Osztott rendszerű hasítótáblák (1. rész)

Az egyenrangú hálózatok világában az osztott rendszerű hasítótáblák (Distributed Hash Tables, DHT) forradalmi újítást jelentettek. Tanuljuk meg, hogyan írhatunk olyan alkalmazást, amely lehetővé teszi, hogy mindenki azonos adatokon dolgozzon.

Az első nemzedékbeli egyenrangú hálózatok (peer-to-peer) áttekinthetetlen, ötletszerűen felépített szerkezete helyett ma már egyre gyakrabban alkalmaznak kiváló teljesítményű, hasznos tulajdonságokkal rendelkező, tervezett topológiakészleteket.

Számos olyan kísérleti DHT-t készítettek különböző egyetemeken, amelyeket a nyílt forráskód közössége felkarolt és megvalósított. Létezik néhány kereskedelmi alkalmazás is, de jelenleg egyik sem érhető el SDK (programfejlesztő készlet) formában, hanem valamelyik boltokban kapható termékbe építve találjuk meg őket. Minden DHT-megoldást elképzelhetünk minden más megoldástól eltérő, önálló entitásként is. Így tulajdonképpen valamennyi létező megoldást befoglalhatnánk egy többdimenziós mátrixba. Kiválasztunk egyet, elvégzünk néhány módosítást, és máris egy másiknál járunk. A már létező kutatói DHT-k, amilyen a Chord, Kademlia vagy a Pastry, így aztán kiváló kiindulási alapként szolgálnak saját fejlesztéseinkhez.

A DHT-k lényegében egy hasítótábla feladatát látják el: kulcs és érték párokat tárolhatunk bennük. Ha ismerjük a kulcsot, megkaphatjuk az értéket. Az értékeknek nem feltétlenül kell a lemezen létezniük, de DHT rendszerünket természetesen valamilyen állandó hasítótáblára alapozhatjuk, például a Berkeley DB-re; ami egyébként ténylegesen el is készült.

A DHT rendszerekben az a különleges, hogy a tároló lekérdezései több gép között oszlanak meg. A már létező elsődleges-másodlagos kiszolgáló alapú adatbázis-szerkezetekkel ellentétben itt minden csomópont egyenértékű és szabadon csatlakozhat, illetve hagyhatja el a hálózatot. Annak ellenére, hogy a hálózat tagjai látszólag áttekinthetetlen módon, véletlenszerűen változnak, a DHT-k igen meggyőző teljesítményt mutatnak.

Kezdjük a DHT-tervezés rejtelmeinek felfedezését egy egyszerű körkörös, kétszeresen láncolt listával. A lista minden egyes csomópontja egy-egy gépet jelent a hálózaton. Minden csomópont az előtte és az utána következő elemről is tárol adatot, mégpedig az adott gép címét. Meg kell határoznunk valamiféle sorrendet, ezért minden csomópontnál megadjuk, hogy melyik lesz a „következő” csomópont. A Chord DHT által alkalmazott módszer szerint a következő csomópontot ilyen módon kapjuk meg: minden egyes csomóponthoz rendeljünk egyedi, k bites azonosítót. A gyűrű elemeit olyan módon rendezzük, hogy az azonosítók óramutató járásával megegyező irányban növekedjenek. Minden csomópont számára az lesz a következő, amelyiknek az azonosítója órairányban a legkisebb távolságra áll tőle. A legtöbb csomópont esetében ez az a csomópont lesz, amelyik a legkisebb, ugyanakkor nagyobb azonosítóval rendelkezik az adott csomóponthoz képest.

Ez alól csak a legnagyobb azonosítójú csomópont a kivétel, ennek az utódja ugyanis éppen a legkisebb azonosítóval rendelkezik. Ezt a távolságmértéket a távolságmódszer valamivel helyesebben adja meg (lásd az *1. listát*).

Minden egyes csomópont önmagában egy szabványos hasítótábla. Ha egy értékre van szükségünk, mindössze annyi a feladatunk, hogy rátaláljunk a megfelelő csomópontra, innen már a hagyományos hasítótábla-keresést vagy -tárolást kell csak elvégeznünk. Az adott kulcshoz tartozó csomópont keresésének egyik egyszerű (és a Chord által is használt) módszere, ha ugyanazt tesszük, mint az adott azonosító leszármazottainak a keresésénél. Először is vegyük a kulcsot, majd egy kereső alapján rendeljünk hozzá valamilyen pontosan k bites másik kulcsot. Tekintsük ezt a számot a csomópont-azonosítónak, majd állapítsuk meg, hogy melyik csomópont lesz a leszárma-

1. lista A ringDistance.py

```
# Ez az órajárással megegyező gyűrű
# távolságfüggvénye. A globálisan megadott k
# értéktől, a kulcsmérettől függ a legnagyobb
# lehetséges csomópont-azonosító 2**k.
def distance(a, b):
    if a==b:
        return 0
    elif a<b:
        return b-a;
    else:
        return (2**k)+(b-a);
```

2. lista A findNode.py

```
# A kezdő csomóponttól indulva keressük meg
# a célkulcsért felelős csomópontot.
def findNode(start, key):
    current=start
    while distance(current.id, key)
    ➥> distance(current.next.id, key):
        current=current.next
    return current

# Keresd a megfelelő csomópontot és kérd
# le a kulcshoz tartozó értéket.
def lookup(start, key):
    node=findNode(start, key)
    return node.data[key]

# Keresd a megfelelő csomópontot és tárold
# az értéket a kulccsal.
def store(start, key, value):
    node=findNode(start, key)
    node.data[key]=value
```

3. lista Az update.py

```
def update(node):
    for x in range(k):
        oldEntry=node.finger[x]
        node.finger[x]=findNode(oldEntry,
(node.id+(2**x)) % (2**k))
```

4. lista A finger-lookup.py

```
def findFinger(node, key):
    current=node
    for x in range(k):
        if distance(current.id, key)
        ➥> distance(node.finger[x].id, key):
            current=node.finger[x]
    return current

def lookup(start, key):
    current=findFinger(start, key)
    next=findFinger(current, key)
    while distance(current.id, key)
    ➥> distance(next.id, key):
        current=next
        next=findFinger(current, key)
    return current
```

zottja, azaz a gyűrű bármely pontjáról indulva az óramutató járásával megegyező irányba gyalogoljunk, addig, amíg rá nem akadunk arra csomópontra, amelynek az azonosítója a legközelebb van, de már nagyobb, mint az adott szám. Az imént megtalált csomópont lesz a felelős az adott kulcs tárolásáért és visszakereséséért (lásd a 2. *listát*). A kulcs készítéséhez érdemes hasítótáblát használnunk, hiszen a hasítótábla általában egyenletes eloszlást készít, így a különféle kulcsok a hálózat csomópontjai közt egyenletesen oszlanak majd meg.

Ez a DHT-kialakítás egyszerű és egyben minimálisan szükséges is, amennyiben osztott rendszerű hasítótáblát szeretnénk létrehozni. Feltéve, hogy a rendelkezésre állással bíró csomópontok statikus hálózatát tekintjük, és az adott kulcshoz tartozó csomópontot keressük, bármilyen kulccsal és csomóponttal kezdhetünk. Fontos azonban észben tartanunk, hogy bár a példakód úgy néz ki, mintha egyszerű, kettős kapcsolatú listát kellene megjegyeznünk, ez csak egy DHT-szimuláció. Egy valódi DHT esetében minden egyes csomópont külön gép lenne, és minden egyes hívást valamiféle foglalat- (socket) protokoll segítségével kellene megvalósítani.

Ha egy kicsit hasznosabbá akarjuk formálni a modellünket, jó lenne feljegyezni, hogy mely csomópontok hagyták el a hálózatot, illetve melyek csatlakoznak hozzá akár szándékosan, akár valamilyen hiba miatt. A képesség beépítéséhez hálózatunkban be kell vezetnünk valamilyen csatlakozó-, illetve elhagyóprotokollt. A Chord csatlakozóprotokoll első lépése, hogy a hagyományos keresési szabály alapján kikeressük az új csomópont azonosítójának az utódját. Az új csomópontot e közé a leszármazott csomópont és annak elődje közé kell beszúrnunk. Az új csomópont elődje keresőszerkezetének egy részéért lesz felelős. Mivel azt szeretnénk, ha minden lekérdezés hibátlanul menne végbe, a kulcsok adott tartományát át kell másolnunk az új csomópontra, mielőtt az elődcsomópont a következő csomópont mutatóját új csomópontunkra változtatná át.

A kilépés nagyon egyszerű: a kilépő csomópont minden adatát az elődjére másolja. Az előd ezután a következő mutatóját a kilépő csomópont utódjára állítja át. A belépő és kilépő kód nagyon hasonlít a hagyományos láncolt listák törlő és beszúró utasításaihoz, kiegészítve a ki- és belépők, valamint a szomszédaik közötti adatcsere-forgalommal. Egy hagyományos láncolt listában éppen azért törlünk egy elemet, mert a benne tárolt adatot meg akarjuk szüntetni. A DHT-k esetében a csomópontok beszúrása és eltávolítása teljesen független az adatok beszúrásától és törlésétől. A DHT-csomópontokat elképzelhetjük úgy is, mint bizonyos állandó hasítótábla-megoldások időnként kiigazított vödreit (bucket), amilyeneket például a Berkeley DB is alkalmaz.

Az, hogy hálózatunkat dinamikus csomópontfelvételre és -kiadásra tettük képessé, miközben a tárolás és a lekérés továbbra is folyamatos, egyértelműen nagy fejlődés. Ugyanakkor a teljesítmény borzalmas – `O(n)` mégpedig `n/2` várható teljesítménnyel. Minden egyes közbenlévő csomópont a hálózat egy másik tagjával hoz létre kapcsolatot, amelyhez (a kiválasztott átviteltől függően) valószínűleg TCP/IP-kapcsolat szükséges. Így aztán az `n/2` csomóponton áthaladva elég lassú lehet.

Hogy ennél valamivel jobb teljesítményt érjen el, a Chord modell egy réteget vezet be, amellyel `O(log n)` teljesítmény érhető el. Ahelyett, hogy egyszerűen csak a következő csomópontot azonosító mutatót tárolnánk, minden csomópont egy `finger table`-t tartalmaz, amelyben k csomópont címe található meg. A jelenlegi csomópont azonosítója és a `finger table`-ban található csomópontok azonosítója közötti távolság exponenciálisan növekszik. Egy adott kulcshoz vezető ösvényen minden csomópont, amin csak áthaladunk, logaritmikusan közelebb lesz, mint az előző, így végül összesen `O(log n)` csomóponton haladunk keresztül.

Hogy a logaritmikus keresések jól működhessenek, az `finger table`-nek mindig frissnek kell lennie. Az elavult táblák ugyan nem rontják el a keresést, amíg minden csomópont friss „következő” mezővel rendelkezik, de a keresés csak akkor lesz logaritmikus, ha a táblák helyesen vannak felépítve. A `finger table` frissítéséhez a tábla k helyére egy-egy címet kell írnunk. Bármely x (ahol x 1-től k-ig terjedhet) helyhez tartozó `finger[x]` értéket úgy kapjuk meg, hogy vesszük a jelenlegi csomópont azonosítóját, és kikeressük az `(id+2(x-1)) mod (2k)` kulcsért felelős csomópont címét (*3. lista*). Amikor visszakeresünk, egyetlen lehetőség helyett minden ugrásnál k elemből választhatunk. Minden egyes csomópontnál, amit csak meglátogatunk, azt a bejegyzést választjuk az `finger table`-ben, amelyik a legközelebb esik a keresett elemhez (*4. lista*).

Eddig többé-kevésbé sikerült megadnunk a MIT csapat által elképzelt és kifejlesztett eredeti Chord DHT-tervet. Ez csak a DHT-jéghegy csúcsa. Rengeteg módosítást el lehet még végezni, amelyek eltérnek az eredeti Chord-leírásban bemutatott képességektől, miközben nem veszítjük el a logaritmikus teljesítményt és a Chord nyújtotta keresési biztonságot.

Az egyik ilyen képesség, ami DHT rendszerünkben jól jöhet az, ha lehetővé tesszük az `finger table` passzív frissítését, hiszen a táblák frissítéséhez időnként ismétlődő keresések szükségesek. A MIT Chordjában a tábla minden k elemére `O(log n)` csomópontot kell elérnünk, ami jelentős hálózati forgalmat is jelenthet. Előnyös lenne, ha a csomópontok fel tudnának venni más csomópontokat a saját `finger table`-jükbe, amikor valamilyen keresés miatt csatlakoznak hozzá.

5. lista Az xor-distance.py

```
def distance(a, b):
    return a^b # Python alatt, a művelet a
               # XOR b-t jelent, nem pedig
               # a szám b-edik hatványát.
```

Minthogy a keresés végrehajtása miatt a párbeszéd már egyébként is megindult, nem jelent túl sok pluszmunkát ellenőrizni, hogy a keresést végző csomópont jó jelölt-e a helyi `finger table`-ba. A Chord `finger table` hivatkozásai sajnos irány nélküliek, mivel a távolságmérték nem szimmetrikus. Egy csomópont nem feltétlenül szerepel a `finger table`-jében található csomópontok finger tábláiban.

A nehézség egyik megoldása lehet, ha a Chord moduláris összeadó távolságmértékét `XOR` alapú távmértékre cseréljük. Két csomópont, mondjuk A és B közötti távolságnak ettől kezdve a csomópont-azonosítóknak megfelelő előjel nélküli egész számok közötti `XOR` művelet eredményét tekintjük (*5. lista*). A `XOR` igen jó távolságmérték, hiszen szimmetrikus. Mivel bármely két csomópontra távolság(A, B) == távolság(B, A), ha A megtalálható B `finger table`-jében, akkor B is megtalálható A `finger table`-jében. Ez viszont azt jelenti, hogy a csomópontok frissíthetik a `finger table`-jüket az őket lekérdező csomópontok címe alapján, jelentősen csökkentve ezáltal a csomópontfrissítéssel járó hálózati forgalmat. Ez egyúttal egyszerűsíti a DHT-alkalmazás programozását, hiszen nem kell egy külön szálat fenntartani a frissítési folyamat ismételt meghívására. Ehelyett egyszerűen csak akkor frissítünk, amikor a keresési függvény meghívódik.

Az eddig bemutatott felépítéssel akadt egy kis gond, nevezetesen, hogy az adott csomóponthoz vezető ösvény igen ingatag. Ha az ösvényen található bármelyik csomópont visszautasítja az együttműködést, a keresés megakad. Bármely két csomópont között pontosan egy ösvény van, így a lerobbant csomópontok között lehetetlen megtalálni az utat. A Kademlia DHT ezt úgy oldja meg, hogy kiegészíti az `finger table`-t, úgy, hogy minden egyes bejegyzés egy helyett j csomópont-hivatkozást tartalmaz, ahol j a teljes hálózatban azonosan van meghatározva. Ettől kezdve minden ugrásra j különböző lehetőség nyílik, így körülbelül `j*log(n)` lehetséges ösvény létezik. Ennél kevesebb létezik, hiszen az ösvények összetartanak, ahogy egyre közelebb kerülnek a célhoz. Mindazonáltal a lehetséges ösvények száma valószínűleg nagyobb mint 1, ami mindenképpen fejlődés.

A Kademlia ennél is továbbmegy, és a vödrökben (bucket) a feljegyzett működési idő (uptime) alapján rendezi a csomópontokat. A régebbi csomópontok kereséseknél előnyben részesülnek, és új hivatkozások csak akkor kerülnek a rendszerbe, ha régiből már nincsen elegendő. A lekérdezések növekvő biztonsága mellett ez a megközelítés egy további előnnyel is jár: nem lehet úgy megtámadni a hálózatot, hogy gyors ütemben készítenek új csomópontokat, megpróbálván kiütni a jó elemeket – a hatást még csak észre se lehet majd venni.

Tudni kell, hogy a fent bemutatott megoldások nem egyetlen DHT-megvalósításhoz kötődnek. Lépésenként építettünk fel egy DHT-tervet az alapoktól kezdve, Chord-szerűvé formáltuk, aztán úgy módosítottuk, hogy inkább a Kademliára hasonlítson. A különféle megközelítések többé-kevésbé összekeverhetők és párosíthatók. A `finger table`-jeink vödreiben lehet egy vagy j hely, attól függően, hogy távolságmértékünkhöz moduláris összeadást vagy `XOR` műveletet használunk-e. Követhetjük mindig a legközelebbi csomópontot, vagy rangsorolhatjuk őket a fennállási idő vagy bármely más érték alapján. Meríthetünk más DHT-elképzelésekből is, például a Pastryból, OceanStore-ból vagy a Coralból. Természetesen a saját ötleteinket is használhatjuk a saját igényeink kielégítésére. A magam részéről én olyan képességekkel egészítettem ki a Chord-elképzelést, mint az anonimitás, Byzantine hibatűrő keresések, geográfiai útkeresés és az üzenetek hatékony szórása belépéskor. Érdekes megcsinálni, és sokkal könnyebb is, mint gondolnánk.

Most, hogy már tudjuk, hogyan kell saját DHT-megoldásokat készíteni, biztosan kíváncsiak vagyunk, vajon milyen őrült dolgokat lehet tenni egy ilyen kóddal. Valószínűleg rengeteg olyan DHT-alkalmazás létezik, amit még fel sem fedeztek, de már most is hallani emberekről, akik olyan feladatokon dolgoznak, mint fájlmegosztás megosztott merevlemez adatmentéshez, DNS helyettesítése egyenrangú hálózati névfeloldó rendszerrel, méretezhető csevegő és kiszolgáló nélküli játékok.

A cikkhez megpróbáltam egy mókás kis példaalkalmazást is összerakni, ami esetleg felkeltheti azoknak a figyelmét, akik látták a Linux Journal weblapján megjelent bemutatómat az egyenrangú (peer-to-peer) Superworms témáról (lásd a *Kapcsolódó címeket*). Az alkalmazás egy osztott kapupásztázó (port scanner), amely az eredményeket egy szimulált DHT-ban tárolja (6. lista; 54 CD Magazin/Hash könyvtára). Ha ténylegesen működő DHT-megoldás lenne, a parancsfájlnak néhány igazán érdekes vonása is lenne. Először is lehetővé tenné, hogy internetpásztázásának eredményeit több gép is közreadja. Ezzel a módszerrel a pásztázás többé már nem kapcsolható egyetlen géphez; továbbá elkerüli a redundáns vizsgálódást. Ha a gép már egyszer át lett vizsgálva, az eredményeket a DHT-ból kapjuk meg, elkerülve a többszörös vizsgálódást. Az adatok tárolásához nincsen szükségünk központi kiszolgálóra, az összes eredmény, illetve a tevékenység szervezése a résztvevőkön múlik. Az alkalmazás némiképpen alattomosnak tűnik, de igazság szerint a DHT könyvtár ismeretében elég nyilvánvaló volt megírása. Ugyanilyen megközelítés alapján készülhetnek el az osztott projektek is.

Kétrészes sorozatunk mostani részében áttekintettük a DHT-k mögött megbúvó elméletet. Legközelebb inkább a DHT használatának gyakorlati előnyeit vizsgáljuk meg valódi alkalmazásokban.

A cikkhez kapcsolódó listák megtalálhatóak az 54. CD Magazin/Hash könyvtárában.

Linux Journal 2003. október, 114. szám

Brandon Wiley (blanu@decentralize.org)
Az egyenrangú hálózatok betyára (peer-to-peer hacker) és a Foundation for Decentralization Research elnöke.

KAPCSOLÓDÓ CÍMEK

Achord ➲ http://thalassocracy.org/achord/achord-iptps.html
Chord ➲ http://www.pdos.lcs.mit.edu/chord
Curious Yellow ➲ http://blanu.net/curious_yellow.html
How Can You Defend against a Superworm? ➲ http://www.linuxjournal.com/article/6069
Kademlia ➲ http://kademlia.scs.cs.nyu.edun