



EÖTVÖS LORÁND TUDOMÁNYEGYETEM
INFORMATIKAI KAR

Megosztott rajztábla

Témavezető: **Gera Zoltán**

Szerző: **Gaál Péter**

Egyetemi tanársegéd

Programtervező informatikus

Budapest, 2020

SZAKDOLGOZAT / DIPLOMAMUNKA

EREDETISÉG NYILATKOZAT

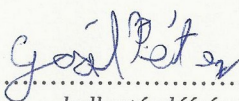
Alulírott **Gaál Péter** Neptun-kód: **XQUW7D**

ezennel kijelentem és aláírással megerősítem, hogy az Eötvös Loránd Tudományegyetem Informatikai Karának, **Programozási Nyelvek és Fordítóprogramok Tanszékén** írt, **Megosztott rajztábla** című szakdolgozatom/diplomamunkám saját, önálló szellemi termékem; az abban hivatkozott szakirodalom felhasználása a szerzői jogok általános szabályainak megfelelően történt.

Tudomásul veszem, hogy szakdolgozat/diplomamunka esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Budapest, 2020. október 12.

.....

hallgató aláírása



EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

SZAKDOLGOZAT-TÉMA BEJELENTŐ

Név: **Gaál Péter**

Neptun kód: **XQUW7D**

Tagozat: **nappali**

Szak: **programtervező
informatikus BSc**

Témavezető neve: **Gera Zoltán**

munkahelyének neve és címe: **ELTE IK 1117 Pázmány Péter
sétány 1/c**

beosztása és iskolai **Tanársegéd, Programtervező**

végzettsége: **Matematikus MSc.**

A dolgozat címe: **Megosztott rajztábla**

A dolgozat témája: A dolgozat célja egy kooperatívan szerkeszthető tábla megvalósítása. A peer-to-peer kliensek kapcsolódnak egymáshoz egyidőben, akik egyidőben rajzolhatnak görbéket, poligonokat, vonalakat, illetve szerkeszthetnek szövegdobozokat, és beilleszthetnek képeket, Stb.. A rajzolások tetszőleges színnel történhetnek. Bármely összekapcsolódott kliensek által a táblán végzett módosítás azonnal megjelenik az összes többi kliens számára. A kliensek elraktározzák a rajzolási parancsokat, így a később bejelentkezett kliensek táblái azonnal szinkronban lesznek a többi bejelentkezett kliensével.

Kérem a szakdolgozat témájának jóváhagyását.

Budapest, 2019.05.24

Tartalomjegyzék

1. Bevezetés.....	1
2. Felhasználói dokumentáció.....	3
2.1.1. Rendszerkövetelmények.....	3
2.2. Telepítés.....	3
2.2.1. GNU/Linux.....	3
2.2.2. Microsoft Windows.....	4
2.3. Eltávolítás.....	4
2.4. Az első indítás.....	5
2.5. Rajzolás.....	5
2.5.1. Váznak.....	5
2.5.2. Vonalak tulajdonságai.....	8
2.6. Állapot alapú rajzolás.....	9
2.7. A hálózat.....	10
2.7.1. Bejövő kapcsolatok fogadása.....	10
2.7.2. Kapcsolódás másik klienshez.....	12
2.7.3. Szinkronizáció.....	13
2.7.4. Hálózati jelszó.....	14
2.7.5. Hálózati kliens fa.....	14
2.7.6. Hálózati Chat.....	16
2.8. Frissítések.....	17
2.9. Összes hibaüzenet listája.....	18
3. Fejlesztői dokumentáció.....	21
3.1. Felhasznált eszközök.....	21
3.1.1. OpenJDK 14.0.1.....	21
3.1.2. Jlink.....	21
3.1.3. OpenJFX 14.0.1.....	22
3.1.4. Gradle 6.6.1.....	22
3.1.5. WaifUPnP és json-simple.....	23
3.1.6. Nullsoft Scriptable Install System.....	24
3.2. A program szerkezete és működése.....	24
3.2.1. Controller réteg.....	25

3.2.1.1	Állapot tároló rendszer.....	27
3.2.1.2	Szabályos alakzatok rajzolása.....	29
3.2.1.3	RemoteDrawLineCommandBufferHandler.....	31
3.2.1.4	Állapotlánc konzisztencia.....	32
3.2.1.5	ChatService.....	32
3.2.1.6	Kliens lista.....	34
3.2.1.7	Kapcsolódási link.....	34
3.2.1.8	TabController.....	35
3.2.1.9	UserID.....	35
3.2.2.	Model réteg.....	36
3.2.2.1	A hálózat működése.....	36
3.2.2.2	NetworkService.....	37
3.2.2.3	Csatlakozás.....	38
3.2.2.4	Jelzések.....	38
3.2.2.5	Szinkronizáció.....	41
3.2.2.6	Pingelés.....	42
3.2.2.7	Újracsatlakozás.....	43
3.2.3.	A View réteg.....	44
3.3.	Tesztelés.....	44
3.3.1.	Tesztesetek.....	46
3.4.	További fejlesztés.....	50
3.5.	Összegzés.....	51
4.	Irodalomjegyzék.....	52

1. Bevezetés

A megosztott rajztábla ötlete akkor fogant meg, amikor néhány hallgatótársammal együtt szerettünk volna megoldani egy analízis feladatot otthonról. Egyikünk sem ismert olyan alkalmazást, amivel közösen egy vászonra rajzolhattunk volna egy időben. Így hát képernyő megosztást kellett használnunk, vagy le kellett valahogy írni karakterekkel az egyenletet. Egyik sem volt elég hatékony. Mindig annak kellett megosztania a saját képernyőjét, aki éppen le akarta írni az egyenlet következő átalakítását. A megosztott képernyő képe a többiekénél gyakran darabos lett vagy szétesett a váltakozó bitráta, és a korlátozott sávszélesség miatt. Szerettem volna egy effektívebb megoldást a problémára.

A megosztott rajztáblát hasonlóan kell elképzelni, mint egy többfelhasználós kooperatív Microsoft Paint-et. Az alkalmazás hibrid peer-to-peer hálózaton keresztül összekapcsolódik más kliensekkel, és a hálózatba kapcsolódott kliensek egyszerre tudnak rajzolni egy, vagy több fehér vászonra tetszőleges vonalat és különböző szabályos alakzatokat. Több vásznat is nyithatnak, amiken párhuzamosan dolgozhatnak, képeket szűrhetnek be a vásznakra, visszavonhatják a módosításaikat, és visszavonhatják a visszavonásokat (Ctrl+Z, és Ctrl+Y vagy Ctrl+Shift+Z mechanizmus). A hálózathoz való csatlakozást korlátozhatják jelszóval, megválaszthatják a rajzoláshoz használt vonalak színét és vastagságát. A hálózaton küldhetnek egymásnak szöveges üzeneteket, pingelhetik egymást, és a routerek mögötti server socketekhez való csatlakozást leegyszerűsíti az UPnP-n keresztül automatikusan konfigurálódó port-forwarding. A hálózat automatikusan szinkronban tart minden módosítást a kliensek képernyőin között, mindezt, a video stream alapú képernyőmegosztáshoz képest, minimális hálózati sávszélesség igény mellett. A megosztott rajztábla ötletének születésekor még nem létezett a Microsoft Whiteboard, ami ugyanezt a problémát oldja meg, bár az asztali alkalmazás verziója csak Microsoft Windows-ra telepíthető. Az alkalmazás fejlesztéséhez OpenJDK 14.0.1-et használtam, így a szoftver platformfüggetlen. Pontosabban minden operációs rendszeren fut, amelyen fut a Java Virtual Machine (JVM) megfelelő verziója. Mivel az OpenJDK 14.0.1-et már nem adták ki 32 bites rendszerekre, ezért az egyik rendszerkövetelmény a 64-bites operációs rendszer megléte.

A fejlesztés során számos nehéz, és néhány egyáltalán nem várt problémára kellett megoldást találni, mint például a szinkronizált állapotalapú visszavonás működésének sajátosságai, a fa struktúrába rendeződött peer-to-peer hálózat koherenciájának megtartása bármely kliens lekapcsolódása esetén, a hálózatba később becsatlakozó kliensek munkamenetének szinkronba hozása, a router tűzfalának megnyitása bejövő kapcsolatok fogadásához. Mindezekre, és még további érdekes problémákra igyekeztem elegáns, és precíz megoldásokat találni, melyeket a felmerült problémákkal együtt a fejlesztői dokumentáció című fejezetben ismertetni is fogok. De előtte részletesen ismertetem a program funkcióit felhasználói oldalról a Felhasználói dokumentáció című fejezetben.

2. Felhasználói dokumentáció

2.1.1. Rendszerkövetelmények

- Operációs rendszer:
 - Ubuntu 16.04 (64-bit) vagy frissebb; egyéb 64-bites GNU/Linux disztribúciókon X11 vagy Wayland fölött.
 - Microsoft Windows 7 vagy frissebb, 64-bites kiadása.
- RAM:
 - **GNU/Linux** esetén a program jellemzően 135-200 MB rendszermemóriát használ, ennyi memóriának mindenképpen szabadon kell lennie a program használatakor. A számítógépnek így javasolt legalább 2 GB RAM-al rendelkeznie.
 - **Microsoft Windows** esetén a program jellemzően 87-150 MB rendszermemóriát használ, ennyinek mindenképpen szabadon kell lennie a program használatakor. A számítógépnek így javasolt legalább 3 GB RAM-al rendelkeznie, a Microsoft Windows-nak a GNU/Linux-hoz viszonyított magasabb rendszermemória igénye miatt.
- Háttértár:
 - **GNU/Linux** esetén legalább 89,9 MB szabad terület.
 - **Microsoft Windows** esetén legalább 70,9 MB szabad terület.

2.2. Telepítés

2.2.1. GNU/Linux

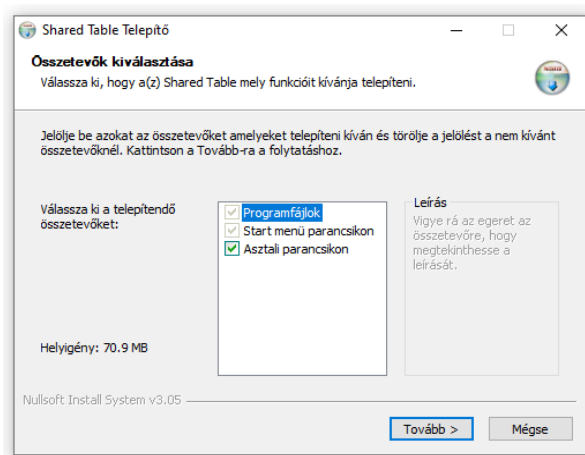
Linux esetében Debian csomaggal (.deb) telepíthetjük az alkalmazást. Debian csomagok telepítéséhez mindenképpen szükség van a root jelszóra. Ubuntu esetében nincs más dolgunk mint kétszer rákattintani a csomagra Nautilus-ban, és az Ubuntu Software alkalmazással már indítható is a telepítés.

Parancssorból való telepítéshez navigáljunk a *sharedtable.deb* fájlt tartalmazó mappába, majd adjuk ki a `sudo apt-get install ./sharedtable.deb` parancsot. A telepítést követően az alkalmazás indítható a *Gnome Applications* menüből, az asztalon elhelyezett parancsikkal, vagy terminálban kiadott *sharedtable* parancssal indítható. Ubuntu esetében az asztalon elhelyezett parancsikon esetében külön engedélyezni kell

az indítást minden új *.desktop* fájl esetében, így itt is. Ezt az asztalon **(nem a fájlkezelőben)** megjelenő ikonra jobb egérgombbal kattintva majd a felugró menüben az *Indítás engedélyezése* opcióra kattintva tehetjük meg. Ekkor eltűnik az ikon végéről a *.desktop* utótag, és működő ikonná válik.

2.2.2. Microsoft Windows

Windowson az itt megszokott futtatható telepítőcsomag áll rendelkezésünkre. Egyszerűen futtassuk az *InstallSharedTable.exe* fájlt a fájlkezelőből. A felugró *Felhasználói fiókok felügyelete* ablakban engedjük meg a telepítőnek, hogy módosításokat hajtson végre a számítógépen és így írhasson a *C:\Program Files* mappába. Ha nem engedélyezi, a telepítő nem indul el! A telepítő program végigvezeti a telepítési folyamatot.



Ábra 1: Telepítés Windowson a telepítőprogram használatával

Várja meg, amíg a telepítés befejeződik, majd zárja be a telepítőt. A sikeres telepítés után, (ha kiválasztotta a telepítésnél) asztali parancsikkal, vagy Start menü parancsikkal futtatható az alkalmazást.

2.3. Eltávolítás

GNU/Linux esetén a program telepítésével és futtatásával létrejött minden fájl eltávolítható a `sudo apt purge sharedtable` parancs kiadásával.

Microsoft Windows esetén szintúgy törölhető minden az alkalmazáshoz kapcsolódó fájl a Start Menü SharedTable mappájában található *Uninstall* parancsikon kiválasztásával. Ekkor elindul a program eltávolítását végző szoftver. Csakúgy, mint a

telepítésnél, itt is engedélyeznünk kell, hogy az eltávolító program módosításokat végezzen a számítógépen. Ezután az eltávolítás gombra kattintva befejezhetjük a műveletet, és bezárhatjuk a programot. Ha a telepítés megkezdésekor már telepítve van egy korábbi verziója a programnak, akkor a telepítő automatikusan meghívja az uninstallert.

2.4. Az első indítás

Az első indítás alkalmával a program egy felhasználói nevet kér tőlünk, és egyedi azonosítót generál (UUID). Ez a két adat az első indítás után tárolódik, így többször nem kell megadnunk/nem lesz generálva. A felhasználói nevet a program arra használja, hogy a többi felhasználó könnyen azonosítani tudjon minket a későbbiekben, a kapcsolódott felhasználók listáján.

2.5. Rajzolás

A program megnyitása után, már más kliensekkel való kapcsolat létrehozása előtt is lehetőségünk van rajzolni, és új vásznakat létrehozni. Természetesen ezek a rajzok szinkronizálódni fognak, amint a kapcsolatot létrehoztuk más kliensekkel.

2.5.1. Vásznak

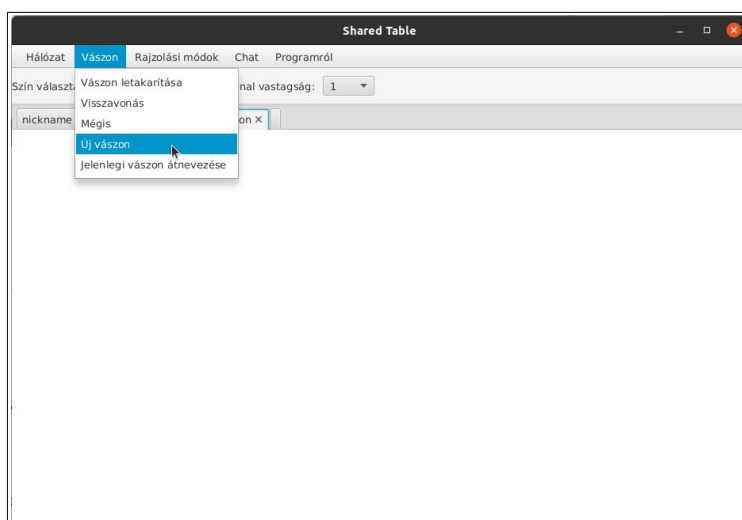
A szoftver egyből létrehoz számunkra egy a felhasználói nevünkkel ellátott vásznat. Rajzolás csak vásznakra történhet. A nevünk azért van rajta, hogy a csatlakozás utáni szinkronizációval hozott vásznak megkülönböztethetők legyenek. Később lehetőségünk van átnevezni bármilyen vásznat, de akár törölhetjük is őket, vagy újakat hozhatunk létre. Ezek a műveletek, mint az összes többi is, mind szinkronban történnek az összes hálózatra csatlakozott felhasználóval.

A vásznakkal végezhető műveletek, a vászon bezárásának kivételével, mind elérhetőek a felső menüsör Vásznon menüpontja alatt. Ezek:

- **Vásznon letakarítása:** minden rajzot eltávolít a vásznonról.
- **Visszavonás:** egyel korábbi állapotba lépteti a vásznat (az állapotokról részletesen az [állapot alapú rajzolás](#) alfejezetben). Elérhető még a Ctrl+Z billentyűkombinációval.
- **Mégis:** ez a visszavonás visszavonásának felel meg, az állapotok létrejöttének időrendjével megegyező irányban halad a tárolt állapotok között. Elérhető még

Linuxon a Ctrl+Shift+Z billentyűkombinációval, illetve Windowson a Ctrl+Y billentyűkombinációkkal.

- **Új vászon:** ez az opció a nevének megfelelően új vásznat (lapot) hoz létre. A kattintás után felugró ablakban meg kell adni a vászon nevét ami akár üres is lehet, majd az enter lenyomásával, vagy az OK gombra való kattintással létrehozhatjuk a vásznat. A vászon neveknek nem kell egyedinek lenniük, mert a program nem név alapján azonosítja a vásznakat. Amennyiben mégsem szeretnénk létrehozni a vásznat, úgy az ablak fejlécében levő X re kattintva félbeszakíthatjuk a folyamatot.
- **Jelenlegi vászon átnevezése:** erre az opcióra kattintva felugrik egy kis ablak, melynek egyetlen mezőjében megadhatjuk a lap új nevét. Működése hasonló az Új vászon létrehozáskor megjelenő ablakkal.
- **Vászon bezárása:** a vászon bezárására az aktív vászon lapfülében, a vászon nevétől jobbra megjelenő szürke X-el van lehetőségünk.
- **Vászon másolása:** az éppen kiválasztott vászonról másolat készíthető, ez másolat néven jelenik meg a többi vászon között.



Ábra 2: Egy opció kiválasztása a menüsor használatával

Az előbbi opciók kiválasztásukkor mindig arra, és csak arra a vászonra vonatkoznak, amelyik éppen aktív. Az aktív vászon lapfülének színe mindig világosabb a többinél, és világoskék keret veszi körbe, ha a program ablaka van fókuszban az operációs rendszer ablakkezelőjében (lásd [4. ábra](#)). Hálózatba csatlakozva minden kliens ugyanazokat a vásznakat látja, bármelyikre bármikor rajzolhat, és minden

változás bármely vásznon közel egy időben a változással megjelenik a hálózat minden tagjánál.

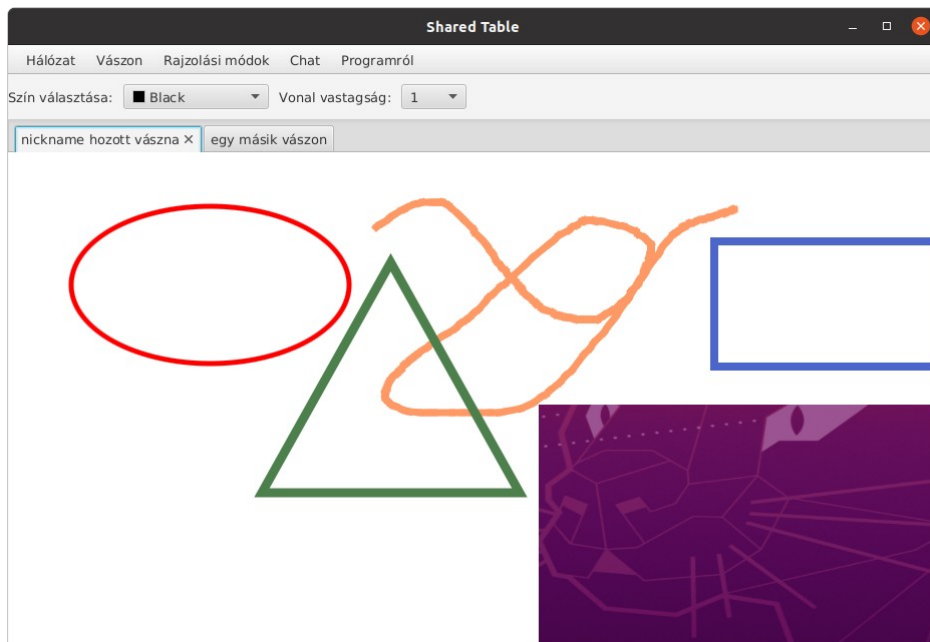
A rajzolásnak több módja is van, egész pontosan 6 darab:

- **Folytonos (tetszőleges) vonal:** Ez az alapértelmezett rajzoló mód a program indítása után. Ekkor egyszerűen nyomjuk le az egér bal egérgombját, majd kezdjük el a kurzort abba az irányba húzni amerre a vonalat vezetni szeretnénk. Az egérgombot akkor engedjük fel ha az adott vonal rajzolásával végeztünk.
- **Szabályos téglalap:** ebben a rajzoló módban a bal egérgomb lenyomásával kijelöljük a téglalap átlójának egyik pontját, majd az egér mozgásával és az egérgomb nyomva tartásával kijelöljük a téglalap átlójának másik végpontját, ezután az egérgomb felengedésével véglegesítjük a téglalapot.
- **Egyenlő szárú háromszög:** ebben a rajzoló módban a bal egérgomb lenyomásával kijelöljük az egyenlő szárú háromszög helyét és méretét meghatározó téglalap átlójának egyik végpontját, majd az egér mozgásával és az egérgomb nyomva tartásával kijelöljük a téglalap átlójának másik végpontját, ezután az egérgomb felengedésével véglegesítjük a kijelölt téglalapba illeszkedő egyenlő szárú háromszöget.
- **Ellipszis:** ebben a rajzoló módban a bal egérgomb lenyomásával kijelöljük az ellipszis helyét és méretét meghatározó téglalap átlójának egyik végpontját, majd az egér mozgásával és az egérgomb nyomva tartásával kijelöljük a téglalap átlójának másik végpontját, ezután az egérgomb felengedésével véglegesítjük a kijelölt téglalapba illeszkedő ellipszist.
- **Kép beszúrása:** A vászonra képet is be lehet szúrni tetszőleges pozícióra, tetszőleges méretben, a szabályos téglalaphoz hasonló módon. A sikeres képbeolvasásról az alábbi [üzenet](#) informál:
 - **Fájlból:** A fájlból való képbeszúráshoz válasszuk a *Rajzoló módok* menüből a *kép beszúrása fájlból* opciót. Ekkor megjelenik egy böngésző ablak, ahol a számítógépen tárolt fájlok közül kiválaszthatjuk a beszúrni kívánt png vagy jpeg formátumú fájlt. A *Szabályos téglalap* rajzolásához hasonlóan elhelyezhetjük a képet a vásznon.
 - **Vágólapról:** A vágólapról való beszúráshoz válasszuk a *Rajzoló módok* menüből a *kép beszúrása vágólapról* opciót. Ekkor amennyiben nem

megfelelő a vágólapon tárolt kép formátuma, például egy képfájl van a vágólapon, és nem vágólapra helyezett kép, úgy erre egy hibaüzenet figyelmeztet. A *gnome-screenshott*tal és az *MS Windows Képmetszőjével* biztosan működik a funkció.



Ábra 3: Sikeres képbeszúrás



Ábra 4: A program által nyújtott rajzolósi módok

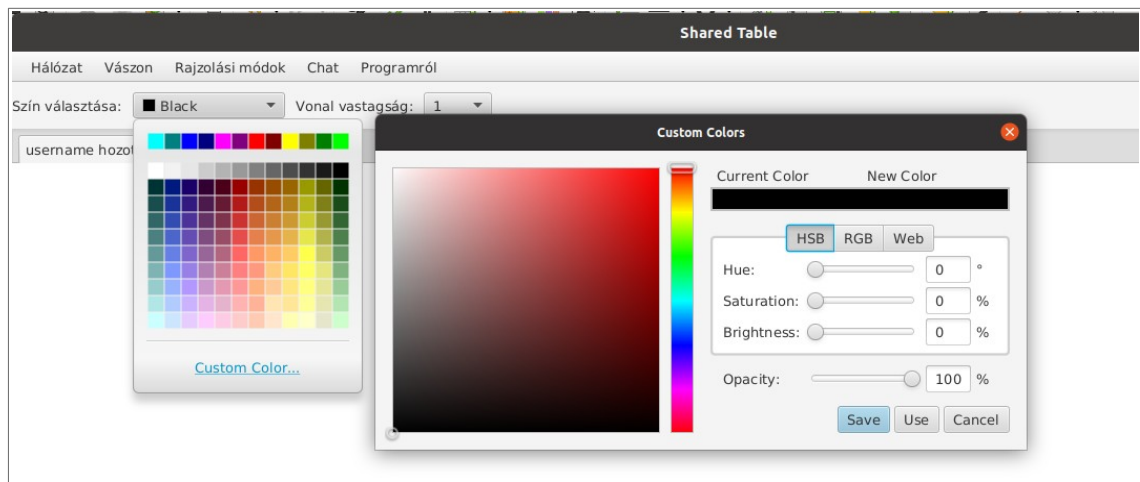
2.5.2. Vonalak tulajdonságai

A különböző rajzolósi módoknál a beállíthatjuk a vonal vastagságát, és színét a menüsor alatt és a vásznak lapfülsora felett levő két legördülő listával.

A színek választása történhet az előre beállított színpalettából való választással, amely egyből megjelenik amint a színválasztó listára kattintunk, vagy történhet egyedi szín választásával. Utóbbi opciót a lista alján levő *Custom color* nevű gombbal érhetjük el. A kívánt szín függőleges csúszkával és a színes négyszög színterének egyik

pontjának kiválasztásával tehetjük meg. Ha kiválasztottuk a megfelelő színt a *Save* vagy a *Use* gombok valamelyikével véglegesíthetjük a választást. Előbbivel menthetjük a kiválasztott színt a tetszőleges színek listáján, így később gyorsabban kiválaszthatjuk az alapértelmezett színpaletta alatti *Custom colors* címkéjű utolsó sorból.

A vonal vastagságát a színválasztó listától jobbra található legördülő listából választhatjuk ki. A kiválasztott tulajdonságok megmaradnak a vásznak közötti váltások után is.



Ábra 5: A színválasztó paletta, és az egyedi színválasztó

2.6. Állapot alapú rajzolás

A szoftver a vásznan végzett módosítások követésével teszi lehetővé a visszavonás, és a visszavonás visszavonásának lehetőségét. A vászon időrendben a következő állapotba kerül:

- Egy tetszőleges vonal húzásával, vagy
- Egy új téglalap kirajzolásával, vagy
- Egy új egyenlő szárú háromszög kirajzolásával, vagy
- Egy új ellipszis kirajzolásával, vagy
- Új kép beszúrásával, vagy
- A vászon letakarításával

Ha visszavonunk, akkor időrendben visszafelé lépkedünk az állapotok között, ha a visszavonást vonjuk vissza, akkor értelem szerűen az időrendben létrejött és tárolt állapotok között előre lépkedünk. Azt viszont fontos szem előtt tartani, hogy ha visszalépünk és egy korábbi állapoton végzünk módosítást, akkor a rendszer minden a

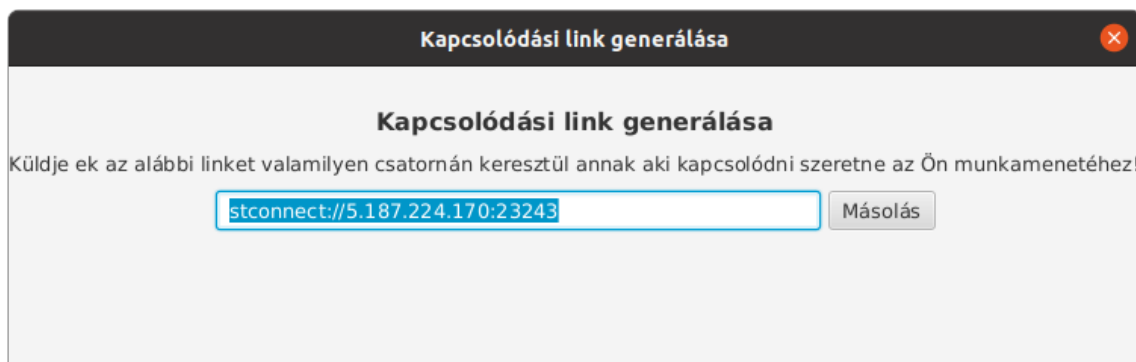
korábbi állapotot követő állapotot töröl, és helyükre a módosítással létrejött állapot kerül, pontosan ugyanúgy, mint ahogy ez a legtöbb népszerű szoftverben is történik, pl.: Microsoft Word, Microsoft Paint stb..

2.7. A hálózat

A következőkben a szoftver lényegét jelentő hálózatban történő munka működését fogom részletezni. Kétféleképpen lehetséges kapcsolatot teremteni egy másik felhasználó kliensével.

2.7.1. Bejövő kapcsolatok fogadása

Az egyik módszer, hogy engedélyezzük a bejövő kapcsolatok fogadását. Ezt a *Hálózat* menü *Bejövő kapcsolatok fogadása* opciójának kiválasztásával tehetjük meg. Ekkor a program generál egy speciális linket (pl.: `stconnect://84.2.187.254:23243`). A link egy új ablakban jelenik meg ([lásd 6. ábra](#)). A link által hordozott információkra szüksége van a kapcsolódó félnek a csatlakozáshoz, így azt valamilyen csatornán (pl.: e-mail, vagy Facebook Messenger) továbbítani kell neki. Ebben segítséget nyújt a generált linktől jobbra található másolás gomb, amely vágólapra helyezi a linket. Ezután nincsen más dolgunk, mint megvárni, hogy a csatlakozó fél hozott, a programindításkor létrejött, vagy offline módban a csatlakozás előtt létrehozott, további vásznai megjelenjenek az eddigi vásznak lapfülei mellett. Mindkét fél megbizonyosodhat a hálózat aktuális állapotáról *hálózati kliens fa* funkció segítségével, melyet [később](#) fogok részletezni.

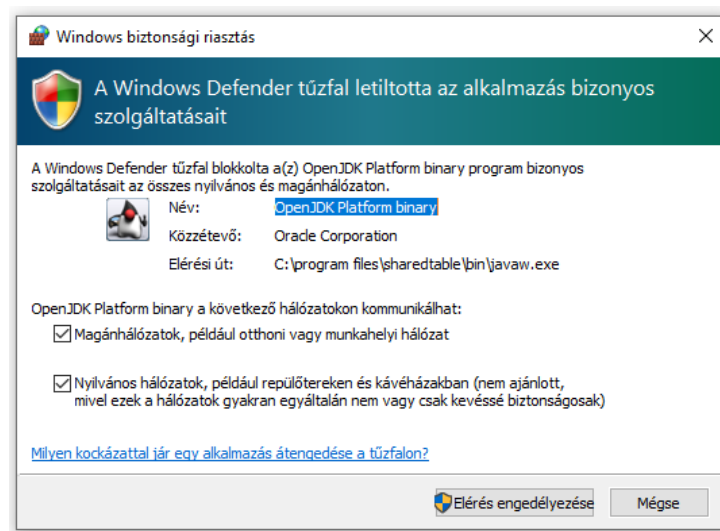


Ábra 6: Egy generált, érvényes kapcsolódási link

A bejövő kapcsolatok fogadása nem lehetséges, amennyiben az operációs rendszer és/vagy az internet hozzáférést lehetővé tevő router tűzfala blokkolja a bejövő kapcsolatot a program nyitott portjaira (melyek jellemzően a 23243, és 23244 portok). Windows esetén a program első indítása után egy felugró ablak figyelmeztet erre, ahol

(amennyiben van rendszergazdai jogosultságunk) a megfelelő gombbal engedélyezhetjük a bejövő kapcsolatokat a program felé. Linux esetén számos egyedi tűzfal megoldás létezik, általában az `ufw` program segítségével elvégezhetjük a beállítást, a `sudo ufw allow 23243` és `sudo ufw allow 23244`.

A szoftver induláskor megkísérli a helyi hálózati útválasztó (router) tűzfalának automatikus beállítását, hogy lehetővé tegye a bejövő kapcsolatok fogadását. Ezt az úgynevezett *UPnP* (Universal Plug and Play) protokoll segítségével teszi. Természetesen a program leállásakor bezárja a két megnyitott portot a tűzfalon. Manapság ez a protokoll széleskörűen támogatott, és alpból engedélyezve is van, azonban előfordulhat, hogy az adott router nem támogatja ezt a protokollt, vagy nincsen engedélyezve a belső rendszerében. Ekkor erre a program egy hibaüzeneten keresztül



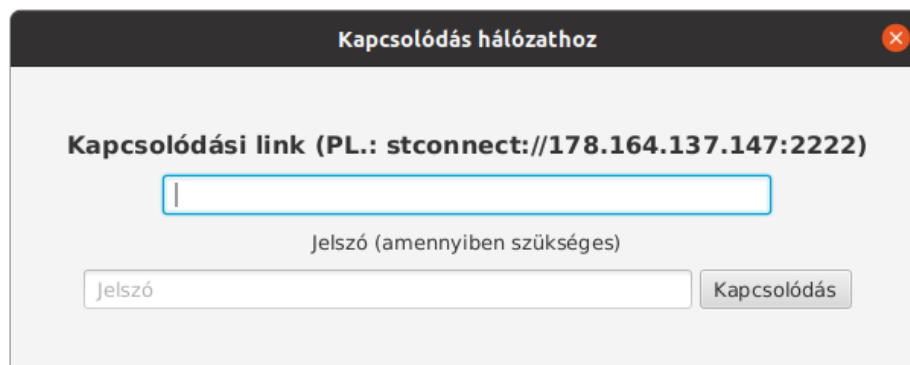
Ábra 7: Windows esetében engedélyezni kell az elérést figyelmezteti Önt. Ennek ellenőrzéséhez illetve beállításához be kell lépni a router operációs rendszerébe, jellemzően böngészőn keresztül. A belépés módját minden esetben a útválasztó kezelési utasítása tartalmazza. Amennyiben a rendszerbe belépve nem engedélyezhető az UPnP protokoll, úgy manuálisan kell beállítani a router tűzfalát. Ezt általában a *Port Forwarding* menüpont alatt teheti meg. A megnyitandó portok az előbb említett hibaüzenet alján lesznek felsorolva. Ha így sem sikerül a konfiguráció, kénytelen lesz Ön csatlakozni a hálózati munkamenet egy résztvevőjéhez.

Ha nem rendelkezik internet kapcsolattal, akkor a program az útválasztótól a gép helyi hálózati IP-címet kéri le. A program belső hálózaton is használható. Ha helyi

hálózaton belül használja a programot, akkor figyelmen kívül hagyhatja az UPnP konfigurációra vonatkozó hibaüzenetet.

2.7.2. Kapcsolódás másik klienshez

A hálózathoz való csatlakozás egy másik módja, hogy mi csatlakozunk egy a bejövő kapcsolatok fogadására alkalmas klienshez. Ezt a *Hálózat* menü *Kapcsolódás más klienshez* menüpontjának kiválasztásával tehetjük meg. Ekkor megjelenik egy ablak, melyben meg kell adnunk a kapcsolódási linket (lásd [9. ábra](#)).



Ábra 9: A hálózathoz való kapcsolódás első lépése

Ezt tetszőleges csatornánk keresztül attól a partnerünktől fogjuk megkapni, akihez csatlakozni szeretnénk. Ha a hálózat jelszóval védett, akkor a jelszó mezőbe be kell írni a hálózati jelszót, egyébként üresen kell hagyni. A hálózati jelszót, ha van, a munkamenet egy részvevőjétől kérheti el. Ha a kapcsolódás sikertelen, arra egy hibaüzenet fog figyelmeztetni (lásd. [10. ábra](#)). Egyébként a kapcsolódás sikeres volt. Sikertelen kapcsolódás oka lehet elavult, vagy rossz kapcsolódási link, az internetkapcsolat hiánya, vagy egyéb hálózati probléma.

A sikertelen kapcsolódást megelőzheti hosszabb tétlenségi idő a program részéről. Ekkor a program várakozik a hálózati kapcsolat létrejöttére. Ha a program nem kap választ a másik féltől kb. 2 perc 10 másodpercen belül, akkor a kapcsolódási folyamat hibaüzenettel leáll.



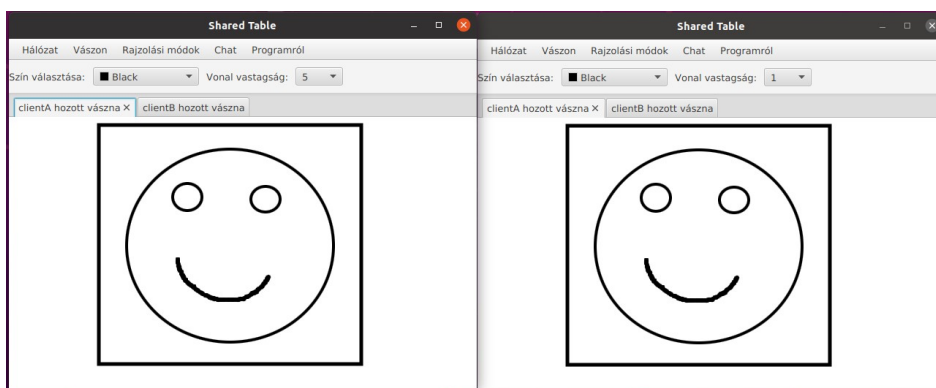
Ábra 10: Sikertelen kapcsolódás

Ne kíséreljen meg csatlakozni olyan klienshez, amellyel Ön már hálózati kapcsolatban van bejövő kapcsolat révén! Ez körkörös adatfolyamot eredményezhet, és a hálózat összeomlásával járhat

2.7.3. Szinkronizáció

Akár kapcsolatot fogadó, akár csatlakozó szerepben van a kliens, a csatlakozást követően szinkronizáció veszi kezdetét. Erre egy felugró ablak figyelmeztet rögtön a kapcsolatfelvételt követően. Ekkor az összes vászon összes állapota szinkronizálódik a hálózatba kapcsolódó kliens és a hálózat eddigi részvevői között. A szinkronizáció alatt a szoftver mindkét oldalon használhatatlan. **Ne kíséreljen meg** rajzolni, képet beszúrni, vagy új vásznat létrehozni a szinkronizáció alatt! A felugró ablak pontosan addig látható ameddig a szinkronizáció tart.

A hálózaton keresztül összekapcsolódott kliensek minden módosítást azonnal továbbítanak így (kb. 0,005-0,5 másodperc késleltetéssel) azok minden kliens vásznán megjelennek.



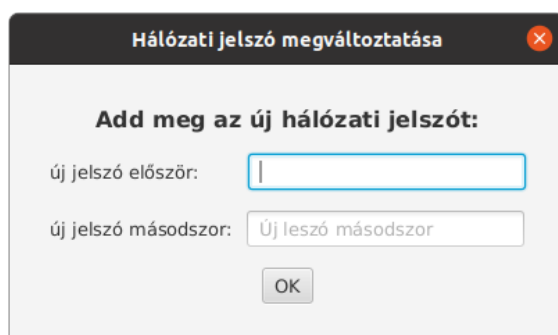
Ábra 11: Sikeres csatlakozás utáni szinkronba állt kliensek a hozott vásznakkal

Csatlakozás után a kliensek ellenőrzik egymás verziószámát. Ha az Ön által telepített verzió verziószáma kisebb, akkor erre a program figyelmeztetni fogja. Ekkor érdemes ellátogatni a szoftver webhelyére az új verzió letöltéséhez. **Saját felelősségére** tovább használhatja a programot így is, de ez nem várt viselkedést válthat ki a hálózat összes klienséből!

2.7.4. Hálózati jelszó

A hálózat védhető jelszóval. Ekkor a csatlakozás, bármely a hálózat részét képező klienshez, csak a megfelelő jelszó megadásával lehetséges. Csatlakozáskor a jelszót a *Kapcsolódás hálózathoz* címkéjű ablak *Jelszóval* jelzett mezőjében kell megadni. Ha nincsen hálózati jelszó, akkor a mezőt üresen kell hagyni.

A hálózat jelszavát bármely kliens módosíthatja. Ezt a menüsor *Hálózat* menüje alatti *Hálózati jelszó megváltoztatása* opcióval tehetjük meg. Ekkor egy felugró ablakban kell megadnunk az új jelszót kétszer, majd az OK gombra kattintani. Ha a két mezőbe beírt jelszó nem egyezik meg pontosan, erre egy hibüzenet figyelmeztet, majd újra kell próbálkoznunk. Ha meg szeretné szüntetni a jelszót, akkor egyszerűen csak hagyja üresen a mezőket, majd nyomjon enter-t, vagy kattintson az OK gombra.



Ábra 12: Hálózati jelszó változtatás

A hálózathoz való csatlakozás lehet jelszó védett, de a kliensek közötti kommunikáció **nem titkosított** a hálózati végpontok között! Ezért **NE adjon meg semmilyen bizalmas adatot a szoftveren belül**, mert azok nyílt wifi hálózatokon, és az internet szolgáltatók útválasztóin mind olvashatóak, és tárolhatóak.

2.7.5. Hálózati kliens fa

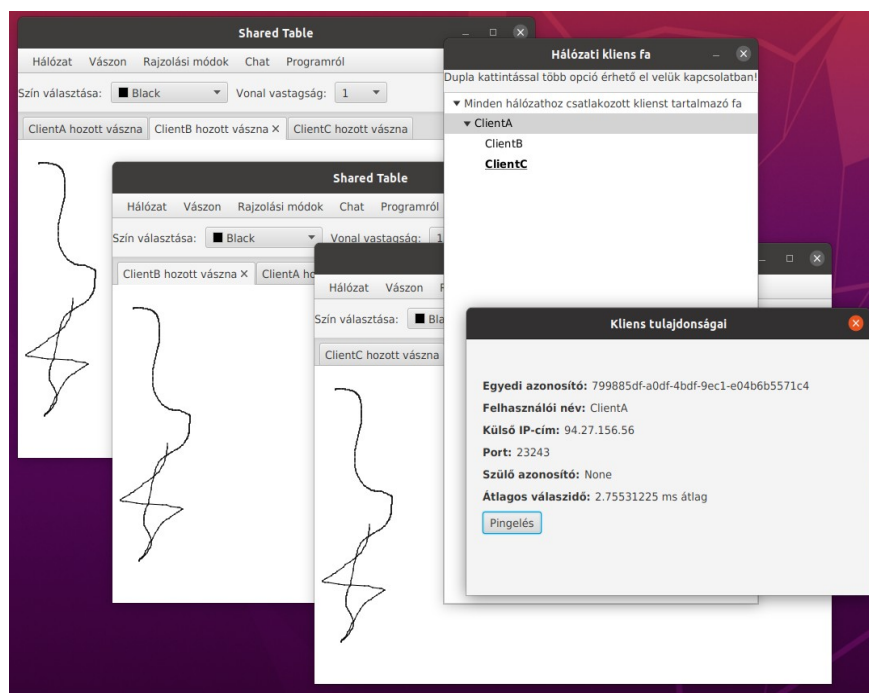
Ha csatlakoztunk egy hálózathoz, vagy hozzánk kapcsolódtak, akkor a hálózat összes jelenlegi kliensét, és a hálózat aktuális topológiáját megtekinthetjük a menüsor Hálózat menüpontja alatti Hálózati kliens fa opcióra kattintva. Ekkor megjelenik egy

ablak, melyben egy fa gráfba rendezve láthatjuk a munkamenet jelenlegi klienseit (lásd [13. ábra](#)). Ebből kiderül, melyik kliens melyik kliensen keresztül csatlakozik a hálózathoz. Saját kliensünk neve **félkövérrel, aláhúzva** jelenik meg a fában.

A kliensekről további információt is megtudhat, ha duplán kattint a felhasználói nevére. Ekkor egy új ablakban olvasható a kliens:

- Egyedi azonosítója
- Felhasználói neve
- Külső hálózati IP-címe
- Bejövő kapcsolatok fogadására alkalmas portja
- Szülő azonosítója (annak a kliensnek az egyedi azonosítója, amelyen keresztül a hálózatba kapcsolódik)
- Az Ön kliense, és az adott kliens közötti átlagos hálózati válaszsideje (ping-je).

Ha a port mellett **-1** jelenik meg, az azt jelenti, hogy az adott kliens nem engedélyezte a bejövő kapcsolatokat, vagy a hálózati konfigurációja nem teszi lehetővé a kapcsolat fogadását.



Ábra 13: Egy hálózathoz kapcsolódott kliens tulajdonságai

Ha nincs szülő azonosító, az azt jelenti, hogy az adott kliens a hálózat gyökere. Vagyis egyedüli olyan eleme, amely bejövő kapcsolat révén a hálózat része, de ő maga nem csatlakozott más klienshez.

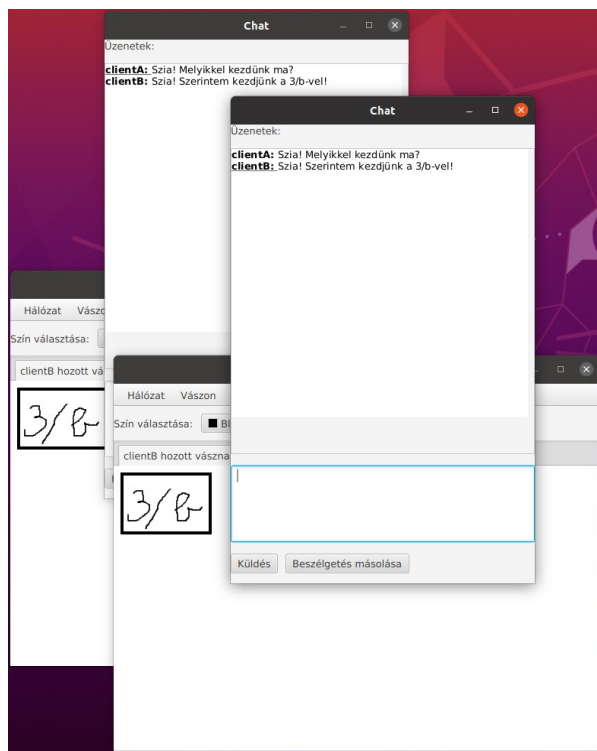
A ping csak akkor jelenik meg, ha a *Kliens pingelése* gombra kattintva elvégzi a tesztelést. Minél távolabb helyezkedik el az ön kliense a fában az adott klienstől, illetve minél több bejövő kapcsolattal terhelt klienshez kapcsolódik, a válaszütem annál nagyobb lesz. A nagy válaszütem azt jelenti, hogy több idő telik el, míg megjelennek az adott kliensnél készített rajzok az Ön képernyőjén.

Nagy válaszütem esetén érdemes lehet megfontolni egy másik klienshez való átcsatlakozást. Ekkor felveheti a kapcsolatot az adott klienssel, hogy elkérje az ehhez szükséges kapcsolódási linket. Haladó felhasználók összeállíthatják saját maguknak is a linket a *Kliens tulajdonságai* panelen megjelenő adatok alapján is (IP, port). Ön a saját kliensét nem pingelheti, de nyugodtan 0 ms-nak veheti.

2.7.6. Hálózati Chat

A felhasználók valószínűleg leggyakrabban valamilyen harmadik féltől származó szoftvert fognak használni a munka közbeni kommunikációra. Azonban előzetes egyeztetés miatt, vagy más egyéb célból lehetőség van szöveges üzeneteket küldeni a programon belül is. Ezt beszélgetést a menüsor *Chat* menüje alatti *beszélgetés megnyitása* opcióra kattintva lehet megtekinteni.

Az ekkor megjelenő ablakban láthatjuk az eddig lefolyt beszélgetést, és a *Beszélgetés másolása* gomb fölötti szövegdobozba beírhatjuk saját üzenetünket, melyet a *Küldés* gombbal, vagy az enter lenyomásával küldhetünk el a többi kliensnek. Az üzenetben használhatunk sortörést a *Shift+Enter* billentyű kombinációval. Korábbi üzenetek megtekintéséhez használja az ablak jobb oldalán megjelenő csúszkát, vagy az egérgörgőt. A *Beszélgetés másolása* gombra kattintva a program vágólapra helyezi az egész beszélgetést, hogy Ön másik programban szerkeszthesse vagy tárolhassa azt. Minden üzenet előtt aláhúzva és félkövérrel szedve megjelenik annak küldője.



Ábra 14: Hálózati chat szinkronba állt, csatlakozott kliensek között.

2.8. Frissítések

A program minden induláskor megpróbál felcsatlakozni a szoftver weboldalát üzemeltető szerverre (<http://gpeter12.web.elte.hu/sharedtable>), hogy frissítéseket keressen. Ha új verziót talál, arról rögtön értesíti Önt. Az ekkor megjelenő ablakban tájékoztatást kap az új verzió verziószámáról, és újdonságairól. Az *Új verzió letöltése* gombra kattintva a szoftver átirányítja az előbb linkelt webhelyre, ahonnan letöltheti az új verziót. A frissítések keresését manuálisan is elindíthatja a menüsor *Programról* menüjének *Frissítések keresése* opciójára kattintva.



Ábra 15: Figyelmeztetés új frissítésre

2.9. Összes hibaüzenet listája

- **A megadott jelszavak nem egyeznek meg:** Jelszóváltoztatásnál nem egyezik meg a két szövegdoz tartalma.
- **Érvénytelen kapcsolódási link:** a megadott kapcsolódási link nem felel meg a formai követelményeknek. stconnect:// előtaggal kell rendelkeznie, majd egy érvényes IP címnek kell következnie, utána egy kettőspont következik, és végül egy érvényes portszámmal zárul.
- **Nem nyitható meg böngésző:** a frissítéskereső megpróbált az alkalmazás weboldalára átirányítani az alapértelmezett böngésző segítségével, ám az operációs rendszer nem támogatja ezt kérést, és nem nyitja meg a weboldalt.
- **A hálózati kapcsolat megszakadt bájt tömb fogadásakor:** az alkalmazás megpróbált képet fogadni egy másik klienstől, de közben a hálózati kapcsolat megszakadt.
- **A megadott hálózati jelszó érvénytelen:** a hálózat, amelyhez kapcsolódni kíván jelszó védett, és nem a megfelelő jelszót írta be, vagy üresen hagyta a Jelszó mezőt a Kapcsolódás hálózathoz című ablakban, pedig az adott hálózat jelszóvédett.

- **Hiba a port megnyitáskor:** az alkalmazás több alkalommal próbált különböző portokat lefoglaltatni az operációs rendszerrel bejövő kapcsolatok fogadásához, de a rendszer minden alkalommal megtagadta ezt.
- **UPnP konfigurálási hiba:** az alkalmazás megpróbálta felvenni a kapcsolatot a helyi hálózati útválasztóval, hogy lekérje a külső IP címet, vagy beállítása a Port forwardingot a bejövő kapcsolatok fogadásához, de a router nem képes vagy hibás beállítás miatt nem hajlandó eleget tenni a kéréseknek. Ha lehetséges, engedélyezze routerén az UPnP protokoll támogatást, vagy állítsa be manuálisan a port forwardingot, és hagyja figyelmen kívül az üzenetet.
- **Ismeretlen operációs rendszer:** az alkalmazásnak nem sikerült a JDK-n keresztül megállapítania az operációs rendszer típusát (Linux/Windows), ezért nem tudja eldönteni, hogy milyen módon formázza a fájlok elérési útvonalait.
- **Felhasználói adatfájl írása/olvasása nem lehetséges:** az alkalmazásnak nincsen jogosultsága, hogy írjon illetve olvasson a `~/.config/SharedTable` (linux) vagy a `[felhasználói mappa]\AppData\Local\SharedTable` (windows) mappákból, vagy a mappa valamilyen okból nem jött létre.
- **Felhasználói adatfájl sérült:** valamilyen okból szintaktikailag helytelen a `userconfig.json`, mely az előbbi hibaüzenetben leírt helyen található. Törölje ki a fájlt a `~/.config/SharedTable` (linux) vagy a `[felhasználói mappa]\AppData\Local\SharedTable` (windows) mappákból, és indítsa újra az alkalmazást!
- **Sikertelen kapcsolódás:** nem lehet kapcsolódni a kapcsolódási linkben megadott helyre. Kérjen másik linket, vagy ellenőrizze, hogy kapcsolódik-e az internethez, illetve érdeklődjön, hogy a fogadó félnél megfelelően van-e konfigurálva a helyi útválasztó és az operációs rendszer tűzfala.
- **nem megfelelő formátumú beillesztési tartalom:** a vágólapról beilleszteni próbált kép formátuma nem megfelelő. Vagy képfájlt helyezett a vágólapra egy fájlkezelőből, vagy a program ami a vágólapra másolta képet nem a szabványos vágólapkép formátumot használja. Ha ezt az üzenetet a fájlból való képbeszúrás opció kiválasztása után látja, akkor valamilyen okból az ön által kiválasztott képfájl tartalmának formátuma nem egyezik meg a fájl kiterjesztése által mutatottal.

- **Hiba a naplófájl létrehozásakor:** az alkalmazásnak nincsen jogosultsága, hogy írjon illetve olvasson a `~/ .config/SharedTable` (linux) vagy a `[felhasználói mappa]\AppData\Local\SharedTable` (windows) mappákból, vagy a mappa nem jött létre. Ezért nem lehetséges naplófájl létrehozása.

3. Fejlesztői dokumentáció

A program fejlesztése során az egyik elsődleges szempont az volt, hogy a használt keretrendszerek által nyújtott szolgáltatásokra támaszkodva könnyen átlátható, és karbantartható kód készüljön. Ennek érdekében igyekeztem minél több helyen a jól bevált design patterneket alkalmazni, mint például a memento¹³, command¹⁴, observer¹⁵, singleton¹⁶, és factory pattern.

3.1. Felhasznált eszközök

3.1.1. OpenJDK 14.0.1

A szoftvert **Java nyelven** (OpenJDK 14.0.1)¹ írtam, ez egy erősen típusos objektum orientált nyelv. A nyelv ezen tulajdonságai elősegítik a gyors munkát, és könnyű részegységekre bontani a szoftvert, ami sokat javít karbantarthatóság szempontjából és kód újrafelhasználás szempontjából is. Céлом volt, hogy a program kifogástalanul fusson Windows rendszereken és Linux alapú operációs rendszereken is. Emiatt nagy segítség volt a Java által biztosított platformfüggetlenség. A hiba keresést nagyban megkönnyíti a Java Virtual Machine (JVM) által biztosított önreflexió, melynek segítségével sokkal több futásidejű információhoz lehet jutni a debuggolás során, mint egy egyből natív kódra forduló program esetében (pl. C/C++). Fontos szempont volt, hogy a választott nyelv erősen támogassa a hálózati programozást, és a szálkezelést. Utóbbiban a nyelv sokat fejlődött az elmúlt pár évben, és rengeteg olyan kódot és könyvtárat tartalmaz a nyelv fejlesztői környezete, amelyek elősegítik és kényelmessé teszik a hálózati programozást.

3.1.2. Jlink

A JDK 9-es verziójával érkezett a **jlink program**², amely képes a szintén JDK 9-el érkezett Java Platform Module System-el³ (JPMS) kompatibilis kódot összelinkelni, és tömöríteni minden függőségével együtt. Ez lehetővé teszi, hogy a szoftvert úgy csomagoljam Windows telepítő programba, és debian csomagba, hogy az már tartalmazza az OpenJDK, és az OpenJFX összes szükséges komponensét, és nincsen szükség arra, hogy a JDK megfelelő verzióját a felhasználó külön telepítse vagy beállítsa. Így a szoftver terjesztése nagyon kényelmes, és megbízható. Ezzel az eszközzel kiküszöböltem a menedzselt nyelvek egyik legnagyobb hátrányát, hiszen sem

a felhasználónak, sem a fejlesztőnek nem kell foglalkoznia a futtatókörnyezet telepítésével. Ez még egy érv volt Java mellett.

3.1.3. OpenJFX 14.0.1

A szoftver felhasználói felületének kialakításához **OpenJFX 14.0.1**^{4.5}-et használtam, mert ez a szoftver platform nagyon jól együttműködik a Java nyelvvel hiszen ön maga is Java-ban íródott, megfelelő teljesítménnyel rendelkezik, nincsen túl nagy overhead-je, teljesen ugyanúgy jelenik meg Windows és Linux rendszereken is, valamint aktív fejlesztés alatt áll a mai napig, de ugyanakkor elég nagy múlttal rendelkezik, hogy megbízható alapot nyújtson a felhasználó felület stabil működéséhez. A program a UI tervezésére biztosít egy deklaratív felületleíró nyelvet, a JavaFX FXML-t, amelyben viszonylag gyorsan le lehet írni a felületi elemek kívánt elhelyezkedését és méretét, valamint így nagyon jól külön lehet választani a program felületének kinézetét annak aktuális funkcionalitásától, mivel így teljesen más nyelven, másik fájlban van elhelyezve a felületet leíró kód és a vezérlők eseménykezelésének kódja. Ez növeli a szoftver modularitását. A Java és az FXML kód jól kapcsolódik egymáshoz a Java kódban használt `@FXML` annotáció segítségével.

3.1.4. Gradle 6.6.1

Projektépítő eszköznek a **Gradle 6.6.1**^{6.7}-es verzióját választottam. Ez a szoftver segít fejlesztői környezettől, és operációs rendszertől függetlenül egységesíteni a fordítási, tesztelési, és szoftverkiadási feladatokat, mint például a Windows installer és a debaian package legenerálása. Így akár integrált fejlesztői környezet nélkül egyetlen paranccsal letölthetőek a szoftver fordításához szükséges függőségek, lefuttathatóak a Unit tesztek, és elvégezhető a fordítás, sőt, még a futtatás is. A teszteléshez használtam a JUnit5¹² keretrendszert is. A Gradle futtatásához, és a program fordításához/futtatásához viszont mindenképpen szükség van az OpenJDK 14.0.1-es verziójára, és arra, hogy a JAVA_HOME környezeti változó az ezt tartalmazó könyvtárra mutasson, mivel maga a Gradle is Java-ban íródott. Megemlítenék néhány fontosabb parancsot, amelyekkel elvégezhetőek a legalapvetőbb műveletek a projekten, valamint még kettőt, amelyek két általam definiált Gradle taskot hajtanak végre. Ezek:

- **gradlew build**: lefordítja az *src* mappában tárolt forráskódot, lefordítja és lefuttatja a Unit teszteket, és összelinkeli a fordított programot a *libs* mappában tárolt két további kódkönyvtárral, melyekről az [alábbi bekezdésben](#) fogok írni.

- **gradlew jlink**: a JVM bájtkódból és a függőségeiből standalone image-t készít, ahogy azt már a [jlink-ről szóló bekezdés](#)ben részleteztem, a build/image mappa, így a programnak az adott operációs rendszerhez készült, hordozható verzióját tartalmazza. Fontos, hogy a build/image/bin/javaw.exe, (Windows esetében) és a build/image/bin/java (Linux esetében) nem indítható a megfelelő parancssori argumentumok nélkül.

Ezek: `-Dprism.lcdtext=false -Dfile.encoding=UTF-8 -m sharedtable/com.sharedtable.controller.MainViewController.`

Az első argumentum egy grafikai megjelenítési hibát küszöböli ki Linux alatt, ami miatt kevésbé olvashatóak a fontok. A második biztosítja, hogy a Stringek minden platformon UTF-8 kódolást használjanak. A különböző kódolás hibát eredményez a kliensek közötti hálózati kommunikációban. Az utolsó argumentum azt írja le a JVM-nek, hogy melyik osztályban van definiálva a program belépési pontja, vagyis a main függvény.

- **gradlew run**: futtatja a class fájlokat.
- **gradlew make_deb**: előkészíti a futtató scripteket és a .desktop fájlt, és átmásolja az image mappa tartalmát a deb_release mappába, és ugyanide regenerálja belőlük a debian-csomagot. Ez a task csak Linuxon működik, és elérhetőnek kell lennie a dpkg-deb programnak.
- **gradlew make_installer**: telepítőalkalmazásba csomagolja a program fájljait. A task futtatásának előfeltétele, hogy az NSIS 3.05-ös verziója telepítve legyen a C:\Program Files (x86)\NSIS helyre. Ha nem ide van telepítve akkor ennek megfelelően módosítani kell a build.gradle fájlban definiált make_installer task, CommandLine parancsának első argumentumát.

3.1.5. WaifUPnP és json-simple

A fejlesztésénél felhasználtam **két** kisebb nyílt forráskódú **Java kódkönyvtár**at, az egyiket a UPnP könnyebb kezelésére, a másikat a JSON fájlbeolvasáshoz, és kiíráshoz. Előbbihez Federico Dossena WaifUPnP⁸ (com.dosse.upnp) java csomagját, utóbbihoz *fangyidong* GitHub felhasználó json-simple⁹ (org.json.simple) java csomagját. Minden a programban felhasznált keretrendszer és programkönyvtár jogszerűen került felhasználásra, hiszen az én projektem GNU General Public License

v3.0 alatt jelent meg GitHub-on, így szabadon használhatók más nyílt forráskódú kódrészleteket a szoftveremben.

3.1.6. Nullsoft Scriptable Install System

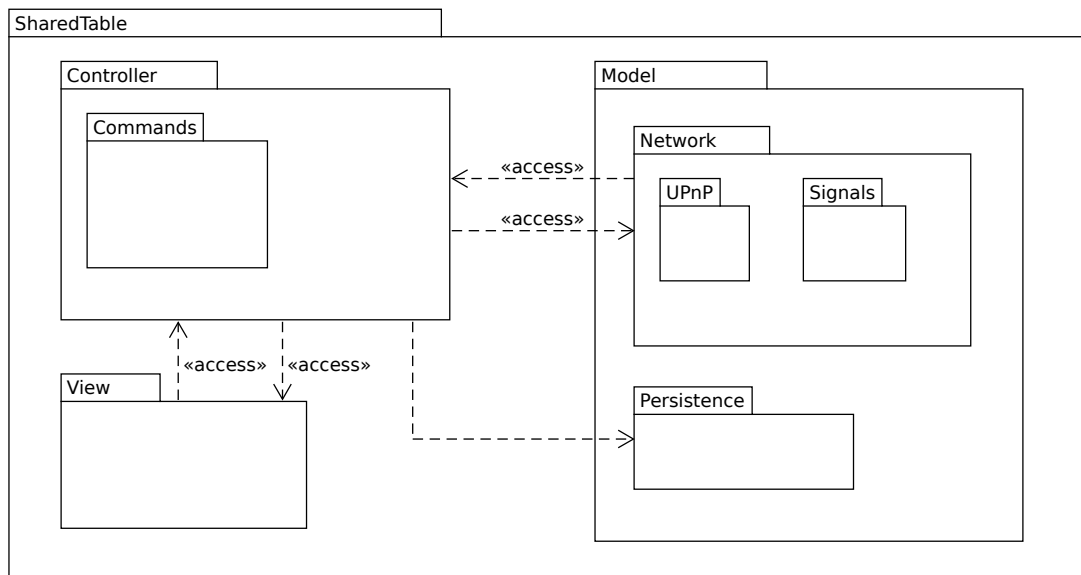
A zökkenőmentes telepítést Windows-on a **Nullsoft Scriptable Install System**^{10,11} (NSIS) által generált telepítő csomag biztosítja. Ahogy az a szoftver nevében is benne van, ezt a telepítőcsomagot egy script alapján generálja az NSIS. Ez a script az `nsisOutput\SharedTableInstaller.nsi` fájlban található. Itt definiáltam, hogy a telepítés milyen szekciókból álljon, milyen fájlokat foglaljon magába a telepítő, hová és milyen regisztrációs bejegyzések kerüljenek, hová kerüljenek a program futtatását lehetővé tevő parancsikonok, és azok milyen argumentumokkal indítsák a `javaw.exe`-t, milyen fájlokat töröljön az uninstaller, és az hova kerüljön. Az egyetlen regisztrációs bejegyzés, amit jelenleg létrehoz és használ a telepítő az a `Software\SharedTable` `insdir`. Az uninstaller ennek a kulcsnak az alapján találja meg a szoftver telepítési helyét. Ha a `.nsi` fájl valamiért átkerülne máshova, akkor módosítani kell a `pf` szekció `File` parancsának argumentumát, mert a jelenlegi helyhez képest relatívan van megadva a `build/image` mappa helye.

A megfelelő Linux disztribúciókon az Advanced Packaging Tool (APT) által kezelt Debian Package (`.deb`) felelős a szoftver telepítéséért, és az eltávolításáért. A `debian` csomag a `deb_release` mappába generálódik. A `deb_release/sharedtable/DEBIAN` mappa tartalmazza a csomag metaadatait a `control` fájlban. A `postinst` fájlban található bash script helyezi el a parancsikonokat az asztalra és a `/usr/share/applications` mappába, továbbá a szimbolikus linket a `/usr/bin` mappába. A `posrtm` script eltávolítja a konfigurációs és naplófájlokat, amiket a program a futásakor létrehoz, valamint az összes parancsikont és szimbolikus linket.

3.2. A program szerkezete és működése

A szoftver elkészítésénél MVC (Model-View-Controller) programtervezési mintát használtam, így a programkód ennek megfelelően három nagy csomagra oszlik szét. A **Model csomag** felelős a felhasználói adatok fájlba való kiírásáért és fájlból való beolvasásáért, valamint a hálózati kommunikáció lebonyolításáért. Értelmszerűen a **View csomag** felel a különböző ablakok megjelenítéséért, és az ablakokra kirajzolt vezérlőkkel kapcsolatos események kezeléséért. A **Controller csomag** pedig főként az

események rajzolási parancsokká alakításáért, a rajzolási parancsok állapotokba szervezéséért, az állapotok kezeléséért, a vásznak kezeléséért, és a bevitt adatok helyesség ellenőrzéséért felel. Az alábbiakban az előbbi három csomag bemutatásán keresztül fejtem ki a program működését.



Ábra 16: csomag diagram

3.2.1. Controller réteg

Ez a réteg felelős a vásznak állapotainak generálásáért, tárolásáért, és az összes vásznon végezhető művelet végrehajtásáért. Az alábbiakban egy tetszőleges vonal rajzolás folyamatának bemutatásán keresztül fogom néhány jelentősebb osztály szerepét és működését ismertetni.

A CanvasController felelős a vásznon létrejött események kezeléséért, és a vásznon állapotával kapcsolatos adatok tárolásáért. A vászonra való vonal rajzolás a következőképpen történik: a view rétegből érkezik az adott vászon állapotát reprezentáló CanvasController objektum felé az egérlenymást kezelő `mouseDown(Point)` metódus hívás. Itt az `isMouseDown` változóban regisztráljuk, hogy a vásznon lenyomott egérgomb mellett tevékenykedik a felhasználó. Ekkor ezt a pontot, ahol le lett nyomva az egér szintén elmentjük a `lastPoint` változóba. Ha ilyenkor mozgatjuk az egeret, akkor (mivel már tudjuk, hogy a bal egérgomb le lett nyomva) a `mouseMove(Point)` metódus már rajzolást fog végezni a vásznon. Ez a metódus egyébként minden alkalommal lefut, ha a kurzor megmozdul.

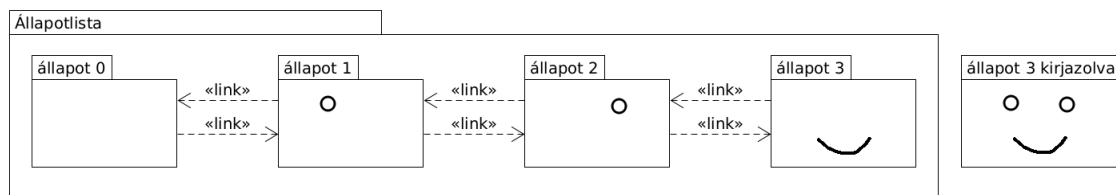
A `mouseDown(Point)` metódus az aktuális rajzolósi módot tároló `currentMode` változó állapotának megfelelő rajzolósi fog végezni. Ezek lehetnek:

- `ContinuousLine` (folytonos tetszőleges vonal rajzolósi)
- `Rectangle` (Téglalap rajzolósi)
- `Ellipse` (Ellipszis rajzolósi)
- `Triangle` (egyenlő szárú háromszög rajzolósi)
- `Image` (kép kirajzolósi)

Az egér mozgatósiakor a `mouseMove(Point)` paraméterül megkapja azt a pontot, ahová mozgult az egér a vásznon. A vonal így két nagyon közeli pontot összekötő egyenes vonalak sokaságából áll össze. Az egyik pont a kurzor korábbi pozíciója (`lastPoint` változó), a másik pont az egér új pozíciója (`p` változó). Értelemszerűen a `lastPoint` felhasználása után frissítjük az értékét (`lastPoint = p`). A tényleges kirajzolósi viszont nem a `CanvasController` végzi. Itt lesz jelentősége a `Command` tervezési mintának. A metódus ezen a ponton létrehozza a rajzolósi módtól függően a megfelelő parancs objektumot, amely rendelkezik egy `execute()` eljárással, és az adott parancs paramétereivel, valamint azzal a `view` csomagbeli `STCanvas` vászon objektummal, amin ténylegesen végre kell hajtani a rajzolósi. Ez a megoldás azért előnyös, mert így kényelmesen ki lehet szervezni a rajzolósi parancsok végrehajtását egy csak ezt a feladatot végző szála. Ezt a szálat **`CommandExecutorThread`** típusú objektum reprezentálja, amelyből minden vászonhoz tartozik egy. Mivel a `Command`-ok végrehajtását egyetlen külön szál végzi, így nem merül fel az a probléma, hogy egy távoli felhasználó és a helyi felhasználó rajzol egyszerre ugyanarra a vászonra, azaz egyszerre használják ugyanazt az erőforrást, mert ez a szál egy `BlockingQueue`-ban pufferelem a beérkező parancsokat és érkezősi sorrendben **egymás után** hajtja őket végre. Továbbá ez a megoldás azért is előnyös, mert így a rajzolósi nem lassítja sem a hálózati inputokat, sem a felhasználói inputokat kezelő szál munkáját. A `CommandExecutorThread` `run()` eljárása lényegében végtelen ciklusban fut és dolgozza fel a `commandQueue` adattagba érkező `Command` példányokat, ameddig nem kerül meghívásra a `timeToStop()` eljárás. A távoli kliensektől külön sorokba kódolt `String`ek formájában érkező `Command`-okat a **`CommandFactory`** osztály alakítja át megfelelő típusú `Command` példányokká.

3.2.1.1 Állapot tároló rendszer

A vászon állapottároló rendszerét működtető **StateMemento**, **StateCaretaker**, és **StateOriginator** osztályok megalkotásakor a *Memento tervezési mintát alkalmaztam*. Miután átkerült a Command példány a parancsfeldolgozó szálhoz, a parancs tárolásra kerül a vászonhoz tartozó StateOriginator példányban. A StateOriginator felel az állapotok létrehozásáért. A tetszőleges vonal apró egyenes vonal komponensei addig gyűlnek a currentCommandList adattagjában, ameddig nem hagyjuk abba a vonal rajzolását, vagyis ameddig nem engedjük fel az egérgombot. Ekkor ezekből létrehoz egy StateMemento példányt, amely a vászon egy új állapotát eredményező változást tárol. Ezeket az állapotátmeneteket tároló objektumokat az adott vászonhoz tartozó StateCaretaker objektum tárolja. Rajta keresztül lehet megkeresni egy adott állapotot, és ő gondoskodik az állapotok láncba fűzéséről is. Láncba fűzés alatt azt értem, hogy minden állapot tárol egy referenciát az őt megelőző állapotra, és egyet az őt követő állapotra is (lásd LinkMementoWithMemento(StateMemento, StateMemento)). Így egy adott állapotból rekurzívan vissza iterálva összegyűjthető az összes parancs, ami az adott állapotot eredményezte az üres vászontól, az első állapottól, hiszen minden StateMemento a két szomszédos állapot közötti különbséget leíró parancsokat tartalmazza.



Ábra 17: Láncolt állapotok szemléltetve

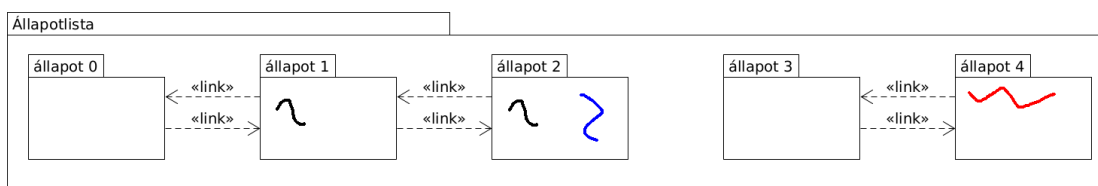
Ez hatékonyabb memóriafelhasználás szempontjából is, mert nincs két olyan StateMemento példány, melyek által tárolt parancsoknak metszete lenne, hiszen az felesleges redundanciát okozna.

Minden új vászon létrehozásakor létrejön egy speciális állapot. A vászon első, üres változata. Ez az állapot nem tartalmaz parancsot, és mindig a Nil UUID-t kapja (00000000-0000-0000-0000-000000000000). Erre a speciális azonosítóra azért van szükség, mert ez az egyetlen állapot, amit nem küldenek el egymásnak hálózaton a

kliensek, de feltételezniük kell, hogy ott van, és visszavonáskor ugyanúgy kell rá hivatkozniuk.

Ahogy keletkeznek az újabb és újabb apró egyenes vonalakat leíró parancsok, úgy továbbbódnak Model réteg NetworkService komponense felé, amely a többi klienssel való hálózati kapcsolattartásért felel. Ennek köszönhetően ők valós időben követhetik a vonal kirajzolását a saját képernyőjükön az első parancstól az utolsóig. Ez a látványos megvalósítás rejteget néhány buktatót magában, amivel fontos külön foglalkozni, ezt a RemoteDrawLineCommandBufferHandler osztály [ismertetésekor](#) fogom megtenni.

A vonal rajzolásának utolsó eseménye az egérgomb felengedése a vonal végén. Ekkor egy insertNewMementoAfterActual(true) hívással lezárjuk az éppen készülő StateMemento-hoz tartozó rajzolási parancsok összegyűjtését a StateOriginator-ban, ami elvégzi az utolsó simításokat az új objektumon, majd átadja azt a StateCaretaker-nek, aki azok tárolásával foglalkozik. Ezt követően pedig értesítjük a többi klienst a Model rétegen keresztül, hogy tegyék meg ugyanezt a hozzájuk eddig beérkezett rajzolási parancsokkal. Az állapotok tárolására azért van szükség, mert az állapotok között vissza, és előre lépkedve lehet megvalósítani a **visszavonást** (Ctrl+Z), és a visszavonás visszavonását (Ctrl+Y, Ctrl+Shift+Z). Számos helyen előfordul a függvények formális paraméterlistájában a **boolean típusú link argumentum**. Ez azért van, mert van egy speciális eset, amikor egy bizonyos ponton meg kell szakítani a StateMemento példányok közötti referencia láncot.



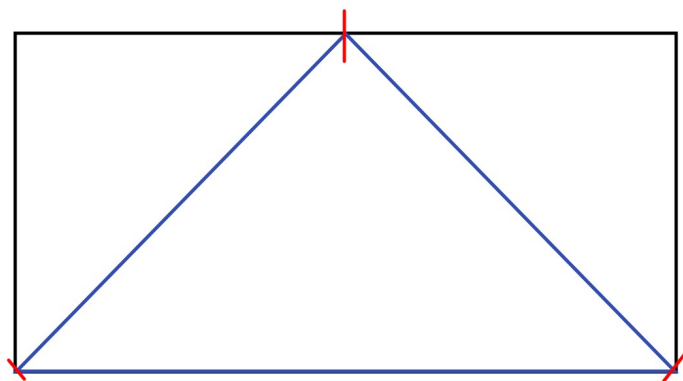
Ábra 18: A 2-es állapot és a ClearCommand által létrehozott 3-as (üres) állapot között nincs láncolás

Ez akkor fordul elő, amikor le kell takarítani a vásznat. A ClearCanvas parancs végrehajtásakor egy új üres StateMemento példányt szúrunk be az aktuális állapot után. Azonban, ha eközött az új üres állapot és az őt megelőző állapot között is kialakítjuk a kapcsolatot, akkor annak ellenére, hogy letöröltük a vásznat, a következő visszavonás alkalmával a program, mikor rekurzívan végig iterál, hogy összegyűjtse az állapot

kirajzolásához szükséges parancsokat, begyűjti és kirajzolja a vászon törlését megelőző parancsokat is. Így aztán a vászon törlés, amely egy vagy több állapottal korábban történt, semmissé válik, hiszen minden visszajön, amit elvileg már egyszer letöröltünk. Ennek a megoldásnak egyébként megvan az az előnye is, hogy az állapotok között vissza lépkedve, a láncszakadáson túl lépve, az összes többi állapothoz hasonlóan probléma nélkül vissza tudunk lépni bármely a vászon letakarítását megelőző állapotba, egészen a legelső állapotig. Ha a vászon letakarításánál egyszerűen csak eldobnám az állapotelőzményeket, nyilvánvalóan az is ugyanúgy megoldaná ez előbb említett problémát, viszont akkor nem lehetne visszalépkedni a program indása utáni legelső állapotig, mert azt/azokat ekkor elvesztenénk. Ha visszalépkedünk az állapotok között, majd módosítást végzünk egy korábbi állapoton, akkor azzal az adott korábbi állapot után beszúrjuk az új állapotot, és minden őt követő eddig tárolt állapotot törlünk, ahogy ezt a működést már más irodai, illetve képszerkesztő programok esetében megszoktuk.

3.2.1.2 Szabályos alakzatok rajzolása

A szabályos alakzatok, és a képek kirajzolása alapjaiban hasonlóan történik, ám van néhány különbség. Ezt a szabályos téglalap kirajzolásának példáján keresztül mutatom be, mert az összes többi alakzat rajzolása már lényegében ugyanúgy működik, mint a téglalapé. Ez azért van így, mert az egyenlő szárú háromszög, az ellipszis, és a kép mérete és pozíciója pontosan megadható ugyanazzal az algoritmussal, amivel meghatározzuk egy téglalap paramétereit is. Például a háromszög esetében húzunk egy téglalapot a bal egérgomb nyomva tartásával és a kurzor egyidejű mozgatásával, a program pedig felhasználja a téglalap felső egyenesének felezőpontját a háromszög felső csúcsának a meghatározásához, az alsó két csúcsát pedig megfelelteti téglalap alsó két csúcsának (lásd [19. ábra](#)). Az ellipszis megoldása szintén nagyon hasonló, a kép beszúrás esetén a kép méretének és pozíciójának meghatározása pedig teljesen megegyezik a téglalapéval.



Ábra 19: Téglalapba illesztett egyenlő szárú háromszög

A **téglalap rajzolása** esetében szintén felhasználjuk az egér lenyomásakor értéket kapó `lastPoint` adattagot. Ez lesz a téglalap egyik sarka. Így a téglalap pozícióját meghatároztuk, az egér mozgásával és a bal egérgomb nyomva tartásával ekkor meghatározhatjuk a magasságát és a szélességét a téglalapnak. Ekkor adódik egy probléma, amivel mindenképpen foglalkozni kell. Ha az egeret ekkor az egérgomb lenyomásával kijelölt pont fölé, és/vagy tőle balra húzzuk, akkor negatív szélességet és/vagy magasságot állítunk be a téglalapnak, mert a kezdőpontot alapból a téglalap bal felső sarkával azonosítjuk. Ezt a hibát küszöböli ki `fixRectangleNegativeWidthHeight(Rectangle)` függvény, ami átpozícionálja és méretezi a téglalapot attól függően, hogy milyen tulajdonságai lettek negatív értékűek. Ezekkel az új paraméterekkel már kirajzolható lesz a téglalap, de mégis olyan méretű lesz és olyan helyre kerül ami felhasználó számára intuitív. Ebben a rajzolási módban minden egyes kurzor mozdulatnál kirajzolódik egy új téglalap és az előző eltűnik, ezek a műveletek semmilyen formában nem jelennek meg, nem hagynak nyomot a hálózaton, és a `StateCaretaker`-ben sem, pusztán a felhasználó pontosabb munkáját segítik elő. Ahhoz hogy mindig csak és kizárólag az előző, ideiglenes téglalap tűnjön el, a program másolatot készít a `Canvas` raszteres képéről az egérgomb lenyomásakor, és elmenti a `currentSnapshot` adattagba. A következő egérmozdulatnál visszaállítja ezt a képet, amin még éppen nem volt rajta a téglalap. Erre már kirajzolhatja az új ideiglenes téglalapot, és így megkapjuk a téglalap aktuális előnézetét. Ezt az algoritmust a program addig ismételi, amíg el nem engedjük az egérgombot. Ha elengedjük az egérgombot, a program újra kirajzolja az egész állapotot, de ezúttal már a `Command`-okat használva. Az összes `Command` minden egérmozdulatnál való kirajzolása túl költségesnek

bizonyult, és a UI fagyásához vezetett. Ezért volt szükség az előbb bemutatott `currentSnapshot`-al megvalósított megoldásra.

Az egérgomb felengedésekor kirajzoljuk a végleges téglalapot, az ehhez tartozó egyetlen parancsot eltároljuk egy új `StateMemento` példányban, és ezt tovább is küldjük a hálózat többi kliensének a `Model` rétegen keresztül. A rajzolás folyamatát az egér lenyomástól az egérmozgatáson át az egér felengedéséig egy semaphore védi attól, hogy ezeken a kódrészekén egyszerre több szál legyen. Ez azért fontos, mert előfordulhat, hogy a hálózati késleltetés miatt, vagy a szálak ütemezéséből fakadóan úgy küldünk a többi kliensnek `signal`-t a következő mementó keletkezéséről, hogy az előzőt még nem zártuk le. Ennek hatására pedig a többi kliens összeomolhat.

3.2.1.3 RemoteDrawLineCommandBufferHandler

Ezen a ponton térnénk vissza a `RemoteDrawLineCommandBufferHandler` (RDLCBH) szerepére és működésére. Az hogy a tetszőleges vonal komponenseit nem egyszerre, egy egységben küldjük szét a hálózaton, akkor, amikor a belőle előálló `StateMemento` keletkezése befejeződött, problémát okoz akkor, amikor a rajzolást egy időben egyszerre több felhasználó teszi ugyanazon a vásznon. Fél kész `StateMemento`-t ugyanis nem szabad tárolni, és foglalkozni kell azzal is, hogy az egyszerre több felhasználótól beérkező parancsok nem ugyanahhoz a `StateMemento`-hoz tartoznak. Felhasználónként és állapotonként ezeket külön kell választani, de mégis egyszerre kell kirajzolni a tartalmukat, ahogy valós időben beérkeznek. Az RDLCBH pontosan ezt a problémát oldja meg. A szerepe hasonló a `StateOriginator`-éhoz, azzal a különbséggel, hogy nem állítja be egy tulajdonságát sem a `StateMemento`-nak, hiszen azt már egy másik kliens megtette. Egy `HashMap`-ban készít minden éppen készülő `StateMemento` számára egy-egy `ArrayList<Command>` példányt, és ezekben tárolja a beérkező parancsokat. Majd ha megkapja valamelyik készülő mementóhoz tartozó lezáró jelzést a készítő kienstől, akkor azt a mementót előállítja, kitörli a `HashMap`-ból a hozzá tartozó `ArrayList<Command>`-t, és átadja a `CanvasController`nek, aki továbbítja a `StateCaretaker`-nek. Innentől ugyanolyan `StateMemento`-nak minősül, mint ami helyben készül, azt leszámítva, hogy különbözik a `creatorID` adattagja a lokális kliens `UUID`-jétől. Így különválasztottuk, a párhuzamosan több felhasználótól érkező `StateMemento`-khoz tartozó parancsokat, és akkor kerültek be a vászon

StateCaretaker-jébe amikor elkészültek, mégis valós időben rajzolhattuk ki a készülő vonalat.

3.2.1.4 Állapotlánc konzisztencia

Annak érdekében, hogy a munka teljesen párhuzamos lehessen ugyanazon a vásznon és a hálózati késleltetés ne eredményezhessen inkonzisztens StateMemento láncokat a különböző kliensek között, minden StateMementoCloserSignal-ban értesíteni kell a többi klienst, hogy az éppen lezárt StateMemento pontosan hova kerül a láncban. Ez azonban önmagában sajnos nem oldja meg problémát. Még így is előfordulhat, hogy két kliens ugyanazon két StateMemento közé szeretne beszúrni új StateMemento-t. Ekkor kérdéses, a két új beszúrandó StateMemento sorrendje, hiszen nincs központi szerver, ami feloldja a konfliktust, és szinkronban tartott órákkal sem lehet pontosan eldönteni a sorrendet, mivel itt ezredmásodpercek dönthetnek a sorrendről. De ilyen esetben a tényleges sorrend nem is lényeges. A fontos az, hogy minden kliens egyértelműen, és pontosan ugyanúgy döntsön ilyen esetekben a sorrendről. Ehhez a StateMemento-k UUID-jén értelmezett teljes rendezés tulajdonságot használok ki. Az ilyen vitás esetekben az érintett StateMemento-kat az UUID compareTo(UUID) függvénye alapján minden kliens maga rendezi, így garantált, hogy a rendezés mindenkinél ugyan azt a sorrendet eredményezi. Elméletben előfordulhat az is, hogy mire egy távoli jelzés hatására egy állapotot be kell szúrni két másik állapot közé, addigra a két állapot már nem szomszédos többé egy másik kliens StateMemento-ja miatt. Ekkor az a kliens, amelyik detektálta ezt jelenséget, lemásolja az érintett vásznat, létrehoz egy új vásznat, és beleírja az érintett vászon általa eddig tárolt saját változatát, amelyet újra láncol, szétküldi az új vásznat, majd törli azt a vásznat amelyen a hiba történt. Így technikailag a kliensek megegyeztek a vászon egy állapotában.

3.2.1.5 ChatService

A Singleton ChatService osztály felel a szöveges üzenetek tárolásáért, azonnali megjelenítéséért, és küldéséért. Ez az osztály köti össze a Model-t a ChatWindowController-rel, ami a chat ablak vezérléséért felel. A frissen inicializált ChatWindowController példány a setChatWindowController(ChatWindowController) metóduson keresztül regisztrálja magát a ChatService-ben, majd megkéri a ChatService-t, hogy minden

korábbi üzenetet írasson ki a felületére. Mivel a `ChatService`-ben található kód akkor is futhat, amikor nincsen megnyitva a chat ablak, ez azzal a veszéllyel járhat, hogy a `chatWindowController_unsafe` adattag esetleg null értékű. Épp ezért ezt az adattagot elburkoltam a `ChatWindowControllerAccessor` objektumba. Ha a `chatWindowController_unsafe` adattag null értékű, akkor hozzáféréskor `ChatWindowControllerNotAvailableException` kivételt dob, így nem férhetünk hozzá ilyen állapotában. Mivel ilyenkor nincsen további teendőnk, így az erre a kivételre vonatkozó catch záradék üresen maradhat.

Minden `WindowController` osztály implementálja az `Initializable` interfészt. Az `initialize(URL, ResourceBundle)` metódus minden alkalommal lefut, amikor `Controller` létrejön az adott ablakhoz. A **`ChatWindowController`** például ezt arra használja, hogy regisztrálja magát a `ChatService`-ben, hogy az képes legyen kiírni rá a később érkező és a korábbi üzeneteket. Az üzenet ablakra való kiírásához a `printMessage(String,String,boolean)` metódust lehet használni. Az üzenet kiírásához szükség van a küldő nicknévére, az üzenetre, és arra hogy az illető a helyi felhasználó-e. Utóbbit a `boolean isMine` argumentummal kapja meg a metódus.

Egy érdekesség a `sendMessage()` eljárással kapcsolatban, hogy ha enterrel küldjük az üzenetet, akkor a `TextInput` példány beilleszti a sortörést az üzenetbe, mielőtt az osztály billentyű eseménykezelő függvénye lefutna. Mivel csak a `Shift+Enter` billentyűkombinációval szabadna sortörést beszúrni, ezért ezt a felesleges karaktert el kell távolítani az üzenetből. Ezt az eljárás úgy teszi meg, hogy lekéri kurzor pozícióját, mielőtt kiüríti a `TextInput` példányt, és feltételezve hogy az enter miatti felesleges sortörés oda került, a kurzor helyén levő karaktert eltávolítja az üzenet `ChatService` felé továbbítása előtt.

Egy másik probléma is felmerül a sortörések miatt. Mivel minden `String`-é kódolt `Command` és `Signal` sortöréssel zárul, ezért az üzenetek sortörés karakterét át kell alakítani valami más karakterré küldés előtt, majd fogadás után, de még kiírás előtt vissza kell alakítani sortöréssé. Ebben az esetben a `\n` karaktert `U+0001` UTF-8 vezérlőkarakterré alakítom, majd vissza. Ezt a két műveletet a `sendMessage()` és a `printMessage(String nickname, String message, boolean isMine)` metódusok végzik el.

3.2.1.6 Kliens lista

A **ClientsWindowController** gyűjti be a **NetworkService**-től a hálózat aktuális állapotával kapcsolatos információkat. Hogy aztán azokból egy rekurzív algoritmussal felépítse a hálózat `javafx.scene.control.TreeItem`-ekből álló strukturális másolatát, és kirajzolja azt egy `javafx.scene.control.TreeView`-n keresztül. Amíg az ablak nyitva van, a hálózatban történt változásokkor a tartalom automatikusan frissül. Ezt az Observer¹⁵ design pattern-el oldottam meg. A **ClientsWindowController** inicializáláskor feliratkozik a **NetworkService** direkt ebből a célból definiált eseményre, a **ClientEntityTreeChange**-re. Az ablak bezáródásakor pedig leiratkozik. Ehhez az eseményhez hoztam létre a **NotifiableClientEntityTreeChange** interfészt, ami garantálja a **NetworkService**-nek hogy a **ClientsWindowController** alkalmas az értesítés fogadására az `notifyClientEntityTreeChange()` eljárás meghívásán keresztül. A **NetworkService** a feliratkozott objektumokat a `clientEntityTreeChangeNotifiables` mezőjében tárolja, így ha később olyan fejlesztésre kerülne sor, ahol egy másik objektumnak esetleg szintén szüksége lenne arra, hogy értesítést kapjon erről az eseményről, akkor ő is feliratkozhat rá, amennyiben implementálja a megfelelő interfészt.

A hálózaton levő kliensek pingelése a **ClientPropertyWindowController**-en keresztül zajlik. Annak érdekében hogy a program felülete ne fagyjon le a pingelés idejére (ami mindig 4-1,5 másodperc), a mérést és a mért adatok kiírását külön szálla szerveztem, amit a **PingThread** beágyazott osztály valósít meg. Itt gyakran találkozhatunk a `runLater()` módszerrel, amely lehetővé teszi, hogy nem UI thread-ről is lehessen módosítani UI elemek értékeit. Ez a módszer a program több más pontján is felhasználásra került.

3.2.1.7 Kapcsolódási link

A **ConnectionLinkWindowController** felel annak az ablaknak a vezérléséért, ami a bejövő kapcsolatok fogadásának engedélyezésekor fogad minket. Ez az osztály végzi a kapcsolódási link generálását, ami hordozza a gép IP-címét és a rajta megnyitott **ServerSocket** port számát. Ezen kívül képes vágólapra helyezni a kapcsolódási linket, annak kényelmesebb továbbításához. A kapcsolódási linket a **ConnectionLink** osztály reprezentálja. A konstruktorán keresztül képes a `String`-ként kapott hivatkozást validálni és komponenseire bontani. Ez a speciális hivatkozás három komponensből áll.

Sorrendben: az `stconnect://` prefixumból, a futó kliens külső hálózati IP címéből, vagy internet kapcsolat hiánya esetén a helyi hálózati IP címből és ezután egy `:` szeparátort követő portszámból.

3.2.1.8 TabController

A vásznak menedzseléséért a `TabController` singleton osztály felel. Ezen a példányon keresztül lehet új lapokat nyitni új vászonnal, illetve ez kezeli ha valamelyik lap bezáródik akár helyileg akár távoli parancsra, vagy épp átneveződik. Ez az objektum tárolja az összes aktív `CanvasController` referenciát, így az ő `getCanvasController(UUID)` függvényén keresztül lehet hozzáférni ezekhez. Ha a program bezáródik akkor a benne definiált `TabController.stop()` eljárás felkészíti a leállásra az összes vászonhoz tartozó `CommandExecutorThread`-et. A `getActualCanvasController()` funkcióval lekérhető ahhoz a vászonhoz tartozó `CanvasController` amely éppen ki van választva a helyi felhasználó által.

Minden vászonhoz tartozik egy `KeyboardEventHandler` objektum, amit a `TabController` hoz létre és kapcsol össze a frissen létrejött vászonnal, a `initKeyboardEventHandler(Stage, CanvasController)` metódust használva. Ez az objektum felelős a billentyű parancsok jelentésének meghatározásáért, ez esetben az undo és redo parancsoknak. Linux és Windows platformok esetén azonban ezek a parancsok különböznek. A GNU/Linux disztribúciók esetében ugyanis a redo parancsot a szoftverekben általában a `Ctrl+Shift+Z` kombinációval adják ki a Windowsnál megszokott `Ctrl+Y` helyett. Így ezeket a parancsokat annak megfelelően kell azonosítani, hogy éppen milyen platformon fut a program. Mint kiderült, a módosító billentyűk hatására a Z billentyűt Linuxon adott esetben csak a `"U+001A"` UTF-8 karakterként lehet azonosítani, ezt teszi az `isZDown(KeyEvent)` függvény.

3.2.1.9 UserID

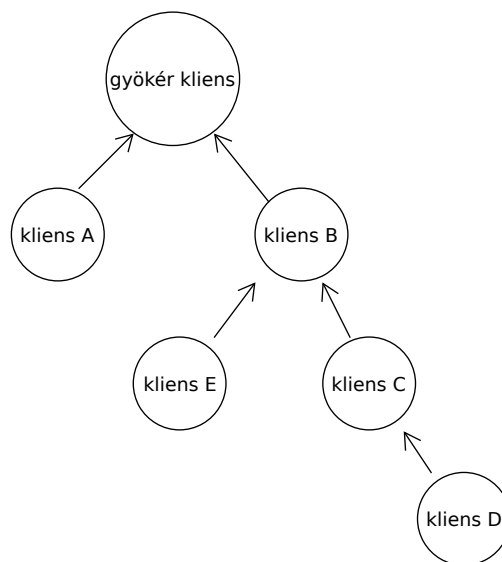
A helyi felhasználó adatainak tárolásáért a `UserID` singleton felel. Ilyen adatok a felhasználó nickneve, egyedi azonosítója, és IP címe. Ezeket az adatokat a `Model`-től kéri le, ahonnan első futás alkalmával megkapja az alapértelmezett adatokat, vagy a perzisztencia csomag által korábbi futás alkalmával letárolt információkat tölti be.

3.2.2. Model réteg

A Model réteg három csomagra oszlik. A network csomag a hálózati kapcsolatfelvétellel, kapcsolatlezárással és kommunikációval foglalkozik, a persistence csomag a felhasználói adatok fájlba kiírását, és az onnan való beolvasását, az ehhez szükséges mappa és fájl létrehozását végzi. A signals csomagban pedig azok az osztályok vannak, amik azokat a jelzéseket definiálják, amikkel minden egyéb olyan kommunikáció valósul meg ami nem a vászon tartalmával kapcsolatos műveletet ír le. Ilyenek például az új bejövő kapcsolat jelzése, az új szöveges üzenet jelzése, vagy a pingelési válaszkérelem. Ezeket a jelzéseket a network csomag generálja, küldi, fogadja, és értelmezi.

3.2.2.1 A hálózat működése

Mielőtt elkezdem részletesen taglalni a network csomag működését, szükségét érzem, hogy nagy vonalakban felvázoljam ennek a **hibrid peer-to-peer** hálózatnak a **működését**. Amennyiben a technikai feltételek adottak, minden kliens kapcsolódhat, vagyis kimenő kapcsolatot indíthat maximum egy másik klienshez, és minden kliens fogadhat egy vagy több bejövő kapcsolatot. Ha elképzeljük az így összekapcsolódó klienseket, akkor egyértelműen látszódik, hogy azok fába rendeződnek.



*Ábra 20: Egy hálózatba
összekapcsolódó kliensek*

Ezen a fán közlekedik minden információ kliensről kliensre. Ha egy kliens fölülről, a fa gyökere felől kap új információt, azt továbbítani kell minden hozzá kapcsolódó, alatta levő, a gyökér klienstől távolabbi kliens irányába. Ha alulról, kap információt akkor azt az információt minden hozzá közvetlenül kapcsolódó kliensnek továbbítja, kivéve azt a klienst, akitől az információt kapta. Így előbb utóbb mindenképpen eljut minden információ minden klienshez. Ez alól egy kivétel van, a [NewClientSignal](#): ez csak felfelé terjed, mivel ez a válasz csak a fa gyökérkliensének releváns. Értelemeszerűen minél távolabb helyezkedik el egy kliens az adott információ forrásától, annál később kapja azt meg. A gyökérkliens speciális szerepet tölt be a hálózatban. Az ő felelőssége, hogy felderítse a hálózatot minden újabb becsatlakozó, és lecsatlakozó kliensnél, és a hálózat részvevőiről, valamint azok helyzetéről informáljon mindenkit.

Fontos hogy minden kliens tisztában legyen a hálózat aktuális struktúrájával, mert csak így lehet megfelelően kezelni azt, hogy ha egy kliens kiesik a hálózathoz. Ekkor ugyanis adott esetben a fa többfelé szakadhat, és így nem csak az adott lecsatlakozó klienssel szakad meg a kapcsolat, hanem a fának azzal a részével is, amivel az eddig le nem csatlakozott kliens fenntartotta a kapcsolatot. Erre jó példa, ha elképzeljük, hogy a [20. ábrán](#) kiesik a B kliens. Ebben az esetben a fa három felé szakad. Az ilyen esetetek kezelésével részletesen a NetworkService-el foglalkozó [rész végén](#) fogok foglalkozni.

3.2.2.2 NetworkService

Az előbb leírt mechanizmusok megvalósításával alapvetően a **NetworkService** singleton osztály foglalkozik. Amikor a program többi komponense arra készen áll, akkor inicializálódik ez az objektum. Létrehozza a NetworkClientEntityTree-t, ami minden kliensnél a hálózat aktuális állapotát hivatott reprezentálni, és ehhez egyből hozzá is adja a mi kliensünket reprezentáló példányt. Ha engedélyezzük a bejövő kapcsolatok fogadását, akkor a program megpróbálja lefoglalni az alapértelmezett portokat. Ezek a 23243 és 23244-es portok. Előbbi Stringek fogadására és küldésére van fenntartva, utóbbi pedig a nyers bájt folyamatok fogadására. Ez esetben képek fogadására és küldésére használatos. Az elkülönítésre azért van szükség, mert egyébként speciális esetekben előfordulhat, hogy a Stringek formájában érkező parancsokat a nyers bájt folyamat olvasó BufferedReader olvassa le, vagy épp fordítva, a kép küldése során előállt bájt folyamat a Stringek fogadására beállított Scanner olvassa le.

Ha az előbbi két port éppen valamiért foglalt, akkor a rá következő két porttal próbálkozik újra. Minden kliens feltételezi, hogy ez a két port szomszédos.

3.2.2.3 Csatlakozás

Abban az esetben, ha a kliens egy helyi hálózati router mögött helyezkedik el, és interneten keresztül szeretne bejövő kapcsolatot fogadni, akkor a program megpróbálja **UPnP protokollon** keresztül megkérni a routert, hogy nyissa meg a tűzfalán a két portot, ehhez a `com.sharedtable.model.network.UPnP` csomagot használja a `prepareReceivingConnections(int)` metódus. Manapság ritka hogy erre ne lenne szükség, mert általában minden gép routeren keresztül csatlakozik az internetre. Ha ez nem működik, mert az UPnP nincs engedélyezve, vagy támogatva a routeren, akkor manuálisan kell beállítani a router operációs rendszerébe a portokra bejövő kapcsolatok átengedését (Port Forwarding).

Eközben inicializálódik a **ConnectionReceiverThread**, ami a nevének megfelelően a bejövő kapcsolatok fogadását végzi. Bejövő kapcsolat esetén létrehoz egy **ConnectedClientEntity** példányt, ami a közvetlen kapcsolatot kezeli. Mivel a **ConnectedClientEntity** a távolról érkező információk fogadását és feldolgozását már külön szálon végzi, így a **ConnectionReceiverThread** visszatérhet arra a pontra a programban, ahol várja a következő beérkező kapcsolatot. **Kimenő kapcsolat esetében** egyszerűen csak socket nyílik a kapcsolat-linkben megadott port felé (és a rá következő portjára). A kimenő kapcsolatok, és a bejövő kapcsolatok külön mezőkben tárolódnak (`upperConnectedClientEntity`, és `lowerConnectedClientEntities`) **NetworkService**-ben, mert így könnyebben lehet implementálni a [fentebb részletezett](#) információ továbbítási algoritmust. Általánosságban kijelenthető, hogy ha egy port tárolására hivatott változó bárhol a programkódban **-1**, vagy erre az értékre tesztelődik, akkor az azt jelenti, hogy az a kliens, amire ez az információ vonatkozik alkalmatlan, vagy azt vizsgáljuk, hogy alkalmatlan-e bejövő kapcsolatok fogadására, hiszen ilyen port nem létezhet.

3.2.2.4 Jelzések

Mivel a további funkciók bemutatásában fontos szerepet játszanak a jelzések (signal-ok), ezért egy bekezdést most szánnék rájuk, mielőtt folytatnám a **network** csomag bemutatását. Minden jelzés saját osztályt kapott a **signals** csomagon belül, és

minden jelzés implementálja a `Signal` interfészt. A jelzéseknek jellemzően két konstruktora van. Az egyiket keresztül az adattagok értékét lehet megadni, és az adattagok típusával megegyezők ennek a konstruktornak az argumentumai. Ezt a konstruktort általában a jelzés feladója használja, hogy küldés előtt létrehozza az objektumot. Küldéskor a felüldefiniált `toString()` függvénnyel egyetlen stringé alakítja a jelzést, ami lehetővé teszi annak egy sorban való továbbítását a megfelelő Socket-eken. Ezután a másik fél a megfelelő jelzés `String[]` típusú argumentumot váró konstruktort használva rekonstruálja a jelzést a másik oldalon. Ezt a rekonstrukciót a `SignalFactory` osztály végzi. A jelzések String-ként egyszerűen azonosítani lehet: minden jelzés, és parancs ; karakterekkel van tagolva, és minden jelzés első tagja a **SIG** substring. A második tag pedig minden esetben a jelzés rövid azonosítója. Például a szöveges üzenet jelzés azonosítója a **CHAT**. Minden jelzés tartalmazza készítőjének azonosítóját is. A további tagok a jelzés egyéb paraméterei, a paraméterek száma jelzésenként változó. Az összes jelzés, és szerepük:

- `ChatMessageSignal`: szöveges üzenetek továbbítására alkalmas, amik a chat ablakban jelennek meg. Hordozza az üzenet azonosítót, a feladó nicknevét, és magát az üzenetet. (azonosító: CHAT)
- `NewTabSignal`: Egy új lap és a rajta levő vászon létrehozására utasít a megadott néven. (azonosító: NEWTAB)
- `RenameTabSignal`: Egy már létező lap nevének megváltoztatására utasít. (azonosító: TABRENAME)
- `CloseTabSignal`: Egy lap, és a rajta levő vászon bezárására, és törlésére utasít. Tartalmazza a vászon egyedi azonosítóját. (azonosító: CLOSETAB)
- `DeleteAfterSignal`: Egy adott memento utáni összes memento törlésére utasít. Akkor kerül kiadásra, amikor egy kliensnél egy a legfrissebbnél korábbi mementón történik változtatás. Tehát **szándékos** `StateMemento` beszúrás történik két `StateMemento` közé. (azonosító: DELAFTER)
- `DisconnectSignal`: Akkor kerül kiküldésre, amikor egy kliens észleli, hogy egy közvetlen gyerekével megszakadt a kapcsolata. Ennek hatására a hálózat gyökérkliense újra felderíti a hálózatot, hiszen nem lehet tudni, hogy a hálózat mekkora részével szakadt meg a kapcsolat a kieső kliens miatt. Ezután informál minden klienst az új hálózatról. (azonosító: DISCONN)

- **DiscoverySignal:** Ezt a jelzést csak a gyökérkliens küldheti ki, ezzel deríti fel a teljes hálózatot. A felderített hálózatról informálja a hálózat összes kliensét az **EntityTreeSignal** segítségével. (azonosító: DISCOV)
- **EntityTreeSignal:** Ezt a jelzést szintén csak a gyökérkliens küldheti ki. Ebbe bele van kódolva a gyökérkliens által rögzített teljes **NetworkClientEntityTree** minden kliense. Így a jelzést megkapók rekonstruálni tudják ezt a mezőt, és a fa állapota szinkronban marad a kliensek között. (azonosító: TREE)
- **NewClientSignal:** Ezt a jelzést mindig az a kliens adja ki, akihez csatlakozott egy új kliens, akkor, amikor csatlakozást követő szinkronizáció megtörtént. Mivel ez a jelzés csak a gyökérkliensnek releváns, ezért csak felfelé kerül továbbításra. (azonosító: CONN)
- **MementoOpenerSignal:** Azt jelzi, hogy egy adott kliensnél egy új **StateMemento** rögzítése elkezdődött egy adott vásznon. (azonosító: OPEN)
- **MementoCloserSignal:** Azt jelzi, hogy egy adott kliensnél egy új **StateMemento** rögzítése befejeződött, a **prevMementoID** által meghatározott **StateMemento**-t követi, a **nextMementoID** azonosítóval rendelkező memento a rá következője, és informál arról is, hogy az adott **StateMemento**-t láncolni kell-e. Ekkor a **RemoteDrawLineCommandBufferHandler**-ben megszakad a parancsok gyűjtése, és a megfelelő paraméterek átadása után az adott **StateCaretaker**-ben létrejön és elhelyeződik az információk alapján az új **StateMemento**. (azonosító: CLOSE)
- **NetworkPasswordChangeSignal:** hatására megváltozik a kapcsolódáshoz szükséges jelszó minden kliensnél. Az alapértelmezett jelszó mindig **NO_PASSWORD**, a jelszó mezők üresen hagyásakor is ez az érték állítódik be. Ezt a konstanszt a **Constants** osztályból kérhetjük le. (azonosító: PASSWD)
- **PingSignal:** A hálózaton direkt vagy indirekt kapcsolatban levő két kliens közötti válaszidő mérésére szolgál. A pingelési kérelem a hálózat minden részvevőjéhez eljut, de csak a célzott kliens (**UUID targetClientID**) ad rá választ. (azonosító: PING)

3.2.2.5 Szinkronizáció

Miután a `ConnectionReceiverThread` fogadta a kapcsolatot, azután a kapcsolódás további teendőiről a `ConnectedClientEntity` kódja gondoskodik. Innentől ezt a klienst egy külön szál szolgálja ki. A frissen indult szál első feladata handshaking process elvégzése. Az előkészített csatornákon keresztül először a csatlakozó kliens elküldi a saját magára vonatkozó információkat, mint például a kliens nickneve, egyedi azonosítója, nyitott bejövő kapcsolatok fogadására alkalmas portja, ha van, stb. Egy szóval bemutatkozik. Amint ezzel végzett, a kapcsolatot fogadó fél, vagyis a felső kliens teszi meg ugyanezt. Ezután a kapcsolatot kezdeményező fél, az alsó kliens elküldi a hálózati jelszót, amit a felső kliens ellenőriz, majd ennek helyességével kapcsolatban választ ad a felső kliens.

Ezután következik a **szinkronizáció**. Mindkét fél elküldi a nála éppen nyitva levő vásznakkal kapcsolatos összes információt. Először csak a vásznakkal kapcsolatos adatokat, a `NewTabSignal` segítségével, hogy minden kliensnél előkészítésre kerüljenek a vásznak mielőtt az állapotok átvitele megkezdődik. Utána az összes `StateMemento`, és a bennük tárolt összes `Command` szinkronizációja következik.

A `StateMemento`-kat fogadó félnél levő `RemoteDrawLineCommandBufferHandler` számára a `MementoOpenerSignal` jelzi, hogy az adott távoli felhasználó részére új puffert kell nyitni. Ezután átküldésre kerülnek a `Command`ok, és végül, ha minden `Command` kiküldésre került, `MementoCloserSignal`-al közöljük a `StateMemento`-val kapcsolatos egyéb információkat, és jelezzük, hogy az előbb megnyitott puffert meg lehet semmisíteni, a `memento`-t leíró információk a `CanvasController`-nek átadhatóak, hogy létrehozza belőle az állapotot és elhelyezze azt a `StateCaretaker`-ben. Az előbbi algoritmus addig ismétlődik, míg minden `memento` átküldésre nem került. Mivel ezeket az információkat mindkét szinkronizációban résztvevő fél a megfelelő irányokba folyamatosan továbbítja, ezért nem csak ez a két kliens, hanem az egész hálózat szinkronba áll. A szinkronizációs adatküldés végét mindkét fél egy „SYNCED” üzenettel zárja. Itt először a felső kliens küldi el az adatokat, és csak utána az alsó. Miután az alsó kliens is elküldte a záró üzenetet, a felső kliens üzenetet küld (`NewClientSignal`-t) a gyökérkliens felé a hálózat új tagjáról, aki erről informálja a hálózatot. Végül a két kliens ellenőrzi, hogy a verziók megegyezők. Ha nem, akkor a régebbi kliens értesíti a felhasználót, hogy az eltérő verzió miatt nem várt viselkedés lehetséges.

A handshaking process és a szinkronizáció alatt a `ConnectedClientEntity` a `sendLockSemaphore`-ral blokkol minden olyan szálát, ami rajta keresztül szeretne üzeni az általa reprezentált kliensnek. Így nem történhet meg, hogy egy harmadik klienstől kap a szinkronizáció alatt álló kliens olyan információt, ami olyan adatra, vászonra vagy állapotra hivatkozik, amivel a kliens még nem rendelkezik. Ez alatt az idő alatt a szinkronizáció az *unsafe* prefixummal ellátott üzenetküldő metóduson keresztül zajlik, amik `private` láthatóságúak, és nem védi őket a semaphore. Ez a semaphore ráadásul véd attól, hogy egyszerre több szál próbáljon írni a kimeneti pufferbe, ezzel összemosott üzeneteket eredményezve.

Ha a kapcsolódás eredményeképpen két hálózat kapcsolódik össze, akkor a semaphore elengedése után a régi gyökér a gyerekeinek küld egy discovery signal-t. Ez azért fontos, mert ugyan az új csatlakozó félről, a másik hálózat régi gyökeréről tájékoztatva lett az új gyökér, de az ő gyerekeiről nem. A discovery signal-ra válaszként ezek a gyerekek is bemutatkoznak az új gyökérnek.

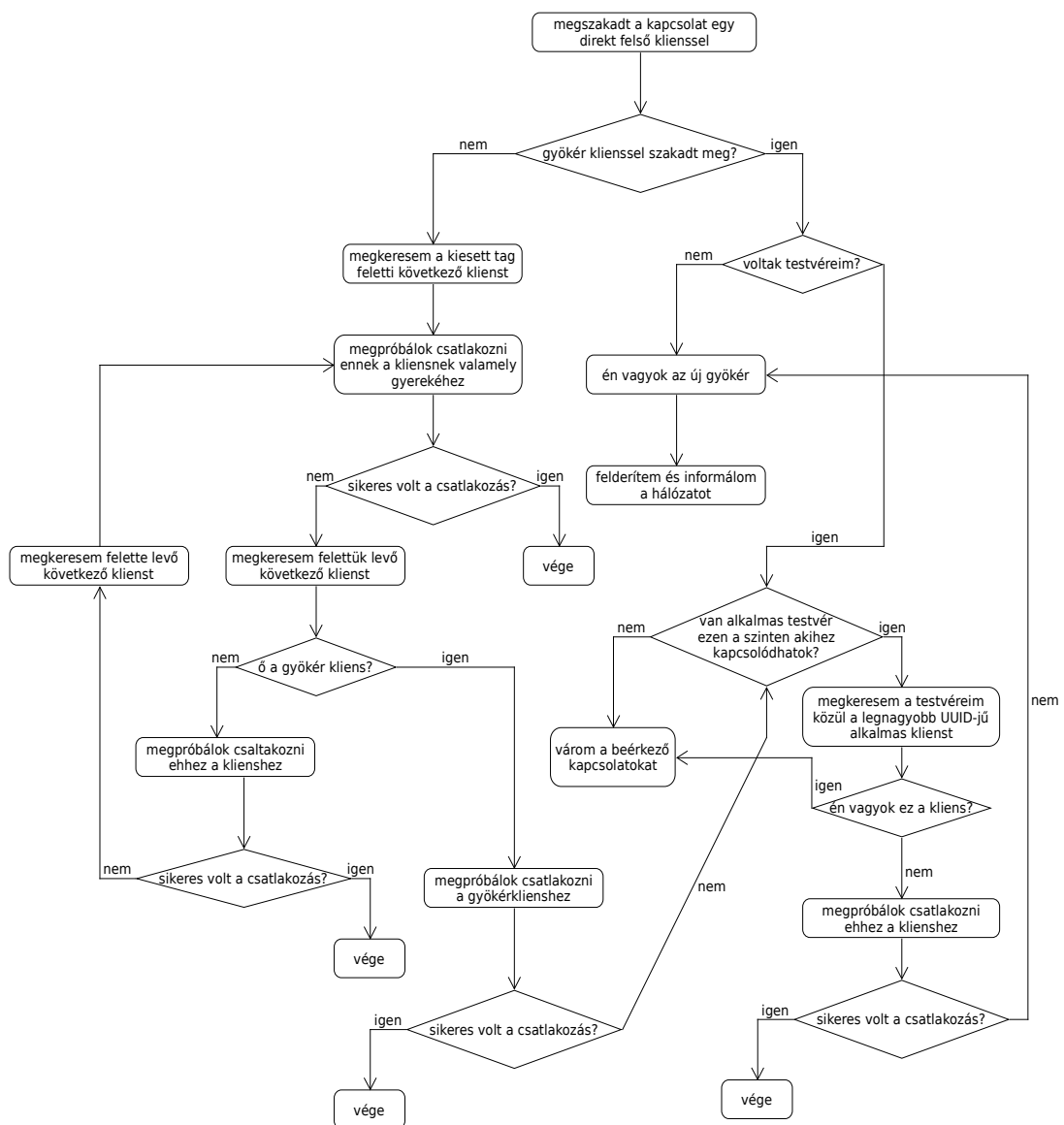
Ezután az eddig a szinkronizációt végző szál, életciklusának végéig, vagyis ameddig a kapcsolat fenn áll az adott klienssel, az adott klienstől érkező üzenetek fogadásával, feldolgozásával, és továbbításával foglalkozik. Minden üzenet, legyen az `Signal` vagy `Command`, sortöréssel végződő, és pontosvesszőkkel tagolt stringként érkezik. A `Command`-ok a `Signal`-okhoz hasonlóan saját `String[]` argumentumot váró konstruktorral rendelkeznek, ami képes létrehozni a megfelelő `Command` altípusú példányt az előbb említett egysoros string-ből. A felüldefiniált `toString()` függvényük ugyanazt a célt szolgálja, mint a `Signal`-ok esetében.

3.2.2.6 Pingelés

A kliensek pingelését a `NetworkService pingClient(UUID)` függvénye valósítja meg. Először szétküldi a hálózaton a `PingSignal`-t, majd felkészíti az összes `ConnectedClientEntity` példányt, hogy várja az adott ping azonosítóval rendelkező válasz beérkezését, hiszen bármilyen irányból érkezhetsz a válasz, mivel minden irányba szétküldtük a jelzést. A `ConnectedClientEntity`, amennyiben a várt azonosítójú ping kérelemre érkezett válasz, rögzíti a válasz idejét, és visszaadja a `NetworkService`-nek. Ezután a válaszütemet megkapjuk az érkezés idejének és a kiküldés idejének különbségéből. A `Controller` réteg `ClientPropertyWindowController` négy ilyen mérés átlagolásával számolja a válaszütemet, és a részeredményeket is kiírja valós időben.

3.2.2.7 Újracsatlakozás

A NetworkService másik nagy feladata a felső kapcsolat megszakadásából következő hálózati szakadás helyreállításának megoldása. Ahogy arról már [korábban](#) írtam ez azért fontos, mert ha két olyan kliens között szakad meg a kapcsolat, akiken keresztül más kliensek is fenntartották a kapcsolatot, akkor nem feltétlenül csak a hálózathoz kieső féllel szakad meg a kapcsolat. Ennek a megoldása ebben a hálózati topológiában nem triviális, de jól bemutatja a program módszerét a megoldásra az alábbi folyamatdiagram.



Ábra 21: Újracsatlakozás folyamata

3.2.3. A View réteg

A view réteg feladata, hogy gondoskodjon az ablakok megfelelő megjelenítéséről, adott esetben hozzáférést biztosítson az adott ablakhoz tartozó Controller példányhoz, amelyből általában hozzá lehet férni az ablakba bevitt információkhoz, vagy amelyen keresztül módosítani lehet az ablak tartalmát.

Az összes View végződésű réteg (a MainView kivételével) a GeneralView osztályból származik, mert ezeknek az ablakoknak a kirajzolása és beállítása csak néhány paraméterben különbözik egymástól. A GeneralView konstruktorán keresztül be lehet állítani: hogy az ablak felületét melyik [FXML fájl](#) tárolja, hogy modális-e az adott ablak, vagyis hogy hozzá lehet-e férni a többi ablakhoz amíg az adott ablak nyitva van, van-e szülője az ablaknak ami ha bezáródik akkor az adott ablaknak is be kell záródnia, illetve, hogy mi legyen az ablak címe. Mindössze két ablak nem modális: a chat ablak, és a kliens fát megjelenítő ablak.

Ebben a csomagban van továbbá az STCanvas, STTab, és STTabPane osztályok, amelyek a nevükből kikövetkeztető JavaFX osztályokból származtatottak. A származtatásra azért volt szükség, hogy megfelelően lehessen elhelyezni az ezekhez a vezérlőkhöz tartozó eseménykezelő, és műveletvégző függvényeket. Így ezek az osztályok tartalmazzák a vezérlőknek szükséges események továbbítását végző, EventHandler-eket, és az STCanvas esetében pedig a legalapvetőbb rajzolóműveletek implementálása is itt történt. Az STCanvas megkapja argumentumként inicializáláskor a hozzá tartozó CanvasControllert, így képes annak továbbítani a rajta fellépő eseményeket és paramétereiket. Az STTab esetében elengedhetetlen volt ez a megoldás, mert ott máshogy nem is lehet hozzáférni a lap bezárásakor létrejövő eseményhez. Az STTabPane pedig a lapok nyilvántartását, és a lapokon végrehajtható műveleteket megvalósító eljárásokat tartalmazza.

Említésre érdemes még ebben a csomagban a MessageBox statikus osztály, amely a figyelmeztető és információs ablakok megjelenítését végző metódusokat tartalmazza.

3.3. Tesztelés

A tesztelés alapvetően kétféleképpen történt a fejlesztés során. A fejlesztés korai és késői szakaszában ún. Manuális Smart Monkey Test módszerrel, és a késői szakaszban a JUnit5 keretrendszerrel megvalósított Unit testeken keresztül.

A szoftvernek viszonylag kevés olyan része van, ami elég komplex, és ráadásul alkalmas arra, hogy **Unit** tesztelni kelljen vagy lehessen. Ez leginkább azért van, mert a szoftver legkomplexebb része, a rajzolás, inputja UI-on keresztül történik, és a Canvas pixeltartalmát a bevitt inputoknak megfelelően nagyon körülményes lenne automatizáltan tesztelni. Tovább nehezíti a problémát az a tény, hogy a másik összetett része a rendszernek az, hogy az összes rajzolás minden kliens között szinkronban történik, hálózati kommunikáción keresztül.

A kontroller csomag három komponensének bizonyos részei viszont alkalmasak voltak erre a típusú tesztelésre. A **StateCaretaker** egésze nagyon jól tesztelhető, és ráadásul ez a módszer kivételesen alkalmasnak bizonyult olyan állapotok előidézésére, amik csak extrém esetekben jöhetnek létre megfelelő számú kliens, megfelelő input megfelelő időben való kiadása esetén. Ilyen például az az eset, amikor két kliens ugyanazon két állapot közé szeretne StateMemento-t beszúrni. A **NetworkService** azon része, amelyik a Model alfejezetben [részletezett](#) újrakapcsolódásért felel, szintén jól tesztelhető volt, ami nagyban felgyorsította a fejlesztést, hiszen manuális tesztelés esetén több, néha 5-6 példányt kellett futtatni a szoftverből, és minden alkalommal az adott tesztesetnek megfelelően össze kellett őket csatlakoztatni, mielőtt a tényleges tesztet el lehetett végezni. Ez rendkívül időigényes folyamat.

A **Manuális Smart Monkey Test**¹⁷ egyszerűbb esete, ahol a legtöbb tesztet el lehetett végezni, amikor egyszerűen két példányt futtatva, majd egy hálózatba összekapcsolva, egyesével tesztelni lehet az összes funkciót. Azonban a párhuzamos munkát egyetlen gépen nem lehet tesztelni, hiszen egy operációs rendszerben egyszerre csak egy ablak kaphat fókuszt, és csak egyetlen egérkurzor áll rendelkezésünkre. Itt kénytelenek vagyunk egy másik számítógépre is telepíteni az alkalmazást minden kódváltoztatásnál újra és újra, majd helyi hálózaton összekapcsolni a minimum két klienst, és egyszerre két egérrel párhuzamosan használni a szoftver funkcióit. És ezeket a tesztek számos hiba kijavításához szükség volt úgy is lefuttatni, hogy különböző operációs rendszereken futottak az összekapcsolt kliensek. A `-Dfile.encoding=UTF-8` kapcsoló [szükségességére](#) is csak az ilyen úton történő tesztelés tudott rávilágítani.

Mivel az egyedi azonosítót, és a nicknevet a program a minden futó példány ugyanabból a fájlból olvassa ki, ezért ha egy rendszer alatt szeretnénk több futó példányt összekapcsolni tesztelés céljából, használnunk kell a `config-path [útvonal]`

parancssori argumentumot futtatáskor, hogy minden futó példány saját userconfig.json fájlt használjon. Ellenkező esetben minden az első példány után induló program ugyanazt az egyedi felhasználói azonosítót fogja használni, ami nem várt viselkedést eredményez. Ezt Gradle-lel való futtatás esetén például a `./gradlew run --args 'config-path ./ucfg2'` paranccsal tehetjük meg.

A Unit testeket a `./gradlew test` paranccsal futtathatjuk le.

3.3.1. Tesztesetek

1. Kliensek összekapcsolása

- Leírás: Három kliensnek sikeresen össze kell kapcsolódnia, és szinkronba kell állnia.
- Előfeltétel: Az operációs rendszernek készen kell állnia a program futtatására.
- Tesztelési lépések:
 1. Nyissuk meg a program első példányát a `./gradlew run --args 'config-path ./ucfg1'` paranccsal.
 2. Nyissuk meg a program második példányát a `./gradlew run --args 'config-path ./ucfg2'` paranccsal.
 3. Nyissuk meg a program harmadik példányát a `./gradlew run --args 'config-path ./ucfg3'` paranccsal.
 4. Adjunk meg mindegyik példányban egy tetszőleges nicknevet.
 5. Az első példány vásznára rajzoljunk egy tetszőleges vonalat a bal egérgomb nyomva tartásával.
 6. Az első példányban engedélyezzük a bejövő kapcsolatok fogadását (Hálózat -> bejövő kapcsolatok engedélyezése)
 7. A második példányt [kapcsoljuk össze](#) az első példánnyal, az első példányból [kinyert](#) kapcsolódási linkkel.
 8. A harmadik példányt [kapcsoljuk össze](#) az első példánnyal, az első példányból [kinyert](#) kapcsolódási linkkel.
- Várt eredmény:
 - Három különböző vászon van megnyitva mindhárom kliensben, ugyanazzal a névvel

- Mindhárom vászon tartalma a nevüknek megfelelően ugyanaz. Tehát egy vásznon van egy tetszőleges vonal mindhárom kliensnél, és ez a vonal mindhárom kliensnél ugyanúgy néz ki, és ugyanott helyezkedik el.

2. Vászna bezárása és megnyitása

- Leírás: A három kliens között szinkronban kell bezáródni és megnyílnia új vásznaknak.
- Előfeltétel: Abban az állapotban vagyunk mindhárom futó klienssel, ahol végeztünk az 1. számú tesztessel.
- Tesztelési lépések:
 1. Zárjuk be a három megnyitott vászon közül tetszőleges kettőt. Az első választott vásznat a második kliensben, a második választott vásznat az első kliensben.
 2. Hozzunk létre egy [új vásznat](#)! (Vászon -> Új vászon)
 3. Minden kliensen állítsuk át a kiválasztott vásznat az újonnan létrehozott vászonra.
- Várt eredmény: Mindhárom kliensben két vászon van megnyitva ugyanazzal a névvel, és mindhárom kliensben ugyanaz a vászon van kiválasztva.

3. Rajzolás

- Leírás: A kliensek között szinkronban kell megtörténnie minden rajzolásnak
- Előfeltétel: Abban az állapotban vagyunk mindhárom futó klienssel, ahol végeztünk az 2. számú tesztessel.
- Tesztelési lépések:
 1. Válasszuk ki a két megnyitott vászon közül az üreset, mindhárom kliensben.
 2. Válasszuk ki a harmadik klienst.
 3. Válasszuk ki a Rajzolási módok -> Téglalap opciót.
 4. A bal egérgomb nyomva tartásával rajzoljunk egy téglalapot.
 5. Válasszuk ki a Rajzolási módok -> Ellipszis opciót.
 6. A bal egérgomb nyomva tartásával rajzoljunk egy ellipszist.
 7. Válasszuk ki a Rajzolási módok -> Folytonos vonal opciót.
 8. [Válasszunk](#) egy feketétől különböző színt.
 9. Válasszunk egy egynél nagyobb vonalvastagságot.

10. Húzzunk egy vonalat.

- Várt eredmény: Mindhárom kliensben ugyanazt az ellipszist és ugyanazt a téglalapot látjuk, és ugyanazt a vonalat látjuk, a választott színnel és vonalvastagsággal.

4. Undo

- Leírás: A visszavonás hatására vissza kell lépnie a vászonnak egyel korábbi állapotba.
- Előfeltétel: Abban az állapotban vagyunk mindhárom futó klienssel, ahol végeztünk az 3. számú tesztesettel.
- Tesztelési lépések:
 1. Kiválasztunk egy tetszőleges klienst.
 2. A megfelelő billentyűkombinációval, vagy menüpont kiválasztásával visszavonunk kétszer.
- Várt eredmény: Mindhárom kliensben eltűnik a vonal és az ellipszis, csak a téglalap látszódik.

5. Redo

1. Leírás: A vászonnak előre kell lépnie időben az állapotok között.
2. Előfeltétel: Abban az állapotban vagyunk mindhárom futó klienssel, ahol végeztünk az 4. számú tesztesettel.
3. Tesztelési lépések:
 1. Kiválasztunk egy tetszőleges klienst.
 2. A megfelelő billentyűkombinációval, vagy menüpont kiválasztásával visszavonjuk a visszavonást kétszer.
4. Várt eredmény: Mindhárom kliensben visszatér a vonal és az ellipszis, pontosan ugyanazt látjuk mindhárom kliensben, mint a 3. teszteset végén.

6. Hálózati kliens fa

- Leírás: A klienseknek tisztában kell lennie a hálózat aktuális állapotával.
- Előfeltétel: Mindhárom kliens (továbbra is) kapcsolatban van az első tesztesetnek megfelelően.
- Tesztelési lépések:
 1. Az első két kliensben válasszuk ki a Hálózat -> Hálózati kliens fa opciót.

2. Zárjuk be a harmadik klienst.

- Várt eredmény:
 - Mindhárom kliens Hálózati kliens fa ablakában ugyanaz a fa látható, most már két klienssel
 - Mindegyik kliens fában a megfelelő kliens van kiemelve.

7. Újracsatlakozás

- Leírás: Teszteljük a hálózati fa szétesése elleni védelmet a gyökérkliens kiejtésével.
- Előfeltétel: Abban az állapotban vagyunk mindkét futó klienssel, ahol végeztünk az 6. számú tesztesettel.
- Tesztelési lépések:
 1. Nyissuk meg a program harmadik példányát a `./gradlew run --args 'config-path ./ucfg3'` paranccsal.
 2. Engedélyezzük ebben a kliensben a bejövő kapcsolatok fogadását.
 3. A harmadik példányt [kapcsoljuk össze](#) az első példánnyal, az első példányból [kinyert](#) kapcsolódási linkkel.
 4. Zárjuk be a fa gyökerét képző első klienst.
- Várt eredmény: UUID értékekétől függően a maradék két kliens automatikusan összekapcsolódott.

8. Chat

- Leírás: Az üzeneteknek meg kell érkeznie valós időben minden klienshez, és az ablak megnyitásakor meg kell jelennie minden korábbi üzenetnek.
- Előfeltétel: Mindhárom kliens kapcsolatban van az első tesztesethez hasonló módon.
- Tesztelési lépések:
 1. Az első két kliensben kiválasztjuk a Chat -> Beszélgetés megnyitása opciót.
 2. Az első kliensben elküldünk egy tetszőleges üzenetet.
 3. A második kliensben elküldünk egy tetszőleges üzenetet.
 4. A harmadik kliensben kiválasztjuk a Chat -> Beszélgetés megnyitása opciót.

- Várt eredmény: Mindhárom kliens chat ablakában ugyanazokat az üzeneteket látjuk ugyanabban a sorrendben.

3.4. További fejlesztés

A program a kitűzött funkcióit így is képes végrehajtani, számos szempontból működhetne effektívebben az alkalmazás.

- A kliensek hálózati struktúrában előállt távolságával egyenesen arányosan nő a válaszidő, hiszen minél messzebb vannak egymástól, annál több klienseknek kell a célállomásig továbbítani a parancsokat. A fa topológiát le lehetne cserélni teljes topológiára, de ehhez egy sokkal összetettebb adattovábbító algoritmusra lenne szükség.
- Mivel a tetszőleges vonal sok apró kis egyenes vonal kirajzolásával jön létre, egy bizonyos vonalvastagság után torzulni kezd a vonal. Ezt meg lehetne oldani például úgy, hogy az egyenes vonalakat ebben az esetben lecseréljük apró, vonalvastagságnak megfelelő méretű telített körökre.
- Az UPnP protokollt használatát le lehetne cserélni egy más megoldással, ami a router beállításaitól függetlenül mindig működik. Ilyen megoldás lenne például a TURN (Traversal Using Relays around NAT) protokoll használata.
- Ha egy felhasználó nagyobb ablakméretet használ, mint a másik, akkor a kisebb ablakban futó kliens nem képes megjeleníteni az adott tartalmat egészen addig, amíg az megfelelő szélességűre nem lesz állítva, mert a vászon olyan részére rajzol ilyenkor, ahová a kisebb ablak esetében nem történik renderelés, hiszen nem is látszik. Ezt a problémát meg lehetne oldani normalizált koordináták használatával.
- Noha sok rajzolási lehetőség áll rendelkezésre az alkalmazásban, szerkesztési opciókban hiányosságokat szenvedett. Több szerkesztést igénylő probléma könnyebben megoldható lenne, ha a program rendelkezne a négyzetes kijelölés, átméretezés, és mozgatás lehetőségével.

3.5. Összegzés

Az [előbb](#) részletezett fejlesztési lehetőségek ellenére a program megoldást nyújt a bevezetésben bemutatott problémára. Valós időben lehet rajzolni, üzeneteket küldeni, képet beilleszteni vágólapról és fájlból is. Lehetővé teszi a teljesen párhuzamos munkát, akár egyetlen vásznon is. Támogatja az állapotalapú visszavonást. Ha a hálózathoz kiesik egy tag, akkor a megfelelő kliensek automatikusan újracsatlakoznak az alkalmas klienshez, ezzel megakadályozva a hálózat szétesését. A hálózat védhető jelszóval, struktúrája megjeleníthető, kliensei pingelhetők.

A kód jól tagolt, és könnyen adhatóak hozzá új funkciók a megfelelő részek szerkesztésével. Az új verziók könnyen terjeszthetők a jlink, a Windows Installer, és a Debian csomag segítségével. A program értesíti a felhasználókat az új frissítésekről, és azok új funkcióiról.

4. Irodalomjegyzék

1. Java Platform, Standard Edition 14 Reference Implementations [Online] [Elérve: 2020.04.20.]
<https://jdk.java.net/java-se-ri/14>
2. Java Platform, Standard Edition Tools Reference jlink [Online] [Elérve: 2020.04.20.]
<https://docs.oracle.com/javase/9/tools/jlink.htm>
3. Understanding Java 9 Modules [Online] [Elérve: 2020.04.20.]
<https://www.oracle.com/corporate/features/understanding-java-9-modules.html>
4. JavaFX – Gluon [Online] [Elérve: 2020.04.20.]
<https://gluonhq.com/products/javafx/>
5. JavaFX Tutorial [Online] [Elérve: 2020.04.20.]
<http://tutorials.jenkov.com/javafx/index.html>
6. Gradle User Manual [Online] [Elérve: 2020.09.29.]
<https://docs.gradle.org/current/userguide/userguide.html>
7. Gradle | Releases [Online] [Elérve: 2020.09.29.]
<https://gradle.org/releases/>
8. adolfintel/WaifUPnP [Online] [Elérve: 2020.04.10]
<https://github.com/adolfinet/WaifUPnP>
9. fangyidong/json-simple [Online] [Elérve: 2020.04.10]
<https://github.com/fangyidong/json-simple>
10. NSIS Users Manual [Online] [Elérve: 2020.04.15.]
<https://nsis.sourceforge.io/Docs/>
11. Download – NSIS [Online] [Elérve: 2020.04.15.]
<https://nsis.sourceforge.io/Download>
12. A Guide to JUnit 5 [Online] [Elérve: 2020.10.01.]
<https://www.baeldung.com/junit-5>
13. Design Patterns - Memento Pattern [Online] [Elérve: 2020.04.20.]
https://www.tutorialspoint.com/design_pattern/memento_pattern.htm
14. Design Patterns - Command Pattern [Online] [Elérve: 2020.04.20.]
https://www.tutorialspoint.com/design_pattern/command_pattern.htm

15. Design Patterns - Observer Pattern [Online] [Elérve: 2020.04.13.]
https://www.tutorialspoint.com/design_pattern/observer_pattern.htm
16. Design Pattern - Singleton Pattern [Online] [Elérve: 2020.04.20.]
https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm
17. What is Monkey & Gorilla Testing? Examples, Difference [Online] [Elérve: 2020.10.13.]
<https://www.guru99.com/monkey-testing.html>