Szegedi Tudományegyetem

Informatikai Tanszékcsoport

A BuddyCast-alapú P2P ajánlórendszer kiértékelése

Szakdolgozat

Készítette:
Csernai Kornél
programtervező informatika szakos hallgató

Témavezető:
Dr. Jelasity Márk
tudományos főmunkatárs

Szeged
2010

Tartalomjegyzék

Feladatkiírás 4
Tartalmi összefoglaló 5
Bevezetés 6

1. Peer-to-Peer hálózatok **7**
1.1. A Peer-to-Peer paradigma 7
1.2. Jelentőség 7
1.3. Fogalmak 8
1.3.1. Peer 8
1.3.2. Overlay hálózat 8
1.3.3. Churn 8
1.3.4. Csatlakozhatóság 9
1.3.5. Tűzfal 9
1.3.6. NAT 9
1.4. Keresés 11
1.4.1. Strukturált és strukturálatlan hálózatok 11
1.5. Fájlcserélő hálózatok 12
1.5.1. Gnutella 12
1.5.2. Napster 12
1.5.3. BitTorrent 12

2. Pletyka protokollok **15**
2.1. Bevezető 15
2.1.1. A protokoll váza 15
2.1.2. Bootstrapping 16
2.2. Peer mintavételezés 17
2.2.1. A lokális információ 17
2.2.2. A protokoll váza 17
2.2.3. Néhány lehetséges megvalósítás 18
2.3. Aggregáció egy hálózat fölött 20
2.3.1. Átlag számítása 20
2.3.2. További aggregációk 20
2.4. Overlay menedzsment: T-Man 21
2.4.1. A topológia felépítés feladata 21
2.4.2. Távolság alapú T-Man 21
2.4.3. A T-Man algoritmus váza 22
2.5. NewsCast 23

2.5.1. Az algoritmus váza 23

3. BuddyCast és implementálása **25**
3.1. A protokoll részletes leírása 26
3.1.1. A lokális információ 26
3.2. Változtatható paraméterek 29
3.3. PeerSim implementáció 29
3.3.1. PeerSim 29
3.3.2. Ciklus illetve esemény alapú szimuláció 30
3.3.3. A megvalósítás technikai részletei 30
3.3.4. Forráskód 31

4. A BuddyCast algoritmus kiértékelése ajánlórendszereken **32**
4.1. Ajánlórendszerek 32
4.2. A felhasznált tanuló adatbázisok jellemzése 33
4.3. A mérések 33

5. Függelék **37**

Nyilatkozat 38
Köszönetnyilvánítás 39

Feladatkiírás

A peer mintavételezés a nagyméretű dinamikus hálózatok legalsó rétegeként fogható fel. A mintavételezés során a résztvevők a hálózatból másik véletlen résztvevők (peer) címeit kapják meg. A mintavételezési funkciónak mindig együtt kell működni valamilyen alkalmazással, pl. információ terjesztés, elosztott adatbányászat, keresés, stb. A feladat egy konkrét alkalmazás kiválasztása után az alkalmazás és a mintavételezés kölcsönhatásának az elemzése, szimuláció útján. Ez a kölcsönhatás valószínűleg sok meglepetést tartogat és általában elég komplex lehet, mivel mind az alkalmazás mind pedig a mintavétel implementációja teljesen elosztott. Egy mintavételező algoritmus például a BuddyCast.

Tartalmi összefoglaló

A szakdolgozat a BuddyCast algoritmus szimulációjának lehetőségére összpontosít. A BuddyCast egy overlay hálózatot menedzselő pletyka alapú algoritmus. A feladat alapján a protokollt a PeerSim szimulátor programban valósítottam meg, amely egy jól skálázódó, jól strukturált, Java programozási nyelvben írt tudományos körökben ismert szimulátor.

A megvalósítás esemény alapú lett, ezáltal valósághűbb futásokat kapunk. Hátránya, hogy rosszabbul skálázódik, mintha ciklusos megvalósítást választottunk volna.

A protokollt korrekt módon kellett megvalósítani, úgy, hogy hű maradjon a specifikációjához. Egyes elhanyagolható részletekről nem rendelkezett a protokoll, ezekről a kérdésekről következetesen döntöttem.

A megvalósítás tisztán Java nyelvű. A fejlesztéshez a NetBeans IDE-t használtam Linux operációs rendszeren, Java 1.6 környezetben. A szimulációkat egy nagy teljesítményű szerver gépen futtattam.

Az eredmények rámutatnak arra, hogy a BuddyCast algoritmus jól konvergál az elvárt offline értékekhez mind a három használt adatbázis esetében, azonban a terhelés az egyik adatbázis esetében annak ritkasága miatt nem elfogadható mértékeket ölt.

Kulcsszavak: buddycast, P2P, peersim, ajánlórendszerek, collaborative filtering

Bevezetés

Napjainkban egyre nagyobb teret hódítanak a Peer-to-Peer (P2P) hálózatok, amelyekben nincsenek erősen kitüntetett számítógépek, minden résztvevő erőforrást ad a rendszerhez. Ezek a hálózatok jól skálázódnak, robosztusak, és nincs egyetlen meghibásodási pontjuk. A P2P hálózatok egyre nagyobb felhasználóbázissal rendelkeznek, egyes rendszerek több millió felhasználóval rendelkeznek. A P2P rendszereknek sok alkalmazása van, ilyen például a fájlcserélés, video közvetítés, GRID rendszerek. A fejlettebb botnetek is P2P rendszert alkotnak.

A P2P rendszerek sok téren különböznek a kliens-szerver rendszerektől. Ezekkel számolni kell egy P2P algoritmus tervezésekor és kiértékelésekor.

Ebben a szakdolgozatban a Tribler nevű P2P tartalomközvetítő rendszerben kiépített BuddyCast pletyka alapú algoritmus szimulálását tűztem ki célul. A BuddyCast algoritmus egyik fő célja, hogy a hálózat strukturáját egy hasonlósági függvény szerint kedvező módon alakítsa.

A szimulálás előtt bevezetést nyújtok az ajánlórendszerekbe. Az ajánlórendszereken lényegében egy gépi tanulásos feladatot oldok meg, azonban nem központosított környezetben, hanem elosztott módon. A BuddyCast algoritmus definiálja az overlay hálózatot, amelyen egy ismert aggregáció szerinti collaborative filtering predikciót értékelek ki.

A méréshez három adatbázist használtam. Az eredmények rámutatnak arra, hogy a BuddyCast algoritmus jól konvergál az elvárt offline értékekhez mind a három esetben, azonban a terhelés az egyik adatbázis esetében annak ritkasága miatt nem elfogadható mértékeket ölt.

1. fejezet

Peer-to-Peer hálózatok

1.1. A Peer-to-Peer paradigma

Napjainkban a számítógépek közti kommunikáció, adatcsere és a számítási feladatok összetett környezetben történnek. Ilyen környezetek a számítógépes hálózatok, ahol az egyes gépek saját feladatkörrel rendelkeznek. A számítógépek egymásnak üzeneteket tudnak címezni, amely egy útvonalon továbbítódik.

Egy lehetséges felépítést (topológia) jelent a *kliens-szerver* (*client-server*) modell, amelyben kétféle szerep van. A szerver egy vagy több kitüntetett számítógép, melynek feladata, hogy kliensek felől érkező kéréseket kiszolgálja. Fontos, hogy szervernek képesnek kell lennie a rá háruló összes kérést kiszolgálnia. A szervertől tehát az összes kliens működése függ. Ha a szerver meghibásodik, az egész rendszer leáll.

Ezzel szemben a Peer-to-Peer (röviden: P2P) szerinti modellben minden számítógép viselkedhet szerverként és kliensként is. Nincs tehát kitüntetett csomópont, így nincs is egyetlen fő meghibásodási pont. A csomópontok közötti kommunikáció megvalósítása változatosabb, és több tervezést igényel, mint a kliens-szerver esetben.

Számos protokoll működik kliens-szerver környezetben, így például a web (http, https), levelezés (smtp, pop3, imap), fájlcsere (ftp, sftp), audió és videó közvetítések, azonnali üzenetküldés, stb…

Kérdés, hogy ezek közül melyeket lehet P2P rendszerben hatékonyan végezni, esetleg hatékonyabban, mint a kliens-szerver környezetben.

Egy tipikus feladat a keresés: egy adatbázisban keresünk egy rekordot. Kliens-szerver esetben nincs más dolga a kliensnek, mint lekérdezni a szerverről az értéket, aki a keresést elvégezni a központi adatbázisában. Elosztott esetben változik a helyzet, ugyanis legtöbb esetben nincs lehetőségünk minden egyes számítógépet megkeresni.

1.2. Jelentőség

A P2P hálózatok napjainkban igen nagy jelentőséggel bírnak. Nap mint nap rengeteg felhasználói él a P2P nyújtotta lehetőségekkel, melyek közül a legtöbb forgalmat a fájlcserélés generálja. Az internet teljes forgalmának igen nagy része P2P alapú forgalom (elsősorban fájlcserélő programok)[31].

A nagy felhasználói bázis miatt célszerű a használt algoritmusokat úgy megtervezni,

hogy azok jól skálázódjanak, és minél takarékosabban bánjanak az erőforrásokkal. A felhasználói élményt maximalizálni kell: a felhasználó a lehető leghamarabb kapja meg az eredményt, a rendszer megbízható és biztonságos legyen.

1.3. Fogalmak

Ahhoz, hogy a P2P hálózatokkal tudjunk foglalkozni, ismernünk kell az ide tartozó fogalmakat. Ebben a részben ismertetem ezek közül a legfontosabbakat.

1.3.1. Peer

A peerek a P2P hálózatok különálló alapegységei, amelyek adott mennyiségű erőforrással és bizonyos tulajdonságokkal rendelkeznek.

Ezek lehetnek:

- sávszélesség, válaszidő
- tároló kapacitás
- számítási kapacitás

Bizonyos szövegkörnyezetben a *csomópont* (*node*) kifejezést használjuk a peer helyett.

A tisztán P2P környezetben – a kliens-szerver modellel ellentétben – a peereket nem irányítja központi egység, így nem is függnek tőle.

A *superpeer* egy kitüntetett peer, amelyet minden más peer el tud érni. Egyes protokollok megkövetelik, hogy a rendszer felállása során (*bootstrapping*) elérhetőek legyenek superpeerek, akik az első szomszédai az újonnan csatlakozó peereknek.

Egy p peer szomszédai (*neighbor*) azok a peerek, amelyekről p rendelkezik információval és aktív kapcsolatban vannak.

1.3.2. Overlay hálózat

*Overlay hálózat*nak nevezzük azokat a (virtuális) hálózatokat, amelyek egy másik hálózatra épülnek. Például egy P2P hálózatban a peerek és a közöttük levő közvetlen kapcsolatok lehetnek az interneten aktív TCP kapcsolattal rendelkező végpontok feletti hálózat.

Az overlay hálózatot sokszor egy gráfként kezeljük, melyben a csúcsok a csomópontok, az élek pedig a csomópontok közötti összeköttetések (szomszédságok). Ha sikerült ezen absztrakt szintje eljutnunk, akkor alkalmazhatjuk a szokásos gráfelméleti megközelítéseket.

1.3.3. Churn

A gyakorlati életben egy működő P2P hálózatban a peerek folyamatosan lépnek be a hálózatba és távoznak el onnan. Ezt a jelenséget nevezzük *churn*-nek. Egy P2P algoritmus vizsgálatakor fontos eldönteni, hogy mennyire tud ellenállni a churn-nek. Sok esetben nem számíthatunk arra, hogy a peerek távozásukat azt megelőzően jelentsék, ezért algoritmusainkat úgy kell megtervezni, hogy az ilyen hatásoknak ellenálljanak.

1.3.4. Csatlakozhatóság

A tűzfalak és NAT-ok igen nehéz problémát jelentenek a P2P alkalmazások számára, ugyanis a sok P2P hálózat és algoritmus hatékonyságának szempontjából fontos, hogy az egyes peerek a többi peerrel jól tudjanak kommunikálni.

Sajnos az interneten ez nem mindig van így. A számítógépek igen nagy hányada tűzfal vagy NAT mögött van. Az ilyen peerek aránya környezet- és alkalmazásfüggő, tipikusan 35% és 90% között van [10, 33, 15, 30, 48].

A tűzfalak két osztályra bontják a peereket: *csatlakozható* (*connectable*) és *nem csatlakozható* (*unconnectable*) A NAT-okon belül több osztály is lehet, annak megfelelően, hogy mennyire "lyukaszthatóak".

1.3.5. Tűzfal

A *tűzfal* tipikusan egy olyan hálózati biztonságtechnikai eszköz, amely szabályok egy megadott halmaza alapján korlátozza az egyes csomópontokon keresztül haladó adatforgalmat. A számítógépek a TCP/IP protokollt használó hálózatokban, így az interneten is, legfőképp TCP és UDP *port*okon keresztül kommunikálnak. A legtöbb alkalmazás jól ismert porton kommunikál, viszont nincs elméleti akadálya annak sem, hogy más portot használjon.

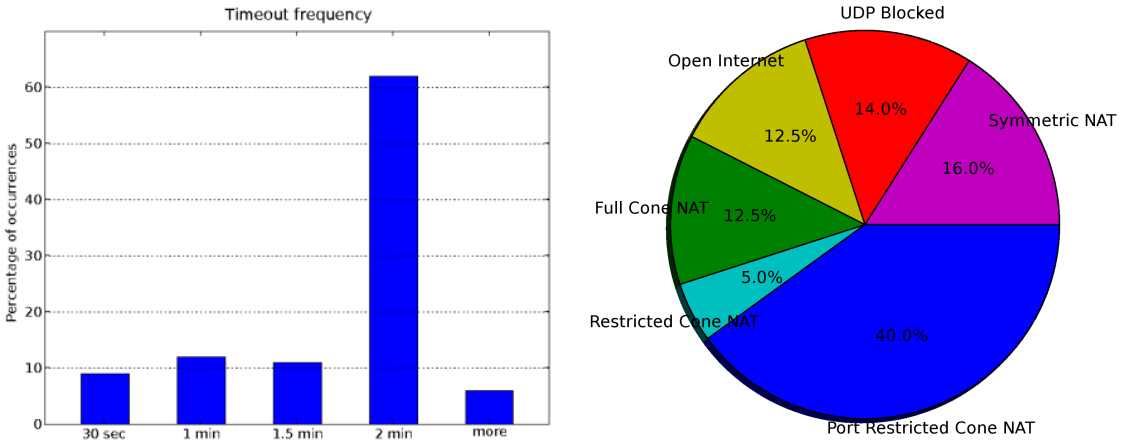
A tűzfalak szabályai tipikusan ezen jól ismert portok lezárását határozzák meg, azonban összetettebb szabályokat is tartalmazhat, amelyekben a csomagok további paraméterei szerint szűrünk.

Az interneten levő csomópontok egy része tűzfal mögött van. Például egy szervezet tűzfalat használ arra, hogy megvédje a belső hálózatát. Sok operációs rendszerben automatikusan be van kapcsolva valamilyen tűzfal.

1.3.6. NAT

A *NAT* (Network Address Translation) eredeti felhasználása az volt, hogy segítsen megfékezni az IPv4 címek nagytempójú megfogyatkozását. A NAT-okat manapság arra használják, hogy egy privát hálózatot egy vagy több publikus IP cím mögé helyezzenek el. A belső hálózat gépei egy vagy több közös külső IP címen érik el az internetet. A megvalósítás lényege, hogy a kifele irányuló csomagok küldő mezőjét áttranszformálja a publikus IP címre, a befele jövő csomagokat pedig a megfelelő címzettnek eljuttatja. A NAT-ok ezért számon tartják, melyik adatfolyamhoz melyik belső gép tartozik, azaz minden egyes külső hálózatbeli $\langle \text{IP}_v, \text{Port}_v \rangle$ párhoz egy $\langle \text{IP}_i, \text{Port}_i \rangle$ párt rendel a belső hálózatból, a kommunikációt ezen a csatornán közvetíti mindkét irányban.

Mivel a kapcsolatok egy idő után lezárulnak, kérdés, hogy mennyi ideig tárolja a NAT ezeket a leképezéseket azután, hogy az utolsó csomagok közvetítette (*NAT timeout*). Az IETF RFC4787-ben tett javaslat[13] szerint 2 percnél semmiképp nem lenne szabad érvényteleníteni egy UDP kapcsolatot, és alapértelmezésként 5 percet javasol, azonban sajnos a NAT-ok elterjedésekor ez a paraméter még nem volt jól meghatározott. A Tribler rendszerben[16] végzett felmérések[47, 10] alapján tudhatjuk, hogy a NAT-ok kevesebb, mint 70%-a legfeljebb 2 percig, és kevesebb, mint 10%-a két percnél tovább őrzi meg a kapcsolatot.



1.1. ábra. A Tribler rendszerben mért[47] NAT timeout érték és kapcsolati típusok eloszlása

A NAT-oknak négy fő csoportját különböztetjük meg aszerint, hogy mennyire engedékenyek (és ezáltal "P2P-barátiak")[21]:

- Full Cone
- Restricted Cone
- Port Restricted Cone
- Symmetric

NAT "lyukasztás"

A lyukasztás ("hole punching", "NAT traversal")[36] egy olyan technika, amellyel elérhetjük, hogy bizonyos körülmények között – a Symmetric típustól eltekintve – a NAT-okon át lehessen jutni. A módszer lényege, hogy van egy harmadik közvetítő fél, aki felé tartja a kapcsolatot a NAT mögötti gép, és feljegyzi, hogy mely portokat engedte be, majd közli ezeket a másik féllel, aki megpróbál ezeken a portokon bejutni.

Léteznek kiforrott lyukasztási technikák[6]. Mivel a TCP állapot használó protokoll, az UDP pedig nem, a UDP esetén könnyebb dolgunk van. Egy felmérés szerint a NAT-ok TCP esetében átlagosan 64%-ban, míg UDP használata esetén 82%-ban lyukasztható-ak[14]. Egy 2005-ös eredmény alapján [18] átlagosan 88%-os TCP áthatolás is elérhető.

Néhány NAT traversal technológia:

- STUN [21, 23]
- TURN [22]
- ICE [20]

Universal Plug and Play

Az *UPnP* (*Universal Plug and Play*)[38] egy konfigurációs protokoll, amellyel a routerünktől kérhetünk alkalmazásunknak engedélyezett nyitott portokat. A technika szabványosítva van, azonban igen kevesen használják.

1.4. Keresés

Gyakori feladat, hogy valamilyen információ után kell keresnünk egy P2P hálózatban. A keresés problémáját többféleképpen is meg lehet oldani. Ezek a megoldások különbözhetnek a hatékonyság, skálázhatóság, robosztusság szempontjából.

1.4.1. Strukturált és strukturálatlan hálózatok

A *strukturálatlan* hálózatban a nincs a peerek között megállapodás azzal kapcsolatban, hogy kinek ki a szomszédja. Ezek a hálózatok tipikusan véletlen módon jönnek létre, pl. egy ilyen hálózat az, ahol minden peernek 10 véletlen szomszédja van.

A *strukturált* hálózatban viszont a peerek valamilyen strukturát alkotnak. Ez a struktura lehet igen összetett is. A struktura fenntartásának költsége van, azonban egyes műveletek a struktura felépítése után már hatékonyabban végezhetőek.

Léteznek hibrid módszerek is, amelyek a két módszert egyesítik, pl. Gnutella.

Keresés strukturálatlan hálózatokban

A strukturálatlan hálózatban a keresés[32] egy triviális módja az *elárasztásos keresés* (*flooding*). A kereső fél egy üzenetet küld az összes szomszédjának, amely tartalmazhat egy lejárati értéket (*Time to Live*, röviden: *TTL*). A lejárati érték minden ugrás (*hop*) után eggyel csökken, és ha eléri a 0-t, a keresés befejeződik. Ezzel lehet ez bizonyos mélységig futó keresést indítani. A TTL megfelelő értékének megtalálása nem triviális feladat. A keresés megállításának másik módja, hogy lejárati üzenetet küldünk, amely jelzi, hogy a keresés befejeződhet. Ha egy node megtalálja a keresett értéket, a válasz üzenetet a keresési útvonalon visszafele elküldi.

A módszer hátránya, hogy nagyon pazarló. Ha a hálózat gráfjában körök vannak, egyes peerek kétszer is megkaphatnak egy üzenetet. Ha sokan keresnek, akkor a hálózat nagyon leterhelődik.

A keresés másik módja a strukturálatlan hálózatokban a *véletlen séta*. A együzenetes véletlen séta során egy keresésről csak egy szomszédnak küldünk keresési kérést. Ez a "sétáló" üzenet továbbítódik a szomszédunk egy másik véletlen szomszédjának. A terhelés nagyságrendekkel jobb lehet, mint az elárasztásos esetben, de az információ megtalálásának ideje nagyságrendekkel rosszabb lehet. Kiküldhetünk k üzenetet is egyszerre. Ekkor a k üzenetünk kb. k-szor több peert fog elérni T idő alatt, mintha egy üzenetet küldtünk volna.

A sétáló üzenetek száma tehát az időigény és a terhelés közötti kompromisszumot jelenti.

Keresés strukturált hálózatokban

Felmerülhet a probléma, hogy a ritka (kevés node-ban megtalálható) információkat csak nehezen tudjuk megszerezni. Ennek kiküszöbölésére jöttek létre az *elosztott hasítótábla* (*Distributed Hash Table*, röviden: *DHT*) alapú rendszerek. A DHT-k lényege, hogy a rendszer $\langle \text{kulcs}, \text{érték} \rangle$ párokat tárol elosztott módon úgy, hogy hatékonyan lehessen keresni és lehetőleg ellenálló legyen a churn-nel szemben.

Az egyik legjobban ismert DHT megvalósítás a Chord[43].

1.5. Fájlcserélő hálózatok

Az egyik legnépszerűbb P2P alkalmazás a fájlcserélés. A P2P fájlcserélés lényege, hogy a rendszerben részt vevő felhasználók valamilyen tartalommal rendelkeznek a saját számítógépükön és ezt a tartalmat másokkal meg szeretnék osztani. Ezt segítik a fájlcserélő programok, amelyek kiépítik a kapcsolatot felhasználók számítógépei közt.

Téves gondolat lehet azonban, hogy minden fájlcserélő alkalmazás illegális tartalmat közvetít. A jogvédő szervezetek üldözik a fájlcserélő közösségeket, egyelőre többkevesebb sikerrel. Valóban, egy nagy részük illegális tartalomra koncentrál, de nem minden fájlcserélő alkalmazás ilyen. Vannak játékszoftver fejlesztő cégek (pl. Blizzard), amelyek a frissítéseket a szoftvereikhez P2P módon közvetítik, ezzel is takarékoskodva a saját sávszélességükkel.

Ebben a részben bemutatok az interneten elérhető számtalan fájlcserélő alkalmazás közül néhányat.

1.5.1. Gnutella

A Gnutella[42] volt az első próbálkozás a decentralizált fájlmegosztásra. Az AOL igen korán leállította a projekt weboldalát, azonban ez nem akadályozta meg a szoftver elterjedését. A elárasztásos (flooding) módon kommunikált, azaz peerek minden egyes keresést minden szomszédnak elküldenek, azok pedig továbbítják a keresést. Ez túl nagy terhelést jelentene egy nagy hálózatban.

A peerek közel 70%-a nem osztott meg fájlokat, és a találatok közel 50%-a a megosztó számítógépek legfelső 1%-ára mutatott[1].

1.5.2. Napster

A Napster egy mp3 zenefájlok megosztására fókuszáló hálózati szolgáltatás[5]. Egy központi számítógép tartja számon a felhasználókat, és hogy az egyes felhasználók milyen tartalommal rendelkeznek. Egy letöltéshez a kliensnek le kell kérdeznie a központi géptől azon kliensek listáját, akik a tartalommal rendelkeznek. Ezután közvetlenül le tudja tölteni a tartalmat.

Mivel a rendszer központi adatbázist használ, amely minden letöltésnél terhelés alatt van, a rendszer nem skálázódik jól.

A jogvédő szervezetek nyomására megszűntették a szolgáltatást, fizetős változata továbbra is működik.

1.5.3. BitTorrent

A BitTorrent[9] napjaink egyik legnépszerűbb P2P fájlmegosztó szoftvere. A protokoll előnye, hogy jól skálázódik, a peerek együttműködnek, egyszerre töltenek felfele és lefele.

Fogalmak

A *torrent* egy olyan fájlformátum, amely a megosztani kívánt fájlokról metaadatokat tartalmaz, ezzel lehetővé teszi azok forgalmazását. Egy torrent tárolhat egyetlen fájlt, de akár egy egész könyvtárszerkezetet is.

Minden fájl *darab*okra (*chunk* vagy *piece*) van felbontva, amelyeket a protokoll előre meghatározott méretű blokkokban közvetít. Azok a kliensek, amelyek egy közös torrent forgalmazásában részt vesznek, egy *swarm*ot alkotnak. A swarm-ok egyedi azonosítója az *Info Hash* [1], amely a SHA-1[4] kódolása a torrent fájlban levő metaadatoknak. Az Info Hash segítségével a peerek megbizonyosodhatnak arról, hogy a megfelelő tartalmat töltik.

Az egyes peereket egy *tracker*nek nevezett központi szerver követi nyomon. A tracker az elsődleges forrása a peereknek, lekérdezhetjük az általa számon tartott peerek egy véletlenszerű részhalmazának adatait ($\langle \mathrm{IP}, \mathrm{Port} \rangle$). A terhelés csökkentése érdekében a trackert bizonyos időközönként (5-50 perc) kérdezzük le.

Azon peerek, amelyek egy torrent összes darabjával rendelkeznek, *seed*eknek nevezzük. A többi peer *leech* [2].

Szokás a peerek összes letöltését és feltöltését figyelembe venni. A *megosztási arány*t a feltöltött és letöltött adatmennyiség hányadosaként definiáljuk. Ezt a kliensek tudják megállapítani és erről periódikusan tájékoztatják trackert.

A *Free-Riding* kifejezést szokás használni akkor, ha valaki csupán lefele tölt, a hálózat számára viszont nem nyújt erőforrásokat. *Hit'n'Run*-ról akkor beszélhetünk, ha egy peer azonnal lekapcsolódik a hálózatról, amint sikerült letöltenie a teljes torrentet. Az *initial seed* az a seed, aki a tartalmat először elérhetővé teszi (ilyenből több is lehet).

Tit-for-Tat

A BitTorrent algoritmusok hatékonysága annak köszönhető, hogy jól meghatározott módon döntik el, hogy melyik peert preferálják a feltöltés során. Ha egy peernek nem töltünk, azt megfojtjuk (*choke*), később feloldhatjuk (*unchoke*). Az, hogy kinek töltünk fel, a "szemet szemért, fogat fogért" (*tit-for-tat*) elv alapján döntjük el. Akitől nagy sebességgel tudunk letölteni, nagy feltöltési sebességet allokálunk neki. Időnként egy véletlenszerű peert is feloldunk (*optimistic unchoking*).

Protokoll kiegészítések

Egyes BitTorrent kliens szoftverek saját DHT hálózatot építenek a felhasználóik felett[49]. Ilyenek például a Vuze (korábban Azureus) és a μTorrent. A DHT hálózat segítségével a tracker által szolgáltatott peereken felül további peerekhez juthat a felhasználó. A módszer lényege, hogy minden peer egy elosztott trackert alkot és ez növelheti az összteljesítményt.

Egy másik módja a peer információszerzésnek, *Peer Exchange*, amely egy gossip algoritmus (ld. Pletyka algoritmusok fejezet). Az egy swarmba tartozó peerek közlik egymással azon peerek listáját, akikkel kapcsolatban vannak, így a harmadik fél is tudomást szerezhet erről az információról.

A *Super-Seeding* (vagy *initial seeding*) [8, 7] egy speciális feltöltési stratégia, amelyet az initial seedek használnak a swarm kezdetén. Az algoritmus lényege, hogy ahhoz, hogy a blokkok cseréje beinduljon, a lehető legtöbb különböző blokkot be kell juttatni a hálózatba. Ezért a super-seedinget alkalmazó seedek úgy tesznek, mintha nem rendelkeznének a teljes torrenttel, és különböző peerek felé más és más darabok jelenlétét jelzik.

[1] Mivel a függvénynek csak véges számú értéke lehet, ütközések előfordulhatnak, azonban ennek valószínűsége kicsi.

[2] A terminológia változó. Egyes helyeken a *peer* megnevezés megegyezik a *leech*el.

Publikus és privát közösségek

A BitTorrent alapú fájlmegosztás során alapvetően kétféle csoportosulás szokott kialakulni[50, 49].

A publikus közösségekre jellemző, hogy a torrenteket bárki elérheti és részt vehet a swarmban. Ilyen torrent fájlok publikus weboldalakon találhatóak, a peerek DHT-ből és PEX-ből is származhatnak. Egy torrent tipikusan több publikus trackert is bejegyez, abból a célból, hogy több peer elérhető legyen.

A privát közösségek ezzel szemben regisztrációhoz kötöttek. Bizonyos esetekben nyitott a regisztráció, más esetekben nem lehet regisztrálni, vagy csak meghívóval. A privát közösségek sajátos szabályrendszerrel rendelkeznek. Ahhoz, hogy egy felhasználó a közösség hosszú távú tagja lehessen, fent kell tartania egy megadott minimális megosztási arányt. A rendszer eltávolítja azokat a felhasználókat, akik nem tartják be a szabályokat. Ennek az ösztönző hatásnak az eredménye, hogy a privát közösségekben jóval nagyobb az átlagos átviteli sebesség[33]. A szabályos működést követő kliens szoftverek a *private* bit észlelésekor nem használnak DHT-t és PEX-t annak érdekében, hogy ne szivárogjon ki a tartalom.

2. fejezet

Pletyka protokollok

2.1. Bevezető

A *pletyka alapú* (*gossip*) protokollok [11, 12, 24, 29, 25, 26] algoritmusok egy csoportja, amelyek lokális információ felhasználásával olyan feladatokat képesek megvalósítani, amelyeket másképp nehézkes lenne. A lokális információ az egyes peerek "memóriája", ami egy részleges nézetet tárol a hálózatból. Működése hasonló az emberek közti pletykáláshoz. Más néven járvány (epidemic) protokollként is szokás nevezni, mert az információ járványszerűen terjed a hálózatban. Nagyon hatékony módja az információ terjesztésének, nincs szükség központi szerverre, és amint egyszer elindul a folyamat, nagyon nehéz megállítani. Ezeket a hasznos tulajdonságokat használjuk ki, amikor gossip algoritmusokat alkalmazunk elosztott rendszerekben. Eredetileg információ terjesztésre használták őket, de más alkalmazásai is vannak:

- alkalmazás szintű multicast
- aggregáció (pl. átlag)
- overlay struktura menedzsment

A gossip algoritmusok előnye tehát, hogy képesek strukturálaltlan és strukturált hálózatokban egyaránt műkdöni, csupán lokális információ felhasználásával.

2.1.1. A protokoll váza

Az egyes peerek meghatározott időközönként kommunikálnak szomszédaikkal és frissítik a lokális információjukat. A gossip protokollokat követő peerek legtöbbször a hálózat nagyságához viszonyítva kevés lokális információval rendelkeznek, mégis, a megfelelő együttműködés során jelentős globális hatással vannak.

A protokoll két szálon fut:

- aktív szál (kliens): periódikusan egy másik peerhez csatlakozik, vele információt cserél (Algoritmus 1).
- passzív szál (szerver): kapcsolatot fogad egy másik peer aktív szálától, vele információt cserél (Algoritmus 2).

Tehát a protokollt futtató peerek periódikusan, T időközönként végrehajtják az algoritmus *aktív* szálát, miközben folyamatosan figyelnek a bejövő kapcsolatokra is, melyeket a *passzív* szálon feldolgoznak.

Első modellünkben feltételezzük, hogy az üzenetek azonnal megérkeznek, és hogy nincs üzenetveszteség és üzenethiba. Minden peer egyszerre küldi az üzenetét, a két üzenetküldés közötti T hosszú intervallumot ciklusnak nevezzük.

Ha PUSH értéke igaz, az azt jelenti, hogy a kiválasztott peernek el kell küldeni a saját lokális információt. Különben csak jelezzük, hogy őt választottuk információcserére. Ha PULL értéke igaz, akkor elvárjuk, hogy a kiválasztott peer információt küldjön. Látható, hogy ha mindkét változó értéke igaz, az algoritmus *szinkron* módon működik, különben pedig *aszinkron* módon.

Algoritmus 1 A gossip algoritmus aktív szála

```
1:  loop
2:     wait(T) {T időegység várakozás, periódikusság}
3:     p ← selectPeer() {kiválasztjuk azt a peert, akivel információt cserélünk}
4:     if push then
5:        state elküldése p-nek
6:     else
7:        send (null) to p {csak jelezzük, hogy őt választottuk}
8:     end if
9:     if pull then
10:       state_p fogadása p-től
11:       state ← update(state, state_p) {az állapot feldolgozásával adódik az új állapot}
12:    end if
13: end loop
```

Algoritmus 2 A gossip algoritmus passzív szála

```
1:  loop
2:     state_p fogadása p-től
3:     if pull then
4:        state elküldése p-nek
5:     end if
6:     state ← update(state, state_p) {az állapot feldolgozásával adódik az új állapot}
7:  end loop
```

2.1.2. Bootstrapping

Amikor egy felhasználó belép a hálózatba, nincs még semmilyen kapcsolata más peerek felé. Ahhoz, hogy a pletyka alapú működés beinduljon, legalább egy peer elérhetőségét tudni kellene.

Egy lehetséges módja ennek, ha superpeereket használunk. A superpeerek címe (vagy a hely, ahol ezeket megtaláljuk) tipikusan be van égetve a szoftverbe, így a kliens aktív

szála először a superpeerek felé kezdeményez, akiktől megtudja néhány közönséges peer címét, és így be tud csatlakozni a hálózatba.

A hálózat felállását nevezzük *bootstrapping*nek. A protokollok vizsgálata során legtöbbször nem foglalkozunk a bootstrapping fázis vizsgálatával, hanem azt feltételezzük, hogy egy már meglévő overlay hálózatból indulunk ki (például konstans fokszámú véletlen hálózat).

2.2. Peer mintavételezés

A *peer mintavételezés* (*peer sampling*) lényege, hogy a P2P hálózatban levő peerek egy csoportja valamilyen feladatot szeretne elvégezni, amihez egy (egyenletes eloszlású) peer minta szükséges a hálózatból.

A következőkben ismertetem, hogy hogyan lehet ezt gossip algoritmusokkal elérni[29].

2.2.1. A lokális információ

Modellünkben[29] minden peernek van egy címe, ahol a többi peer megtalálhatja (pl. IP cím). Minden peer lokálisan eltárolja a hálózat egy részleges *nézet*ét (*view*). A nézet egy olyan lista, amely peerek címét és a bejegyzés korát tartalmazza. A bejegyzések sorrendjét megtartja, és értelmezettek a szokásos lista műveletek (első, utolsó, iterálás, hozzáadás, törlés, véletlen minta, permutáció, stb...). A bejegyzések egyediek, szokás *csomópont leíró*nak (*node descriptor*) is nevezni őket.

2.2.2. A protokoll váza

Az információcsere célja, hogy az egyes peerek részleges, időről-időre változó (frissülő) nézete a hálózatban részt vevő csomópontok (egyenletes eloszlású) véletlen mintáját képezze.

Az algoritmus három paramétert vár:

- c: a nézet maximális mérete.
- H (healing): az elévülés mértéke, nagyobb érték esetén a régi bejegyzések nagyobb mértékben tűnnek el.
- S (swap): a csere mértéke, nagyobb értékek esetén nagyobb eséllyel kerülnek be a kapott leírók a saját nézetbe.

Az algoritmus lépései a következők:

1. Először a peer kiválasztja azt a szomszédját, amelyikkel információt fog cserélni. Ezt a VIEW.SELECTPEER() eljárás végzi.
2. Ha küldeni kell üzenetet, a bufferbe belekerül a saját nézetből a H legrégebbitől eltekintve $c/2-1$ véletlenszerű elem és a saját leíró, melynek kora 0 ($H \leq c/2$). A buffert elküldjük a cél peernek.

3. Ha érkezett információ, egyesíteni kell a saját lokális információnkkal. Ezt fogja a VIEW.SELECT() elvégezni, amelynek megvalósítását az Algoritmus 5 mutatja be. A megvalósítás a kapott paraméterek alapján egy új nézetet hoz létre, ügyelve arra, hogy a nézet mérete ne haladja meg c-t. A két listát összevonja, majd eltűnteti az ismétlődő elemeket. Ezután legalább olyan hosszú a nézet, mint korábban, így szükség lehet további elemek eltávolítására. Először eltávolít a legrégebbi elemek közül legfeljebb H-t, majd az első legfeljebb S elemet (a lista elején azok az elemek szerepelnek, amelyekkel a peer már korábban rendelkezett). A továbbiak közül véletlenszerűen választ, amíg el nem éri a megfelelő c méretet.

Algoritmus 3 Az aktív szál

```
1:  loop
2:    wait(T) {T időegység várakozás, periodikusság}
3:    p ← view.selectPeer() {kiválasztjuk azt a peert, akivel információt cserélünk}
4:    if push then
5:      buffer ← ((MyAddress, 0)) {0 a kezdeti kor}
6:      view.permute() {véletlenszerűen permutáljuk a nézet elemeit}
7:      vigyük a H legrégebbi elemet a view végére
8:      buffer.append(view.head(c/2 − 1)) {a buffer végére helyezzük el a nézet első
        c/2 − 1 elemét}
9:      buffer elküldése p-nek
10:   else
11:     (null) küldése p-nek {csak jelezzük, hogy őt választottuk}
12:   end if
13:   if pull then
14:     buffer_p fogadása p-től
15:     view.select(c, H, S, buffer_p) {a két nézet egyesítése, azaz update()}
16:   end if
17:   view.increaseAge() {növeljük a kor értékeket a nézetben, idő szimulálása}
18: end loop
```

2.2.3. Néhány lehetséges megvalósítás

Ezt a keret algoritmust három dimenzió mentén szabályozhatjuk, a következőkben megemlítek néhány esetet[29]:
Peer szelekció (peer selection). Az algoritmusban definiáljuk a SELECTPEER() metódus működését, amely minden aktív lépés elején meghatározza a következő peert, akivel kommunikál a protokoll.

- rand: Egyenletes eloszlás szerint véletlenszerűen választunk a peerek között.
- tail: A legrégebbi peert választjuk.

Nézet propagáció (view propagation). Aszinkron, vagy szinkron kommunikáció.

- pushpull: A peernek küldünk üzenetet, és fogadunk is tőle. Az üzenetekre válaszolunk.

Algoritmus 4 A passzív szál

1: **loop**
2: buffer$_p$ fogadása p-től
3: **if** pull **then**
4: buffer ← ((MyAddress, 0)) {0 a kezdeti kor}
5: view.permute() {véletlenszerűen permutáljuk a nézet elemeit}
6: vigyük a H legrégebbi elemet a view végére
7: buffer.append(view.head($c/2 - 1$)) {a buffer végére helyezzük el a nézet első $c/2 - 1$ elemét}
8: buffer elküldése p-nek
9: **end if**
10: view.select(c, H, S, buffer$_p$) {a két nézet egyesítése, azaz update()}
11: view.increaseAge() {növeljük a kor értékeket a nézetben, idő szimulálása}
12: **end loop**

Algoritmus 5 A view.select(c, H, S, buffer$_p$) metódus

1: view.append(buffer$_p$) {a nézetünk végére csatoljuk az új információt}
2: view.removeDuplicates() {töröljük az ismétlődő elemeket}
3: view.removeOldItems(min(H, view.size - c)) {a kor alapján töröljük régi elemeket, legfeljebb H darabot}
4: view.removeHead(min(S, view.size - c)) {legfeljebb S elemet törlünk a nézet elejéről}
5: view.removeAtRandom(view.size - c) {törlünk véletlenszerű elemeket, amíg el nem érjük a c méretet}

– push: A peernek küldünk üzenetet, de nem várunk választ. Az üzenetekre nem válaszolunk.

Nézet szelekció (view selection). A H és S paraméterek.

– blind: $H = 0$, $S = 0$, vakon választunk egy véletlen részhalmazt.

– healer: $H = c/2$, $S = 0$, tartsuk meg a legfrissebb bejegyzéseket.

– swapper: $H = 0$, $S = c/2$, minimalizáljuk az információveszteséget.

2.3. Aggregáció egy hálózat fölött

Gyakori feladat, hogy a felhasználók fölött aggregációkat végzünk. Minden felhasználó rendelkezik egy numerikus értékkel, pl. egy szavazati értékkel, a feladat meghatározni az összes felhasználó értékének az átlagát. A kliens-szerver esetben nincs más teendőnk, mint minden felhasználó értékét begyűjteni, majd kiszámolni az átlagot. A P2P hálózatokban viszont nem kivitelezhető, hogy minden peert megkeressünk. Itt jönnek segítségünkre a gossip algoritmusok.

A következőkben bemutatok egy egyszerű gossip aggregációs algoritmust[25], amely kiszámítja a hálózatban tárolt számok átlagát.

2.3.1. Átlag számítása

Ahhoz, hogy algoritmusunk teljes legyen, a korábbiakban ismertetett három dimenzió mentén definiálni kell az algoritmus működését. A SELECTPEER() metódus a *rand* módszert használja. A peerek lokális információja kibővül egy numerikus értékkel, ami a globális átlag közelítése, kezdetben a peerhez rendelt szám. A frissítés definíciója legyen

$$\text{update}(p, q) := \frac{p + q}{2} \tag{2.1}$$

ahol p és q a két kommunikáló peer lokális becslése az átlagra. Egy csere után a számok összege változatlan marad, azonban eloszlik a két peer között. Az eljárás tehát nem változtatja meg a rendszer átlagát, de csökkenti a szórást, az egyes értékek az átlaghoz konvergálnak.

2.3.2. További aggregációk

Az UPDATE() definíciójának megfelelő változtatásával további aggregációkat is elérhetünk:

$$\text{update}(p, q) := \sqrt{pq}, \text{mértani átlag} \tag{2.2}$$

$$\text{update}(p, q) := \min(p, q), \text{minimum} \tag{2.3}$$

$$\text{update}(p, q) := \max(p, q), \text{maximum} \tag{2.4}$$

2.4. Overlay menedzsment: T-Man

Gossip algoritmusok overlay hálózatok felépítésére is használhatóak. A megfelelő overlay hálózat használata nagyon fontos a keresés, tartalommegosztás, forgalomirányítás, aggregáció szempontjából.

A *T-Man*[26] egy generikus pletyka alapú protokoll, amellyel sokféle overlay hálózat fenntartható egy lokális rendezési függvény megadásával. Az algoritmus jól skálázódik és gyors; a hálózat méretének logaritmusával arányosan konvergál.

Az overlay hálózat megfelelő kiépítése fontos lehet például egy útvonal-tervezési feladat esetében.

2.4.1. A topológia felépítés feladata

Tegyük fel, hogy egy konstans k fokszámú véletlen hálózatból indulunk ki. Ezen felül kikötjük, hogy egy peernek legfeljebb c szomszédja lehet. A peerek halmaza legyen N, melynek mérete lehet nagyon nagy is. Egy nagy P2P hálózatban nem kivitelezhető, hogy a peereknek egy kis konstans számnál több szomszédjuk legyen.

Legyen R egy *rangsoroló függvény* (*ranking function*) a peerek fölött, amely minden peer esetén az általa preferált peereket előnyben részesíti. A R bemeneti paraméterül kap egy kiindulási peert és peerek egy $\{x_1, \ldots x_m\}$ halmazt, kimenete a peerek halmazának egy olyan rendezése, amelyben elöl szerepelnek azok a peerek, akiket a kiindulási peer szívesen látna szomszédként. A peerek leírója kibővül egy tulajdonság vektorral, amely tartalmazhat pl. IP címet, sávszélességet, stb...

A feladat az, hogy elérjük, hogy minden $x \in N$ peer nézete (view_x) pontosan az első c elemét tartalmazza a "jó" rangsornak, azaz $R(x, N \setminus \{x\})$-nek. Ezt a topológiát nevezzük *cél topológiá*nak.

Amennyiben churn is jelen van, a cél topológiának fenntartásáról beszélhetünk. Ilyenkor az a cél, hogy a cél topológiához a lehető legközelebb kerüljünk.

2.4.2. Távolság alapú T-Man

Az R előállításának egyik módja, hogy olyan távolságfüggvényt használunk, amely a peereket egy térben helyezi el. Ekkor a rangsor a távolság szerinti növekvő sorrend.

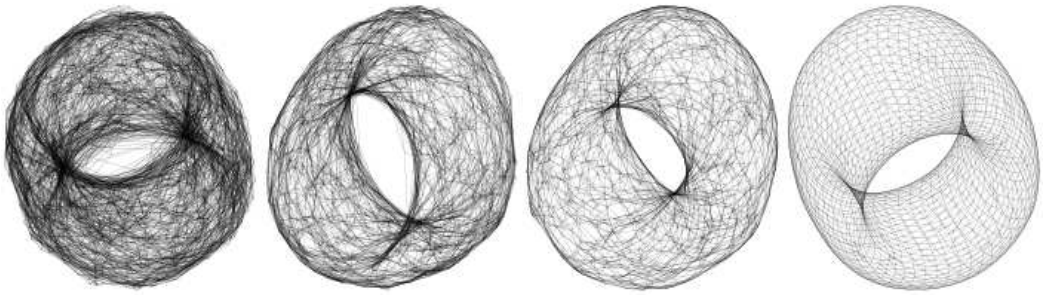
Egydimenziós eset: Vonal és gyűrű

A peerek egy valós számot tárolnak (0 és $N-1$ között). A $d(p,q) = |p-q|$ távolságfüggvény egy vonal, a $d(p,q) = \min(N - |p-q|, |p-q|)$ pedig egy gyűrű topológiát definiál.

Többdimenziós eset: háló, cső, tórusz

Az egydimenziós esetet általánosíthatjuk többdimenziós esetté, a peerek ekkor egy n-dimenziós vektort tárolnak. A háló esetében a távolságfüggvény a Manhattan távolság, azaz

$$d(p,q) = \|p-q\|_1 = \sum_{i=1}^{n} |p_i - q_i| \tag{2.5}$$



2.1. ábra. Egy tórusz topológia kialakítása 2500 csomópontból[26], 3, 5, 8, 15 ciklus után.

$n = 3$ esetén, ha a gyűrűhöz hasonlóan körkörös távolságfüggvényt használunk az egyik koordinátára, csövet kapunk; ha kettőre, akkor pedig tóruszt. A tórusz előállításának folyamatát mutatja be a 2.1 ábra.

2.4.3. A T-Man algoritmus váza

Algoritmus 6 A T-Man aktív szál

```
1:  loop
2:    wait(T) {T időegység várakozás, periodikusság}
3:    p ← selectPeer() {kiválasztjuk azt a peert, akivel információt cserélünk}
4:    if push then
5:      buffer ← ((MyAddress, MyProfile))
6:      buffer ← merge(view, {MyDescriptor})
7:      buffer ← merge(buffer, rnd.view)
8:      buffer elküldése p-nek
9:    else
10:     (null) küldése p-nek {csak jelezzük, hogy őt választottuk}
11:   end if
12:   if pull then
13:     buffer_p fogadása p-től
14:     buffer ← merge(buffer_p, view)
15:     view ← selectView(buffer)
16:   end if
17: end loop
```

Két kulcs metódus a SELECTPEER() és a selectView(). A SELECTPEER() veszi az aktuális nézetet (view) és alkalmazza rajta az R rendezést, majd veszi az így kapott lista első elemét. A SELECTVIEW() rendezi a buffert, így a rendezés szerinti első c bejegyzést adja vissza.

Az RND.VIEW az egész hálózat egy véletlenszerű mintája, amit a peer sampling szolgáltatás segítségével kapunk meg. Jelentősége nagy átmérőjű hálózatoknál van.

Algoritmus 7 A T-Man passzív szál

```
1: loop
2:    buffer_p fogadása p-től
3:    if pull then
4:       buffer ← ((MyAddress, MyProfile))
5:       buffer ← merge(view, {MyDescriptor})
6:       buffer ← merge(buffer, rnd.view)
7:       buffer elküldése p-nek
8:    end if
9:    view ← selectView(buffer)
10: end loop
```

2.5. NewsCast

A pletyka alapú *NewsCast* algoritmusok[28, 44] kibővítik a lokális információt egy időbélyeg (*timestamp*) mezővel, amely az adott információ "frissességét" hivatott jelezni. A NewsCast algoritmus véletlen hálózatokat tud létrehozni, ahol az új információ felváltja a régit, ezzel robosztusságot biztosítva. A lokális információt *cache*-nek nevezzük (mely c hosszú), és minden bejegyzése egy alkalmazás-specifikus adatot és egy időbélyeget tartalmaz. A gossip algoritmus figyelembe veszi az egyes bejegyzésekhez tartozó időbélyeget, és a frissebb bejegyzéseket preferálja. Az egyes peereket ügynököknek (*agent*) is nevezzük, és megvalósítják a GETNEWS() és UPDATENEWS() metódusokat.

2.5.1. Az algoritmus váza

Algoritmus 8 A NewsCast algoritmus aktív szála

```
1: loop
2:    wait(T) {T időegység várakozás, periodikusság}
3:    p ← selectPeer() {kiválasztjuk azt a peert, akivel információt cserélünk}
4:    cache.add(agent.getNews())
5:    cache.removeOlderThan(cT) {cT-nél régebbi elemek eltávolítása}
6:    cache elküldése p-nek
7:    if pull then
8:       cache_p fogadása p-től
9:       agent.newsUpdate(cache_p)
10:      cache.append(cache_p)
11:      az ágensek között a legújabb elemet tartsuk meg
12:      az időbélyegek alapján a c legújabb elemet tartsuk meg
13:   end if
14: end loop
```

Algoritmus 9 A NewsCast passzív szála

```
1:  loop
2:      cache_p fogadása p-től
3:      agent.newsUpdate(cache_p)
4:      cache.append(cache_p)
5:      az ágensek között a legújabb elemet tartsuk meg
6:      az időbélyegek alapján a c legújabb elemet tartsuk meg
7:      if pull then
8:          cache elküldése p-nek
9:      end if
10: end loop
```

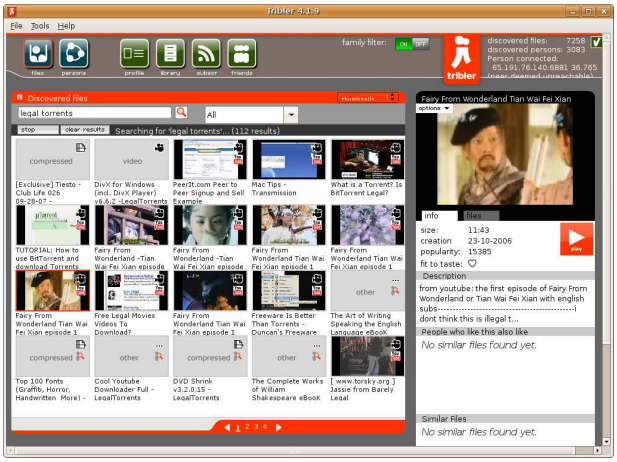
3. fejezet

BuddyCast és implementálása

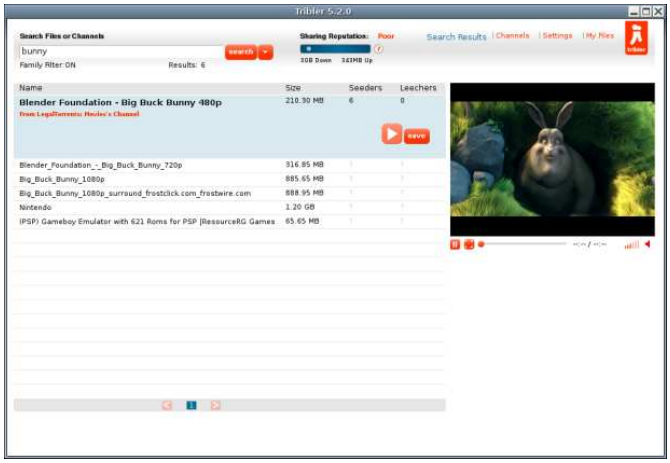
A *BuddyCast*[39] egy NewsCast-ra épülő protokoll, amelyet a Tribler[16] rendszerhez fejlesztettek ki, azzal a céllal, hogy a felhasználók közötti kapcsolatok alapján előnyös tulajdonságú overlay hálózatot építsen. A Tribler egy működő BitTorrent alapú fájlcserélő rendszer. amely a BuddyCast-ra épül. A protokollt ajánlásra használják, a felhasználóhoz igyekszik közel tartani azokat a felhasználókat, akik hasonló ízlésűek. Fő funkciói:

- peerek feltérképezése (peer discovery)
- tartalom feltérképezése (content discovery)
- megfelelő overlay hálózat felépítése (semantic overlay forming)

Az eredeti változat 2005-ben készült, melyet 2006 januárjában építettek be a Tribler rendszerébe, így éles, gyakorlati környezetben (az interneten) is ki tudták próbálni, hogy hogyan teljesít. Megfigyelték, hogy a peerek egy része tűzfal vagy NAT mögött van. A churn jelentős problémát okozhat, ugyanis egyes peerek nem jelzik távozásukat szabályosan.



3.1. ábra. A Tribler egy korábbi változata működés közben[40]



3.2. ábra. A Tribler egy újabb változata működés közben[45]

3.1. A protokoll részletes leírása

Lássuk, hogy hogyan is működik a protokoll[46]. Mivel a protokoll pletyka alapú, ezért van egy aktív és egy passzív szála. Az aktív szál meghatározott időközönként fut le minden csomóponton (ciklus hossz). A peereknek lokális információjuk van, és ezen lokális információt üzenetek formájában közvetítik a szomszédoknak, akik visszaküldik a saját lokális információjukat. Az üzenetek feldolgozásra kerülnek, a lokális információk egyesülnek egy meghatározott módon.

3.1.1. A lokális információ

A felhasználók egyedi azonosítóval rendelkeznek (PermID). Ez segít nyomon követni a felhasználókat és megfékezni a rossz szándékú klienseket.

A Tribler kliens ellenőrzéseket (dialback) végez annak érdekében, hogy kiderítse, hogy a kliens tűzfal vagy NAT mögött van-e. Ha NAT mögött van, megkísérel egy UPnP konfigurációt. Minden kliens tehát ismeri a saját csatlakozhatóságát, és ez a lokális információ része.

A lokális információ listákból áll, amelyek korlátozott méretűek. Minden felhasználónak van preferenciája. A *preferencia lista* (*preference list*) azon torrentek listája (halmaza), amelyet a felhasználó letöltött. It azt feltételezzük, hogy amit a felhasználó letöltött, az érdekelte is. A torrenteket a Hash Info értékük alapján azonosítunk.

Az algoritmus használata során a preferencia lista bővül. A felhasználók egymásnak elküldik preferencia listájukat, majd az ajánlás részeként egy meghatározott hasonlósági függvény meghatározza a két lista közötti hasonlóságot, amelyet a két felhasználó hasonlóságaként értelmezünk. A leghasonlóbb felhasználók között egy speciális *Taste Buddy* (röviden *TB*) kapcsolat jön létre.

A C *kapcsolati lista* (*Connection List*) azon peereket tartalmazza, akikkel aktív TCP kapcsolatot tart fent a protokoll. Ez a lista három részre oszlik:

– C_T Kapcsolatban levő Taste Buddy lista (Connectible Connected Taste Buddy List): Ide azon hasonló peerek kerülnek, akik nincsenek tűzfal vagy NAT mögött, ezért tudunk csatlakozni hozzájuk (és már csatlakoztunk is).

– C_R Kapcsolatban levő véletlen peer lista (Connectible Connected Random Peer List): Azon peerek listája, akik felénk kapcsolatot létesítettek, de nem áll fenn Taste Buddy reláció.

– C_U Nem kapcsolható csatlakozott peer lista (Unconnectible Connected Peer List): Azon nem kapcsolható peerek listája, akik hozzánk csatlakoztak.

A C_C lista a kapcsolódásra jelölt peerek listája (Connection Candidates List). Bizonyos körülmények között közülük keresünk meg valakit, hogy egy alkalmas Taste Buddy-t találjunk.

A B blokk lista (Block List) célja, hogy számon tartsa azon peereket, akikkel a közelmúltban információt cseréltünk. A blokk lista célja, hogy megakadályozza a felesleges kommunikációt, minden egyes peert csak bizonyos idő elteltével kereshetünk meg. B két részre oszlik: bejövő (B_R) és kimenő (B_S) blokk lista.

A protokoll aktív szálát az Algoritmus 10 mutatja be. A passzív szál hasonló, az Algoritmus 11 leírja.

Algoritmus 10 A BuddyCast aktív szál

```
1:  loop
2:    wait(T) {T időegység várakozás, alapesetben 15 másodperc}
3:    B frissítése: azon peerek feloldása, akiknek lejárt a blokk idejük
4:    if C_C üres then
5:      C_C ← ismert superpeerek (5 darab)
6:    end if
7:    rand ← random(0,1)
8:    if rand ≤ α then
9:      Q ← a leghasonlóbb Taste Buddy a C_T listából {exploitation}
10:   else
11:     Q ← egy véletlen peer a C_R listából {exploration}
12:   end if
13:   connect(Q) {TCP kapcsolat felépítése}
14:   blockPeer(Q, B_S, blokkidő) {peer blokkolása, alapesetben 4 órára}
15:   Q eltávolítása C_C-ből
16:   if sikeres kapcsolódás Q-hoz then
17:     buddycast_message ← createBuddyCastMessage(Q)
18:     buddycast_message elküldése Q-nak
19:     buddycast_reply fogadása Q-tól
20:     C_C ← fillPeers(buddycast_reply)
21:     addConnectedPeer(Q)
22:     blockPeer(Q, B_R, blokkidő)
23:   end if
24: end loop
```

A kihasználási-felfedezési arány (α)

A SELECTPEER() megvalósítása alapján az alábbiak egyikét hajtja végre:

Algoritmus 11 A BuddyCast passzív szál

loop
  buddycast_reply fogadása Q-tól
  $C_C \leftarrow$ fillPeers(buddycast_reply)
  addConnectedPeer(Q)
  blockPeer(Q, B_R, blokkidő)
  buddycast_message $\leftarrow$ createBuddyCastMessage(Q)
  buddycast_message elküldése Q-nak
  blockPeer(Q, B_S, blokkidő)
  Q eltávolítása C_C-ből
end loop

Algoritmus 12 createBuddycastMsg(Q)

buddycast_msg_send $\leftarrow$ BuddyCastMessage()
msg.preferences $\leftarrow$ a peer 50 legújabb letöltött torrentje
msg.tasteBuddies $\leftarrow C_T$
msg.peerpreferences $\leftarrow$ a C_T-ben levő szomszédok legfeljebb 10 preferenciája
msg.randomPeers $\leftarrow C_R$

Algoritmus 13 addConnectedPeer(Q)

if Q csatlakozható **then**
  $\text{Sim}_Q \leftarrow$ getSimilarity(Q) {Hasonlóság Q és az aktív peer között}
  $\text{Min}_{\text{Sim}} \leftarrow$ a legkevésbé hasonló peer C_T-ben
  if $\text{Sim}_Q \geq \text{Min}_{\text{Sim}}$ **or** (C_T nincs tele **and** $\text{Min}_{\text{Sim}} > 0$) **then**
    $C_T \leftarrow C_T + Q$
    Ha C_T mérete meghaladja a korlátot, a legkevésbé hasonlót átrakjuk C_R-be (onnan esetleg a legrégebbit ki kell dobni)
  else
    $C_R \leftarrow C_R + Q$
    Ha C_R mérete meghaladja a korlátot, a legrégebbi peert kidobjuk
  end if
else
  $C_U \leftarrow C_U + Q$
  Ha C_U mérete meghaladja a korlátot, a legrégebbi peert kidobjuk
end if

– az eddig ismert TB-k felé indít egy kapcsolatot, ezt nevezzük *kihasználás*nak (*exploitation*)

– a véletlen szomszédok közt választ, ezt nevezzük *felfedezés*nek (*exploration*)

Az α paraméter egy 0 és 1 közötti szám, a *kihasználási-felfedezési arány* (*exploitation-to-exploration ratio*). Az algoritmus minden egyes lépésben α valószínűséggel kihasznál, $1 - \alpha$ valószínűséggel felfedez.

3.2. Változtatható paraméterek

A BuddyCast algoritmus eddig ismertetett változatában bizonyos konstans numerikus értékeket használtunk, pl. a listák mérete. Ezen értékek megváltoztathatóak, a különböző értékekkel más és más eredményt érhetünk el. A megváltoztatható értékek a következők (zárójelben egy-egy példa érték):

– C_T, C_R, C_U, C_C méretei (10, 10, 10, 50)

– Az üzenetben a peerek preferencia listájának maximális mérete (50)

– A ciklus hossza (15 másodperc)

– α, a kihasználási-felfedezési arány (0.5)

– Blokkolás ideje (4 óra)

– Az üzenetben küldött

  - saját preferencia lista mérete (50)
  - TB lista mérete (10)
  - TB preferencia listák mérete (20)
  - véletlen peer lista mérete (10)

3.3. PeerSim implementáció

A BuddyCast algoritmus vizsgálatához egy PeerSim nevű szimulátor programot használtam. A szimulátor lehetővé tette, hogy az algoritmus konvergencia, terhelés, tárigény és egyéb tulajdonságait vizsgáljam.

3.3.1. PeerSim

A PeerSim [27, 34] egy nyílt forráskódú P2P szimulátor, amely jól skálázódik nagy hálózatok esetén is. A program Java-ban íródott, amely egy objektum-orientált programozási nyelv. Ennek megfelelően a program felépítése erősen moduláris, így könnyen bővíthető. Az osztályok jól strukturáltak, a P2P hálózatok legfőbb tulajdonságait figyelembe veszik. Sok hasznos osztály rendelkezésre áll, így nekünk azt már nem kell megírni.

3.3.2. Ciklus illetve esemény alapú szimuláció

A PeerSim alapvetően kétféle szimulációt támogat.

Az egyik lehetőség, hogy az összes protokollt egy rögzített sorrendben végrehajtjuk. Az üzenetek lényegében nulla idő alatt érkeznek meg.és elküldésük sorrendjében lesznek feldolgozva. Ezt nevezzük *ciklus alapú*(*cycle-driven*) szimulációnak. Ez a módszer viszonylag jól skálázódik, azonban bizonyos esetekben nem eléggé tükrözi a valóságot.

Ezzel szemben az *esemény alapú* (*event-driven*) szimulációra az jellemző, hogy az üzenetek késleltetése szabályozható, és egy valósághűbb környezetet teremt, azonban kevésbé skálázódik.

3.3.3. A megvalósítás technikai részletei

A BuddyCast egy protokoll, így a PeerSim csomagban[37] található Protocol interfészt valósítja meg. A BuddyCast protokoll megvalósításához az esemény alapú környezetet választottam, így az ennek megfelelő interfészt implementálja a protokoll (EDPROTOCOL). A program négy fő osztályból áll:

- BUDDYCAST, amely maga a megvalósítás, azaz egy esemény alapú protokoll.
- BUDDYCASTINITIALIZER, amely beindítja a gépezetet: inicializálja a superpeereket és elküldi az aktív szálat üzemeltető első ciklikus üzenetet.
- BUDDYCASTMESSAGE, azaz üzenetekben küldött tényleges információ.
- TASTEBUDDY, a Taste Buddy rekord.

A BUDDYCAST osztály a lehetséges paraméterek felsorolásával indít, majd definiálja a listákat és egyéb lokális információkat. A konstruktor feltölti a paramétereket és inicializálja a listákat

Mivel a BuddyCast egyik célja, hogy overlay hálózatot hozzon létre, az osztály megvalósítja a Linkable interfészt, azaz egy szomszédságot definiál. A szomszédok a Taste Buddy listán levő peerek.

A passzív és aktív szálak a korábbiakban definiáltak alapján van megvalósítva. A bootstrapping lényege, hogy a superpeereket megkeresik a peerek. A superpeereknek nincs preferenciájuk, így mindenkivel egyenlően bánnak hasonlóság szempontjából.

Az osztály további, nagyobbik részét a listák karbantartása teszi ki. A hatékonyság érdekében megfelelő adatszerkezeteket kell használni. Az egyes listákhoz más és más műveleteket kritikusak. Ilyenek például a bővítés, keresés, rendezés, véletlen minta, egyesítés. Általánosan elmondható, hogy kompromisszumot kell kötni a tárigény és a futásidő között (*trade-off*). A szűk keresztmetszet gyakran csak nagyobb hálózatok szimulálásakor volt észlelhető. Az optimalizáláshoz nagy segítséget nyújtottak *profiler* eszközök, melyekkel nyomon lehet követni, hogy mely komponensek igényelnek sok tárat vagy időt.

Ez a megvalósítás inkább a futásidőt optimalizálja, így helyenként nagy tárigénye lehet, azonban a szimulációk gyorsan lefutnak.

A teljes forráskód megtalálható a `http://svn.csko.hu/buddycast/` SVN tárolóban.

3.3.4. Forráskód

A következő minta kódrészlet szemlélteti a BuddyCast passzív szál megvalósítást PeerSimben. Mivel a megvalósítás esemény alapú, ezért a PROCESSEVENT() függvényt kell felüldefiniálni. A passzív szálon felül a protokollnak egy periodikusan küldött speciális üzenettel jelezzük, hogy az aktív szálnak le kell futnia. Ezt az üzenetet a protokoll újraidőzíti.

```
333     public void processEvent(Node node, int pid, Object event) {
334         if (event instanceof CycleMessage) {
335             /* Cycle message, schedule an other message in timeToWait time */
336             EDSimulator.add(delay, CycleMessage.getInstance(), node, pid);
337 
338             /* Do the active BuddyCast protocol */
339             work(pid);
340         } else if (event instanceof BuddyCastMessage) {
341             /* Handle incoming BuddyCast message */
342             BuddyCastMessage msg = (BuddyCastMessage) event;
343 
344             /* See if the peer is blocked */
345             if (isBlocked(msg.sender, recvBlockList)) {
346                 return;
347             }
348 
349             int changed = 0;
350             changed += addPreferences(msg.sender, msg.myPrefs);
351 
352             /* Use the Taste Buddy list provided in the message */
353             for (TasteBuddy tb : msg.tasteBuddies.values()) {
354                 if (addPeer(tb.getNode()) == 1) { /* Peer newly added */
355                     updateLastSeen(tb.getNode(), tb.getLastSeen());
356                 }
357                 changed += addPreferences(tb.getNode(), tb.getPrefs());
358 
359                 addConnCandidate(tb.getNode(), tb.getLastSeen());
360             }
361 
362             /* Use the Random Peer list provided in the message */
363             for (Node peer : msg.randomPeers.keySet()) {
364                 if (addPeer(peer) == 1) { /* Peer newly added */
365                     updateLastSeen(peer, msg.randomPeers.get(peer));
366                 }
367                 addConnCandidate(peer, msg.randomPeers.get(peer));
368             }
369 
370             /* Put the peer on our lists */
371             addPeerToConnList(msg.sender, msg.connectible);
372 
373             /* If the message wasn't a reply to a previous message */
374             if (msg.reply == false) {
375                 /* If the sender is not blocked as a recipient */
376                 if (!isBlocked(msg.sender, sendBlockList)) {
377                     /* Create the reply message */
378                     BuddyCastMessage replyMsg = createBuddyCastMessage(msg.sender);
379                     /* Set the sender field */
380                     replyMsg.sender = CommonState.getNode();
381                     /* It's a reply */
382                     replyMsg.reply = true;
383                     /* Send the message */
384                     Node senderNode = msg.sender;
385                     ((Transport) node.getProtocol(FastConfig.getTransport(pid))).send(
386                             CommonState.getNode(),
387                             senderNode,
388                             replyMsg,
389                             pid);
390                     /* No longer a candidate */
391                     removeCandidate(msg.sender);
392                     /* Block the peer so we won't send too many messages to them */
393                     blockPeer(msg.sender, sendBlockList);
394                 }
395             }
396             /* Block the peer so we won't receive too many messages from them */
397             blockPeer(msg.sender, recvBlockList);
398         }
399     }
```

A BuddyCast eseménykezelő eljárás

4. fejezet

A BuddyCast algoritmus kiértékelése ajánlórendszereken

4.1. Ajánlórendszerek

A BuddyCast algoritmust egy speciális feladaton szimuláltam, amely az ajánlás feladata. Az ajánlás feladata[2]: adott felhasználók (*user*) C halmaza és termékek (*item*) S halmaza. C és S lehetnek egészen nagyok is, akár milliós nagyságrendűek is. Legyen u a hasznosság függvénye, $u : C \times S \rightarrow R$, ahol R egy megadott intervallumba eső valós vagy nemnegatív egész számok. Minden egyes $c \in C$ felhasználóra meg kell keresnünk azt az $s' \in S$ terméket, amelynek a haszonértéke a legmagasabb, azaz

$$\forall c \in C, \quad s'_c = arg\,_{x \in S} max(c, s) \tag{4.1}$$

Az u tetszőleges függvény lehet, azonban sokszor egy értékeléssel van reprezentálva, pl. egy 10-es skálán 7.

Az *ajánlórendszer*ek (*recommendation system*) tehát olyan rendszerek, amelyek a felhasználók választásainak ismerete mellett a felhasználók számára egy döntést hoznak meg: *vajon melyik az termék, amelyet a felhasználó legjobban preferálna?* A termék sokféle lehet, pl. film, TV műsor, zene, kép, újsághír, weboldal, személy, stb...

Ezeket az algoritmusokat, amelyek a felhasználók korábbi döntéseire alapulnak, a *collaborative filtering* (röviden: *CF*) algoritmusok[35] közé soroljuk. Azon az egyszerű heurisztikán alapszanak, hogy azok a felhasználók, akik a korábbi döntéseikben egyeztek (nem egyeztek), a továbbiakban is egyezni (nem egyezni) fognak. A CF algoritmusoknak több változata van, ezek közül egy lehetséges a *felhasználó alapú CF* (*user-based CF*). Ez a módszer egy felhasználó egy termékre adott értékelésének modellezéséhez aggregálja a többi felhasználó értékelését. Egy széles körben elterjedt aggregáció a következő[41]:

$$\hat{r}_{u,i} = \frac{\sum_{v \in N_u} s_{u,v} \left(r_{v,i} - \bar{r}_v \right)}{\sum_{v \in N_u} |s_{u,v}|} + \bar{r}_u \tag{4.2}$$

ahol $r_{u,i}$ és $\hat{r}_{u,i}$ az ismert és a jósolt értékelése az i terméknek az u felhasználó által, $\bar{r}_u$ és N_u jelentik az átlagos értékelést és a felhasználó szomszédait, $s_{u,v}$ megadja az u és v felhasználók hasonlóságát (például koszinusz hasonlóság [3] vagy Pearson hasonlóság).

Adatbázis	Jester	MovieLens	BookCrossing
Felhasználók száma	73421	71567	77806
Termékek száma	100	10681	185974
Tanító adatbázis mérete	3695834	9301274	397011
Tanító adatbázis ritkasága	5.03E-01	1.22E-02	2.74E-05
Lehetséges értékek	-10, . . . , 10	1, . . . , 5	1, . . . , 10
Kiértékelő adatbázis mérete	440526	698780	36600
MAE	0.93948	4.52645	2.43277

4.1. táblázat. Az adatbázisok tulajdonságainak összehasonlítása

Az ajánlórendszerek elfogadhatóan működnek centralizált esetben, amikor minden adat a felhasználókról és a termékekről egy központi rendszerben tárolódnak. Ebben a fejezetben azonban egy teljesen elosztott P2P rendszerben szimuláljuk az ajánlás feladatát, [35] nyomán. Feltételezzük, hogy a felhasználók és az értékelések statikusak, a szimuláció során nem változnak. Algoritmusunk feladata, hogy egy megfelelő overlay hálózatot építsen, az ajánlás minősége csak a Taste Buddy listák tartalmától függ.

4.2. A felhasznált tanuló adatbázisok jellemzése

A szimulációkat három különböző adatbázison futtattam, név szerint a MovieLens [19], Jester [17] és Book Crossing [51] adatbázisokon. A következőkben bemutatom ezeket.

A 4.1. táblázat bemutatja a három adatbázis alapvető tulajdonságait. A ritkaság (sparsity) a jelen levő és a lehetséges értékelések számának hányadosát jelenti. Látható, hogy a ritkaság tekintetében az adatbázisok igen változatosak és a Book Crossing adatbázis kifejezetten ritka. Ez jelentős hatással lesz az algoritmusunkra.

Ahhoz, hogy egy ajánlórendszer teljesítményét kiértékeljük, az adatbázis tanító és kiértékelő részre lett bontva. Az algoritmusunk a tanító adatbázist használja arra, hogy a modellt felépítse és döntéseket hozzon a kiértékelő adatbázison, amelyet aztán összevetünk a tényleges értékelésekkel. Az MAE (Mean Absolute Error) egy szokványos alapvető mértéke a jóslás teljesítményének, az átlagos abszolút hiba,

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|f_i - y_i| \tag{4.3}$$

Szimulációnk során a táblázatban található MAE értékeket szeretnénk elérni teljesen elosztott P2P rendszerben a BuddyCast overlay építő algoritmussal.

4.3. A mérések

Miután a BuddyCast algoritmus PeerSim megvalósítása elkészült, elkészítettem a megfelelő konfigurációs állományokat. Ezekben beállítottam a node-ok számát (amely egyenlő a tanuló adatbázis méretével) és betöltöttem a tanító adatbázist.

A futás technikai részletei a következők voltak:

– kiindulási overlay hálózat egy 10 fokszámú véletlen gráf

– ciklusidő: 15 másodperc

– a Taste Buddy lista maximális mérete 100

– a hasonlóság mátrix előre ki volt számolva, ezzel növelve a sebességet és a memóriaigényt

– a peerek egymásnak a teljes preferencia listájukat átküldték, valójában csak egy referenciát, ezzel nagyban csökkentve a memóriaigényt

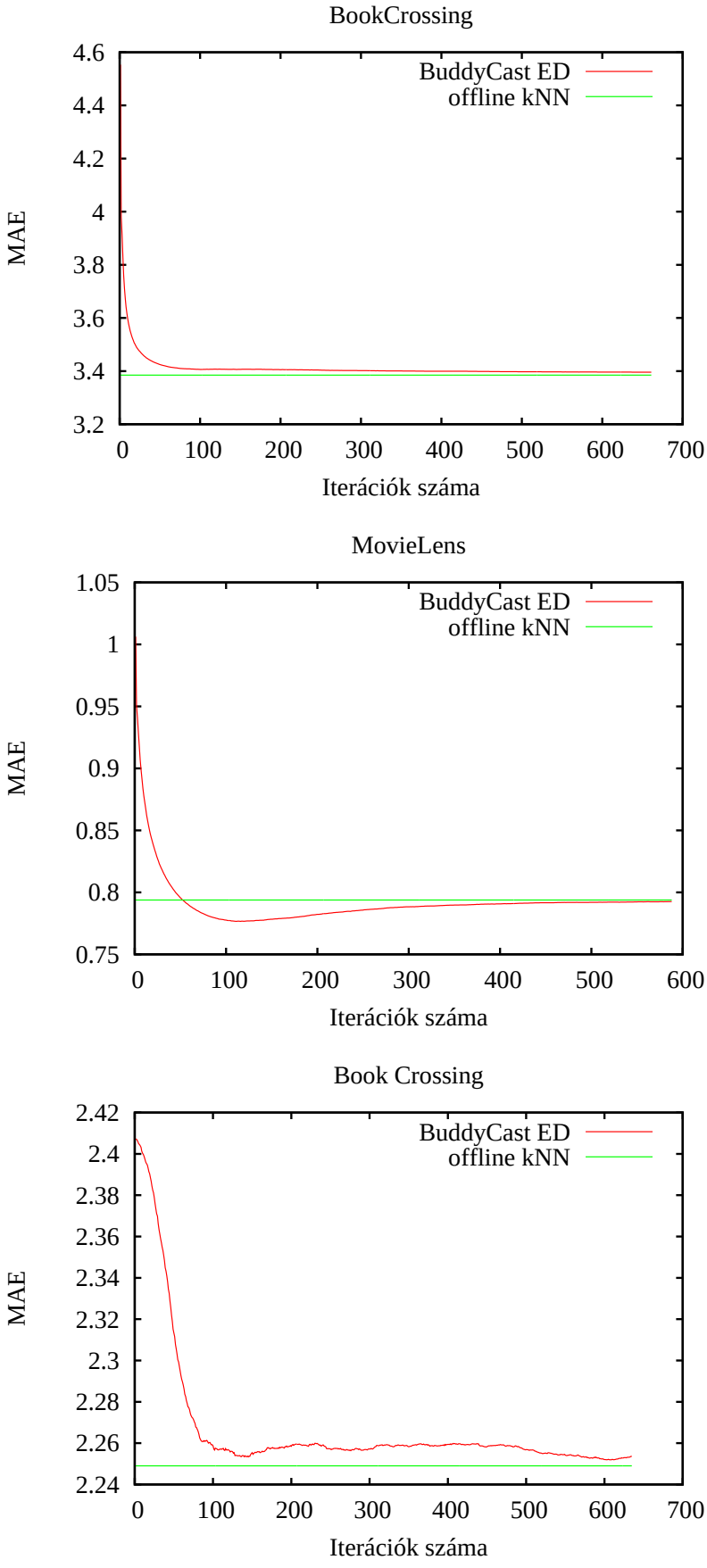
A szimulációkat Linux operációs rendszeren futtattam, Java 1.6-os környezetben, egy Dell PowerEdge R710 szervergépen, amelyben 2 darab egyenként 4-magos Intel Xeon X5560 processzor van 48 GB memóriával és 1 TB merevlemezzel. A futtatókörnyezet tehát nagyon előnyös volt, a futások sok memóriát igényeltek és több órán keresztül futottak.

Az 4.1 ábra bemutatja az algoritmus teljesítményét az egyes adatbázisokon. A futások két szakaszra bonthatóak. Az első szakasz 100-200 iterációig tart, amikor az algoritmus kezdi nagyvonalakban felépíteni a szomszédsági listák legfontosabb elemeit. Az algoritmus ezután további 200-300 iterációban javít az ajánlás értékén. Látható, hogy mindhárom esetben az algoritmus jól konvergál. A MovieLens esetében az algoritmus az offline értékek alá képes menni. A Book Crossing esetében van a legnehezebb dolga, mert az egy nagyon ritka adatbázis.

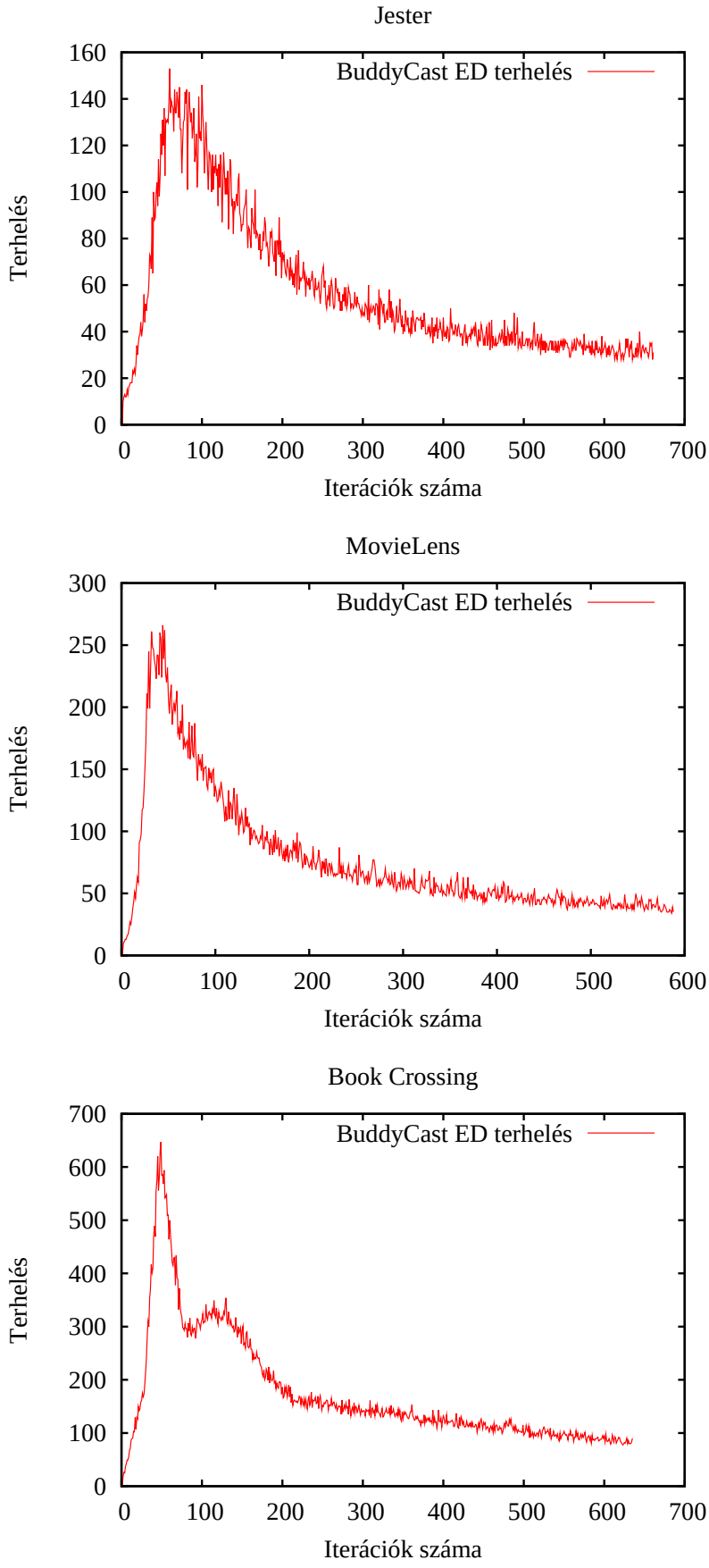
Az 4.1 ábra bemutatja az algoritmus által generált terhelést az egyes peereken. A terhelés mértéke a maximális szelekciók száma, azaz, hányszor választotta a SELECTPEER() célpontul azt a peert, amelyet a legtöbbször választották.

Az ábrákról leolvasható, hogy az algoritmus első szakaszában a terhelés igen nagy értékeket vesz föl. Ez kifejezetten a Book Crossing adatbázis esetén óriási, amely egyben a legritkább adatbázisunk. Egyes csomópontoknak akár 600 kérést is teljesíteniük kell egy 15 másodperces szakaszban, amely terhelés nem megengedhető, nagy valószínűséggel elérhetetlenné teszi a csomópontot.

Mindhárom esetben 100-200 iteráció után a terhelés csökkenő tendenciát mutat. Ez a blokklisták használata miatt van, amelyek gondoskodnak arról, hogy egy peert ne keressünk meg újra egy bizonyos időközön belül.



4.1. ábra. A három adatbázison futtatott BuddyCast esemény alapú szimuláció MAE eredményei



4.2. ábra. A három adatbázison futtatott BuddyCast esemény alapú szimuláció terhelés eredményei

5. fejezet

Függelék

Nyilatkozat

Alulírott programtervező informatikus szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem Informatikai Tanszékcsoport Mesterséges intelligencia tanszékén készítettem, programtervező informatikus diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat a Szegedi Tudományegyetem könyvtárában, a kölcsönözhető könyvek között helyezik el.

Szeged, 2010. május 15. .....

aláírás

Köszönetnyilvánítás

Megköszönöm Jelasity Márk témavezetőmnek, hogy biztosította a technikai hátteret és tanácsaival segítette a szakdolgozat elkészülését.

Köszönet Ormándi Róbertnek és Hegedűs Istvánnak az adatbázisok előkészítéséért és a további technikai jellegű támogatásukért.

Köszönöm Vinkó Tamásnak, hogy a BuddyCast-tal kapcsolatos tapasztalatát megosztotta velem és saját szimulátorával segítette munkámat.

Irodalomjegyzék

[1] Eytan Adar and Bernardo A. Huberman. Free riding on gnutella. *First Monday*, 5:2000, 2000.

[2] Gediminas Adomavicius and Alexander Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, 17(6):734–749, 2005.

[3] Gediminas Adomavicius and Er Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17:734–749, 2005.

[4] National Security Agency. Secure hash standard. *Federal Information Processing Standards Publication 180-1*, April 1995.

[5] S. Bellovin. Security aspects of napster and gnutella. 2001.

[6] Andrew Biggadike, Daniel Ferullo, Geoffrey Wilson, and Adrian Perrig. NATBLASTER: Establishing TCP connections between hosts behind NATs. In *Proceedings of ACM SIGCOMM ASIA Workshop*, April 2005.

[7] Zhijia Chen, Yang Chen, Chuang Lin, Vaibhav Nivargi, and Pei Cao. Experimental analysis of super-seeding in bittorrent. In *ICC*, pages 65–69, 2008.

[8] Zhijia Chen, Chuang Lin, Yang Chen, Vaibhav Nivargi, and Pei Cao. An analytical and experimental study of super-seeding in bittorrent-like p2p networks. *IEICE Transactions*, 91-B(12):3842–3850, 2008.

[9] Bram Cohen. Incentives build robustness in bittorrent. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, 2003.

[10] L. D'Acunto, J. A. Pouwelse, and H. J. Sips. A measurement of NAT and firewall characteristics in peer-to-peer systems. In Lex Wolters Theo Gevers, Herbert Bos, editor, *Proc. 15-th ASCI Conference*, pages 1–5, P.O. Box 5031, 2600 GA Delft, The Netherlands, June 2009. Advanced School for Computing and Imaging (ASCI).

[11] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, New York, NY, USA, 1987. ACM.

[12] P. T. Eugster, R. Guerraoui, A. M. Kermarrec, and L. Massoulie. Epidemic information dissemination in distributed systems. *Computer*, 37(5):60–67, May 2004.

[13] F. Audet and C. Jennings. Network Address Translation (NAT) Behavioral Requirements for Unicast UDP. *Internet RFC 4787*, January 2007.

[14] Bryan Ford and Pyda Srisuresh. Peer-to-peer communication across network address translators. In *In USENIX Annual Technical Conference*, pages 179–192, 2005.

[15] Aditya Ganjam and Hui Zhang. Connectivity restrictions in overlay multicast. In *Proc. 14th international workshop on Network and operating systems support for digital audio and video (NOSSDAV '04)*, pages 54–59, New York, NY, USA, 2004. ACM.

[16] P. Garbacki, A. Iosup, J. Doumen, J. Roozenburg, Y. Yuan, Ten M. Brinke, L. Musat, F. Zindel, F. van der Werf, M. Meulpolder, and Others. Tribler protocol specification.

[17] Ken Goldberg, Theresa Roeder, Dhruv Gupta, and Chris Perkins. Eigentaste: A constant time collaborative filtering algorithm. *Information Retrieval*, 4(2):133–151, 2001.

[18] Saikat Guha and Paul Francis. Characterization and measurement of tcp traversal through nats and firewalls. In *Internet Measurment Conference*, pages 199–211, 2005.

[19] Jonathan L. Herlocker, Joseph A. Konstan, Al Borchers, and John Riedl. An algorithmic framework for performing collaborative filtering. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR '99)*, pages 230–237, New York, NY, USA, 1999. ACM.

[20] J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. *IETF draft draft-ietf-mmusic-ice-19*.

[21] J. Rosenberg and J. Weinberger and C. Huitema and R. Mahy. STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). *Internet RFC 3489*, March 2003.

[22] J. Rosenberg and R. Mahh and P. Matthews. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). *IETF draft draft-ietf-behave-turn-16*.

[23] J. Rosenberg and R. Mahy and P. Matthews and D. Wing. Session Traversal Utilities for NAT (STUN). *Internet RFC 5389*, October 2008.

[24] Márk Jelasity, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In Hans-Arno Jacobsen, editor, *Middleware 2004*, volume 3231 of *Lecture Notes in Computer Science*, pages 79–98. Springer-Verlag, 2004.

[25] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems*, 23(3):219–252, August 2005.

[26] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. T-Man: Gossip-based fast overlay topology construction. *Computer Networks*, 53(13):2321–2339, 2009.

[27] Márk Jelasity, Alberto Montresor, Gian Paolo Jesi, and Spyros Voulgaris. The Peersim simulator. `http://peersim.sf.net`.

[28] Mark Jelasity and Maarten Van Steen. Large-scale newscast computing on the internet. Technical report, Technical Report IR-503, Vrije Universiteit Amsterdam, Department of Computer Science, Amsterdam, The Netherlands, October 2002.

[29] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3):8, August 2007.

[30] Raúl Jiménez, Flutra Osmani, and Björn Knutsson. Connectivity properties of mainline bittorrent dht nodes. In *Proc. 9th International Conference on Peer-to-Peer Computing*, pages 262–270, 2009.

[31] Thomas Karagiannis, Andre Broido, Nevil Brownlee, Kimberly C. Claffy, and Michalis Faloutsos. Is p2p dying or just hiding? In *Proceedings of the GLOBECOM 2004 Conference*, Dallas, Texas, November 2004. IEEE Computer Society Press.

[32] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 258–259, New York, NY, USA, 2002. ACM.

[33] J. J. D. Mol, J. A. Pouwelse, D. H. J. Epema, and H. J. Sips. Free-riding, fairness, and firewalls in P2P file-sharing. In *Proc. 8th IEEE International Conference on Peer-to-Peer Computing*, pages 301–310. IEEE CS, sep 2008.

[34] Alberto Montresor and Márk Jelasity. Peersim: A scalable P2P simulator. In *Proceedings of the Ninth IEEE International Conference on Peer-to-Peer Computing (P2P 2009)*, pages 99–100, Seattle, Washington, USA, September 2009. IEEE. extended abstract.

[35] Róbert Ormándi, István Hegedűs, and Márk Jelasity. Overlay management for fully distributed user-based collaborative filtering. In *Euro-Par 2010*, 2010.

[36] P. Srisuresh and B. Ford and D. Kegel. State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs). *Internet RFC 5128*.

[37] Peersim. Javadoc documentation. http://peersim.sourceforge.net/doc/index.html.

[38] Universal Plug and Play. http://www.upnp.org/.

[39] J.A. Pouwelse, J. Yang, M. Meulpolder, D.H.J. Epema, and H.J. Sips. Buddycast: an operational peer-to-peer epidemic protocol stack. In G.J.M. Smit, D.H.J. Epema, and M.S. Lew, editors, *Proc. of the 14th Annual Conf. of the Advanced School for Computing and Imaging*, pages 200–205. ASCI, 2008.

[40] Johan Pouwelse. Video search and playback in zero-server p2p systems, p2p 2010. September 2008.

[41] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. Grouplens: an open architecture for collaborative filtering of netnews. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work (CSCW '94)*, pages 175–186, New York, NY, USA, 1994. ACM.

[42] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. *Peer-to-Peer Computing, IEEE International Conference on Peer-to-Peer Computing (P2P'01)*, 0:0099, 2001.

[43] Ion Stoica, Robert Morris, David Karger, Frans M. Kaashoek, and Hari. Chord: A scalable peer-to-peer lookup service for internet applications, 2001.

[44] Norbert Tölgyesi and Márk Jelasity. Adaptive peer sampling with newscast. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009*, volume 5704 of *Lecture Notes in Computer Science*, pages 523–534. Springer-Verlag, 2009.

[45] Tribler.org. http://www.tribler.org/.

[46] Tribler.org. The current tribler protocol specification.

[47] Tribler.org. Measurements of nat/firewall characteristics. http://www.tribler.org/trac/wiki/NATMeasurements.

[48] Susu Xie, Gabriel Y. Keung, and Bo Li. A measurement of a large-scale peer-to-peer live video streaming system. *Parallel Processing Workshops, International Conference on*, 0:57, 2007.

[49] C. Zhang, P. Dhungel, D. Wu, and K. W. Ross. Unraveling the bit-torrent ecosystem. In *Technical Report, Polytechnic Institute of NYU*, May 2009.

[50] Chao Zhang, Prithula Dhungel, Di Wu, Zhengye Liu, and Keith W. Ross. Bittorrent darknets. In *Proceedings of IEEE Conference on Computer Communications (IEEE INFOCOM '10)*, March 2010.

[51] Cai-Nicolas Ziegler, Sean M. McNee, Joseph A. Konstan, and Georg Lausen. Improving recommendation lists through topic diversification. In *Proceedings of the 14th international conference on World Wide Web (WWW '05)*, pages 22–32, New York, NY, USA, 2005. ACM.