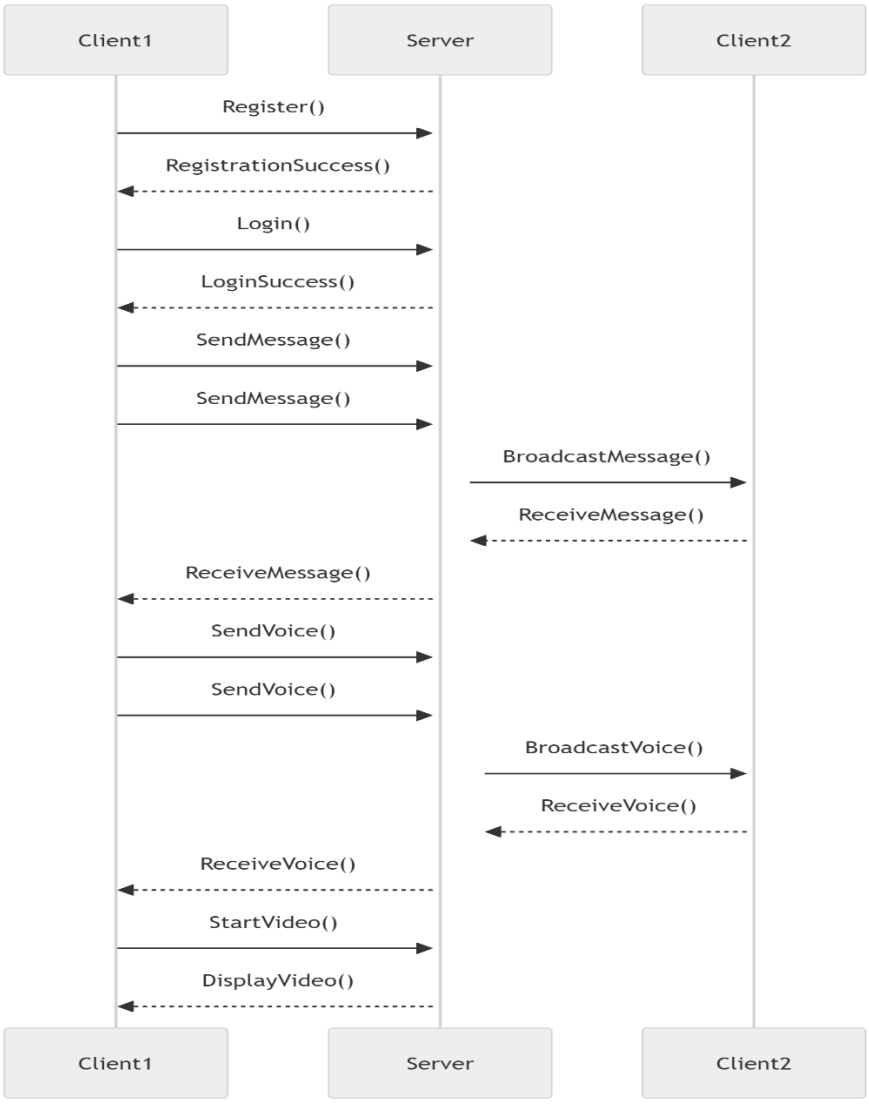
Az üzenetküldést és fogadást a szerver központi komponense biztosítja. A SendMessage metódus továbbítja az üzenetet minden kapcsolódott felhasználónak. Az alkalmazás az üzeneteket byte tömbbé alakítja és UTF-8 kódolással rögzíti, majd továbbítja.

A hangüzenetek kezeléséhez az NAudio könyvtárat használom, amely lehetővé

teszi a hangok felvételét, lejátszását és továbbítását. Az NAudio segítségével a kliens felvételezi a hangüzenetet, majd azt byte tömbként küldi el a szervernek, amely továbbítja a címzett felhasználónak.

Az adatokat XML formátumban tárolom, hogy újraindítás esetén is elérhetők legyenek, lényegében ez az adatbázisom. A SaveUsers() és SaveServers() metódusok felelősek az adatok mentéséért, míg a LoadUsers() és LoadServers() metódusok az adatok visszatöltéséért. Ezek a metódusok az XML serializer és deserializer objektumokat használják az adatok strukturált mentésére és visszatöltésére.

A videó kommunikáció terén az alkalmazás jelenleg csak a felhasználó saját kameraképének megjelenítésére képes. A képek mentéséhez az AForge keretrendszert használom, amely lehetővé teszi a videó stream feldolgozását és a képek rögzítését.



A diagram egy kliens-szerver architektúrát ábrázol, amely két fő komponensből áll: A kliens és a szerver közötti kommunikáció a következőképpen történik:

1. A felhasználó bejelentkezik a kliensbe.
2. A kliens elküldi a bejelentkezési adatokat a szervernek.
3. A szerver ellenőrzi a bejelentkezési adatokat.
4. Ha a bejelentkezési adatok helyesek, a szerver bejelentkezteti a felhasználót, és elküldi a felhasználói adatokat a kliensnek.
5. A felhasználó elkezdhet kommunikálni más felhasználókkal.
6. Amikor a felhasználó üzenetet küld, a kliens elküldi az üzenetet a szervernek.
7. A szerver továbbítja az üzenetet a címzett felhasználó kliensének.
8. Amikor a felhasználó fogad egy üzenetet, a kliens megjeleníti az üzenetet a felhasználónak.
9. Amikor a felhasználó hangüzenetet küld, a kliens felveszi a hangüzenetet, és elküldi azt a szervernek.
10. A szerver továbbítja a hangüzenetet a címzett felhasználó kliensének.
11. Amikor a felhasználó fogad egy hangüzenetet, a kliens lejátssza a hangüzenetet a felhasználónak.
12. Amikor a felhasználó videohívást kezdeményez, a kliens elküldi a híváskérést a szervernek.
13. A szerver továbbítja a híváskérést a címzett felhasználó kliensének (ami jelen eseteben önmaga)

# 1. FELHASZNÁLT ESZKÖZÖK

A program megvalósításához alapvetően egyetlen egy dolgot használtam, méghozzá a C# programozási nyelvet a Visual Studio 2022 felületével. Mivel céljaim közt több magasabb szintű funkció megvalósítása is szerepelt, ezért több alapvetőnek nem mondható csomagot és eszköz használatát vettem igénybe. Ezen eszközök és csomagok alkalmazása lehetővé tette, hogy a C# nyújtotta lehetőségeket mélyebben kihasználjam, és így gyorsítsam munkámat, aminek eredményeképpen magasabb színvonalú alkalmazást hozhattam létre. A Visual Studio 2022 lehetőséget biztosított a projektek

könnyű kezelésére és menedzselésére, valamint támogatta a kód szerkezetének és formázásának ellenőrzését, ami hozzájárult a kód minőségének javításához és az esetleges hibák korai felfedezéséhez. // belerakni a következő fejezetbe

## 1.1. Visual Studio 2022

A Visual Studio eredetileg a C# fejlesztés támogatására jött létre, és azóta is az egyik legnépszerűbb és legtöbbet használt fejlesztői környezet a C# programozási nyelvvel való alkalmazásfejlesztéshez. A Microsoft a C#-ra és a .NET keretrendszerre összpontosítva hozta létre a Visual Studio-t, és folyamatosan fejleszti annak funkcionalitását, hogy még hatékonyabb és produktívabb eszközt biztosítson a fejlesztők számára.
A Visual Studio 2022 előnyei közé tartozik a különböző programozási nyelvek széles körű támogatása. A C# fejlesztők előszeretettel választják a Visual Studio-t annak felhasználóbarát felülete és gazdag funkciókészlete miatt. A támogatás kiterjed a .NET keretrendszer más részeire is, beleértve az ASP.NET, Xamarin és Unity fejlesztést.
Szorosan integrálódik más Microsoft szolgáltatásokkal és platformokkal, mint például az Azure és a GitHub, ami további előnyöket nyújt az alkalmazásfejlesztés során.

## 1.2. C# Programozási Nyelv

A C# (C Sharp) egy modern, objektumorientált programozási nyelv, amelyet általában a Microsoft .NET keretrendszerrel együtt használnak alkalmazásfejlesztéshez. A C# nyelvet kifejezetten a Microsoft tervezte és fejlesztette ki, azzal a céllal, hogy egy egyszerűen tanulható és használható, de erőteljes és kifinomult nyelvet biztosítson a fejlesztők számára. Ennek eredményeképpen a C# népszerű választás a szoftverfejlesztési projektek esetében. A C# nyelvnek számos előnye van, amelyek hozzájárulnak a hatékony és produktív kódoláshoz. Például a C# erős típusossággal rendelkezik, ami lehetővé teszi a fordítási időbeni hibák korai felfedezését, és így segít megelőzni a futási időbeli hibákat. Emellett a C# támogatja a modern programozási paradigmákat, mint például az objektumorientált és az eseményvezérelt programozást, így lehetővé téve a fejlesztők számára a különböző fejlesztési módszertanok alkalmazását a projekt igényeinek megfelelően. A C# nyelv szorosan integrálódik a .NET keretrendszerrel, amely széleskörű osztálykönyvtárat és funkciókészletet biztosít

az alkalmazások fejlesztéséhez és üzemeltetéséhez. Ennek köszönhetően a C# nyelvvel írt alkalmazások könnyen hordozhatók és skálázhatók, és lehetőség van azok gyors és hatékony fejlesztésére.

## 1.3. Felhasznált csomagok

Az alábbi pontokban részletezem jobban az általam vélt legfontosabb csomagokat, amelyek a munkámat nagy mértékben előre segítették, így megengedve, hogy a lényegi részre koncentráljak. Az AForge nevezetű csomag az egyik alapvető eszközöm volt a projekt során. Ennek segítségével könnyedén lehetett létrehozni és szerkeszteni különböző típusú képeket, ami elengedhetetlen volt a projekt szempontjából. A csomagnak több része is van, amelyek mindegyike fontos szerepet játszott a fejlesztésben. Az AForge.Imaging csomag például lehetővé tette számomra a képek manipulálását. Ez a funkcionalitás különösen hasznos volt a képek különböző effektekkel történő ellátásában és a vizuális elemek finomhangolásában. Az AForge.Math csomag segítségével pedig matematikai műveleteket végezhettem a képekkel. Ennek a csomagnak az alkalmazása lehetővé tette például a képek alakjainak és méreteinek pontos manipulálását, ami kulcsfontosságú volt a projektben. Továbbá, az AForge.Video csomag lehetővé tette számomra a webkamera képének kezelését és megjelenítését az alkalmazásban. Ennek révén dinamikus és valós idejű interakciót biztosíthattam a felhasználók számára a kamera által rögzített tartalommal. Az NAudio csomagot használtam hangfelvételek készítésére és lejátszására a projektben. Ez a csomag kiemelkedően fontos volt a hang alapú funkciók implementálásában, például hangvezérlés vagy hangüzenetek lejátszása terén. A JSON fájlok írását és olvasását a Newtonsoft.Json csomaggal valósítottam meg. Ennek segítségével könnyedén kezelhettem a strukturált adatokat.

### 1.3.1. AForge, Imaging, Math, Video

Az AForge egy nagyon sokoldalú eszköz a képfeldolgozáshoz és videómanipulációhoz a WPF alkalmazásokban. Ez a csomag lehetővé tette számomra, hogy hatékonyan kezeljem és manipuláljam a vizuális tartalmat az alkalmazásomban. Önmagában az AForge nem kínál annyi lehetséges opciót, viszont a különféle extra csomagjai nagyon sok interaktív dolgot tartalmaznak, a következőkben ezt fogom kitárgyalni.

Az AForge.Imaging csomag lehetővé tette a képek manipulálását és különböző effektek hozzáadását a WPF alkalmazásomban. Ez a csomag számos algoritmust kínál a képfeldolgozáshoz, például élek detektálását, zajcsökkentést tesz lehetővé. Emellett lehetőség van a képek tulajdonságainak finomhangolására, például a színárnyalatok, a kontraszt vagy a fényerő beállításával. Én személy szerint leginkább a kamera beállításánál használtam ezt a csomagot, ott is azért mert lehetőséget nyújtottam a klienseknél, hogy a meglévő kameraképet tudják állítani, például szürkeárnyalatosra áttudja a felhasználó rakni a megjelenített képet.

Az AForge.Math csomag matematikai műveleteket kínál a képekkel és videókkal kapcsolatban a WPF alkalmazásomban. Ennek segítségével könnyedén tudtam számításokat végezni a képek tartalmával, például a pixel értékekkel vagy a képek geometriai tulajdonságaival kapcsolatban. Ezáltal lehetőség van például az objektumok méretének, alakjának vagy pozíciójának meghatározására. A legkiemelendőbb használat számomra a Camera esetében volt, a meglévő képen különféle objektumokat, összeálló ponthalmazokat kerestem és ezeket kék vonallal összekötöttem. Például tárgyakat kerestem így a képen.

Az AForge.Video csomag lehetővé tette a videók kezelését és manipulációját. Ez a csomag lehetővé teszi a videók lejátszását, rögzítését és feldolgozását az alkalmazásban. Emellett számos beállítási lehetőséget kínál, például a videó felbontását, képkockasebességét vagy formátumát illetően. Van például egy „VideoCaptureDevice” osztály mely a videoforrások kezelését teszi lehetővé, azaz kitudjuk választani, hogy melyik kamerát használja az alkalmazás. [1] [2] [3] [4]

[1] https://github.com/andrewkirillov/AForge.NET

[2] https://github.com/andrewkirillov/AForge.NET/tree/master/Samples/Imaging

[3] https://github.com/andrewkirillov/AForge.NET/tree/master/Samples/Math

[4] https://github.com/andrewkirillov/AForge.NET/tree/master/Samples/Imaging

### 1.3.2. NAudio

Az NAudio csomagot az alkalmazásom Client részénél a hangrögzítési és lejátszási funkciók megvalósítására alkalmazom. Ennek a csomagnak a használatával képes vagyok a hangok digitális formátumba történő átalakítására, ami azt jelenti, hogy a fizikai hanghullámokat számítógépes adatokká alakítom át, és ezáltal lehetővé teszem a számítógépen történő további feldolgozást és tárolást. Az NAudio segítségével képes vagyok a mikrofonról érkező hangokat rögzíteni, majd ezeket a digitális hangfolyamatokat kezelni és tárolni a programban.

Az alkalmazásban emellett a NAudio lehetővé teszi számomra a hangok lejátszását és keverését is. Ez azt jelenti, hogy a felhasználó által küldött vagy a szerverről érkező hangüzeneteket le tudom játszani a kliensek számára, lehetőséget adva a hangkommunikációra az alkalmazáson belül. Emellett a csomag lehetővé teszi a hangok hangerősségének, hangszínének és egyéb paramétereinek finomhangolását, hogy a felhasználók igényeinek megfelelő hangélményt nyújtsak számukra.

Összességében az NAudio csomag segítségével tudom biztosítani a hangkommunikációt és a hangfeldolgozást a Kliensnél, ezáltal hozzájárulva az alkalmazás funkcionalitásának és felhasználói élményének javításához. Lényegében a Voice Communicationt így tudom megvalósítani. [5]

### 1.3.3. Newtonsoft JSON

A C# maga rendelkezik olyan beépített függvénykönyvtárral, ami képes JSON formátumot kezelni, azonban vannak esetek, amikor ez nem nyújt optimális megoldást.

A NewtonSoft Json egy olyan ingyenesen elérhető külső könyvtár, aminek módszerei egyszerűsitik a JSON kiterjesztésű fájlok, írását és olvasását. A csomaghoz C# esetén a NuGet Packageken keresztül férhetünk hozzá. [1] [6]

Egyik leghasznosabb funkciója, hogy az általunk létrehozott osztályokat

[5] https://github.com/naudio/NAudio
.

[6] https://www.newtonsoft.com/json

könnyedén szerializálhatjuk JSON formátumúvá, szövegként kezelve. Ha pedig deszerializálunk, akkor a JSON objektumot stringként tudja majd kezelni, amiből képes vissza alakítani azt egy saját típusú objektummá.

Az alkalmazásban a Newtonsoft.Json könyvtárat arra használom, hogy könnyedén kezeljem a JSON formátumot az alkalmazásom kommunikációs rétegében. Amikor a szerverről érkezik egy JSON formátumú üzenet, például szöveges üzenet vagy parancs, azt a JsonConvert.DeserializeObject<T>() metódussal deszerializálom az alkalmazásom saját osztályává, hogy könnyen kezelhessem és feldolgozhassam. Ugyanígy, amikor az alkalmazásom üzenetet vagy parancsot küld a szervernek vagy más klienseknek, először az adott objektumot alakítom JSON formátummá a JsonConvert.SerializeObject() metódus segítségével. Emellett a hang üzeneteket is JSON formátumban küldöm a kliensek között, és ezeket a hangüzeneteket is a Newtonsoft.Json könyvtárral kezelem, például a hang mintákat vagy azokkal kapcsolatos metaadatokat alakítom JSON formátummá. Összességében a Newtonsoft.Json könyvtárat az alkalmazásban a kommunikációhoz, adatátvitelhez és adatkezeléshez használom, mivel egyszerű és hatékony módot kínál a JSON formátumú adatok kezelésére és átalakítására.

# 2. MEGVALÓSÍTÁS

Az alkalmazásom két fő komponensből áll, az egyik a Server, A Server feladata a kommunikáció biztosítása, üzenetváltások kezelése és tárolása, valamint a regisztráció és felhasználókezelés menedzselése. Ez a rész teszi lehetővé az üzenetek továbbítását és az interakciót a kliensek között.

A másik rész a Kliens. A Kliens a Serveren keresztül folytatja le a kommunikációt, lehetővé teszi a szöveges és hang alapú üzenetváltást. Eredeti terveim szerint teljes körű videokommunikációt kívántam megvalósítani az alkalmazásban, azonban ennek megvalósítása nem sikerült teljes mértékben. Bár a felhasználó láthatja a saját videokamera képét és néhány műveletet végezhet vele, például mentheti azt, az élő videofolyam nem kerül átvitelre a másik kliens felé. A video kommunikáció terén az asztali alkalmazások fejlesztésekor még mindig tapasztalhatók bizonyos hiányosságok és nehézségek. Egyéb platformok, például a webes alapú alkalmazások esetén már kiforrottabb megoldások érhetőek el. Ilyen például a webRTC (Web Real-Time Communication), amely lehetővé teszi a valós idejű kommunikációt böngészők között.

A webRTC könnyen integrálható webalkalmazásokba, biztosítva a zavartalan és gyors hang- és videóátvitelt, valamint keresztplatformosságot biztosítva, így különböző eszközökön és operációs rendszereken is működik. De mivel a tématerv elején abban állapodtam meg a témavezetőmmel, hogy asztali alkalmazás lesz így végül nem szerettem volna ezen változtatni. Az egyes funkciókat külön fájlokban rendeztem, és ezeket a következő fejezetekben részletesen bemutatom.

## 2.1. Server

A *Server.cs* fájlban található funkciók megvalósítása számos lépésen ment keresztül annak érdekében, hogy hatékonyan és optimálisan működjön a projektem. A szerver számos feladatot lát el a kommunikáció során, és ennek megfelelően különböző funkciók felelnek meg ezeknek a feladatoknak.

A projektem legfőbb alapja ez volt, hiszen a kommunikáció fenntartásához egy megbízható és hatékony szerverre van szükség. Ennek megfelelően a *Server.cs* fájlban szereplő funkciók tervezése és megvalósítása kiemelten fontos volt

Az *Alert* és *SendMessage* metódusok például az üzenetek továbbítását végzik. A tervezés során először meg kellett határozni az üzenetküldés alapvető mechanizmusát.

A *SendMessage* metódus felelős az üzenetek továbbításáért a szerveren belül. Először is fogadja a paramétereket, amelyek között szerepel egy szöveges üzenet (text) és egy opcionális odd paraméter, ami egy IPEndPoint objektum. Ez az opcionális paraméter lehetővé teszi az üzenet célzott küldését egy adott vevőnek, de ha nincs megadva, akkor az összes kapcsolódott felhasználónak elküldi az üzenetet.
A metódus fő lépése, hogy végigmegy az összes szerverhez kapcsolódott felhasználón (amelyek a Users listában vannak tárolva). Minden egyes felhasználóhoz elküldi az üzenetet a *Chat.SendCommand* metódus segítségével. Ez a *SendCommand* metódus felelős az üzenet továbbításáért az adott felhasználó IP-címére a megadott szerverparancs segítségével. [2]
Fontos kis információ, amit pár sorral ezelőtt említettem, hogy az opcionális címzés ellenőrzése is megtörténik a *SendMessage* metódusban. Ha az odd paraméter nem lett megadva (azaz null), akkor az üzenetet minden felhasználónak elküldi. Ha az odd paraméter meg van adva, akkor az üzenetet csak azoknak a felhasználóknak küldi el, akiknek az IP-címe nem egyezik meg az odd paraméterben megadott címmel. **(lásd 2.1. ábra)**

```
void SendMessage(string text, IPEndPoint odd = null)
{
    foreach (User user in Users)
    {
        if (odd == null) Chat.SendCommand(ServerCommands.TextMessage, user.CommandsIpAddress, text);
        else if (odd.ToString() != user.CommandsIpAddress.ToString())
        {
            Chat.SendCommand(ServerCommands.TextMessage, user.CommandsIpAddress, text);
        }
    }
}
```

**2.1. ábra** – SendMessage metódus

Az Alert metódus fő feladata az üzenet rögzítése és továbbítása. Először is, az üzenetet hozzáadjuk egy sortöréssel a rögzítéshez, majd az üzenetet byte tömbbé alakítjuk UTF-8 kódolással. Ezt követően az üzenetet a *fs.Write* metódus segítségével rögzítjük egy fájlban, amely tartalmazza az összes rögzített üzenetet a szerveren.

Miután az üzenet rögzítésre került, a korábban tárgyalt *SendMessage* metódust hívjuk meg az üzenet továbbítására minden kapcsolódott felhasználónak. **(lásd 2.2 ábra)**

```
void Alert(string text, IPEndPoint odd = null)
{
    text += Environment.NewLine;
    byte[] ByteMessage = Encoding.UTF8.GetBytes(text);
    fs.Write(ByteMessage, 0, ByteMessage.Length);
    SendMessage(text, odd);
}
```

**2.2. ábra** – Alert metódus

Van két metódus, ami arra hivatott, hogy a Serverhez csatlakozás/lecsatlakozás esetében információt osszon meg a jelenlévő felhasználókkal.

Az *OnConnect* metódus azoknak a tevékenységeknek a végrehajtását irányítja, amelyek akkor történnek meg, amikor egy új felhasználó csatlakozik a szerverhez. Első lépésként létrehoz egy *ShortUserInfo* objektumot, amely tartalmazza az új felhasználó rövid információit, például a bejelentkezési nevét és a becenevét. Ezután rögzíti az eseményt az *Alert* metódus segítségével, amely az üzenetet és az opcionális címet kezeli. Az Alert metódus először rögzíti az üzenetet egy fájlban, majd az üzenetet továbbítja minden kapcsolódott felhasználónak a *SendMessage* metóduson keresztül.

Az *OnDisconnect* metódus hasonló módon működik, de azokra a tevékenységekre összpontosít, amelyek akkor történnek, amikor egy felhasználó lecsatlakozik a szerverről. Először is ellenőrzi, hogy a lecsatlakozott felhasználó objektuma nem üres-e, majd létrehozza a *ShortUserInfo* objektumot, hogy rögzítse a felhasználó lecsatlakozását. Ezután az *Alert* metódust hívja meg az üzenet rögzítésére és továbbítására az összes kapcsolódott felhasználónak.

A *Connect* és *Disconnect* metódusok kulcsfontosságú szerepet töltenek be az új felhasználók csatlakozásának és lecsatlakozásának biztosításában a szerveren. Ezeknek a metódusoknak az alapos megtervezése és végrehajtása létfontosságú a szerver egészének hatékony működése szempontjából.

A *Connect* metódus első lépése az új felhasználó objektumának (user) ellenőrzése, hogy valóban létezik-e, és hogy megfelelően van-e inicializálva. Ha a felhasználó objektum nem üres, akkor folytatódik a csatlakozási folyamat. Ebben a folyamatban szerepel a felhasználó már létező csatlakozásának ellenőrzése. Ha a felhasználó már csatlakozott a szerverhez, akkor a metódus hamissal (false) tér vissza, hogy ne engedje meg a duplikált csatlakozást. Ez elengedhetetlen abban, hogy egy adott felhasználó ne tudjon kétszer is belépni. Ezt követően ellenőrzöm, hogy a felhasználó jelenleg nincs-e csatlakozva másik szerverhez. Ha a felhasználó nincs másik szerverhez csatlakozva, és a szerveren jelenleg kevesebb felhasználó van, mint a maximális felhasználószám (MaxUsersCount), akkor a csatlakozást engedélyezem. Ebben az esetben a felhasználóhoz hozzáadjuk a szerver felhasználóinak listáját, ezt majd a kliens esetében kifejtem jobban mert ott látható, és igaz azaz true értékkel térünk vissza, hogy jelezzük a sikeres csatlakozást.

Fontos szempont az is, hogy a felhasználó által megadott jelszót ellenőrizzük a szerveren tárolt jelszóval. Ha a felhasználó megadott jelszava megegyezik a szerveren tárolt jelszóval, akkor engedélyezzük a csatlakozást. Ha nem, akkor a csatlakozást megtagadjuk, és hamissal térünk vissza.

A Disconnect metódus felelős a felhasználó biztonságos lecsatlakozásáért a szerverről. Ez a folyamat magában foglalja a felhasználó jelenlegi szerverkapcsolatának ellenőrzését, majd a felhasználó szerverkapcsolatának lezárását és eltávolítását a felhasználók listájából. A metódus továbbá ellenőrzi, hogy a felhasználó a kijelentkezés pillanatában kapcsolódik-e a szerverhez. Ha a felhasználó éppen másik szerverhez kapcsolódik, akkor a metódus automatikusan kitoloncolja a felhasználót a jelenlegi szerverről.

Szerintem fontos megjegyezni, hogy mindkét metódus célja, hogy biztosítsa a szerveren való hatékony és biztonságos csatlakozást és lecsatlakozást, valamint, hogy megfelelően kezelje a felhasználók számának és a szerverterhelésnek az ellenőrzését a zavartalan működés érdekében. Ez legfőképpen sok kliens esetében lényeges, mivel alapvetően lokálisan futtatom a projektem, így a számítógép erőforrása korlátozott.

Ezek voltak a *Server.cs* legfontosabb metódusai és maga a szerver lényege. A

következőkben bemutatom a többi részét is a szervernek, a többi kisebb fájlt is ami kapcsolódik hozzá és ami arra lett létrehozva, hogy egy működőképes korrekt *Server* működhessen.

### 2.1.1. Chat

A Chat.cs fájl alapvető feladata a szerveren belüli kommunikáció kezelése és koordinálása. Ennek megfelelően ez a fájl felelős az üzenetek, hangüzenetek és fájlok fogadásáért, továbbításáért és kezeléséért a kliensek között. Emellett gondoskodik a kliensek frissítéséről és az esetleges fájlok rendelkezésre állásának ellenőrzéséről is.
Az osztály konstruktora során először inicializálja az adatbázist a *Database.Init()* hívás segítségével. Ez a lépés biztosítja, hogy az alkalmazás megfelelően működjön az adatbázissal. Ezután a konstruktor átveszi a *ServerWindow* objektumot, amelyet az ablakkezelésért felelős osztályként definiál. A konstruktor további lépéseként lekérdezi az adatbázisban tárolt felhasználók és szerverek számát, majd megjeleníti ezeket a számokat a szerver naplójában a *window.AddLog()* hívások segítségével. A felhasználók és szerverek számának kiírása információval szolgál arról, hogy az adatbázis sikeresen betöltődött-e, vagy esetleg új adatbázist kellett létrehozni.
Ezután a konstruktor aszinkron módon frissíti a kliensek és szerverek listáit a *window.ShowUsers_async()* és *window.ShowServers_async()* metódusok segítségével. Ezek a metódusok a GUI frissítését végzik, hogy a felhasználók és szerverek listája mindig naprakész legyen.
A konstruktor végül létrehoz két új szálat: a *CommandsRecieverThread*-et és a *VoiceRecieverThread*-et. Ezek a szálak felelősek a kliensektől érkező parancsok és hangüzenetek fogadásáért. A *CommandsRecieverThread* a *CommandsReciever* metódust futtatja szálként, míg a *VoiceRecieverThread* a *VoiceReciever* metódust. A szálak létrehozása biztosítja, hogy a szerver egyszerre képes legyen fogadni és kezelni az üzeneteket és hangüzeneteket, anélkül, hogy a fő szálat blokkolná. **(lásd 2.3. ábra)**

```
public Chat(ServerWindow sw)
{
    Database.Init();
    window = sw;
    int a = Database.GetUsersCount();
    window.AddLog("SERVER", "Chat is loaded.");
    string text1 = Database.GetUsersCount() > 0 ? $"User database loaded " +
        $"({Database.GetUsersCount()})" : $"The user database was not loaded, a new one was created.";
    window.AddLog("SERVER", text1);
    string text2 = Database.GetServersCount() > 0 ? $"Server database loaded ({Database.GetServersCount()})"
        $"The server database was not loaded, a new one was created.";
    window.AddLog("SERVER", text2);
    window.ShowUsers_async();
    window.ShowServers_async();
    CommandsRecieverThread = new Thread(CommandsReciever) { Name = "Commands Reciever" };
    CommandsRecieverThread.Start();
    VoiceRecieverThread = new Thread(VoiceReciever) { Name = "Voice Reciever" };
    VoiceRecieverThread.Start();
}
```

**2.3. ábra** – Chat metódus

A CommandsReciever(Socket clientSocket) metódus kliensről érkező parancsok fogadásáért felelős. Ennek érdekében a paraméterként kapott Socket objektumot használja a kommunikációhoz. Amikor egy üzenet érkezik a kliensről, a metódus először JSON formátumban fogadja az üzenetet a Socket objektum segítségével. Ezután a JSON formátumban kapott üzenetet átalakítja az alkalmazás számára értelmezhető objektummá a NewtonSoft.Json segítségével. Ez a strukturált formában történő átalakítás megkönnyíti a parancsok feldolgozását és a megfelelő kezelőmetódusoknak történő továbbítását. A JSON használata lehetővé teszi az üzenetek egyszerű és hatékony kezelését a kliens és a szerver közötti kommunikáció során.

A VoiceReciever(Socket clientSocket) metódus feladata a hangüzenetek fogadása a kliensről. A paraméterként kapott Socket objektum segítségével figyeli a kliensről érkező hangüzeneteket. Amikor érkezik egy hangüzenet, a metódus feldolgozza azt, majd továbbítja a megfelelő kezelőmetódusnak a hangüzenet feldolgozására.

A *SendMessage(string text, IPEndPoint odd = null)* metódus az üzenetek továbbításáért felel a szerveren belül. A *text* paraméter tartalmazza az üzenetet, amelyet továbbítani kell. Az opcionális *odd* paraméter egy IPEndPoint objektum, amely egy adott vevőt céloz meg az üzenet küldésére. Ha ez a paraméter nincs megadva, az üzenetet az összes kapcsolódott felhasználónak elküldi. Lényegében ugyan az, mint a Server.cs esetében.

Az *Alert(string message)* metódus rögzíti és továbbítja az üzeneteket. A *message* paraméter tartalmazza az üzenetet, amelyet rögzíteni és továbbítani kell. Az üzenetet először rögzíti egy fájlban, majd továbbítja az üzenetet minden kapcsolódott felhasználónak a *SendMessage* metóduson keresztül.

Az *UpdateClient(Socket clientSocket)* metódus felelős a kliensek frissítéséért. A paraméterként kapott Socket objektum segítségével küldi az aktuális szerverinformációkat és frissítéseket a kliensnek.

A *SendFile(string fileName, Socket clientSocket)* metódus fájlok küldéséért felelős a kliens részére. A *fileName* paraméter tartalmazza a küldendő fájl elérési útvonalát, míg a *clientSocket* paraméter egy Socket objektum, amely a kliens kapcsolatát jelképezi.

A *SendHasFile(string fileName, IPEndPoint odd)* metódus ellenőrzi, hogy egy adott fájl rendelkezésre áll-e a szerveren. A *fileName* paraméter tartalmazza a vizsgálandó fájl elérési útvonalát, az opcionális *odd* paraméter pedig ismételten a szokásos IPEndPoint objektum, amely egy adott vevőt céloz meg az ellenőrzés során.

### 2.1.2. Database

A *Database.cs* fájl felelős az alkalmazás adatbázisának kezeléséért és az adatok tárolásáért. Az osztály statikus és tartalmaz két központi adatszerkezetet: a *Users* és a *Servers* Dictionary-t magyarul szótárakat. Ezek tárolják a felhasználók és a szerverek adatait, és az azokhoz kapcsolódó információkat.

Az osztályban előre definiáltam változókat, bár inkább kulcsszavakat, mint a *UsersFolderPath*, *ServersFolderPath*, *ServersHistoryFolder*, *FilesPath*, és *ClientFilePath*, amelyek az alkalmazás különböző részeinek elérési útvonalát tárolják. Ezek a konstansok azért vannak definiálva, hogy a későbbiekben a kimentett fájloknak az elérését megkönnyítsem**. (lásd 2.4. ábra)**

```
public static class Database
{
    13 references
    public static Dictionary<string, User> Users { get; private set; } = new Dictionary<string, User>();
    17 references
    public static Dictionary<string, Server> Servers { get; private set; } = new Dictionary<string, Server>();

    public static string UsersFolderPath = Environment.CurrentDirectory + @"/Data/users";
    public static string UsersFilePath;
    public static string UsersFilePath_backup;

    public static string ServersFolderPath = Environment.CurrentDirectory + @"/Data/servers";
    public static string ServersFilePath;
    public static string ServersFilePath_backup;

    public static string ServersHistoryFolder = Environment.CurrentDirectory + @"/Data/histories";

    public const string FilesPath = @"Files/";
    public const string ClientFilePath = @"Chat_Client.exe";
```

**2.4. ábra** – Database-ben definiált Path változók

Az Init() metódus az adatbázis inicializálásáért felelős. Ez a függvény beállítja az adatbázis fájlneveit és az alapvető könyvtárstruktúrát. Ezenkívül betölti a

felhasználók és szerverek adatait, és ellenőrzi, hogy legalább egy szerver létezik-e. Ha nincs még szerver, akkor egy alapértelmezett szerver létrehozása történik a "Developer server" címmel.

Az osztályban számos segédmetódust is létrehoztam, melyek a felhasználókkal és szerverekkel kapcsolatos műveleteket végzik. Ilyen például a *GetUsersCount()* és *GetServersCount()* metódus, amelyek visszaadják a tárolt felhasználók és szerverek számát. Ez legfőképpen azért is fontos, mert egy adott szerveren korlátozva van a userek száma, és nem lenne optimális ezt az értéket átlépni. Lényegében érvényét vesztené a korlátozás.

Az adatbázis manipulálására szolgáló legfontosabb metódusok közé tartozik a CreateUser() és CreateServer() metódusok, amelyek új felhasználókat és szervereket hoznak létre a tárolt adatokban. Ezek a metódusok ellenőrzik a kapott paramétereket, és csak akkor hajtják végre a létrehozást, ha azok megfelelőek.

Az első lépés mindkét metódusban az adatok ellenőrzése. A *CreateUser()* metódus ellenőrzi a felhasználói név, bejelentkezési név, jelszó és e-mail cím hosszát és érvényességét. A *CreateServer()* metódus szintén ellenőrzi a szerver címét, azonosítóját, maximális felhasználószámát és opcionális jelszavát.

Miután az ellenőrzés sikeresen megtörtént, létrejön az új felhasználó vagy szerver objektum a megfelelő paraméterekkel. A *CreateUser()* metódus létrehoz egy új User objektumot a megadott felhasználói adatokkal, beleértve a bejelentkezési nevet, jelszót, e-mail címet stb. Ugyanezen elven a *CreateServer()* metódus létrehoz egy új Server objektumot a megadott szerverinformációkkal.

Végül mindkét metódus beilleszti az újonnan létrehozott felhasználót vagy szervert az adatbázisba. A *CreateUser()* metódus hozzáadja az új felhasználót a *Users* Dictionaryhoz a bejelentkezési név alapján, míg a *CreateServer()* metódus hozzáadja az új szerver objektumot a *Servers* Dictionaryhoz a szerver azonosítója alapján. Majdnem ugyan az a kettő metódus, csak picit más dolgokat kezelnek.

Adatintegritás szempontjából nagyon fontos mind a kettő metódus, mivel biztosítják, hogy csak érvényes és megfelelő adatokkal lehessen új felhasználókat és szervereket regisztrálni az alkalmazásban.

Az adatok mentését és betöltését nagyon fontos dolognak tartottam már az alkalmazás készítésének első pillanatától fogva. A *SaveUsers()* és *SaveServers()* metódusok felelősek az adatok tartós tárolásáért, míg a *LoadUsers()* és *LoadServers()* metódusok gondoskodnak az adatok visszatöltéséről az alkalmazás újraindítása vagy

más működési szükségletek esetén.

Először is, a *SaveUsers()* és *SaveServers()* metódusok létrehozzák az adatokat tartalmazó objektumokat (UsersSavingData és ServersSavingData), amelyek segítségével strukturált formában tárolják a felhasználók és szerverek adatait. Ezek az objektumok tartalmazzák a szükséges információkat a felhasználók és szerverek tulajdonságairól, például a nevüket, bejelentkezési adataikat, jelszavaikat, stb.

A *SaveUsers()* metódus összegyűjti a felhasználók adatait a Users szótárból, majd ezeket az adatokat elmenti a megfelelő fájlba. Ehhez először létrehoztam egy XML serializer objektumot (UsersXML), majd ezzel konvertáltam át az adatokat XML formátumba. A metódus végül a serializer segítségével az adatokat kiírja egy fájlba.

Kísértetiesen hasonlóan működik a SaveServers() metódus is, de itt a Servers szótárból gyűjti össze a szerverek adatait, majd menti el azokat egy fájlba.

A LoadUsers() és LoadServers() metódusok a mentett adatok visszatöltését végzik. Először ellenőrzik, hogy a megfelelő mappastruktúra és fájlok léteznek-e a szükséges adatokhoz. Ezután betöltik az adatokat a megfelelő fájlokból. Ehhez először létrehoznak egy XML deserializer objektumot (UsersXML és ServersXML), majd ezzel olvassák be az adatokat a fájlokból. Végül ezeket az adatokat visszaalakítják a megfelelő objektumokká, és elhelyezik az adatbázisban.

A kódról összességében elmondható, hogy nemcsak képes felhasználók és szerverek kezelésére, de biztosítja azoknak a létrehozását, eltávolítását, valamint az adatok mentését és betöltését is. Emellett lehetőséget biztosít a felhasználók közötti kommunikációra, üzenetek, hangüzenetek és fájlok továbbítására és kezelésére. A szerver oldalon a kliensek csatlakozásának, lecsatlakozásának, valamint az ezek közötti kommunikáció koordinálására is képes.

Mindezek mellett az adatbázis kezelése is komplex, lehetővé téve új felhasználók és szerverek hozzáadását, azok adatainak módosítását és eltávolítását. Az adatok strukturált tárolása XML formátumban biztosítja az egyszerű és hatékony adatelérést, míg az adatok mentése és betöltése garantálja, hogy az alkalmazás mindig naprakész és pontos adatokkal dolgozzon.

Ez a kód azonban nemcsak a funkcionalitásban gazdag, hanem jól strukturált és átlátható is. Az osztályok és metódusok megfelelő elnevezése, valamint a kommentek segítik az olvashatóságot és karbantarthatóságot. Összességében ez a kód egy komplex rendszert hoz létre egy viszonylag kis terjedelemben, ami kiváló példa a hatékony és strukturált szoftverfejlesztésre. Legalábbis nagyon közérthető, hogy mit csinál a kód.

### 2.1.3. ServerWindow

Mivel bemutattam a Server, Chat és Database alapvető funkcionalitását így most az utolsó pontként rátérek magára a ServerWindow-ra amely a felhasználó számára egy grafikus felületet nyújt és ott látják az információkat. Ezt is két részre szednéd picit.

A *ServerWindow.xaml* fájl valójában a szerveralkalmazás közvetlen kapcsolatát jelenti a felhasználóval. Itt határozzuk meg azt, hogy milyen módon jelenik meg és milyen interakciós lehetőségeket kínál a felhasználóknak. Ha például a felhasználó be szeretné állítani a szerver beállításait vagy figyelemmel kívánja kísérni a szerveren zajló eseményeket, ezt az ablakot használja.

Az XAML fájlban található különböző UI elemek, mint például gombok, címkék, szövegdobozok és listák, pontosan meghatározzák, hogy milyen eszközökkel és információkkal rendelkeznek a felhasználók az alkalmazás használata során. A gombokra kattintva indíthatnak vagy állíthatnak le szerverfunkciókat, a címkéken keresztül pedig információkat kapnak a szerver állapotáról vagy esetleges hibáiról.

Az elrendezési és formázási elemek, például a rácsok (Grid), segítenek abban, hogy az ablak szervezett és átlátható legyen. Ezáltal a felhasználók könnyen megtalálhatják azokat az elemeket, amelyekkel interakcióba szeretnének lépni, és könnyen átláthatják az általuk megjelenített információkat.

Összességében a *ServerWindow.xaml* fájlt én egyfajta kapuként definiálnám a szerveralkalmazásban, ahol a felhasználók belépnek és kommunikálnak a szoftverrel. Ennek a fájlnak a megfelelő kialakítása és konfigurálása fontos szempont volt elsősorban a fejlesztés miatt, hogy átlátható legyen, és természetesen felhasználói szempontból is. **(lásd 2.5. ábra)**


**2.5. ábra** – Server Ablaka

A *ServerWindow.xaml.cs* fájl a ServerWindow grafikus felületének mögötti kódot tartalmazza, és a "backend" részét képezi az alkalmazásnak. Ez a fájl végzi a különböző események kezelését, kommunikál a háttérben futó szerverrel és adatbázissal, valamint frissíti a felhasználói felületet az adatok alapján.

Ez a C# fájl tartalmazza az eseménykezelő függvényeket, amelyek reagálnak például a gombokra kattintásra vagy az ablak bezárására. A szerverrel való kommunikációért felelős metódusok is itt találhatóak, amelyek lehetővé teszik a felhasználók számára, hogy indítsák vagy állítsák le a szerver működését, valamint más szerverrel kapcsolatos műveleteket hajtsanak végre.

Ezenkívül a *ServerWindow.xaml.cs* fájl gondoskodik az adatok megjelenítéséről és frissítéséről a felhasználói felületen. Például, ha új felhasználók csatlakoznak a szerverhez, vagy ha fontos események történnek, például üzenetek érkeznek vagy új szerverek jönnek létre, akkor ezek az információk automatikusan frissülnek az ablakban.

Az adatbázisba való hozzáférés, az adatok lekérése és mentése, valamint azok megjelenítése a felhasználói felületen is itt történik.

Vegyük ezeket szépen sorba.

A *Window_Loaded* metódus a ServerWindow osztály része, és akkor hívódik meg, amikor az ablak betöltődik. Ennek a metódusnak a célja az ablak inicializálása és az alapvető beállítások végrehajtása annak érdekében, hogy az ablak megfelelően működjön az alkalmazásban. A metódusban először inicializáltam az ablakban található

különböző UI elemeket, például gombokat, címkéket, listákat stb. Ezeknek az elemeknek a példányait a XAML fájlban definiáltam és adtam meg a nevüket, hogy hozzáférhessek hozzájuk a kód részben. Ebben a metódusban például az *UserInfo* és *ServersInfo* nevű címkéket inicializáltam, amelyek a felhasználói információkat és a szerverinformációkat jelenítik meg az ablakban.

Ezután beállítottam az ablakban található UI elemek alapértelmezett értékeit, például a címkék tartalmát, a gombok háttérképét és a lista elemek tartalmát. Például a *ServerTitle* címke tartalmát beállítottam a szerver verziójával (Version), és betöltöttem azokat az erőforrásokat, amelyekre az ablaknak szüksége van a működéséhez, mint például képek, ikonok vagy más médiafájlok. A *BitmapImage* típusú változók, mint például *Button_Background_1* és *Button_Background_2*, azokat az erőforrásokat tartalmazzák, amelyeket a gombok háttérképének beállítására használtam. **(lásd 2.6. ábra)**

```
void Window_Loaded(object sender, RoutedEventArgs e)
{
    UserInfo.Content = "";
    ServersInfo.Content = "";

    ServerTitle.Content = "Server" + Version;
    Button_Background_1 = LoadBitmapFromResource("Button_Background_1");
    Button_Background_2 = LoadBitmapFromResource("Button_Background_2");
    Tabs.Background = new SolidColorBrush(Colors.Transparent);
    Tabs.BorderBrush = new SolidColorBrush(Colors.Transparent);
    foreach (TabItem tab in Tabs.Items)
    {
        tab.Visibility = Visibility.Collapsed;
    }
    AddLog("SERVER", "Window loaded.");
    chat = new Chat(this);
}
```

**2.6. ábra** – Window_Loaded metódus

Végül hozzáadtam eseménykezelőket a különböző UI elemekhez, hogy reagáljak a felhasználó interakcióira, például kötöttem hozzá a gombok kattintásához vagy az egér mozgatásához szükséges műveleteket.

Az AddLog_delegate és AddLog metódusok felelősek az ablakhoz tartozó naplónak (log) a frissítéséért. Az AddLog_delegate egy delegált típusú metódus, amelyet a AddLog metódus hív meg. A Dispatcher.Invoke segítségével ezek a metódusok szinkronizálják a napló frissítését a grafikus felület szálával, biztosítva a megfelelő működést az alkalmazásban. **(lásd 2.7. ábra)**

```
1 reference
public void AddLog_delegate(string sender, string text)
{
    Dispatcher.Invoke(new AddLogDelegate(AddLog), sender, text);
}

7 references
public void AddLog(string sender, string text)
{
    if (Log.Text != "") Log.Text += $"{Environment.NewLine}[{sender}]: {text}";
    else Log.Text += $"[{sender}]: {text}";
    Log.ScrollToEnd();
}
```

**2.7. ábra** -AddLog és AddLog_delegate metódusok

A ServerWindow_Closing metódus az ablak bezárásáért felelős. A ServerWindow_Closing metódus az ablak bezárásakor fut le, és felelős a chat kapcsolatok lezárásáért, valamint az alkalmazás leállításáért.

A Cut_Down_Button_Click, Mouse_Enter_Cut_Down_Button és Mouse_Leave_Cut_Down_Button metódusok a "Minimize" vagy "Minimálisra csökkentés" gombhoz kapcsolódnak. A Cut_Down_Button_Click metódus határozza meg azt a műveletet, amelyet a felhasználó kattintáskor végez, vagyis az ablak minimalizálása. A Mouse_Enter_Cut_Down_Button és Mouse_Leave_Cut_Down_Button metódusok az egér belépése és kilépése eseményeire reagálnak, és megváltoztatják a gomb kinézetét az egérmozgatás során. Azaz az egér belépése és kilépése eseményeire reagálnak, és dinamikusan változtatják a gomb kinézetét az egérmozgatás során, így vizuálisan jelzik a felhasználónak, hogy az egérkurzor éppen az adott gomb felett van. Ezáltal a felhasználó könnyebben észleli és érti, hogy az adott gomb interaktív, és reagál az egérmozgásra.

A Window_MouseDown metódus például akkor hívódik meg, amikor a felhasználó valahová kattint az ablakban, és lehetővé teszi az ablak húzását az egérmozgatással, ha az egér bal gombját lenyomták. Ez növeli az interaktivitást és a felhasználói élményt.

Csináltam egy nagyon érdekes metódust, ezekhez felhasználtam pár StackOverFlow kódot, míg végül kitaláltam, hogyan is kellene megvalósítani.

A LoadBitmapFromResource metódust azért találtam ötletesnek mert így az alkalmazásban lévő képek betöltése közvetlenül az alkalmazás erőforrásaiból történik, anélkül, hogy külső fájlokat kellene használni.

Ez azért fontos, mert lehetővé teszi az alkalmazás számára, hogy önellátó legyen képek terén, ami javítja az alkalmazás hordozhatóságát és biztonságát. A képek az alkalmazás

részét képezik, így az alkalmazás futtatásához nem szükséges azok külön kezelése vagy letöltése. Ezenkívül a metódus dinamikusan kezeli a képek elérését, mivel az aktuális alkalmazás gyűjteményéből, azaz az aktuálisan futó alkalmazásból tölti be a képeket. Ez biztosítja a helyes elérési útvonalat és minimalizálja a hibalehetőségeket, amelyek az elérési útvonal megadásával jelentkezhetnek.

Így a LoadBitmapFromResource metódus valójában egy hatékony eszköz az alkalmazás grafikus felületének kezelésében, amely lehetővé teszi a dinamikus és rugalmas GUI kialakítását, anélkül, hogy külső fájlokhoz kellene folyamodni, vagy a felhasználónak manuálisan kellene biztosítania a képeket az alkalmazáshoz. Ezáltal növeli az alkalmazás egyszerűségét és hordozhatóságát, miközben javítja az GUI teljesítményét és skálázhatóságát. [7] [8] **(lásd 2.8. ábra)**

```
public static BitmapImage LoadBitmapFromResource(string Path)
{
    Path = @"Resources/" + Path + ".png";
    Assembly assembly = Assembly.GetCallingAssembly();
    if (Path[0] == '/')
    {
        Path = Path.Substring(1);
    }
    return new BitmapImage(new Uri(@"pack://application:,,,/"
    + assembly.GetName().Name + ";component/" + Path, UriKind.Absolute));
}
```

**2.8. ábra** – LoadBitmapFromResource

Az *ShowUsers* metódus felelős az "UsersList" elem tartalmának frissítéséért azáltal, hogy törli az összes elemet a listából, majd végig iterál a felhasználókon az adatbázisban, hozzáadva minden felhasználó becenevét az "UsersList" elemhez.

A *ShowUsers_async* metódus egy olyan mechanizmust alkalmaz, amely lehetővé teszi a *ShowUsers* metódus hívását egy másik szálon, anélkül hogy blokkolná a fő felhasználói felület (UI) szálat.

Ez azt jelenti, hogy a felhasználó továbbra is tudja használni az alkalmazást, miközben az adatok frissítése zajlik a háttérben.

Amikor az alkalmazás egy adatbázis módosítást végrehajt, például egy új felhasználó hozzáadását vagy egy meglévő frissítését, ezek a műveletek gyakran más szálakon történnek annak érdekében, hogy ne blokkolják az UI szálat. Az *ShowUsers_async* metódus lehetővé teszi, hogy ezek a háttérfolyamatok aszinkron módon frissítsék az

[7] https://stackoverflow.com/questions/569561/dynamic-loading-of-images-in-wpf

[8] https://stackoverflow.com/questions/347614/storing-wpf-image-resources

"UsersList" elem tartalmát a *ShowUsers* metóduson keresztül, miközben a felhasználó folytatja az alkalmazás használatát.

Ez azért fontos, mert az alkalmazás reakcióképességének fenntartása kulcsfontosságú egy jól használható alkalmazás esetében. Ha az adatok frissítése a fő UI szálon történne, az blokkolhatná az alkalmazás válaszreakcióit, és lassíthatná az UI-t, ami zavaró lehet a felhasználók számára. Az aszinkron hívások lehetővé teszik az adatfrissítéseket, miközben az alkalmazás továbbra is reagál a felhasználó interakcióira, ezáltal javítva az általános felhasználói élményt.

A ShowServers metódus hasonlóan működik a "ShowUsers" metódushoz, de a "ServersList" elem tartalmát frissíti, törölve az összes elemet a listából, majd végig iterálva a szervereken az adatbázisban, hozzáadva minden szerver címét az "ServersList" elemhez. Az alábbi ábrán látható a kimenet, hogy milyen funkcionalitása van ennek a metódusnak. **(lásd 2.9. ábra)**


**2.9. ábra** – ServerList megjelenítése és annak tartalma

A ShowServers_async metódus ugyanolyan módon működik, mint az ShowUsers_async, azaz lehetővé teszi a ShowServers metódus hívását egy másik szálon. Ennek eredményeként az adatok frissítése aszinkron módon történik, ami azt jelenti, hogy a háttérfolyamatok nem blokkolják az UI szálat.

Az alkalmazásban, ha például új szervereket adok hozzá vagy a meglévőket frissítem az adatbázisban, ezek a műveletek más szálakon végezhetők el annak érdekében, hogy ne akadályozzák az UI-t. Az ShowServers_async metódus lehetővé teszi, hogy ezek a háttérfolyamatok frissítsék az "ServersList" elem tartalmát a ShowServers metóduson

keresztül, miközben a fő UI szál szabadon marad a felhasználói interakciók kezelésére.

A UsersList_SelectionChanged metódus akkor hívódik meg, amikor a felhasználó kiválaszt egy elemet az "UsersList" listából, és megjeleníti a kiválasztott felhasználó részletes adatait.

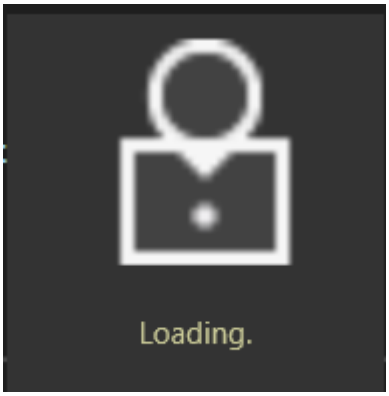
A Send_Click metódus a "Küldés" gomb lenyomásakor hívódik meg a parancs beíró mezőben, és feldolgozza a felhasználó által bevitt parancsot, például lehetővé teszi a szerverek létrehozását és értesítések küldését.

A ServersList_SelectionChanged ugyan azon elven működik, mint a UsersList_SelectionChanged, azaz ha rákattintok a ServerList listából egy elemre akkor megjeleníti a kiválasztott Server részletes adatait.

Ezzel lényegében letudtam a Server összes funkcionalitását. A végén a Használat részben kifejtem a Server és a Kliens felületének minden kis részét, pont azért, hogy egyben minden be legyen mutatva.

## 2.2 Kliens

Elérkeztünk végre a Klienshez. A kliens belépési pontja az Init.xaml és az Init.xaml.cs fájl. Ez egy kezdeti ablakot hoz létre a Chat kliens számára. A fő szerepe az, hogy kezdeti inicializációt végezzen, például beállítsa a szükséges konfigurációkat és inicializálja a kapcsolatot a szerverrel. Emellett ellenőrzi a szerver verzióját, hogy biztosítsa a kompatibilitást és a frissítéseket. Ez a kezdeti ablak mutatja a felhasználónak a betöltési folyamatot, miközben a szükséges előkészületek zajlanak a Chat alkalmazás használatához. **(lásd 2.10. ábra)**




**2.10. ábra** – Init ablak

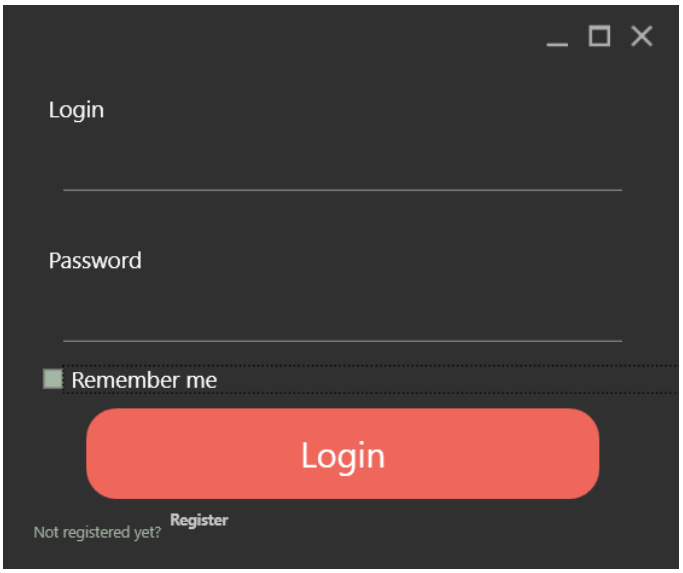
Van egy *Check* metódus, ami arra szolgál, hogy ellenőrizze a szerver verzióját, és az alapján döntsön a további teendőkről. Először lekéri a szerver verzióját a

*Chat.GetServerVersion()* függvénnyel. Ha a verzió lekérése során valamilyen hiba történik, és a visszatérési érték "ERROR", akkor a metódus leállítja az animációt (*AnimationThread.Abort()*) és beállítja a kezdeti ablak szövegét "Chat unavailable"-re (*SetText_async("Chat unavailable")*). Ha viszont sikeresen megszerezte a szerver verzióját, akkor azt összehasonlítja a helyi verzióval (*Library.Version*). Ha megegyezik a két verzió, akkor a *LoadForm* delegáltat meghívja a fő szálon. Ha nem egyeznek meg a verziók, akkor is ugyanazt a *LoadForm* delegáltat hívja meg.
A könnyebb megértés miatt kifejtem a delegáltat, mivel ez korábban is előjött már mint fogalom. A delegált egy olyan típus, amely hivatkozást tartalmaz egy vagy több metódusra. Gyakorlatilag lehetővé teszi, hogy egy függvényt egy változóként kezeljünk, és aztán ezt a változót átadhassuk más függvényeknek, tároljuk egy adatszerkezetben vagy visszatérési érték lehet. A delegáltak segítségével lehetőség van arra, hogy dinamikusan hivatkozzunk vagy hivatkozási pontokat adjunk át különböző metódusokhoz, ami rugalmasabbá teszi a programozást. A delegált típus definiálja a metódus paramétereit és visszatérési értékét, és lehetővé teszi, hogy hivatkozzunk arra a metódusra, amelyet az adott delegált típus tartalmaz.

*Amennyiben sikeresen lefutott a Check metódus akkor a Success metódus* hatásásra az Auth.xaml meghívódik, azaz betöltődik a következő ablak.

### 2.2.1. Auth

Az *Auth* ablak az alkalmazás bejelentkezési felülete, ahol a felhasználók bejelentkezhetnek vagy regisztrálhatnak a chat alkalmazásba. Az ablakban található felhasználónév és jelszó mezők segítségével adhatják meg az szükséges adatokat. A gombok lehetővé teszik a bejelentkezést vagy regisztrációt, és a felhasználók választhatnak az emlékezés funkció használata között. Az ablak megjelenése részletesen kialakított, beleértve a színeket, a betűméreteket, a gombok stílusát és az ikonokat is. Az eseménykezelők segítségével az ablak kezeli a felhasználók interakcióit, például a bejelentkezés/regisztráció gombra kattintást vagy az ablak bezárását. Összességében az Auth ablak fontos funkciója a felhasználók azonosítása és hitelesítése a chat alkalmazásban, valamint lehetőséget biztosít az új felhasználók regisztrációjára.
**(lásd 2.11. ábra)**




**2.11. ábra –** Authentication ablak login része

Az AES (Advanced Encryption Standard) az Auth ablakban alkalmazott titkosítási mechanizmus. Ezt a mechanizmust azért használom, hogy a felhasználók bejelentkezési adatait, mint például a felhasználónév és jelszó, biztonságosan tároljam a felhasználó eszközén. Így azokat csak az alkalmazás megfelelő működéséhez szükséges esetekben kellhet az alkalmazásnak elküldenie a szervernek.

Az AES használata nagyban hozzájárul az alkalmazás biztonságához, mivel a tárolt adatokat titkosított formában tárolja. Így még akkor is, ha valaki hozzáfér az eszközhöz, nem tudja könnyen kiolvasni a felhasználói adatokat, mivel azok titkosított formában vannak tárolva. [3]

Az Auth ablakban alkalmazott AES két fő funkciót lát el: az adatok titkosítását és azok visszafejtését. Az adatok titkosításához a felhasználó által megadott jelszót és a bejelentkezési adatokat felhasználva egy titkosított stringet generál. Ezt a titkosított stringet tárolja el a felhasználó eszközén. Az adatok visszafejtéséhez pedig a tárolt titkosított stringet és a megfelelő jelszót felhasználva visszafejti az eredeti, olvasható formába.

Az AES használata növeli az alkalmazás biztonságát, mivel még akkor sem olvashatóak ki a felhasználói adatok, ha valaki hozzáfér az eszközhöz vagy az adatokhoz. Ezzel tudtam biztosítani, hogy a felhasználók bejelentkezési adatai teljes biztonságban legyenek.

Ehhez nézzünk egy gyakorlati példa. Tegyük fel, hogy van egy szövegünk, például a " Hello, world!", amit titkosítani szeretnénk.

**Kulcs generálása:** Az AES titkosításhoz egy titkos kulcsra van szükségünk. Ez a kulcs lehet bármilyen véletlenszerűen generált bájtsorozat. Általában 128, 192 vagy 256 bit hosszú kulcsokat használunk. Például legyen ez a kulcs: `mysecretkey`.

**Blokkokra bontás**: Az AES blokk titkosítás, tehát a szöveget blokkokra bontjuk. A leggyakoribb blokkméret az AES esetében 128 bit (16 bájt).

**Padding (kitöltés)**: Ha a szöveg hossza nem osztható 16-tal, akkor kitöltjük a hiányzó részt. Például: `"Hello, world!"` -> `"Hello, world!    "` (4 szóköz a kitöltéshez).

**Titkosítás**: Az AES titkosítja a blokkokat a kulcs segítségével. Az első blokkot az előző blokk eredményével (vagy egy kezdeti vektorral) kombináljuk. Az eredmény egy kódolt blokk.

**Visszafejtés**: A visszafejtés során ugyanezeket a lépéseket alkalmazzuk a kódolt blokkokra, de a kulcsot megfordítva használjuk.

**Példa:**

Szöveg: "Hello, world!"

Kulcs: "mysecretkey"

Titkosított szöveg: "9d4e1e23bd5b727046a9e3b4b7db57bd"

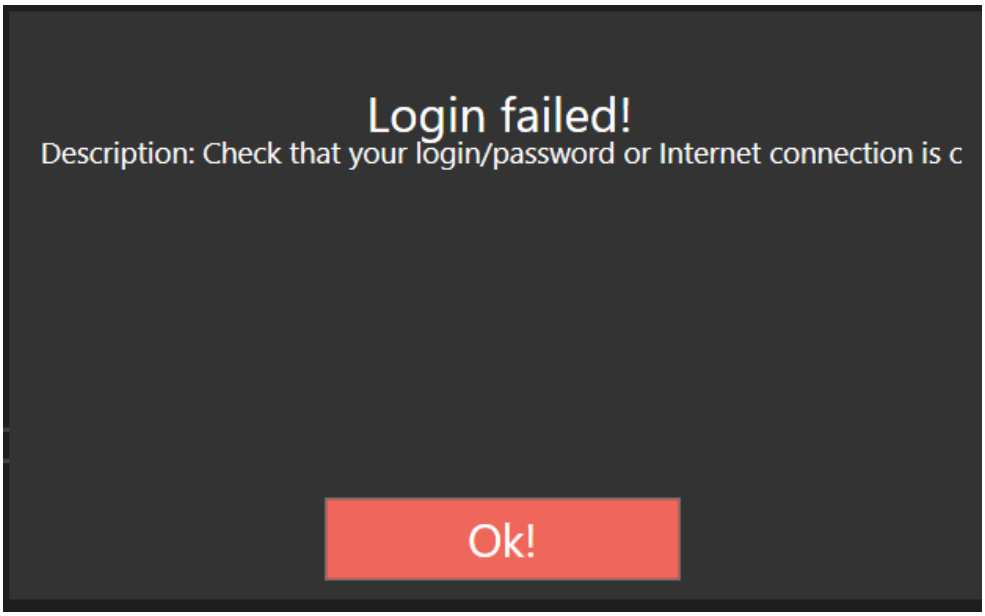
Fontos, hogy itt fontos szerepet kap a Client-nél létrehozott Chat.cs is és az abban található metódusok. Ezért azt is kielemzem. De vegyük szépen sorjában a lehetőségeket.

Az *Auth* ablakban található bejelentkezési működés a *Chat* osztály metódusainak felhasználásával történik. Amikor a felhasználó megadja a felhasználónevét és jelszavát, majd rákattint a "Login" gombra, az *Auth* osztály *Login_Button_Click* eseménykezelője hívódik meg.

Ez az eseménykezelő először ellenőrzi, hogy a felhasználó be van-e jelentkezve a szerverre. Ehhez meghívja a *Chat.LogIn* metódust, átadva neki a megadott felhasználónevet és jelszót. A *LogIn* metódus továbbítja ezeket az adatokat a szerver felé, majd várakozik a válaszra.

A szerver válaszát az Auth osztály *RecieveCommand* metódusa dolgozza fel. Ez a metódus a Chat osztály *CommandsUdp Receive* metódusát használja, hogy fogadjon egy üzenetet a szerverről. A CommandsUdp egy UdpClient típusú objektum. Az UdpClient lehetővé teszi UDP csomagok küldését és fogadását a hálózaton. A CommandsUdp objektumot ebben a specifikus alkalmazásban a szerverrel történő kommunikációra használom, például a kliens parancsainak és válaszainak küldésére és

fogadására A kapott üzenetet JSON formátumban várja, és azt próbálja deszerializálni egy ClientCommand objektumba. Ha ez sikeres, akkor a *Params* tulajdonságba menti az üzenet paramétereit, amelyek között lehet az üzenet típusa és egyéb információk. Visszatérve az *Auth* osztály *Login_Button_Click* eseménykezelőjéhez, ha a bejelentkezés sikeres volt, azaz a *LogIn* metódus *true* értéket adott vissza, akkor az *Auth* ablak *Success* metódusát hívja meg. Ez a metódus a felhasználót átirányítja a fő alkalmazásablakhoz. **(lásd 2.12. ábra)**




**2.12. ábra** – Login failed hibaüzenet

Ha a bejelentkezés sikertelen volt, akkor a felhasználót értesíti egy hibaablakkal, hogy ellenőrizze a megadott adatokat vagy az internetkapcsolatot. Ha a szerver nem érhető el vagy más hiba történik a bejelentkezés során, akkor szintén egy hibaüzenet jelenik meg.

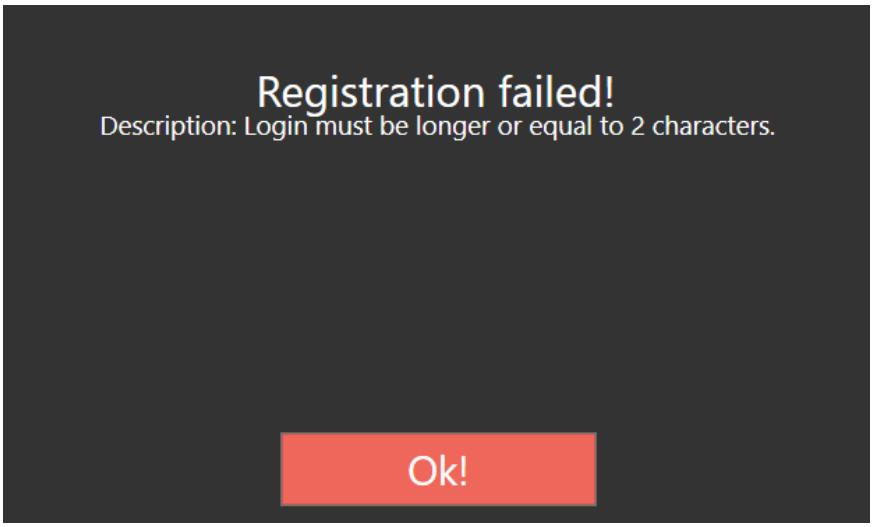
A *Login_Button_Click* metódus a felhasználó által az Auth ablakban végzett műveletek kezeléséért felelős. Amikor a felhasználó rákattint a "Login" vagy "Register" gombra, ez a metódus aktiválódik

Először is, a metódus beolvassa a beviteli mezőkből a felhasználó által megadott adatokat, mint például a bejelentkezési név, jelszó, becenév és e-mail cím.

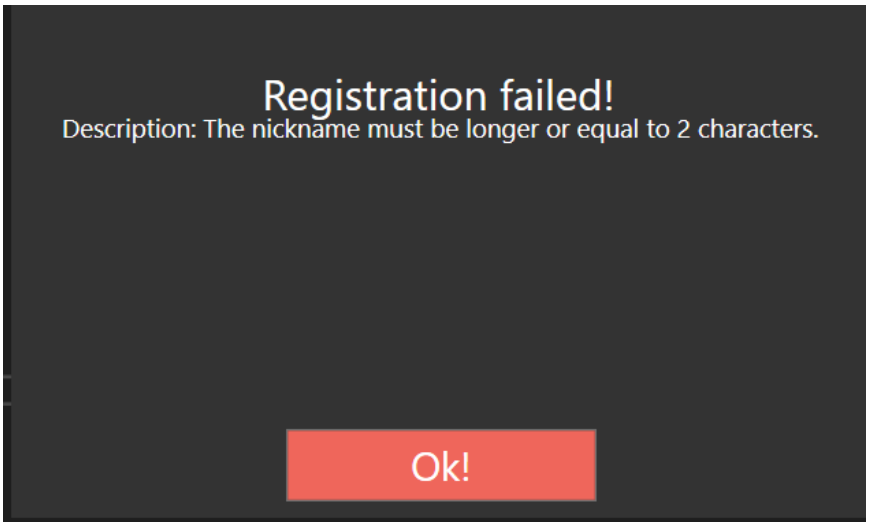
Ezután a metódus ellenőrzi, hogy a felhasználó éppen bejelentkezni próbál-e (*IsLoggining* igaz értékkel). Ha igen akkor ellenőrzi, hogy a felhasználó azt kérte-e, hogy a bejelentkezési adatokat mentse (*WillSave*). Amennyiben, ha azt kérte, akkor a bejelentkezési adatokat titkosítva elmenti a "user.data" fájlba. Amennyiben a felhasználó nem kéri a bejelentkezési adatok mentését, akkor ellenőrzi, hogy létezik-e "user.data" fájl, és ha igen, akkor törli azt. És végül, ha a bejelentkezési név és jelszó

mezők nem üresek, megpróbál bejelentkeztetni a megadott adatokkal a *Chat.LogIn* metódus segítségével.

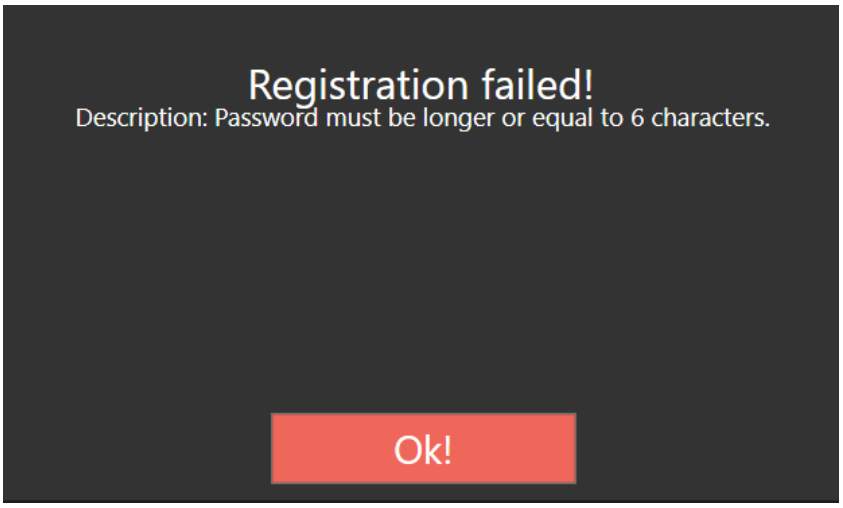
Ha a felhasználó regisztrálni próbál (*IsLoggining* hamis értékkel), akkor a metódus ellenőrzi, hogy a megadott adatok megfelelő hosszúságúak-e, és hogy az e-mail cím tartalmaz-e "@" karaktert. És ha minden feltétel teljesül, akkor a *Chat.Register* metódust hívja meg a megadott regisztrációs adatokkal, és megjeleníti az eredményt egy értesítő ablakban. **(lásd 2.13. ábra), (lásd 2.14. ábra), (lásd 2.15. ábra)**




**2.13. ábra** – Login (felhasználónév mező) hibaüzenet




**2.14. ábra** – Nickname/Displayname hibaüzenet




**2.15. ábra** – Password mező hibaüzenet

Végül, ha minden sikerült, és a felhasználó sikeresen be tudott jelentkezni akkor maga a kommunikációra is alkalmas Main rész betölt, azaz az alkalmazás azon része, ami a beszélgetést is lefolytatja, röviden tömören.

### 2.2.2. Chat ablak (Main)

Ez az ablak sok kisebb funkciót is megvalósít. A legelső dolog, amivel találkozik egy adott felhasználó az egy Start_Page nevezetű Panel. Ez mindössze azt a célt szolgálja, hogy a felhasználó itt tudja kiválasztani azt a Servert, amelyre fel akar csatlakozni, vagy van egy LOG OUT nevezetű gomb, amivel az adott felhasználó ki tud jelentkezni, és akár új néven tud regisztrálni, vagy ha van meglévő fiókja akkor be tud jelentkezni is. Amennyiben kiválasztotta a *Server*-t a felhasználó, akkor ez a Panel eltűnik, azaz *Hidden* állapotba vált. Ezután jelenik meg lényegében a Kliens igazi része. **(lásd 2.16. ábra)**


**2.16. ábra** – Server választó

Jobb oldalt található egy *Users* lista, ezt a szervernél említettem már korábban. Itt az összes, az adott szerveren megtalálható felhasználót kilistázza, szépen egymás alá. Úgy van megoldva a nevek rendezése, hogy a csatlakozás időpontja határozza meg azt, hogy ki hol helyezkedik el a listában. Azaz időrendben az első a legfelső, és utána a többi felhasználó.

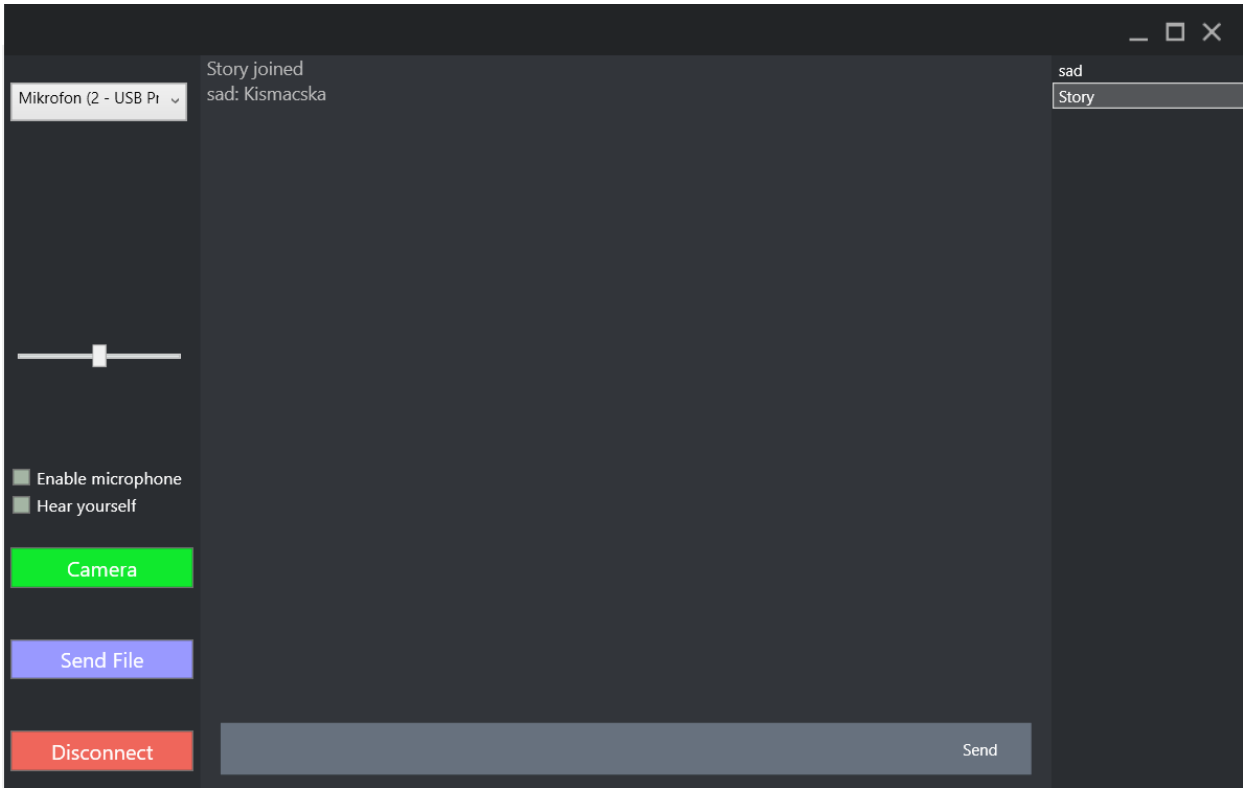
Továbbá található a felületen egy *History* nevezetű TextBox. Ez arra szolgál, hogy a kliens megnyitása óta, a szerveren történt üzeneteket megjelenítse. Azaz, ha például User1 elküld egy üzenetet, aztán csatlakozik User2 akkor nem látja User2 a User1 által kiküldött üzenetet. Természetesen fontos megjegyezni, hogy a *Users* listában kilehet választani adott felhasználót és célzottan neki is lehet küldeni üzenetet, ilyen esetben csak a két fél látja azt az adott üzenetet, és senki más. Ha nincs kiválasztva senki, vagy ha saját magunkat választjuk ki akkor alapértelmezetten broadcastbe megy ki az üzenet, azaz minden egyes szerveren található felhasználó megkapja az üzenetet. A *Server* ablakában csak annyi látható, hogy egy TextMessage nevezetű Command kimegy, azaz üzenetváltás történt a szerveren.

A *History* alatt található egy *Message* nevezetű TextBox és hozzá egy *Send* feliratú Button azaz gomb. A *Message*-be betudunk írni egy általunk írt bármilyen üzenetet, és a *Send* hatására kitudjuk küldeni. Röviden tömören ennyi a funkciója,
Bal oldalt található egy pár dolog, menjünk szépen sorba.

Legfelül látható egy kis lenyíló lista, itt az összes a számítógépünkhöz

csatlakozó bemeneti voice eszköz, azaz mikronok vannak kilistázva. Itt ki tudjuk választani, hogy melyiket szeretnénk használni a jövőben.

**(lásd 2.17. ábra)**




**2.17. ábra** – Kliens ablak és egy üzenet

Alatta található egy kis csúszka azaz Slider ami azt a feladatot tölti be, hogy a hangerőt tudjuk szabályozni vele. Ez a hang kommunikáció során lesz fontos.

Ehhez kapcsolódik kettő kis CheckBox is, ami azt szabályozza, hogy a bemeneti voice eszköz be legyen kapcsolva. Két opció van. Van egy, ami az *Enable microphone*, ez ténylegesen csak bekapcsolja a mikrofont és ezáltal a másik kliensben/kliensekben lehet hallani az adott felhasználó hangját. Ilyenkor mi magunk nem halljuk a saját hangunk. És van egy olyan opció is ami a *Hear yourself* amely azt teszi lehetővé, hogy ha bepipáljuk akkor azon felül, hogy engedélyezi a mikrofont, azon túl saját magunk halljunk a hangunkat, azaz tudunk egy egyfajta önellenőrzést is tartani, hogy minden rendben van-e. Ilyenkor más kliensek felé nem megy át a hang, ami azt jelenti, hogy senki más nem hallja a mi lágy hangunkat. Ezen két lehetőségnél merül fel a Slider haszna, hiszen tudjuk állítani, hogy a hang, milyen hangerővel menjen át a másik kliens felé.

Az utolsó három fő dolog az ablakban az három darab Button azaz gomb. Az egyik a *Camera* amely a *CameraWindow*-ot megnyitja, erre később visszatérek még.

A *Send File* egy lehetséges továbbfejlesztési lehetőség lenne, ami a kliensek közti fájlküldést tenné lehetővé, tervben volt az implementálása, de végül nem csináltam meg. Az utolsó a *Disconnect,* ami nemes egyszerűséggel lecsatlakoztatja a Kliens a szerverről teljesen. Az itt tárgyalt dolgokat a következőkben részletesebben kifejtem, hogy hogyan is működnek valójában.

A *Window_Loaded* metódusban történik a kezdeti inicializáció néhány fontos részletének beállítása. **(lásd 2.18. ábra)**

```
void Window_Loaded(object sender, RoutedEventArgs e)
{
    MyWO = new DirectSoundOut(Library.Latency);
    MyBuffer = new BufferedWaveProvider(Library.SoundFormat);
    MyVolume = new VolumeSampleProvider(MyBuffer.ToSampleProvider());
    MyBuffer.BufferLength = 65536;
    MyWO.Init(MyBuffer);
    MyWO.Init(MyVolume);
    MyWO.Play();

    new Thread(GetServers).Start();
    Chat.InitVoice();
}
```

**2.18. ábra** – Kliens ablak Window_Loaded metódusa

Egy *MyWO* objektumot hoztam létre a *DirectSoundOut* osztály segítségével a hang lejátszásához. Ez az osztály lehetővé teszi a hang lejátszását a DirectSound API-n keresztül, és az *Library.Latency* értéket használja a késleltetés beállításához.

Ezután létrejön a *MyBuffer* objektum a *BufferedWaveProvider* osztály segítségével, amely a hangbuffert kezeli. A *Library.SoundFormat* segítségével inicializálódik, ami a hang formátumát tárolja. A hangbuffer olyan adatszerkezet, amely lehetővé teszi a hangminták gyors és hatékony kezelését és átvitelét. Azt tárolja, hogy a hangot hogyan kell lejátszani vagy továbbítani a hangkártyának, vagy a hálózati csatlakozónak.

A hang lejátszásának és keverésének beállítása történik a következő sorokban. Az *Init* metódusmeghívásával inicializálódnak az előzőleg létrehozott objektumok. A *MyWO* objektumhoz a *MyBuffer,* majd a *MyVolume* objektumok inicializálása történik.

Végül pedig a *MyWO* objektum lejátszását elindítom a *Play* metódus meghívásával, és egy új szál indul el a *GetServers* metódus végrehajtására, valamint a *Chat.InitVoice()* metódus inicializálja a hang funkciókat.

A korábban említett voice communication miatt jött létre a *LoadMicros* nevű metódus. A *LoadMicros* metódus felelős a mikrofonok listájának frissítéséért. Első lépésként a *Microphones* elem tartalmát kiüríti, majd meghatározza a rendszerben

elérhető mikrofonok számát a *WaveIn.DeviceCount* segítségével. Egy *for* cikluson keresztül iterál végig az összes mikrofonon, és az adott mikrofon tulajdonságait lekérve (*WaveIn.GetCapabilities*) hozzáadja azokat a *Microphones* listához. Végül az alapértelmezett kiválasztott mikrofont beállítja, és eltárolja az azonosítóját (*DeviceID*).

A *Send_Click* metódus a "Küldés" gomb eseménykezelője. Első lépésben lekéri a felhasználó által beírt üzenetet a *Message.Text* tulajdonságból, majd ellenőrzi, hogy az üzenet hossza nagyobb-e nullánál, és tartalmaz-e valamilyen szöveget. Ha az üzenet első karaktere "/", akkor ellenőrzi, hogy a parancs alakú-e, és ha igen, akkor kiemeli a parancsot a szövegből. Ha az üzenet nem parancs, akkor ellenőrzi, hogy a szöveg hossza nem nagyobb-e a maximálisan engedélyezett üzenet hosszánál (*MaxMessageLength*). Ha minden ellenőrzés sikeres, akkor továbbítja az üzenetet a *Chat.SendTextMessage* metódusnak, majd törli a *Message.Text* tartalmát.

Az *AddUser* metódus felelős új felhasználó hozzáadásáért a szerver felhasználói listájához. Első lépésben létrehoz egy *User* objektumot a kapott *ShortUserInfo* paraméter alapján. Ezután végigmegy a *ServerUsers* listán, és ellenőrzi, hogy a felhasználó már szerepel-e benne. Ha nem, akkor inicializálja a felhasználó hangját (*u.WO.Init(u.Buffer)* és *u.WO.Init(u.Volume)*), beállítja a hangerejét (*u.Volume.Volume = Volume*), majd hozzáadja a felhasználót a *ServerUsers* listához és a *Users* listához is.

Az *AddUser_async* és *RemoveUser_async* metódusok aszinkron módon adják hozzá illetve távolítják el a felhasználókat a kliens alkalmazásból. Az *AddUser_async* függvény a *Dispatcher.Invoke* segítségével hívja meg az *AddUser* metódust, mely felelős az új felhasználó hozzáadásáért. Ez biztosítja, hogy az operáció a felhasználói felület szálán (UI thread) fusson, így garantálva a megfelelő UI frissítést. Hasonlóképpen, a *RemoveUser_async* is az *RemoveUser* metódust hívja meg aszinkron módon az eltávolításhoz, biztosítva a helyes UI frissítést a *Dispatcher.Invoke*-en keresztül.

A *CommandsReceiver* metódus egy végtelen ciklust indít el, amely folyamatosan figyeli és kezeli a szerverről érkező parancsokat. Először frissíti a *Users* listát az összes szerveren található felhasználóval. Ezután végigmegy a beérkező parancsokon, és azok típusától függően megfelelően kezeli őket. Például, ha egy új felhasználó csatlakozik (*OnUserConnect*), akkor az *AddUser_async* metódus hívódik meg az új felhasználó adataival. Ha egy felhasználó lecsatlakozik (*OnUserDisconnect*), akkor pedig a *RemoveUser_async* metódus hívódik meg a megfelelő felhasználó adataival. Emellett a szöveges üzeneteket is kezeli, és megjeleníti azokat a

felhasználóknak.

A *VoiceReceiver* metódus felelős a hangüzenetek fogadásáért és kezeléséért. Ez a függvény egy végtelen ciklusban fut, és folyamatosan figyeli a hangüzenetek érkezését a szerverről. Amikor érkezik egy hangüzenet, a függvény először eltávolítja az üzenet hosszát jelző első bájtot, majd XML deszerializálással kinyeri a VoiceCommand objektumot az üzenetből. Ezután kivonja a hangmintákat tartalmazó részt a bájttömbből, és végigmegy az összes szerveren található felhasználón, hogy megkeresse az üzenet küldőjét. Ha megtalálja a megfelelő felhasználót, akkor hozzáadja a hangmintákat a felhasználó hangbufferéhez.

A ShowNewMessage metódus lépésről lépésre hajtja végre az üzenetek megjelenítését a *History* TextBox-ban. Először is, amikor meghívódik ez a metódus, akkor a meghívó átadja neki az új üzenetet szöveg formájában. Ez az üzenet az aktuális beszélgetés részét képezi, és meg kell jeleníteni a felhasználónak. A metódus ezután hozzáadja az új üzenetet a *History* TextBox-hoz, azaz a szövegdobozhoz, amely a korábbi üzeneteket tartalmazza. Ezzel a frissítéssel az új üzenet azonnal látható lesz a felhasználó számára. Fontos azonban, hogy az ablak tartalma automatikusan görgetődjön az új üzenet alá, így a felhasználó mindig az utolsó üzenetet látja. Ez a görgetési művelet biztosítja, hogy az új üzenet mindig azonnal észrevehető legyen, még akkor is, ha a korábbi üzenetek között kell görgetni az ablakban. A ShowNewMessage metódus tehát a felhasználóval való kommunikáció során felmerülő üzenetek megjelenítését és navigálását teszi lehetővé a *History* TextBox-ban. A ShowNewMessage_async metódus pedig az *Dispatcher.Invoke* segítségével hívja meg a S*howNewMessage* metódust az UI szálán, biztosítva ezzel a helyes UI frissítést. Ez különösen fontos azokban az esetekben, amikor a metódus a UI szálán kívülről, például egy másik szálról kerül meghívásra. **(lásd 2.19. ábra)**

```
void ShowNewMessage(string text)
{
    History.AppendText(text);
    History.ScrollToEnd();
}

///<summary> Adding a message to history </summary>
2 references
void ShowNewMessage_async(string text)
{
    Dispatcher.Invoke(new ShowNewMessageDelegate(ShowNewMessage), text);
}
```

**2.19. ábra** – ShowNewMessage metódus

Az *InputVoice_Checked* metódus a mikrofon bekapcsolásáért felelős. Amikor a felhasználó bekapcsolja a mikrofont az alkalmazásban, ez a metódus hívódik meg. Ennek az az első lépése, hogy elindítja a hangrögzítést a *StartRecording* metódus segítségével.

A *StartRecording* metódus célja a mikrofon beállítása és a hangrögzítés indítása. Először is beállítja a *InputEnabled* flaget, jelezve, hogy a mikrofon aktív állapotban van. Ezután inicializál egy új *WaveInEvent* objektumot (WI), amelyet a hangrögzítéshez használ. Beállítja a megfelelő hangformátumot a *WaveFormat* segítségével, majd hozzárendeli a *DataAvailable* eseménykezelőt, amely meghívódik, amikor új adatok érkeznek a mikrofonból. A *DeviceNumber* tulajdonságot az aktuális mikrofon kiválasztott eszközének indexével állítja be, amelyet a *DeviceID* változó tárol. Végül elindítja a hangrögzítést a *StartRecording* metódussal.

Az *InputVoice_UnChecked* metódus a mikrofon kikapcsolását kezeli. Amikor a felhasználó kikapcsolja a mikrofont az alkalmazásban, ez a metódus hívódik meg. Ennek a lépése, hogy leállítja a hangrögzítést a *StopRecording* metódus segítségével.

A *StopRecording* metódus ehhez kapcsolódik, ez a hangrögzítés leállításáért felelős. Egyszerűen csak kikapcsolja a *WaveInEvent* objektumot, így megszakítva a hangrögzítést.

Végül pedig, a *Voice_Input* metódus a mikrofonból érkező hang feldolgozásáért felel. Amikor új hangadatok érkeznek a mikrofonból, ez a metódus hívódik meg. Először is megvizsgálja, hogy a "ListenToMyVoice" zászló be van-e kapcsolva. Ha igen, akkor hozzáadja a hangot a saját helyi bufferhez (*MyBuffer*), ami azt jelenti, hogy a felhasználó saját hangját is hallhatja. Ezután továbbítja az érkező hangadatokat a Chat.SendVoice metódusnak, amely elküldi azokat a másik félnek.

Az *Exit* metódus felelős az alkalmazásból való kilépésért. Először ellenőrzi, hogy a felhasználó jelenleg egy szerveren van-e bejelentkezve (*IsOnServer* változó). Ha igen, leállítja a kommunikációt a szerverrel a *Chat.Stop* hívásával. Ezután ellenőrzi, hogy van-e aktív hangrögzítés (WI változó). Ha van, lezárja a hangrögzítést, majd nullázza a hivatkozást az objektumra, hogy felszabadítsa a memóriát. Végül megszakítja az alkalmazás futását a *Environment.Exit(0)* hívásával.

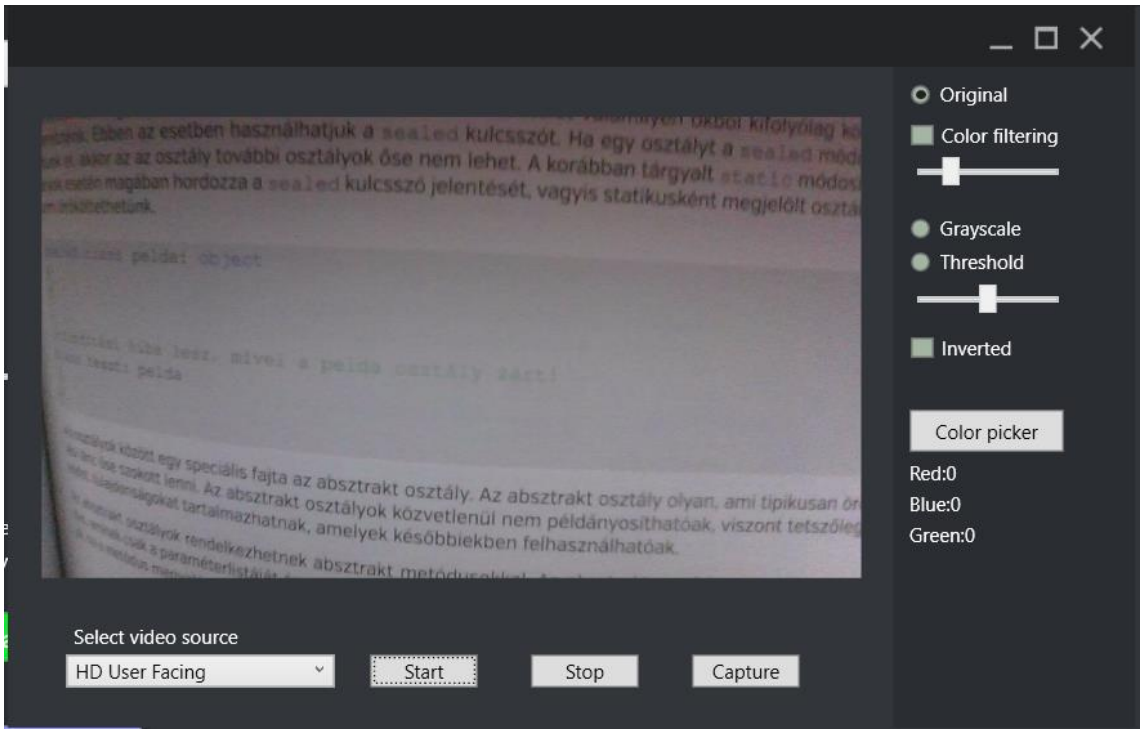
Az *Environment.Exit(0)* hívás egy olyan utasítás, amely azonnal leállítja az alkalmazás futását, és kilép a programból. A paraméterként átadott 0 érték azt jelzi, hogy a program normál módon fejeződött be, nincsenek hibák vagy kivételek, amelyek miatt le kellene állítani az alkalmazást.

A *Disconnect_Click* metódus kezeli a "Disconnect" gomb kattintását, ami a felhasználót kilépteti a jelenlegi szerverről. Először beállítja a *IsOnServer* flaget hamisra, jelezve, hogy a felhasználó nincs többé a szerveren. Majd leállítja a kommunikációt a szerverrel a *Chat.Stop* hívásával. Ezt követően megszakítja a parancs- és hangvételi szálakat, hogy megállítsa a kommunikációt a szerverrel. Végül elindít egy animációt, amely átmenetileg csökkenti a kezdőlap méretét és láthatóságát, hogy visszatérjen a kezdőállapotba.

A *Camera_Window_Show* metódus akkor hívódik meg, amikor a felhasználó rákattint a "Camera" gombra, hogy megnyissa a kameraablakot. Ez létrehoz egy új *CameraWindow* objektumot, majd megjeleníti azt a felhasználónak. Ezáltal lehetővé teszi a felhasználó számára, hogy megnyissa a kameraablakot és használja azt a video kommunikációhoz. Amit a bevezetőben is említettem a video kommunikáció nem egészen úgy van megoldva ahogyan eredetileg szándékoztam volna. Térjünk is rá a kamerára.

### 2.2.3. Camera

A *CameraWindow* egy olyan alkalmazás vagy alkalmazásrész, amely lehetővé teszi a felhasználóknak, hogy a saját kamerájuk segítségével élőképet nézzenek meg, és különböző képszűrőket alkalmazzanak rá. A felhasználói felületen láthatóak a kamera által rögzített képek, a videóeszközök listája, a kezelőgombok a kamera indításához és leállításához, valamint a képek manipulálását lehetővé tevő szűrők és beállítások. A háttérben lévő kód felelős a képek feldolgozásáért és megjelenítéséért, például a képek konvertálásáért a *Bitmap* és a *BitmapImage* formátumok között, valamint a pixeladatok kezeléséért és szűrők alkalmazásáért. A különféle metódusokon és lehetőségeken végig megyek, hogy érthetőbb legyen. **(lásd 2.20. ábra)**




**2.20. ábra** – Camera ablak

Több olyan metódus van, amely a *CameraWindow* osztályban található tulajdonságokat (properties) vezérlik és kezelik. Minden tulajdonság egy adott képszűrő vagy képmanipulációs funkció állapotát vagy beállításait tárolja.

Például, a *Original* tulajdonság azt tárolja, hogy az eredeti kép látható-e vagy sem. Ha ez a tulajdonság *true* értékre van állítva, akkor az eredeti, a kamera által rögzített kép jelenik meg a felhasználói felületen. Ha *false* értékre van állítva, akkor más képszűrők vagy manipulációs funkciók alkalmazhatóak, és a képernyőn megjelenő kép ennek megfelelően változik.

Hasonlóan, a *Grayscaled* tulajdonság azt tárolja, hogy a felhasználó a szürkeárnyalatos szűrőt alkalmazta-e a képre vagy sem. Ha ez a tulajdonság *true* értékre van állítva, akkor a kép szürkeárnyalatos lesz, ellenkező esetben nem. **(lásd 2.21. ábra)**


**2.21. ábra** – Grayscale beállítás

Minden tulajdonsághoz tartozik egy getter és egy setter metódus. A getter metódus lekérdezi a tulajdonság aktuális értékét, míg a setter metódus beállítja az értéket, és gondoskodik arról, hogy az érték változásakor kiváltódjanak az események (például az OnPropertyChanged esemény), ami lehetővé teszi a felhasználói felület frissítését az új értékek alapján.

Ezek a tulajdonságok és az azokhoz tartozó metódusok együtt biztosítják, hogy a CameraWindow osztály megfelelően reagáljon a felhasználói interakciókra és a képszűrők beállításainak változásaira, és megfelelően frissítse a felhasználói felületet az aktuális képstátusz alapján.

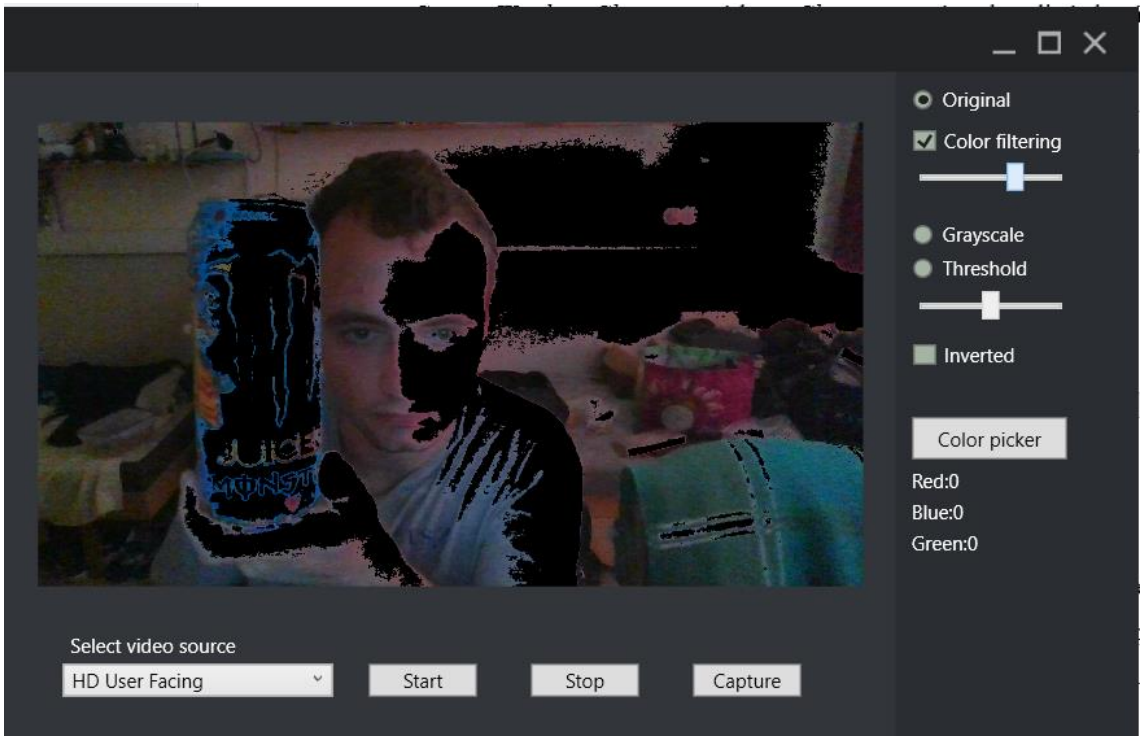
A konstruktor létrehozza az ablakot, beállítja annak DataContext-jét, inicializálja a videóeszközöket, alapértelmezett értékeket állít be az egyes képszűrők és paraméterek számára, és hozzárendeli az ablak bezárásának eseménykezelőjét.

Az eseménykezelő metódusok (pl. *CameraWindow_Closing, btnStart_Click, btnStop_Click, video_NewFrame*) az ablak eseményeire válaszolnak. Például a *CameraWindow_Closing* metódus a *Closing* eseményt kezeli, és bezárás esetén leállítja a kamerát és visszaállítja az egérmutatót.

A *video_NewFrame* metódus felelős a videoframe-ek fogadásáért és feldolgozásáért. Ez a metódus akkor fut le, amikor a kamera új képkockával frissíti a képet, és a megjelenített kép frissítéséért felel.

Először a metódus létrehoz egy *BitmapImage* objektumot, amelyet a videókép megjelenítésére fogunk használni. Ezután klónozza a kapott képkockát, hogy ne módosítsa az eredeti képet, majd elkezdi megvizsgálni, hogy melyik képszűrő van bekapcsolva (*ColorFiltered, Grayscaled, Thresholded*).

Ha a *ColorFiltered* be van kapcsolva, akkor az algoritmus egy EuclideanColorFiltering szűrőt alkalmaz a képre a felhasználó által kiválasztott szín alapján. **(lásd 2.22. ábra)**




**2.22. ábra** – Color filtering

Az EuclideanColorFiltering egy olyan szűrő, amely az euklideszi távolság alapján válogatja ki azokat a pixeleket a képen, amelyek megfelelnek a megadott színhez képest. A szűrő lényegében azt határozza meg, hogy a pixel színe milyen messze van egy adott referencia színtől az RGB (vagy RGB és YCbCr) színtérben.

A szűrő működése során a referencia szín és a pixel színe közötti euklideszi távolságot számolja ki az RGB színtérben. Ez a távolság a színek közötti "távolságot" méri a térben, amely magában foglalja a vörös, zöld és kék csatornák értékeit. Minél nagyobb az euklideszi távolság, annál nagyobb a színkülönbség.

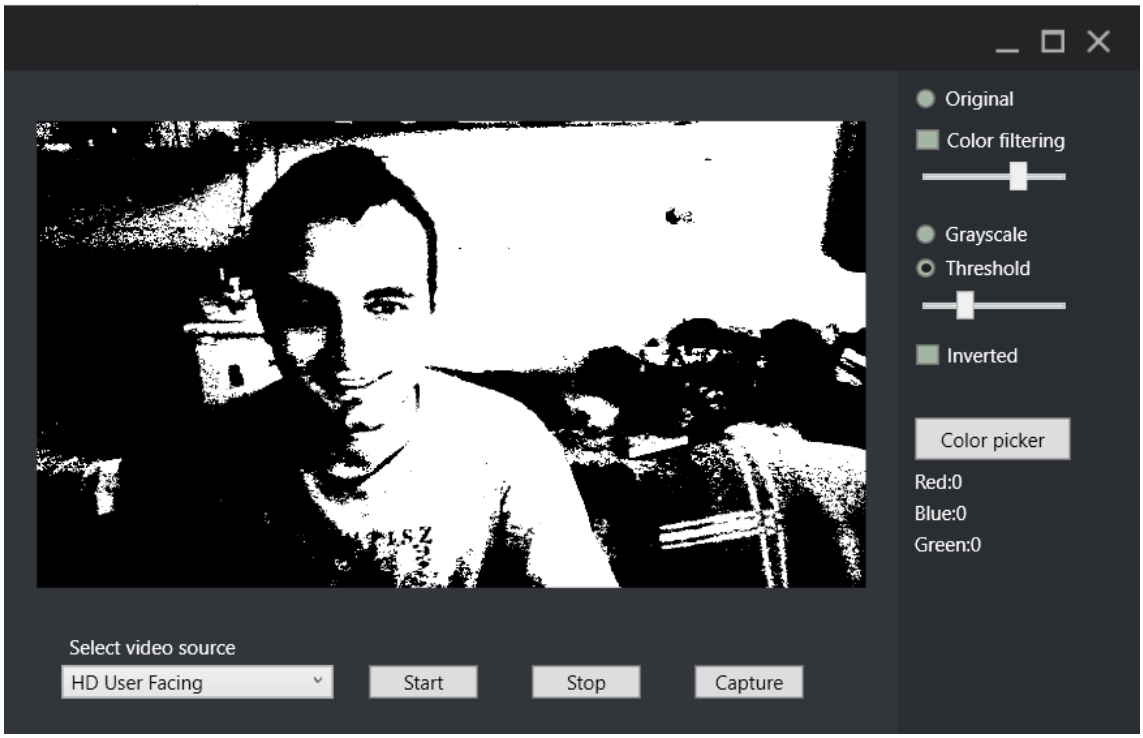
Az algoritmus ezt a távolságot összehasonlítja egy beállított küszöbértékkel. Ha a pixel színeinek euklideszi távolsága kisebb, mint a küszöbérték, akkor a pixel megfelel a szűrő kritériumának, és bekerül az eredményképbe. Ellenkező esetben nem kerül be az eredményképbe, tehát "kitörlődik".

Ez az eljárás lehetővé teszi például egy adott szín vagy színtartomány kiválasztását a

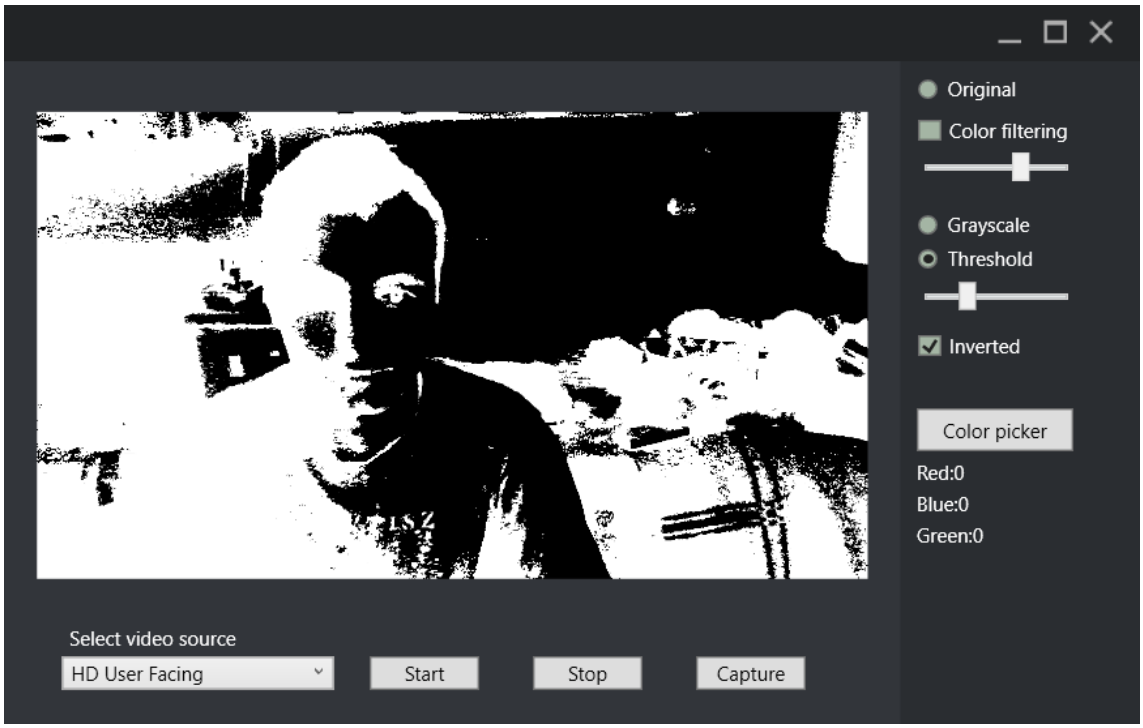
képen, és más színek elutasítását, ami hasznos lehet például objektumok vagy alakzatok szegmentálására vagy kiemelésére. Én itt alakzatok kiemelésére használtam. Ezt majd a megfelelő kódrészletnél természetesen elmondom.

Visszatérve a lehetőségekre, ha a *Grayscaled* opció van bekapcsolva, akkor a képre egy Grayscale szűrőt alkalmaz, ami átváltja a színes képet szürkeárnyalatosra.

És végül ha a *Thresholded* van bekapcsolva, akkor a képre egy küszöbölő szűrőt alkalmaz a felhasználó által beállított küszöbérték alapján, ami megkülönbözteti a világos és sötét pixeleket. **(lásd 2.23. ábra), (lásd 2.24. ábra)**




**2.23. ábra** – Threshold




**2.24. ábra** – Threshold Inverted opcióval, felcseréli a feketét a fehérrel és fordítva

Miután a megfelelő képszűrőt kiválasztottuk az algoritmus beállítja a *BitmapImage* objektumot az új kép forrásaként, majd a *videoPlayer* nevű *Image* objektumhoz rendeli ezt az új képet, hogy megjelenítse azt a felhasználónak. Ezáltal a felhasználó a kamera képét látja az ablakban a kiválasztott képszűrőkkel vagy az eredeti formájában, attól függően, hogy mik vannak bekapcsolva.

Van egy *videoPlayer_MouseEnter* metódus is. Ez a metódus akkor fut le, amikor a felhasználó az egerét belép a videolejátszó (videoplayer) felületére. Ha éppen a színválasztó módban van, akkor megváltoztatja az egér mutatóját az alkalmazásban előre definiált színválasztó mutatóra. Ehhez egy korábban létrehozott XAML erőforrást (*ResourceDictionary*) használ, amely definiálja a színválasztó mutatót. Ebben a fájlban az *x:Key* attribútummal van hivatkozva a *CursorPicker* objektumra, amely a mutatót tartalmazza. A *CursorPicker* objektum a *picker.cur* fájlra hivatkozik, ami maga a színválasztó mutatót (pipettát) tartalmazza. Ezáltal, amikor a felhasználó belép a videolejátszó felületére, és aktív a színválasztó mód, az egér mutatója megváltozik a pipetta mutatóra, ami lehetővé teszi a felhasználó számára, hogy színeket válasszon ki a képből és azt az ablak jobb oldalán megjelenítse szépen lebontva, hogy az adott pixel mennyi pirosat, kéket és zöldet tartalmaz.

A *videoPlayer_MouseLeave* metódus a videolejátszó felületéről való kilépéskor fut le. Ez a függvény egyszerűen visszaállítja az egér mutatóját az alapértelmezettre,

ami ugyebár egy nyíl.

A *videoPlayer_PreviewMouseLeftButtonDown* eseménykezelő akkor fut le, ha a bal egérgombal kattintunk a videolejátszó felületén. Ha éppen színválasztó módban van, akkor ez a függvény meghatározza az egér helyzetére kattintva a kiválasztott képpont színét, és beállítja a *Red, Green* és *Blue* változókat ennek megfelelően. Ezután kikapcsolja a színválasztó módot, és visszaállítja az egér mutatóját az alapértelmezettre.

A *FindCorners* függvény feladata a képen található sarkok megtalálása. Ehhez az összes kiválasztott szűrőt alkalmazza a képre, majd a BlobCounter osztállyal megtalálja a képen lévő objektumokat, és ezekből a sarkokat. Végül az így megtalált sarkokat visszaadja a feldolgozó függvénynek.

A *PaintCorners* függvény a megtalált sarkok között kék vonalakat rajzol a megadott bitmapre. Ez segít megjeleníteni a felhasználónak a megtalált sarkokat a képernyőn.

A *ToPointsArray* metódus egy *List<IntPoint>* objektumból készít egy .NET pontok tömbjét (*System.Drawing.Point[]).* Ennek a metódusnak a célja, hogy a *AForge.NET* pontok listáját átalakítsa egy *.NET* pontok tömbjévé, hogy azok könnyebben használhatók legyenek az alkalmazásban. A *List<IntPoint>* objektum az *AForge.Imaging* névtérből származik, és *IntPoint* objektumokat tartalmaz, amelyek egész számokból állnak.

A *GetVideoDevices* függvény összegyűjti és listázza a rendelkezésre álló videóeszközöket, és ezeket hozzáadja a *VideoDevices* kollekcióhoz.

A *StartCamera* és *StopCamera* függvények kezelik a kamera indítását és leállítását. A *StartCamera* függvény beállítja a kamera indítását, ha rendelkezésre áll videóeszköz. A *StopCamera* függvény pedig leállítja a kamerát, ha az éppen működik.

Végül pedig, a Camera_Picture_*Capture* metódus végzi a készítést. Ellenőrzi, hogy a kamera működik-e, majd elkészíti a pillanatképet a kamera aktuális képkockájából, és megnyitja a megörökített képet egy új ablakban a mentéshez.
Ehhez kapcsolódik a CreateCameraImage metódus. Ez egy BitmapImage objektum létrehozására szolgál a kamerafelvételből. Ennek a metódusnak a célja, hogy visszaadjon egy képet a kamerafelvételből. Az aktuálisan megjelenített képet vagy a kamera jelenlegi állapotát lehet használni ehhez.

Amennyiben rámegyünk a Capture gombra abban az esetben megnyílik a Save ablak is, a következő részletben ezt tárgyalom ki.

### 2.2.4. SaveCameraPicture

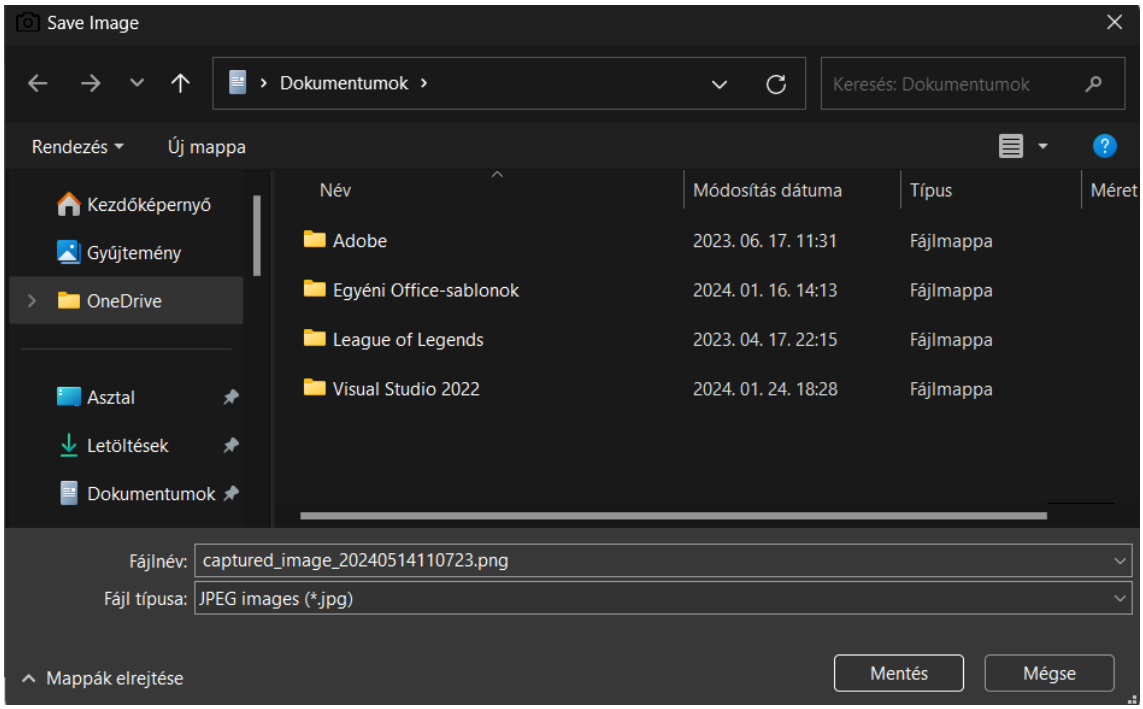
Ez az osztály egy kamerafelvételből készült képet jelenít meg és lehetővé teszi annak mentését. A felhasználó az ablakon kiválaszthatja a kép mentésének formátumát, majd elmentheti azt a megadott fájl elérési útvonalra. **(lásd 2.25. ábra)**


**2.25. ábra** – Save image ablak

Az osztályban lévő *_currentImage* privát adattag egy *BitmapImage* objektumot tárol, amely a jelenleg megjelenített képet reprezentálja.

Az osztály konstruktora egy *BitmapImage* objektumot vár paraméterként, amelyet az osztály létrehozásakor átadhatunk neki. Ez a kép lesz megjelenítve az ablakon. A konstruktorban az *InitializeComponent()* metódus meghívása történik, ami inicializálja az ablakot és annak UI elemeit.

Az osztályban található *SaveButton_Click* metódus felelős a kép mentéséért. Amikor a felhasználó rákattint a "Save" gombra, ez a metódus hívódik meg. Először egy mentési dialógusablakot jelenít meg, ahol a felhasználó kiválaszthatja a kép mentésének formátumát és megadhatja a fájl elérési útvonalát. Ha a felhasználó kiválasztott egy helyet és megadott egy fájlnevet, akkor a *SaveImageToFile* metódust hívja meg a kép mentéséhez. Az alábbi képen ez látható, működés közben. **(lásd 2.26. ábra)**




**2.26. ábra** – Save Image windows-os ablaka

A *SaveImageToFile* metódus felelős a kép tényleges mentéséért. Először a *BitmapImage* objektumot konvertálja egy *Bitmap* objektummá, majd ezután használja a *PngBitmapEncoder* osztályt a kép mentéséhez. A mentési folyamat során a kiválasztott fájl elérési útvonalát használja.

A következő nagyobb fejezetben kitárgyalom, hogyan is kell használni az alkalmazást.

# 3. ÖSSZEGZÉS

Az alkalmazás funkcionalitásában megvalósítja azt, amit a tématerv során kiírtam. Megvalósítok egy Kliens-Server kapcsolatot peer-to-peer módon. A kliensek közti kommunikációra megvan valósítva egy szöveges chat is, emellett a hang alapú kommunikáció is elérhető és tökéletesen működik. A videochat része nem teljesen fedi le a tématervben tárgyaltakat, viszont úgy gondolom, hogy az egyéb funkcionalitásokkal, mint például a szűrések, a képkimentés, ezekkel végül sokkal többet hozzátettem az alkalmazáshoz.

# TOVÁBBFEJLESZTÉSI LEHETŐSÉGEK

Számos továbbfejlesztési lehetőség rejlik még az alkalmazásban.

**Videohívás két kliens között:**

Jelenlegi állapotban az alkalmazás csak egy kliens és egy szerver közötti kommunikációt tesz lehetővé. A fejlesztést jó lenne idővel kiterjeszteni, hogy közvetlenül két kliens között is létrejöhessen a videohívás, peer-to-peer alapon.

Emellett nem csak peer-to-peer módon lehetne megvalósítani az alkalmazást hanem webes környezetben is.

**Csoportos videóhívások:**

Ez szorosan kapcsolódik az első fejlesztéshez. Ez lehetővé tenné, hogy egyszerre több felhasználó kapcsolódhasson be egy hívásba, ami különösen hasznos lehet ha több emberrel szeretnénk egyszerre beszélni.

**Üzenetküldés és fájlmegosztás:**

Bár a szöveges chat funkció már része az alkalmazásnak, további fejlesztésként be lehetne vezetni a fájlmegosztási lehetőséget is, ami a fejlesztés végső stádiumában tervben is volt. Ez lehetővé tenné, hogy a felhasználók gyorsan és egyszerűen osszanak meg dokumentumokat, képeket vagy egyéb fájlokat a hívások során.

**Virtuális háttér és effektek:**

Az alkalmazásban lehetőségként belehetne vezetni virtuális hátteret. Ez különösen hasznos lehet a magánszféra védelme érdekében, illetve professzionális megjelenést biztosíthat üzleti hívások során. A háttér cseréjéhez képfelismerő algoritmusokat lehetne alkalmazni, amelyek felismerik és elkülönítik a beszélőt a háttértől.

**Beépített naptár és hívástervezés:**

A felhasználói élmény javítása érdekében érdemes lenne integrálni egy beépített naptárat, amely lehetővé teszi a hívások előre tervezését és időpontok foglalását.

Akár Google Naptárral is összekötni.

**Képernyőmegosztás:**

A képernyőmegosztás funkció bevezetésével a felhasználók megoszthatnák a saját képernyőjüket másokkal.

**Hang- és videófelvételek készítése:**

A felhasználók számára biztosíthatnánk a hívások rögzítésének lehetőségét, így később visszanézhetik a beszélgetéseket.

# Irodalomjegyzék


[1] Szerző: Ruzsinszki Gábor, Lábadi Henrik
Cím: JSON Serialization – Newtonsoft
Megjelenés: 2024
Url: https://csharptutorial.hu/docs/hellovilag-hellocsharp/12-modern-alkalmazasfejlesztes/json-serialization-newtonsoft/
Utolsó megtekintés: 2024.05.21.

[2] Szerző: Ruzsinszki Gábor, Lábadi Henrik
Cím: TCP/IP adatátvitel
Megjelenés: 2024
Url: https://csharptutorial.hu/docs/hellovilag-hellocsharp/12-modern-alkalmazasfejlesztes/tcp-ip-adatatvitel/
Utolsó megtekintés: 2024.05.21.

[3] Szerző: Ruzsinszki Gábor, Lábadi Henrik
Cím: Az AES szabvány
Megjelenés: 2024
Url: https://csharptutorial.hu/docs/hello-vilag-hello-kriptografia/titkositasi-modszerek/az-aes-szabvany/
Utolsó megtekintés: 2024.05.21.

## Nyilatkozat

Alulírott Bán János programtervező informatikus BSc szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Szoftverfejlesztés Tanszékén készítettem, programtervező informatikus BSc diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Diplomamunka Repozitóriumában tárolja.

dátum, 2024.05.21. aláírás/név

# Köszönetnyilvánítás

Ezúton is szeretném megköszönni a támogatását Alexin Zoltánnak aki szakmai tanácsokkal ellátott és segített, hogy minél minőségibb szakdolgozatot készíthessek.