Felhő alapú terhelés elosztás vizsgálata adaptív módszerekkel

Tudományos Diák konferencia dolgozat

2014.

Szerzők:

Jánoky László Viktor	ZE5LCX	jlaci@innosys.hu
Puskás Péter	HM9LM6	puskas.bme@gmail.com

Konzulensek:

Dr. Ekler Péter	Ekler.Peter@aut.bme.hu
Dr. Goldschmidt Balázs	balage@iit.bme.hu

Tartalom

1. Bevezetés ... 3
1.1 A dolgozatról ... 3
1.2 A készítőkről ... 3

2. A feladat és probléma bemutatása ... 4
2.1 Aktualitás ... 4
2.2 A probléma ... 4
2.3 A cél ... 7

3. Források ... 8
3.1 Irodalomi kutatás ... 8
3.2 Internetes források ... 8

4. A rendszer tervezése ... 9
4.1 Tervezési szempontok ... 9
4.2 Követelmények ... 11
4.3 Architekturális döntések ... 11
4.3.1. A rendszer funkciók mentén komponensekre bontása 12
4.3.2. Skálázhatóság és szolgáltatás biztonság javítása 13
4.3.4. A rendszer költséghatékonnyá tétele és felhőhöz illesztése 14
4.4 Technológiai döntések ... 17
4.4.1. Felhő platform választása 17
4.4.2. Nyelv és technológia választás 19
4.4.3. Komponens specifikus döntések 22
4.4.4. Kommunikációs modell 28
4.5 A tervezés összegzése ... 30

5. A rendszer fejlesztése ... 32
5.1 Fejlesztési folyamat ... 32
5.1.1 Web modul 33
5.1.2 System modul 33
5.1.3 Common modul33
5.1.4 Protocols modul 33
5.2 Eszköz választás ... 33

6. A kész rendszer vizsgálata ... 34
6.1 Telepítési környezet ... 34
6.2 Mérési eszközök és konfiguráció ... 34
6.2.1 Szerver oldal 34
6.2.2 Kliens oldal 35
6.2.3 Mérési helyek 38
6.3 Eredmények értékelése ... 38
6.3.1 Válaszidő mérése (RT) 38
6.3.2 Feldolgozási idő mérése (RPT) 38
6.3.3 Utazási idő mérése (RTT) 38
6.3.1 A teszt 39
6.3.2 B teszt 42
6.3.3 C teszt 45
6.3.4 D teszt 48
6.3.5 E teszt 51

7. Levont következtetések, tapasztalatok ... 54
7.1 Kezdeti céloknak megfelelés ... 54
7.2 Korábban hozott döntések értékelése ... 54

8. Függelék ... 54
8.1 Nyers mérési eredmények ... 54

1. Bevezetés

1.1 A dolgozatról

A téma aktualitását a felhő alapú megoldások térnyerése adja, a platform jellegéből adódóan a terhelés elosztása számos kérdést vet fel. Egy ilyen rendszer tervezése során több szempont átértékelődik a korábbi gyakorlatokkal szemben, illetve újak merülnek fel. Legyen szó a költségek minimalizálásáról, biztonságról, környezetvédelemről vagy rendelkezésre állásról, a felhő alkalmazása miatt ezen aspektusokat alaposan meg kell vizsgálni.

A dolgozat célja ezeknek a kérdéseknek a vizsgálata, mérések által alátámasztott következtetések levonása. Végső soron célunk egy olyan terhelés elosztási stratégia megvalósítása és eredményességének igazolása mérések segítségével, amely a felhő erősségeire támaszkodva, a körülményekhez adaptívan alkalmazkodva a lehető legjobb megoldást nyújtja a felmerült problémákra.

A kutatás keretében létrehoztunk egy komplex, felhő alapú általános kérés kiszolgáló architektúrát, mely a terhelést futási időben osztja el adaptív módon. A dolgozat keretein belül bemutatjuk a kidolgozott architektúra elemit, illusztráljuk a komponensek szerepeit és igazoljuk az architektúra működésének helyességét.

A hatékony terhelés elosztás igazolása céljából többféle mérést végeztünk, melyek bizonyítják a kidolgozott megoldás létjogosultságát. Ezen mérések eredményt szintén a dolgozatban elemezzük és értékeljük. A kidolgozott megoldás mindenképp egy általános eszköz nagy méretű, felhő alapú rendszerek tervezéséhez és több gyakorlati alkalmazásban is közvetlenül alkalmazható.

1.2 A készítőkről

A dolgozatot Jánoky László Viktor (felelős szerző) 7. féléves mérnök informatikus hallgató, valamint Puskás Péter 7. féléves mérnök informatikus hallgató írta, Dr. Ekler Péter és Dr. Goldschmidt Balázs iránymutatásával.

2. A feladat és probléma bemutatása

2.1 Aktualitás

A téma aktualitását a felhő alapú szolgáltatások viharos sebességű terjedése adja.

Az ilyen jellegű szolgáltatások új kihívások elé állítják a nagyméretű, komplex rendszerek tervezőit, hisz eddig nem létező problémákra kell megoldást találniuk, egy korábbinál jóval tágabb eszközkészlet segítségével.

Ez a paradigma váltás indokolja meg a témában való komolyabb elmélyedést. Az ezen a téren szerzett tapasztalat később még rendkívül hasznos lehet, hisz a felhő szolgáltatások terjedésével az igény is folyamatosan nő a terület jobb megismerése iránt.

2.2 A probléma

Egy általános szolgáltatást nyújtó rendszer működése során az erőforrásokat felhasználásuk módja szerint három csoportba sorolhatjuk be; aktuálisan a kiszolgálásra fordított (azaz az aktuális terhelés erőforrás igénye), rendelkezésre álló és a rendszer teljes kapacitását jelentő.

2.2.1. Ábra - Egy szolgáltatást nyújtó rendszeren belül az erőforrások megoszlása felhasználás szerint

Ezeket a csoportokat különböző, a működési környezettől erősen függő szempontok szerint jellemezhetjük. Egy tradicionális, dedikált hardveren futó rendszer esetén ez a jellemzés az alábbi módon épül fel:

Az aktuális terhelés erőforrás igénye:
Ezalatt azt az erőforrás mennyiséget értjük amit a rendszer az aktuális terhelés kiszolgálására használ.

- Külső tényezőtől függ (a rendszer terhelésétől).

- Mérete az időben gyorsan változhat, csak becslést tudunk adni egy jövőbeni állapotára .

- A rendszer optimalizálásával csökkenthető a mérete.

Rendelkezésre álló erőforrások:
A rendszer által lefoglalt, de a kiszolgálásban éppen nem használt erőforrások. Ennek a mérete szabja meg, hogy mekkora megemelkedett terhelést tud még a rendszer fennakadás nélkül kiszolgálni. Általánosságban minél nagyobb ez a mennyiség annál kevésbé lesz a rendszer érzékeny a növekvő terhelésre.

- Méretét az üzemeltető szabja meg és a kapacitás erejéig lehet növelni.

- Az üzemeltetése költséges, ezért minél közelebb van az aktuális erőforrás igényhez annál hatékonyabb a rendszer, sajnos a teljesítmény rovására.

Teljes kapacitás:
A rendszer számára elérhető maximális mennyiségű erőforrás. Teljes kihasználtság mellet mérete megegyezik a rendelkezésre álló erőforrásokéval, egyéb esetekben érdemes lehet a költségek csökkentése végett nem kihasználni a teljes kapacitást.

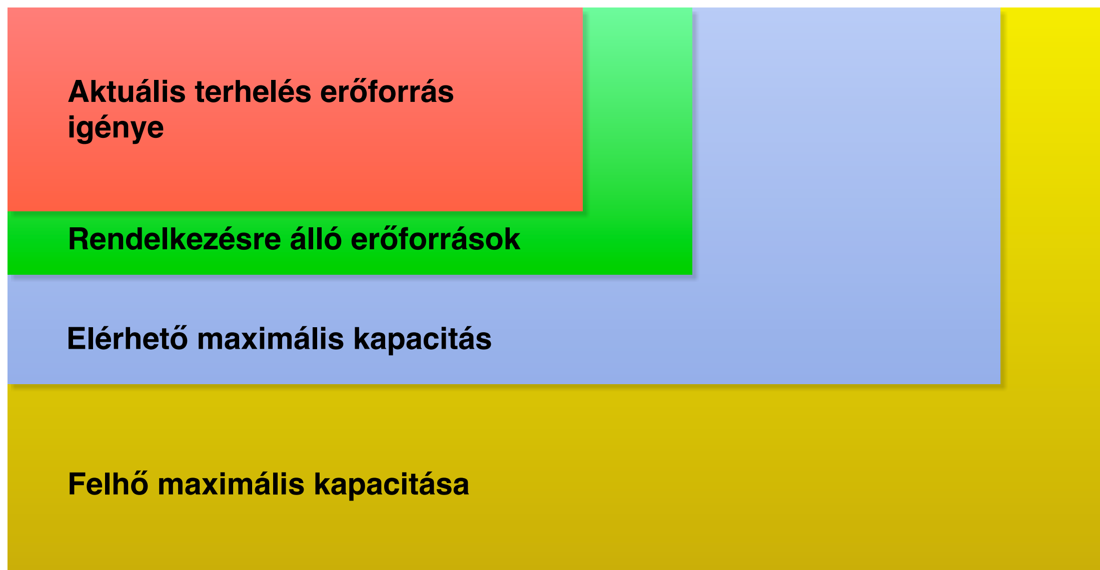
- Kiépítése költséges, ezért jól bevált módszerek vannak az adott követelményeket kielégítő, minimális kapacitás kiszámolására.

- Mérete csak nehezen változtatható, időben költséges.

Ezekből a megállapításokból következik, hogy a tradicionális rendszerekben a terhelés elosztás egy viszonylag statikus erőforrás mennyiség minél hatékonyabb kihasználásáról szól úgy, hogy az még tudja biztosítani a szolgáltatás működésével szemben támasztott követelményeket (rendelkezésre állás, maximális terhelés, adatbiztonság).

Ha a felhő technológiát tekintjük ott ezen megállapítások nem állják már meg maradéktalanul a helyüket.

A technológiai váltás eredményeként eddig nem kivitelezhető, vagy nem praktikus megoldások kivitelezhető lehetőségekké váltak. Egy hagyományos infrastruktúrát használó rendszerhez képest egy direkt a felhőre tervezett rendszer számottevően rugalmasabb tud lenni.



2.2.2. ÁBRA - EGY FELHŐBEN FUTÓ SZOLGÁLTATÁS ERŐFORRÁS FELHASZNÁLÁSÁNAK FELOSZTÁSA

Az aktuális terhelés erőforrás igénye:

Nem változik jelentősen, a felhőben alkalmazott virtualizáció miatti kisebb overhead ugyan jelen van, ám hogy azt a terheléshez számítjuk-e vitatható.

Rendelkezésre álló erőforrások és kapacitás:

Ez a két fogalom a felhő vonzatában szorosan összefügg így fontos tisztázni a különbségeket. Az adott felhőtől függően, egy rendszer rendelkezésére álló erőforrások mennyisége dinamikusan változhat, akár a teljes felhő kapacitása által megszabott határig, azaz kapacitás alatt itt a rendszer által igénylehető maximális mennyiségű erőforrást értjük. Ezzel ellenben a rendelkezésre álló erőforrások alatt az éppen lefoglalt erőforrás mennyiségre gondolunk, így ezen megállapítások függvényében elmondhatjuk:

- A felhőben a maximális kapacitásra (azaz a felhő kapacitására) minden korábban leírt dolog igaz, hisz ez jár valódi fizikai kapacitás növeléssel.
- Az általunk elérhető kapacitás az adott felhő szabályaitól függ, mind a kapacitás növelés költsége, mind a módja.
- A rendszer kapacitásának bővítése általánosságban nem olyan költséges mint tradicionális esetben, hisz nem történik fizikai bővítés, pusztán a felhő kapacitásából kap többet.

Mindennek az a következménye, hogy egy felhő alapú rendszer sokkal tágabb eszközkészlettel rendelkezik a terhelés elosztás és skálázhatóság megoldásának módját illetően mint egy tradicionális rendszer.

Ebből a nagyobb eszközkészletből számunkra leginkább a dinamikusan változtatható kapacitás a legérdekesebb, hisz ez azt jelenti hogy a terhelés elosztásnál az alap probléma változik.

A statikus kapacitás méretezés fontossága háttérbe szorul, hisz nem onnan származik a költségek túlnyomó része, ellenben a mindenkori lefoglalt erőforrás mennyiséget kell optimalizálni a terhelés kiszolgálására.

Ez a követelmény változás megindokolja, hogy alaposabban megvizsgáljuk a terhelés elosztás és skálázódás kérdését.

2.3 A cél

A korábbi megállapítások alapján célunk egy olyan általános rendszert megtervezni, elkészíteni és vizsgálni amely a terhelés elosztás és skálázódás szempontjából direkt a felhőre van optimalizálva.

Ezen célunk megvalósítása során kiemelten fontosnak érezzük az egyes lépések elméleti hátterének kutatását, majd ezek ellenőrzését mérések segítségével.

Nagy méretű, komplex rendszerek tervezése egyáltalán nem új probléma az informatika területén. A rendszer egyedisége abban rejlik, hogy az eddigi tapasztalatokat, best practice-eket, az új futási környezet fényében vizsgálja meg.

Az új platform egyrészt új kihívásokat jelent, hisz pont ezek a bevett megoldások nem alkalmazhatóak itt maradéktalanul, ugyanakkor új lehetőségeket is teremt.

Számos eddig tervezési idejű lépés a felhő segítségével részben, vagy egészben átvihető futásidőbe, ugyanakkor számos eddigi fix vagy egyszeri költség is üzemeltetési, "futás időbeli" költséggé válik.

Célunk egy teljes rendszer elkészítési folyamatának végigkövetése, tapasztalatok szerzése, következtetések levonása és mérések elvégzése. Az így elkészült rendszer egyrészt rendelkezésre fog állni későbbi felhasználásra, valamint a szerzett tapasztalatok alapján ez, vagy egy esetleges másik rendszer fejlesztése is meggyorsítható.

3. Források

Az irodalom kutatás során egyaránt merítettünk elméleti és gyakorlatiasabb forrásokból. A dolgozat során ezekre az itteni jelölésükkel fogunk hivatkozni, például [1] vagy [W2].

3.1 Irodalomi kutatás

[1] Kate Matsudaira – Scalable Web Architecture and Distributed Systems, The Architecture of Open Source Applications vol. II. (ISBN 978-1-1055-7181-7)

[2] Pokharel, M. - Cloud Computing in System Architecture, Computer Network and Multimedia Technology (ISBN 978-1-4244-5272-9)

3.2 Internetes források

[W1] Amazon AWS dokumentáció - http://aws.amazon.com/documentation/

[W2] Spring Framework dokumentáció - http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/

[W3] Spring Boot dokumentáció - http://docs.spring.io/spring-boot/docs/1.1.8.RELEASE/reference/htmlsingle/

[W4] Yong Mook Kim technikai blogja - http://www.mkyong.com/

[W5] ZeroTurnaround / RebellLabs - http://zeroturnaround.com/rebellabs/

[W6] Alexis Lê-Quôc, Mike Fiedler, Carlo Cabanilla - Datadog - The Top 5 AWS EC2 Performance Problems - http://www.datadoghq.com/wp-content/uploads/2013/07/top_5_aws_ec2_performance_problems_ebook.pdf

4. A rendszer tervezése

4.1 Tervezési szempontok

A rendszer tervezése során számtalan szempontot mérlegelni kell, melyek közül több is erősen hat egymásra. Pár általános szempontot az [1]-es forrásra hivatkozva fölsorolunk, majd később a tervezési folyamatot leíró részben elemezzük az egyes döntések eredményét ezen pontok szerint.

Rendelkezésre állás:

Annak a valószínűsége, hogy egy megadott idő intervallum egy adott pillanatában az elvárásoknak megfelelően működik a rendszer.

A rendszer által kiszolgált üzleti folyamatok jellegétől függően a szolgáltatás kiesése akár megengedhetetlen is lehet (például repülés irányító rendszerek), vagy komoly anyagi következményekkel is járhat. Éppen ezért ez az egyik kulcsfontosságú tervezési szempont.

Mivel a rendszer elosztott, ezért külön figyelmet kell fordítani az egyes komponensek meghibásodásának észlelésére és megfelelő beavatkozásra. Ugyanakkor a felhő maga is potenciális hibaforrás lehet, az hogy mennyire kell ezzel számolnunk azon is múlik hogy milyen formában vesszük igénybe. Egyáltalán nem közömbös a felhő szolgáltatás szintje ([1]PaaS, [2]IaaS), illetve hogy privát vagy publikus felhőről van-e szó, egy vagy több felhasználóval.

Mivel a privát felhők rendelkezésre állása nem témája ezen dolgozatnak a továbbiakban úgy tekintjük, hogy a felhő szolgáltatók betartják az adott szolgáltatásra vonatkozó [3]SLA-t, és az esetleges privát felhőktől is hasonló szintű szolgáltatás minőséget várunk el, azonban hogy ezt hogy biztosítjuk, arra nem térünk ki.

Teljesítmény:

Sokféle jellemzővel mérhetjük egy rendszer teljesítményt. Lehet a párhuzamosan kiszolgált kérések számát mérni, lehet egy konkrét kérés kiszolgálásának idejét alapul venni.

Mivel jelen esetben egy általános szolgáltatásról beszélünk ezért legcélszerűbb egy adott időegység alatt kiszolgált kérések számát vizsgálni. Ugyanakkor nem szabad elfeledkezni az egyes kérések teljes kiszolgálási idejének (beérkezéstől a kiszolgálásig eltelt idő) minimumon tartásáról sem, hisz ez az paraméter érzékelhető leginkább felhasználói oldalról.

---

[1] Platform as a Service - Futási környezet mint szolgáltatás

[2] Infrastructure as a Service - Infrastruktúra mint szolgáltatás

[3] Service level agreement - Szolgáltatási szint megállapodás

Megbízhatóság:

Megbízhatóság alatt a rendszer konzisztens és determinisztikus viselkedését értjük. Mind a nyújtott szolgáltatások oldalról, mind a rendszer belső felépítésének oldaláról fontos a megbízhatóság.

Az itt elvárt tulajdonságok kapcsán hasonlóságot figyelhetünk meg a tranzakció kezelésnél megszokott [4]ACID tulajdonságokkal. Nem eredményezheti egyetlen művelet sem a rendszer inkonzisztens állapotba kerülését, a már sikeresen elvégzett műveletek eredménye nem veszhet el.

Az elosztott környezet miatt azonban ezen tulajdonságok biztosítása összetett kihívást jelent, aminek gyakran csak erős kompromisszumokkal lehet megfelelni.

Skálázhatóság:

Szintén egy összetett fogalom, jelen kontextusban a rendszer azon képességét értjük alatta, hogy alkalmazkodjon a különböző mennyiségű kiszolgálandó kérésekhez.

Egy skálázható rendszer további erőforrások hozzáadásával tudja növelni a teljesítményét, optimális esetben az erőforrás és teljesítmény között egyenes arányosság áll fenn, ami tetszőleges hozzáadott erőforrás mennyiségig igaz marad. Természetesen ilyen rendszer nem létezik, de minél jobban közelíti a rendszer ezt a tulajdonságot annál jobbnak tekinthetjük skálázhatóság szempontjából.

Üzemeltethetőség:

Egy rendszer üzemeltethetősége hatással van az összes többi szempontra és az egész rendszer élettartama során fontos szerepet tölt be. Az üzemeltethetőséghez tartozik a karbantarthatóság, módosítások, beállítások elvégzése, egy esetleges hiba felderítése és elhárítása mennyire komplex feladat.

Költségesség:

Másik fontos szempont egy rendszer költsége ahol a költség alatt a teljes élettartam alatti költségeket értjük ([5]TCO). Fontos mérlegelni a különböző életciklus fázisokhoz tartozó költségeket a rendszer előrelátható élettartamának figyelembevételével.

Egy nagyon egyszerű példánál maradva egy adminisztrációs eszköz elkészítése bár lehet hogy költséges a fejlesztési fázisban, de a rendszer működése során többszörösen megtérülhet.

A felhőben való működés is egyedi kérdéseket vet fel, hisz az egyszeri fix infrastruktúra építést költségét cseréljük egy változó, de időtartam függő költségre.

---

[4] Atomicity, Consistency, Isolation, Durability

[5] Total cost of ownership - teljeskörű birtoklási költség

4.2 Követelmények

Az egyik legfontosabb lépés egy szoftver fejlesztése során a követelmények feltárása és rögzítése, hisz ezek alapvetően határozzák meg a felhasználandó technológiákat, módszereket.

Miután tisztáztuk milyen szempontokat kell figyelembe venni egy általános webes rendszer tervezésekor a konkrét esetre vonatkozóan is felsorolhatjuk a követelményeket.

Mint korábban a célkitűzésből kiderült a fejlesztés során egy általános rendszert igyekszünk létrehozni, így a követelmények inkább technikai jellegűek mintsem az üzleti felhasználásra vonatkozóak, lévén azt nem kötöttük egyáltalán ki.

Pontosítva a korábban megfogalmazott „általános szolgáltatást nyújt” kifejezést, a rendszer több felhasználó számára (akár párhuzamosan), állapot mentes webes szolgáltatásokat nyújt. Fontos megjegyezni, hogy csak a szolgáltatások állapotmentességét kötöttük ki, a mögöttes erőforrás amit a felhasználó manipulál a szolgáltatáson keresztül rendelkezhet és nagy valószínűséggel rendelkezik is állapottal.

A rendszernek könnyen bővíthetőnek és modulárisnak kell lennie, hogy az esetleges üzleti alkalmazásokhoz lehessen igazítani. A rendszer működéséért felelős komponenseknek határozottan el kell különülnie az üzleti logikáért felelős komponensektől.

A rendszernek jól skálázhatónak kell lennie a terhelés méretétől függően, a skálázás és terhelés elosztás a felhő platform figyelembe vételével kell hogy történjen.

A felhasználók lehetnek anonimak, ugyanakkor a rendszernek támogatnia kell a felhasználók azonosítását, engedélyeik és szerepköreik kezelését, a felhasználók személyes adatait megfelelően védeni kell.

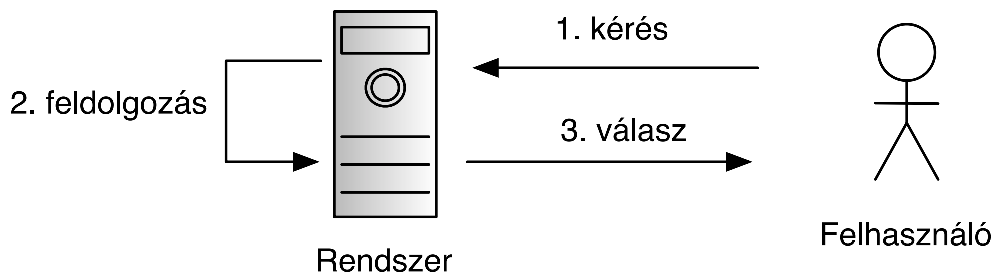
4.3 Architekturális döntések

A tervezési szempontok és követelmények mérlegelése után, egy azokat minél inkább kielégítő architektúra megtervezése a cél. Ez egy iteratív folyamat, amelynek minden állomása során a kapott eredményt megvizsgáljuk és a levont következtetése függvényében haladunk tovább.

Egy általános három rétegű architektúrából indultunk kis és azt az alábbi lépésekkel építettük fel.

4.3.1. A rendszer funkciók mentén komponensekre bontása

Egy általános kérés kiszolgálásának modellje az alábbi lépésekből áll:



4.3.1.1. Ábra - Általános kérés kiszolgáló rendszer működése

1. Kérés fogadása – a rendszer fogadja a felhasználó kérését

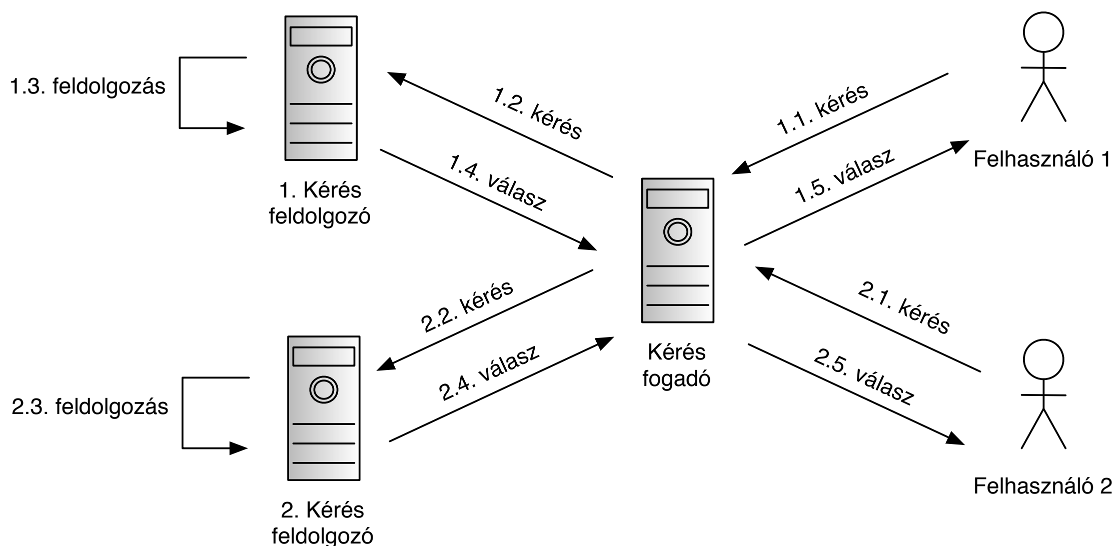
2. A kéréshez feldolgozás – a kéréshez kapcsoló műveletek elvégzése

3. Válasz küldése – az eredmény visszaküldése a felhasználóhoz

Ha abból indulunk ki hogy a rendszerünket a felhasználók weben keresztül fogják használni, ez nagy valószínűséggel HTTP protokoll felett fog történni. A HTTP mint kérés-válasz alapú protokoll nem engedi hogy a szerver kezdeményezze a kommunikációt. Ez azt eredményezi, hogy addig nem zárhatjuk le szerver oldalról a kommunikációt amíg a kérés feldolgozása be nem fejeződött, még ha az időben költséges is, hisz máskülönben nem lenne módunk kiszolgálni a klienst. Ez viszont azt jelenti, hogy a kérés kiszolgálásának állapotát fent kell tartani a szerver oldalán egészen amíg a válasz el nem készül. Könnyen belátható hogy ez sok felhasználó és sok függőben lévő kérés esetén igen költséges lehet.

> Kis kitérő, ez a probléma egyáltalán nem új keletű, számos megoldás létezik rá. Példának tekinthetjük Java Enterprise Edition 6 óta a Servlet 3.0 szabványt. Itt az aszinkron feldolgozással igyekeznek orvosolni a hosszan tartó kérések által okozott teljesítmény csökkenést, azaz amíg egy kérés elvégzése folyamatban van az nem blokkolja a kérésfeldolgozó erőforrásokat. Ennek ellenére itt ugyanúgy fent kell tartani az egyes kérések állapotát.
>
> Másik megoldás lehet a WebSocket protokoll alkalmazása ami valódi full-duplex kommunikációt valósít meg. Sajnos ehhez külön támogatás szükséges mind szerver mind kliens oldalon és ez a protokoll nem olyan

Ha külön vesszük a konkrét kérés feldolgozását a kérések fogadásától és válasz küldésétől akkor erőforrásokat spórolhatunk azon a komponensen ami a kommunikációt végzi. Az így felszabaduló kapacitást fordíthatjuk akár több felhasználó egyidejű kiszolgálására. Az alábbi modell ezt a folyamatot mutatja be.



4.3.1.2. ÁBRA - TÖBB KÉRÉS FELDOLGOZÓS RENDSZER MŰKÖDÉSE

Ahhoz hogy ezt megtehessük a kérések szempontjából tranzisztensnek kell lennie, hogy melyik feldolgozón futnak. Ezt a legkönnyebben úgy érhetjük el, ha a kérések nem rendelkeznek a feldolgozón tárolt állapottal. Egy teljesen állapotmentes rendszer nem tud minden üzleti igénynek megfelelni ezért valahogy mégis kezelni kell az állapotokat, erre több megoldás is létezik:

- A háromrétegű architektúra adatrétegében tároljuk az állapotot, ám ezzel csak delegáltuk a problémát, nem megoldottuk. Ezen a szinten sokkal formálisabban kezelhető a probléma és már számos bizonyított módszer létezik a leküzdésere.

- Maga a kérés tárolja az aktuális állapotot.

4.3.2. Skálázhatóság és szolgáltatás biztonság javítása

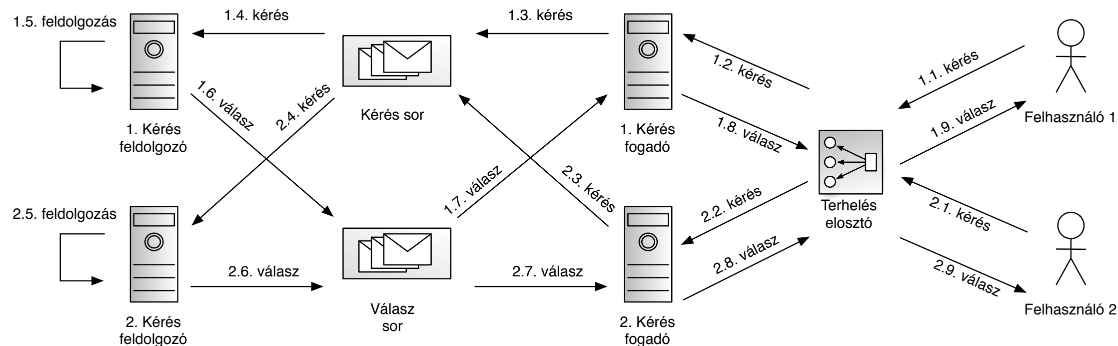
Ha leküzdöttük a feldolgozó függetlenség problémáját további két másik megoldandó kérdéssel találjuk szemben magunkat, nevezetesen a skálázhatóság és rendelkezésre állás problémájával. Az ezen modell szerinti rendszer a kérés feldolgozók számának növelésével már horizontálisan is skálázható, ám a kérés fogadó csak vertikálisan tud skálázódni, hisz csak egy darab van belőle.

A skálázhatóságon túl a rendelkezésre állás szempontjából sem előnyős hogy a kérés feldolgozó egyedi hibapont (SPOF), annak kiesésével az egész szolgáltatás elérhetetlené válik.

Erre a két kérdésre a legkézenfekvőbb válasz a kérés fogadó többszörözése lenne, ám ez szintén felvet újabb kérdéseket. Ha több kérésfogadó és több kérés feldolgozó van párhuzamosan a rendszerben hogyan biztosítjuk, hogy a kérésfogadók ismerjék a kérés feldolgozókat? Mi szerint döntsék el hogy melyik kérésfeldolgozóhoz továbbítsák a kéréseket? A felhasználók melyik kérés fogadóhoz tudnak csatlakozni?

Ezen kérdésekre keressük a megoldást, első lépésként a kérés fogadók ne közvetlen kapcsolatban álljanak a kérésfeldolgozóval, hanem iktassunk be egy-egy üzenet sort a két komponens közé, amikben a kéréseket és elkészült válaszokat tároljuk. Ezzel megoldjuk a kérés fogadók és feldolgozók között több-több kapcsolat kérdését. hisz a két fajta szereplőnek csak a közös üzenet sorokat kell ismernie.

Ha több üzenet fogadó is van, akkor a felhasználókat valamilyen szempont alapján be kell osztani az egyes feldolgozókhoz, ez történhet véletlenszerű módon (pl. DNS round-robin), bízva a terhelések statisztikailag egyenletes eloszlásában vagy valamely kifinomultabb terhelés elosztóval (Haproxy, Amazon Elastic Load Balancer). Eddig így néz ki a modellünk:



4.3.2.1. Ábra - Lecsatolt kérés fogadó/feldolgozókkal tervezett rendszer működése

A rendszer ezen felállása szerint már a kérés feldolgozók és fogadók száma is tetszőlegesen növelhető és léteznek elosztott üzenet sorok is (RabbitMQ, ActiveMQ, Amazon SQS) , tehát elméletben a rendszer már jól skálázható.

A skálázhatóságon túl a rendelkezésre állás terén is előrelépést jelent ez az architektúra. Javul a hibatűrés és ezáltal a rendelkezésre állás hisz több kiszolgáló is futhat egyidejűleg, és egy-egy példány kiesése pedig még nem jelenti a rendszer teljes üzemképtelenségét.

A másik szempont ahol javulást érünk el az a megbízhatóság, hisz azáltal hogy a kiszolgáló belső állapotából kivezettük a kéréseket és válaszokat az üzenet sorokba, egy esetleges rendszerhiba esetén is később helyreállíthatóak maradnak az üzenetek, amennyiben a hiba a sorok működését (és adat megőrzését) nem érinti.

4.3.4. A rendszer költséghatékonnyá tétele és felhőhöz illesztése

Az előző lépésekkel elkészült architektúra már jól skálázható, egy ilyen alapú rendszert már tudnánk méretezni a kapacitás tervezés tradicionális eszközeivel ha ismernénk a paramétereit.

Mivel a kapacitás tervezéshez ismerni kell a konkrét felhasználási módot, a folyamatokat és azok erőforrás igényét ezért az most nem jöhet szóba, de mit tehetünk ha mégis általános esetben is biztosítani szeretnénk hogy a rendszer megfelelően ki tudja szolgálni megnövekedett terhelés?

A válasz az adaptív terhelés elosztásban és skálázásban rejlik. Lényegében azáltal hogy a felhő alkalmazása miatt realitássá vált a kapacitás menet közbeni változtatása, ez a feladat tervezés időből átkerült futás időbe.

Hogy az adaptív terhelés elosztás és skálázódás egyáltalán szóba jöhessen a rendszernek meg kell felelnie pár kritériumnak:

1. Jól skálázhatónak kell lennie, azaz a teljesítmény és erőforrások között megközelítőleg egyenes arányosságnak kell fennállnia.

2. A rendszernek rendelkeznie kell az aktuális terhelésről szóló információkkal, hogy ez alapján be tudjon avatkozni.

3. Léteznie kell valamilyen szabályrendszernek ami leírja, hogy milyen típusú terhelés esetén milyen típusú új erőforrást kell a rendszerbe vinni.

A skálázhatóságot már korábban megvizsgáltuk, a második pont teljesítéséhez pedig számtalan eszköz létezik, sőt a felhő platform maga is nyújthat ilyen jellegű szolgáltatásokat.

A harmadik pont erős összefüggésben van az aktuális terhelés mérésével, hisz annak alapján lehet képe a szabályozandó rendszerről. Érdemes megvizsgálni a milyen paraméterek mérése reprezentálja legjobban a terhelést, milyen gyakorisággal kell ezeket a méréseket elvégezni és milyen érzékenynek kell lennie a szabályzásnak.

Ez a problémakör tervezés időben nehezen eldönthető, hisz ezek a paraméterek nagyban függenek az aktuális implementációtól, felhasznált technológiáktól, futási környezettől. Nem tudjuk például pontosan megmondani, hogy mennyi ideig fog tartani egy új erőforrás felvétele a rendszerbe, hisz az a teljes felhő állapotától is függ, amit ekkor még nem ismerhetünk.

A legelőnyösebb megoldás talán egy tanuló algoritmus implementálása lenne, ami megfigyelve a rendszer működését az adott pillanatban és a múltbeli adatokra támaszkodva egyre optimálisabb megoldást tudna találni a pillanatnyilag szükséges kapacitás mértékére. Sajnos egy ilyen algoritmus tervezése túlmutat jelen dolgozatunk témáján, ám ha a tervezés során figyelembe vesszük ezt a lehetőséget mint jövőbeni cél nagyban megkönnyíthetjük a későbbi implementálását.

A rendszer aktuális terhelésének mérésén kívűl az erőforrások kezelése is megoldandó kérés, nem elég döntést hozni új erőforrás felvételéről, azt valóban létre is kell hozni és a rendszerbe integrálni. Ilyen erőforrás lehet például egy adatbázis, de maguk a kérés feldolgozók és fogadók is.

Ezeket a feladatokat vagy elosztjuk az egyes komponensek között - bár ez rengeteg megoldandó problémát vet fel, mint például egy konzisztens modell fenntartása az aktuális rendszerállapotról, vagy egy központi komponensbe szervezzük.

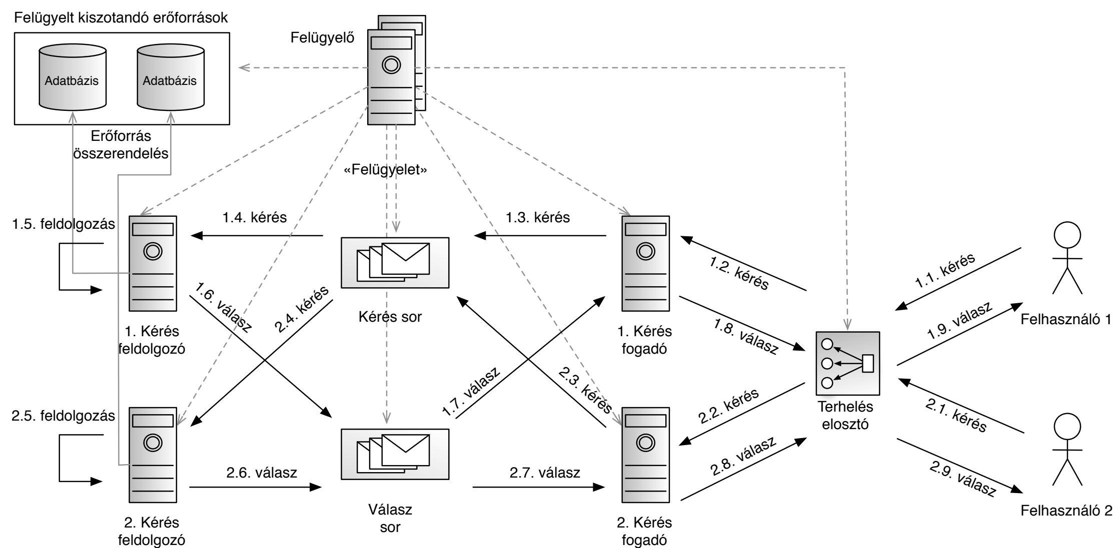
A központi komponens bevezetésével ugyanakkor ismételten egyedi hibapontot vittünk be a rendszerbe, amit késöb kezelnünk kell (replikációval például). Ez a központi elem (nevezzük innentől felügyelőnek) lesz felelős a rendszer állapotának felügyeletéért, az erőforrások kiosztásáért és a kapacitás kezeléséért.

Mint egyszeres elem kérdés a felügyelő skálázhatósága. Mivel a rá eső terhelés nem a rendszer terhelésétől, hanem a méretétől függ, ezért sokkal jobban lehet vele számolni, meg lehet adni felső határokat, összességében jól lehet méretezni.

Igazán jó megoldást természetesen az jelentene ha nem csak horizontálisan, de vertikálisan is skálázható lenne a felügyelő. Ezt elérhetjük a rendszer particionálásával, azaz egy felügyelő alá maximálisan megadott számú felügyelt komponens tartozna, és csak azok felett rendelkezne. A teljes rendszert érintő döntéseket egy magasabb szinten lévő felügyelő hozhatná meg, aki az alárendelt felügyelőket kezelné. Ez a hierarchia igény szerint tovább építhető lenne, ám a felügyelő/felügyelt arány folyamatosan romlana.

Számításaink alapján, nem az egy felügyelő horizontális skálázódása lesz a szűk keresztmetszet, mindenesetre az egyes komponensek tervezése során igyekezni fogunk figyelembe venni a későbbi, esetleges felügyelő particionált működést. Ezáltal ha a mérési eredmények megcáfolják az eredeti elképzelésünket viszonylag könnyen fogunk tudni igazodni az új helyzethez.

Az eddigi modellünk kiegészítve a felügyelővel és az általa felügyelt kiosztandó erőforrásokkal:



4.3.4.1. ÁBRA - KÖZPONTI FELÜGYELŐ ÁLTAL KEZELT RENDSZER MŰKÖDÉSE

4.4 Technológiai döntések

Miután a rendszer általános architektúráját megterveztük a továbblépéshez azonosítani kell a felhasználni kívánt technológiákat, hogy azok igényei és erősségei szerint testre szabjuk az eddigi tervet.

Döntéseink során igyekeztünk minél szélesebb körben támogatott technológiákat választani a későbbi felhasználás megkönnyítése érdekében.

Fontos cél ugyanakkor, hogy a választott technológiától függetlenül általánosíthassuk eredményeinket és tapasztalatainkat. Ezt úgy tudjuk a legjobban elérni, ha tisztában vagyunk az alkalmazott technológiák egyediségeivel, azok hatásait ki tudjuk szűrni a végeredményből. Ehhez elengedhetetlen az egyes megoldások behatóbb tanulmányozása.

4.4.1. Felhő platform választása

Korábban már megállapítottuk hogy számos tervezési aspektusnál igencsak fontos szerepet játszanak a választott felhő platform jellegzetességei, ezért adja magát a döntés hogy először ezt válasszuk ki. Szerencsére az éppen aktuális trendek miatt bőven van választási lehetőségünk.

Amazon AWS:

Az egyik első felhő szolgáltatás (2006-ban indult), sok funkciót kínál ami beleilleszkedik a korábban elkészített tervünkbe. Ilyenek többek között az elosztott üzenetsor szolgáltatás (SQS), terhelés elosztás (Elastic load balancing), valamint a kérésre indítható virtuális gépek (EC2) vagy alkalmazás példányok (Elastic Beanstalk).

A felhő jellege több felhasználós publikus felhő, de van lehetőség virtuális privát felhők üzemeltetésére is (VPC szolgáltatás).

A számos nyújtott szolgáltatás között vannak IaaS (EC2, VPC), PaaS (Elastic beanstalk) és SaaS (SQS, S3) szolgáltatások is, így könnyen megtalálható az egyensúly a kontroll és üzemeltetési komplexitás között.

Előnyök:

- Bizonyított platform
- Széles választék egymással jól együttműködő szolgáltatásokból és több platformra SDK a használatukhoz
- Jól dokumentált

Hátrányok:

- Viszonylag zárt, későbbi esetleges platform váltás nehézkes (bár vannak open-source, AWS interfészt imitáló alternatívák, pl. Project Eucalyptus)
- A PaaS szolgáltatások használata nehezítheti az üzemeltetést

Digital Ocean:

Viszonylag új cég a piacon (2011), ugyanakkor rendkívül gyorsan növekszik, 2014 júliusában már az 5. legnagyobb hosting provider. Használata viszonylag egyszerű, viszont csak IaaS szolgáltatásokat nyújt.

Előnyök:

- Olcsó
- Egyszerű

Hátrányok:

- Csak IaaS

Google App Engine:

A Google PaaS felhő szolgáltatása, kifejezetten nagy megbízhatóságú, skálázható webalkalmazások számára lett kitalálva. Számos szolgáltatást nyújt ami számunkra is hasznos lehet-

Ugyanakkor mivel PaaS, ráadásul az adattárolás tekintetében kifejezetten egyedi megoldásokkal működik (GQL, olyan SQL szerű nyelv amiben nincs join) kérdéses lehet az elkészült rendszer hordozhatósága.

Előnyök:

- Bizonyított platform
- Hasznos szolgáltatások
- Jól dokumentált és támogatott

Hátrányok:

- Zárt és egyedi, platform váltás nehézkes (itt is léteznek open-source alternatívák)

Redhat Openshift:

A Redhat által biztosított PaaS és SaaS felhő szolgáltatás, a fókusz itt is az automatikus, platform által biztosított skálázáson van. A Google App Engine-nel ellentétben az Openshift nem egyedi, zárt platformot biztosít hanem az iparban elterjedt megoldásokból építkezik, mint például a JBoss java alkalmazás szerver, vagy MySQL adatbázis szerver.

A Redhat által nyújtott publikus felhő szolgáltatásokon kívűl maga a platform is elérhető, hisz az alapját képző szoftver nyílt forráskódú (OpenShift Origin). Ezáltal egy esetleges későbbi migráció privát felhőbe viszonylag könnyen kivitelezhető.

Előnyök:

- Elterjedt szolgáltatások használata, nyílt platform
- Egyszerű fejlesztési modell

Hátrányok:

- Túlságosan leegyszerűsített szolgáltatások, kevés dolgot konfigurálhat a fejlesztő

Döntés:

Mindezeket a szempontokat mérlegelve az Amazon AWS mellet döntöttünk, hisz egyrészt itt kaptuk meg a legnagyobb szabadságot, hogy a különböző konfigurációkat kipróbáljuk és kísérletezzünk az egyes megoldásokkal.

Másfelől az árazás is itt a leginkább kedvező számunkra, hisz a használat arányában fizetünk, nem szintenként és nem havi díj alapon.

Másik fontos szempont volt még, hogy a regisztráció után személyes megkeresést kaptunk az Amazon európai oktatásért felelős részlegének egy munkatársától, aki miután útbaigazított minket, felvett az Amazon Activate nevű programba, ahol hasznos információkat és támogatást kaptunk az AWS használatát illetően.

4.4.2. Nyelv és technológia választás

A nyelv kiválasztásakor többek között a hordozhatóságot és támogatottságot mérlegeltük, hisz ha egy később felhasználható, általános rendszert akarunk fejleszteni ez a két dolog elengedhetetlen.

Másik fontos szempont volt a nyelvel való ismertségünk, ezek alapján választottuk a Java-t a rendszer alapját felépítő nyelvként, mégpedig az 1.7-es kiadását.

A 1.8-as verzió lassú piaci adoptációja miatt döntöttünk az 1.7-es mellett, bár így sajnos nem tudtuk alkalmazni az új kiadás kényelmi funkciót, mint a lambda függvények bevezetése.

Miután a nyelvet meghatároztuk a felhasználandó komponenseket, könyvtárakat, keretrendszereket kellett kellet mérlegelnünk. Itt az alábbi lehetőségeket vizsgáltuk meg.

Java Enterprise Edition:

Széles körben elterjedt Java alapú szerveroldali platform, tipikus feladatokra szabványok gyűjteménye. Fő erőssége abban rejlik, hogy a legtöbb általánosan felmerülő problémára létezik általa nyújtott megoldás.

Jól dokumentált, ugyanakkor a verziók közötti fejlődés miatt ügyelni kell, hogy az elavult forrásokra. Az újabb verziói előre mutatóak, ugyanakkor a fejlesztési folyamat viszonylag lassú, és az új változatok iparban történő elterjedése nagyon lassú, emiatt két verzió között viszonylag nagy változások történnek.

A platform csak a szabványokat definiálja, azok implementálása gyáró függő, tehát elméletileg az elkészült kód hordozható. Sajnos a tapasztalat nem ezt mutatja, gyakran még az adott gyártó különböző verziójú termékei között sem lehet a kódot egy az egyben felhasználni.

Előnyei:

- Elterjedt, jól dokumentált
- Nagyon sok kész szolgáltatás

Hátrányai:

- “Nehézsúlyú”
- Gyártók közötti migráció nehézkes
- A hosszútávú továbbfejlesztés nehéz

Spring Framework:

Történelmileg a Spring keretrendszer volt a korábbi Java Enterprise Edition verziók alternatívája, fő előnyei abban rejlettek hogy a keretrendszer könnyebb, kevésbé “betolakodó”.

A JEE 6 megjelenésével számos eddigi előnye eltűnt a Springnek, ám népszerűsége még mindig jelentős. Nem csoda, hisz számos jövőbemutató, és népszerű projekt kapcsolódik ehhez a keretrendszerhez és ezek száma egyre csak gyarapodik.

A keretrendszer alapvető filozófiája más mint a JEE esetében, az egyik fő szempont a modularitás, a fejlesztéshez nem szükséges a teljes funkció portfóliót használnunk, így a külső függőségek száma viszonylag alacsony maradhat.

A másik letagadhatatlan előnye a Springnek a JEE-vel szemben a homogenitás, hisz míg előbbit a [6]JCP keretében számtalan entitás szerkeszti, addig Spring - bár nyílt forráskódú - a Pivotal Software gondozásában áll.

Előnyei:

- “Pehelysúlyú”
- Elterjedt, jól dokumentált
- Sok támogató projekt
- Nem igényel külön alkalmazás szervert
- Egységes

Hátrányai:

- Pár közösségi projekt minősége változó lehet
- Sok régi dokumentáció a verziók gyors váltakozása eredményeként

---

[6] Java community process

Döntés:

A Spring keretrendszer mellet döntöttünk, hogy minél általánosabb lehessen a rendszer, hisz így nem kötöttük a használatát egyetlen gyártó alkalmazásszerveréhez sem.

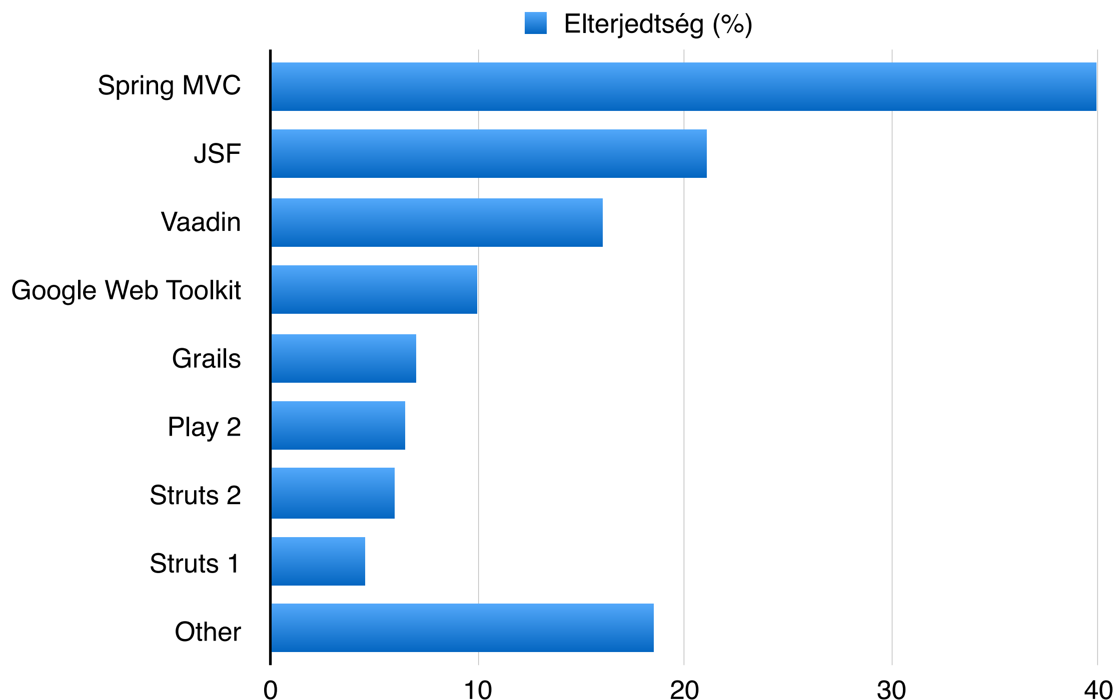
Továbbá számos Spring komponens is jól beleilleszkedik az eddig megtervezett rendszerünkbe, például a natív Spring Remoting, ami HTTP protokoll felett biztosít távoli függvény hívási és sorosítási funkciókat.

Másik fontos döntési szempont volt, hogy egy-egy új szerver példány elindítása minél könnyebb és gyorsabb legyen. Ehhez a Spring Boot projekt nyújt támogatást, segítségével önállóan futtatható csomagokba szervezhetjük az alkalmazásainkat, így csak egy Java 1.7 futátására képes környezetet kell biztosítanunk az egyes példányoknak.

Másik fontos előnye a Spring Boot-nak az Acutator nevű modul, ami a rendszer állapotának monitorozásához nyújt támogatást. Ez mint korábban már megtárgyaltuk elengedhetetlen az adaptív terhelés elosztáshoz és skálázódáshoz.

Mindezeken túl a Spring webalkalmazások fejlesztéséhez is rendelkezik megoldásokkal, a webes keretrendszer (Spring MVC) kiegészíthető több, például biztonsági modullal (Spring Security). Ezek segítségével elkészíthetünk egy rendszer állapot monitorozó és adminisztrációs felületet nyújtó webalkalmazást.

A RebelLabs [W5] 2014-es jelentése szerint a Spring MVC a legelterjedtebb Java alapú webes keretrendszer, így nem meglepő hogy számos segédanyag létezik hozzá az aktív fejlesztői közösség mellet.



4.4.2.1 JAVA ALAPÚ WEBES KERETRENDSZEREK ELTERJEDTSÉGE (2014) - FORRÁS: REBELLABS [W5]

Összességében a Spring minden felmerülő igényünkre rendelkezik megoldással és Spring alapú technológiák felhasználásával később a rendszer könnyedén kiegészíthető további komponensekkel, a technológiai homogenitást fenntartva.

4.4.3. Komponens specifikus döntések

Kérés fogadó (Message server):

A bejövő kéréseket fogadja és beteszi a kérés sorba, majd a kész válaszokat a válasz sorból visszajuttatja a kliensekhez, ennek részleteit a Kommunikációs modell (4.4.4) fejezetben fejtjük ki.

A kérés fogadók számának növelésével tudjuk az egyidejűleg kiszolgálható kliensek számát növelni, ám amint a bejövő kérések feldolgozási ideje meghaladja a kérés feldolgozók átbocsátóképességét a kiszolgálás egyre lassulni fog, a kérés sor pedig nőni mindaddig amíg az arány helyre nem áll.

Ezt a problémát csökkentendő bevezethetjük az aszinkron kiszolgálást, azaz a kliens nem rögtön a választ kapja meg a kérése eredményeként, hanem egy nyugtázást, hogy a rendszer fogadta azt. Ezáltal plusz információval ruházzuk fel a klienst, aki így tudja hogy elérte-e a rendszert csak még várnia kell a válaszra vagy az egyáltalán nem elérhető.

Ezt a modellt szintén a következő fejezetben részletezzük, hisz még számos technikai döntés fogja befolyásolni. Összességében annyit elmondhatunk, hogy az ily módon történő kommunikációnak az az előnye, hogy pusztán a kérés fogadók számának növelésével tudjuk a rendszer megbízhatóságát növelni, ha a teljesítményét nem is feltétlenül.

Technikai jellemzői:

- Spring MVC alapú, Spring Boot támogatással önálló csomagban
- A kliensekkel HTTP/HTTPS protokoll felett REST API-n keresztül kommunikál
- A felügyelővel a Spring HTTP feletti remotingját használva kommunikál
- A kérés és válasz sorral szintén HTTP és REST alapon kommunikál, azonban ezt elfedi előlünk az Amazon Java SDK által biztosított kliens.

Kérés feldolgozó (Worker server):

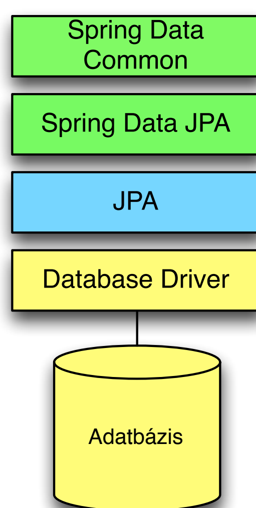
A kérés sorból veszi ki a kéréseket, majd feldolgozva azokat a választ a válasz sorba helyezi.

Mivel egyelőre általános, azaz specifikus üzleti logika nélküli rendszerről beszélünk, ezért nem tudjuk hogy milyen erőforrásokra lesz szüksége a kérés feldolgozónak a valódi feladatok ellátásához. Feltételezhetjük azonban, hogy adatbázis kapcsolatra mindenképpen szüksége lesz, ezeket az erőforrásokat a felügyelő fogja kiosztani a szerverek között.

A kérés feldolgozók számának növelésével (horizontális skálázással), javíthatunk a rendszer átbocsátó képességén, hisz ezek a komponensek felelősek a valódi kérések végrehajtásáért, a többi igazából csak kiszolgáló szerepet tölt be.

Technikai jellemzői:

- Spring Boot alapú
- A felügyelővel a Spring HTTP alapú remotingját használva kommunikál
- Az adatbázissal a Spring Data segítségével kommunikál, [7]JPA mint [8]ORM szolgáltató fölött, [9]JDBC segítségével
- A kérés és válasz sorral HTTP és REST alapon kommunikál, azonban ezt elfedi előlünk az Amazon Java SDK által biztosított kliens.



4.4.3.1 AZ ADATBÁZIS KAPCSOLAT RÉTEGEI

Felügyelő (Overseer):

A felügyelő nyilvántartja a rendszer erőforrásait (adatbázisok, kérés fogadók, kérés feldolgozók), azok állapotait és ő felel az erőforrás összerendelésekért.

Amikor egy új feldolgozó, vagy kérés fogadó példányt indítunk az először regisztrálja magát a felügyelőhöz, lekéri a konfigurációs adatokat (timeout értékek, thread pool méretek, reporting gyakorisága) majd igényli a működéséhez szükséges erőforrásokat (adatbázisok, üzenet sorok). A felügyelő az erőforrások kiosztásakor figyelembe veszi azok terheltségét, egy bizonyos szint felett ha lehetősége van rá új erőforrást visz be a rendszerbe.

Azokat az erőforrásokat, amik nem képesek maguktól periodikus jelentések küldésére (például az üzenetsorok), a felügyelő pollingal figyeli, amennyiben egy meghatározott ideig nem válaszol, az erőforrás kikerül a

[7] Java Persistence API

[8] Object-relational mapping

[9] Java Database Connectivity

rendszerből, az ezt használó szerver példányok pedig értesülnek erről és új példányt kapnak (amennyiben létezik ilyen a rendszerben).

A rendszer terhelését figyelve a felügyelő magától is dönthet úgy, hogy új erőforrásokat vezet be vagy állít le. Egy jó metrika például a rendszer állapotának megfigyeléséhez a kérés sor méretének vizsgálata, amennyiben a benne lévő kérések száma az időben egyre növekszik, a felügyelő új feldolgozó szervert indíthat.

A rendszer pillanatnyi állapotát a külső hívások számára is elérhetővé teszi a felügyelő, így az adminisztrációs felületről tudunk erről tájékozódni és ha szükséges kézzel beavatkozni.

Technikai jellemzői:

- Spring Boot alapú
- A Spring HTTP alapú remotingját használva ajánl ki szolgáltatásokat
- Központi konfigurációt nyújt az egyes komponensek számára

Kérés sor:

A kérés sorban tároljuk a kérés fogadó által befogadott kéréseket, konkrét esetben az Amazon SQS-t használjuk, ami az egyik első AWS szolgáltatás volt, mára igen kiforrottnak tekintjük.

Az SQS elosztottan működik, nagy megbízhatóságú és jól skálázódik, ugyanakkor nincs garancia arra, hogy egy üzenetet csak egyszer kap meg a fogadó. Erre később az üzenet feldolgozó implementációjakor figyelnünk kell.

Működési elve a következő, a beérkezett üzeneteket kétféleképpen lehet kinyerni a sorból, short és long pollingal. Az első esetben az Amazon SDK-n meghívott függvény azonnal visszatér, maximum a megadott számú üzenettel, de lehetséges, hogy eredmény nélkül tér vissza ha nincs üzenet a sorban. Long polling esetén viszont csak akkor tér vissza a függvény, ha a vagy érkezik megfelelő mennyiségű üzenet, vagy a timeout letelik, mi ezt a módszert fogjuk alkalmazni.

Ha már valamely módon kinyertünk egy üzenetet a sorból, akkor ameddig a sorra jellemző visibility timeout idő le nem telik, a többi kliens számára az üzenet láthatatlan lesz a sorban. Ha még ez előtt töröljük az üzenetet, az már senki számára nem lesz lekérhető, ha ez nem történik meg akkor az üzenet visszakerül a sorba.

Ez a viselkedés nagyon hasznos nekünk, hisz ez azt jelenti, hogyha egy kérés feldolgozása közben esetlegesen összeomlana a feldolgozó, a kérés akkor sem veszik el, a visibility timeout leteltével visszakerül a sorba, hogy egy másik feldolgozó lekérhesse.

Technikai jellemzői:

- SaaS felhő szolgáltatás
- REST interfész Amazon SDK által elfedve
- Elosztott, jól skálázható, nagy megbízhatóságú
- Az egyszeres vétel nem garantált
- A kölcsönös kizárás egy üzenetre viszont igen

Válasz sor:

Itt megint előkerül a kommunikáció kérdése, szinkron kliens-szerver kommunikáció esetén (jelen technikai korlátok között) ugyanannak a szervernek kell kiszolgálnia a klienst amelyikhez a kérés befutott. Ez viszont azzal járna, hogy a válasznak a kiszolgáló szerverhez kell érkeznie, vagy a kiszolgáló szervernek meg kellene ismernie a számára elkészített választ.

Bármely megoldás alkalmazása azonban nem kívánatos mellékhatásokkal járna. Első esetben új kapcsolatot vezettünk be a kérés fogadók és feldolgozók között, ami azzal járna hogy az eddigi modellünk amiben a két típusú példány száma egymástól függetlenül változtatható borulna.

Második esetben a sorban lévő üzenetek közül a kérés fogadónak kell tudnia kiválasztani azokat a kéréseket amik a nála lévő kliensekhez tartoznak, ez viszont azzal jár, hogy fent kell tartanunk a megkezdett kommunikáció állapotát és folyamatosan vizsgálnunk a sort.

Látható, hogy eben ez esetben nem a szokásos sor adatszerkezet a legoptimálisabb, sokkal jobb lenne egy érték, a kérés azonosítója szerint kereshető adatstruktúra.

Szerencsére pont ezt a célt szolgálják a kulcs-érték alapú NoSQL adatbázisok, amik ráadásul nagyszerű horizontális skálázódásukkal jól beleillenek a rendszerünkbe válasz sorként. Több alternatíva is rendelkezésünkre áll ilyen komponensből, például a Memcached, Redis vagy az Amazon saját szolgáltatása, a DynamoDB. Végül az utóbbi mellett döntöttünk, a jól skálázhatóság és költséghatékonyság mérlegelése mellett.

Technikai jellemzői:

- Amazon DynamoDB
- NoSQL adatbázis, csak kulcs alapján kereshető
- Jól skálázódik

Adatbázis(ok):

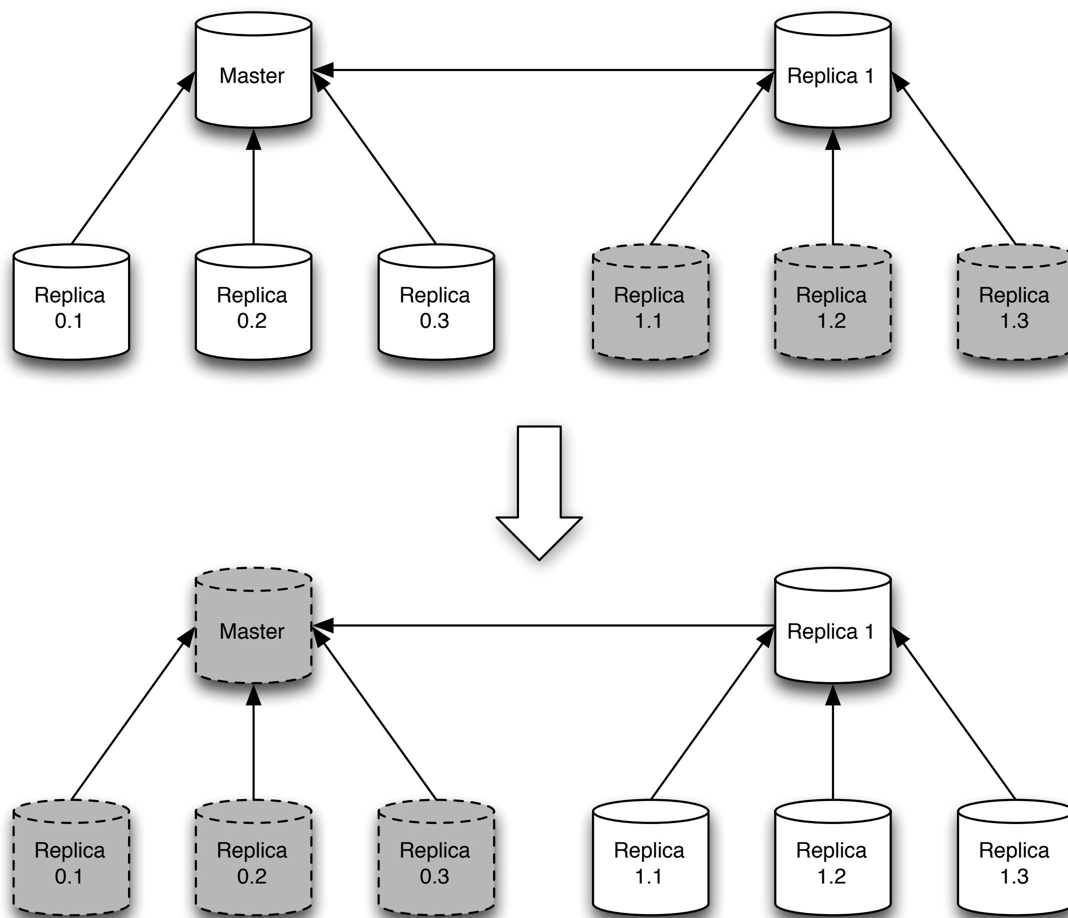
Bár az általános rendszer működéséhez nem szükséges az adatbázis használat, a valódi felhasználás esetén több mint valószínű hogy szükség lesz rá. Mivel az üzleti folyamatok elvégzéséért a kérés feldolgozó felelős, az adatbázis kapcsolatot is ezek a példányok fogják használni.

A hagyomások relációs adatbázisok skálázódása egyáltalán nem egy egyszerű probléma, ezért itt igyekszünk már korábban bizonyított, elterjedt megoldásokat alkalmazni.

Ilyen megoldás a MySQL beépített read replication protokollja, ami olvasási másolatok létrehozását teszi lehetővé egy mester példányról. Az Amazon RDS szolgáltatás (ahol menedzselt adatbázis szerverek közül választhatunk) is támogatja ezen replikák létrehozását és kezelését, egy közös interfészen keresztül.

Az RDS szolgáltatáson keresztül egy példányról maximum 5 másolatot készíthetünk, ám másolatról is lehet másolatot készíteni, legfeljebb 3 mélységig. Ez azt jelenti, hogy összesen 1 + 5 + 25 = 31 adatbázis példányunk lehet.

Probléma az Amazon RDS-el a kötelezően kijelölendő karbantartási ablak, ám az ezáltal okozott kiesést ellensúlyozhatjuk a replikált példányok megfelelő szervezésével, a replikát előléptetve a karbantartás idejére (majd az új mester karbantartásakor újra megismételni).



4.4.3.2. Karbantartás idejére a replikák előléptetése

Technikai jellemzők:

- Amazon RDS - MySQL 5.6
- Több olvasási másolat, eltérő karbantartási ablakokkal
- Kapcsolata az alkalmazásokkal a Java MySQL Connector segítségével

Adminisztrációs alkalmazás:

Bár nem szigorúan része az architektúrának, úgy döntöttünk hogy készítünk egy adminisztrációs alkalmazást, amely a későbbiekben nagyban megkönnyíti a rendszer állapotának monitorozását.

A felügyelő által figyelt metrikákat vezetjük ki az adminisztrátor számára is olvasható formában a felületre. Ezáltal a mérések elvégzése közben folyamatos vizuális visszajelzést kapunk a teljesítmény adatokról, amik hasznosak az esetleges hibák felderítésekor is.

Az aktuális állapot figyelésén túl lehetőségünk van beavatkozásra is, ezáltal méréseink során egyébként nem triviális forgatókönyveket is kipróbálhatunk, például mi történik ha egy kérés beérkezése után megáll a kérés fogadó szerver.

Mivel ezen alkalmazás segítségével jelentősen lehet befolyásolni a rendszer működését, a biztonság itt kritikus. Hogy megvédjük a rendszert a külső, rosszindulatú beavatkozástól ezért ez az alkalmazás csak a virtuális privát felhőn belülről érhető el, és ott is felhasználónév/jelszóval védett. A biztonsági funkciókat a Spring security projekt nyújtja.

Technikai szempontból a webalkalmazás Spring MVC alapú, de a gyárilag nyújtott JSP alapú megjelenítést lecseréltük a Thymeleaf template engine-re, ami modernebb és könnyebben használható platformot nyújt a fejlesztéshez. A Spring MVC moduláris felépítésének és a Thymeleaf natív Spring támogatásának köszönhetően ez a döntésünk viszonylag zökkenőmentesen kivitelezhető volt.

Technikai jellemzők:

- Spring MVC alapú, Spring Boot-al
- Thymeleaf template engine
- Biztonság VPC és Spring Security segítségével

4.4.4. Kommunikációs modell

Korábban már utaltunk az aszinkron és szinkron kommunikációra a kliens és szerver között, most ezt vizsgáljuk meg részletesebben. Természetesen mindkét modellnek megvannak a maga előnyei és az ide tartozó optimális felhasználási módja.

Aszinkron kommunikáció:

Ennek a modellnek a lényege, hogy amikor egy kliens kérést küld a szerver felé a kérés nem kerül azonnal kiszolgálásra, először a rendszer nyugtázza a kérés befogadását és tudatja a klienssel hogy hol fogja megtalálni a választ és előreláthatóan mikor.

Ezután a kliens a megadott idő elteltével a kapott címen próbálkozik, ami alapján a kérésfogadó megnézi, hogy a válasz sorban megtalálható-e a kész válasz. Amennyiben igen azt visszaadja a kliensnek és törli a sorból, ha nincs akkor újabb becslést ad a kliensnek, hogy mikor próbálkozzon újra.

Ennek a megközelítésnek az a nagy előnye, hogy nem kell ugyanannak a kérés fogadónak válaszolnia egy kérés során az adott kliensnek, ezáltal akár egy kérés közben is megállhatnak szerverek, a kérés kiszolgálása ettől függetlenül zavartalan marad.

A rendszer terhelését is csökkenti ez a modell, hisz gyakorlatilag a válaszra várás logikáját kliens oldalra vittük át és a folyamatos kapcsolatokat sem kell fenntartani amíg várunk a feldolgozás befejeződésére.

Azáltal, hogy a kliensnek nem kell megvárnia a tényleges végrehajtást egy válasz fogadása előtt, különbséget tud tenni aközött, hogy a rendszer terheltsége miatt lassú a kiszolgálás, vagy egyáltalán nem elérhető a szerver.

Ugyanakkor ez a modell komplex kliens oldali logikát vár el, közvetelenül nem alkalmazható minden esetben, például egy webböngésző nem támogatja natívan ezt a működést. Ha webes alkalmazást akarnánk fejleszteni ami így kommunikál a rendszerrel, akkor saját klienst kellene írnunk ezekhez a hívásokhoz.

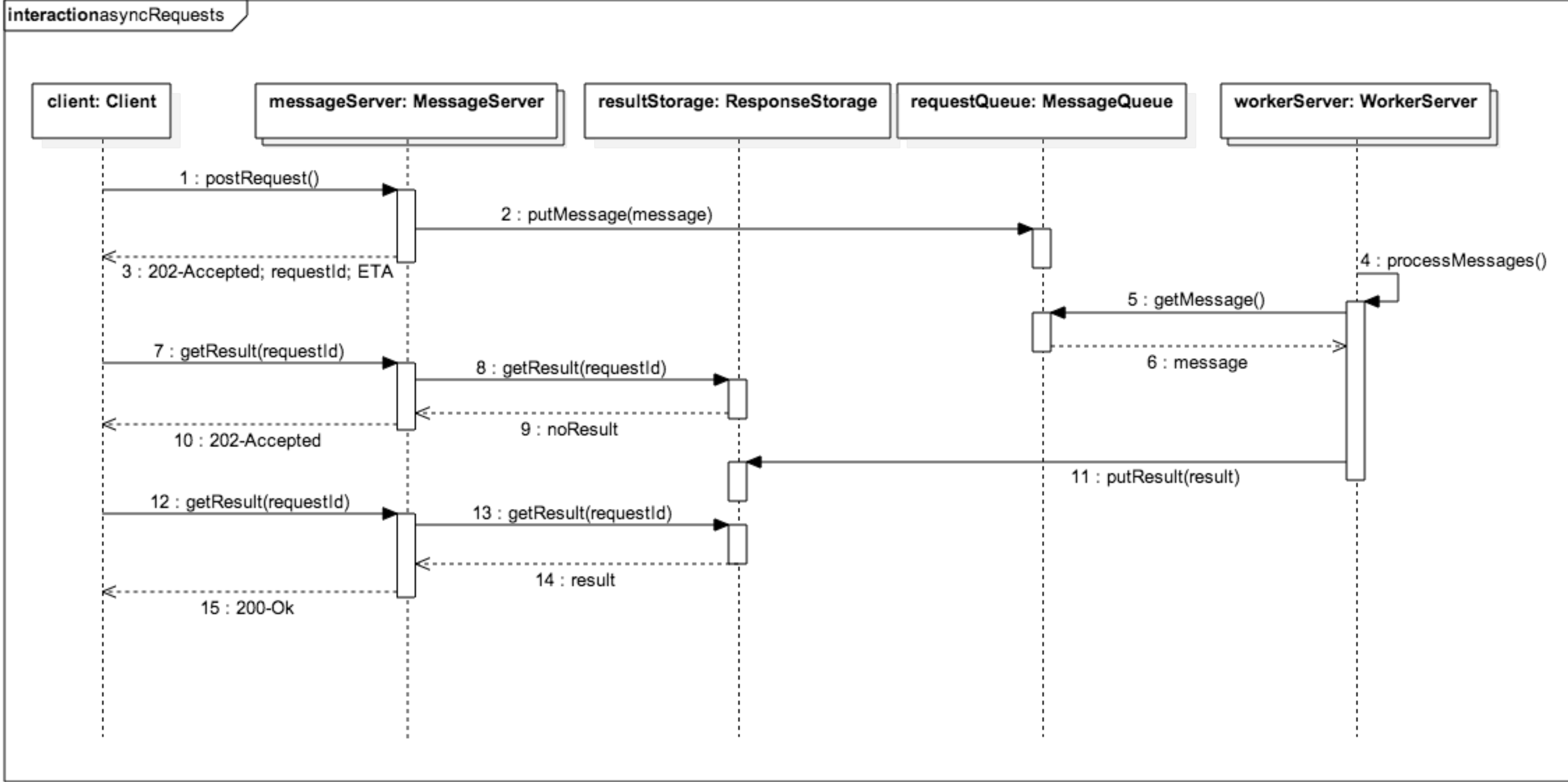
Előnyei:

- Megbízhatóság
- Kis terhelés szerver oldalon
- A kliens különbséget tud tenni a nem elérhető és leterhelt szerver között

Hátrányai:

- Kliens oldali komplexitás

4.4.4.1. ÁBRA - ASZINKRON KOMMUNIKÁCIÓ



Szinkron kommunikáció:

Az aszinkron kommunikációtól annyiban különbözik, hogy itt nem a kliens pollingolja a kérés fogadót, hanem a kérés fogadó tesz egy becslést a kérés sorba tételekor és azután aszerint vizsgálja a válasz sort.

Ennek eredményeként a kliens elől elfedjük az aszinkron feldolgozást, ezzel spórolva a kliens oldali logikán.

Ugyanakkor ha a kérés beérkezése, de a kiszolgálás előtt megáll a kérésfogadó szerver akkor a kérést nem fogjuk tudni kiszolgálni.

Előnyei:

- Egyszerűbb kliens
- Gyorsabb lehet

Hátrányai:

- Megbízhatóság csökken
- Bonyolultabb szerver oldali logika

4.5 A tervezés összegzése

Összességében tehát egy Amazon AWS alapú rendszert fejlesztünk, ahol az egyes komponensek Java nyelven íródnak a Spring keretrendszer felhasználásával.

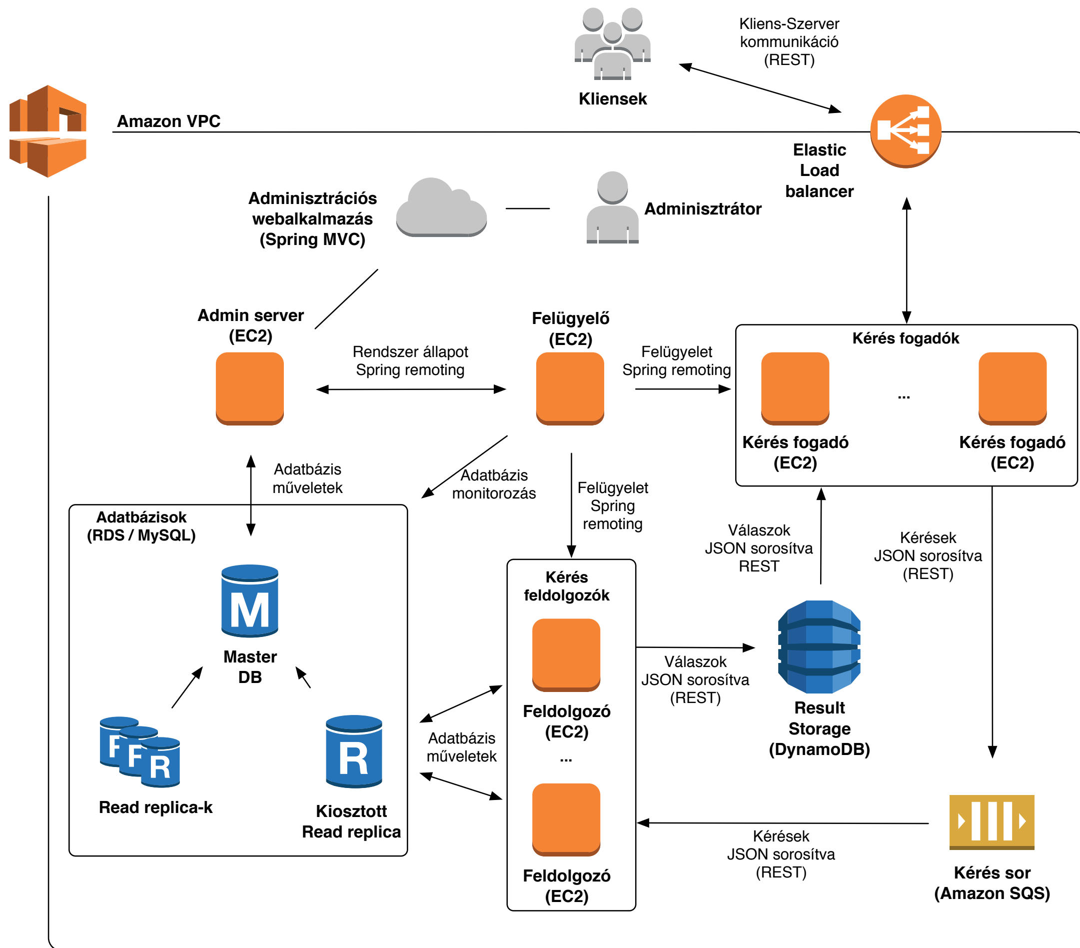
Futtató környezetnek az Amazon EC2 platformot választjuk, ahol Ubuntu 14.04 Server operációs rendszereken futnak az alkalmazás példányok.

Ahol lehet igénybe vesszük a SaaS szolgáltatásokat, ilyen például az üzenetsor és adatbázisok, ám a fejlesztés során igyekszünk ezek konkrét implementációjától független rendszert készíteni, a szolgáltatások egyediségét egy plusz absztrakciós szinttel elfedve.

A korábban felsorolt kommunikációs modellek közül az aszinkront választottuk, ám a szinkron jövőbeni implementálását sem zárjuk ki, úgy alakítjuk ki a rendszert, hogy a két módszer párhuzamosan is tudjon működni.

A rendszer egyelőre amennyire lehet általános és a modularitása miatt a későbbiekben jól illeszthető feladatok széles palettájához.

Az Amazon AWS által nyújtott modellezési eszközökkel az alábbi modellt készítettük a rendszer jelen állapotának szemléltetésére.



4.5.1. A MEGTERVEZETT RENDSZER AZ AMAZON AWS MODELLEZÉSI ESZKÖZEIVEL ÁBRÁZOLVA

5. A rendszer fejlesztése

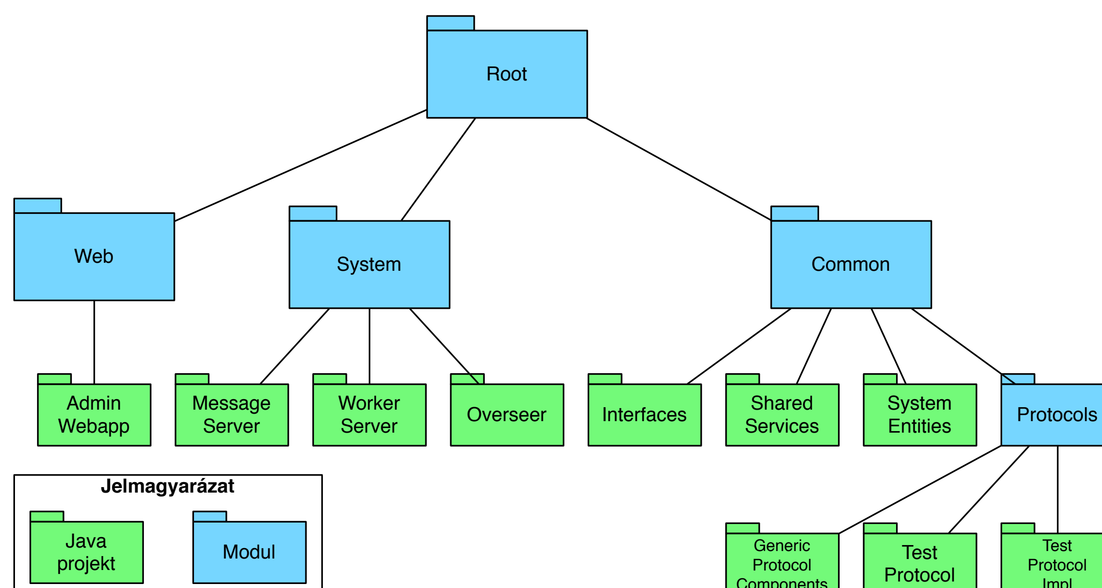
A tervezés végeztével a rendszer és a mérések elvégzéséhez szükséges eszközök implementálása a következő lépésünk. Ebben a fejezetben a fejlesztés során felmerült kérdéseket, szerzett tapasztalatokat, levont következtetéseket összegezzük.

5.1 Fejlesztési folyamat

Költség takarékossági szempontból a fejlesztés során amikor lehetett igyekeztünk lokális erőforrásokat használni. Ezt szerencsére lehetővé tette az Amazon azáltal, hogy az ott használt szolgáltatásokat (DynamoDB, SQS) a felhőn kívülről is elérhetővé tette (bár ekkor a válaszidő értelemszerűen megnövekedett).

A fejlesztés során először az egyes komponensek vázát készítettük el, majd miután a kommunikációt is létrehoztuk közöttük elkezdtük a funkciókat egymásra épülésük függvényében implementálni. Értelemszerűen először a felügyelővel kezdtük, hisz hozzá fordul induláskor a többi komponens a beállításaiért.

Az architektúra modularitását igyekeztünk a fejlesztés során is fenntartani, valamint ahol lehet javítani rajta. Ennek eredményeként az alábbi projekt struktúra állt elő (a projekt szervezéséhez és dependenciáinak kezeléséhez a Mavent használtuk, ezzel részletesebben a következő részben foglalkozunt).



5.1.1. A RENDSZER PROJEKT STRUKTÚRÁJA

Az egyes modulokat részletesebben alább vizsgáljuk meg.

5.1.1 Web modul

Ebben a modulban vannak a webes komponensek projektei, jelen esetben csak a már korábban elemzett adminisztrációs webalkalmazás található itt, de későbbiekben amennyiben bővíteni akarjuk a rendszert webes komponensel az is ide kerülhet.

5.1.2 System modul

A rendszer komponensei vagy azok kódbeli reprezentáció találhatóak itt. A kérés fogadó, feldolgozó és felügyelő implementációi találhatóak ez alatt a modul alatt.

5.1.3 Common modul

A több komponens által is használt projektek gyűjtő csoportja. Például a SystemEntities projektben a rendszer komponenseket feleltetjük meg kódbeli entitásoknak és képezzük le az előbbiek műveleteit az utóbbiakéra, egyelőre csak interfész szinten (ezek implementáció vagy a SharedServices projektben, vagy az adott felhasználási helyen találhatóak).

A SharedServices-ben olyan szolgáltatások találhatóak amiket több komponens is használ, például a kérés sor kapcsolat implementációja.

Az Interfaces projekt feladata a távoli szolgáltatások interfészeinek összegyűjtésee.

5.1.4 Protocols modul

Az általános rendszeren belül a konkrét üzleti logikát az ebben a projektben lévő alprojektek szabják meg. A rendszer teljesítményének teszteléséhez készítettünk egy teszt protokollt, ami különböző erőforrás igényű műveleteket tartalmaz, erre később még részletesebben kitérünk.

5.2 Eszköz választás

A fejlesztés támogatásához több eszközt is használtunk, itt azokat soroljuk fel ami a projekthez kapcsolódik, nem az egyedi fejlesztőkhöz.

Git:

Verziókezeléshez a Git-et választottuk, a saját szerverünket pedig egy Amazon EC2 példány futtatta, így biztosítva volt a magas rendelkezésre állás és az adatbiztonság.

Maven:

A projekt strukturálására és függőségeinek kezelésére a Maven-t használtuk. Segítségével egyszerűen fordítható és tesztelhető, majd telepíthető a teljes projekt.

SonarQube:

A kód minőség ellenőrzésére és a projekt fejlődésének követésére használtuk. A Spring keretrendszer alkalmazása miatt az egyes szabályokat testre kellet szabnunk, ám még így is nagy segítségnek bizonyult.

6. A kész rendszer vizsgálata

Az eddig leírtak ellenőrzése céljából az elkészült rendszert egy sor tesztnek vetettük alá, hogy megfigyeljük a viselkedését a különböző esetekben. Sajnos anyagi korlátok miatt nem tudtuk tetszőlegesen nagy méretig skálázni a rendszert, úgyhogy a méréseink során igyekeztünk inkább a viselkedést és jelenségek vizsgálatát előtérbe helyezni, mint a puszta teljesítmény adatokét.

6.1 Telepítési környezet

A mérések elvégzéshez a rendszer már teljes egészében az Amazon AWS-en futott. Költség és teljesítmény szempontok mérlegelésével úgy döntöttünk, hogy t2.small EC2 példányokat fogunk futtatni, ezek paraméterei.

EC2 példányok:

- 1 CPU
- 2GB Memory,
- 8GB EBS Disk.

DynamoDB beállításai:

- 10 Provisioned Read Capacity Unit
- 5 Provisioned Write Capacity Unit
- Key: ResponseId (String)

SQS üzenetsor beállításai:

- 30s Visibility Timeout
- 256 KB Max Message Size
- 1 day Message Retention Period

RDS adatbázis beállításai:

- t2.micro host
- 5GB SSD
- MySQL 5.6

6.2 Mérési eszközök és konfiguráció

A mérések elvégzéséhez több eszközre is szükségünk volt. Egyrészt konkrét szolgáltatások kellenek a szerver oldalon amit mérni tudunk, hisz eddig csak általános rendszerről beszéltünk, másfelől kell egy mérési eszköz is.

6.2.1 Szerver oldal

A korábban már említett TestProtocol projekt keretein belül implementáltuk a rendszer által nyújtott, tesztelést segítő szolgáltatásokat. Ezek elkészítése közben igyekeztünk különböző erőforrás igényű műveleteket végezni, hogy tudjuk figyelne a rendszer reakcióját ezekre. Végül öt teszt szolgáltatást implementáltunk:

Test A - Olcsó számítás:

Az első tesztünk egy egyszerű és olcsó számítási művelettel jár a feldolgozó oldalon, két számot összeadunk és az eredményt küldjük vissza válaszként.

Test B - Drága számítás:

A második teszt esetén két nagy (500-1000), négyzetes mátrixot töltünk fel először véletlen szerű számokkal, majd szorzunk össze. Ennek a műveletnek a fejlesztése során nem volt különösebben fontos szempont a minél jobb optimalizáció, hisz pont számítás intenzív feladatot akartunk létrehozni. Ez azt eredményezte, hogy a naív implementáció miatt ez egy igencsak drága művelet, mint később az eredményeknél látni fogjuk.

Test C - Olcsó adatbázis:

Ebben a tesztben egy olcsó adatbázis műveletet hajtunk végre, egy véletlenszerű számokkal megtöltött táblából választunk ki egy rekordot.

Test D - Drága adatbázis:

Ebben a tesztben egy valamivel drágább adatbázis műveletet végzünk, egy véletlen szerűen választott rekordot illesztünk egy másik táblabeli rekordhoz.

Test E - Drága hálózat:

Itt egy relatív nagy üzenetet küldünk a kliensnek, sajnos ennek a tesztnek az implementációja során beleütköztünk az Amazon AWS korlátaiba, hisz az üzenet sorba maximum 256 KB méretű üzeneteket rakhatunk, a DynamoDB-be pedig 400KB-ot.

A figyelmes olvasónak feltűnhet, hogy az eddigi konvenciókkal szemben nincsen olcsó hálózat teszt, ezt az indokolja hogy az összes többi tekinthető ilyen tesztnek, hisz azok válaszai maximum százas nagyságrendű karakterből állnak.

6.2.2 Kliens oldal

A rendszer teljesítményének méréséhez egy saját, egyedi programot készítettünk. Ebben implementáltuk a korábban tárgyalt aszinkron kommunikációt. A program szintén Java nyelvű és parancssorból futtatható, ezáltal több helyről is könnyebben tudtunk tesztelni.

Bemenet:

Bementeként a program egy-egy teszt leírását várja el, ezt egy XML struktúra adja meg. A bemeneti nyelv "testrun"-okból épül fel, ezen belül lehet egy kérést szimuláló "single" és több párhuzamos kérést szimuláló "paralell" teszteset, "testcase". Megadhatjuk, hogy egy teszteset hányszor fusson le, vagy egy párhuzamos kérésből hány legyen egyszerre.

A program a HTTP Basic access authentication-t is támogatja, azaz felhasználónév, jelszó párossal azonosíthatjuk a felhasználót. Ezt a "credentails" alatt tehetjük meg, míg a tesztelendő szolgáltatásokat a "services" alatt sorolhatjuk fel.

Példának tekintsük meg az alábbi bemenetet.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<tests>
    <credentials>
        <credential id="u1">
            <user>Testuser1</user>
            <password>testpass</password>
        </credential>
    </credentials>
    <services>
        <service id ="s1">
            <name>Service 1</name>
            <url>http://...</url>
        </service>
    </services>
    <testcases>
        <testcase id="t1">
            <serveice>s1</serveice>
            <credential>u1</credential>
        </testcase>
    </testcases>
    <testruns>
        <testrun>
            <single>
                <testcase runs="3">t1</testcase>
            </single>
            <parallel>
                <testcase sim="3">t1</testcase>
            </parallel>
        </testrun>
    </testruns>
</tests>
```

Kimenet:

Az alkalmazás a tesztek eredményét egy .csv fájlba menti, amely az alábbi értékeket tartalmazza.

Client IP	Request ID	Service ID	Request send Time	Response receive Time	RT	RPT	RTT	Comment

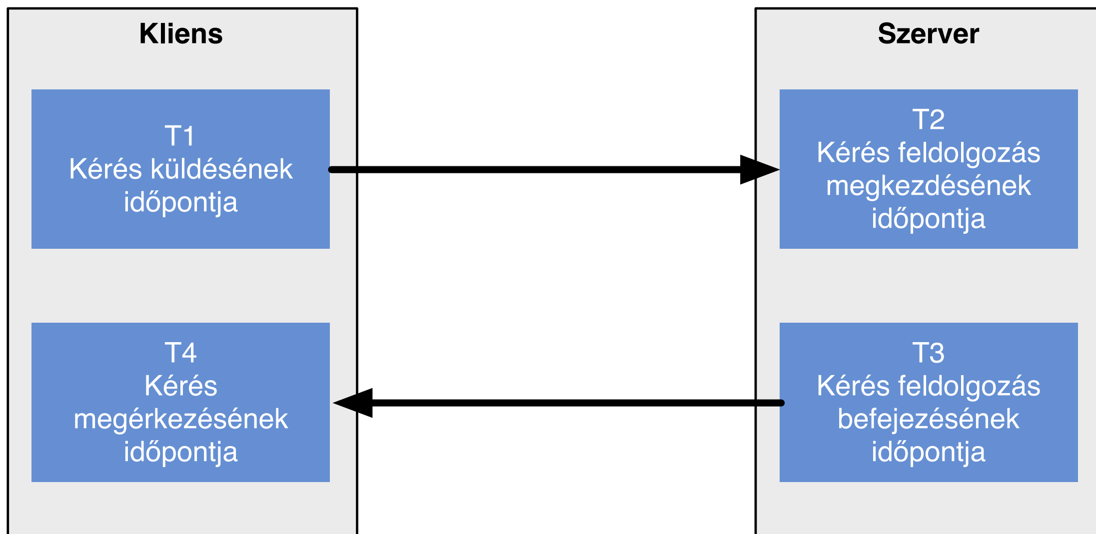
Client IP:
A tesztelő gép IP címe.

Request ID:
A tesztelendő kérés azonosítója, ami alapján azonosítani lehet a konkrét kérést.

Service ID:
A tesztelendő szolgáltatás azonosítója, ami megadja, melyik szolgáltatást teszteljük.

Request send Time:
A kérés indításának ideje milliszekundum pontossággal.

Comment:
Sikeresen megérkezett-e a válasz a tesztprogramhoz, vagy az esetlegesen fellépő hiba.



RT (Response time):
T4 - T1, a kérés elküldésétől a válasz beérkezéséig eltelt teljes idő.

RPT (Response processing time):
T3 - T2, a kérés feldolgozásának ideje a rendszeren.

RTT (Round trip time):
RT - RPT, a kérés "utazási" ideje, beleértve a kliens-szerver és a rendszeren belüli továbbításának idejét is.

Méréseink tervezése során felmerült az óra szinkronizáció problémája. Ahhoz, hogy a kliens és szerver oldalán mért időket össze tudjuk hasonlítani a két órának szinkronban kell járnia. Legalább olyan pontosnak kell lennie, hogy az az eredmények szempontjából elhanyagolható hibát okozzon csak a két idő különbsége.

Sajnos az óra szinkronizációra használt [10]NTP protokoll nem tud ilyen pontosságot garantálni az Amazon AWS felhőben lévő szerver és helyi kliens gép órái között. A különbség több tíz, de akár száz milliszekundum is lehet, ami összemérhető léptékű a tesztjeink futási idejeivel.

Ezen limitáció hatásainak minimalizálása érdekében csak olyan értékeket számoltunk, ami megkapható egy számítógép órája alapján. Például a hálózatban töltött időt nem a triviális (T2-T1) + (T4-T3) formulával számoltuk, hanem kihasználtuk azt a tényt, hogy a teljes idő és számítási idő különbsége biztosan ezt adja meg, amiket viszont egy-egy óra alapján könnyedén kiszámolhatunk.

[10] Network Time Protocol

6.2.3 Mérési helyek

A méréseket több helyről végeztük, elkerülendő az adott mérési hely sajátosságaiból adódó torzításokat az adatokban.

Egyrészről használtuk a saját rendelkezésre álló erőforrásainkat, de a legpontosabb méréseket a BME Informatikai Központ felhő szolgáltatása által nyújtott virtuális gépekről tudtuk elvégezni, hisz itt ipari szintű szolgáltatásokat tudtunk igénybe venni, amiket nem befolyásolt a fogyasztói hálózat.

6.3 Eredmények értékelése

Minden tesztesetnél megvizsgáltuk az teljes válaszidőt (RT), a feldolgozási időt (RPT) és a hálózati időt (RTT).

Az egyes mérések eredményeinek értékeléséhez fontos megjegyeznünk az alábbiakat.

6.3.1 Válaszidő mérése (RT)

A teljes válaszidő mérését az aszinkron kommunikációs modell miatt nagyban befolyásolja a rendszer által küldött becslés (ETA) a válasz elkészülési idejére vonatkozóan. Azért ilyen meghatározó ez az érték, mert ennek letelte után fordul a kliens az elkészült válaszért a szerverhez.

A kiinduló értéke a mérés során a becslésnek 50ms volt, jól meg is figyelhető, hogy 50 ms-os periodicitással érkeznek nagy tömegben a válaszok.

6.3.2 Feldolgozási idő mérése (RPT)

A feldolgozási idő a kérés feldolgozásának szerveren mért ideje, ennek nagysága erősen függ a szerver pillanatnyi terhelésétől. Mivel publikus felhőben dolgoztunk az adott virtuális gép host gépének terhelése zajként jelenik meg. "Bérlőtársaink" erőforrás foglalása is érzékelhető (CPU Steal time) [W6].

6.3.3 Utazási idő mérése (RTT)

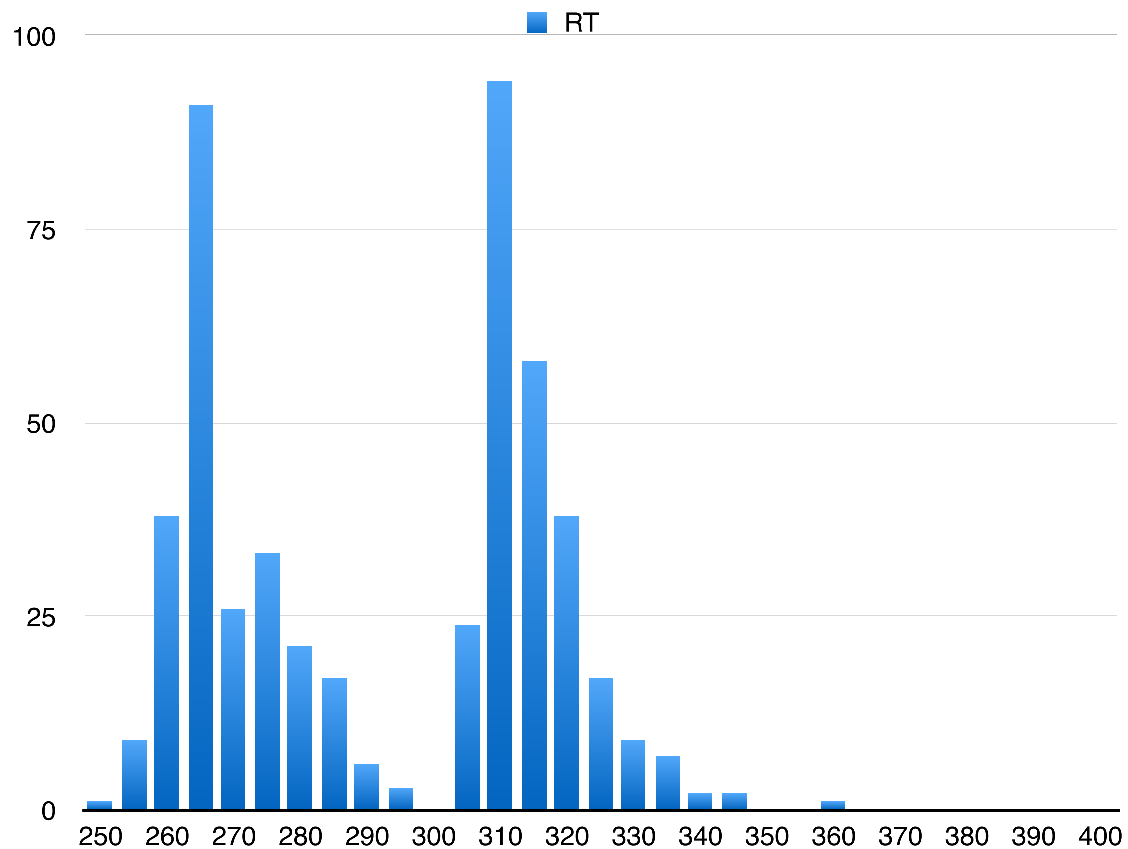
A korábban említett óraszinkronizációs probléma miatt sajnos nem volt lehetőségünk az egyes hálózati utak sebességét külön-külön mérni, így ez az érték inkább az adott szolgáltatás teljes hálózati terhelését mutatja.

Mérési eszközeink pontatlansága miatt ezeket az eredményeket azonban érdemes fenntartásokkal kezelni, rájuk alapozni csak a pontatlanságuk figyelembe vételével szabad!

6.3.1 A teszt

Az A teszt kis költségű számítást (összeadás) végez a szerver oldalon, majd annak eredményével tér vissza.

Válaszidő (RT):

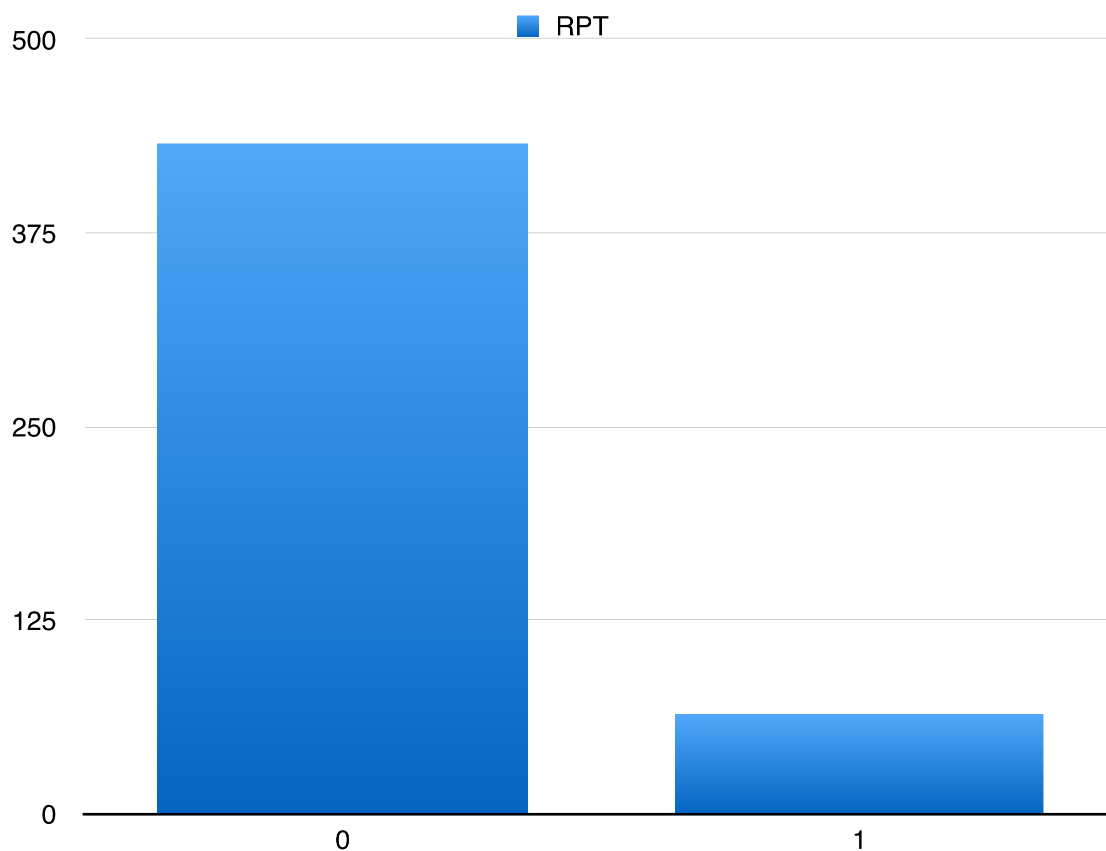


6.3.1.1 Válasz idő az A teszt esetén

Átlag	Medián	Módusz	Min	Max
291,04	304	264	250	358

A válaszidő eredményein jól látszik a korábban már említett csoportosulás az 50 ms-os ETA periódusainak környékén. Az ettől eltérő válaszidők a hálózati késleltetésnek, vagy a kérés fogadó feldolgozási sebességének tudhatóak be, hisz amikor beérkezik egy kérés a feldolgozónak a DynamoDB-hez kell fordulnia, hogy lekérje a kész választ. Bármelyik komponens késik az mérhető lesz a teljes válaszidőn is.

Feldolgozási idő (RPT):



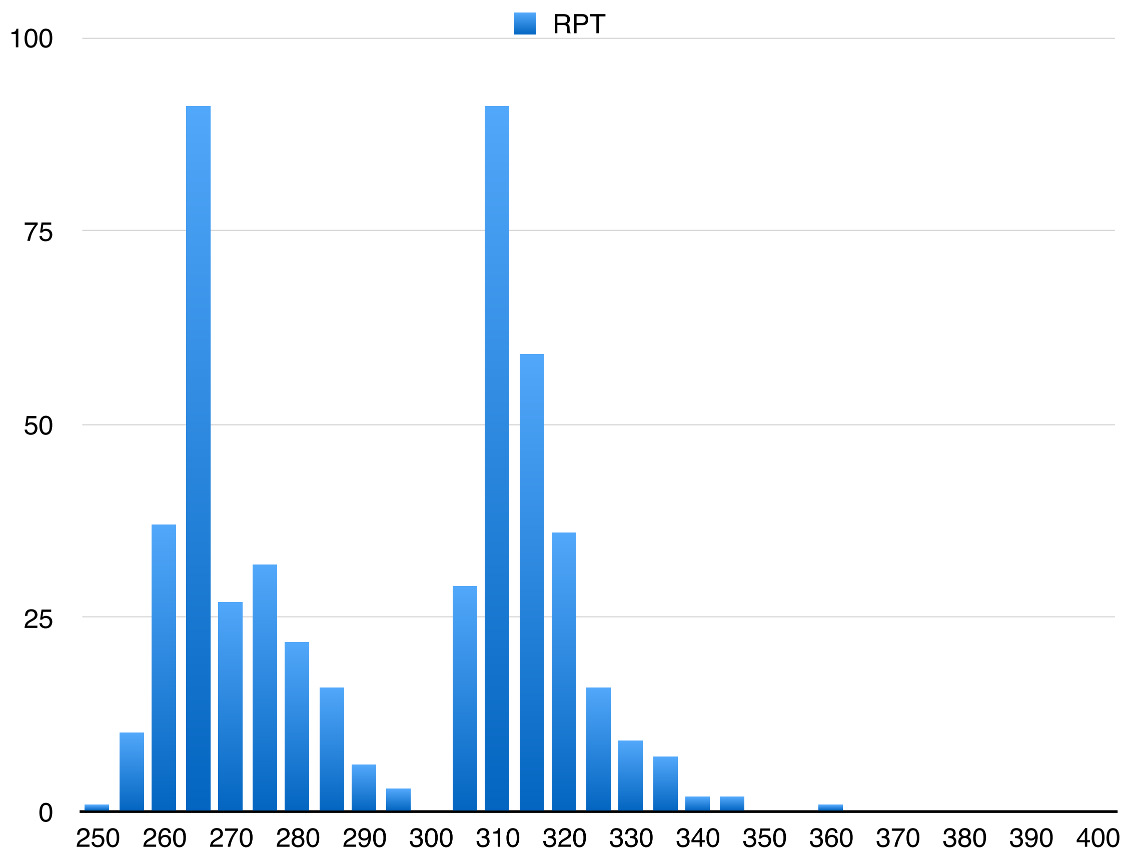
6.3.1.2 Feldolgozási idő az A teszt esetén

Átlag	Medián	Módusz	Min	Max
0,14	0	0	0	5

A feldolgozási idő jól láthatóan pusztán két értéket vesz fel, 0 és 1, ez az időmérés milliszekundumos pontosságának tudható be. A szerver oldalon elvégzett művelet (összeadás) nagyon egyszerű, ennek időbeli költsége nem milliszekundumos nagyságrendben történik.

Ennek következtében elmondhatjuk, hogy a teljes válasz időt nem befolyásolja döntően a feldolgozási idő.

Utazási idő (RTT):



6.3.1.3 Utazási idő az A teszt esetén

Átlag	Medián	Módusz	Min	Max
290,9	304	264	250	358

Az utazási időn is jól kivehető az ETA becslés miatti periodicitás, mivel a kérés feldolgozási időt nem tudtuk mérni, ezért jelentősen csak az RPT értéke befolyásolja az RT értékét.

Összegzés:

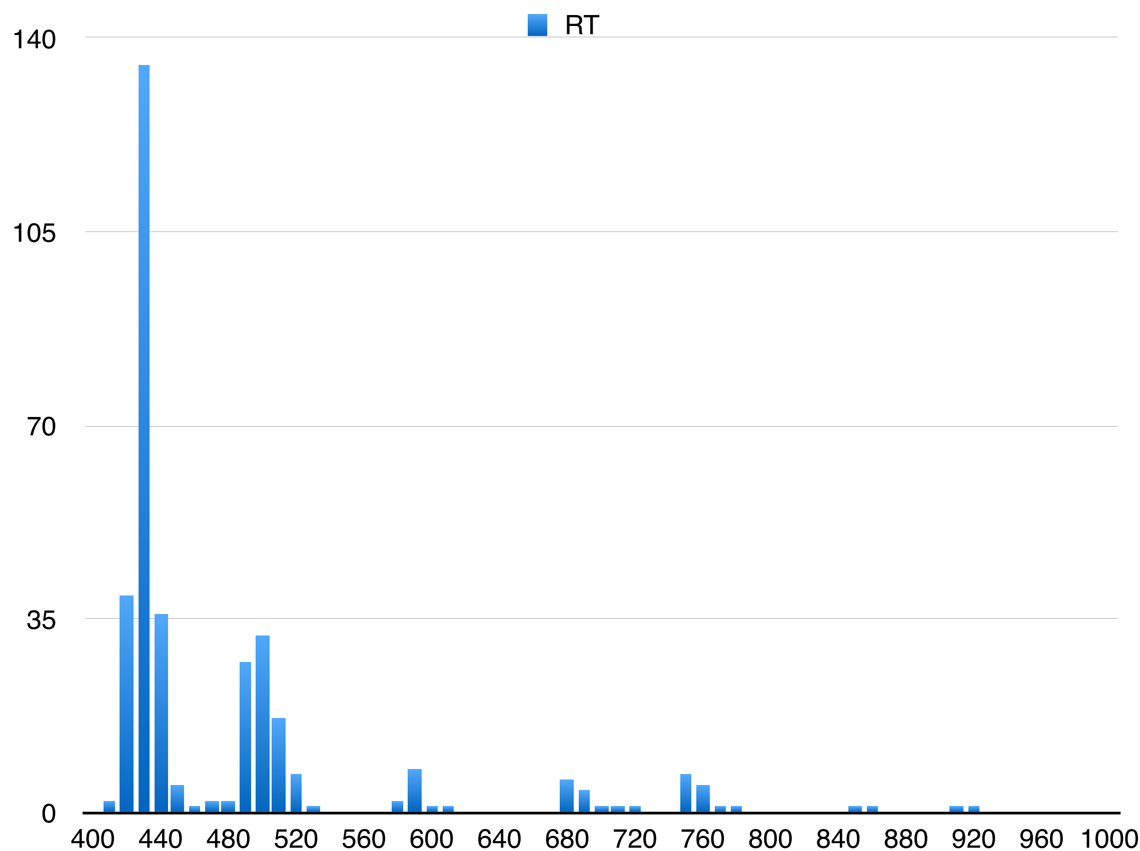
Összességében a kis költségű számítási teszt alátámasztotta a várakozásainkat, tényleg nem a feldolgozási idő határozta meg a kiszolgálás sebességét. Sokkal meghatározóbb az utazási idő, ami alatt értjük a rendszeren belüli üzenet áramlás és a kliens-szerver közötti üzenet áramlás idejét.

Érdemes megfigyelni, hogy a választott kommunikációs modell milyen jelentősen befolyásolta a mérési eredményeinket. Optimistább ETA becsléssel csökkenthetjük a válaszidők periodikus előfordulását, javítva az átlagos kiszolgálási időn, de mindezt csak a kérés fogadó szerverek terhelésének rovására tehetjük meg.

6.3.2 B teszt

A B teszt során egy költségesebb számítási műveletet végzünk el a szerver oldalon. Két véletlenszerűen megtöltött 500x500-as mátrixot szorzunk össze.

Válaszidő (RT):

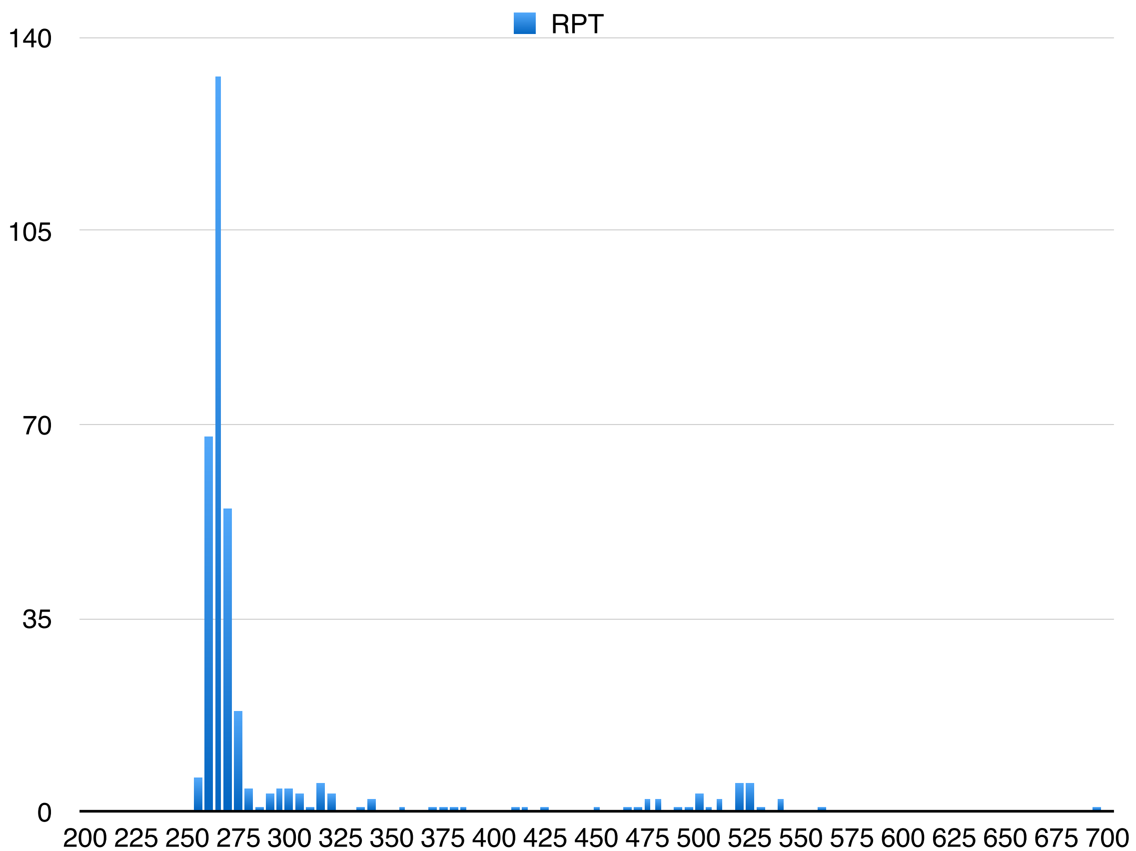


6.3.2.1 Válasz idő a B teszt esetén

Átlag	Medián	Módusz	Min	Max
478,08	430	425	410	1032

A számítás intenzív teszt során már megnövekedett válaszidőt tapasztaltunk, jóval nagyobb értékek között mint korábban. A becslés miatti periodicitás itt is jelen van, ám hatása már nem érződik annyira, lévén a feldolgozási idő is jelentős.

Feldolgozási idő (RPT):



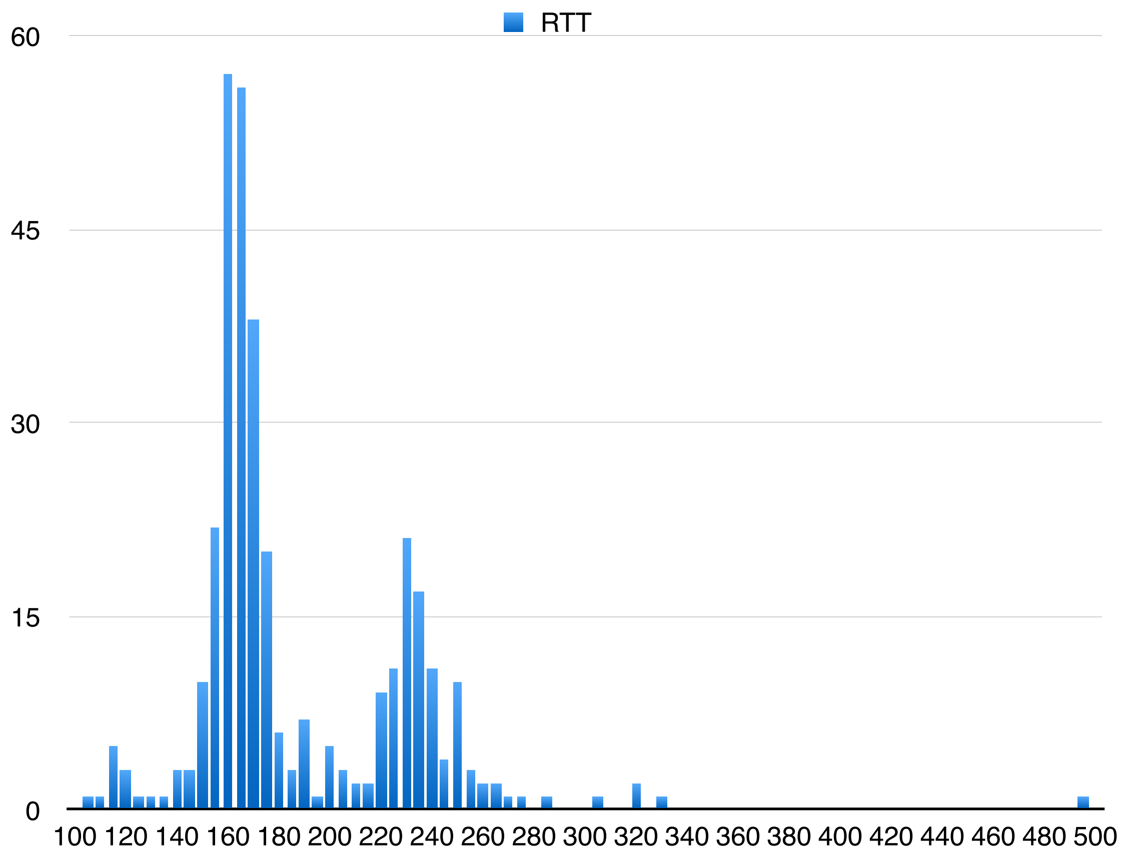
6.3.2.2 Feldolgozási idő a B teszt esetén

Átlag	Medián	Módusz	Min	Max
292,08	264	263	254	740

A feldolgozási idő ezen teszt esetnél már összemérhető a teljes kiszolgálási idővel. Észrevehetjük, hogy bár a kérések nagy része a 260 ms-os kiszolgálási idő környékén mozog, pár esetben jelentős eltérés tapasztalható.

A tapasztalt anomáliák kiváltó okai nagy valószínűséggel a korábban már tárgyalt virtualizációs problémák, valamint a futó szerver példányok szűkös erőforrásai.

Utazási idő (RTT):



6.3.2.3 UTAZÁSI IDŐ A B TESZT ESETÉN

Átlag	Medián	Módusz	Min	Max
185,9	167	158	102	765

Az utazási idő esetén ismét látjuk a korábban már megtárgyalt aszinkron kommunikáció jellegzetességeiből adódó anomáliákat.

Amit érdekes megfigyelni, hogy az előző teszthez képest csökkent az átlagos utazási idő, ezt annak tudhatjuk be, hogy mivel a kérések feldolgozása tovább tart, a válaszok küldése is az időben jobban elválasztott a beérkezésüktől. Ennek következtében az üzenet továbbító komponenseken (válasz és kérés sor, SQS és DynamoDB) az azonos időegységre jutó terhelés is csökken, egyenletesebbé válik.

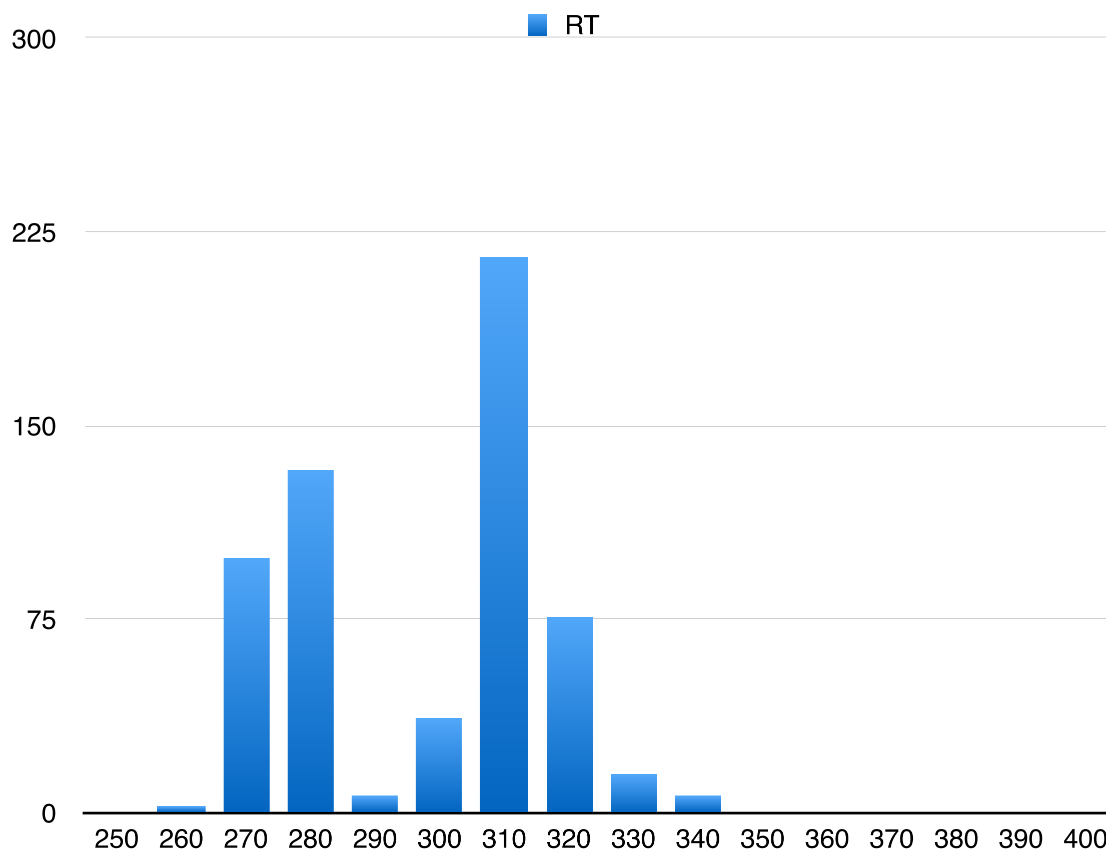
Összegzés:

Ezen teszt során nagyon fontos tapasztalatot vonhattunk le a hálózati késleltetés csökkenéséből, növelni kell az üzenet továbbító komponensek kapacitását, ezzel javíthatjuk a teljes válaszidőt. Szerencsére mindkét esetben (SQS és DynamoDB) ezt könnyen megtehetjük, akár futásidőben is. Érdemes lehet a rendszert később tovább fejleszteni, hogy ezeket az értékeket is adaptívan módosítsa.

6.3.3 C teszt

A C jelű teszt során kis költségű adatbázis műveletet végeztünk szerver oldalon, majd ennek eredményével tértünk vissza.

Válaszidő (RT):

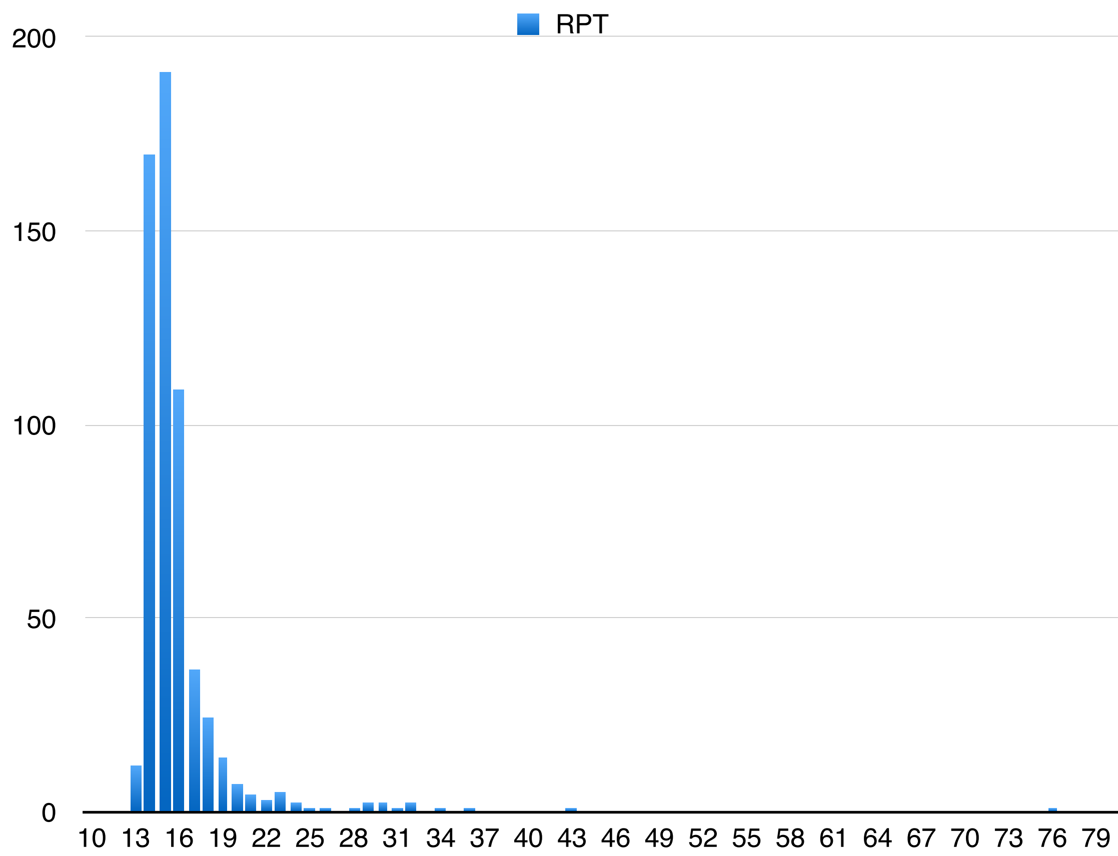


6.3.3.1 Válasz idő a C teszt esetén

Átlag	Medián	Módusz	Min	Max
294,76	301	301	258	916

A kis költségű adatbázis művelettel járó teszt esetén hasonlóságot vélhetünk felfedezni a kis költségű számítási művelettel járó tesztesettel. Itt is megfigyelhető a periodicitás és itt is a hálózati idő dominál a teljes válaszidő eldöntése szempontjából.

Feldolgozási idő (RPT):



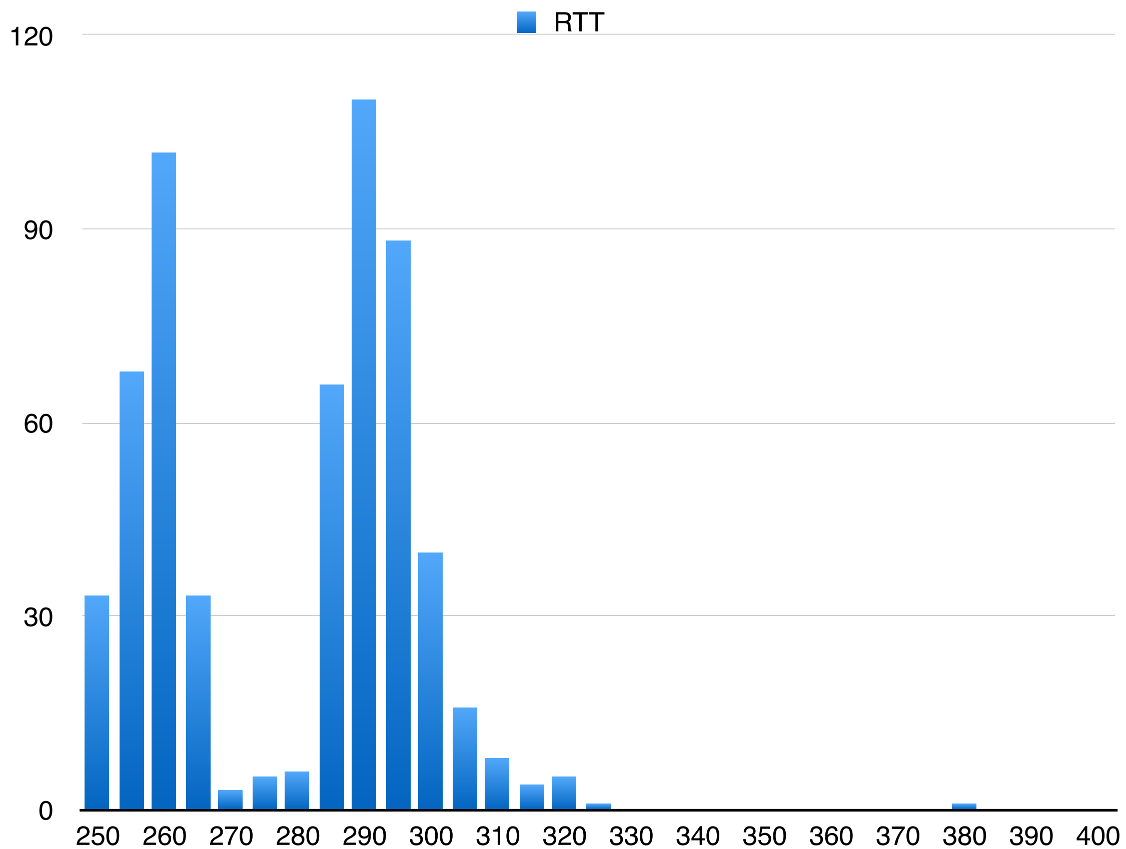
6.3.3.2 Feldolgozási idő a C teszt esetén

Átlag	Medián	Módusz	Min	Max
15,88	15	15	13	76

Ebben a teszt esetben egy kis táblából választottunk ki az elsődleges kulcsa alapján egy kis méretű rekordot.

A feldolgozási időkből látjuk, hogy egy modern relációs adatbázis kezelő milyen konzisztens viselkedést mutat. Az egyszerű SELECT művelet válaszideje végig alacsony maradt, nincsenek kiugró anomáliák, így a többi érték számítását nagyban megkönnyíti ez a viselkedés.

Utazási idő (RTT):



6.3.3.3 Utazási idő a C teszt esetén

Átlag	Medián	Módusz	Min	Max
278,87	285	288	241	884

Az utazási, vagy hálózati idő eloszlásában itt is látható a periodicitás. A válasz üzenetek azonos, kis mérete miatt más adatra nem tudunk következtetni a hálózati idő alapján.

Jelen esetben is, akárcsak az olcsó számítási műveletnél ez az idő befolyásolta leginkább a teljes válasz időt.

Összegzés:

Összességében nagy hasonlóságot fedeztünk fel az olcsó adatbázis és számítási műveletek között, ebből több következtetést is levonhatunk.

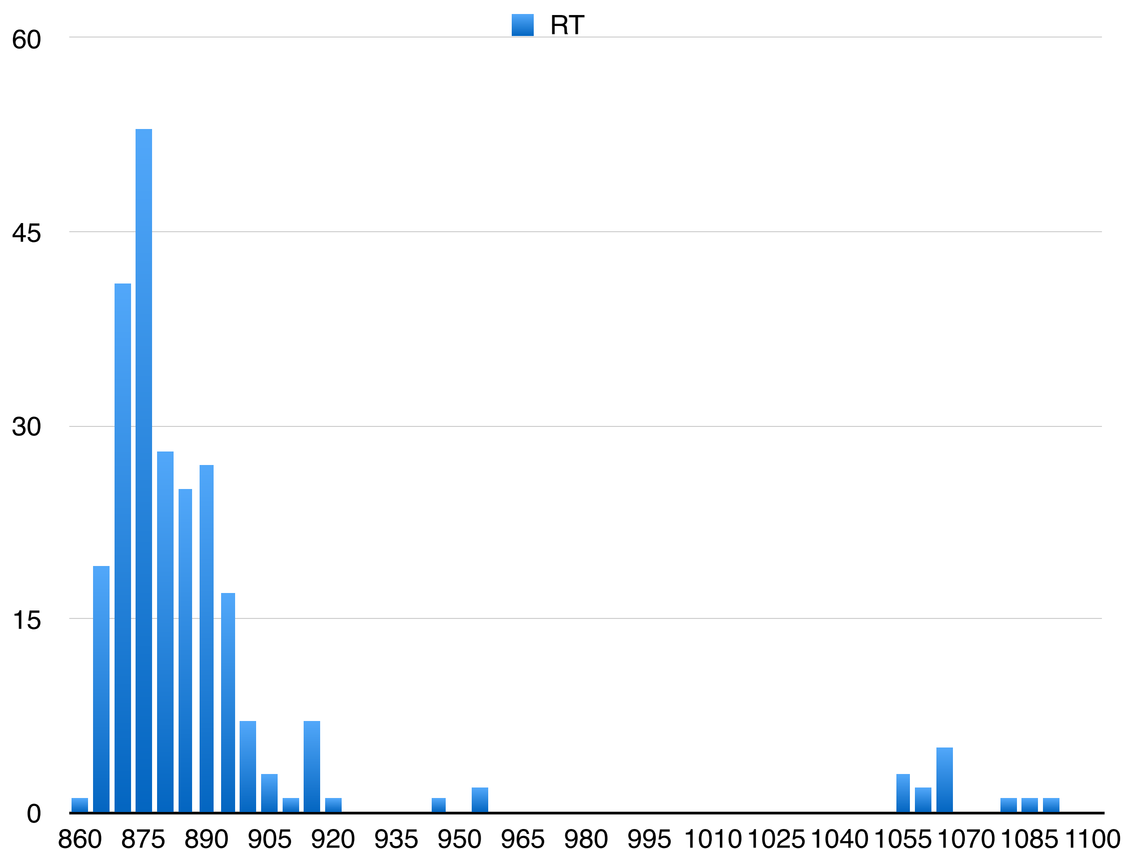
Egyrészt látjuk, hogy a relációs adatbázis kezelés egy igen kiforrott technológia, az egyszerűbb műveletek ebben az esetben számottevően drágábbak mint az egyszerű számítási műveletek (a teljes időre viszonyítva, magukban tízszeres a különbség).

Másrészről a rendszer méretezéséhez megfelelőnek tűnik az adatbázis szerver méretezése, bár messze menő következtetéseket egy ilyen egyszerű teszt esetén még korai lenne levonni.

6.3.4 D teszt

A D teszt során egy költséges adatbázis műveletet végrehajtása volt a cél, ezt több SELECT és JOIN utasítás segítségével értük el.

Válaszidő (RT):

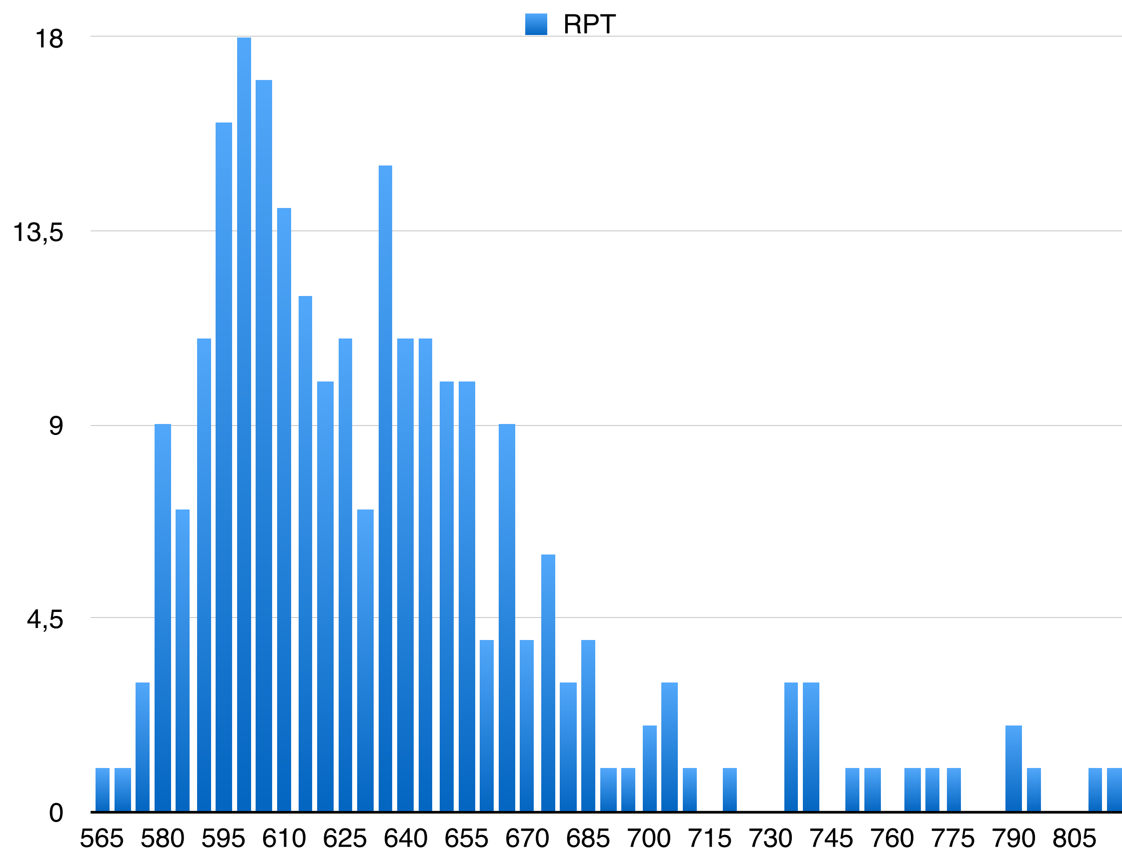


6.3.4.1 Válasz idő az D teszt esetén

Átlag	Medián	Módusz	Min	Max
899,712	877	875	860	2490

Megfigyelhetjük, hogy a feldolgozási idő növekvésével a korábban jellemző periodicitás egyre inkább eltűnik. Ez nem meglepő, hisz egyrészt a feldolgozás komplexitásának növekedésével nem csak annak időbeli költsége nő, de a szórása is. Ezt számtalan tényező okozhatja, a hosszabb műveletek során sokkal jelentősebb hatással lehetnek a művelet végzése közben fellépő terhelések, az ütemező sajátosságai, vagy a virtualizált platform által okozott teljesítmény ingadozások.

Feldolgozási idő (RPT):



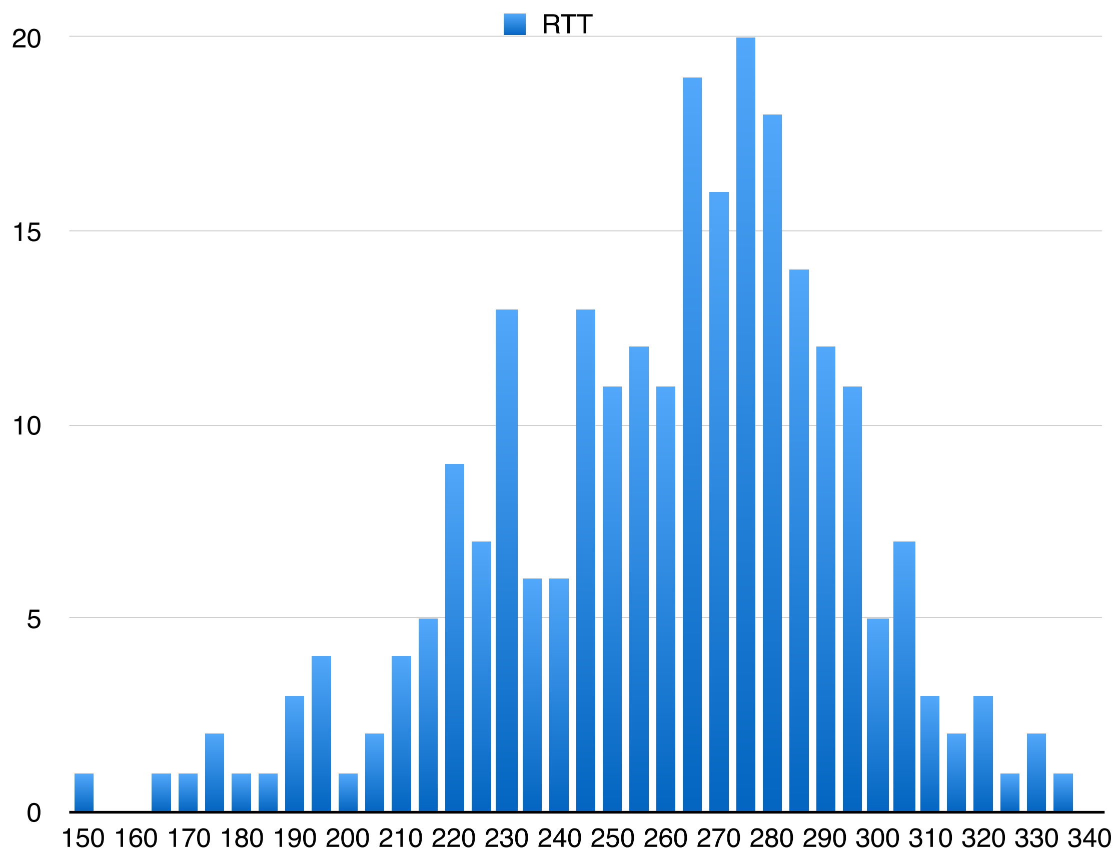
6.3.4.2 Feldolgozási idő a D teszt esetén

Átlag	Medián	Módusz	Min	Max
633,97	623	602	562	902

A drága adatbázis kérés feldolgozási idejei még mindig elég konzisztensnek mondhatóak, ám már itt is jelentek meg nagyobb késleltetésű kérések.

Az itt mért értékeket már jelentősen befolyásolhatják az adatbázis kapcsolat fölé épített rétegek (JPA, Spring Data) különböző jellemzői, mint például a cachelés vagy lazy loading.

Utazási idő (RTT):



6.3.4.3 UTAZÁSI IDŐ A D TESZT ESETÉN

Átlag	Medián	Módusz	Min	Max
265,73	263	271	143	1853

Az utazási idő hasonlít az egyszerű adatbázis művelettel végzett teszt idején tapasztalt utazási időre, ám az ETA első periódusából adódó csúcs itt nincs jelen, hisz eddigre sokkal ritkábban készültek el az adatbázis műveletek. Ebből azt a tanulságot vonhatjuk le, hogy ebben az esetben túl optimista volt a rendszer a kérés kiszolgálási idejét illetően.

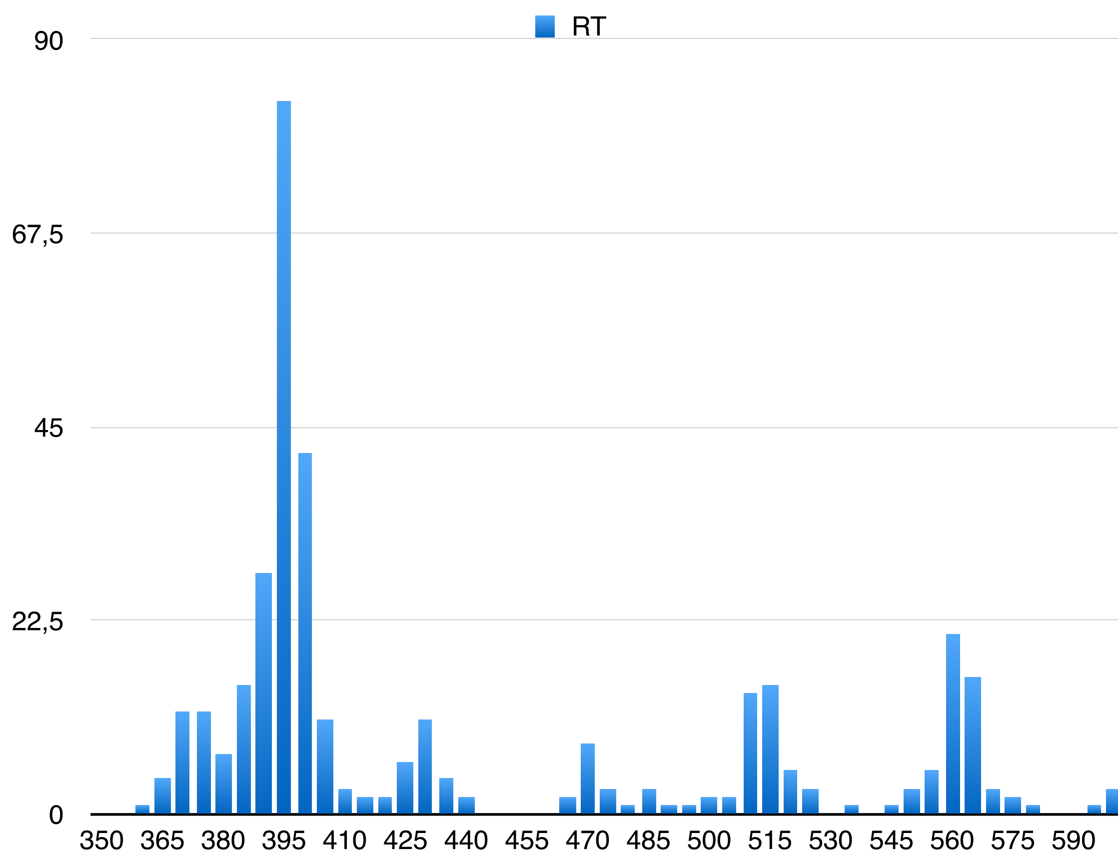
Összegzés:

A drágább adatbázis műveletet megvizsgálva hasznos tapasztalatokat szereztünk a kérések végrehajtási idejének becslésére vonatkozóan. Bár nagyobb terheléssel járt az adatbázis számára ez a teszt, az itt is konzisztens eredményeket mutatott. A teszthez az Amazonon igényelhető legkisebb teljesítményű adatbázist használtuk, így ez a jövőben még sokáig skálázható vertikálisan.

6.3.5 E teszt

Ez a teszt a többitől kicsit eltérő volt, itt leginkább a kérés fogadó áteresztő képességét vizsgáltuk azáltal, az adott válaszok méretét a többihez képest relatív nagyra választottuk.

Válaszidő (RT):

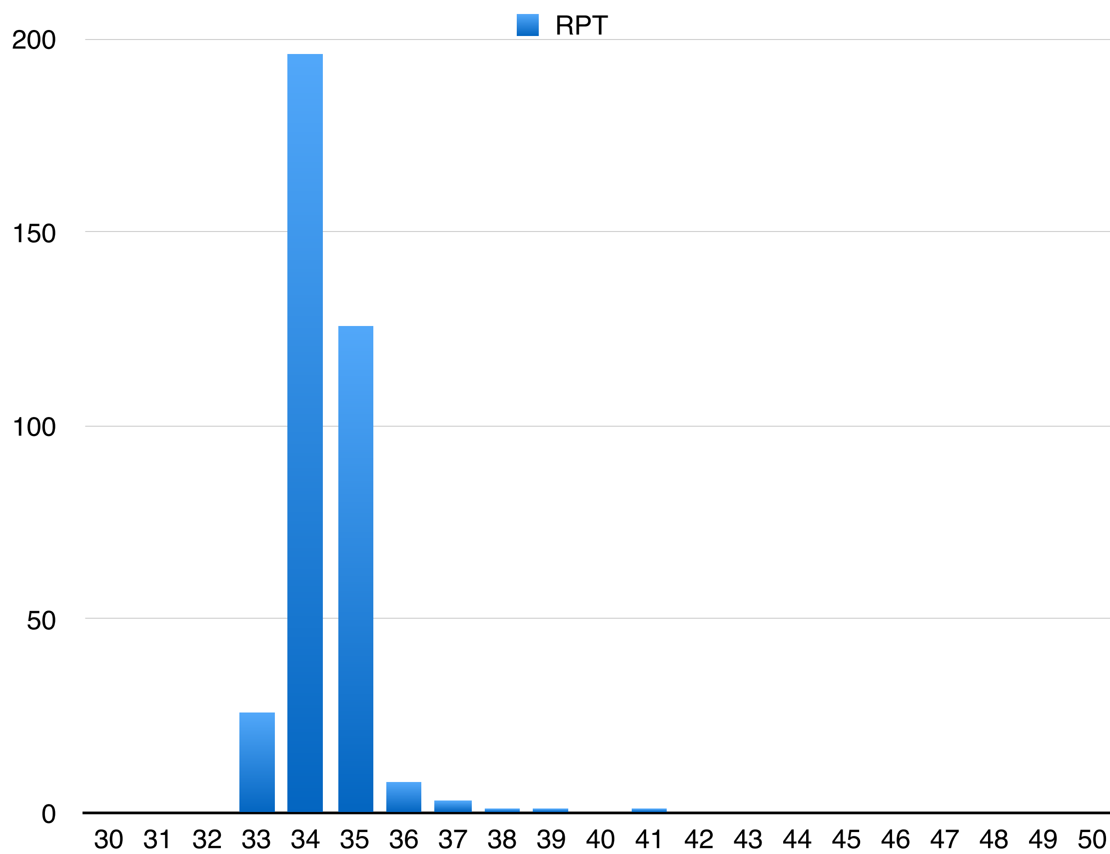


6.3.5.1 VÁLASZ IDŐ AZ E TESZT ESETÉN

Átlag	Medián	Módusz	Min	Max
442,6	397	394	356	1198

A teljes válaszidő sajnos itt erősen függ a kliens kapcsolatától a szerverhez. Emiatt következtetéseket erről a mérésről csak ennek figyelembevételével szabad levonnunk.

Feldolgozási idő (RPT):

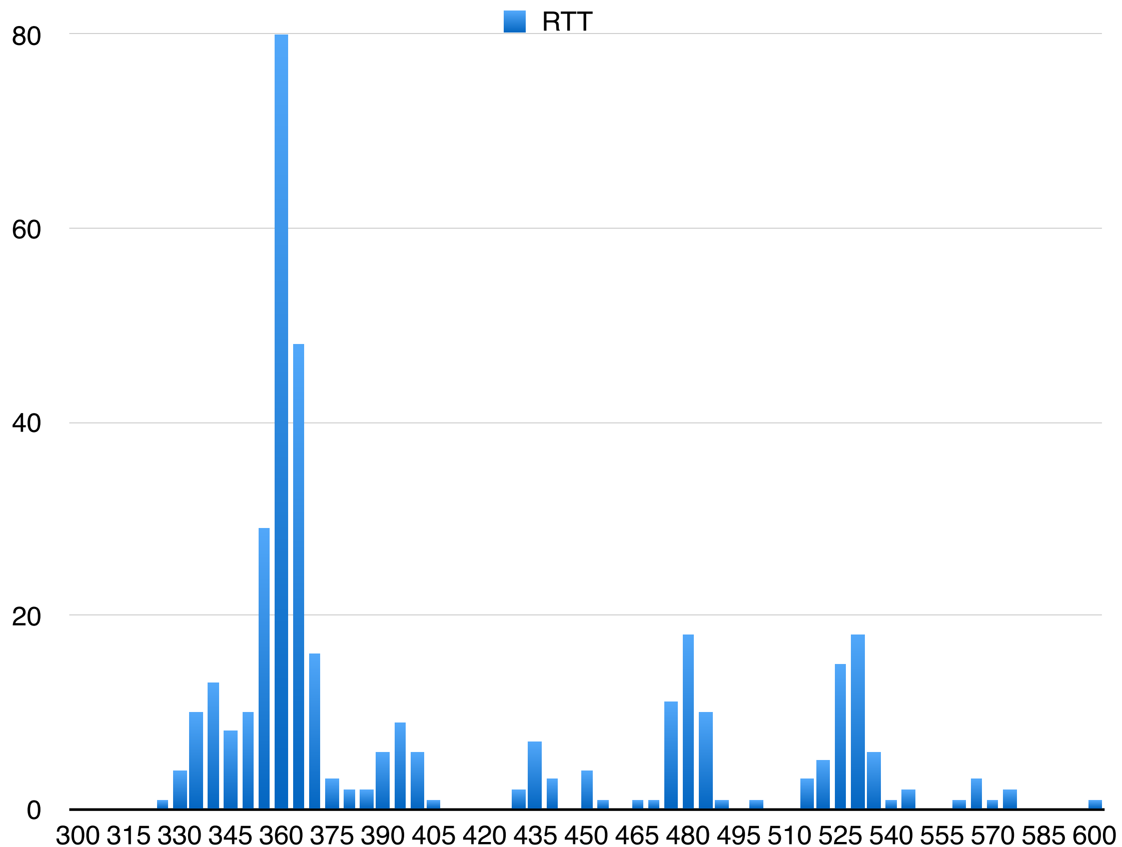


6.3.5.2 Feldolgozási idő az E teszt esetén

Átlag	Medián	Módusz	Min	Max
35,44	34	34	33	138

A feldolgozási idő bár nem elhanyagolható, nem a fő befolyásoló tényezője a teljes válaszidő alakulásának. Látjuk, hogy a válaszok előállítása konzisztensen történik, nagy anomáliák nincsenek jelen az eredmények között.

Utazási idő (RTT):



6.3.5.3 UTAZÁSI IDŐ AZ E TESZT ESETÉN

Átlag	Medián	Módusz	Min	Max
407,15	363	359	321	1162

A tesztet leginkább befolyásoló tényező, sajnos a hálózati késleltetések jelentős zajt visznek bele.

Összegzés:

A válaszok mérete sokkal jobban befolyásolta a hálózati kommunikációt, mint a kérés fogadó áteresztő képességét. Mivel szerveroldalon az Amazon által garantál sávszélességgel rendelkezünk, feltételezhető hogy a kliens-szerver kapcsolat volt a fő befolyásoló tényező ebben a mérésben.

A későbbiekben érdemes lehet számottevően több kliens használatával tesztelni a rendszert, az egyes kapcsolatok folyamatos monitorozásával a pontosabb eredmények érdekében.

7. Levont következtetések, tapasztalatok

7.1 Kezdeti céloknak megfelelés

Az eredeti célkitűzésnek, miszerint egy felhőben működő, adaptív terhelés elosztású rendszert tervezünk és fejlesztünk sikerült megfelelni. A kész rendszer a felhőben fut, a terhelés függvényében adaptívan tud erőforrásokat bevinni és közöttük a terhelést elosztani.

A kész rendszer az architektúrájából adódóan megbízható, a magas rendelkezésre állás szintén biztosított. A központi, futás közbeni konfigurálhatóság segítségével nem kell az egész rendszernek leállnia azok változásakor.

További komponensek hozzáadásával a rendszer horizontálisan jól skálázható, egyedül a felügyelőt kell vertikálisan skálázni. Ugyanakkor fontos megjegyezni, hogy nem ennek a komponensnek a skálázódása adja meg a rendszer maximális méretének plafonját, azt előbb elérjük praktikus okokból (üzenetsorok, adatbázisok akkorára skálázása már nem megoldható az eddig felhasznált technikákkal).

7.2 Korábban hozott döntések értékelése

Mivel a döntési folyamat iteratív, folyamatos mérési és tapasztalati visszacsatoláson alapult nem maradt olyan döntés a rendszerben amit megbántunk volna.

Egy vitatható döntés lehet az aszinkron kommunikáció alkalmazása, hisz ez a modell a relatív magas válaszidő miatt nem alkalmas gyors választ igénylő szolgáltatások implementálására.

Szerencsére erre nincs is szükség, a kommunikáció ezen modelljét olyan felhasználási területeken kell alkalmazni, ahol a válaszidő nem kritikus. Más területeken pedig implementálhatjuk a szinkron modellt, hisz fel van erre is készítve a rendszer.

Technikai döntéseink között sem volt olyan ami hátráltatott volna minket a rendszer elkészítése során.

A felhő platform választását illetően érdekes lett volna megvizsgálni az új piaci szereplők által nyújtott szolgáltatásokat (például DigitalOcean), ám sajnos ez nem fért bele az adott időkeretbe. Mindent összevetve nem találkoztunk gátló tényezővel az Amazon AWS használata közben, látszik hogy kiforrott szolgáltatásról van szó.

8. Függelék

8.1 Nyers mérési eredmények

A nyers mérési eredményeket méretükre való tekintettel nem mellékeltük itt, de az alábbi linken határozatlan ideig megtalálhatóak lesznek:

https://dl.dropboxusercontent.com/u/12210338/eredmenyek.zip