Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

Sallai Tamás

LIFERAY PORTÁLRENDSZER SKÁLÁZHATÓSÁGÁNAK VIZSGÁLATA SZÁMÍTÁSI FELHŐBEN

KONZULENS

Berényi Zsolt

BUDAPEST, 2012

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Sallai Tamás**, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2012. 05. 18.

.....

Sallai Tamás

Összefoglaló

Manapság a webes megjelenés a legtöbb cég számára kritikus, és ez a tendencia a jövőben is folytatódni fog. Ennek legfontosabb céleszköze a vállalati portálrendszerek, amelyek a legtöbb tipikusan felmerülő felhasználási területre beépített megoldást kínálnak, valamint modulárisan bővíthetőek. Infrastrukturális oldalról a számítási felhő terjed rohamosan, aminek a segítségével egy távoli szolgáltatótól lehet bérelni az erőforrásokat, így nem szükséges saját eszközökkel rendelkezni.

Általános probléma azonban, hogy a látogatottság növekedésével már egy kisebb vállalat is átlépi azt a terhelést, amit egyetlen gép kínál. Egy darabig megoldás a szolgáltatások szétbontása, valamint a vertikális skálázás, azonban ezeknél robusztusabb megoldásra van szükség. A számítási felhő hiába ad korlátlan erőforrást, ha a webportál működése nincs elosztva.

Diplomamunkám témája a Liferay portálrendszer eloszthatóságának a vizsgálata a tipikus használati esetekre, valamint ennek a felhőbe való telepítése. Ennek kapcsán megvizsgálom annak a lehetőségét, hogyan oldható meg az, hogy egy ilyen telepített rendszer a terhelés változásával automatizált módon tudja a rendelkezésre álló erőforrásokat szabályozni.

Abstract

Presence on the web is critical for most companies, and this is not likely to change in the future. Enterprise Portal Systems are used for this purpose, as they provide build-in solutions for the common use-cases and are modularly extendable. Infrastructure-wise the computing cloud is expanding rapidly, allowing the rental of resources from a third party, making owning the appliances unnecessary.

It is a common problem that with the increase of users even a smaller company faces that a single computer can not service the traffic. Short term solution may be the distribution of services and vertical scalability, but a more robust solution is needed. The computing cloud's infinite resources are also useless if the portal can not scale horizontally.

My final thesis's theme is the analysis of the Liferay portal for the common use-cases, and it's deployment to the cloud. I also examine the possible solutions to how an installed system can allocate resources in response to increased traffic and free them without human intervention.

1 Bevezető

A mai trendek azt mutatják, hogy a cégeknek elengedhetetlen a webes megjelenés, azonban az ehhez szükséges infrastruktúra kiépítése és üzemeltetése továbbra is drága feladat. Az utóbbi években azonban robbanásszerűen kezdtek elterjedni a felhőszolgáltatások, amelyek pont erre nyújtanak kényelmes, megbízható, valamint használatarányos megoldást. Portáloldalon a Liferay pozíciója egyértelműen stabil, a vállalati környezet, valamint a közösségi hozzájárulások is pozitív tendenciát mutatnak. Természetesen adódik a következtetés, hogy a portált a felhősített infrastruktúrán futtassuk, és ezzel nem csak a használatarányos költségeket, de az egyszerűbb karbantartást és az elasztikus skálázhatóságot is megnyerjük.

Diplomamunkám témája ezen két technológia integrációja, valamint annak az elemzése, hogy egy magyar vállalatnak milyen előnyökkel, illetve veszélyekkel járhat a bevezetés.

Diplomamunkám első felében bemutatom a felhasznált technológiákat és módszereket, majd áttérek a saját munkám ismertetésére. Az első fejezet az általános technológiai áttekintővel indul, itt a Javás architektúrák elosztását elemzem, valamint az elérhető felhő megoldásokat. Ezután áttérek a Liferay bemutatására, először itt is egy általános áttekintést adok, majd a portletfejlesztéshez szükséges elméleti háttérrel folytatom, végül rátérek az eloszthatósági kérdéskörre. A fejezet végén bemutatom a kiválasztott felhőszolgáltatást, amelyre az elkészített alkalmazás is épül.

A saját munkám bemutatásánál először az elméleti oldalról közelítem meg a feladatot, itt a probléma elemzése után a portletfejlesztés tervezési oldalát mutatom be, majd rátérek az algoritmus tervezésére. Ezután következik a megvalósítás rész, itt bemutatom a portlet fejlesztését, valamint a Liferay felhőbe költöztetésének lépéseit. Az utolsó előtti fejezetben az elkészített algoritmus és a portlet gyakorlati kipróbálásának a tapasztaltait mutatom be. Végül következik az elemzés, ennek során valós üzemeltetési adatokon futtatom és vizsgálom az algoritmust, így reálisabb képet kapva arról, hogy számszerűen mennyire is lehet hasznos.

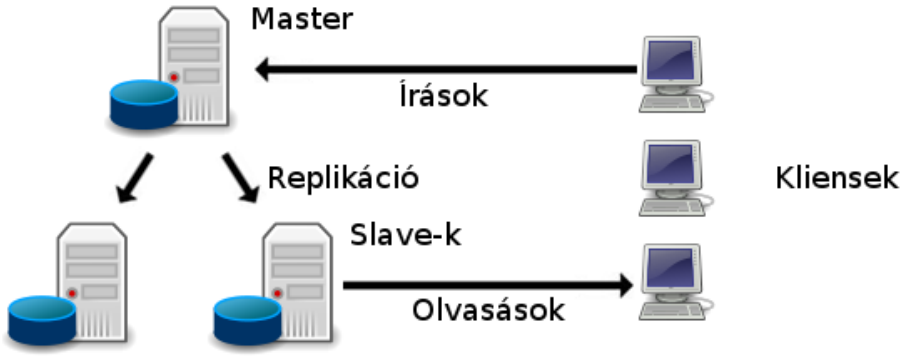
1.1 Javás webalkalmazások skálázhatósági kérdései

A tipikus Javás webalkalmazások architektúrája egy adatbázisból valamint egy alkalmazásszerverből áll, ehhez kapcsolódnak a kliensek HTTP protokollon keresztül. Ha ezt több gépre szeretnénk elosztani, akkor egyrészt az adatréteget, másrészt pedig a feldolgozási réteget kell elosztanunk külön-külön. Az előbbire többféle megoldás is létezik. Léteznek *NoSQL* adatbázisok, ezeket a legtöbb esetben arra tervezték, hogy a particionálást jól kezeljék, ilyen esetben tehát egyszerű dolgunk van. Ezekből vannak dokumentum alapú, gráf alapú, valamint kulcs-érték párokon alapuló megoldások. Hátrányuk az, hogy nem szabványosítottak, ezáltal az alkalmazások hordozhatósága nem megoldott.

Kép 1: 1-1 gépes Javás architektúra

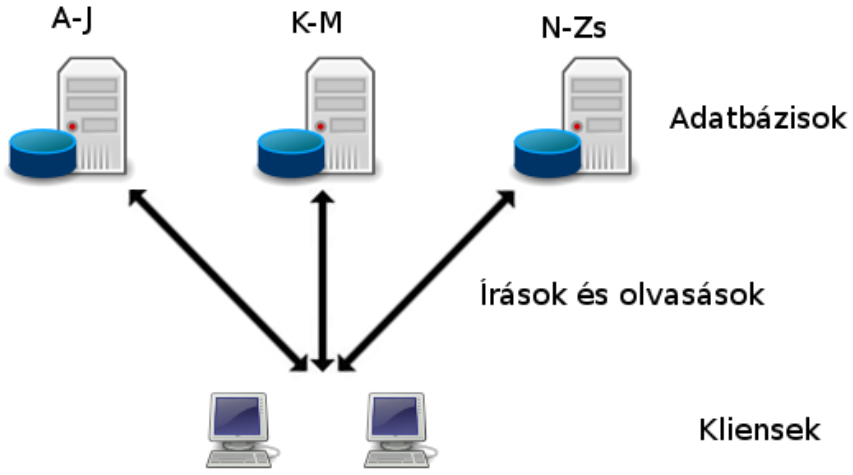
Mivel a létező és fejlesztés alatt álló alkalmazások túlnyomó többsége megköveteli az SQL alapú adatbázist, ezért szükséges foglalkozni ezzel a kérdéskörrel is. Ezek nem szervezhetőek triviálisan klaszterbe, de vannak azért kielégítő megoldások. Legegyszerűbb megvalósítás a *Master-Slave replikáció*, ennek során az adatmódosítási műveletek kizárólag a megkülönböztetett master gépre érkeznek, ez pedig propagálja ezeket a slave-knek, amikről pedig csak olvasni lehet. Ezzel a szűk keresztmetszet a master lesz, aminek elégnek kell kiszolgálnia az írási-, valamint azokhoz egyetlen olvasási műveletet. Mivel a webalkalmazások tipikusan olvasásintenzívek, ezért ezzel

drasztikusan lehet növelni az átbocsájtó képességet. Előnye továbbá, hogy az írásoknál nincs szükség több gép közötti együttműködésre, így megelőzhetőek az adatkorrupciós és szinkronizációs hibák. Ezen kívül az alkalmazáson rendszerint nem nehéz megvalósítani az írási és olvasási műveletek különbontását. Ha magas rendelkezésre állásra is fontos tervezni, akkor meg kell még oldani a master gép kiesését, és egy slave-et kell előléptetni automatikusan.



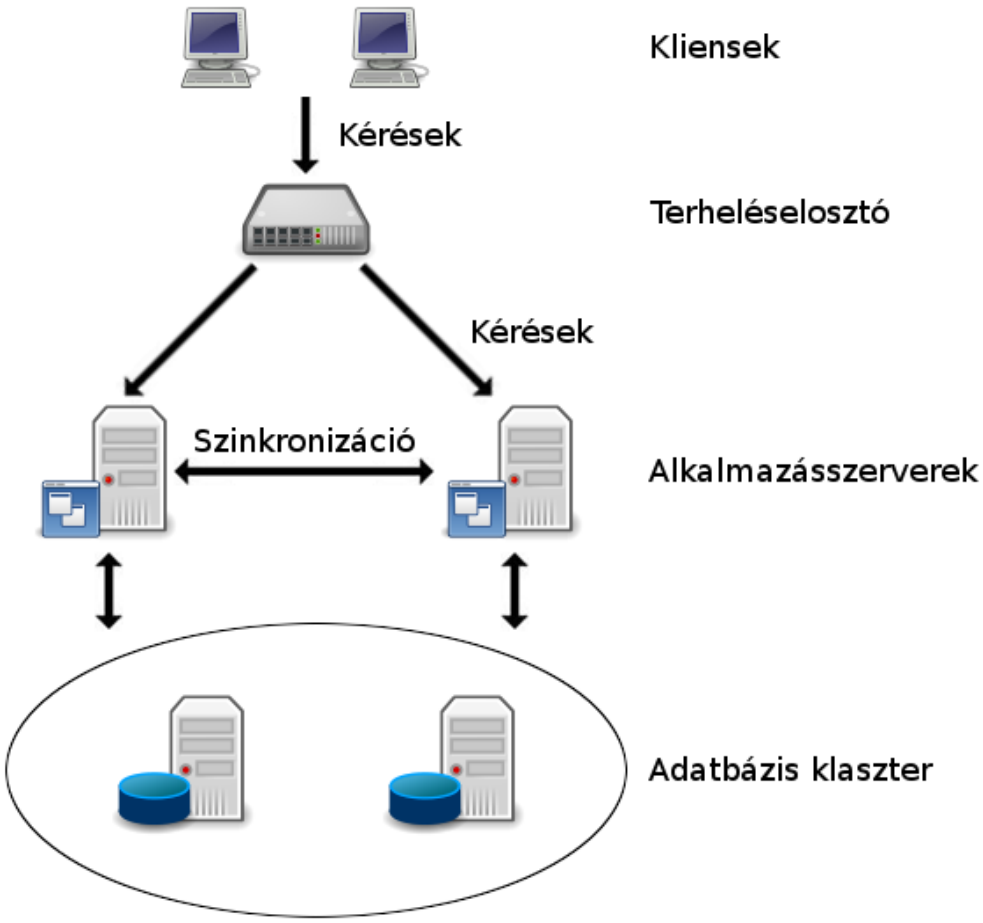
Kép 2: Master-Slave replikáció

Kép 3: Multi-master replikáció



Kép 4: Sharding

Megoldható az ún. *Multi-master replikáció*, ez hasonlít a Master-Slave megvalósításra, de több master gép is van egyidejűleg. Ennek előnye, hogy az írási szűk keresztmetszetet ki lehet kerülni, viszont a megvalósítás lényegesen bonyolultabb, és a gépek között szükséges a szinkronizáció. A gyakorlatban számítani kell adatinkonzisztenciára is. Harmadik megoldásként létezik a *sharding*[1], aminek során előre meghatározott attribútumok értékei mentén osztjuk szét az adatokat a gépek között. Például ha az adatok rendszerint felhasználókhoz tartoznak, akkor a felhasználói név szerint ha ABC rendben a halmaz első felében keresünk, akkor az A gépen tároljuk az adatot, egyébként a B-n. Amennyiben az adatok feloszthatóak ilyen részekre úgy, hogy a lekérdezések és módosítások az esetek túlnyomó többségében egyetlen részt érintenek csak, akkor ezzel a szervezéssel nagyon hatékonyan lehet skálázni. Hátránya, hogy az alkalmazást jelentősen módosítani kell ahhoz, hogy ezt a szervezést támogassa, valamint nem feltétlenül léteznek ilyen törésvonalak. Probléma lehet az is, ha nagyszámú olyan lekérdezést futtatunk, ami az összes gépet érinti, ugyanis akkor jelentősen fog lassulni a rendszer.



Kép 5: Általános elosztott Javás architektúra

Alkalmazásszerver szinten a skálázhatóság a munkamenetek elosztását, valamint a gyorsítótárak és egyéb állapotok szinkronizálását jelenti. Amikor már több kiszolgáló is üzemel, rendszerint szükséges egy terheléselosztót is üzembe helyezni, így a felhasználók csak azt látják, és az transzparensen delegálja a kéréseket. A munkamenetek skálázása vagy fürtözéssel történik, vagy tipikusan inkább ún. *session affinitással*. Ennek lényege, hogy egy konkrét klienstől érkező összes kérést mindig ugyan az a kiszolgáló kap meg, így nem szükséges replikálni. A gyorsítótárak és az állapotok szinkronizációjáról a következő fejezetekben részletesen kitérek.

1.1.1 Állapotmentesség

Legtöbb esetben kívánatos körülmény az, ha a webalkalmazás belső állapottól mentes, mivel ennek tudatában nem kell arra tervezni, hogy ezek az adatok is replikálódjanak a

klaszteren belül. Emiatt legtöbb esetben rossz tervezési döntés ennek a be nem tartása, és a gyakorlatban a gyorsítótárakon kívül nem tárolunk mást belső állapotként. Vannak megoldások ezeknek az adatoknak az elosztására is, ilyet kínál például a *Terracotta*[2] is, ugyanakkor ezek használatával pont azt a sebességbeli előnyt veszítjük el, ami miatt memóriában tárolnánk adatot, ugyanis ebben az esetben a módosításokat egy lassú hálózaton is át kell küldeni. Ezek miatt ilyen megoldás bevezetése csak olyan esetben lehet indokolt, amikor legacy alkalmazásokat kell felkészítenünk elosztott futásra, ugyanakkor nem számítunk annyira jelentős terhelésre, hogy ez szűk keresztmetszetet jelenthetne.

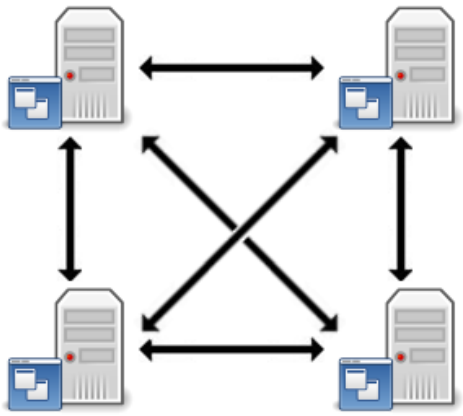
1.1.2 Munkamenetek replikációja

A munkamenetkezelés a felhasználók bejelentkezésében és nyomkövetésében fontos szerepet játszik, emiatt elosztott környezetben sem lehet megkerülni a replikációját. A mai terheléselosztók beállíthatóak úgy, hogy egy felhasználótól érkező kéréseket mindig egy adott kiszolgáló felé küldi, ezzel megkerülhető a munkamenetek másolása más alkalmazásszerverekre. Érdemes azonban végiggondolni, hogy mégis milyen lehetőségek lennének rá, ugyanis ezzel meg tudnánk oldani a skálázhatóság mellett a magas rendelkezésre állást is a munkamenetek tekintetében, és a legtöbb alkalmazásszerver ad rá beépített lehetőséget. A megvalósítás tekintetésben többféle megoldás is létezik.

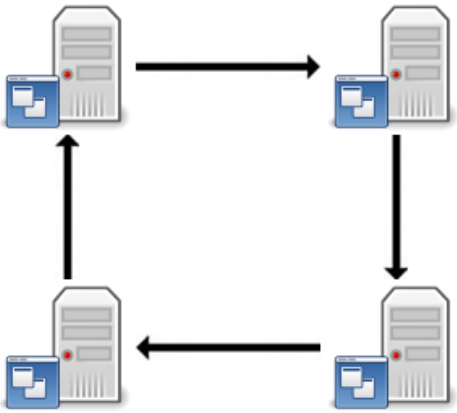
Egyrészt le lehet másolni az összes munkamenetet a klaszter összes többi tagjára, és a módosításokkor pedig frissíteni azokat. Nyilván ez nem működőképes nagyobb méretű fürtök esetén, valamint ha sűrű frissítések szükségesek, ugyanakkor a gyakorlatban egy pár gépes környezetben nem jelent szűk keresztmetszetet.

Másik lehetőség, hogy a kiszolgálókat egy virtuális gyűrűre fűzzük fel, és mindegyik az utána következőre másol, így minden adat két példányban lesz meg, így ha bármelyik kiesik, akkor a felhasználói adatok egy előre ismert gépen elérhetőek lesznek. Ez bonyolultabb megoldás, viszont a méret növelésével nem nő a terhelése. Ilyen elven működik például a Glassfish skálázhatósági megoldása. Ezeken kívül a munkameneteket egy elosztott P2P alapú tárolóba is lehet tenni, ahol az adatok

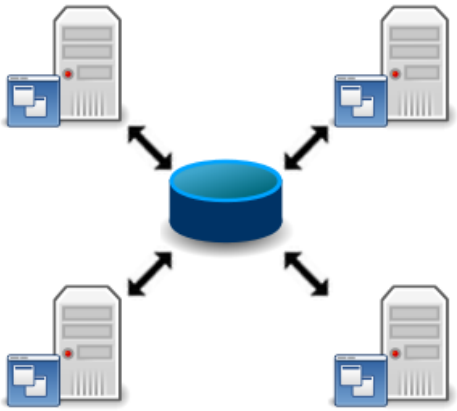
bárhonnan elérhetőek és automatikusan több példányban szétoszlanak, ezáltal biztosítják a magas rendelkezésre állást. Ez a legátlátszóbb megoldás, ugyanis egygépes környezetben is ugyanúgy kell használni mint klaszterben, hátránya viszont hogy lehetséges az, hogy az összes művelet egy távoli gépen történik. Mivel azonban a fürt tagjai általában nagy sebességű hálózaton érik el egymást, ezért a felhasználó felé már nem lesz észrevehető a különbség. Ezt a megoldást használja a Terracotta *Web Sessions* terméke.



Kép 6: Mindenhova másolódik a munkamenet



Kép 7: A következő szerverre másolódik a munkamenet



Kép 8: Közös tárba replikálódnak a munkamenetek

1.1.3 Gyorsítótárak skálázhatósága

Webalkalmazások klaszterben futtatásánál legtöbbször a legnehezebb feladat a gyorsítótárak skálázásának megvalósítása. Már egygépes környezetben, ha az adatbázis elválik az alkalmazásszervertől, akkor is szükséges az adatok rövid idejű memóriában tárolása a kívánt válaszidők eléréséhez. Ez elosztott környezetben hatványozottabban jelenik meg, ugyanis nem csak azt kell megoldani, hogy együtt tudjanak működni ezek a gyorsítótárak, emellett gyorsan elérhetőnek és sűrűn frissíthetőnek kell lenniük, valamint az alkalmazásban a lehető legkevesebb hibalehetőséget kell adniuk, mert az egyik legnehezebben felderíthető és kijavítható hibafajtát idézi elő a hibás működésük.

Első esetként tekintsünk egy olyan rendszert, ahol minden alkalmazásszervernek saját, lokális gyorsítótára van. Ilyenkor az elvárt funkcionalitáshoz elegendő csak a törlés műveletről értesíteni a klaszter többi tagját, ugyanis azok a következő hozzáféréskor az adatbázisból le tudják kérni a frissített adatot. Ennek természetesen a legegyszerűbb módja egy broadcast üzenet, amiben az adatot módosító rendszer tájékoztatja a többit a műveletről, amelyek ennek hatására törlik a saját gyorsítótárukból a referált adatot, majd később amikor szükségük lesz rá, újra betöltik. Ezt ki lehet egészíteni egy olyan megoldással, hogy a módosított adat új tartalmát is elküldi az első kiszolgáló az üzenetben, ezáltal nincs szükség minden módosítás után N-1 db (N a klaszterben részt

vevő gépek számát jelenti) olvasás műveletre is. Ennek hátránya, hogy lehet, hogy olyan adat is benne lesz a kiszolgálók memóriájába, amire nem is volt szükségük.

Másik megoldás egy központi, ugyanakkor belsőleg elosztott gyorsítótár használata, amelyhez minden kiszolgáló hozzáfér hálózaton keresztül. Ennek előnye, hogy mindig konzisztens marad, tehát sosem fordulhat elő olyan eset, hogy egyik kiszolgáló a régi, másik pedig új adatot látja. Hátránya viszont, hogy ebben az esetben is a hálózaton keresztül kell elérni az adatokat az operatív memória helyett, ami nagyságrendi lassulást von maga után. Ezt a kérdéskört érdemes egy kicsit közelebbről is szemügyre venni: ha figyelembe vesszük azt a tényt, hogy gyorsítótár helyett az adatbázisból kell kiolvasni az adatokat, ahhoz képest nyilván gyorsabb lesz ez a megoldás. Emiatt ha az alkalmazás még így is megfelelő idők alatt válaszol, akkor ez biztosan megfelelő megoldás lesz, és az adatbázist jelentősen tehermentesíti. Ez azonban összetettebb rendszereknél általában nem így van, mert azok építenek arra, hogy a belső memóriából gyorsan elővehetőek az adatok, és csak ezzel együtt éri el a válaszidő a kívánt szintet. Ilyen esetben viszont ez a nagyságrendi lassulás elfogadhatatlan lenne. A gyakorlat azt mutatja, hogy az olyan rendszereknél, ahol a gyorsítótárazás nélküli válaszidő az elfogadható szint fölött van, ott nem az adatok elérése, hanem azok feldolgozása tart hosszú ideig, emiatt részeredményeket is tárolnak. Ezek azonban ugyanúgy elmenthetőek mint a nyers adatok, így hiába hosszabb az elérési idő, a megspórolt feldolgozási teljesítményhez képest ez jelentősen kisebb, így a hálózaton keresztüli elérés nem fog jelentősen rontani a sebességen. Azonban ez egy olyan kérdéskör, amire érdemes figyelni alkalmazásfejlesztés közben.

Fejlesztési oldalról a gyorsítótárak leggyakoribb felhasználási területe a JPA másodszintű gyorsítótára, ami átlátszóan beépül az alkalmazás és az adatbázis közé. Ennek köszönhetően kevés programozói hibát okozhat, így a fejlesztést lényegesen nem akadályozza, valamint hasonlóan fog működni egy-, és többgépes környezetben is.

1.1.4 Elérhető számítási felhő megoldások

Számtalan cég kínál különböző felhő megoldásokat, de nagy, megbízható és globálisan elérhető szolgáltató jelenleg 3 van a piacon. Egyrészt a Microsoft az *Azure*[3]

platformmal, amely azonban .NET-re érhető el elsősorban, így ezzel a továbbiakban nem foglalkozok. Másodikként van a Google *AppEngine*[4], amely PaaS (Platform as a Service) szolgáltatást kínál. Ez Javás, viszont mivel csak teljes platformot kínál, ezért rendkívül kötött, így nem lehet alternatívája a hagyományos üzemeltetésű alkalmazásoknak, nincs is rá migrációs út. Végül az Amazon *EC2*[5] maradt, amely teljes értékű számítógépeket kínál rugalmas módon, amire bármi telepíthető és szabadon konfigurálható. Ezzel ki lehet váltani a hagyományos infrastruktúrát, létező alkalmazásokat is lehet rá migrálni. Ezek miatt ezt a platformot választottam a diplomamunkám alapjául.

1.2 Liferay portálrendszer

1.2.1 Áttekintő

A *Liferay*[6] a jelenleg elérhető Javás portálrendszerek közül az egyik legnagyobb és a legszélesebb körű szolgáltatásokat nyújtja. Nyílt fejlesztési modellje mellett stabil gazdasági vállalat áll mögötte, valamint egyre növekvő számú felhasználó. Ezekre tekintettel a jövője biztosnak tűnik, és továbbra is integrálni fogja az innovatív megoldásokat. Számos szabványt támogat, közöttük talán a legfontosabb, hogy a Javás portleteket tudja futtatni. Hátránya, hogy a futtatásának jelentős erőforrásigénye van, ami kis látogatottságú oldalak üzemeltetését jelentősen megdrágítja a konkurens megoldásokkal, leginkább a PHP (Personal Home Page, vagy újabban PHP: Hypertext Preprocessor) megoldásokhoz képest. Viszont képes több gépre is skálázódni[7], így a látogatottság növekedésével is lépést tud tartani.

Kép 9: Liferay logó

1.2.2 Portletfejlesztés Liferay alá

1.2.2.1 Portlet szabvány

A *portlet*[8][9] egy olyan webalkalmazás, ami a weboldalnak csak egy részét foglalja el, így egyszerre több is lehet egy oldalon. A portlet konténer oldja meg azt, hogy ezek az alkalmazások el legyenek különítve egymástól, valamint kezeli az életciklusukat. A portlet szabvány írja le azokat a megkötéseket és konvenciókat, amik szerint kialakítva egy Javás webalkalmazást, az képes portlet környezetben futni. Ennek a szemléletnek a legfontosabb része a szeparáció, a gyakorlatban azonban sűrűn van szükség együttműködési lehetőségekre, amikor is az szükséges, hogy webalkalmazások elérjék egymást, tipikusan üzenetküldéssel. A portlet szabvány erre is kínál megoldásokat. A weboldal széttagolására és így a részek szeparációjára a HTML is kínál lehetőséget az *IFrame* (Inline Frame) segítségével, azonban ezt nem célszerű használni, mivel sokkal nehezebben lehet az integrációt megoldani, valamint a háttérben elég nagy számú HTTP kérést generál egy ilyen oldal. Portleteket használva ezek a hátrányok kikerülhetőek.

Fontos technológiai fogalom ebben a témakörben a *portlet híd* (Portlet Bridge). Mivel a szabvány egy, a szervlethez csak hasonló struktúrát ír le, a létező webes keretrendszerek nem képesek ilyen környezetben működni. Erre jelentenek megoldást a hidak, amelyek ezeket a szükséges csatolásokat végzik el. A gyakorlatban ezek részleges integrációs megoldások, általában az így integrált keretrendszer nem lesz teljes funkcionalitású. Érdekesség, hogy akár nem csak Javás portleteket is lehet így fejleszteni: például PHP-sat is.

A portlet szabvány nincs Javához kötve, bármilyen JVM (Java Virtual Machine) felett futó programozási nyelven készülhetnek portletek.

Egy látványos példát mellékeltem az „A” Függelékben.

1.2.2.2 ServiceBuilder

A *ServiceBuilder*[10] a Liferay saját megoldása az adatbázisstruktúra generálására és az adatok elérésére. Tehát végeredményben az *ORM* (Object-Relational Mapping)-et váltja ki, valamint a *DAO* (Data Access Object) osztályokat generálja. Lényege, hogy egy

service.xml leíró fájlban a portlet fejlesztője meghatározza a perzisztens entitásokat és a struktúrájukat, ezután a ServiceBuilder ebből legenerálja a szükséges adat-, és elérési osztályokat. Végül a fejlesztő a generált osztályokba felveszi a szolgáltatásokat megvalósító metódusokat, így nem kell foglalkozni például az adatbázis konfigurációjával. A ServiceBuilder lehetőséget nyújt lokális és távoli interfészek generálására, így akár webszolgáltatáson keresztül is publikálható az üzleti logika, autorizációval kiegészítve. Ún. *Upgrade Process*-ek segítségével lehetőséget biztosít az adatbázis struktúrájának telepítési idejű megváltoztatására, ezzel is tovább redukálva a fejlesztési időt.

A megoldás nagy előnye az egyszerűség és az integráltság, mivel a Liferay beépített portletei is ezt használják. Ezen kívül a tipikus felhasználási esetek mellett pár speciálisat is kínál (pl. webszolgáltatáson keresztüli elérés). Ezeken kívül agresszív gyorsítótárazást is biztosít, valamint integrálódnak a platform-, valamint a saját fejlesztésű modulok tranzakciói.

Hátránya, hogy bizonyos sűrűbben használt funkciókat nem támogat, például az entitások közötti kapcsolatokat. Ezen kívül a többszintű gyorsítótárazás is sok esetben feleslegesen bonyolulttá teszi a rendszert. Használata előtt mindenképpen mérlegelni kell, hogy van-e olyan aspektus, ami miatt később terhes lesz a használata.

1.2.2.3 Platformszolgáltatások

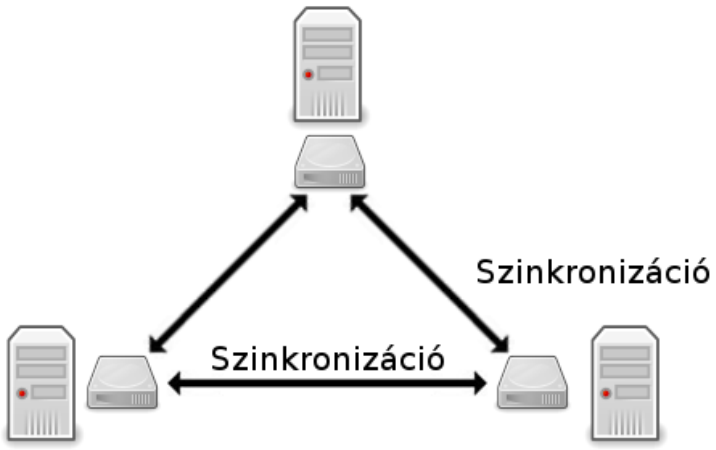
A Liferay, mint komplex és sok helyen használt rendszer, számos szolgáltatást nyújt, ami miatt megéri használni[11] Ezek közül talán a leggyakrabban használt a felhasználókezelés. Ennek során nem csak az éppen belépett felhasználót tudja azonosítani, hanem kész regisztrációs folyamatot nyújt. Az összes elvárható funkcionalitásra kész megoldást kínál, integrálódik *OpenID* szolgáltatókkal, lehetőséget ad elfelejtett jelszó visszaszerzésére, felhasználók deaktiválására, *CAPTCHA* (Completely Automatical Public Turing Test to Tell Computers and Humans Apart)-ra, valamint a fontosabb adatokat is kezeli. Ezen kívül megoldást nyújt jogosultságkezelésre is: itt a legtöbb helyen elterjedt *RBAC* (Role-Based Access Control)-t alkalmazza.

Lehetőséget kínál egy alkalmazáson belüli több weboldal kiszolgálására ún. *virtualhosting* segítségével, valamint közösségek létrehozása is lehetséges egy-egy oldalon belül, saját weboldalaikkal és felhasználóikkal. Az oldalak létrehozására és konfigurálására, valamint az oldalra portletek helyezésére és konfigurációjára felhasználóbarát felületet nyújt. Segédosztályokkal támogatja a HTTP kérésekből érkező paraméterek kezelését, a gyorsítótárak használatát, az ún. *szép URL*-ek létrehozását, rendelkezik központi keresővel, amihez akár saját portletek tartalma is integrálható. A fentiek csak néhány példát mutatnak arra, hogy számtalan ponton nyújt kiegészítő szolgáltatásokat a platform.

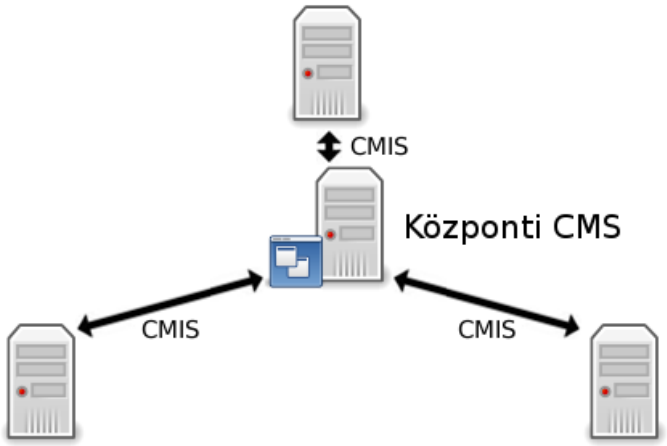
1.2.3 Speciális skálázhatósági kérdések

1.2.3.1 Dokumentum könyvtár skálázása

A Liferay egyik beépített szolgáltatása a *Dokumentum Könyvtár*, ami a JSR 170[12] (Content Repository for Java Technology API) és JSR 283[13] (Content Repository for Java Technology API Version 2.0) szabványokat megvalósító tárolót és ahhoz elérést biztosít. Ez egy praktikus szolgáltatása a portálrendszernek, ugyanis segítségével tetszőleges fájlt tudunk hierarchiában, akár integrált jogosultságkezeléssel támogatott formában tárolni. A skálázhatósággal a probléma, hogy alapértelmezésként az adatokat a fájlrendszeren tárolja, ami nehezen osztható el több gépre. Erre megoldás lehetne, hogy klaszteres fájlrendszerre költöztetjük a tárolót, ugyanakkor ez jelentősen bonyolítaná az architektúrát. Legegyszerűbb megoldás az, ha egy központi *CMS*-t állítunk fel, pl egy Apache Jackrabbit-et[14], majd a Liferay példányokat erre konfiguráljuk. Másik megoldás, ha adatbázisba migráljuk, aminek a skálázására amúgy is megoldást kell találni. Ezen kívül megoldható, hogy az Amazon S3-ban tárolja a fájlokat, erre kitérek a későbbi fejezetekben.



Kép 10: Skálázás elosztott fájlrendszeren



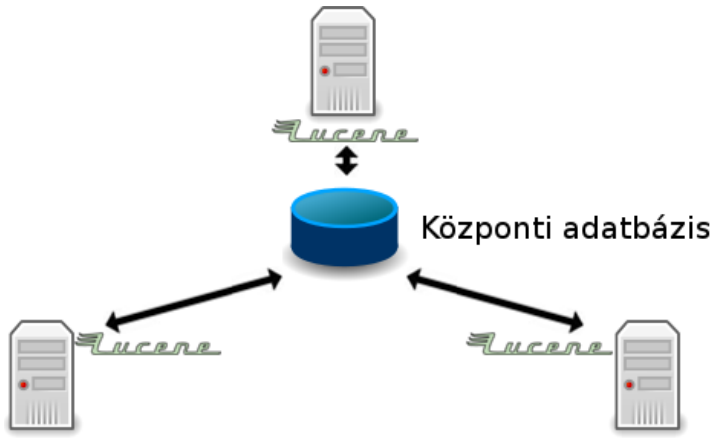
Kép 11: Skálázás központi CMS rendszer használatával, CMIS protokollon keresztül

Kép 12: Adatbázis alapú CMS használata

Kép 13: CMS adatok tárolása az Amazon S3-ban

1.2.3.2 Elosztott keresés

A Liferay által szállított keresőmegoldás, a *Lucene*[15] alapértelmezésként a fájlrendszeren tárolja az indexeket, ami nem kifejezetten kedvez az elosztott működésnek. Megoldható az is, hogy minden példány a saját kereső indexet tartja karban, és mindegyiket együtt frissítjük, ennél azonban sokkal jobb megoldás, ha a központi példányt áttesszük adatbázisba.



Kép 14: Lucene indexek tárolása a központi adatbázisban

1.2.3.3 Ütemező elosztása

A feladatok ütemezésére a Liferayben a *Quartz Scheduler*[16] kínál megoldást. Mivel a legtöbb feladat végrehajtása adatbázismódosítással jár együtt, ezért itt az elosztottság azért szükséges, hogy pontosan egyszer fusson le egy munka, valamint hogy ne fussanak egyszerre. Terracottá-val egyszerűen integrálható a legújabb verzió, sajnos a Liferay alapértelmezettként nem ezt szállítja. A Quartz képes az adatbázis alapú elosztott működésre, ezért az alapbeállításokkal futó portál ütemezője magától is skálázható.

1.2.3.4 Adatbázis gyorsítótár skálázása

A gyorsítótárak megfelelő skálázása kritikus egy klaszter helyes működéséhez. A Liferay alapértelmezésként *Ehcache*-t[17] használ, ez pedig szépen integrálódik a Terracottá-val[18], köszönhetően annak is, hogy egyetlen cég áll mögöttük. Ez az elosztást hatékony módon oldja meg.

1.3 Amazon Web Services

Kép 15: Amazon Web Services logó

1.3.1 Koncepció

Az Amazon EC2 (Elastic Computing Cloud) arra épül, hogy az üzleti felhasználók dinamikusan tudnak erőforrásokat lefoglalni amikor többletterhelést kell kiszolgálniuk, valamint elengedni, ha megszűnik ez az igény. Alapja az *instance* (példány), egy logikai számítógép, amihez operációs rendszer szinten biztosított a hozzáférés, tehát ez egy IaaS (Infrastructure as a Service) cloud szolgáltatás. Egy instance beállítási ideje pár perc, a felhő mérete pedig több ezer számítógép, így gyorsan tud reagálni a terhelésben bekövetkezett változásokra. Mivel operációs rendszer szinten konfigurálhatóak a számítógépek, ezért ez nem jelent különbséget a hagyományos szerverüzemeltetéshez képest. A felhasznált erőforrásokat számtalan tekintetben monitorozzák, és az igénybevétel arányában alakul ki a számlázott pénzösszeg. Tehát a hagyományos üzemeltetéssel szemben a nagy állandó költségeket sok apró változó költséggé alakítja.

1.3.2 Példányok és blokkeszközök

Az EC2-ben a példány (instance) egy számítógépet jelent, amihez a felhasználó teljes mértékben hozzáfér az interneten keresztül. Különböző méretekben vehetőek igénybe, mindegyik meghatározott CPU erőforrással és dedikált memóriával rendelkezik, az ára is ettől függ. Minden operációs rendszernek szüksége van blokkalapú tárolóra, ami rendszerint beépített merevlemez, itt azonban ez nem alkalmazható. Erre megoldás az *EBS* (Elastic Block Storage), ami hálózaton felcsatolt, blokkosan elérhető eszközt jelent. Minden instance-hoz kötelezően tartozik legalább egy, ahol az OS található, ezen kívül felcsatolható tetszőleges számú. Ezek a további eszközök az instance terminálása után is megmaradnak, és másikhoz csatolhatóak, így a rajta levő adatok perzisztensek lesznek.

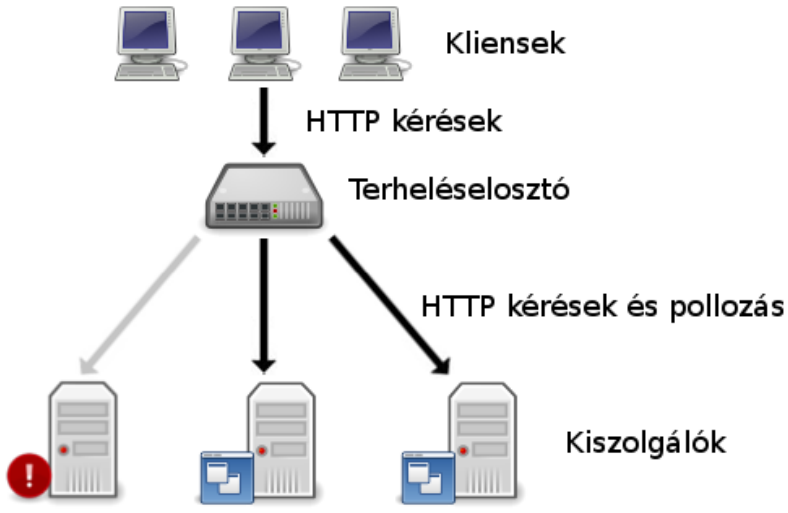
Az EBS eszközök ára a lefoglalt kapacitással arányos, és óránként kerül számlázásra. A tartósságuk a hagyományos merevlemezekéhez hasonló, de annál némileg magasabb.

1.3.3 S3

Az *S3* (Simple Storage Service) egy vödör alapú tároló, amibe fájlokat és egyéb adatokat lehet menteni valamint visszatölteni. Előnye, hogy nagymértékben elosztott, és bárhonnan hozzáférhető. Ide lehet snapshotokat készíteni az EBS eszközökről is. Elosztottsága miatt rendkívül magas adattartósságot kínál, ezáltal ideális biztonsági mentésekhez is, valamint hosszútávú adattárolásra.

1.3.4 Terheléselosztók és IP címek

A felhőben lehetőség van *virtuális terheléselosztók* üzembeállítására, amelyek a bejövő HTTP kéréseket a beállított kiszolgálók felé multiplexálja. Beállítható a session affinitás, tehát egy látogatótól érkező kéréseket mindig egy szerver felé küldi. Ezen kívül periodikusan tudja ellenőrizni a kiszolgálókat, így ha valamelyik meghibásodna, akkor ezt érzékeli, és oda nem küld több kérést.

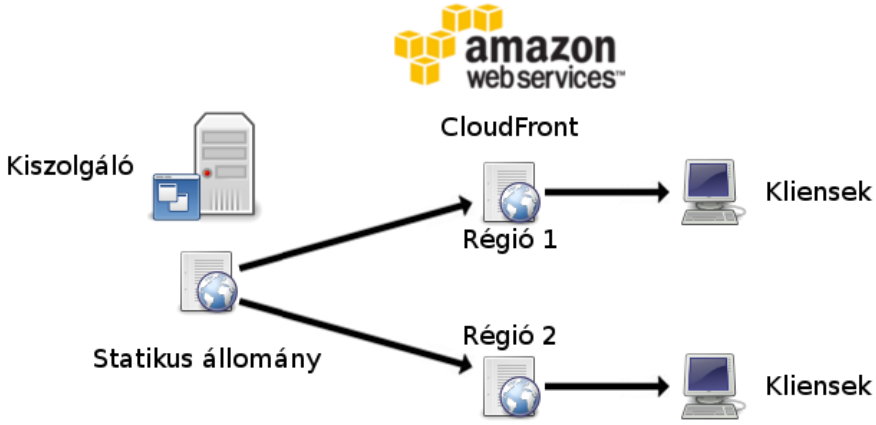


Kép 16: Terheléselosztás

Az instance-okhoz lehetséges fix IP címet igénybe venni, ami akkor lehet szükséges, ha leállítás és újraindítás után sem szeretnénk azt elveszíteni. Ez futó instance-hoz ingyenes, csak a parkoltatásért kell fizetni.

1.3.5 CloudFront

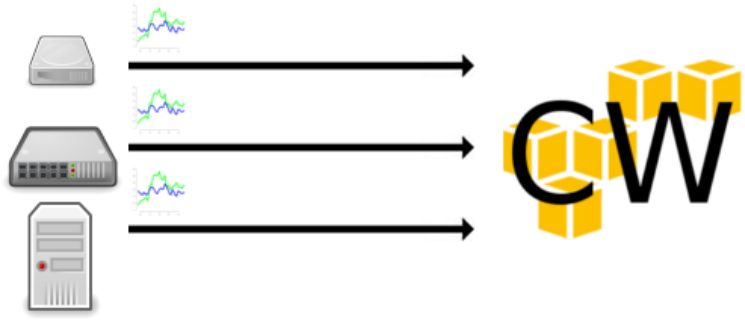
Webalkalmazások teljesítményét jelentősen lehet javítani azzal, ha a statikus fájlokat egy külön kiszolgálón is tároljuk, így az eredeti szerver valóban csak a dinamikusan generált adatokat fogja szolgáltatni. Ennek a neve a *CDN* (Content Delivery Network). Erre az Amazon megoldása a *CloudFront*, ami a beállított statikus fájlokat letölti, tükrözi a regionális kiszolgálókra, majd ide irányítja a forgalmat. Mivel a látogatóhoz közel történik a kiszolgálás, ezért használatával általában jobb válaszidőket lehet elérni mint egy saját megoldással.



Kép 17: Amazon CloudFront áttekintő

1.3.6 CloudWatch

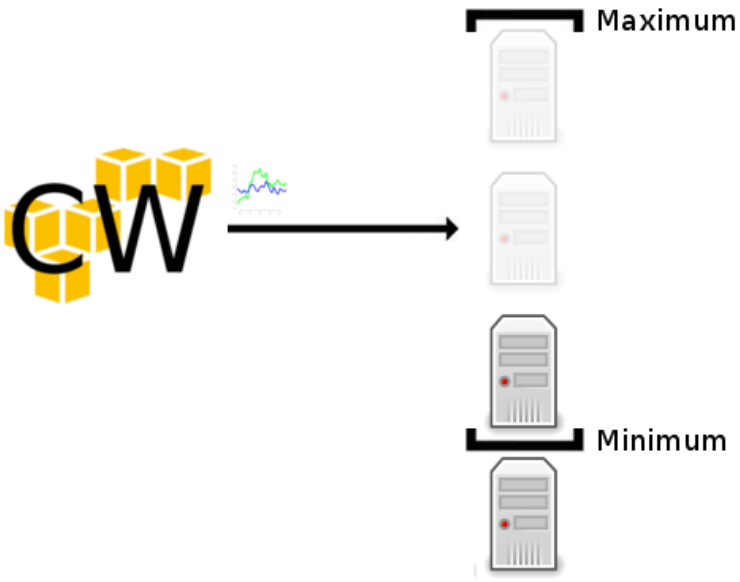
A *CloudWatch* az Amazon Web Services monitorozási megoldása, amivel figyelni lehet többek között az EC2-ben indított instance-okat, blokkeszközöket és terheléselosztókat. Alapvetően csak néhány metrikát figyel, viszont lehetőség van adatot feltölteni, mely utána historikusan is elérhető lesz. Ezzel végeredményben bármilyen működési adatot be tudunk kötni a CloudWatch-ba. A monitorozás egy fontos pontja, hogy beállíthatóak automatizált jelzések, amelyekkel incidensekről kaphatunk tájékoztatást: pl. leáll egy szolgáltatás, vagy hosszabb ideje nem érkezett adat. A felvitt metrikák természetesen le is tölthetőek.



Kép 18: Metrikák feltöltése a CloudWatch-ba

1.3.7 Autoscale

Lehetőség van egyszerű szabályok mentén a CloudWatch adatai alapján az instance-ek automatikus indítására és leállítására[19]. Ennek folyamán az *Autoscale* figyeli a gépek állapotát rendszeres keep alive jelzésekkel, valamint metrikák alapján indítási vagy leállítási döntéseket hoz. Nagy előnye, hogy nem kell a vezérléshez dedikált instance-et futtatni, mivel az is meghibásodhat, valamint egyszerűen konfigurálható. Hátránya viszont, hogy csak az instance-ek számát tudja kezelni, a típusait nem, tehát nem tudja az alkalmazást kisebb gépről nagyobbra költöztetni.



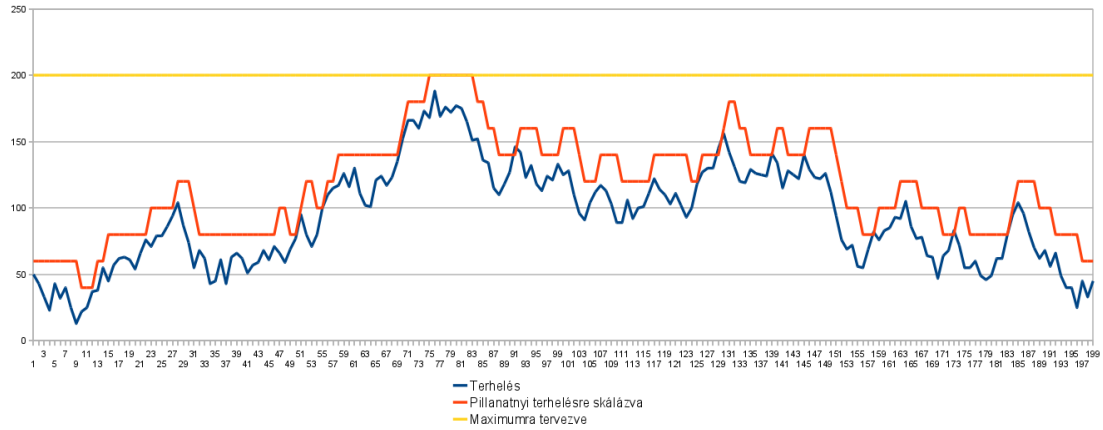
Kép 19: Autoscale működése CloudWatch metrikák alapján

1.3.8 Állandó és változó költségek

A felhőben futó alkalmazások vizsgálata során fontos megvizsgálni a költségszerkezetet. Mivel itt nem egy fix havidíjas szolgáltatás van fix korlátokkal, hanem egy állandó költség nélküli konstrukció a legtöbb irányban korlátok nélkül, ezért az árazás sem annyira egyértelmű. Elsősorban az instance-ok futásáért kell fizetni, ami arányos a lefoglalt erőforrások (CPU és memória) méretével. Ezen kívül fontos tétel a tárolt adatok ára. Az EBS esetében a lefoglalt tárkapacitásért, S3-nál pedig a ténylegesen eltárolt adatmennyiségért kell fizetni. Ez utóbbinál igénybe lehet venni a csökkentett tartósságú tárolót, ami kevésbé elosztott tárolásért cserébe alacsonyabb költségeket jelent. Harmadik fontos tétel a hálózaton átvitt adatmennyiség, ami tekintve, hogy a rendszer kizárólag az interneten keresztül érhető el, számottevő. Itt a konkrét adatmennyiség után kell fizetni.

1.3.9 Javás alkalmazások skálázása a felhőben

Tekintettel arra, hogy az instance-ek nagyban megegyeznek a fizikai számítógépekkel, a futtatott alkalmazások egyszerűen átköltöztethetőek, nincs olyan technológiai akadály, ami miatt jelentősen módosítani kellene őket. Mivel egyszerűen indítható és leállítható egy instance, ezért az alkalmazásfejlesztésnél érdemes ezt figyelembe venni, így könnyen bővíthető/zsugorítható lesz a kiszolgáló-kapacitás. Ezzel megoldható olyan működés, ami mindig a tényleges pillanatnyi terhelést szolgálja ki, így a túlterhelődés elkerülése és a költséghatékony működés egyaránt megvalósítható.



Kép 20: Szükséges kapacitás a terhelés arányában a maximumra, illetve dinamikusan skálázva

2 A megoldott feladat

2.1 A feladat leírása

A feladat első része a Liferay portálrendszert Amazon EC2 környezetben futtatni, olyan módon, hogy automatikusan újabb erőforrásokkal lehessen bővíteni, valamint ezeket az erőforrásokat kivenni a fürtből. A második rész egy olyan szoftverkomponenst fejleszteni, amely a Liferay-es fürtöt a terhelés függvényében tudja bővíteni, illetve szűkíteni. Az előzőek felhasználásával egy olyan eszközt kell készíteni, amelyik hatékonyan képes segíteni a változó terhelésű weboldalak üzemeltetését.

2.1.1 Kitűzött célok

A feladat megfogalmazásából is látszik, hogy a legfőbb cél, hogy a szoftver az üzemeltetést hatékonyan tudja segíteni, ezáltal a kezdeti beállítást és időnkénti finomhangolásokat leszámítva, amennyiben nem történik váratlan esemény, automatikusan le tudja kezelni a napon belüli terhelésingadozásokat. Célja a programnak, hogy kellő mértékben konfigurálható legyen, tehát elvárható a használat keretein belüli testreszabhatóság. A programnak naplóznia kell minden beavatkozást, hogy támogassa a hatékony hibakezelést. Ezen kívül a választott megoldásoknál törekedtem arra, hogy lehetőség szerint a felhasznált szoftverek későbbi verzióival is kompatibilis maradjon.

Nem cél ugyanakkor, hogy a terhelés elosztása mellett magas rendelkezésre állást biztosítson. Természetesen az egyszerű megoldásokat a diplomamunkámban bemutatom. Nem cél továbbá, hogy a végtelenségig, extrém méretekig lehessen skálázni a fürtöt. Ökölszabályként mindig a magyar viszonylatokból indulok ki, egy, a magyar piacon üzemelő magasabb terhelésű oldalt tekintek alapnak.

2.2 Felhasznált szoftverek

Diplomamunkám a Liferay 6.1-re épül. Az elosztott működés Terracotta-n alapul, annak az aktuális legfrissebb verzióján, a 3.6.2-esen. Ehhez kapcsolódóan az aktuális nightly

verzióból teszek hozzá Tomcat7-es csatolót, mert a stabil verzió azt még nem támogatja. Operációs rendszerként az Ubuntu 11.04 32bites verziója, adatbázisnak pedig a tárolóban fentlevő MySql-t használok.

2.3 Tervezés

2.3.1 Koncepció

Az alapkoncepció az, hogy mindent központosítok, amit csak lehet és nem szűk keresztmetszet. Ebből kiindulva egy központi adatbázisra építkezek, és bár ezen a szinten is léteznek fürtözésére bevált megoldások (lásd Bevezető fejezetek), nem foglalkozok velük, hiszen ez túlmutatna a diplomamunkám témáján. Ezenkívül a Dokumentum Könyvtár, valamint a keresőindex elosztott működésével sem foglalkoztam. Az előbbi képes az Amazon S3-ba dolgozni, ez a legjobban elosztott megoldás. A keresőindexet pedig legegyszerűbb adatbázisban tárolni. Ezenkívül az adatbázisba kerül még az ütemező is: a Quartz képes magát elosztani JDBC-n át.

Az elosztott gyorsítótárat Terracottával biztosítottam, aminél a koordinációs szerverként egy központi elemet használtam. Léteznek rá megoldások, amivel tükörcsoportokat lehet kialakítani, így ennek a rétegnek a bővítése is megoldható.

A munkamenetek elosztása szintén Terracottán keresztül történik.

2.3.2 A magas rendelkezésre állás lehetőségei

Kitekintésként megvizsgálom az elméleti lehetőségét annak, hogyan lehetne a terheléselosztási fürtnél a magas rendelkezésreállást megvalósítani. Ehhez először azonosítani kell azokat a pontokat, amelyek meghibásodásakor szolgáltatáskimaradás léphet fel.

A legtriviálisabb ezek közül az adatbázis. Ennek a megoldására sok ipari megoldás létezik, vannak fürtözött adatbázisok, amelyek kiesés esetén is tovább működnek, de házilag is készíthetünk ilyen megoldást. Ehhez először is replikálni kell az adatokat egy másik kiszolgálóra, tipikusan aszinkron módon, mivel az elosztott gyorsítótár miatt

nincs szükségünk a Master-Slave replikációra. Ezután egy hearthbeat-tel figyelnünk kell az aktív adatbázist, és ha meghibásodást észlelünk, akkor a háttérrendszerre átállni.

Másik pont a Terracotta kiszolgáló meghibásodása. Ha ez megtörténik, akkor szétesik az elosztott fürt, és új elemeket sem lehet bevonni. Erre létezik olyan megoldás, hogy ún. tükörcsoportokat hozunk létre, amelyek képesek átvenni a klienseket kiesés esetén, mert folyamatosan szinkronizálódnak az adatok.

Frontend oldalon a kiszolgálók kiesése nem okoz gondot, egészen addig, ameddig még szerepel élő instance, mivel a terheléselosztás csak működő kiszolgálókra irányít forgalmat. Ez megkerülhető olyan módon, hogy mindig legalább két kiszolgálót futtatunk.

A terheléselosztó kiesése esetén elérhetetlen lesz a fürt, ugyanakkor ez viszonylag ritka. Az S3-ba mentett adatok rendkívül biztosak, ahova a Dokumentum Könyvtár ment, az Amazon 99.999999999%-os tartósságot és 99.99%-os elérhetőséget ígér.

2.3.3 A portlet tervezése

A portlet feladata egy egyszerű felület biztosítása konfiguráció, illetve a naplóbejegyzések megjelenítése céljából. Ez utóbbi egy érdekes problémát is felvet. A Liferay összes komponense alapértelmezésben a Tomcat-es beépített naplózást használja, ami fájlba ment. Mivel a gép leállításakor ez elveszik, ezért az alkalmazás naplóit nem érdemes ide tenni. Az adatbázis azonban perzisztens, ezért itt célszerű a bejegyzéseket tárolni.

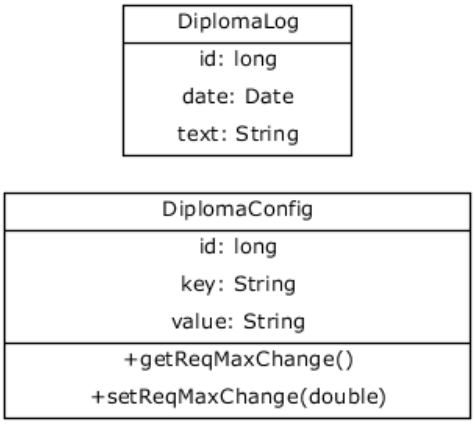
Ehhez az adatbázisban két táblára van szükség:

1. Tárolni kell a naplóbejegyzéseket: itt fontos az időpont és a szöveg
2. Tárolni kell a konfigurációkat: ehhez egy kulcs-érték párokat tároló adattáblára van szükség.

Érdemes megfigyelni, hogy a Liferay esetében elvileg nem lenne szükségünk konkrétan táblákat létrehoznunk az adatbázisban, mivel az *Expando* modul segítségével ezt

dinamikusan is meg tudnánk tenni. A rugalmasságért azonban az áttekinthetőség és a performancia romlásával kell fizetnünk: ezt csak apró kiegészítések esetén érdemes használni.

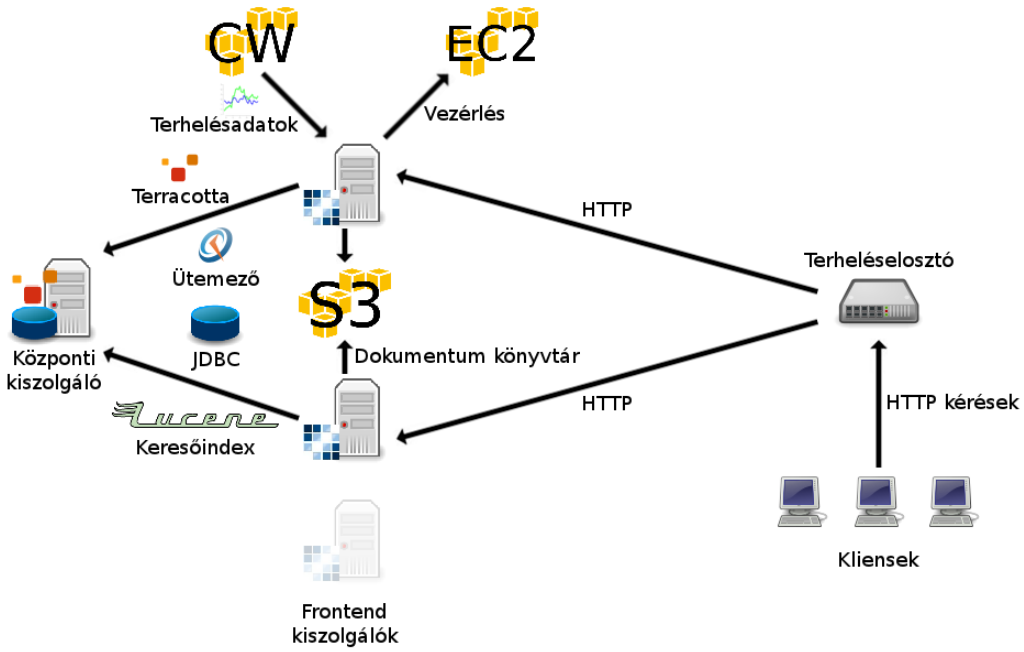
A táblák és a létrehozó SQL-ek generálását a ServiceBuilder segítségével végeztem el.



Kép 21: A szolgáltatásréteg leírása

2.3.4 Architektúra

Az alkalmazás központosított részei egyetlen gépre kerülnek, ezen fut az adatbázis és a Terracotta Server is. A frontend kiszolgálók ehhez kapcsolódnak, ez szolgálja ki az adatokat, a keresőindexet és az ütemezőt is. A Dokumentum Könytár az Amazon S3-hoz kapcsolódik. A frontendek egy terheléselosztón keresztül érhetőek el, az instance-ek egyike pedig az Amazon CloudWatch-al kommunikálva kéri le a terhelésadatokat. A beavatkozáshoz az Amazon EC2-höz fordulnak, így indíthatóak és leállíthatóak a gépek.

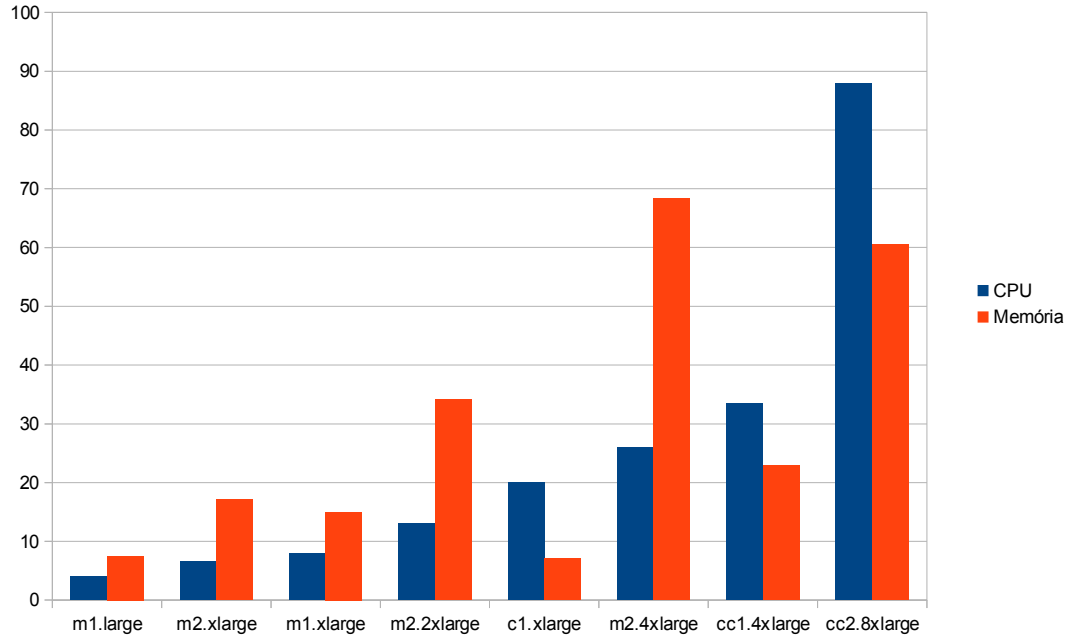


Kép 22: Architektúra

2.3.4.1 A skálázó algoritmus

2.3.4.1.1 Miért nem megfelelő az Autoscale?

Az Amazon EC2-ben beépített megoldása az Autoscale, mely terhelés vagy egyéb metrikák alapján képes a kiszolgálók darabszámát szabályozni. Felmerül a természetes kérdés, hogy miért nem megfelelő ez, miért kell egy újabb megoldást fejleszteni? Az Autoscale legnagyobb hibája, hogy csak az instance-ek darabszámát tudja beállítani, a típusait nem, márpedig nagyságrendi különbség van a legkisebb és a legnagyobb kapacitású gépek között mind teljesítményben, mind árban[20], ezt szemlélteti az alábbi grafikon is. Tekintettel arra, hogy az elvárt kiszolgálási kapacitás intervalluma a völgyidőszakban tapasztalható minimális terheléstől mindössze a legnagyobb méretű kiszolgálóból 2-4 db-ig terjed, ezért a típusokon való fellépcsőzés fontosabb, mint az egyes kiszolgálók darabszáma.



Kép 23: A 64 bites instance típusok, CPU szerint rendezve

2.3.4.1.2 A megoldandó probléma

A probléma leegyszerűsítve annyiból áll, hogy különböző erőforrásokkal, óradíjjal rendelkező instance típusokból kiindulva kell megtalálnunk a legolcsóbb konfigurációt, amely optimálisan képes kiszolgálni a folyamatosan változó terhelést. Néhány dolgot azonban figyelembe kell vennünk ahhoz, hogy használható megoldást kapjunk. Egyrészt nem minden instance rendelkezik elegendő erőforrással a portál működtetéséhez (hiszen korlátos mennyiségű memóriával rendelkeznek). Ezenkívül a legkisebb méretű típus, a micro, huzamosabb CPU igénybevétel esetén erősen lekorlátozódik, ami a gyakorlati használatban megbízhatatlanná teszi. Valamint figyelembe kell vennünk azt is, hogy a kiszolgálók elindítása is időbe kerül, és ekkor sem csökkenhet le a teljesítmény az elvárt szint alá. Végül pedig egy kritikus szolgáltatás erőforrásainak méretezésénél, persze az ügyfelek igényeit figyelembe véve, mindig érdemes valamilyen szintű többletben gondolkodni .

Tehát első lépésben meg kell határoznunk azokat az instance típusokat, amelyeken megbízhatóan fut a portálrendszer. Mivel általában rendelkezésünkre áll olyan mérés

vagy becslés, amely megadja, hogy mennyi memóriára van szüksége, ezért egyszerűen csak azokat a típusokat vesszük figyelembe, amelyek ennél több erőforrással rendelkeznek, micro típusút pedig sohasem.

Ezután ki kell számolnunk, hogy milyen konfiguráció tudja kiszolgálni az igényeket. Ennek a kiválasztásáról a következő fejezetekben írok részletesen. Miután megvannak számszerűen, hogy milyen instance-ekből hány darabra van szükség, akkor elindítjuk azokat, amelyek nem futnak jelenleg. Ezzel párhuzamosan megvizsgáljuk, hogy azok közül, amelyeket le lehet állítani, melyek azok, amelyek nélkül még az elvárt szint fölött marad a kapacitás. Ez egy újabb optimalizációs feladat. Ennek folyományaként a gyakorlatban lényeges többlet keletkezik, amely a következő ütemezésig meg is marad.

Az ütemezés gyakoriságának meghatározásához szükségünk van arra, hogy mennyi idő alatt képes elindulni a portálrendszer, és biztonsággal e fölé állítani.

2.3.4.1.3 A szükséges kapacitást számoló optimalizációs algoritmus

Az optimalizációs algoritmusnak tehát rendelkezésére áll az, hogy melyik instance típus mennyi kérést tud kiszolgálni, mennyibe kerül, valamint a pillanatnyi terhelés. Ebből kell visszaadnia, hogy melyik típusból mennyi gépnek kell futnia, hogy az összkapacitás a terhelés fölött legyen, az ára pedig minimális. Megkötés még, hogy minden instance-t vagy elindítunk, vagy nem, emiatt az optimalizálás eredményének mindenképpen egésznek kell lennie. Ez jól formalizálható *Egészértékű Programozási*[21] (IP) feladatnak, amelyre léteznek megoldó algoritmusok.

Az IP feladat lényege, hogy egy egyenlőtlenségrendszert írunk fel olyan módon, hogy bal oldalon a változók egy lineáris kombinációja áll, jobb oldalon pedig egy konstans. A bal oldal együtthatóiból képzett mátrix lesz az A mátrix, a jobb oldal konstansai pedig a $\underline{b}$ vektor. Ezen kívül meghatározunk még egy célfüggvényt, ez is a változók egy lineáris kombinációja lesz, ennek az együtthatói alkotják a $\underline{c}^T$ vektort. A feladat megoldása teljesíti az összes egyenlőtlenséget úgy, hogy a célfüggvény minimális lesz.

Amennyiben a terhelést *req*-val, az instance típusok árait $cost_0$, $cost_1$, …, $cost_n$-el, a kapacitásukat cap_0, cap_1, …, cap_n-el, az algoritmus eredményét pedig x_0, x_1, …, x_n-el jelöljük, akkor a probléma felírása az alábbi:

$$cap_0 * x_0 + cap_1 * x_2 + \ldots + cap_n * x_n \geq req$$
$$\min\{cost_0 * x_0 + cost_1 * x_1 + \ldots + cost_n * x_n\}$$

Tehát az algoritmus bemenete az alábbi (a nagyobb egyenlő miatt negáljuk mindkét oldalt):

$$A = \begin{bmatrix} -cap_0 & -cap_1 & -cap_2 & \cdots & -cap_n \end{bmatrix}$$

$$\underline{b} = \begin{bmatrix} -req \end{bmatrix}$$

$$\underline{c}^T = \begin{bmatrix} cost_0, cost_1, \cdots, cost_n \end{bmatrix}$$

Ezt egy IP megoldóba betöltve kiadja az x_0, x_1, …, x_n értékeket, amelyek pedig az optimális megoldást fogják jelenteni.

2.3.4.1.4 A leállítható gépeket számoló optimalizációs algoritmus

Az előző algoritmushoz hasonlóan, itt is rendelkezésre állnak az instance típusokról az adatok, valamint a szükséges kapacitás, ezen kívül pedig az, hogy éppen most milyen konfiguráció fut, valamint az előző optimalizáció eredménye. A feladat az, hogy megtaláljuk, hogy melyik leállítandó gépek leállításával maradunk még az elvárt szint fölött, ugyanakkor minimalizáljuk a megmaradó gépek árát. Az előző fejezetben bevezetett jelölésrendszerhez vezessük be az aktuálisan futó gépek számát, ezt jelöljük $curr_0$, $curr_1$, … , $curr_n$-nel, valamint az előző optimalizáció eredményét, ezt pedig des_0, des_1, …, des_n-el. A gépek minimális száma a jelenleg futó és az elvárt mennyiség közül a kisebbik lesz, ezzel biztosítjuk, hogy mindig legyen legalább annyi, amennyi az előző optimalizáció szerint szükséges.

Így a probléma felírása az alábbi lesz:

$cap_0 * x_0 + cap_1 * x_2 + \ldots + cap_n * x_n \geq req$

$x_0 \leq curr_0$

$x_1 \leq curr_1$

$\ldots$

$x_n \leq curr_n$

$x_0 \geq \min(des_0, curr_0)$

$x_1 \geq \min(des_1, curr_1)$

$\ldots$

$x_n \geq \min(des_n, curr_n)$

$\min\{cost_0 * x_0 + cost_1 * x_1 + \ldots + cost_n * x_n\}$

Az algoritmus bemenete pedig az alábbi lesz:

$$A = \begin{bmatrix} -cap_0 & -cap_1 & -cap_2 & \cdots & -cap_n \\ 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & & & & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ -1 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 0 & \cdots & 0 \\ \vdots & & & & \vdots \\ 0 & 0 & 0 & \cdots & -1 \end{bmatrix}$$

$$\underline{b} = \begin{bmatrix} -req \\ curr_0 \\ curr_1 \\ \vdots \\ curr_n \\ -min(des_0, curr_0) \\ -min(des_1, curr_1) \\ \vdots \\ -min(des_n, curr_n) \end{bmatrix}$$

$$\underline{c}^T = \begin{bmatrix} cost_0, cost_1, \cdots, cost_n \end{bmatrix}$$

Az IP megoldóba betöltve az x_0, x_1, …, x_n azokat a gépeket fogja tartalmazni, amelyeket meg kell tartani.

2.4 Megvalósítás

2.4.1 Az alkalmazás felépítése

Az elkészített alkalmazás egy Liferay plugin, amely ütemezetten, előre beállított intervallum szerint fut. Ennek a konfigurációját a Liferay-specifikus portlet leíróba kellett helyezni, a helye *WEB-INF/liferay-portlet.xml*. Ehhez létrehoztam egy *MessageListener*-t, amelyet ütemezni lehet, ez meghíváskor példányosítja az AWS elérését lehetővé tevő osztályt a konfigurációval, majd elvégzi az optimalizációt.

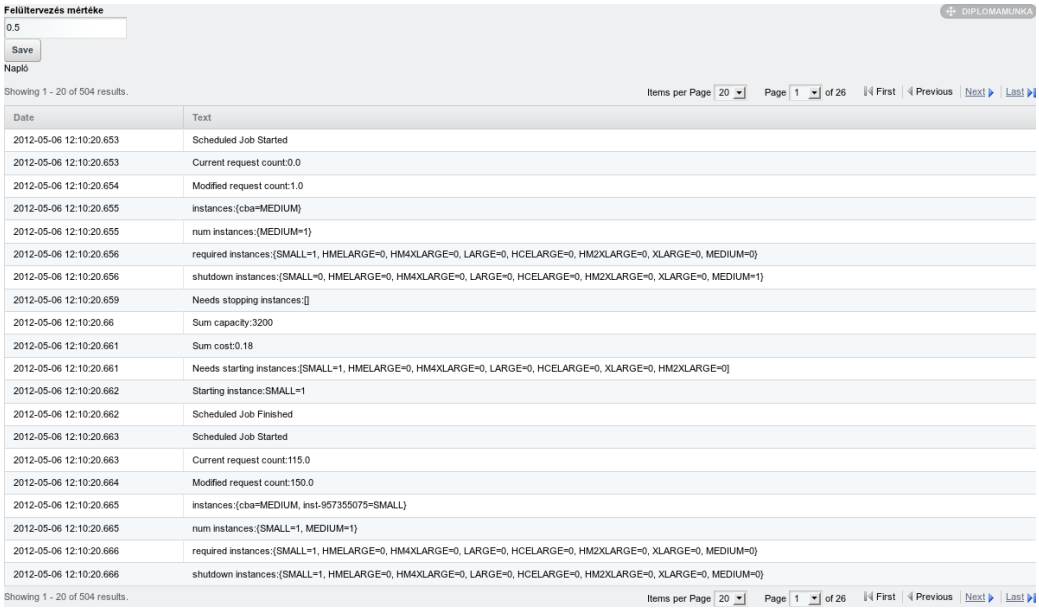
A fő futási ciklusban először lekéri az AWS-ről az aktuális terhelést, majd a futó instance-k típusait. Ezután le tudja futtatni az optimalizáció első és második fázisát, ahogy az előző fejezetben részletesen ismertettem. Ezek során kiszámolja az ideális konfigurációt, valamint hogy a jelenlegiek közül mely gépek állíthatóak le. Végül már nem marad más, csak érvényesíteni kell a változtatásokat, tehát el kell indítani és le kell állítani instance-eket.

2.4.2 A portlet felépítése

A szolgáltatásrétegnek két feladata van. Egyrészt kezelnie kell a dinamikus konfigurációt, másrészt a naplóbejegyzéseket tudja tárolni. Az alkalmazásban az egyetlen dinamikusan konfigurálható pont a felültervezés mértéke, erre elkészítettem a getter/setter-t, amely az adatbázisból dolgozik. A naplózáshoz a bejegyzések elmentése, valamint megjelenítése triviális.

Megjelenítési oldalról egy JSP oldalt írtam, amely egyrészt megjelenít egy form-ot, aminek a segítségével a konfiguráció megtekinthető illetve módosítható. Másrészt egy lapozható listázót, amely táblázatban sorolja fel a naplóbejegyzéseket.

Technikai oldalról az *Alloy UI*[22]-t használtam a form-hoz, a Liferayes *SearchContainer*-t[23], amely szintén Alloy alapú, pedig a táblázat megjelenítéséhez.



Kép 24: A portlet felülete

2.4.3 Műveletek az AWS felé

A felépítés során figyelembe vettem, hogy nem feltétlenül csak ez az egy alkalmazás fut az AWS profilban, ezért úgy kellett felépítenem a műveleteket, hogy ne zavarja a többi futó instance-t. Alapvetően négy műveletet kellett megoldanom, hogy az alkalmazásom futni tudjon.

Egyik legfontosabb művelet az aktuális terhelés lekérése. Ezt úgy oldottam meg, hogy a CloudWatch-ból lekérem a terheléselosztóra a RequestCount metrika SUM-ját az elmúlt percre, ez tartalmazza az a perc alatti terhelés összegét. Ez a metrika mindig létezik, a terheléselosztó magától tölti, ezért ez biztonsággal felhasználható.

Ennél trükkösebb probléma az aktuálisan futó instance-k lekérése. Ennél figyelembe kell venni, hogy csak a terheléselosztóba regisztrált gépek szerepeljenek az eredménybe. Első lépésként tehát lekérem az EC2-től a beregisztrált instance-kat. Ez azonban csak az ID-ket adja vissza, ezért az ezekhez tartozó instance-kat még le kell kérni egy külön kérésben. Ez azonban csak az ún. *Reservation*-öket adja vissza a gépekhez. A Reservation az együtt indított gépeket fogja össze, hogy egyszerűbb legyen kikeresni,

hogy ki, mikor és mit indított el. Innen már egyszerűen csak meg kell nézni, hogy a visszakapott Reservation-ökhöz tartozó instance-ek közül melyek szerepelnek a terheléselosztóban, és ez lesz a művelet eredménye. Még annyi átalakítást célszerű végezni, hogy itt id – típus formában tudjuk visszakapni, de a későbbiekben típus – szám formában használjuk. Ez egy egyszerű összegzéssel elérhető.

Végül szükségünk van két olyan műveletre, amivel elindítható és leállíthatóak lesznek instance-k. Ezekre van egyszerű művelet, csak annyira kell pluszban figyelni, hogy a terheléselosztóba regisztrálni, valamint onnan törölni is kell őket.

2.4.4 AWS konfiguráció

Ahhoz, hogy az alkalmazás futni tudjon, és hogy be tudjuk konfigurálni, számos dolgot létre kell hoznunk a felhőben. Egyrészt szükségünk van egy koordinációs instance-re. Ezt először is el kell indítani, valamint célszerű egy fix IP címet rendelni hozzá (Elastic IP Address). Szükségünk van egy *Security Group*-ra, ami a tűzfalszabályokat adja meg: ezt célszerű olyanra konfigurálni, hogy a gépek elérjék egymást, valamint a külvilág felé HTTP-n és SSH-n keresztül elérhetőek legyenek (8080,22-es portok). Ezen kívül szükségünk van egy publikus-privát kulcspárra, hogy az instance-ekbe ténylegesen is be tudjunk lépni. Létre kell hoznunk egy terheléselosztót, amely majd a közös belépési pontja lesz az alkalmazásnak. Ezután pedig az *IAM*-ben (Identity and Access Management) létre kell hoznunk egy olyan felhasználót vagy kulcsot, amellyel az alkalmazásunk eléri az AWS API-t.

2.4.5 Konfigurációs lehetőségek az alkalmazásban

Az AWS felé számos konfigurációs paraméterre szükségünk van. Először is meg kell adni, hogy melyik régiót akarjuk használni. Az Amazon 8 ilyet nyújt, én az EU-WEST-et választottam, mert ez van a legközelebb. Meg kell adnunk a terheléselosztó nevét, a Security Group-ot, valamint a kulcspár nevét (ezeket az előző fejezetben hoztuk létre).

Mindezeken kívül szükségünk lesz majd egy *AMI* (Amazon Machine Image) id-re: ez adja meg, hogy az elindított gépen milyen állapot (OS, telepített szoftverek) legyenek. Itt azonban belefutunk a tyúk-tojás problémába. Az elindított gépre fel kell telepítenünk

a bekonfigurált Liferay-t, valamint a fel kell tennünk az alkalmazást is. Az alkalmazás konfigurációjához tudnunk kell az AMI id-t, amit viszont csak akkor tudunk legenerálni, ha a végleges kép összeállt. Többféle megoldás is adható erre a problémára, pl. adhatunk egy szöveges megjegyzést, és aszerint választjuk ki a használt képet. Én egy publikusan elérhető helyre tettem ki az AMI id-t, és futás során innen kéri le az alkalmazás.

Az alkalmazáson belül konfigurálható az, hogy mennyi időnként fusson le az ütemezett feladat, ezt alapértelmezésben 10 percnek vettem. Ez idő alatt kényelmesen elindul az OS és a Liferay alaptelepítés is, valamint a fürtbe is be tud lépni.

Ezenkívül az instance kapacitások és költségek konfigurálhatóak. Ennek során minden típushoz meg kell adni, hogy mennyibe kerül óránként az adott instance típus, valamint biztonsággal hány kérést tud kiszolgálni percenként.

2.4.6 Instance konfiguráció

Kétféle instance-t kell konfigurálnunk. Először is van egy közös pont, ahova az adatbázist és a Terracottá-t kell tenni, valamint létre kell hozni a frontendekből az AMI-t. Az előbbi az egyszerűbb, az adatbázis létrehozása és konfigurálása tárolóból rutinfeladat, nem részletezem. A Terracotta telepítése szintén egyszerű, letöltés és kicsomagolás után akár indítható is lenne. Annyit kell rajta konfigurálni, hogy a *tc-config.xml*-be be kell írnunk azt a címet, ahol elérhető. Ha az instance-hez csatoltunk fix IP címet, akkor azt, egyébként a belső elérhetőségeit kell tenni.

A frontend kiszolgálók összeállítása már sokkal munkásabb. Először fel kell tennünk Java-t, majd egy vanilla Liferay-t, ez utóbbi letöltés és kicsomagolás után rendelkezésre is áll. Mivel az alkalmazás feltételezi, hogy a gép indulás után beépül a fürtbe, ezért be kell állítanunk, hogy indulás után a Liferay is automatikusan elinduljon. Ehhez létre kell hozni egy shell scriptet, amely belép a Liferay/Tomcat könyvtárba[24], majd elindítja a portált. Ezután futtatási jogot kell rá adni, majd az *update-rc.d* paranccsal az indítási sorba tenni.

Miután kész a környezetbeállítás, nekiállhatunk a Liferay konfigurációnak. Ehhez először is a *webapps/ROOT/WEB-INF/classes* alatt kell dolgoznunk. Első körben a gyorsítótárakat tesszük Terracotta alá. Előzetesként szükségünk van az *ehcache-terracotta*, a *quartz-terracotta* és a *terracotta-toolkit* jarokra, ezetek a *webapps/ROOT/WEB-INF/lib* alá kell másolni. Létre kell hoznunk a my-ehcache könyvtárat, a leírás[25] alapján. Ezen belül is *hibernate-terracotta.xml* és *liferay-multi-vm-terracotta.xml* fájlokban is át kell írnunk a Terracotta szerver elérést. Ezután a *portal-ext.properties*-ben az alábbi konfigurációt kell beállítanunk:

```
ehcache.multi.vm.config.location=/my-ehcache/liferay-multi-vm-terracotta.xml
net.sf.ehcache.configurationResourceName=/my-ehcache/hibernate-terracotta.xml
hibernate.cache.region.factory_class=net.sf.ehcache.hibernate.EhCacheRegionFactory
```

Ezzel megtörtént a cache koordináció Terracotta alá bevezetése. Következő lépés a keresőindexek központosítása az adatbázisban. Ehhez egyetlen sort kell beilleszteni a *portal-ext.properties*-be:

```
lucene.store.type=jdbc
```

Keményebb munka az ütemező elosztása. A *portal-ext.properties* nem támogat néhány szükséges felülírást, emiatt a *portal.properties*-t tömörítjük ki, majd módosítjuk benne az ide vonatkozó részeket:

```
memory.scheduler.org.quartz.jobStore.class = org.terracotta.quartz.TerracottaJobStore
memory.scheduler.org.quartz.jobStore.tcConfigUrl = <ide kerül a Terracotta szerver elérése>
persisted.scheduler.org.quartz.jobStore.class = org.terracotta.quartz.TerracottaJobStore
persisted.scheduler.org.quartz.jobStore.tcConfigUrl = <ide kerül a Terracotta szerver elérése>
#persisted.scheduler.org.quartz.jobStore.dataSource=ds
#persisted.scheduler.org.quartz.jobStore.isClustered=false
#persisted.scheduler.org.quartz.jobStore.misfireThreshold=60000
#persisted.scheduler.org.quartz.jobStore.tablePrefix=QUARTZ_
#persisted.scheduler.org.quartz.jobStore.useProperties=false
```

Ezzel megtörtént az ütemező elosztása. Ezután szükségünk van a Dokumentum Könyvtár S3-ba tevésére. Ehhez a *portal-ext.properties*-be vegyük fel az alábbi sorokat:

```
dl.store.impl=com.liferay.portlet.documentlibrary.store.S3Store
dl.store.s3.access.key=<Ide jön az AWS access key>
dl.store.s3.secret.key=<Ide jön az AWS secret key>
dl.store.s3.bucket.name=<Ide jön, hogy melyik vödröt akaruk használni>
```

Ehhez szükségünk van egy AWS access és secret kulcsra, amelyeket szintén az IAM-ben tudunk generálni. Végezetül ne felejtsük el az adatbáziselérést beállítani.

A munkamenetek elosztásához a Tomcat-et kell konfigurálnunk. Ehhez először is szükségünk van arra, hogy a *terracorra-sessions* és *terracotta-toolkit* jar-okat a *Tomcat/lib-be* bemásoljuk. Ezután a *conf/Catalina/localhost/ROOT.xml*-be kell felvennünk egy bejegyzést:

```
<Valve className="org.terracotta.session.TerracottaTomcat70xSessionValve" tcConfigUrl="<ide kerül a Terracotta szerver elérése>"/>
```

Ezzel készen is van a beállítás, nincs más hátra, mint a lefordított alkalmazást bemásolnunk a deploy könyvtárba, hogy indításkor automatikusan feltelepítődjön, majd a gépet leállítva készítsünk belőle egy AMI-t, aminek az id-jét az előző fejezetben ismertetett módon kezeljük a későbbiekben.

2.5 Tesztelés

2.5.1 Beállítások

Ahhoz, hogy egyáltalán megfigyelhető legyen élesben az alkalmazás, az előző fejezetekben ismertetett számos beállítást el kell végezni, valamint méretezéssel kapcsolatos tervezői döntéseket meg kell hozni. Először is meg kell határozni az egyes instance típusok kapacitását. A tapasztalat azt mutatja, hogy főként olvasásból álló oldalt egy SMALL típus másodpercenként 10 lekérést ki tud szolgálni, ez percenként 600. Mivel egy nagyobb weblapot tekintünk alapnak, ezért biztonsággal vegyük a felét, tehát processzoregységenként 300 kérés/perc lesz a kapacitás. Ennek megfelelően ezekkel az értékekkel inicializálom az algoritmust:

Név	Költség (dollár / óra)	Kapacitás (kérés / perc)
m1.small	0.09	300
c1.medium	0.18	600
m2.large	0.36	1200
m1.xlarge	0.506	2400
m2.xlarge	0.72	1950
m2.2xlarge	0.744	3900
m2.4xlarge	1.012	7800
c1.xlarge	2.024	6000

Ezután, mivel ez egy tesztkörnyezet, meg kell határozni a terhelési görbét. Jelenleg arra vagyunk kíváncsiak, hogy egy felfutásra megfelelően nő a kapacitás, valamint a visszaesést is rendben képes lekezelni az algoritmus. Ennek megfelelően úgy választottam meg a terhelésértékeket, hogy alacsonyról induljon, fusson fel ehhez képest magasra, ott maradjon egy pár negyedórát, utána induljon el lefele. A lefutásban egy pici emelkedést is tettem. Ennek megfelelően a következő sorozatból fogom a terheléseket venni:

100, 100, 180, 230, 300, 380, 490, 640, 800, 1000, 1300, 1700, 2100, 2900, 3500, 4000, 4000, 4000, 4000, 3000, 2500, 3000, 1000, 300, 100, 100, 100, 100

Az algoritmushoz még meg kell állapítani egy értéket, hogy mennyivel tervezzen az aktuális terhelés fölé, ezt 30%-nak vettem. Valamint szükséges, hogy a futás kezdetekor milyen típus fut, ezt pedig c1.medium-nak választottam.

A paraméterezést követően el kell végezni az AWS konfigurációt, létre kell hozni és be kell állítani a gépeket. Ezek pontos lépéseit az előző fejezetben már részletesen kifejtettem.

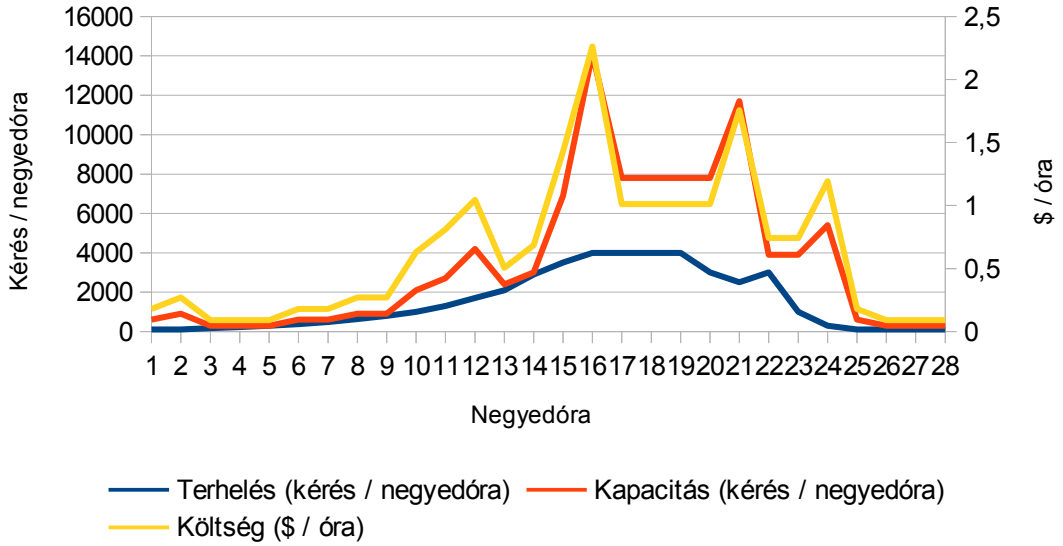
Végül meg kell határozni, hogy mennyi időnként fusson az algoritmus. Azt tapasztaltam, hogy 2-3 perc alatt biztonsággal elindul a Liferay és az operációs rendszer, ezután még kell egy rövid idő mire a terheléselosztóba megtörténik a regisztráció és az állapotellenőrzés. Végül a biztonságot szem előtt tartva 15 percre vettem, annyi idő alatt egy összetettebb oldalnak is el kell indulnia.

2.5.2 Futtatás

A koncepció egyszerű, mindössze annyiból áll, hogy elindítottam a központi kiszolgálót, a kezdeti frontend-et, beregisztráltam a terheléselosztóba, majd *JMeter*-rel [26] beállítottam az aktuális terhelést, valamint futás közben figyeltem a működést. Ezt negyedóránként felülvizsgáltam, majd a teszt végén leállítottam mindent.

2.5.3 Tapasztalatok

Futás közben minden kérést ki tudott szolgálni a fürt, valamint a konzolon figyelve valóban helyesen indultak és álltak le az instance-k. A mért kapacitás-, és költségértékeket az alábbi grafikon mutatja be:

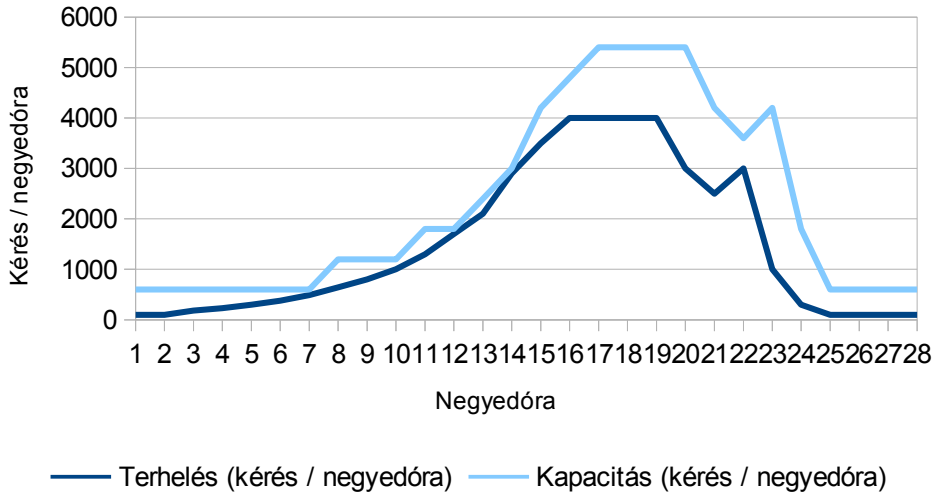


Kép 25: Mért értékek a tesztelés során

A teljes futás összköltsége 4.4815$ volt. A maximum terhelést kiszolgáló infrastruktúra optimálisan egy m2.2xlarge és egy m1.small instance-ból áll, ezek összkapacitása 4200 kérés/perc, a költsége pedig 0.834/óra. Ha a teszt időtartama alatt folyamatosan ezek futottak volna, akkor az összköltség 23.352$ lett volna, tehát az üzemeltetési költségeket mintegy 80%-al csökkentettük. Természetesen ennek ára van. Mint ahogy a grafikonon is látszik, az éles felfutásnál a kapacitás éppen csak fölötte van a terhelésnek. Mivel 30%-os tartalékot állítottunk be, ezért az algoritmus csak akkor tudja garantálni a kapacitástöbbletet, ha a terhelés két negyedóra között nem nagyobb, mint 30%. Ez valós körülmények között azonban nem feltétlenül igaz.

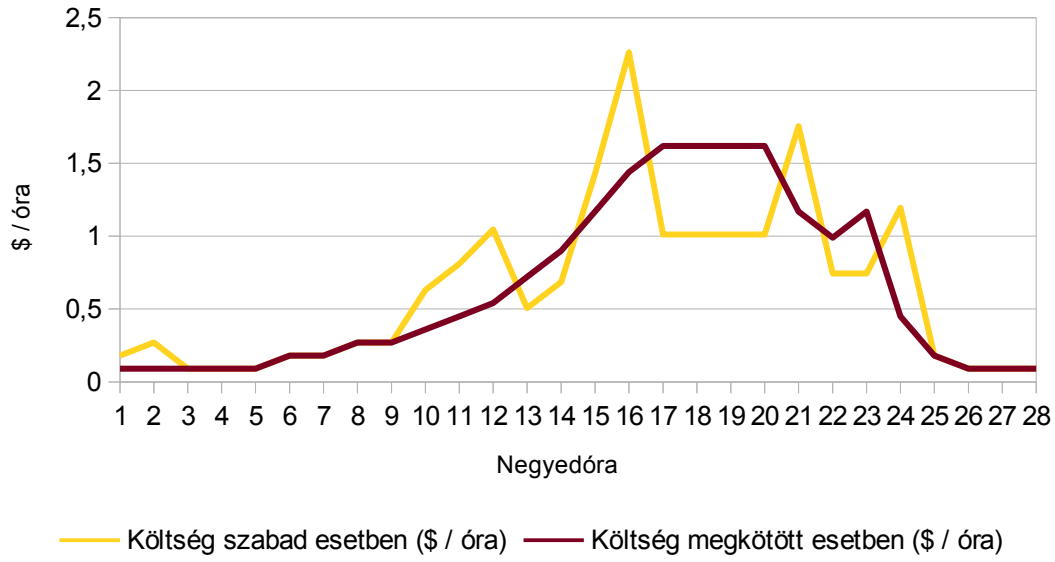
Érdekes megfigyelni a grafikonon az „ördögszarvakat”, a kiugró rövid kapacitástöbbletet a terhelés változására, függetlenül attól, hogy az lefele vagy felfele változik. Erre a magyarázat az, hogy felskálázásnál az algoritmus elindít egy nagyobb kiszolgálót, ugyanakkor a kisebbeket sem állíthatja le, így jelentkezik a jelentős többlet. Egy negyedórával később a frissen elindított gépek üzembe állnak, a régiek leállíthatóak, emiatt visszaesik az optimális szintre a fürt. Lefutásnál hasonló történik, a nagyobb típus helyére több kisebb lép, de az előbbi csak egy negyedórával később állítható le. Figyeljük meg, hogyha az algoritmusnak megkötjük, hogy csak egyetlen

típusból indíthat, akkor ez a jelenség megszűnik, az alábbi grafikonon ugyanaz a futás kizárólag c1.medium kiszolgálókkal:



Kép 26: Tesztelési eredmények csak medium kiszolgálókkal

Ennek a jelenségnek egy érdekes hozadéka, hogy sűrűn és nagymértékben változó terhelés esetén az általános algoritmus magasabb költséggel dolgozik, mint hogyha megkötnénk egyetlen instance típusra. Figyeljük meg ugyanerre a tesztfutásra a költséggrafikont szabad és m1.small típusra megkötött futásra:



Kép 27: Tesztelés költségeinek összehasonlítása szabad és megkötött esetben

A megkötés nélküli esetben az összköltség 4.4815$, ha megkötjük a kiszolgáló típusát, akkor pedig 4.41$.

2.6 Esettanulmány

Az előző fejezetben egy mesterséges teszteseten vizsgáltuk az alkalmazást, most viszont nézzük meg egy valós példán keresztül! Ehhez a Budapesti Értéktőzsde weblapjának üzemeltetési-, és látogatásadatait mutatom be, és erre futtatom a szimulátort.

Elöljáróban meg kell határoznunk azt, hogy a jelenlegi infrastruktúra az AWS-ben milyen instance-nak felelhet meg. A www.bet.hu-t jelenleg 2 db HP DL380 G5 szerver szolgálja ki. Ezen szerverek CPU-inak frekvenciatartománya 2-3GHz körül van, az AWS-ben egy processzoregység 1-1.2GHz, ezért egy processzormag teljesítményét 2 egységnek veszem. Mindkét szerver négymagos, tehát számítási teljesítményben egy Extra Large (m1.xlarge) típusnak feleltetem meg. A fizikai szerverekben 8GB RAM áll rendelkezésre, a portál azonban 1GB-n is elfut. Mivel az összes instance rendelkezik legalább 1GB memóriával, ezért úgy veszem, hogy bármelyiken képes futni a portál. A permanens háttértár jelen esetben nem jelenthet szűk keresztmetszetet, mivel az AWS-ben tetszőleges méretű diszk csatlakoztatható.

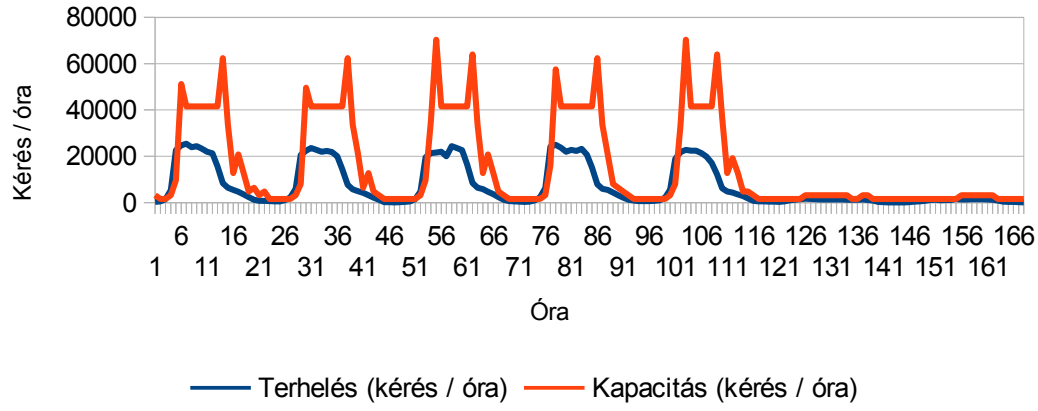
A terhelésadatok órás bontásúak, így az algoritmust is célszerű órásra állítani. Természetesen valós üzemeltetési körülmények között a nagyobb felbontás előnyösebb, itt viszont alkalmazkodni kell a historikus adatokhoz. Mivel több évnyi adat áll rendelkezésre, ezért kezdetben csak az utolsó teljes hetet mutatom be. A következő fejezetben a historikus adatokat minimalista idősoros elemzés alá veszem, és ebből próbálok következtetéseket levonni, valamint konfigurálni az algoritmust a jobb eredmények érdekében.

Ezek alapján az alábbi konfigurációt használom:

Név	Költség (dollár / óra)	Kapacitás (kérés/óra)
m1.small	0.09	1600
c1.medium	0.18	3200
m2.large	0.36	6400
m1.xlarge	0.506	12800
m2.xlarge	0.72	10400
m2.2xlarge	0.744	20800
m2.4xlarge	1.012	41600
c1.xlarge	2.024	32000

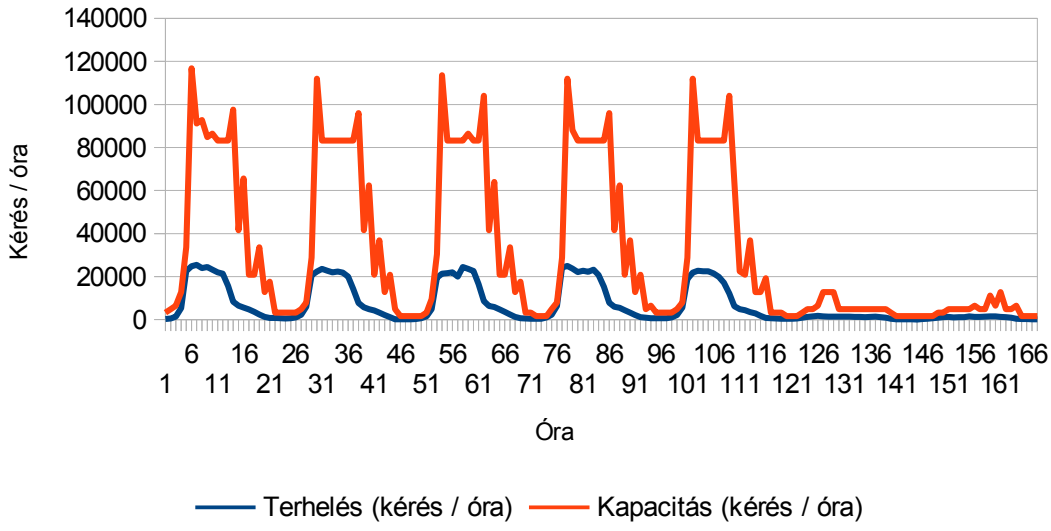
2.6.1 Az algoritmus vizsgálata egy hétre

Első közelítésként vizsgáljuk meg, hogy 30%-os fölé tervezéssel hogyan boldogul az algoritmus!



Kép 28: 30%-os fölé tervezéssel futtatott algoritmus

Az összköltsége 85.882$ lett, ha az eredeti felállást, tehát a 2db m1.xlarge instance futtatását vesszük alapul, az pedig 170.016$ lenne. Viszont hiába a jelentős költségmegtakarítás, az algoritmus a héten 13 órában is alultervezett, emiatt túlterhelődött a fürt. Mivel ennek lehetnek nehezen belátható következményei is, ezért próbáljunk meg tenni ellene valamit. Első közelítésben nézzük meg mi történik, ha a 30%-os tartalékot megnöveljük! Ebből a megközelítésből kijön, hogy ha a sértések számát 0-ra akarjuk csökkenteni, akkor 2.5-szeres túltervezésre van szükség.



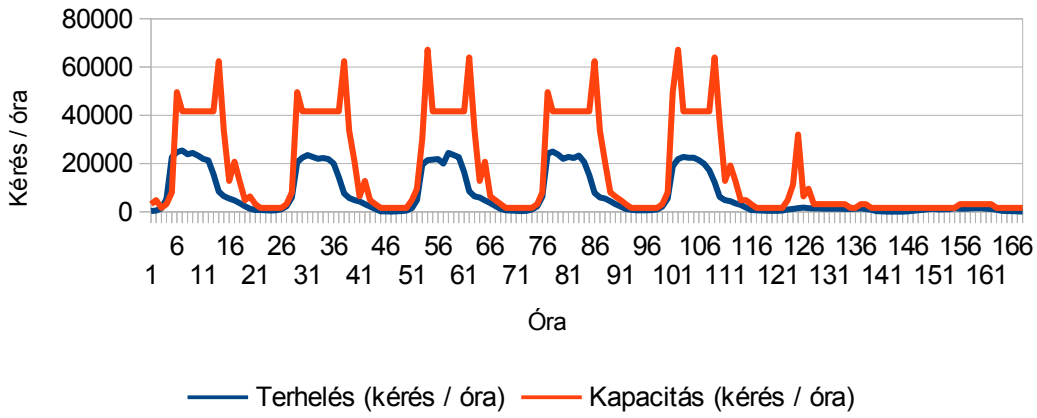
Kép 29: 250%-os fölé tervezéssel futtatott algoritmus

Ebben az esetben a sértések száma 0, ugyanakkor az összköltségünk 161.148$-ra növekedett, tehát ezzel az árelőnyt el is vesztettük. Megállapíthatjuk, hogy a reggeli felfutás olyan mértékű, amit ilyen egyszerű felültervezéssel nem tudunk korrigálni. Ugyanakkor jó periodikusságot fedezhetünk fel az adatokban, nagyjából minden nap egy hirtelen felfutással kezdődik, a csúcson nagyjából konstans a terhelés, végül a nap végére lecsillapodik a látogatás, éjszakára pedig szinte teljesen megszűnik. Próbáljuk meg, hogy milyen eredményt ad egy olyan módosítás, hogy a felültervezés függ a 24 órával ezelőtt mért terheléstől is! Jelen esetben a konkrét terhelésértékeket nem akarjuk felhasználni, hiszen nem feltétlenül lesz pont akkora a látogatottság mint előző nap, pl. a hétvégék tipikusan ilyenek. Tehát induljunk ki abból, hogy megnézzük, hogy 24 órával ezelőtt mekkora volt a terhelésváltozás, erre hagyjunk rá 30%-ot, de a kérésszámhoz képest is legyen legalább ekkora többlet. Tehát a pontos képlet az alábbi lesz:

Jelöljük r_i-vel az i. órában mért terhelést! Ekkor az x. órában a tervezett terhelés a következő lesz:

$$\max(\max(r_{i-23}-r_{i-24},0)*1.3+r_i,r_i*1.3)$$

Ezzel futtatva az alábbi eredményt kapjuk:

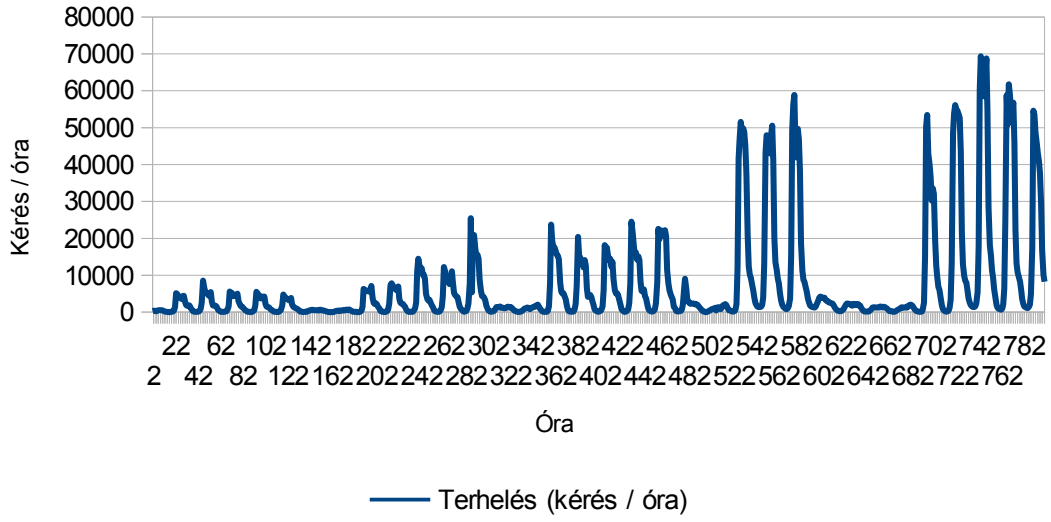


Kép 30: Előző napi adatok alapján korrigált algoritmus

Összességében látszik, hogy az első napot leszámítva, sikerült elkerülnünk a sértéseket. Érdekességképpen megfigyelhető a szombati szokásos felfutás, azonban amikor látszik, hogy a terhelés nem követi a szokásos menetrendet, akkor szépen lecsökken a kapacitás is.

2.6.2 A gazdasági világválság miatt megemelkedett forgalom kezelése

A teljes időszak összesen 33639 órát tartalmaz, ez majdnem 4 év terhelésadata. A legfrissebb adatsor 2011. december 6-ai, ebből visszaszámolva rájöhetünk, hogy üzemeltetésileg történt egy érdekes esemény. Amikor beütött a 2008-as gazdasági válság, ahogy a tőzsdeindexek esni kezdtek, úgy az egekbe szökött a látogatószám, egyik napról a másikra a szokásos terhelés többszörösét kellett kiszolgálniuk a szervereknek. Az alábbi képen jól látszik a különbség. Az előtte levő időszakban az 5000 körüli érték bőven átlépi az 50000-et is.



Kép 31: A Budapesti Értéktőzsde weblapjának terhelése a válság kirobbanásának idején

Hogyan lehetne egy ilyen eseményt kezelni? Vizsgáljuk meg először, hogy az eredeti algoritmus hogyan boldogul a problémával! Állítsunk be 30%-os rátartást, és egyelőre ne vegye figyelembe az előző napot. Azt kapjuk, hogy 62 esetben történik sértés, az összköltségünk pedig 407$. Ez elég magas. Most nézzük meg, hogy ha az előző pontban

ismertetett módon korrigálunk az előző nap változásaival, akkor mit kapunk! Ilyenkor 15 sértés mellett 441$ az ár. Jelentősen csökkentettük a kapacitáshiányt, az ár pedig nem emelkedett számottevően. Elemezve az adatokat látszik, hogy a legtöbb sértés a hét első napján keletkezik, ezért korrigáljunk még az 1 héttel előtte levő változásokkal is! Ebben az esetben a képlet erre módosul:

$\max(\max(\max(r_{i-23}-r_{i-24}, r_{i-7*24-1}-r_{i-7*24}),0)*1.3+r_i, r_i*1.3)$

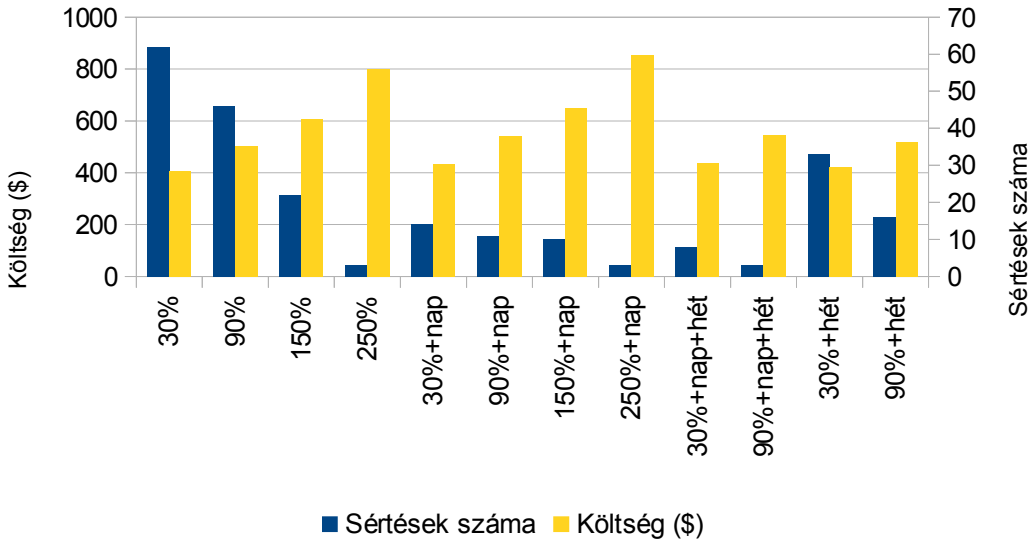
Erre futtatva a sértések száma 7 lesz, 443$-os összköltséggel.

Az alábbiakban egy táblázatban összefoglalom a különböző algoritmusokat:

Algoritmus leírása	Sértések száma	Költség ($)
30%-os rátartás	62	407
90%-os rátartás	46	505
150%-os rátartás	22	609
250%-os rátartás	3	800
30%-os rátartás, figyelembe véve az előző nap változásait is	14	433
90%-os rátartás, figyelembe véve az előző nap változásait is	11	542
150%-os rátartás, figyelembe véve az előző nap változásait is	10	648
250%-os rátartás, figyelembe véve az előző nap változásait is	3	853

30%-os rátartás, figyelembe véve az előző napi és az előző heti változásokat	8	437
90%-os rátartás, figyelembe véve az előző napi és az előző heti változásokat	3	547
30%-os rátartás, figyelembe véve az előző heti változásokat	33	423
90%-os rátartás, figyelembe véve az előző heti változásokat	16	520

Ugyanez grafikonon:



Kép 32: Különböző megoldások összehasonlítására a válság miatt megnövekedett terhelés kiszolgálására

Látható, hogy az előző nap, illetve az előző hét figyelembe vétele nagyban segít a sértéseket leszorítani, és csak alacsony költségemelkedést hordoz magában. Az egy számjegyű sértést tekinthetjük elfogadhatónak, ugyanis az első napokban még nincs előző napi/heti adatunk, így a reggeli felfutásra se tudunk felkészülni rendesen.

Érdemes megfigyelni, hogy ha a maximumra tervezünk, akkor ebben az időszakban a csúcsterhelés 39374 kérés/óra volt, amit 2db m2.4xlarge instance tudna csak kiszolgálni, aminek erre az időszakra vetített költsége 1619$ lenne. Ehhez képest mindegyik fentebb ismertetett megoldás jelentősen olcsóbb.

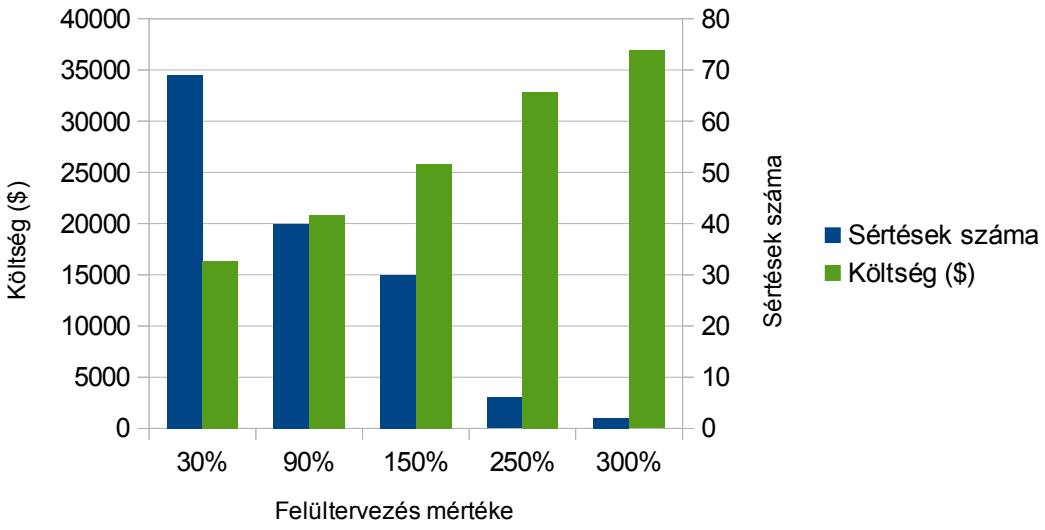
2.6.3 A teljes adatsor áttekintése

Végül vizsgáljuk meg a teljes adatsorra az összköltséget és a sértések számát. Az előző fejezetben már láttuk, hogy az előző napi és az előző heti adatok figyelembe vételével hatékonyabban tudjuk kezelni a nap eleji felfutásokat, ezért ezeket az optimalizációkat használjuk a teljes adatsor elemzésénél is.

Vizsgáljuk meg, hogyan teljesít az algoritmus a maximumra tervezett kapacitással szemben, különböző többletértékeket beállítva! Az abszolút maximum terhelés 69374 volt, amit 2db m2.4xlarge instance tudna kiszolgálni. Ennek a teljes adatsorra, tehát 33647 órára, vetítve az összköltsége 68085$. Tekintsük ezt a kiindulási alapnak a továbbiakban, amihez a költségeket tudjuk viszonyítani. Értelemszerűen ebben az esetben soha sincs kapacitásprobléma, tehát a sértések száma 0. A mért eredményeket az alábbi táblázatba foglalom össze:

Többlet mértéke (%)	Sértések száma (db)	Költség ($)	Költség aránya a referenciaértékhez képest (Költség / 68085)	Elméleti rendelkezésreállás (1-Sértések száma / 33647)
30	69	16322	24%	0.9979
90	40	20798	30.5%	0.9988
150	30	25781	37.9%	0.9991
250	6	32839	48.2%	0.9998
300	2	36900	54.2%	0.9999

Grafikonon ábrázolva:



Kép 33: Különböző paraméterezések összehasonlítása a teljes adatsorra

Jól látható, hogy a költségek emelése mellett drasztikusan tudjuk csökkenteni a sértések számát. Ezen kívül érdemes megfigyelni, hogy minimális sértésszám mellett is csak a referenciaköltségek felénél tartunk. 2 db sértés a teljes adatsoron még 9999-es rendelkezésre állást biztosít. Valamint a megnövekedett terhelés nem feltétlenül jelenti a szolgáltatás összeomlását, ez még számos más tényezőtől is függ: pl. a terhelés karakterisztikájától, valamint hogy mennyire határoztuk meg pontosan egy kiszolgáló terhelhetőségét.

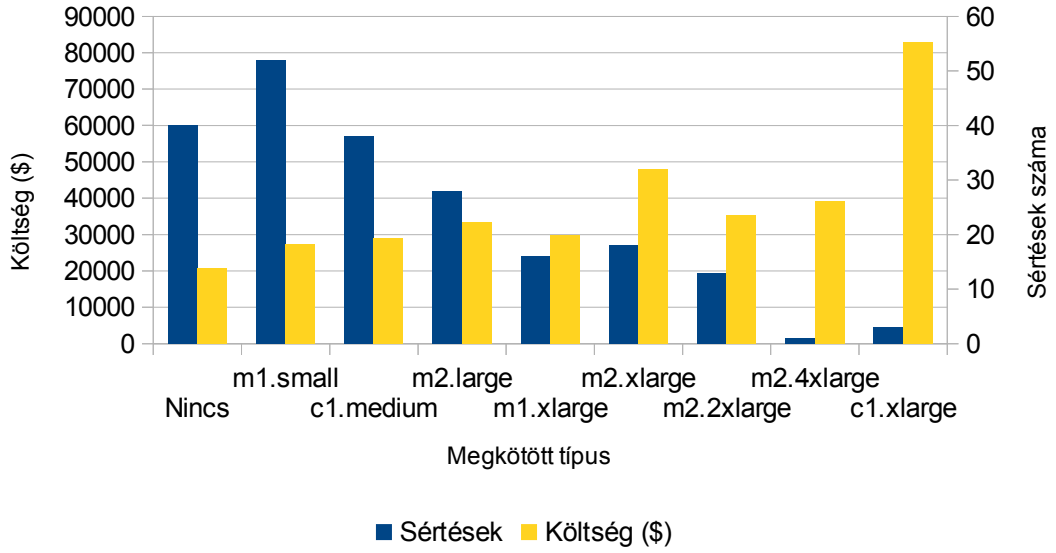
Ezen kívül érdemes végiggondolni, hogy egy nem várt, hirtelen terhelésnövekedést kezelt le az algoritmus hatékonyan, amit a kézi üzemeltetés valószínűleg nem lett volna képes. A maximumra tervezésnél eddig nem vizsgáltuk, hogy előre nehezen lehet megmondani, hogy mennyi is lesz a maximális terhelés. Emiatt a változásokat sokkal nehezebb lekövetni, mivel az architektúra is olyan módon áll össze, hogy nehéz a fürtbe újabb gépeket léptetni. Valamint egy új kiszolgáló elindítása számos lépésből áll, ami ha nincs automatizálva, akkor jelentős mennyiségű időbe is telik. Így ha az üzemeltetés nincs felkészítve tartalékokkal és egy skálázási tervvel, akkor könnyen belefutnak abba a szituációba, hogy egy hirtelen esemény hatására bekövetkező terhelésugrást nem képesek kiszolgálni, és akkor valóban össze fog omlani a fürt. Ehhez képest már elhanyagolható kompromisszum, hogy ha évente egy pár órában alátervez az algoritmus. Ezen kívül még azt is figyelembe kell venni, hogy az adatok órásak, viszont a valóságban akár perces felbontásúak is lehetnek. Emiatt sokkal gyorsabb a beavatkozási lehetőség.

2.6.4 Összehasonlítás az Autoscale-el

Hasonlítsuk össze az algoritmust az Amazon beépített skálázó megoldásával, az Autoscale-el! Érdemes megfigyelni, hogy az Autoscale az algoritmusom egy speciális esete, mely szerint egyetlen instance típussal dolgozik kizárólag. Ennek megfelelően futtassuk le az összes típusra egymás után, és hasonlítsuk össze a kapott eredményeket! Természetesen itt is meg kell választani az algoritmus ráhagyását, ezt itt 90%-nak veszem, ez elfogadható kompromisszum a költség és a rendelkezésreállás között. A futtatások eredményét az alábbi táblázatba gyűjtöttem össze:

Instance megkötés	Sértések	Költség ($)
Nincs	40	20798
m1.small	52	27244
c1.medium	38	28945
m2.large	28	33374
m1.xlarge	16	29679
m2.xlarge	18	47820
m2.2xlarge	13	35186
m2.4xlarge	1	39182
c1.xlarge	3	83034

Grafikonon ábrázolva:



Kép 34: Az algoritmus összehasonlítása az Autoscale-el

Látható, hogy mind közül a megkötés nélküli eset került a legkevesebbe, ugyanakkor a többihez képest magas sértésaránnyal dolgozott. A grafikon alapján az m1.xlarge megkötött eset tűnik a legjobb ár/érték arányúnak, azonban a számok ebben az esetben csalókák. Az előző fejezetben tárgyalva jól látszik, hogy ha a megkötés nélküli algoritmusnak magasabb ráhagyást állítunk be, akkor nagyjából ugyanakkora költséggel fogja elérni a sértésszintet. Másrészről pedig a megkötött esetnél előre ki kell választani a használt instance típust, akkor pedig még nehéz megalapozott döntést hozni a későbbi terhelés karakterisztikájáról. A megkötés nélküli algoritmusnál erre nincs szükség, és összességében elég jó eredményt ért el a mérések alatt. Összességében tehát a használata indokolt az Autoscale-el szemben. Hátránya viszont, hogy a konfigurációja bonyolultabb és egy folyamatosan futó gép szükséges hozzá, ezzel szemben az Autoscale konfigurálható az Amazon konzoljáról.

2.7 Kitekintés

Az elkészített alkalmazást számos ponton tovább lehet még fejleszteni. Felsorolásszerűen egy lista a lehetséges irányokról:

- Jelenleg a leállított gépek naplófájlja elveszik, ez viszont hibakereséshez fontos információkat is hordozhat. Szükséges lenne, hogy egy gép leállítása előtt ezt a fájlt kimentse az algoritmus egy permanens tárhelyre.
- Jelenleg kizárólag a bejövő kérések számából indul ki az algoritmus. Érdemes lenne megvizsgálni, hogy a futó gépek egyéb jellemzői alapján mennyivel lehetne optimálisabb megoldást kihozni, pl. CPU és hálózati forgalom alapján.
- Az előző napi és az előző heti adatok vizsgálata feltételezi, hogy egy nap 24 órából áll, ez azonban az óraátállításokkor nem igaz. Jobb lenne, ha az algoritmus ezt is figyelembe venné.
- A vizsgálat során kizárólag az instance-ek költségeit vettem figyelembe, az egyéb költségeket, pl. a terheléselosztó üzemeltetését, a forgalom alapján történő számlázást, valamint a tárolási költségeket nem.
- Jelenleg az algoritmus előre beállított időközönként fut, amik között feltételezzük, hogy az elindított gépek belépnek a fürtbe. A gyorsan változó terhelés kiszolgálására hatékonyabb lenne egy olyan algoritmus, ami mondjuk percenként fut le, de akkor csak elindít gépeket, és csak a biztos elindulás után állítja le őket. Ezzel valószínűleg még drasztikusan lehetne csökkenteni a költségeket és sokkal dinamikusabban lehetne kezelni a változásokat.
- Jelenleg a portlet nincs lokalizálva.

3 Összefoglalás

Diplomamunkám alapvetően három részre osztható. Egyrészt megvizsgáltam a Liferay lehetőségeit az elosztott működésre, valamint elhárítottam a felmerült akadályokat. Második részben megvizsgáltam az elérhető felhőplatformokat, különösképpen az Amazon Web Services-t, majd pedig kialakítottam, megvizsgáltam és az elérhető megoldásokkal összehasonlítottam egy olyan megoldást, amellyel a fürt mérete beavatkozás nélkül követi a terhelés változásait. Végül pedig Liferay alá fejlesztettem egy egyszerű portletet, amely adatbázisban tárol néhány adatot, valamint azt megfelelő formában megjeleníti.

A tapasztalatom az, hogy a Liferay elosztott működése megoldható. Bár nem zökkenőmentes ez a folyamat, valamint éles üzemeltetés közben bizonyára előjöttek volna olyan előre nem látott problémák, amelyeket nem vizsgáltam a munkám során, koncepcionális akadálya nincs a fürtben való futásnak. Ugyanakkor, mivel nagy szoftverről van szó és rengeteg komponensből áll, a hibakeresése és a fejlesztése sokszor nehézkes, elsőre nehezen átláthatóak az összefüggések. A dokumentáció is helyenként erősen hiányos, elsődlegesen a kód olvasásából lehet rájönni egyes funkciók működésére.

Az Amazon felhőszolgáltatásáról alapvetően jók a tapasztalataim, általában megbízhatóan működött. A számításaim alapján manapság nem éri meg olyan architektúrát alkalmazni, amely nem képes a felhőben futni, mivel az üzemeltetési költségekben jelentős különbségeket állapítottam meg a skálázódó, valamint a maximumra tervezett infrastruktúra között. A felhőről a véleményem az, hogy egy olyan vállalatnak, amelynek nincsenek nagyon speciális igényei, nem érdemes saját infrastruktúrát fenntartani és üzemeltetni, mivel ez nagyon rugalmatlan. Ugyanakkor a monitorozásra kiemelt hangsúlyt kell fordítani, ha az algoritmusban valami hiba van, akkor könnyen elszállhatnak a költségek is.

A Liferay alá történő portletfejlesztés akkor egyszerű, ha az igények alapján hasonlít már meglevő funkcióra az alkalmazás. Amire a keretrendszer ad támogatást azt egyszerű megoldani, amire nem, annak a megoldása hosszú körüljárásokkal valósítható csak meg. Jó példa erre a Maven támogatás, ami elvileg támogatott, ugyanakkor sokkal kevésbé működőképes mint az eredeti, Ant-os fordítás. UI szintjén azonban sokat segítenek a beépített újrafelhasználható komponensek, ezekkel nagyon egyszerűen meg lehet valósítani látványos, valamint a megjelenítésbe beleillő funkciókat.

Összességében a Liferay skálázódása az Amazon felhőjében megoldható és működőképes koncepció, azonban rá kell számolni a technológiák költségeit (TCO, Total Cost of Ownership[27]), amely adott esetben jelentősen megdrágítja a kifejlesztést, valamint az üzemben tartást is.

4 Irodalomjegyzék

[1] Database Sharding: http://www.codefutures.com/database-sharding/

[2] Terracotta: http://terracotta.org/

[3] Azure: www.windowsazure.com

[4] AppEngine: https://developers.google.com/appengine/

[5] AWS: https://aws.amazon.com/

[6] Liferay: http://www.liferay.com/

[7] LPS-23106: http://issues.liferay.com/browse/LPS-23106

[8] JSR168:
http://developers.sun.com/portalserver/reference/techart/jsr168/pb_whitepaper.pdf

[9] JSR286: http://jcp.org/aboutJava/communityprocess/edr/jsr286/

[10] Service Builder:
http://www.liferay.com/community/wiki/-/wiki/Main/Service+Builder

[11] Richard Sezov, Jr: Liferay in Action, ISBN 9781935182825

[12] JSR170: http://jcp.org/en/jsr/detail?id=170

[13] JSR283: http://jcp.org/en/jsr/detail?id=283

[14] Apache Jackrabbit: http://jackrabbit.apache.org/

[15] Apache Lucene: https://lucene.apache.org/core/

[16] Quartz Scheduler: http://quartz-scheduler.org/

[17] Ehcache: http://ehcache.org/

[18] LPS-21069: http://issues.liferay.com/browse/LPS-21069

[19] AutoScale: https://aws.amazon.com/autoscaling/

[20] EC2 instance típusok: https://aws.amazon.com/ec2/instance-types/

[21] IP feladat: http://en.wikipedia.org/wiki/Integer_programming

[22] Alloy UI: http://alloy.liferay.com/

[23] SearchContainer:
http://www.liferay.com/community/wiki/-/wiki/Main/SearchContainer

[24] Liferay fórumbejegyzés: https://www.liferay.com/hu/community/forums/-/message_boards/view_message/13136626

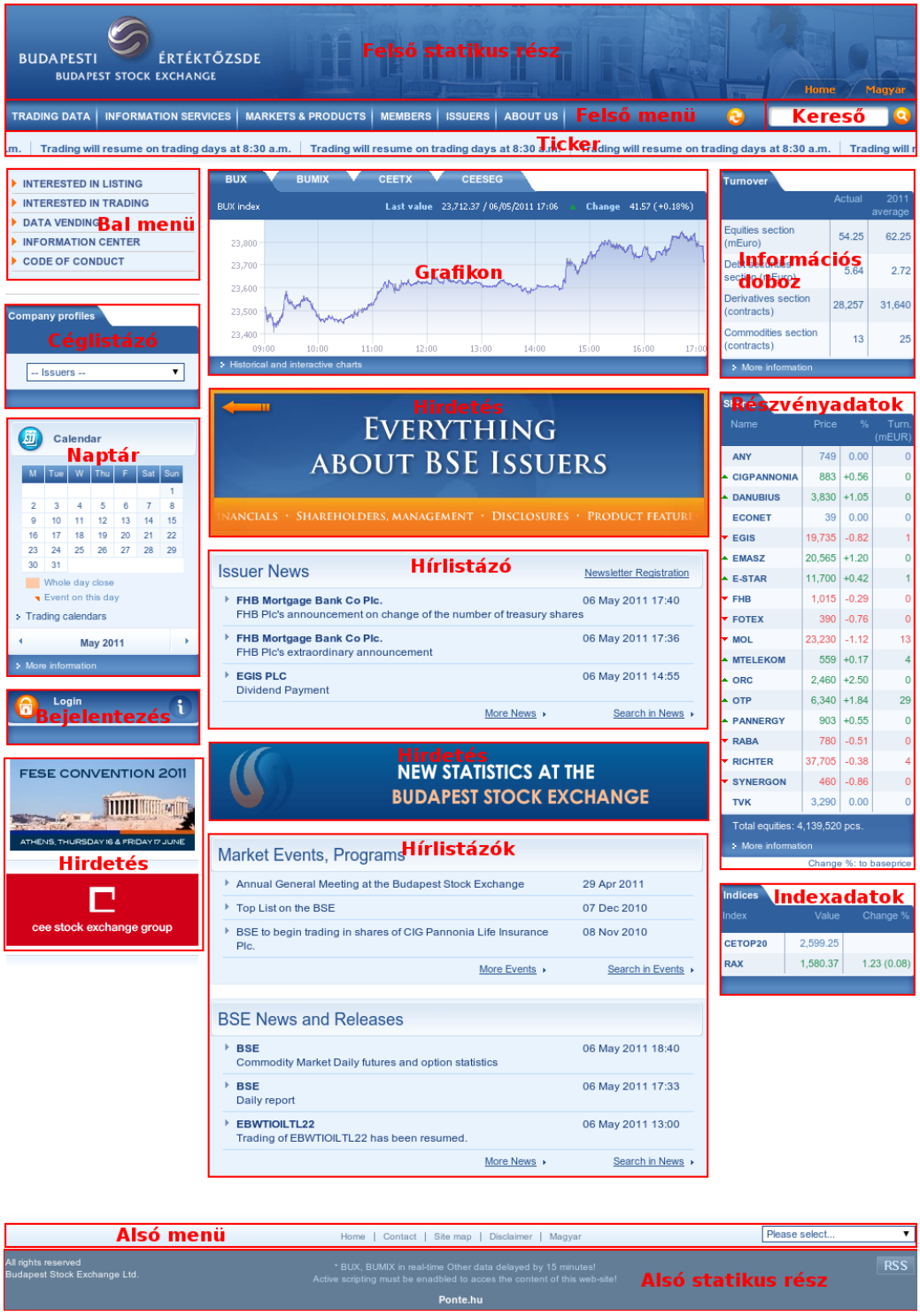
[25] EhCache Terracotta integráció: http://www.liferay.com/web/mika.koivisto/blog/-/blogs/how-do-i-cluster-liferay-with-terracotta-

[26] Apache JMeter: https://jmeter.apache.org

[27] TCO: http://en.wikipedia.org/wiki/Total_cost_of_ownership

5 Képjegyzék

6 „A” Függelék



Kép 35: Egy weboldal, kiemelve a részalkalmazásokat