Nyílt forráskódú elosztott rendszerek

Trencséni Márton, mtrencseni@scalien.com
Gazsó Attila, agazso@scalien.com

Abstract

Az előadás tézise, hogy néhány éven belül nyílt forrású elosztott rendszerek fognak nagy skálájú, nagy megbízhatóságú webes háttérarchitektúráját szolgáltatni. Ezen rendszerek jelenleg a nagy Internetes cégek, mint Google, Amazon és Facebook által nyilvánosságra hozott architektúrák, mérnöki tapasztalatok, illetve forráskódok alapján készülnek. Az előadásban ismertetjük az iparág által felhalmozott tapasztalokat és tervezési pontokat, melyek segítségével jobban érthetők lesznek az elosztott rendszerek közötti különbségek, előnyök, hátrányok, informált döntések hozhatók. Az előadás második felében a jelenleg is elérhető elosztott rendszereket ismertetjük, különös hangsúlyt fektetve a saját készítésű Keyspace kulcs-érték adatbázisra.

1 Bevezetés

Ma már a legtöbb alkalmazás *web alkalmazás*, fusson az az Interneten vagy céges belső hálózaton. A web alapú alkalmazások sajátossága, hogy a felhasználó adatait a szolgáltató rendszerein tárolják, és az alkalmazás futása során a szolgáltató rendszere is számításokat végez. A webes alkalmazások természetüknél fogva elosztott rendszerek, általában három különböző számítógépen fut a browser, az alkalmazás szerver és az adatbázis szerver. Egyre inkább igény van arra, hogy ezek az elosztott rendszerek, pontosabban az alkalmazás és adatbázis réteg skálázhatók és/vagy hibatűrők legyenek, azaz minél több klienst ki tudjanak szolgálni, minél több adatot tudjanak tárolni, illetve egyes komponensek meghibásodása esetén a rendszer összeségében tobább üzemeljen.

Ebben az előadásban a *nyílt forráskódú* elosztott rendszerekről lesz szó. Ezeket a szoftvereket általában néhány éve kezdték el fejleszteni, és egy-két kivételtől eltekintve de facto standard megoldások (mint pl. Mysql nyílt forrású adatbázisok terén) még nincsenek. Az előadás alaptézise, hogy néhány éven belül létezni fognak produkciós rendszerekben használható skálázható, hibatűrő rendszerek; az előadás ezeknek a rendszereknek a rövid történetével kezdődik, majd néhány, már használható rendszert mutat be, különös hangsúlyt fektetve a szerzők saját (Scalien Kft.) készítésű nagy megbízhatóságú kulcs-érték adatbázisára, a *Keyspace*-re.

Az előadás első részében általános elveket ismertetek melyek az elosztott rendzserek megértéséhez elengedhetetlenek: shared nothing architektúra, CAP háromszög, konzisztencia kérdések.

A legelső, nagyon nagy webes alkalmazásokat kiszolgáló elosztott rendszerek nagy Internetes cégeknél alakultak ki. Néhány esetben cikkekben publikálták a rendszer működését, néhány rendszernek pedig kiadták a forráskódját is. A jelenleg fejlesztés alatt álló nyílt forráskódú projektek is ezen — komoly mérnöki tudást és tapasztalatokat képviselő — rendszerekből merítenek ötleteket és általános

elveket, gyakran ezeket a rendszereket duplikálják. Ezért az előadás első részében *a Google Chubby, BigTable, MapReduce és az Amazon Dynamo* belső használatban lévő elosztott rendszereit ismertetem a fontosabb tervezési pontokra koncentrálva.

Az előadás második részében 1.x verziónál tartó, jelenleg is fejlesztés alatt álló, *a saját fejlesztésű Keyspace* rendszert mutatom be.

2 Elosztott rendszerek tulajdonságai

A webes alkalmazások és az open-source világában az ún. *shared nothing* [1] elosztott architektúra dominál, ami lényegében azt jelenti, hogy különálló szerverek együttesen alkotnak egy elosztott rendszert, de nincsen szorosan csatolva (pl. hardveres vagy operációs rendszer szinten) a gépek memóriája (shared memory) vagy diszkjei (shared disk).

Az elosztott rendszereknél alapvető ökölszabály az ún. *CAP (consistency, availability, partition tolerance) tézis* [2], mely azt mondja ki, hogy a felsorolt három tulajdonság közül nem valósítható meg mindhárom egyszerre shared nothing architektúrákban. A három fogalom tömör magyarázata:

1. *Konzisztencia:* ez elosztott rendszerhez intézett, egymást követő írás és olvasás műveletek esetén, melyeket potenciálisan más-más szerver szolgál ki, milyen garanciákat nyújt a rendszer arra, hogy az olvasás során az előzőleg beírt adatot viszontlátjuk.
2. *Rendelkezésre állás:* a rendszer képes kérések (írás és olvasás műveletek) kiszolgálására néhány szerver kiesése mellett is.
3. *Particíció tolerancia:* a rendszer működése, amennyiben a szervereket összekötő hálózat (hub, switch, router, kábel) meghibásodása esetén a rendszer kettő vagy több különálló hálózatra esik szét.

Két rövid példán keresztül ecseteljük, hogy a "CAP háromszögben"elhelyezett különböző rendszerek hogyan viselkedhetnek.

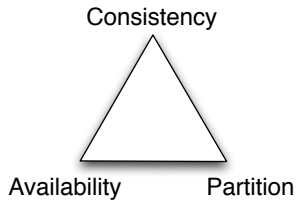

Figure 1: A CAP háromszög.

Első példaként képzeljünk el egy $n = 3$ szerverből álló rendszert, ahol induláskor mindhárom szerver szerint az `mtrencseni` felhasználó (az elosztott adatbázisban) tárolt születési dátuma `1881-04-24`, majd megváltoztatjuk `1981-04-24`-re, de a változás csak az 1. és 2. szerveren történik meg, és azok, mielőtt továbbítanák a változást a harmadikhoz, hiba folytán lekapcsolódnak. Újra lekérdezve a születési dátumot a még rendelkezésre álló szervertől a régi, elavult, "rossz" adatot kapjuk vissza. Egy ilyen esetet engedélyező rendszert *gyengén konzisztensnek* nevezünk; ez a furcsa, de nagy rendelkezésre állású és mindenféle particionálást magában foglaló működés előnyös, amikor egy "régi, elavult, kicsit rossz" adat visszakapása

előnyösebb, mint hibával visszatérni. Ilyen, gyengén konzisztens rendszer a később bemutatott Amazon Dynamo.

Második példaként vegyünk egy "többség alapú" rendszert, amelynél írás és olvasás műveletekhez a szerverek többsége rendelkezésre kell álljon. Egy ilyen rendszer csak akkor fejez be írás műveletet, ha a szerverek többségére kikerült. A fenti $n = 3$ példánál maradva, az írás műveletnél a kliens fele akkor jelez OK-t a rendszer, ha legalább két szerverre kikerült az új `1981-04-24` adat. Amennyiben két szerver kiesik, a rendszer nem tud olvasás műveletet kiszolgálni, mert többség kell; viszont ha kettő rendelkezésre áll, akkor mindig vissza tudja adni a "jó, friss" értéket hiszen a rendelkezésre álló két gépből legalább az egyiken megvan. Ilyen, erősebb garanciát biztosító rendszereket erősen konzisztensnek nevezünk; az ilyen rendszer, ahol a működéshez többség kell, egészséges állapotában nagyon hasonlít egy hagyományos, egy szerveres rendszerre. Az erős konzisztencia ára, hogy a szerverek többsége egy partícióban rendelkezésre kell álljon. Ilyen, erősen konzisztens rendszer a saját készítésű Keyspace.

3 Google architektúra

A Google néhány évvel ezelőtt cikkek formájában nyilvánosságra hozta a belső rendszerének leírását. A rendszer a Google keresőjére lett optimalizálva, azóta azonban teljesen más alkalmazások is futnak fölötte, pl. Google Mail és Google App Engine, ami az eredeti rendszer robosztusságát jelzi.

A Google architektúrája a következő elemekből épül fel:

1. Chubby: elosztott lock szerver [4].
2. Google File System (GFS): nagy teljesítményű elosztott file rendszer [5].
3. MapReduce: elosztott batch feldolgozó rendszer [6].
4. Bigtable: tábla alapú elosztott adatbázis [7].

A Chubby egy elosztott lock szerver, amelyt más szolgáltatások (pl. GFS vagy Bigtable) használnak jól ismert elosztott primitívként. Egy Chubby cella több tízezer másik szerveren futó elosztott rendszereket szolgál ki, amelyek master választásra vagy alapvető metaadatok (pl. mely szerverek részei a rendszernek?) megosztására használják azt. A Chubby egy erősen konzisztens rendszer, lényegében a többségi alapú Paxos [3] algoritmust valósítja meg, melyről később még lesz szó a Keyspace kapcsán.

A GFS a Google második generációs nagy teljesítményű elosztott filerendszere, mely a keresőhöz szükséges nagy mennyiségű, szekvenciális írásokhoz (pl. weboldalak lementése), és kisebb, véletlenszerű olvasásokhoz (pl. keresésnél) lett optimalizálva. Master-alapú filerendszer, ahol a master tárolja az tárolja az összes metaadatot, és ún. chunkszerverek tárolják a chunkokra bontott fileokat, 64MB-os blokkokban, replikálva. Érdekesség, hogy mivel egy master szerver van, ezért az összes metaadatot memóriában tárol, hogy a kliens kéréseket megfelelő sebességgel kiszolgálhassa, ami bizonyos korlátokat jelent a rendszerre nézve (metaadat mérete). Egy GFS rendszer néhány millió file-t, petabyte mennyiségű adatot tárol. Konzisztencia szempontjából a metaadatok erősen konzisztensek, mivel a master szerveren keresztül történik a változtatásuk, míg az chunkok lényegében egy eventual consistency modelt követnek, melyre az elosztott filerendszert használó alkalmazásnak fel kell készülnie.

Míg az eddig felsorolt rendszerek file vagy adatbázis rendszerek voltak, a MapReduce egy elosztott job-kezelő rendszer, melyet a Google a keresőjének alapjául

szolgáló index előállításához használ. A MapReduce lényege, hogy a feladatot a funkcionális nyelvekből ismert egy Map és egy Reduce lépésre bontja, amelyeket a rendszer automatikusan szétoszt és két fázisban végrehajt. A legegyszerűbb példa weblapokban szavak előfordulását számolja ki: a Map lépésben egy weblapból kiszedi a w szavakat, és $(w, darab)$ alakú kulcs-érték párokat ír ki; a Reduce lépésben a w szót tartalmazó párokat található darabszámokat összeadja, így megkapható a szavak előfordulása egy adott mintában. A Map és Reduce lépések eloszthatók, így nagy mennyiségű adatot lehet egyszerre, gyorsan feldolgozni. A Google esetében a MapReduce rendszer GFS vagy Bigtable fölött fut.

Az utolsó Google rendszer amit itt említünk a tábla (sor/oszlop) alapú tárolásra használt Bigtable. A hagyományos relációs adatmodell helyett a Bigtable egy, elosztott módon is implementálható, lényegében kulcs-érték adatmodellt kínál. A Bigtable adatokat `(sor, oszlop) -> adat` címzéssel kaphatja vissza a kliens; illetve egy plusz `verziót` is megadhat, amivel az egy adat egy régebbi verzióját kaphatja vissza, ti. a Bigtable változtatás esetén automatikusan tárolja a régi verziókat is. A hozzáfárás sor (és oszlop) szinten történik, és csak sor szintű módosítások végezhetők tranzakciósan. A Bigtable GFS fölött fut, és Chubby-t használ master kiválasztásra és metaadat tárolására.

4 Amazon Dynamo

Az Amazon több belső elosztott rendszert is üzemeltet: egy részük az `amazon.com` online boltot szolgálja ki, egy másik részük az Amazon Web Services (AWS) rendszert alkotják. Itt az online boltnál használt Dynamo rendszert mutatjuk be röviden a 2007-ben kiadott cikk alapján [8].

A Dynamo egy gyengén konzisztens rendszer, melynek a célja, hogy minden esetben kiszolgálja a kliens kéréseit — akkor is, ha nem teljesen friss adattal tud dolgozni valamilyen hiba vagy hálózati partíció miatt. Ezt az online bolt követeli meg, melynek mindig működnie kell ("always-on experience"), ti. ha nem működik, akkor jól becsülhető, lényeges pénzügyi veszteséget szenved a cég. A rendszer kulcs-érték alapon működik, a kulcs-érték párok többszörösen replikálva vannak, ahol a gyenge konzisztencia miatt ugyanazon adat több verziója lehet a rendszerben, így az adatok családfaszerűen verziókkal vannak ellátva. Amennyiben egy adatnak több verziója van jelen, azt előbb-utóbb észleli a rendszer, és alkalmazás-specifikus konfkliktus feloldó algoritmus újra előállít egy konzisztens elosztott állapotot. Ezért ezt a modellt *eventual consistency*-nek is hívják (kb. "előbb-utóbb konzisztens lesz").

Például, tegyük fel hogy az `mtrencseni` vásárlónak két könyv van a kosarában, A és B. A vásárlás folyamata közben a felhasználó kosarát tároló szervek hálózati hiba miatt lekapcsolódnak, ezért a rendszer nem éri el a kosár legutolsó állapotát, így a rendszer a kosarat üresnek jelzi a felhasználónak. A felhasználó érzékeli a hibát, és újra belerakja az A és B könyvet, majd kicsit később egy új C könyvet. Közben a hálózati hiba helyreáll, és a rendszer érzékeli, hogy a felhasználónak két különböző verziójú kosara van a rendszerben. Ilyenkor egy alkalmazás (kosár alkalmazás) specifikus konfliktus feloldó specifikus algoritmus előállít egy áj, konzisztens állapotot, pl. a kosarak unióját képzi. A vásárló úgy is ellenőrzi a kosarát fizetés előtt.

5 Scalien Keyspace

A saját készítésű, kulcs-érték alapú Keyspace adatbázisrendszer az első nyílt forráskódú adattároló amit bemutatunk. A Keyspace az eddig bemutatott rendszerek közül a leginkább a Google Chubby rendszeréhez hasonlít. A Keyspace egy konzisztensen replikált adatbázis: replikált, mert az összes szerver ugyanazt az adatot tárolja; konzisztens, mert a CAP háromszögben a konzisztenciára helyezi a hangsúlyt (vs. "eventual consistency"), és garantálja, hogy sikeres írások után az olvasások tükrözik az írást, akármilyen hálózati vagy szerver hiba esetén is.

Hasonlóan a Chubby-hoz a Keyspace is a Lamport-féle Paxos algoritmust valósítja meg, mely egy többségi algoritmus; a Keyspace cellákat $n = 3$ konfigurációban futtatva, pl. egy szerveres 95%-os rendelkezésre állás 99.27%-ra javítható (ld. táblázat).

szerverek	többség	rendelkezésre állás
1	1	95.00%
2	2	90.25%
3	**2**	**99.27%**
4	3	98.59%
5	3	99.88%
..	..	..

Table 1: Rendelkezésre állás különböző méretű Keyspace cellák esetében.

A rendszer lelkét alkotó elosztott algoritmus, a Paxos miatt a Keyspace minden praktikusan előálló hálózati vagy szerver hiba esetet kezel, és tovább üzemel, amennyiben a szerverek többsége rendelkezésre áll és kommunikál:

1. Szerverek leállnak és újraindulnak: a Keyspace programot futtató szerverek leállhatnak és újraindulhatnak, elveszítve a memóriában tárolt állapotot, de nem a diszkre kiírt adatokat.
2. Hálózati partíciók: hubok és routerek tönkremehetnek, a hálózat átmeneti részekre esését okozva.
3. Csomgagveszteség, duplikáció és átrendeződés: Operációs rendszerek hálózati stackje és routerek eldobhatnak és átrendezhetnek üzeneteket. A TCP-szerű protokollok garantálják ezen esetek kezelését, míg az UDP-szerűek nem. A Keyspace mindkét fajta hálózati protokoll fölött tud futni.
4. Hálózati késleltetések: terhelt helyi hálózatokon és WAN-okon mint az Internet az üzenetek több másodperces késéssel érkezhetnek meg a címzetthez.

A Keyspace más kulcs-érték adatbázisokhoz képest viszonylag kiterjedt adathozzáférési API-val rendelkezik, mivel támogat plusz atomi műveleteket mint `TESTANDSET` és `ADD`, illetve listázási műveleteket. Ezen kívűl az olvasási műveleteknek létezik piszkos ("dirty") verziója is, mely semmilyen konzisztencia garanciát nem ad, viszont akár egyedülálló szerver is ki tudja szolgálni.

- `GET(key)`: visszaadja a `key`-hez tartozó értéket, ha létezik az adatbázisban.
- `SET(key, value)`: beállítja a `key` értékét, átírva az előző értéket ha létezett az adatbázisban.

- `TEST-AND-SET(key, test, value)`: atomi módon átírja `key` értékét `value`-ra, ha a jelenlegi értéke `test`.
- `ADD(key, a)`: a `key` értéket számként értelmezi és hozzáad `a`-t.
- `RENAME(key, newKey)`: átnevezi `key`-t `newKey`-re, megtartva az értékét.
- `DELETE(key)`: kitörli `key`-t és az értékét az adatbázisból.
- `REMOVE(key)`: kitörli `key`-t és az értékét az adatbázisból, visszaadja az értéket.
- `PRUNE(prefix)`: kitörli az összes kulcs-érték párt amely `prefix`-szel kezdődik.
- `LIST-KEYS(prefix, startKey, count, next, forward)`: legfeljebb `count` kulcsot ad vissza, melyek `prefix`-szel kezdődnek, a `startKey` kulcstól indulva. Amennyiben a `startKey` kulcs nem létezik az adatbázisban, lexikografikusan a következő kulcsnál kezdődik. Amennyiben `startKey` létezik, átugorható `next = true` beadásával. Ez webes "lapozott" oldalak előállításánál hasznos.
- `LIST-KEYVALUES(prefix, startKey, count, next, forward)`: ugyanaz, mint `LIST-KEYS`, de a kulcsokon kívűl az értékeket is visszaadja.
- `COUNT(prefix, startKey, count, next, forward, forward)`: visszaadja a kulcsok számát, melyeket az ugyanezen paraméterekkel meghívott `LIST` adna vissza.
- `DIRTY-GET(key)`: mint az előző `GET`, de konzisztencia garanciák nélkül.
- `DIRTY-LIST-KEYS(prefix, startKey, count, next, forward)`: mint az előző `LIST-KEYS`, de konzisztencia garanciák nélkül.
- `DIRTY-LIST-KEYVALUES(prefix, startKey, count, next, forward)`: mint az előző `DIRTY-LIST-KEYVALUES`, de konzisztencia garanciák nélkül.
- `DIRTY-COUNT(prefix, startKey, count, next, forward)`: mint az előző `COUNT`, de konzisztencia garanciák nélkül.

A Keyspace adatbázist saját, nagy hatékonyságú protokollon ill. adminisztációs és tesztelési célból HTTP illetve HTTP+JSON API-n keresztül lehet elérni. A nagyhatékonyságú aszinkron architektúra miatt a Keyspace nagy számú konkurrens műveletet tud kiszolgálni (ld. köv. ábra).

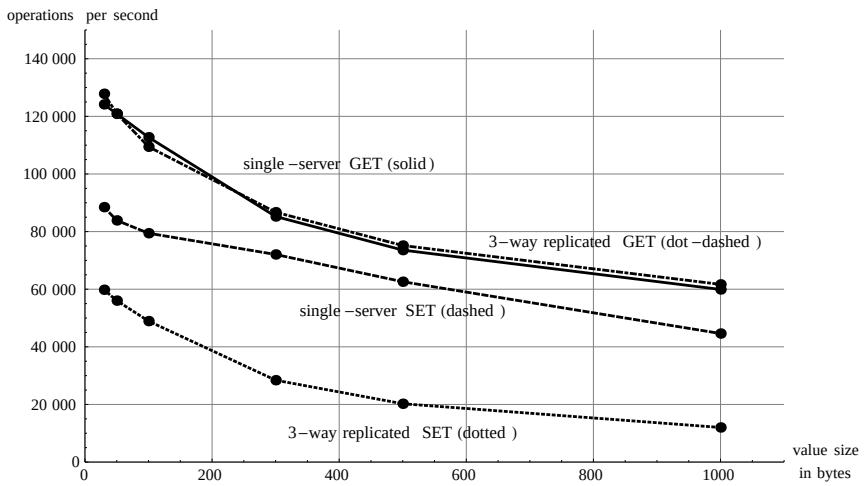

Figure 2: Keyspace bulk adatátviteli sebességek.

6 Konklúzió

References

[1] M. Stonebraker. *The Case for Shared Nothing*, Database Engineering, Volume 9, Number 1 (1986).

[2] E. Brewer. *Keynote Address*, Symposium on Principles of Distributed Computing (2000).

[3] L. Lamport, *Paxos Made Simple*, ACM SIGACT News 32, 4 (Dec. 2001), pp. 18-25.

[4] M. Burrows, *The Chubby Lock Service for Loosely-Coupled Distributed Systems*, OSDI '06: Seventh Symposium on Operating System Design and Implementation.

[5] S. Ghemawat, H. Gobioff, S. Leung, *The Google File System*, 19th ACM Symposium on Operating Systems Principles (2003).

[6] J. Dean, S. Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, OSDI'04: Sixth Symposium on Operating System Design and Implementation (2004).

[7] F. Chang et al., *Bigtable: A Distributed Storage System for Structured Data*, OSDI'06: Seventh Symposium on Operating System Design and Implementation (2006).

[8] W. Vogels et al., *Dynamo: Amazon's Highly Available Key-value store*, SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (2007), pp. 205-220.

[9] M. Trencseni, A. Gazso *Keyspace: A Consistently Replicated, Highly-Available Key-Value Store*, `http://scalien.com/whitepapers`.