

**Budapesti Műszaki és Gazdaságtudományi Egyetem**

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Frank Dávid

**ELOSZTOTT TÁBLÁZATKEZELŐ WEBALKALMAZÁS SZÁMÍTÁSINTENZÍV FELADATOKHOZ**

Önálló laboratórium

KONZULENSEK

Pataki István (Morgan Stanley)

Szöllösi Loránd (Morgan Stanley)

Imre Gábor (BME AUT)

BUDAPEST, 2014

# Bevezetés

Az elmúlt években több ingyenes és üzleti táblázatkezelő alkalmazás jelent meg a weben, ezek közül a legelterjedtebb a Google Docs Spreadsheet. Ezek mind támogatják a felhasználók közötti valósidejű szinkronizációt, ám a számítások elvégzése minden gépen különállóan hajtódik végre. A Google megoldása a számításokat a felhasználó böngészőjében végzi, ehhez a számítómotor javascriptben van implementálva. Ez a megoldás nem hatékony azokban az esetekben, amikor a felhasználók helyi hálózaton vannak és ugyanazt a dokumentumot nézik – pl. konferenciákon, üzleti megbeszéléseken.

Ebben a projektben egy olyan megoldás prototípusát dolgozom ki, mely nem csupán az adatokat szinkronizálja, hanem a számításokat a felhasználók számítógépein elosztottan hajtja végre. Céljaim között vannak:

* táblázatkezelő függvények párhuzamosíthatóságának vizsgálata,
* egy keretrendszer kidolgozása, mely lehetőséget biztosít új táblázatkezelő függények hozzáadására,
* egy protokoll kidolgozása, mely a kliensek közötti kommunikációt vezérli,
* néhány alapvető függvény implementálása demonstrációs célra,
* egy webes felhasználói felület létrehozása, ahol a felhasználó grafikusan is kezelheti a programot.

Az így létrehozott platform várakozásaim szerint egyszerűbb számításoknál megközelíti a már létező megoldásokat, míg nagyobb számításigénynél meghaladja azokat.

# Az rendszer felépítése

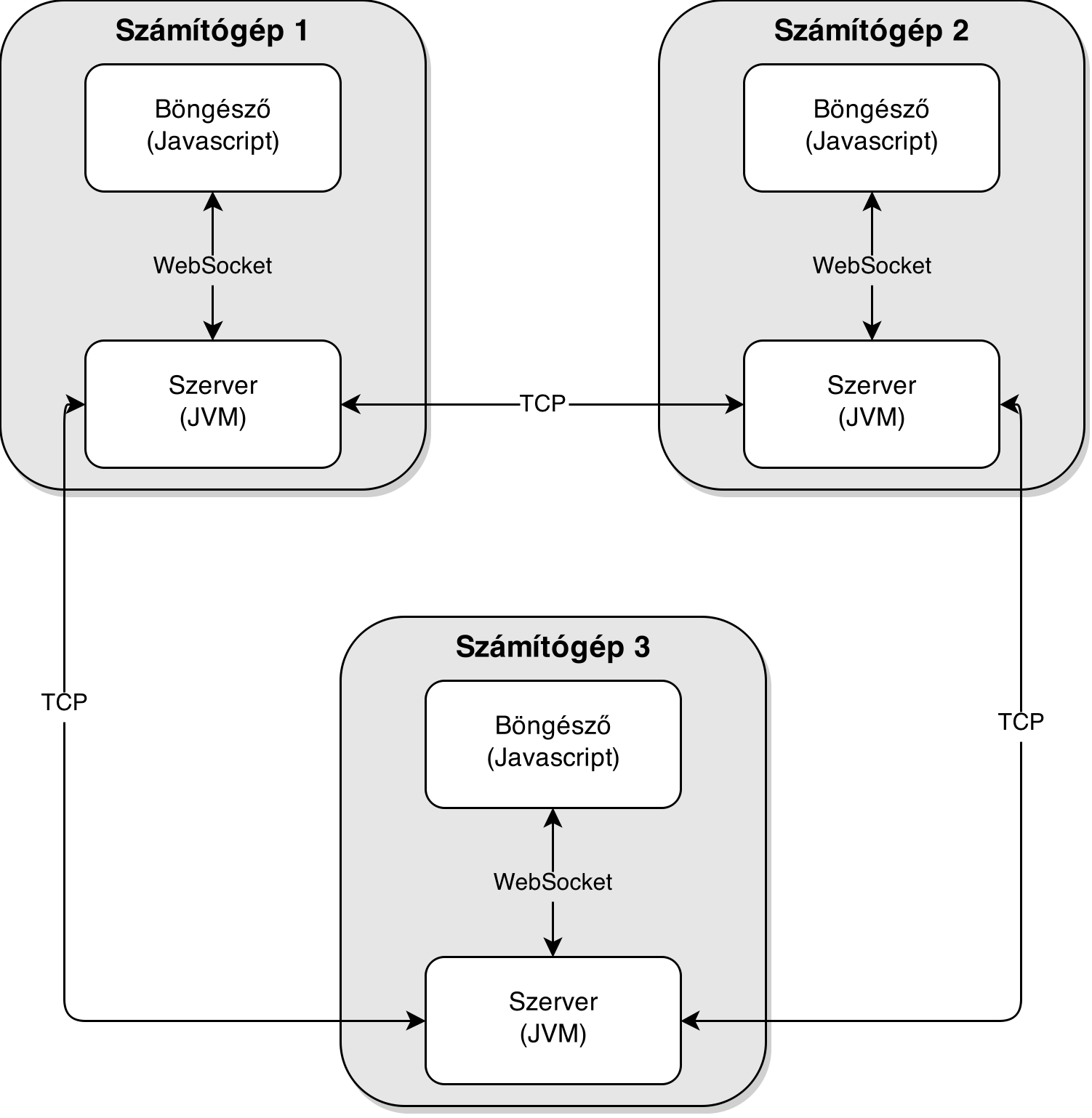
A cél, hogy a számítógépek a helyi hálózaton minél hatékonyabban tudjanak kommunikálni és a számításokat végezni. A felépítésre több koncepció jön szóba. Egyik megoldás lehetne tisztán javascriptben megoldani a számítás elosztását, és a számítások elvégzését. A mai böngészőkbe épített javascript motorok már igen nagy teljesítményre képesek. Ugyanakkor sajnos jelenleg a böngészőben futó javascript kód csak egy szálon futhat, és javascript futása közben a böngésző nem reagál a felhasználói felület változásaira, azaz irreszponzív lesz az alkalmazás. Mivel az alkalmazásban hosszú idejű számításokra tervezek, ez elfogadhatatlan.

Másik megoldás egyfajta kliens-szerver modell kialakítása, amiben a javascript szerepe csupán az adatok megjelenítése, a számítások elvégzését a szerver végzi. Ennek a megoldásnak komoly előnye, hogy a szerver tetszőleges nyelven és környezetben lehet fejlesztve, melyet a feladatnak megfelelően lehet választani. Ennél a megoldásnál a kliens oldal irreszponzivitása sem áll fent.

A kliens-szerver struktúra tehát jónak tűnik, kérdés az, hogy elosztott, peer-to-peer környezetben hogyan legyen a szerver kijelölve. A prototípusban olyan megoldást dolgoztam ki, ahol minden számítógép szerverként is és kliensként is funkcionál, mindegyik böngésző (kliens) az azonos gépen lévő szerverhez kapcsolódik. A szerverek felelősek az egymás felderítéséért, a számítások elosztásáért és elvégézéséért. Kommunikáció egy gépen belül a böngésző és a szerver között, illetve a szerverek között folyik. A böngésző és a szerver közötti kommunikáció WebSocket[[1]](#footnote-1) technológia segítségével történik. A WebSocket protokoll lehetőséget ad TCP feletti full-duplex kapcsolatra böngésző és a szerver között a hagyományos kliens-szerver modellel szemben, melyben a szerver a klienst csak abban az esetben tudja frissíteni, ha a kliens lekérdezi(polling). A WebSocket protokoll 2011-ben lett szabványosítva, a ma elterjedt böngészők új verziói mind támogatják, az Internet Explorerben a 10-es verziótól felfelé érhető el.[[2]](#footnote-2)

A szerverek egymás között TCP-n keresztül kommunikálnak. A lehetséges legnagyobb megbízhatóság és sebesség miatt teljesen hurkolt topológia van a szerverek között. Mivel ezres nagyságrendű a lehetséges nyitott socketek száma, a jelenlegi prototípusban – és feltételezve a jövőbeli felhasználási módokat –, nem okoz gondot a túl nagy mennyiségű konkurrens kapcsolat. A szerverek közötti kommunikációról a későbbiekben részletesen lesz szó.

A felépítést három kapcsolódó kliens esetén az 1. ábra mutatja.



1. ábra

A későbbiekben a kliens szót vegyesen használom mind a hálózaton lévő számítógépek megnevezésére, mind az egy számítógépen belül a böngésző jelölésére, ez remélhetőleg nem okoz félreértést.

# Használt platformok és könyvtárak

A szerver tetszőleges programozási nyelven, tetszőleges könyvtárak használatával lehet írva. Korábbi tapasztalataim alapján a Scala[[3]](#footnote-3) nyelvet választottam. A Scala egy viszonylag új nyelv, az első verzió 2003-ban jelent meg. A Scala egy több programozási paradigmát magába foglaló és támogató nyelv, mind a funkcionális, mind az objektumorientált paradigmának fontos szerepe van a nyelvben. Hasonlóan a Javahoz, szigorúan típusos, így megfelelő összetett, nagy programok írására. A Scala a JVM-re lett tervezve, ami azt jelenti, hogy a Scalaban írt programok a Javaban írt társaikhoz hasonló bájtkódra fordulnak. Emiatt a nyelv teljesen kompatibilis a Javaval, így a Javaban írt kódrészletek (osztályok, interfészek, metódusok) különösebb erőfeszítés nélkül használhatóak Scalaból, illetve a Scalaban írt kódrészletek kényelmeseb használhatóak Javaból – utóbbinál egy-két kitétel adott.

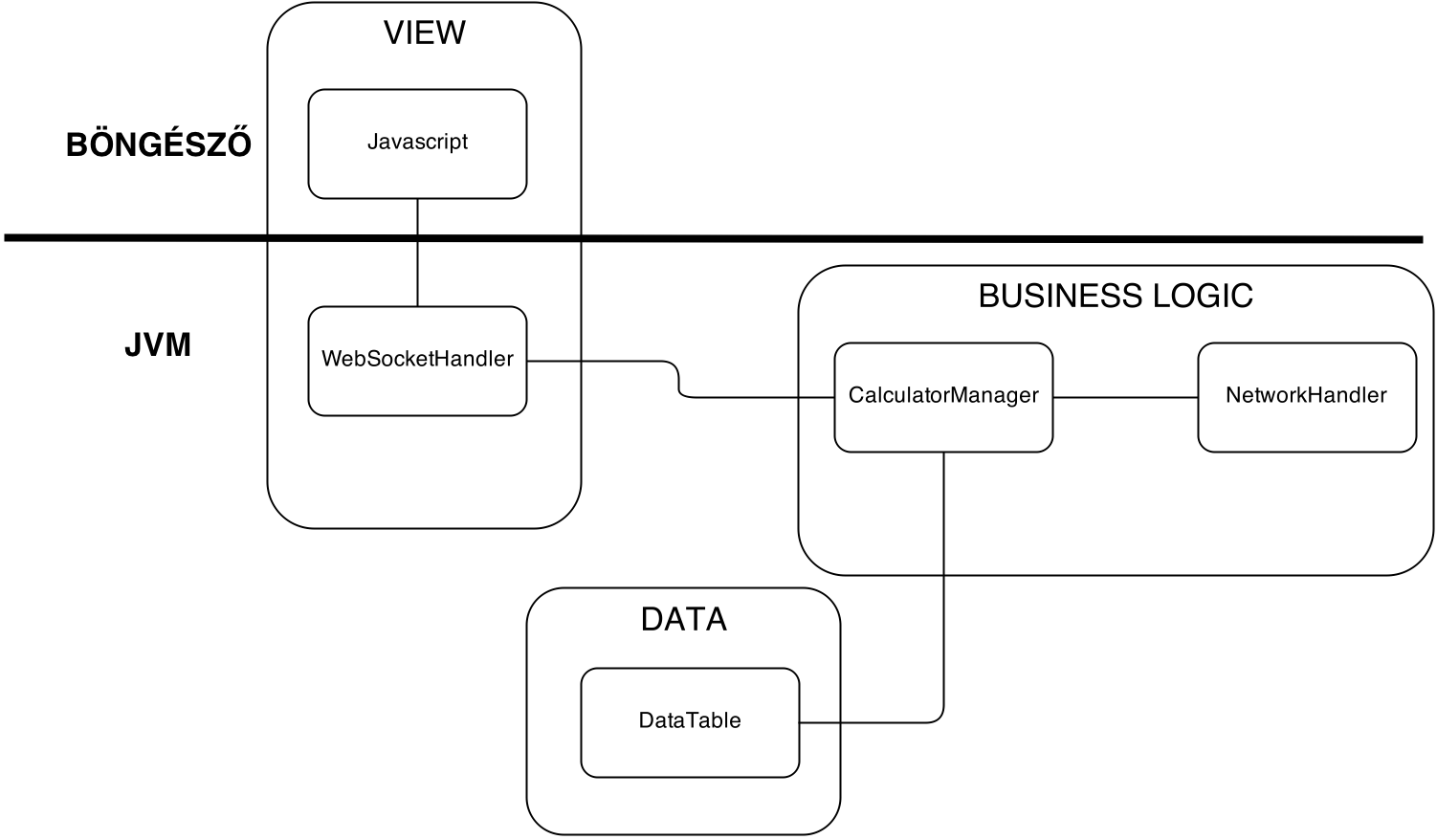
A kliens oldalon a Javascript adott lehetőségként. A kód egyszerűsítése érdekében több nyelv elérhető, melyeket íródott programokat lefordítva[[4]](#footnote-4) Javascriptet kapunk eredményként. Ezek közül a Livescript[[5]](#footnote-5) nyelvet használom a kliens oldal programozására. A kliens oldali formázás a LESS[[6]](#footnote-6) nyelven történik, mely hasonló módon CSS-re fordul.

A használt keretrendszerekről részletesen írok a hozzájuk kapcsolódó fejezetek alatt.

# Az alkalmazás felépítése

A program egy módosított háromrétegű alkalmazásként épül fel, amiben a megjelenítő réteg szét van vágva egy, a JVM-en futó WebSocket kapcsolatért felelős modulra, és a böngészőben futó megjelenítő modulra. Az üzleti logika réteg tartalmazza a számításért felelős modult és a gépek közötti kommunikációért felelős modult. Az adatelérési réteg a prototípusban csupán egy memóriában levő adattábla, mely alapvetően egy hashtáblára épül, a prototípusban perzisztenciával nem foglalkozom.

A modulok áttekintését a 2. ábra mutatja.

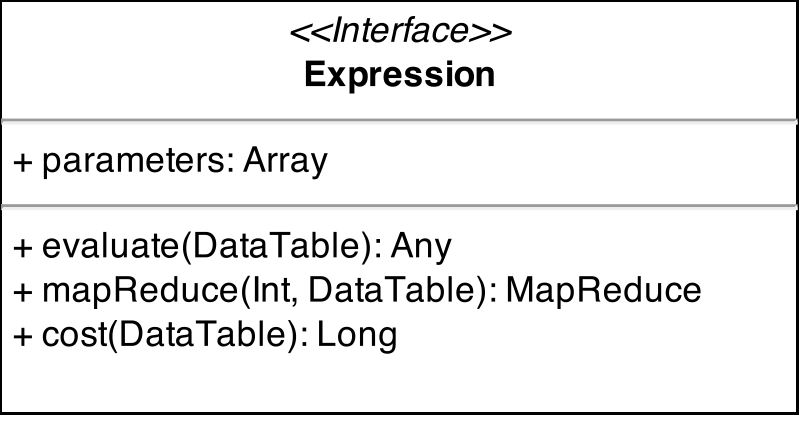


2. ábra

# Adattípusok, függvények

Az elterjedt megoldások szám, szöveg, dátum és logikai típusokat támogatnak. Ebben az alkalmazásban ezt kiterjesztem, és mátrix típus is használható. Egy mátrix többféleképpen megadható. Megadható a mezők tartalma konstansként, vagy létrehozható a mátrix cellák tömbjeként – ebben az esetben a bal felső és jobb alsó cella kijelölésével történik a mátrix megadása. A mátrixok kiválóan használhatóak nagyköltségű műveletek tesztelésére.

A prototípusban három függvényt vizsgálok, az összeadást, a szorzást és egy különleges függvényt, mely a paraméterként megadott számig elszámol, és 0-val tér vissza. Az összeadás és szorzás különböző adattípusokra működik – azaz túl van terhelve.

A függvények általános interfészát a 3. ábra mutatja. A függvény paramétereit a parameters nevű tömb tárolja hivatkozás formájában (azaz a konkrét értékek nincsenek behelyettesítve), az evaluate metódus a DataTable-ből tudja a paraméterek konkrét aktuális értékét kiolvasni. Például, ha a Sum függvénynek két paramétere van, A1 és A2 cellák, akkor a parameters mező tartalma [A1, A2], és a DataTable.get(A1) megadja a konkrét értékét a mezőnek. Az ütemező (lásd később) felelőssége, hogy egy kifejezés kiértékelésekor az összes paraméter már ki legyen értékelve. A függvényekhez minden esetben fenn van tartva egy költség, mely azt határozza meg, hogy a függvény kiszámolásához nagyságrendileg mekkora erőforrás szükséges. Ennek a szám fontos az ütemezésben (lásd később). A mapReduce metódus az első paraméterként megkapott költségű részfeladatokra próbálja bontani a feladatot.

3. ábra

# Függények párhuzamosítása

A projektben kulcsfontosságú szerepe van a táblázatkezelő által támogatott függvények párhuzamosításának. A fent említett függvényeket vizsgálom párhuzamosítás szempontjából.

A párhuzamosítás a MapReduce modell mentén történik úgy, hogy a műveletek felosztják magukat több map műveletre, melyeket párhuzamosan lehet végezni, és egy reduce műveletre, mely a map eredményeket kombinálja.

## CountTo

A rendszer tervezési fázisában azt a műveletet vizsgáltam, amit szinte magától értetődő párhuzamosítani: az adott ideig tartó számolást (CountTo függvénynek hívom). A CountTo(i) függvényt úgy lehet felbontani *n* feladatra, hogy a paramétert *n*-nel osztjuk, azaz létrejön *n* feladat, mely mindegyike *i*/*n*-ig számol.

## Sum

Az összeadás függvény többféleképp paraméterezhető. Támogat egyszerű számösszegzést (a paraméterként kapott számokat összegzi), valamint mátrixok összeadását. Egyforma mátrixok összegzésének eredménye egy mátrix.

Illetve lehetőség van egy mátrix elemeinek összegzésére. Ehhez egy paraméterrel kell hívni, mely paraméter mátrix típusú. Párhuzamosítás szempontjából fontos kiemelni ezt a paraméterezést. A mátrixok létrehozhatóak cellatömbbel is (lásd korábban), ezek remekül párhuzamosíthatók.

Cellatömbök összegzésének párhuzamosítása a következő ábrán látható módon történik:

|  |  |  |
| --- | --- | --- |
| SUM | SUM | SUM |
| SUM | SUM | SUM |

|  |
| --- |
| SUM |

SUM

Azaz résztömbökre kell osztani a mátrixot és az összegzést azon belül elvégezni, majd az összeget összegezni.

# Ütemezés

Egy cella tartalmának megváltoztatása miatt tipikusan több más cellát újra kell számolni. Az ütemező feladata az elvégzendő számítások ütemezése egy gépen belül több szálon, valamint a gépek között a feladatok elosztása. A prototípusban mindig az a gép a felelős a számítások menedzseléséért, amelyik a cella módosítását kezdeményezte. A prototípus készítésekor feltételezem, hogy több cellamódosítás nem esik egybe, azaz egy cella megváltoztatásakor az összes kezdeményezett számítás lefut további cella megváltoztatása előtt. Későbbi stádiumban ettől a feltételezéstől el lehet tekinteni, ez nem változtatja lényegesen a megoldást – ütközéskor csupán az összes gépen, melyen számítás van folyamatban, az ütközést fel kell dolgozni (az változtatott cellákat feljegyezni), és az aktuális számítást újraindítani a megváltozott cellákkal.

Az ütemező felelőssége, hogy a cellák jó sorrendben legyenek újraszámolva, azaz ha az A2 cella paramétere az A1 cella, az A1 cella kiszámítása mindenképp megelőzze az A2 cella kiszámítását. Az ütemező feladat, hogy a feladatokat elossza a résztvevő számítógépek között, és gépen belül különböző szálak között.

Az optimális ütemezés a hálózaton lévő összes számítógép között egyenletesen osztaná el a feladatokat a feladatok költségei alapján, figyelembe véve a függőségeket (amikor egy cella értéke egy másik cella értékétől függ). Még a függőségek figyelembevétele nélkül is a probléma ismert NP-teljes problémákra vezethető vissza (partíció probléma, részhalmaz összeg probléma)[[7]](#footnote-7). Az ütemezéskor fontos, hogy az ütemezési algoritmus ne legyen túl komplex, futásideje elhanyagolható legyen. Ezért az ütemező algoritmus kialakításánál heurisztikák alkalmazásával próbálok minél jobb teljesítményt elérni.

Az ütemezőnek figyelembe kell vennie, hogy külön szálon való végrehajtásnak körülbelül 5-15μs overheadje, míg másik gépen való végrehajtásnak 4-10ms hálózati kommunkációs overheadje van. Ezeket az időket saját számítógépmen, otthoni helyi hálózaton mértem, terhelt hálózat esetén a hálózati késleltetés ezt jóval meghaladhatja. Ezek az idők magukba foglalják azt is, hogy ezekben az esetekben az ütemezőnek az adott, várakozó feladatot külön fel kell jegyeznie.

Azt is fontos figyelembe venni, hogy a külön szálon végrehajtás hálózati késleltetés ideje, illetve a külön szálon való futtatás ideje nem változik arányosan a kiértékelésre elküldött függvények számával, hanem körülbelül konstans. Emiatt érdemes a függvényeket összefogni, és egyben kezelni őket.

Az ütemezőnek tehát négy feladata van:

* a nagy költségű függvényeket fel kell bontania
* a kis költségű függényeket egybe kell csomagolnia
* a függvények kiértékelését helyes sorrendben kell végrehajtania
* a függvények kiértékelési módját meg kell választania

## Nagy költségű függvények felbontása

Erről az előző fejezet szólt, az ütemező az ott bemutatott elemeknek delegálja a feladatot.

## Kis költségű függvények egybecsomagolása

Azon függvények, melyeknek egy szülője van – azaz egy függény kimenetétől függnek, egybecsomagolhatóak a szülőjükkel, így egy komplex kifejezés jön létre, mely két kifejezés egymás utáni kiértékeléséből áll.

## Függvények helyes sorrendben végrehajtása és kiértékelési módja

Az ütemező a végrehajtandó kifejezéseket sorrendbe teszi. A sorrendezéshez topologikus rendezést használ. A sorrend felállítását követően a sorrend szerint hajtja végre az függvényeket. A függvények költsége alapján dönt, hogy az adott függvényt hogyan hajtja végre:

* a saját szálán
* új szálon
* másik számítógépen.

Az utóbbi két esetben várólistára teszi a függvényt, hogy elkerülje azon függények kiértékelését, melyek a listában az aktuális függvény után következnek, és függnek a kimenettől.

# Párhuzamosság

Párhuzamos programozás támogatására rengeteg osztálykönytár érhető el. A projekt során többet teszteltem különböző szempontok szerint.

## Akka[[8]](#footnote-8)

Az Akka keretrendszer az Erlangban bevezetett actor-modell megvalósítása. A megoldás lényegét az adja, hogy a rendszerben saját szállal rendelkező objektumok futnak. Ezek az objektumok belül egyszálúak, és egymás állapotához nem férnek hozzá. A különböző objektumok aszinkron üzenetváltások segítségével tudnak kommunikálni.

## Scala fork-join pool

## Scala Futures and Promises[[9]](#footnote-9)

# Kommunikáció a számítógépek között

A rendszerben magas rugalmasságra kell tervezni, azaz lehetőséget kell teremteni egy-egy kliens bekapcsolódására, illetve távozására – pl. valaki elhagyja az ülést egy konferencián. Tervezésnél azt is figyelembe kell venni, hogy egyszerre nagy mennyiségű kliens érheti el a dokumentumot, sőt akár párhuzamos módosítások is lehetségesek. Ezekben az esetekben az adatok konzisztenciájának megőrzéséhez és a hatékony számításhoz különböző megoldásokat kell alkalmazni. A következőkben ezeket ismertetem. A megoldások egy része csupán megtervezve lesz, az implementálást egy későbbi projektre (pl. szakdolgozat) üzemezem.

A hálózati kapcsolatok kezelésére sok lehetőség adott, több osztálykönyvtár is elérhető a magas szintű hálózat kezelésére.

Egyik lehetőség lehet, hogy a teljes kommunikáció broadcast üzeneteken keresztül történik, azaz minden kliens lát minden üzenetet, és a számára fontosakat dolgozza fel. Ez a megoldás elsőre előnyösnek tűnik, de hosszútávon több gond is van vele sebesség, karbantarhatóság és hibakeresés szempontjából. Ezt a megoldást a magas szintű könytárak sem támogatják.

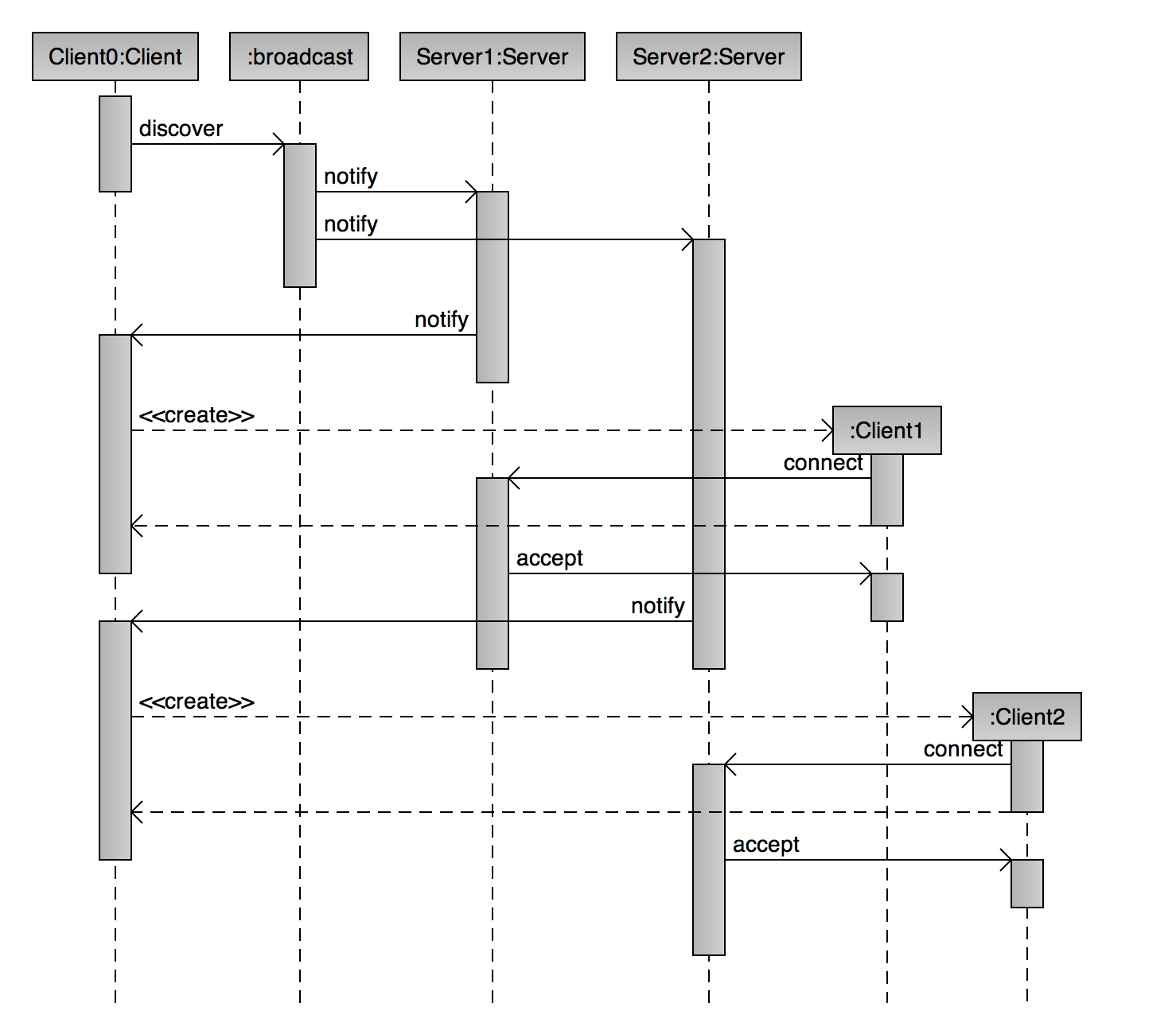
Ennél jobb megoldás, ha a számítógépek között folytonos kapcsolat van. A legegyszerűbb és leggyorsabb megoldás, ha a számítógépek között teljesen hurkolt topológia van, azaz minden kliens minden másik klienssel fenntart egy kapcsolatot. Ennek előnye, hogy bármely két gép minimális késleltetéssel tud kommunikálni igény esetén.

A magas szintű hálózatkezelésre a szabadon elérhető KryoNet osztálykönyvtárat[[10]](#footnote-10) használom, mely absztrachálja a kommunikációs réteget és szerializációt, így objektumokat van lehetőségünk küldeni és fogadni. A KryoNet a TCP és UDP szállítási protokollokat támogatja. Az alkalmazásban kulcsfontosságú a megbízható átvitel, így a gépek TCP-n keresztül fognak kommunikálni.

A KryoNet kliens-szerver modellre lett tervezve, így léteznek szerverek és kliensek – ezek különböznek a korábban említett kliens (böngésző) és szerver (JVM) fogalmaktól, ebben a fejezetben csupán a JVM-ek közötti kommunikációról van szó. A kliensek tudnak kapcsolatot kezdeményezni, a szerverek pedig a kapcsolatkérést el tudják fogadni. Az alkalmazás peer-to-peer jellegéhez oly módon lehet ezt hozzáigazítani, hogy minden JVM-ben fut egy szerver, valamint bizonyos mennyiségű kliens. A következő szabályok szerint történik a kapcsolatok kialakítása.

## Új kliens kapcsolódása

Amikor kapcsolódik valaki a rendszerhez, elindul az ő szervere. Ezután létrejön az első kliens, mely egy broadcast üzenetet küld. A broadcast üzenetre a hálózaton lévő összes szerverről kap választ – a sajátjáról is. A válaszok közül a számítógép kiválogatja azokat az IP-címeket, melyek nem az ő hálózati kártyájához tartoznak, és létrehoz egy-egy klienst mindegyik szerverhez való kapcsolódásra, majd mindegyik klienssel kapcsolódást kezdeményez. Így egy gépen annyi kliens van, amennyi gép a kapcsolódáskor már a hálózaton volt. A kapcsolódási folyamatot a 4. ábra mutatja. A Client0, Client1, Client2 kliensek a frissen kapcsolódó géphez tartoznak, míg a Server1 és Server2 már a kapcsolódott gépeken lévő szerverek.



4. ábra

## Kliens kilépése

A gépek folyamatos kapcsolatot tartanak fent, és másodpercenként többször értesítik az összes többi gépet az állapotukról (pl. aktuális számítási kapacitásukról). Ha egy klienstől adott időn belül nem érkezik ilyen csomag, az adott klienst kilépettnek tekintjük.

## Cella módosítása

# Webes felhasználói felület

# Rövidítésjegyzék

|  |  |
| --- | --- |
| CSS | Cascading Style Sheets |
| HTML | Hypertext Markup Language |
| JVM | Java Virtual Machine |

# Tartalomjegyzék

1. Bevezetés 2

2. Az alkalmazás felépítése 2

3. Használt platformok és könyvtárak 4

4. A program felépítése 5

5. Adattípusok, függvények 6

6. Függények párhuzamosítása 6

6.1 CountTo 7

6.2 Sum 7

7. Ütemezés 8

7.1 Nagy költségű függvények felbontása 9

7.2 Kis költségű függvények egybecsomagolása 9

7.3 Függvények helyes sorrendben végrehajtása és kiértékelési módja 9

8. Párhuzamosság 9

8.1 Akka 10

8.2 Scala fork-join pool 10

8.3 Scala Futures and Promises 10

9. Kommunikáció a számítógépek között 10

9.1 Új kliens kapcsolódása 11

9.2 Kliens kilépése 12

9.3 Cella módosítása 12

10. Webes felhasználói felület 12

10.1 Performance comparison of different multithreading libraries 12

11. Rövidítésjegyzék 13

12. Tartalomjegyzék 13

1. [http://dev.w3.org/html5/websockets](http://dev.w3.org/html5/websockets/) [↑](#footnote-ref-1)
2. <http://caniuse.com/websockets> [↑](#footnote-ref-2)
3. <http://www.scala-lang.org> [↑](#footnote-ref-3)
4. <http://en.wikipedia.org/wiki/Source-to-source_compiler> [↑](#footnote-ref-4)
5. <http://livescript.net> [↑](#footnote-ref-5)
6. <http://lesscss.org> [↑](#footnote-ref-6)
7. <http://www.cs.bme.hu/algel/12elo-2012.pdf> [↑](#footnote-ref-7)
8. <http://akka.io> [↑](#footnote-ref-8)
9. <http://docs.scala-lang.org/overviews/core/futures.html> [↑](#footnote-ref-9)
10. <https://github.com/EsotericSoftware/kryonet> [↑](#footnote-ref-10)