Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

Modern peer-to-peer hálózatokon alapuló megoldások hatékonyságvizsgálata

Tudományos Diák Konferencia dolgozat

2016.

Szerző:
Pásztor Dániel, AXEG1Q

Konzulens:
Dr. Ekler Péter, egyetemi adjunktus
Automatizálási és Alkalmazott Informatikai Tanszék

Tartalom

1. Összefoglaló

A dolgozatom aktualitását a peer-to-peer hálózatok töretlen népszerűsége adja, melynek elosztott jellege rengeteg kérdést vet fel mind hatékonysági, mind biztonságtechnikai szempontból. Bár maga az ötlet szinte egyidős az internettel, igazán elterjedni csak a 2000-es évek elején kezdett, amikor elérhetővé váltak a kereskedelmi szélessávú internetszolgáltatások.

Az első alkalmazás, mely népszerűvé tette a peer-to-peer (P2P) technológia fogalmát, az 1999-ben kiadott Napster volt, a felhasználók között közvetlen fájlmegosztást hozott létre. Bár nem erre szánták, de a rengeteg jogvédett tartalom (elsősorban zene) megosztása miatt később kénytelenek voltak visszavonni ezt a szolgáltatásukat. Ekkor viszont már több millió felhasználójuk volt, akik megismerkedtek a fájlmegosztás lehetőségeivel, így ők ezek után más alternatívák után kezdtek keresni.

A Napster bezárása után számtalan alkalmazás jött létre, de a P2P fájlmegosztás következő fontos állomása a BitTorrent protokoll, illetve az azt implementáló alkalmazás megalkotása lett. A protokoll teljesen nyílt felhasználású, bárki tehet hozzá javaslatokat, amik idővel akár a fő protokollba is bekerülnek, illetve könnyedén ki is tudja egészíteni saját funkciókkal. Nyílt forráskódja, a kisméretű alkalmazás és a viszonylag egyszerű protokoll miatt hamar elterjedt, a mai napokban szinte kizárólag ezt használják fájlmegosztásra.

Az eredeti megjelenése óta azonban számos kiegészítés jött ki hozzá, melynek legfőbb célja a fájlmegosztás hatékonyabbá, illetve biztonságosabbá tétele volt. Dolgozatomban ezeket a különböző kiegészítéseket fogom megnézni, különös figyelmet fordítva a DHT alkalmazására, melynek hatékonyságát szimulátorban fogom ellenőrizni. Bár a ma ismert egyik legnagyobb DHT rendszer pont a BitTorrent alkalmazások által alkotott rendszer, számos más konkrét és potenciális felhasználási területei vannak, melyekre a dolgozatom végén fogok kitérni.

2. Bevezetés

Ebben a fejezetben bemutatom a peer-to-peer hálózatok fogalmát, történetét, majd ezt követve a BitTorrent protokoll különböző tulajdonságait.

2.1. Peer-to-peer hálózatok

Az interneten használt protokollokat két különböző csoportra oszthatjuk aszerint, hogy a protokollban résztvevők milyen szerepet töltenek be:

A *szerver-kliens* modellben egy (vagy pár) szerver szolgál ki több klienst. Tipikus használata például a weboldalaknál látható, ahol a böngésző (kliens) elkéri az oldal szolgáltatójától (szerver) a megjelenítésre szánt weboldalt.

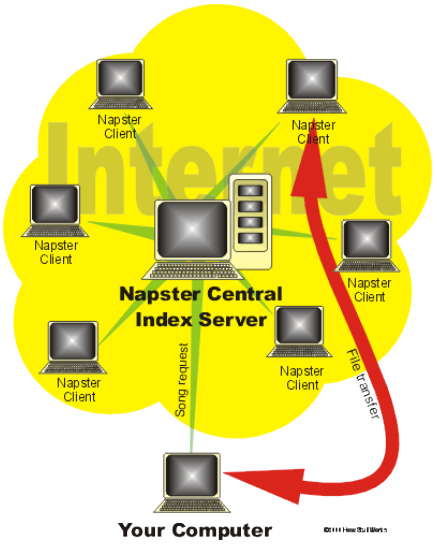
Ábra 1. Szerver-kliens, illetve peer-to-peer modell [1]

A *peer-to-peer* (röv. P2P) modellben nincsen kitüntetett szerep, a résztvevők (*peerek*) mind azonos szerepet töltenek be. Ilyen például egy Skype-beszélgetés, ahol a résztvevők közvetlen egymással kommunikálnak.

Az internethez csatlakozó minden eszköznek van címe (IP-cím), mellyel tudják azonosítani magukat. Még mielőtt bármilyen kommunikáció létrejöhetne, a hálózatban a feleknek meg kell ismerniük egymást. Ezt egy szerver-kliens modellben általában egyszerű megoldani: a szervernek vagy állandó az IP-címe, ami be van égetve a kliensekbe, vagy pedig DNS feloldással kanonikus nevekből (pl. *google.com*) kérhetjük el a címét. Miután a kliens küld egy üzenetet, a szervernek elég megnézni, honnan érkezett az üzenet, és oda küldi vissza a választ.

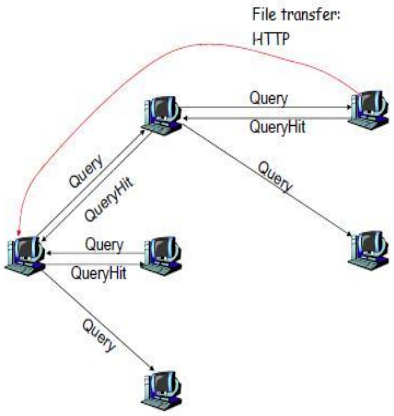
Peer-to-peer hálózatoknál ezt már nehezebb megoldani. Az egyik probléma, hogy a legtöbb eszközt általában egy *router* köti össze az internettel, ami a NAT nevezetű folyamat miatt lehetetlenné teszi, hogy valamilyen router beállítás módosítás vagy külső koordináció nélkül két peer között létrejöjjön a kapcsolat. A dolgozatomban ezzel a problémakörrel majd később fogok foglalkozni. Egy másik probléma, amivel még foglalkozni fogok, hogy hogyan találhatunk más résztvevőt az interneten a protokollunkhoz.

A már említett Napster egy központi szervert üzemeltetett arra, hogy számon tartsa a felhasználók IP-címét, illetve az általuk megosztott fájlokat. Ha egy felhasználó rákeresett egy fájl-névre, akkor ez a szerver küldte vissza, milyen IP-címeken tudja elérni a fájlt. Ezután a felhasználó felvette a kapcsolatot az adott eszközzel, és megindulhatott a fájlcsere. Végül ez a központi szerver pecsételte meg a Napster sorsát: hosszú pereskedés után utasították, hogy kapcsolják le a szervert, a nélkül pedig a felhasználók nem tudták megtalálni egymást, és így kihalt a hálózat.



Ábra 2. Keresés a Napster hálózatában [2]

A Napster sikerét követte a Gnutella alapú alkalmazások. Az ilyen hálózatok teljesen decentralizáltak voltak, nem volt egy központi szerver. Egy fájl keresésekor a felhasználó az általa ismert összes csomóponthoz elküld egy kérést az adott fájl után, melyet ezután az összes csomópont továbbít az általuk ismert összes csomóponthoz, egészen addig, ameddig a keresés el nem ér egy bizonyos mélységet. Ha bármelyik résztvevő azt látja, hogy nála megtalálható a fájl, egy üzenettel jelzi azt a keresés indítójának, mellyel megindulhat a fájlcsere. Bár az elején jól működött, a Napster összeomlása után bekövetkező rengeteg új felhasználó hatására kiderült, hogy ez a rendszer nem skálázható jól, sok eszköz esetén nehezen és lassabban fog menni a keresés, valamint a fájlmegosztás is. Ezeket a problémákat későbbi kiadásokban orvosolták.



Ábra 3. Keresés a Gnutella hálózatában [3]

2.2. BitTorrent protokoll

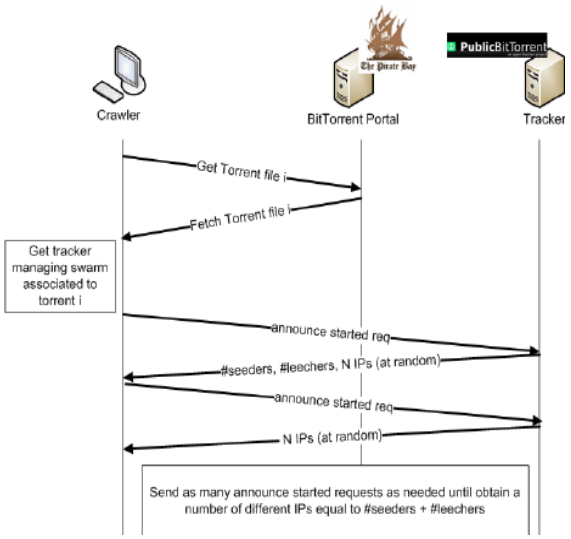
A BitTorrent protokollt 2001-ben adták ki az azt implementáló alkalmazással, forráskóddal és dokumentációval együtt, mellyel bárki elkészíthette a saját torrent alkalmazását, bármilyen platformra. A legfontosabb célkitűzései közé tartozott a minél gyorsabb, hatékonyabb és

skálázhatóbb rendszer létrehozása. Ezeket a már akkor is létező technológiák ötvözéseiből sikerült elérni.

Azt már a Napster elődjénél, a Usenet-nél is ismerték, hogyha egy nagy fájlt több kisebb csomagra bontják, és úgy töltik fel, sokkal hatékonyabb lesz a megosztás, ugyanis ilyenkor hiba vagy kapcsolatszakadás esetén elég volt csak a hibás részt újra tölteni, nem kellett az egész fájlt elölről kezdeni. Ennek analógiájára a BitTorrent protokollban is a megosztásra váró fájlokat több részre (*piece*) bontjuk, melyeket megérkezésükkor egyben le is tudunk ellenőrizni.

A protokoll alapját a *.torrent* fájl képezi. Ez tekinthető egy leíró-fájlnak: tartalmazza többek között a megosztásra kerülő fájlok neveit, az egyes részek ellenőrző *hash* (lsd. később) kódját, magát a torrentet azonosító hash kódot, illetve a *trackerek* listáját. Ezt a fájlt a felhasználónak kell valahogy megszereznie, álltalában különböző torrent-weboldalakról.

Egy tracker leginkább a Napster központi szerveréhez hasonlítható, egy kritikus különbséggel. Hasonlóan tartalmazza azoknak a felhasználóknak az IP címeit, akik részt vesznek a torrent megosztásában, viszont csak a torrentek azonosítóját ismeri, azt nem, hogy az milyen fájlokat tartalmaz, így nehezebb jogilag bizonyítani, hogy a tracker jogsértést követ-e el, ha rajta keresztül egy jogvédett fájlt osztanak meg. A Napster-hez hasonlóan egy peer itt is a trackertől kéri el, kik vesznek részt a torrent megosztásában, hogy utána felvegye velük a kapcsolatot.



Ábra 4. Egy fájl letöltése a BitTorrent protokoll segítségével

Az eredeti specifikációban [4] egy torrenthez egyetlen tracker volt rendelve, így ha az nem volt elérhető, a letöltés se indulhatott meg. Későbbi kiegészítések engedélyezték egy torrenthez több tracker (egy tracker-lista) rendelését, melyek helyettesítik a kiesőket, illetve bevezették a *Peer Exchange* (PEX) protokollt, mellyel peerek már egymás között is megoszthatták az általuk ismert résztvevőket, így még a tracker megszűnése esetén is lehetséges új peerek felfedezése.

A letöltés hatékonyságát a kölcsönös megosztás (*tit-for-tat*) elve teszi ideálissá. Egy felhasználó szívesebben osztja meg olyannal a fájl egy részét (*piece-t*), aki számára is küldött már adatot. Ezzel a technikával könnyű kiszűrni a kártékony letöltőket, akik nem akarnak visszatölteni a hálózatba, vagy szándékosan lassítva, akár hibás adatot osztanak meg.

A fájlok sok részre való osztása jelentősen felgyorsítja a fájlmegosztást is. Egy új torrent esetén egy résztvevő már abban a pillanatban megosztóvá tud válni, amint sikerül letöltenie az első piece-t. A BitTorrent hatékonyságát mi sem bizonyítja jobban a rengeteg (többek között

legális) felhasználása. Talán az egyik legjobb példa a világ legnagyobb MMORPG, a World of Warcraft javításainak (*patch*) letöltése. Egy ilyen patch a kisebb (heti) 100Mb-tól akár a 10Gb-os nagyságrendig is terjedhet, ezt pedig a több milliós aktív felhasználóbázisnak kell minél gyorsabban eljuttatni, melyet egy saját alkalmazáson keresztül teszik meg, a BitTorrent protokoll segítségével.

Bár a torrentezést tipikusan P2P alkalmazásnak tekintik, mint az imént is írtam, a fájlmegosztás beindításához szükséges egy központi, tracker szerver. Már nem egyszer előfordult, hogy egy tracker beszüntetése után rengeteg (többek között legális) fájl vált elérhetetlenné, pedig megosztók lettek volna. Ennek megoldására vezették be a trackerless torrenteket, melyek az *elosztott hash táblát* (*Distributed Hash Table, DHT*) használják peerek keresésére.

3. Irodalomkutatás

Az egyik leghíresebb peer-to-peer fájlmegosztó protokoll, a BitTorrent [4], melynek részletes specifikációja megtalálható a jelzett forrásban. A kötelező elemeken kívül ajánlásokat is tesznek bizonyos algoritmusok implementációjára, melyeket sikeresen alkalmaztak már más klienseknek, így itt minden szükséges információ elérhető egy jól működő torrent kliens készítéséhez.

A BitTorrent protokollt számos módon egészítették ki, ezek egyike Mainline DHT [5], mely a torrentek alapvető működéséhez szükséges trackerek kiváltásáért volt felelős. A megjelölt oldalon található ennek a specifikációja. Mivel már szinte az összes torrent alapbeállításként be van kapcsolva, így az MLDHT napjaink egyik legnagyobb elosztott hash-tábláját alkotja.

Peer-to-peer hálózatok különböző paramétereit nehéz egy jól kontrollált, stabil környezetben megvizsgálni. Ennek megoldására fejlesztették ki a szerzők a PeerSim nevű szimulációs környezetet [6], melyben könnyedén megírhatjuk a saját protokollunkat, és azt így különböző körülmények között vizsgálhatjuk. Akár egy milliós nagyságrendű hálózatot is képes szimulálni, mellyel messze felülemelkedik a legtöbb hasonló programon.

A PeerSim szimulációs környezet megértésén segít egy egyszerű protokoll megismerése. Először egy, a ciklus alapú modellben implementált protokollt [7] vizsgáltam meg, majd ugyanennek a protokollnak az eseményvezérelt változatát tanulmányoztam [8]. Ezekkel sikerült egy átfogó képet kapnom a PeerSim képességeiről.

Ehhez a környezethez letölthető számos, mások által írt protokoll, többek között a MLDHT alapját képező Kademlia protokoll [9]. A letölthető fájlok között található egy részletesebb leírás is, melyben ismertetik a különböző implementál osztályok funkcióit, majd egy egyszerű kísérlet során megmérik az így létrehozott hálózatban a keresési időt.

A DHT elengedhetetlen része a legtöbb mai teljesen decentralizált P2P fájlmegosztó rendszereknek. A teljes elosztottság miatt viszont felmerülnek bizonyos támadási lehetőségek, mellyel a hálózat bizonyos tulajdonságait tudják befolyásolni. Ezek egy része a Sybil típusú támadás, melynek fő tulajdonsága, hogy a (rendszeresen P2P) rendszerek bizalom alapú részét játszák ki azzal, hogy rengeteg hamis felhasználót juttatnak a rendszerben, melyeket a támadó irányít [10]. (A nevét egy híres disszociatív személyiségben szenvedő betegről kapta.) A megjelölt forrásban két ilyen típusú támadást elemeznek ki, illetve úgynevezett „*honeypot*”-okkal (kártékony csomópontok becsalogatására szolgáló eszköz) vizsgálják a kártékony csomópontok jelenlétét a valós hálózatban.

A peer-to-peer hálózatok fontos feladata a csomópontok közötti kapcsolatok létrehozása. Bár a Kademlia alapú rendszerek feltételezik a résztvevő csomópontok tranzitivitását, ez sajnos nem mindig teljesül az interneten. Ennek kiszűrésére a különböző implementációk különböző megoldásokat adtak, az egyes hálózatokban mérhető keresési idők pedig jól jellemzik a megoldás hatékonyságát. Ennek mérése során [11] fedezték fel, hogy a MLDHT-ban használt algoritmus hatástalan az interneten található csomópontok nagyobbik része ellen, így pedig az itt mérhető keresési idők is nagyságrendekkel nagyobbak.

Ahhoz, hogy a jövőben még inkább elterjedjenek a DHT-t használó alkalmazások, fontos paraméter a kereséshez szükséges idő. Már vannak ötletek például videók keresésére, illetve megtekintésére DHT hálózatokon keresztül, ehhez viszont elengedhetetlen, hogy az adatot tartalmazó csomópontot minél hamarabb megtalálják. A MLDHT bizonyos paramétereinek módosításával sikerült egy másodperc alatti keresések elvégzésére [12], mely messze felülmúlja a most alkalmazott megoldásokat.

4. Az elosztott hash tábla (DHT)

Ebben a fejezetben ismertetem az elosztott hash táblákat először általánosan, majd a kiegészített BitTorrent protokollban használt változatát.

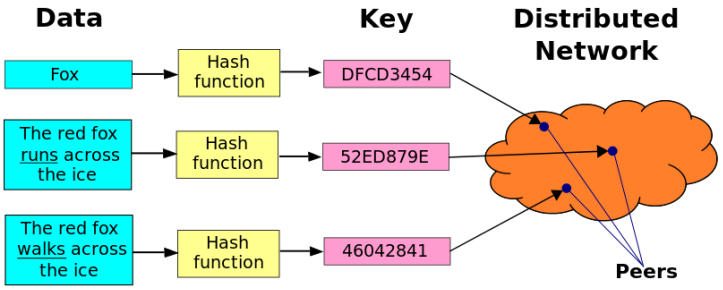
4.1. Bevezetés

A tábla lényegében egy asszociatív tömb, melyben kulcs-érték párokat tudunk tárolni. A hash tábla a párok tárolási módját adja meg, itt a tömbben a keresett értéket a kulcs hash értékével fogjuk tudni kikeresni. A hash függvény legtágabb értelmezésében egy olyan függvény, mely egy tetszőleges méretű bemenetet leképez egy fix méretű értékre. A kimenet mérete csakis az implementáció szabja meg, például a Java-ban a minden osztályra definiált *hashCode()* függvény 32 bites, míg a BitTorrent protokollban is használt SHA-1 hash-ek 160 bitesek.

Egy hash tábla attól lesz elosztott, hogy tárolásában több eszköz is részt vesz úgy, hogy mindegyik az egésznek csak egy kis részét tárolja. A feladat ezután az, hogy egy kulcsról egyértelműen el tudjuk dönteni, melyik eszközön kell keresni. Ennek a megoldására alapvetően két megoldás létezik, két különböző helyzetre:

- Előre ismert elemszám esetén (*N*): Ekkor például megoldás lehet, ha megszámozom az eszközöket *0-(N-1)* között, majd a kulcs hash-értékének veszem az N modulóját, mellyel megkapom, melyik indexű eszközhöz kell fordulnom. Jellemzően szerver-kliens architektúrában található ilyen.
- Ismeretlen elemszám esetén: Erre a helyzetre több különböző megoldás létezik. Mindegyikben közös azonban, hogy a résztvevők valamilyen szabály alapján kapni fognak egy azonosítót. Az adott értéket abban a csomópontban kell eltárolni, mely egy előre definiált távolságfüggvény szerint a lehető legközelebb van a csomópont azonosítójához. Tipikusan peer-to-peer hálózatokban fordul elő, ahol a csomópontok állandó ki,-és belépése miatt szükséges az adatot redundánsan tárolni, illetve bizonyos időközönként frissíteni a helyét.

A dolgozatomhoz a második eset kapcsolódik, ezzel fogok a továbbiakban foglalkozni.



Ábra 5. Egy elosztott hash tábla modellje

4.2. Mainline DHT

A BitTorrent protokoll kiegészítése, mely tartalmazta az általuk Mainline DHT-ként nevezett elosztott hash táblát, 2005-ben adták ki [5]. A sors fintora volt, hogy tőlük teljesen függetlenül a Vuze kliens fejlesztőcsapata pár héttel megelőzve adták ki a saját, belső fejlesztésű DHT

rendszerüket, mely ezzel inkompatibilis volt. Később egy plugin segítségével oldották meg, hogy a kliensük a szabványos DHT-re is tudjon csatlakozni.

A Mainline DHT a rengeteg más peer-to-peer alkalmazás alapját adó Kademlia DHT-ra épít [13], melyben a keresési idő és a hálózat nagysága között logaritmikus összefüggés van. Ebben minden csomópont maga választja az azonosítóját, mely 160 bit méretű, és hivatalosan véletlenszerű. Ezután egy azonosító és egy kulcs hash-értékének távolságát egy egyszerű kizáró-vagy (XOR) kapcsolattal kaphatjuk meg.

Ahhoz, hogy a táblázatban keresni lehessen, fontos még, hogy a résztvevők valamilyen szempont alapján eltárolják egymás címét. Itt az MLDHT egy kicsit eltér a Kademlia protokolltól. A csomópontokat *vödrökben* (*bucket*) tároljuk. Mindegyik vödörnek meg van adva, hogy a benne tárolt azonosítóknak milyen tartományba kell esnie (ez kezdetben egy vödörnél természetesen a teljes címtartomány). Egy vödör legfeljebb 8 csomópontot tartalmazhat. Ha egy teli vödörhöz akarnánk új csomópontot adni, csak akkor válhat szét két, az eredeti tartomány felét tartalmazó vödörre, ha az eredeti tartományba esett a saját csomópontunk azonosítója. Ezzel a módszerrel azt fogjuk elérni, hogy nagyobb eséllyel fogunk olyan csomópontot tárolni, melynek azonosítója közel esik a sajátunkhoz.

Fontos, hogy a résztvevők csak „jó” csomópontokat tároljanak, melyek aktív szerepet játszanak a kérések kiszolgálásának. Ez a specifikációban a következőképp van definiálva: egy csomópont jó, ha:

- válaszolt az elmúlt 15 percben egy kérésre,
- vagy bármikor válaszolt egy kérésre, és az elmúlt 15 percben intézett egy kérést a tároló csomópont felé.

Ha egy teli vödörhöz akarunk egy új csomópontot adni, és a vödör nem válhat ketté, akkor további vizsgálatra van szükség:

- ha a vödörben csak jó csomópontok találhatóak, akkor az újonnan érkezőt egyszerűen eldobjuk,
- ha van olyan csomópont, mellyel már rég kommunikáltunk, akkor az új csomópontot felvesszük egy ideiglenes listába, majd a legrégebben kommunikált csomópontnak küldünk egy kérést. Ha nem válaszol, a helyére rakjuk az új csomópontot, különben pedig a következő inaktív csomópontnak küldünk kérést. Ezt addig folytatjuk, míg be nem kerül az új node, vagy csak jó csomópont marad a vödörben.

Annak megakadályozására, hogy a bucketek tartalma elöregedjen, a specifikáció a következő ajánlást teszi: minden vödör tartsa számon, mikor következett be az alábbi események közül valamelyik:

- egy, az adott buckethez tartozó csomópont válaszol,
- egy új csomópontot adunk a buckethez,
- egy csomópont átkerül az ideiglenes listából.

Ha bármelyik esemény régebben következett be, mint 15 perc, akkor egy, a vödör tartományába tartozó (egyébként véletlenszerűen választott) azonosítóra indítunk egy keresést.

Egy adott kulcshoz tartozó érték megtalálása vagy beszúrása ezután a következőképp fog történni: Először kiszámításra kerül a kulcs SHA-1 hash értéke. Megvizsgáljuk, hogy az általam tárolt csomópontok közül melyek vannak ehhez a legközelebb. Ezeket a csomópontoknak

küldünk egy-egy kérést, hogy küldjék vissza az általuk tárolt, a keresett kulcshoz legközelebb álló csomópontokat. Ezt a keresést addig folytatjuk, ameddig már egyik csomópont se tud közelebbi csomópontról, vagyis *iteratív keresést* végzünk. Ezzel a protokoll arra épít, hogy a hálózatban a kapcsolatok tranzitív: ha A tud kommunikálni B-vel és C-vel, akkor B is tud C-vel. Ennek a párja a rekurzív keresés, amit például a Gnutella használ.

A Mainline DHT alapvető feladata a trackerek kiváltása. Ezt a fent leírt kereséssel már könnyedén meg tudja tenni. A torrent azonosítója pont egy SHA-1 hash, így elég megkeresni az ehhez tartozó legközelebbi csomópontokat, amik így lényegében betöltik a tracker szerepét: nekik kell jeleni a letöltési szándékot, és ők fogják visszaküldeni a peereket. Fontos, hogy mivel ebben a hálózatban a csomópontok szabadon léphetnek ki és be, ezért bizonyos időközönként meg kell ismételni ezt a keresést, így garantálva, hogy mindig a legközelebbi csomópontokon legyen bejelentve a letöltési szándék.

Ahhoz, hogy egy új felhasználó részévé válhasson a hálózatnak, szükséges valahonnan egy résztvevőt megismernie. Ez lehet egy, a programkódba égetett csomópont címe (úgynevezett *bootstrap node*), de akár a BitTorrent protokollon keresztül is találhatunk ilyen csomópontot. Ha sikerült egy résztvevőt találni, rajta keresztül elindít egy keresést a saját azonosítójára, ezzel egyszerre feltölti a saját címtereit, illetve a keresés során talált csomópontok is tudomást szereznek róla. A későbbiekben ezt nem feltétlen kell megismételni, ugyanis feltéve, hogy azonos azonosítót használunk a későbbi indításkor is, nyugodtan elmenthetjük a kapcsolatainkat, amiket majd visszatöltéskor ellenőrzünk.

5. A szimuláció módja

Ebben a fejezetben mutatom be magának a szimulációs környezetet, a PeerSimet, illetve a különböző mérésekhez megvalósított osztályokat.

5.1. PeerSim

Bár a hálózat tulajdonságait valós körülmények között is tudom mérni, annak vizsgálatára, hogy bizonyos javítások milyen hatással vannak különböző méretekben, szükséges volt egy olyan környezetre, mely képes peer-to-peer rendszerek szimulációjára. Konzulensi ajánlásra, illetve egyéb alkalmazásokkal való összehasonlítás után végül a PeerSim-re esett a választásom.

A PeerSim [6] egy Java nyelven írt szimulációs motor, melynek fő irányelvei közé tartozik a skálázhatóság, modularitás és könnyű konfigurálhatóság. Ezt egészíti ki a könnyen érthető forráskód, illetve a rengeteg komment, mely megkönnyíti a rendszer átláthatóságát.

A szimulációban minden egyes résztvevőt egy *Node* azonosít, mely szabadon léphet be, illetve ki a hálózatban. Minden egyes *Node*-on *Protocol*-ok futnak, melyek befolyásolják a csomópontok közötti kommunikációt. A protokoll típusától függ, hogy minden egyes csomópontnak külön van példányosítva egy (így állapotfüggő protokollok is létrehozhatók), vagy az egész rendszerben csak egy létezik, és az fut minden *Node*-on. A protokollok mellett léteznek még a *Control*-ok. Ezek a csomópontoktól függetlenül, bizonyos időközönként futnak le, funkciójuktól függően három kategóriába lehet sorolni őket:

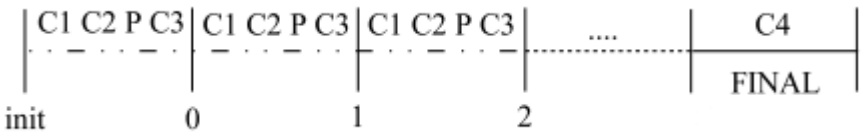
- *beavatkozó*: Befolyásolja a hálózat állapotát, például új csomópontokat ad hozzá, új kereséseket indít.
- *inicializáló*: A szimuláció indítása előtt a protokollok helyes felkonfigurálásáért felel.
- *megfigyelő*: A hálózat állapotáról gyűjt be adatokat, és azokat elmenti.

Az, hogy a szimuláció során milyen komponensek legyenek jelen a rendszerben, illetve a hálózat mérete mekkora legyen, egy konfigurációs fájlban kell megadni (erre példát a függelékben lehet látni). Ez egy könnyen olvasható szöveges fájl, mely egy egyszerű rendszert biztosít arra, hogy a hálózatot könnyedén tudjuk konfigurálni, a programkód újra fordítása nélkül. A *Java Expression Parser (JEP)* segítségével pedig akár változókat is definiálhatunk ezekben a fájlokban, melyeket a konfiguráció során több helyen fel tervezünk használni, illetve egyszerűbb matematikai kifejezések is állhatnak benne.

Fontos még a szimuláció előtt megérteni, hogyan és milyen sorrendben fognak az egyes komponensek meghívódni a szimuláció során. A PeerSim rögtön két, elvben különböző rendszert biztosít:

- Ciklus alapú: Ebben a modellben a végrehajtás ciklusokra van osztva. Az inicializálás után minden egyes ciklusban először a kontrollok, majd a *Node*-ok protokolljai fognak sorra kerülni. A szimuláció akkor ér véget, ha valamelyik kontroll leállítja, vagy a ciklusszámláló eléri a konfigurációs fájlban található értéket. Jól működik egyszerűbb szimulációkhoz, itt akár 1 millió csomópontból álló rendszert is képes kezelni a motor, kis memóriaszükséglettel.

- Esemény-vezérelt: Ebben a modellben a protokollok végrehajtását események fogják befolyásolni, míg a kontrollok a konfigurációs fájlban definiált időközönként fognak elindulni. Egy eseményt bármelyik kontroll vagy protokoll indíthat, melyben specifikálni kell, hogy melyik *Node*-nak mikor, és melyik protokollnak kell kezelnie azt. Ebben a motorban használható a *Protocol*-ból származó *Transport* interfész is, mellyel a szállítási különböző tulajdonságait tudjuk szimulálni.



Ábra 6. Ciklus alapú modell (C jelzi a kontrollokat, P a protokollt)

Ábra 7. Eseményvezérelt modell (A C kontroller 100 egységenként elindít egy protokollt, mely más protokollokat is indíthat)

Fontos, hogy egy szimuláció determinisztikus legyen, vagyis azonos paraméterek mellett azonos eredmények jöjjenek ki, így egy adott kísérlet könnyen megismételhető, illetve a bekövetkező hibák is könnyen reprodukálhatók. Ezt szerencsére a PeerSim biztosítja nekünk mindaddig, ameddig mi nem használunk nem-determinisztikus elemeket. (Maga a PeerSim egy szálon fut, ezzel ellehetetlenítve bármilyen versenyhelyzetet több szál között, illetve biztosít egy *Random* osztály példányt, melynek *magját (seed)* a konfigurációs-fájl tartalmazza. Külső kód esetén arra kell figyelni, hogy csakis a PeerSim által biztosított *Random* példányt használja, illetve ha van valami művelet, mely több szálon fut, a műveletnek hatása ne függjen az operációs rendszer ütemezésétől.)

Bár nagyon hasznos a PeerSim, sok kisegítő funkcióval együtt, megvannak a hátrányai is:

- Igaz, hogy a forráskódja tele van kommentelve, viszont nincs semmilyen dokumentáció, mely az egész rendszert ismertetné. Összesen 3 darab 10-oldalas leírás pár egyszerűbb esetről, és ezekből is csak egy foglalkozik az esemény-vezérelt modellel.
- Nem lehet különböző *Node*-okat létrehozni, így például annak tesztelése, hogy különböző verziójú protokollok milyen hatékonyan kommunikálnak egymással (ami gyakran előfordul valós rendszerekben), meglehetősen nehéz.
- A szállítási réteget nem igazán, vagy csak nehezen lehet rétegezni. Arra van lehetőség, hogy egy véletlenszerű késleltetést létrehozó réteg fölé helyezzek egy réteget, mely csomagvesztést okoz, de például azt már nem lehet megoldani, hogy két, különböző késleltetést megvalósító réteget futtasson egyszerre.
- Nem lehetséges kontrollok elindítása valamilyen esemény hatására, csakis a konfigurációs fájlban definiált paraméterek alapján fog futni, így például annak megoldása, hogy egy kontroll csak akkor avatkozzon be, ha már létrejöttek a kapcsolatok a hálózatban, csakis egy külső változó figyelésével lehetséges.

Én a szimuláció során az esemény-vezérelt használtam, mivel az közelebb van a valós hálózathoz, jobban lehet benne az idő múlását szimulálni, és így sokkal több befolyásoló tényezőt

tudtam vizsgálni, mint a ciklus alapú rendszerben. Ebben a rendszerben pedig praktikusan a milliszekundumot tekintem az idő alapegységének.

5.2. A megvalósított szimulációs modulok

5.2.1.Mainline-DHT protokoll

A megvalósítás során figyeltem, hogy a megvalósított DHT protokoll teljesen megfeleljen a specifikációban leírtaknak, a fontosabb paraméterek pedig, hasonlóan a többi modulhoz, könnyen konfigurálható legyen. Ügyeltem arra is, hogy a kód minél általánosabb legyen, így adott esetben könnyen fel lehet használni a valós hálózatban is. A későbbiek során, amikor módosítottam a kódon, mindig figyeltem arra, hogy bizonyos változókkal az eredeti állapot is futtatható legyen, így bármikor könnyedén össze tudom hasonlítani az eredeti hálózattal.

Minden egyes csomópontnak szükséges egy 160-bites azonosító. Ennek tárolására, illetve az ezzel végezhető műveletek összefogására hoztam létre az *ID* osztályt, mely egy 5 elemű int tömbben tárolja az azonosítót (5 * 32 bit = 160 bit), „big endian” módon (vagyis a tömb 0. indexe a legmagasabb helyiérték). Itt külön figyelni kell arra, hogy a Java nem ismer unsigned típusú változókat, ezért például összehasonlításkor vagy shifteléskor helytelen eredmény jöhet ki, ha nem a megfelelő műveletet végezzük el (pl. *Integer.unsignedCompare()* helyett *compareTo())* .

A DHT csomópontok (nem összetévesztendő a szimulációs motor által használt csomópontokkal (*GeneralNode*)) tárolására is szükség van egy osztályra (*DHTNode)*. Ennek tárolnia kell a csomópont azonosítóját, a hozzá tartozó *Node* referenciát (ez megfelel a valós környezetben az IP-cím és port szám kombinációjával), illetve egyéb statisztikát arról, mikor volt kérésem felé utoljára, mikor válaszolt, illetve neki mikor volt kérése felém utoljára annak érdekében, hogy eldönthessem, jó csomópont-e.

A *DHTNode*-ok tárolására szolgál a *Bucket* osztályt, melynek fő funkciója a tároláson kívül a csomópontok karbantartása: a *getQuestinableNode()* függvény adja vissza, melyik a következő csomópont, mely felé kérést kell intézni (ez normál esetben az a csomópont, amivel 15 perce nem kommunikáltunk; egy *DHTNode* törlésénél pedig véletlenszerűen átrakunk egy csomópontot az ideiglenes listából.

A *Bucket*-ek összefogására szolgáló segédosztály a *BucketStorage*, mely tartalmaz minden fontos adminisztratív jellegű munkát a vödrökkel kapcsolatban. Kikeres egy *DHTNode*-ot a vödrökből (ha már megismertük egyszer), szükség esetén hozzáadja a megfelelő *Bucket*-hez (ha eddig nem létezett), vagy ideiglenes listához. Betelt *Bucket* esetén, ha megfelel a feltételnek, felezi a vödröt, illetve a frissítési művelet is innen indítható.

A tárolásnál fontos mérték a *matchingPrefixBitCount* (nem összekeverendő a Bucket osztályban található *matchingBitCount*, mely később kerül felhasználásra), mely megmutatja, hogy a két azonosító hány darab legfelső bitje egyezik meg. Ez kiváltja a címtartományok tárolását, ugyanis, ha végig gondoljuk a specifikációban leírtakat, a vödrök osztódásakor valójában az a cél, hogy az új vödörben tárolandó címek mindenképp legalább 1 bittel közelebb lesznek a saját azonosítómhoz. A vödröket *ArrayList*-ben való tárolással így gyorsan ki lehet keresni egy adott címhez tartozó bucket-et:

```
private Bucket findBucket(ID lookingForID){
    int matchingBit = mMyID.getMatchingPrefixBitCount(lookingForID);
    int index = Math.min(matchingBit, mBuckets.size() - 1);
    return mBuckets.get(index);
}
```

A hálózatban való keresésekhez szükséges adatokat tárolja el a *SearchEntry* osztály. Ez tárolja a keresendő címet, a legközelebb lévő csomópontokat, azokat a csomópontokat, melyeknek már küldtünk keresési kérést, de még nem válaszoltak, és egy harmadik halmazban pedig azokat, akik már választ is küldtek. Itt kerül feldolgozásra a válasz is (ha érkezik), a *getNextNodeToSearch()* függvény pedig visszaadja a következő keresendő csomópont címét.

Fontos megjegyezni, hogy a csomópontok közötti kommunikációban nem a *DHTNode* referenciákat használjuk, hiszen azokban olyan statisztikák is vannak, melyeket a résztvevők maguk gyűjtenek össze. A valóság mintájára (ahol az üzenetben az azonosító és a hozzá tartozó IP-cím + portot küldjük) létrehoztam a *NodeIDAddress* osztályt, melynek egyedüli szerepe, hogy egy *ID* és egy *Node* referenciát tároljon. A kommunikáció során az üzenetekben ennek az osztálynak a példányai lesznek.

Az üzenetek kezelésére hoztam létre a *Transaction*, illetve a *TransactionManager* osztályt. Az eredeti szabványban minden üzenetnek van egy azonosító, melyet a küldő a válaszban vissza fog kapni, így be tudja azonosítani, melyik kérésre jött a válasz. Bár elméletben az azonosító + válaszcím együtt határolná be a megfelelő kérést, mivel az ajánlott 2 bájt 65565 különböző értéknek felel meg, a valóságban elég minden üzenetet különböző azonosítóval küldeni (amit egy számláló garantálni is tud), így pedig a beazonosítás is egyszerűbb lesz.

Ezeknek a komponenseknek az összefogására, illetve a protokoll kezelésére jött létre a *DHTProtocol* osztály, mely implementálja a szimulációhoz szükséges *EDProtocol* interfészt. Legfőbb függvénye a *processEvent()*, mely paraméterként kapja többek között az eseményhez tartozó általunk készített objektumot. Ez az objektum nálam mindenképp a *SimpleEvent* osztály, vagy egy leszármazottja, melynek fő funkciója az esemény beazonosítása (ezt egy int típusú *type* mezővel teszi meg). Ezután megteszem a szükséges lépéseket az esemény lekezelésére. Így a *processEvent()* függvényem struktúrája a következő lesz:

```
public void processEvent(Node node, int pid, Object event) {
    switch (((SimpleEvent)event).getType()){
        case Message.MSG_RESPONSE:
            HandleResponse((Message) event);
            break;
        case Message.MSG_PING:
            HandlePingRequest((Message) event);
            break;
        /*...*/
        case RetryEvent.TIMEOUT:
            RetryEvent timeout = (RetryEvent) event;
            Transaction transaction =
mTransactionManager.getTransaction(timeout.msgID);
            if (transaction != null) {
                HandleTimeout(transaction);
            }
            break;
        /*...*/
        case ActionEvent.START_NODE_SEARCH:
            ActionEvent search = (ActionEvent) event;
            startNodeSearch(search.getTargetID(),search.getOriginType());
            break;
        /*...*/
        case ActionEvent.CHECK_BUCKET_STATUS:
            long nextScheduleTime = mBucketStorage.checkStaleBuckets();
            EDSimulator.add(nextScheduleTime,event,node,dhtID);
            break;
    }
}
```

Itt nem soroltam fel az összes lehetőséget, csak pár példát hoztam arra, hogy milyen események vannak. Alapvetően négy kategóriába oszthatók az események:

- Üzenetek (*Message*): Ezek azok az események, amikor két *DHTProtocol* kommunikál egymással. Ebbe tartozik a specifikált 4 alapművelet (ping, findnode, getpeers, announce), illetve az ezekre küldött válasz. Tartalmaz még egy tetszőleges (*Object*) típusú mezőt is, melyben az üzenet tartalma van (ez vagy *QueryBody*, vagy *ResponseBody* osztályból származik). Megtalálható még a forrás és a cél csomópontok *Node* referenciája (hasonlóand a valóságban használt UDP-csomagok IP-cím mezőihez). Ezek az üzenetek, ha van konfigurálva, egy *Transport* (szállítási) rétegen keresztül közlekednek, mely véletlenszerű késleltetést, illetve csomagvesztést szimulál.
- Ismétlés (*RetryEvent*): Idetartoznak azok az események, amikor valamilyen műveletet újra kell próbálni. Legfontosabb a *Timeout*, mely tartalmazza annak az üzenetnek a tranzakció-azonosítóját, amelyik nem válaszolt időben, így azt az üzenetet megpróbálhatjuk újra küldeni.
- Akciók (*ActionEvent*): Ezek a bizonyos műveletek végrehajtására szolgáló események. Erre leginkább ütemezési szempontból van szükség. Természetesen meg van engedve, hogy egy külső kontroll vagy protokoll, miután megszerezte a referenciát egy adott *DHTProtocol* osztályra, bármilyen publikus függvényt meghívjon rajta, így viszont úgy fog az adott csomópont egy akciót végrehajtani, hogy nem rajta van a sor az ütemezésben. Ennek elkerülésére inkább egy ilyen eseményt küldök az adott *Node*-nak, így a protokoll maga fogja feldolgozni azt, és maga fogja indítani a szükséges műveleteket.
- Statisztikai események: Ezek olyan események, amiket kizárólag adatok gyűjtésére használok fel. Ilyen például a *PeerConnect*, melyet akkor küldünk az adott csomópontnak, ha egy peer fog rákapcsolódni (mivel nem lehet különböző

csomópont a hálózatban, ezért minimális szinten a *DHTProtocol* fogja betölteni a torrent protokollban résztvevő peerek szerepét is)

Fontos még a *DHTCommonConfig* osztály is, mely csak statikus változókat tartalmaz. Ezek befolyásolják a hálózat bizonyos paramétereit, többek között például a vödrök méretét, egy keresésnél a párhuzamos kérések számát, meddig fusson az adott keresés (megálljon, ha megtalálta a célpontot, vagy addig folytassa, ameddig nincsen meg a K db legközelebbi csomópont), illetve különböző időzítéseket. Innen az összes paraméter befolyásolható a szöveges konfigurációs fájlból.

5.2.2. Inicializátorok

Természetesen arról is gondoskodnunk kell, hogy a hálózatunk a szimuláció megkezdésekor már helyes állapotban legyen. Ezt a PeerSim inicializátorokkal teszi lehetővé. Ezek olyan *Control* objektumok, melyeket a konfigurációs fájlban más kulcsszóval hozunk létre, így még a szimuláció előtt le fognak futni. Két feladatot kell megoldaniuk:

- Minden csomóponthoz hozzá kell rendelni egy véletlenszerűen választott azonosítót.
- Valamilyen módon pár kapcsolatot minden csomópontban létre kell hozni, hogy képesek legyenek a kommunikációra.

Az azonosítók generálásához hoztam létre a *CustomIDGenerator* osztályt, mely egyrészt képes a hálózatban szereplő összes *DHTProtocol*-nak egy új azonosítót adni, illetve biztosít egy statikus *getRandomID()* függvényt. Ezt a függvényt majd más vezérlők is felhasználják, amikor szimuláció közben újabb csomópontokat illesztenek be.

A kapcsolatok kialakítására két különböző megoldást is létrehoztam, melyek különböző valós eseteket hivatott szimulálni:

- *BootstrapNodeInitializer*: A hálózat összes szereplője kezdetben csak a 0. indexű csomópontot ismeri, így az első kérést is neki fogják küldeni. Erre példa az új csomópontok becsatlakozása a hálózatba.
- *RandomConnectionNodeInitializer*: A hálózat csomópontjai véletlenszerűen ismernek egyéb csomópontokat. Itt a megvalósításban nem végzek semmilyen optimalizációt: minden egyes csomópont azonosítótól függetlenül kap N darab (konfigurációs fájlból állítható) csomópont címet, amit ezután vagy eltárol (ha befér a *Bucket* közé), vagy nem. Bár nem a legpontosabb megoldás, de a szimuláció szempontjából eléggé stabil eredményeket hozott, ezért nem fejlesztettem tovább az algoritmust. Ez azt a helyzetet modellezi, amikor egy csomópont már része volt a hálózatnak egyszer, és most újra csatlakozni akar.

Mindkét esetben a kapcsolatok létrehozása után küldök egy eseményt a hálózatban lévő összes csomópontnak, hogy indítson meg egy keresést a saját azonosítójára, ezzel még több, a saját azonosítójához közelebb eső csomópontot találva.

5.2.3. Kontrollok

A kontrollok fogják befolyásolni a hálózat állapotát szimuláció közben, például új csomópontokat adnak a hálózathoz, vagy akár új kereséseket is indítanak.

A *Turbulence* osztály egyfajta turbulenciát visz a résztvevők számába: minden egyes meghívása során egy konfigurálható szórású, természetes eloszlású darabszám csomópontot veszünk ki, vagy adunk hozzá a hálózathoz, az előjel függvényétől. Ezzel lehet vizsgálni a hálózat dinamikusságát: állandóan új csomópontok jönnek be, illetve meglévők lépnek ki, így célszerű lehet vizsgálni, hogy ez milyen hatással van például a keresési időkre egy ideális hálózathoz képest.

A *SinChurn* kontroll hasonlóan a hálózat méretét változtatja, viszont itt az idő függvényében egy szinuszos görbe a cél. Ez a *churn* jelenséget hivatott szimulálni: mérésekkel bizonyítható, hogy a valós hálózat mérete egyáltalán nem állandó (mivel például a legtöbben éjszaka kikapcsolják a gépjeiket), de viszonylag jól közelíthető egy szinuszos jellel, egy napos periódusidővel. Hasonló jelenség tapasztalható nagyobb ünnepnapokon is, ezzel a dolgozatom során nem foglalkoztam.

A *DHTNodeAdder* osztály ezekkel szemben egy teljesen átlagos lineáris méretű növekedést/csökkentést valósít meg. Ez hasznos hosszabb szimulációk futtatásánál, így ugyanis több különböző méretű hálózatot is lehet tesztelni egy futtatás során, nem kell állandóan felügyelni a szimulációt.

A *FindGenerator* osztály véletlenszerűen kiválaszt a hálózatból egy csomópontot, és elküld neki egy keresés kezdése eseményt, melyben a célazonosító egy szintén a hálózatban lévő, létező azonosító. Mivel a DHT fő funkciója a keresés, ezért elengedhetetlen a kereséssel kapcsolatos statisztikák mérése, ennek a forgalomnak a generálására pedig tökéletes ez az osztály. Bizonyos helyzetekben fontos lesz, hogy csak az olyan keresések eredményeit mérjük, amelyeket ez a kontroll hozott létre: ennek érdekében a SearchEntry osztály tartalmaz egy *originType* nevű mezőt, melyben a keresés eredetének azonosítóját tárolhatjuk. Ez általában 0 (pl. ha egy vödör frissítése miatt indítunk), a *FindGenerator* által indított kereséseknek viszont van egy konkrét definiált értéke.

A *DDOSTester* kontroll a hálózat egyik biztonsági rését kihasználva indít támadást egy ártatlan fél számára. Ennek a működéséről, illetve ezzel a témával később részletesen is fogok foglalkozni.

5.2.4.Megfigyelők

A megfigyelők szerepe, hogy számon tartsák a hálózat állapotát, statisztikákat gyűjtsenek és összegezzenek, és ezeket valamilyen úton elérhetővé tegyék a külvilág számára. Bár én egy külön kategóriába sorolom őket a kontrollerektől, valójában az ide tartozó osztályok is a *Control* interfészt implementálják, vagyis ugyanúgy ütemeződnek, hajtódnak végre, és ugyanúgy lehet őket konfigurálni.

A *SearchObserver* a *FindGenerator* által indított kereséseket felügyeli. Ezt sok, statikusan elérhető *IncrementalStats* típusú változóval valósítja meg, az egyes csomópontok ezeken a változókon keresztül jelzik a keresés végén a mért adatokat (pl. meddig tartott a keresés, hány csomópontot sikerült elérni, stb.). Az *IncrementalStats* a PeerSim-be épített statisztikai elem, melynek tetszőleges értékeket beadva, számon tartja az értékhalmaz minimumát, maximumát, átlagát, négyzetösszegét, illetve ezekből tud varianciát és standardszórást számolni. Fontos, hogy minden egyes futás végén törli a számlálókat, vagyis mindig az éppen aktuális ciklusban összegyűjtött adatok vannak összegezve. Ezzel a szimuláció elején szinte elkerülhetetlen előkerülő kiugrások, szélsőséges értékek kiszűrése miatt van szükség.

A *PeerConnectObserver*, illetve a *NodeNotFoundObserver* valójában majd a DDOS tesztelése során fog használatra kerülni. A nevükből is adódóan az első megfigyelő arról gyűjt adatot, ha egy csomópont, mint peer csatlakozik egy másik csomóponthoz, míg a második azt szolgálja, hogy az ismeretlen/nem létező csomópontnak küldött üzenetek számát tartja nyilván.

5.2.5.Szállítási elemek

Az IPv4-es internet protokollban a hálózatban résztvevő eszközök egy 32 bites (4 bájt) hosszúságú címet kaptak. Bár sokáig azt hitték, hogy az ezzel elérhető körülbelül 4.3 milliárd cím bőven elég lesz minden egyes eszköznek, az internet rohamos terjedésével, illetve a technológia rohamos fejlődésével bebizonyosodott, hogy ez nem így van. Annak érdekében, hogy továbbra is használhassuk az IPv4-es címzést, fejlesztették ki a routerek *Network Address Translation (NAT)* funkcióját. Bár a dolgozatomban nem ezzel részletesen nem tervezek foglalkozni, mégis szorosan kapcsolódik a témámhoz, ezért adok egy rövid jellemzést róla.

A NAT lényegében szétválasztja az internetet és a routerre csatlakozó eszközöket, ezzel egy belső (privát) hálózatot létrehozva. A belső hálózat eszközei kapnak egy, a router által kijelölt IP címet, a router maga pedig rendelkezik egy publikus IP címmel az internet felé. Bármikor, amikor a belső hálózatról egy kérést küldünk az internet felé, a router feljegyzi a kérést intéző számítógép címét, majd az üzenet forrás-címét módosítja a saját, publikus IP címére, ezzel elérve, hogy a belső hálózat összes eszköze egy publikus cím alatt látszódjanak. Ha megérkezik a válasz, akkor a router a belső feljegyzései, illetve a csomag címéből dönti el, a belső hálózat melyik eszközének kell továbbítani a bejegyzést. (Bár a portok szerepéről itt nem írtam, de természetesen azok is szükségesek a sikeres NAT folyamathoz)

Ebből is látható, hogy bár a belső hálózat probléma nélkül tud kapcsolatot létesíteni a külső világgal, ez fordítva már nem igaz. Ahhoz, hogy a belső hálózatba eljusson a megfelelő csomag, léteznie kell egy feljegyzésnek, ehhez viszont a belső hálózatról kell kezdeményezni a küldést. Bár vannak erre megoldások (a legjobb erre a helyzetre az UPnP, amivel lényegében az alkalmazás tudja jelezni a routernek, hogy milyen csomagokat továbbítson felé), ezeket sajnos nem támogatja minden eszköz.

Emellett még egy fontos tényező lesz a tűzfalak használata. A mai világban nagyon sok eszközön, illetve routeren fut valamilyen típusú tűzfal, mely a kéretlen adatcsomagokat szűri ki (pl. Windows rendszereken alap beállításként fut). Így előfordulhat az, hogy A csomópont tud kéréseket küldeni B csomópontnak, és a válaszait is tudja fogadni, viszont B csomópont kérései már nem érnek el A-ba.

Egy fontos kérdés még megmaradt: a bejegyzések meddig legyenek érvényesek? Bizonyos esetekben ezt könnyű eldönteni, például ha az üzenet a TCP szállítási réteget használja (mint például az eredeti BitTorrent protokoll), az ugyanis eredően egy kapcsolatot fog kialakítani, melynek életciklusát az üzenetek mezőiből könnyű követni. A mi esetünkben viszont UDP csomagokat használunk, mely teljesen státusz mentes, ezért itt egyszerűen idő alapú bejegyzéseket használnak. Sajnos az nincs egységesítve, hogy egy bejegyzés mennyi ideig éljen, teljesen a gyártókra van bízva, de általában ez 30 másodperc és 5 perc között van.

A NAT és a tűzfal modellezésére alkottam meg a *NATTransport* osztályt. Ez a *Transport* interfészt implementálja, így alkalmas lesz üzenetek küldésére a csomópontok között.

Az osztály első feladata eldönteni, hogy az adott csomópont, ahonnan származik a kérés, milyen tulajdonsággal rendelkezzen. Három különböző lehetőséget veszek figyelembe:

- Teljesen nyílt, nem használ se tűzfalat, se NAT-ot.
- A csomópont egy NAT-réteg mögött van, de nem használ tűzfalat.
- A csomópont tűzfalat is használ.

Ezeknek az arányait a konfigurációs fájlból lehet beállítani.

A következő lépés, ha a küldő csomópont használ NAT-ot vagy tűzfalat, hogy feljegyezze az üzenet küldésének időpontját. Sajnos a PeerSim adottságai miatt a következő lépésnél kicsit ellen kell mondanunk a valóságnak: megvizsgáljuk, hogy a fogadó fél tudja-e fogadni az üzenetemet, vagyis létezik a fogadó félen egy bejegyzés, és az általa feljegyzett idő nem volt-e túl régen (ez sima NAT esetében perces nagyságrend, míg tűzfal használatánál másodperces). Ez azért problémás, mert így a hálózat által létrehozott késleltetés előtt fogjuk megnézni, hogy a fogadóoldal tudja-e fogadni az üzenetünket, de sajnos ezt nem lehet könnyen megoldani.

6. Mérések

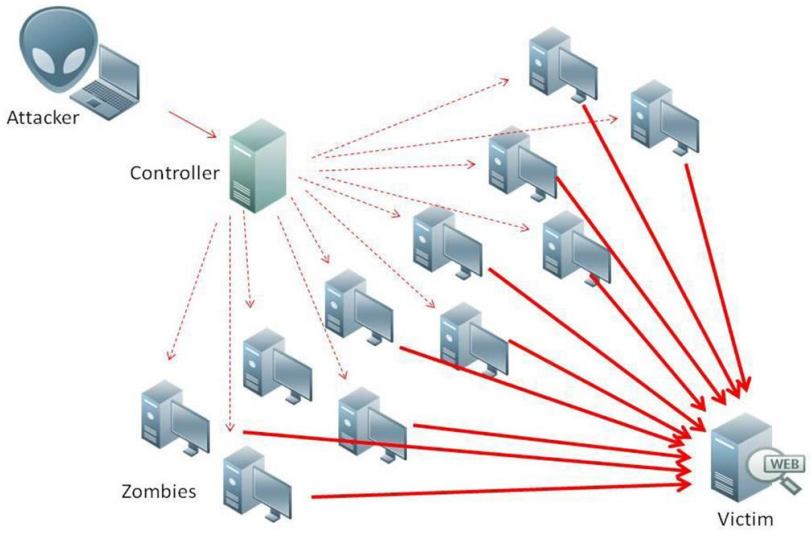
Az alábbi fejezetben ismertetem azokat a témaköröket, amelyeket érdemesnek láttam a fent megvalósított szimulációs környezetben vizsgálni. A szükséges alapismeretek ismertetése után pedig bemutatom a mérési eredményeimet, és értékelem azokat.

6.1. DDoS lehetőségek

A *Denial of Service* (magyarul szolgáltatásmegtagadással járó támadás, vagy túlterheléses támadás) egy, a mai napokban is gyakran használt kibernetikai támadás, melyben a cél egy informatikai szolgáltatás megbénítása, használhatatlanná tevése. Ennek egy változata a *Distributed Denial of Service* (elosztott túlterheléses támadás), melyben a támadás egyszerre több, egymástól elkülönült eszköz végzi.

A DDoS azt a tényt használja ki, hogy a támadásra kijelölt szerver erőforrásai korlátosok, így ha azt elárasztjuk rengeteg kéréssel, akkor a többi felhasználóktól származó kéréseket is csak lassan, vagy rosszabb esetben sehogy se fogja tudni kiszolgálni. Népszerűsége abból eredhet, hogy bár egy ilyen támadás végrehajtásához csak az alapszintű informatikai tudás szükséges, mégis látványos eredményeket tud felmutatni. Híresebb esetek közé tartoznak például a *Lizard Squad* támadásai, mellyel olyan óriási cégek szervereit sikerült elérhetetlenné tenni, mint például a Sony, vagy az Electronic Arts. Egy ilyen támadásnak viszont lehetnek közvetett hatásai is: 2016. 10. 21.-án ismeretlen tettesek egy hatalmas támadást indítottak a Dyn nevezetű DNS szolgáltató ellen. Miután a szervereik nem bírták a terhelést, rengeteg nagyobb weboldal vált elérhetetlenné (pl. Github, CNN, Paypal), mivel DNS feloldás nélkül a böngészők nem tudták elkérni az adott URL-hez tartozó IP címet.

Az egyik fő eleme a DDoS támadásoknak a rengeteg eszköz, melyről az internet felé kéréseket lehet küldeni. Egy nagyobb támadásnál ez a szám már olyan nagyságrendben mozog, amit egy ember maga nem tud biztosítani. Erre a célra jöttek létre a *botnet*ek, illetve a hozzájuk kapcsolódó vírusok, mellyel észrevétlenül tudnak terjedni az interneten ártatlan felhasználók eszközeire, hogy később pár egyszerű utasítással koordinálni lehessen őket.



Ábra 8. DDoS támadás [14]

Felmerülhet a kérdés, hogy a P2P protokollokat nem lehet-e ilyen célokra felhasználni. Ha megvizsgáljuk például a BitTorrent protokollt, észre lehet venni, hogy bizony számos alkalom adódik arra, hogy befolyásoljuk a peerek eszközeit a kérések küldésében:

- A letöltött .torrent fájl többek között trackereket is tartalmaz, ahonnan a fájlcserében résztvevők listáját tudjuk elkérni. Ha módosítjuk a torrent fájlban a tracker IP címét, akkor a kliens arra a címre fog kérést küldeni. Ha ezt a fájlt sokan indítják el, minden egyes kliens kérést fog indítani az ártatlan szerver felé.
- Az adott tracker tulajdonosa is indíthat egy támadást azzal, hogy a peerek helyett a kiszemelt szerver címét küldi el, így a kliens megpróbál kapcsolatba lépni vele.
- Egy, a BitTorrent protokoll kiegészítését célzó *Peer Exchange* (PEX) protokoll segítségével akár egy kártékony peer megadhatja a célpont IP címét, azzal a szándékkal, hogy a másik fél megpróbálja felvenni vele a kapcsolatot.

Bár a fenti esetek közül bármelyik előfordulhat, a valóságban azonban nem igazán történnek ilyenek. Rossz .torrent fájl esetén a felhasználó általában hamar letölt helyette egy másikat, a kártékony trackerek általában hamar kihalnak, és a fájlmegosztáshoz elég szokott lenni 25-30 peer, így ha van ennyi stabil peer, a kliensek nem fognak újat keresni.

Ez persze nem jelenti azt, hogy sose történt volna ilyen. Egy különleges eset volt, amikor rengeteg kisebb webszerver omlott össze, mivel nem bírta kiszolgálni a rengeteg TCP szinkronizációs kérést, ráadásul az összes kérés kínai IP címről jött. Mint kiderült, a *Nagy Kínai Tűzfal* (*Great Firewall of China*) hibás beállítása miatt az összes thepiratebay.org trackerre irányuló kérés a DNS feloldás hatására módosított IP-címet adott vissza, így amikor a kínai felhasználók elindították a torrent kliensüket, a rengeteg betöltött torrent fájl automatikusan elkezdett kapcsolatokat létesíteni különböző szerverekkel szerte a világon, amik nem bírták az óriási terhelést [15].

Érdemes tehát megvizsgálni, a DHT esetén milyen lehetőségeink vannak arra, hogy egy kiszemelt IP címet leterheljünk kérésekkel. Két esetben adunk át egy másik személynek címet:

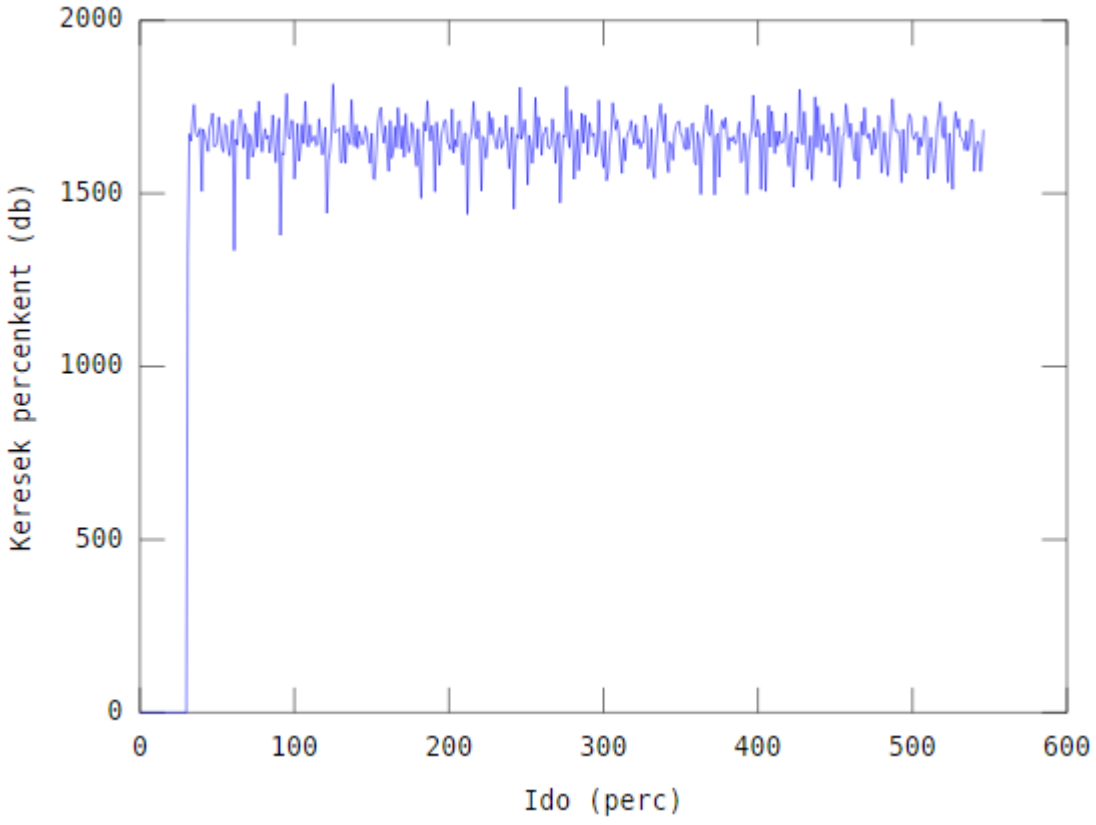
- *FindNode* kéréssel fordul hozzánk, ekkor az általunk ismert, a keresett azonosítóhoz legközelebbi csomópont azonosítóját és címét küldjük vissza. (Ezzel UDP kéréseket tudunk a célpont felé generálni)
- *GetPeers* kérésnél az adott azonosítóhoz tárolt peerek címét küldjük vissza. (Ez alapesetben TCP kapcsolatot fog létrehozni)

Könnyen belátható, hogy egyik esetben se akadályoz meg semmi, hogy az áldozat IP címét küldjem el, bármikor egy kérést intéznek felém. Bár már ez a tény is baj, ez ellen sajnos nem igazán tudunk tenni semmit: egy üzenetváltás során nem tudunk semmilyen módon meggyőződni az adatok valódiságáról.

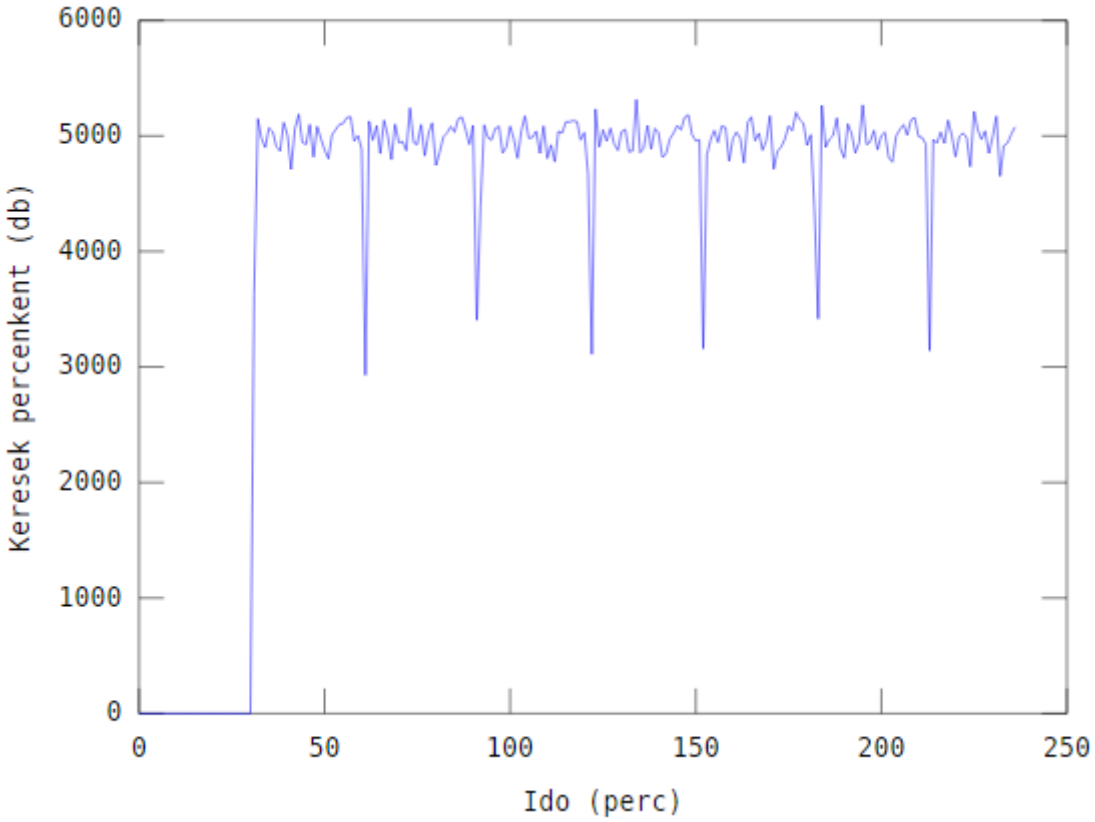
Viszont ez még alapjaiba nem jelentene óriási veszélyt, hiszen ahhoz, hogy akkora forgalmat generáljunk, ami be tud lassítani egy szervert, szükséges, hogy egyszerre nagy mennyiségű csomópont forduljon hozzánk a fent felsorolt kérésekkel. Ennek az esélye normális körülmények között elenyésző. A probléma ott keletkezik, hogy a csomópontok az azonosítójukat maguk választhatják meg, ezzel akár forgalmasabb helyekre is be tudják magukat illeszteni. Elég figyelni a híresebb torrent oldalakat, és a legnagyobb töltésszámmal rendelkező torrent azonosítójához közeli címet választani, így garantálni tudom a szükséges forgalom mennyiségét.

Bár könnyű kiszámolni, mivel fog járni különböző esetekben egy ilyen támadás, a biztonság kedvéért a szimulátorban is le akartam ellenőrizni. Ennek érdekében hoztam létre a *DDOSTester* kontrollt, melynek működése a fent leírt működést valósítja meg. Bizonyos idő után (amit a konfigurációs fájlból adhatunk meg) generálunk egy véletlenszerű azonosítót, mely megfelel a híresebb torrent azonosítójának. A torrent segítségével generálunk 8 csomópontot, melynek azonosítói közelebb helyezkednek el hozzá, mint bármelyik másik résztvevő a hálózatban. Ezt a 8 csomópontot felkonfiguráljuk a támadás típusától függően (TCP esetén peer-ként hozzáadjuk a 0. indexű csomópontot, és rajta keresztül mérjük az új kapcsolatokat; UDP esetén egy null *Node*-ot adunk hozzá a bucketekhez, és a *NodeNotFoundObserver* gyűjti a csatlakozások számát). Miután a hálózatban elhelyeztük a káros csomópontokat, kiadjuk az összes többinek a keresési utasítást, ahol a célazonosító a torrenttel egyezik meg.

A szimuláció során feltételezem, hogy az egyes csomópontok 15 percenként indítanak új kereséseket az adott torrent iránt. Egy nagyobb torrent akár 50 ezer letöltővel is rendelkezhet, így a szimulációt is ekkora mérettel futtatom le.



Ábra 9. TCP kérések száma percenként



Ábra 10. UDP kérések száma percenként

Látható, hogy mindkét esetben óriási mennyiségű kapcsolat jön létre. TCP esetén minden egyes bejövő kérésre előbb létre kell hozni a kapcsolatot, melynek így egy háromfázisú kézfogáson kell átesnie, mielőtt a szerver beleolvashatna a kérésbe, és eldobhatná azt.

Az UDP kapcsolatnál ez a veszély nem áll fent, itt viszont a sávszélesség lefoglalása okozhat gondot. Bár valószínűleg egy nagyobb vállalat szervere gond nélkül ki tudja szűrni az UDP csomagokat, egy kisebb vállalat, vagy akár az otthoni routerek nem fogják bírni, és ezzel az összes oda irányuló kapcsolat be fog lassulni, illetve megnő a csomagvesztési arány is. Ilyen módon már hatásosan támadtak meg az interneten híresebb személyiségeket (pl. *streamer*-eket).

Szerencsére erre a problémára már terjesztettek elő egy megoldást. Bár lehetne próbálkozni azzal, hogy valahogyan más csomópontokkal igazoljuk, hogy a küldő csomópont mennyire „hihető”, ez azonban rengeteg egyéb támadást engedne meg a hálózaton, illetve magát az implementációt is bonyolítaná. Helyette azt korlátozzák be, hogy egy csomópont szabadon megválaszthassa az azonosítóját.

Ezt igazából csak egy adattal lehetséges lekorlátozni: a külső, publikus IP címhez kell kötnie a csomópontnak az azonosítóját. Bár maga az elv egyszerű, a megvalósításban több problémát meg kell oldani:

- Az IPv4 címünk 32 bites, míg a csomópont azonosító 160 bit hosszúságú. Maga a hálózat hatékonysága azon múlik, hogy az azonosítók elméletben bármilyen értéket felvehetnek, az viszont egyértelműen látszódik, hogy a 32 bites címtérhez matematikailag is képtelenség hozzárendelni egy 160 bites címteret.
- A NAT miatt több csomópont is futhat egy publikus IP cím mögött, márpedig célszerű lenne, ha hozzájuk különböző azonosító tartozna.

Végül a következő formulát alkották meg [16]:

```
hash = crc32c((ip & 0x030f3fff) | (r << 29))
```

Ahol az r szám egy tetszőlegesen választott szám alsó 3 bitje. Az csomópont ezután úgy választja meg az azonosítóját, hogy az első 21 bit megegyezzen a generált hash első 21 bitjével, illetve az utolsó bájt a tetszőlegesen kiválasztott szám legyen.

Ahhoz viszont, hogy ezt az értéket le tudjuk generálni, szükséges, hogy a belépő csomópont megismerje a publikus IP címét. Mivel ezt csakis az interneten lévő felhasználók ismerik meg, ezért egy trükköt kell alkalmazni. Ha egy csomópont a kérésnél úgy érzékeli, hogy az azonosító nem teljesíti a szükséges feltételeket, egy hibaüzenetet küld vissza, melyben többek között maga az IP cím is benne lesz. Ennek a hibaüzenetnek a tudatában a csomópont képes lesz egy érvényes azonosítót generálni.

6.2. Keresési idő mérése, a NAT hatása

A következő méréseim a keresési időkre irányultak. Ennek vizsgálatára több különböző mód lehetséges, mely különböző helyzeteknek felelnek meg. Ezeknek jelentését fogom a következőkben részletezni.

Bár a felhasználó végeredményben a keresési időt fogja tapasztalni a program használata során, nem feltétlen ez jelzi legjobban a keresés hatékonyságát. Az én szimulációimban nem foglalkoztam részletesebben a hálózat késleltetési tulajdonságaival, mivel maga a protokoll se veszi figyelembe ezeket. A valóságban előfordulhat olyan helyzet, hogy a lassú válaszidő miatt a keresés is hosszabb ideig tart, emiatt szerintem jobb mérőszám a szükséges üzenetváltások száma, mely a hálózati tulajdonságoktól teljesen független lesz.

A keresési idő természetesen attól is függ, hogy meddig folytatjuk a keresést. Az egyik mérési módnál mindaddig folytatjuk a keresést, ameddig nem találtuk meg a legközelebb lévő K csomópontot (alapesetben 8), melyek válaszolnak is a kéréseinkre, ez megfelel az átlagos használathoz. A másik lehetőség, hogy addig folytatódik a keresés, ameddig meg nem találjuk a keresett azonosítót, vagy nem találunk már közelebb lévő azonosítót. Ez jobb képet adhat magáról a hálózat hatékonyságáról, a kapcsolatok szerveződéséről, hiszen így nem kell megvárni az összes csomópont válaszát.

Mielőtt részletesebb vizsgálatokat készítenék a hálózatról, megmérem a szimulátorban tapasztalható keresési időt, hogy összehasonlíthassam a valós környezetben mérhető válaszidőkkel.

A szimulációhoz az alábbi beállításokat választottam (a specifikációban leírtak):

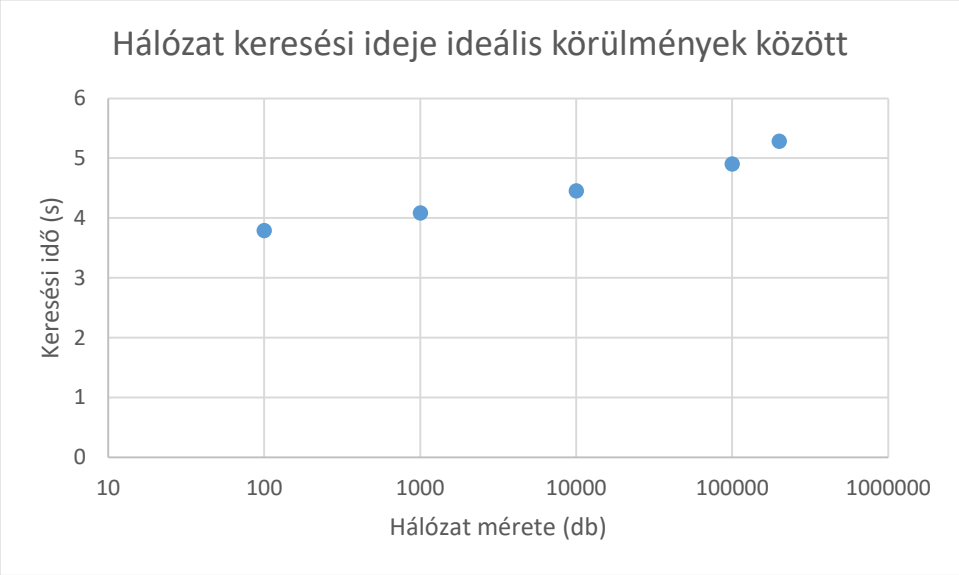
- a vödrökben tárolt csomópontok darabszáma 8,
- a vödrök frissítési ideje 15 perc,
- egy csomópont jó, ha 15 perce volt kérése, vagy volt kérése és válaszolt 15 perce
- a keresés során a legjobb 8 csomópontot keressük,
- egy kereséshez legfeljebb 4 kimenő kérés tartozhat egyszerre,
- a hálózat késleltetése egyenletes eloszlású, 30ms és 400ms között,
- egy üzenet két csomópont között véletlenszerűen elveszhet, melynek esélye 0.1%,
- a hálózat indulásakor véletlenszerűen alakítunk ki kapcsolatokat,

- bizonyos időtartalmanként (kisebb hálózatoknál 1ms, nagyobbaknál 10ms, hogy elférjen a memóriába) generálunk egy keresést egy véletlenszerűen választott csomópont egy másik csomópont azonosítója között,
- egy kérés válaszára 5 másodpercig várakozunk,
- egy csomópont elérhetetlennek minősül, ha egymás után 3-szor nem válaszol a kérésünkre

Bár itt még nem veszünk figyelembe más hatásokat (például csomópontok ki,-belépését), a keresési paraméterek megegyeznek a specifikációnak, tehát egy nagyságrendi mérésre jónak kell, hogy legyen. Összehasonlításképp a valós környezetben körülbelül 1-1.5 percig tart egy keresés:

Táblázat 1. Ideális hálózatban való keresési idő a hálózat méretének függvényében

n(db)	t(s)
100	3.79
1000	4.08
10000	4.45
100000	4.9
200000	5.28



Ábra 11. A hálózatban való keresési idő grafikonja

Látható, hogy a szimulátorban mért idők egy-két nagyságrenddel kisebbek a valós környezethez képest, mely egyáltalán nem nevezhető normálisnak. Miután ellenőriztem a szimuláció helyességét, a valós környezetben mért eredményeket kezdtem el vizsgálni. Itt egyből feltűnt a probléma: a legtöbb kérésünkre nem érkezett semmilyen válasz. Kicsit tovább vizsgálva a helyzetet arra jutottam, hogy a legtöbb csomópont a hálózatban számomra nem elérhető (körülbelül a megismert csomópontok 55% nem válaszolt).

Ennek irányába elkezdtem kutakodni, és így bukkantam rá a már ismertetett *Kapcsolódási tulajdonságai a Mainline BitTorrent DHT csomópontoknak*. Bár ezt a papírt 2009-ben írták, a szerzők által tapasztalt 36%-os elérhető csomópont-arány körülbelül megegyezik az általam mért 45%-al, és a dolgozatukban tett megállapítások véleményem szerint ma is megállják a helyüket.

A méréseiket a következő módon végezték: miután elindítottak és sikeresen elhelyeztek pár darab csomópontot a hálózatban, elkezdték vizsgálni a feléjük intézett kéréseket. Ha egy eddig nem tesztelt csomópont indította, feljegyezték egy listára. Egy csomópont tesztelése abból állt, hogy különböző módokon kéréseket intéztek felé, majd vizsgálták, érkezik-e rá válasz:

- ugyanarról az IP-címről, illetve port-számról indították a kérést, ahonnan érzékeltél a csomópontot,
- ugyanarról az IP-címről, de más port-számról indították a kérést,
- egy másik IP-címről indították a kérést

Ezt a mérést kétszer végezték el: közvetlenül a csomópont megismerésekor, illetve öt perccel később. Az így mért eredmények, táblázatba foglalva:

Pattern	Nodes (%)	Possible cause(s)
UUU–UUU	10.6	Firewall
RUU–UUU	**31.3**	Port restricted cone
RUU–RUU	2.8	and symmetric NAT
RRU–UUU	0.8	Restricted cone NAT
RRU–RRU	2.0	
RRR–UUU	2.7	Full cone NAT and real churn
RRR–RRR	**35.5**	Open Internet
UUU–RRR	7.6	Behavior not matched
RRU–RRR	1.7	
Other	5	Rest of the cases

Ábra 12. A hálózatban szereplő csomópontok kapcsolati tulajdonságai

Az egyik fontos megállapítása a kísérletnek, hogy az összes csomópont mindössze 35.5%-a érhető el bárhonnan. Ez, bár nem a legegészségesebb a hálózat szempontjából (hiszen így 10 csomópont terhelését 3,5-nek kell teljesíteni), még nem befolyásolná a keresési időket, mivel a protokollban csak jó csomópontokat oszthatunk meg másokkal (olyanokat, akik képesek válaszolni). A probléma a második, illetve a harmadik főcsoporttal van: ezek olyan csomópontok, melyek bár képesek válaszolni annak a kéréseire, akivel már korábban felvette a kapcsolatot, de egy külső címről már nem lehet kapcsolatot létesíteni vele. Ez megtöri a hálózat tranzitivitási tulajdonságát, mely viszont szükséges az iteratív keresés helyes működéséhez. (Itt érdemes megjegyezni, hogy rekurzív keresés esetén ilyen probléma nem fordulna elő, hisz ott a csomópontok a kérés beérkezésére maguk folytatják azt.)

Ennek tudatában írtam meg a fentebb részletezett *NATTransport* osztályt, mely az első, illetve a második főcsoportba tartozó elemeket hívatott szimulálni (melyek az egész hálózat 44,7%-át adja).

Ezt figyelembe véve megismétlem a mérésemet a fent leírt paraméterekkel, illetve a következőkkel kiegészítve:

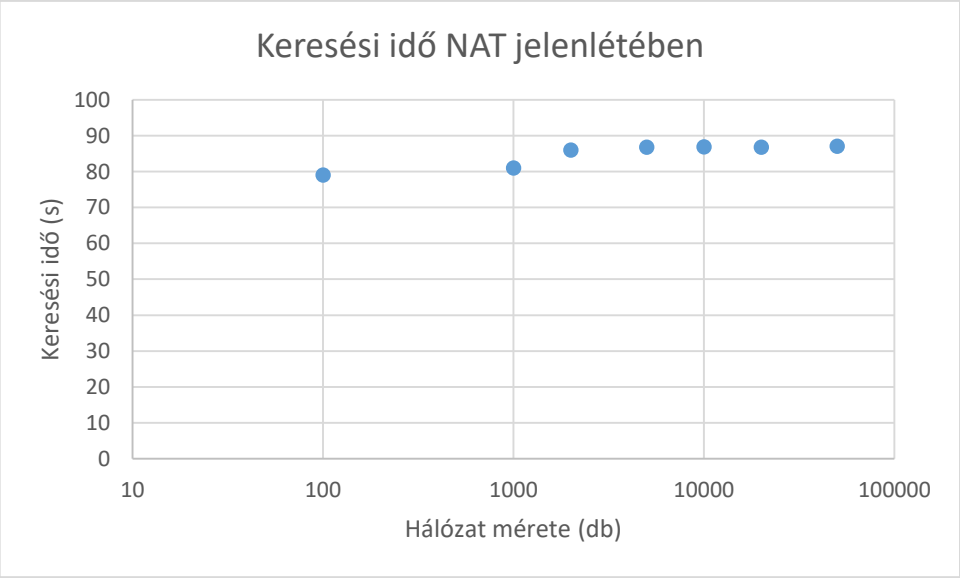
- a NAT bejegyzések életideje 2 perc,
- a hálózat 50% használ NAT-ot, 10% agresszív tűzfalat, a maradék 40% a pedig szabadon elérhető

- a 0. indexű csomópont garantáltan elérhető legyen (ez azért szükséges, mert így később az új résztvevők nem egy elérhetetlen csomóponton keresztül próbálnak becsatlakozni.

Így a következő értékeket mértem:

Táblázat 2. Keresési idő NAT jelenlétében különböző méretű hálózatoknál

n(db)	t(s)
100	79
1000	81
2000	86
5000	86.8
10000	86.9
20000	86.83
50000	87.06



Ábra 13. NAT jelenlétében mért keresési idő eredményének grafikonja

Látható, hogy ezek az idők már annyira nem is függenek a hálózat méretétől, megegyeznek a valós hálózatban mért értékekkel. Ez azzal indokolható, hogy háromszor próbálunk meg üzenetet küldeni elérhetetlen csomópontoknak, mely így 15 másodpercig be van ragadva, ezek az idők pedig két nagyságrenddel nagyobbak az átlagos üzenetküldési időnek.

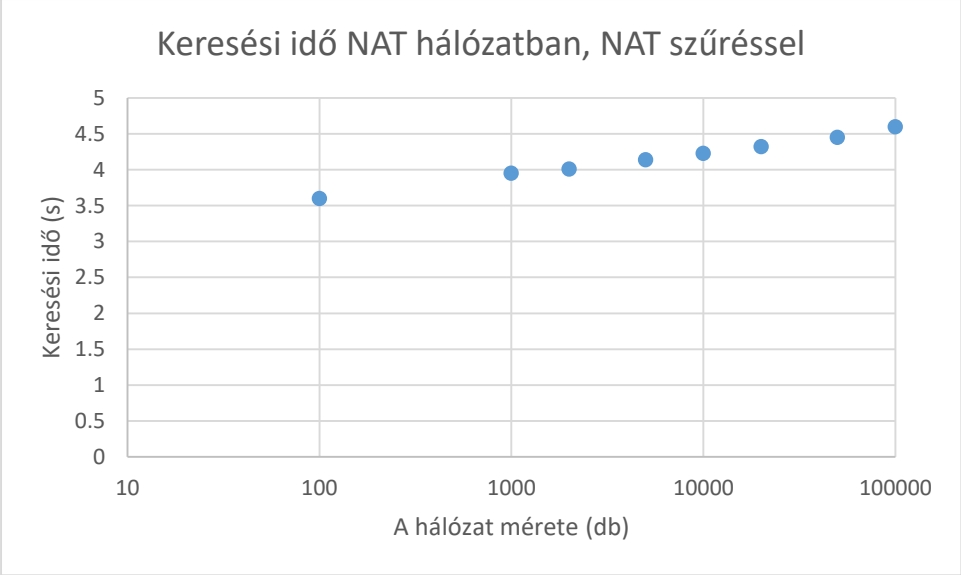
Kérdés, hogy tudunk-e valamit tenni a NAT mögött lévő csomópontok kiszűrésére. Elnézve a korábban bemutatott táblázatot a kapcsolódási tulajdonságokról, feltűnhet, hogy a legnagyobb csoport, mely káros a hálózatra nézve, a második főcsoport: ide tartoznak a csomópontok 34.1%-a, több mint a hálózat harmada). Erre a csoportra jellemző az, hogy bár azonos IP-cím, illetve port-számról küldött kérésre képesek válaszolni, de egy másik port-számról küldöttre már nem. Ezt figyelembe véve módosítanám a „jó” csomópont definícióját: kiegészíteném azzal, hogy bármikor képes volt egy másik portról indított kérésre válaszolni.

Emellé a hálózat minél gyorsabb stabilizálódása érdekében még egy beállítást hozzáadtam a protokollhoz: egy új csomópont felismerésekor, ha el tudjuk tárolni, azonnal indítsunk egy kérést felé, így hamar kiderül, hogy jónak minősíthető-e.

Ezekkel a tulajdonságokkal újra lefuttattam a szimulációt, és a következő értékeket mértem:

Táblázat 3. Keresési idő NAT hálózatban, a csomópontok elérhetőségének vizsgálatával

n(db)	t(s)
100	3.6
1000	3.95
2000	4.01
5000	4.14
10000	4.23
20000	4.32
50000	4.45
100000	4.6



Ábra 14. A szűrt NAT hálózatban mért keresési idők grafikonja

Látható, hogy ezzel a szűréssel jelentősen felgyorsul a keresés ideje, már az ideális hálózat keresési idejét is megelőzi. Ez azért lehetséges, mert a 40%-os elérhetőségi arány miatt nem az összes csomópont vesz részt, ezért a valódi hálózat mérete is kisebb a látszólagosnál.

Természetesen a szimuláció sose tükrözi pontosan a valós hálózatot, mivel a többi csoporttal nem is foglalkoztam a méréseim során. Viszont a vizsgált három főcsoport (teljesen elérhető : 35.5%, tűzfalat használók: 10.6%, teljes NAT használók: 34.1%) együtt már egy jelentős arányt lefed a teljes hálózatból, ezért ha nem is hozna akkora javulást, mint mértem, valószínűleg így is drasztikus hatásai lennének, ha ezt a plusz vizsgálatot bevezetnénk a csomópontokra.

Bár az eredeti specifikációban nem szerepelt, egy későbbi kiegészítésben bevezettek egy változót az üzenetekben, mellyel jelezni lehetett, hogy csak olvasni akarjuk az értékeket, ne adjon hozzá a válaszoló csomópont a saját táblájához, részben a NAT mögött futó alkalmazások miatt. Ezzel az a gond, hogy minden csomópontnak magának kéne eldöntenie, hogy alkalmas-e kívülről

kapcsolatok fogadására, ezt pedig egyedül lehetetlenség. Az én megoldásom azzal biztosít többet, hogy így a kérést kiszolgáló oldalon tudjuk viszonylag jó eredménnyel eldönteni, hogy képes-e a csomópont kapcsolatok fogadására.

6.3. Perfect Storing hatása a keresési időre, szórásra

A méréseim során felfigyeltem még egy potenciális lehetőségre a sebesség javítása érdekében. A csomópontok a vödrökben teljesen véletlenszerűen tárolják az általuk ismert szomszédokat. Esetleg ha lenne valami rendszer a tárolásban, nem érhetnénk-e el jobb keresési időket?

Ennek a mérésére itt megváltoztattam a keresés módját: egy keresés csakis addig folytatódik, ameddig meg nem találom a keresett csomópontot. Erre azért van szükség, mert mint az előző kísérleteimből is tapasztaltam, a kiküldött kérések nagy része a talált csomópontok ellenőrzésére volt szükség. A javulás mértékének mérésére felhasználtam még pár mérőszámot, melyet itt ismertetek.

A hálózat egyik legfőbb tulajdonsága, melyet más tanulmányokban is használnak, a *teljes bit prefix* (*complete bit prefix*). Ez azt mutatja meg, hogy a teljes, 160 bites címtérből hány kezdő bit van, melynek az összes kombinációja megtalálható a hálózatban. Egy adott hálózatméret mellett arra kell törekedni, hogy ez minél inkább közelítse a $\log_2 N$ értéket (ahol N a csomópontok száma), ugyanis így egyenletesen fog eloszlani a terhelés a csomópontok között. Pontosan ebből kifolyólag is van előírva, hogy az azonosítót véletlenszerűen kell választani.

Egy ezzel szorosan összefüggő mérőszám a *bitjavulás* száma. Ez azt adja meg, hogy egy keresés során egy sikeres üzenetváltás hatására hány bittel kerülünk közelebb a keresett azonosítóhoz. A mérés során a keresett csomópont azonosítója által hozott javulást nem számítjuk, hiszen az nagyságrendekben befolyásolja az eredményt (mivel ebben az esetben 160 bit lesz az egyezés, így sokszor 146-150 bitjavulást is jelenthet).

Egy kis gondolkodással belátható, hogy az előző fejezetben ismertetett sikeres üzenetváltások száma (hány lépést végzünk), illetve a bitjavulás szorzata (lépésenkénti javulás) körülbelül a teljes bit prefixet fogja becsülni (ez pont az utolsó lépés kihagyása miatt nem fog pontosan egyezni).

Először állapítsuk meg, hogy a mostani tárolási formával mennyi a *várható bitjavulás* értéke. Feltételezem, hogy az adott vödör, amiben keressük az új értéket, tele van véletlenszerűen választott azonosítókkal. Az így levezetett közelítő képlet, magyarázat nélkül:

$$E_b = \sum_{k=0}^{\infty} 1 - \left(\frac{2^k - 1}{2^k}\right)^n$$

Ahol *n* jelöli az egy vödörben tárolt értékek számát. Vizsgáljuk meg, hogy ez különböző vödör-méretekre mekkora értéket ad (ennek számolására a WolframAlpha nevű weboldalt használtam):

n(db)	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**
E_b**(bit)**	2	2.666	3.505	4.421	5.377	6.355	7.344	8.338	9.336

Ábra 15. Várható bitjavulás a tárolt csomópontok számának függvényében, normális esetben

Az én ötletem ezzel szemben az, hogy az adott vödörhöz tartozó intervallumban próbáljunk meg minél jobban elosztani a tárolandó azonosítókat. Ehhez felhasználjuk a *Bucket* osztály *matchingBitCount* mezőjét, mely megadja, az adott vödörben hány prefix bit egyezik meg az azonosítók között. Egy vödör akkor tekinthető ideálisnak, ha az egyező biteket lehagyva az azonosítók teljes bit prefixe megegyezik a vödörben tárolt címek kettes alapú logaritmusával. (Vagyis például 1 egyező bitnél, és 4 tárolt értéknél a következőképpen alakulnak a címek: 100…,101…,110…,111…).

Ebben az esetben is levezethető a fenti esetben is bemutatott várható bitjavulás képlete, mely a következőképpen fog kinézni (ismét csak közelítőképlet):

$$E_b = k + 1 \text{ ,ha } n = 2^k$$

Ezzel a következő értékeket kapjuk:

n(db)	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**
E_b(bit)	2	3	4	5	6	7	8	9	10

Ábra 16. Várható bitjavulás a tárolt csomópontok számának függvényében, javított esetben

Látható, hogy bár annyira nem drasztikus a növekedés, de például a most használ n=8 esetén is ez 13%-os javulást eredményez.

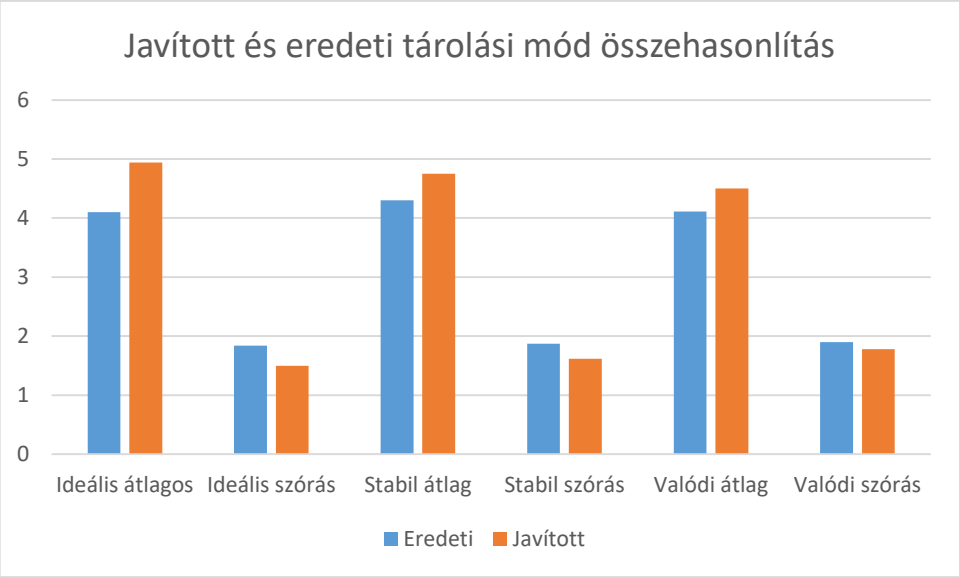
Ezeket a paramétereket három módon mértem meg:

- *Ideális* esetnek azt tekintettem, amikor minden csomópont automatikusan jónak minősült, így a csomópontok az egyes kérésekre olyan címeket is küldtek, amiket amúgy normális körülmények között nem tettek volna. Ezzel jól tudjuk közelíteni, és ezzel ellenőrizni az általunk számolt, ideális értéket.
- *Stabil* esetnek tekintem, amikor már a specifikációt követem a csomópontok megítélésével kapcsolatban, de a hálózat mérete stabil, nem változik.
- *Valódi* helyzetnek tekintem, amikor a hálózatba dinamikusan lépnek ki és be csomópontok. Ezt az előző fejezetben leírt *Turbulence* és *SinChurn* osztály segítségével valósítottam meg.

A mérés során még célszerűnek tekintettem a javulás szórását is mérnem, mellyel lényegében a keresési idők szórásának mértékére tehetek következtetést.

Táblázat 4. A különböző helyzetekben mért bitjavulás várható értéke, illetve szórása

	Ideális átlagos	**Ideális szórás**	**Stabil átlag**	**Stabil szórás**	**Valódi átlag**	**Valódi szórás**
Eredeti	4.4	1.84	4.3	1.87	4.11	1.9
Javított	4.98	1.5	4.75	1.62	4.5	1.78



Ábra 17. A különböző helyzetekben mért értékek grafikus összehasonlítása

Látható, hogy a javított tárolás minden esetben jobb eredményt mutatott, bár ennek hatása már megkérdőjelezhető. Mint azt már tapasztaltam, a keresés során még nagyobb hálózatoknál is általában 3-4 kéréssel eljutunk a legközelebbi csomóponthoz, ezután a maradék 6-8 kérés már csak a talált csomópontok elérhetőségét biztosítja, mely a tárolási formától teljesen független. A 3-4 kéréssel a valódi rendszerben körülbelül 200ms–ot tudunk spórolni, viszont cserébe bonyolultabb lesz a csomópontok tárolása, illetve felmerülhet, hogy ezt a tulajdonságot nem lehet-e kihasználni különböző támadások esetén.

7. Egyéb alkalmazási területek, jövőbeli lehetőségek

Bár ebben a dolgozatban leginkább a torrentezésre való hatásával foglalkoztam, természetesen a DHT alkalmazási területe messze túlmutat azon. Mint ahogy a szimulációkban is láthattuk, helyesen megvalósítva egy nagyon hatékony struktúrát biztosít arra, hogy egy kulcs-érték párt eltároljunk akár több millió résztvevő esetén is, ezzel szétosztva a tárolással együtt járó erőforrás-szükségletet.

Egyre több és több keresőmotor található az interneten, mely peer-to-peer technológiát használ az indexelt adatok tárolására, és a bennük való keresésnél. Általános működésük, hogy a felhasználó böngészését figyelve indexeli az egyes oldalak tartalmát, amit aztán elérhetővé tesz a többi felhasználó számára, DHT-n keresztül. Az ilyen alkalmazások legnagyobb előnye az, hogy garantáltan cenzúra,- és reklámmentes. Könnyen bizonyítható, hogy a Google is a keresési eredményeit szinte teljesen a felhasználóra szabja: figyelembe veszi a tartózkodási helyet, az érdeklődési köröket, politikai és vallásos hiteket, és ez alapján állítja össze az eredményeket. Ráadásul a keresési motorja (természetesen) zárt, míg ezzel szemben a P2P hálózatok általában teljesen nyílt forráskódúak, így könnyen le lehet ellenőrizni, hogy a keresést milyen paraméterek befolyásolják. Ilyen szolgáltatást biztosít például a Faroo, vagy a YaCy.

Egy másik irány lehet a felhőben való fájlok tárolása. A Storj nevű alkalmazás szintúgy a DHT segítségével dolgozik, itt a fő feladat a felhasználó adatainak biztonságos eltárolása. Ezt úgy teszik meg, hogy teljesen kiszervezik a tároláshoz szükséges tárhelyet: bárki jelentkezhet a szabad merevlemezével, melyet hajlandó felajánlani fájltárolásra, és minden gigabájt után fizet a cég a tulajdonosnak. A felhasználó adatait sok apró részre („shard”-ra) bontják, amelyeket titkosítanak, majd feltöltik a DHT hálózatba. Természetesen a redundanciára is figyelnek arra az esetre, ha valaki kiesne a hálózatból, alapértelmezetten 3 különböző helyen tárolnak el minden shardot.

Érdekes terület lehet még a weboldalak P2P alapú tárolása. Egyelőre csak tracker alapú megoldásokról olvastam, melyek lényegében egy torrentet osztanak meg, amit egy speciális kulccsal írnak alá, ezzel igazolva a tulajdonost. Egy ilyen weboldalt ezek után pedig nagyon nehéz, szinte már lehetetlen lenne megszüntetni, mivel ahhoz az oldal megosztásában szerepet játszó összes résztvevőt meg kéne találni, ez pedig adott esetben elérheti az ezres nagyságrendet is.

Én a kutatásaimat a privát DHT hálózatokkal fogom folytatni. Ezek abban térnének el a dolgozatom során bemutatott publikus hálózattól, hogy a hálózat résztvevőit minden egyes csomópont nyilván tartaná, és csak azzal kommunikálna, aki benne van ebben a listában. Egy új csomópont felvételét csakis egy mester csomópont kezdeményezhetne, illetve az ő utasítására törölni is lehet a résztvevők közül. Természetesen ehhez szükséges az egyes csomópontok azonosítása: ezt egy publikus-privát kulcspáros titkosítással oldanám meg, mely segítségével akár maga a kommunikáció is titkosítva folyhatna.

8. Összefoglaló

A dolgozatomban ismertettem a peer-to-peer hálózatok alapjait, illetve az egyik legelterjedtebb fájlmegosztó protokoll, a BitTorrent működését. Ezt követte a DHT-k bemutatása, majd a torrentezésnél is használt Mainline DHT protokollal foglalkoztam.

Leírtam a mérésekhez szükséges szimulátor, a PeerSim működését. Bemutattam az esemény-vezérelt modelljét, illetve az általam tapasztalt hiányosságait. Végül felvázoltam az általam létrehozott komponenseket, melyekkel később a különböző méréseket végeztem.

A kutatásaim során arra jutottam, hogy bár a világ egyik legnagyobb DHT táblájáról van szó, vannak még a megvalósításban súlyos hiányosságok:

- A saját azonosító választást könnyen a rendszer ellen lehet fordítani például egy DDoS támadás indítására. A dolgozatomban nem tértem ki, de ez más támadási formáknak is utat nyit, melyeket mások is vizsgáltak már [10]. Ennek megoldása az azonosítók IP-címhez való részleges kötése.
- Egy másik jelentős probléma az olyan csomópontok tárolása, melyek megtörik a tranzitivitási tulajdonságot. Ahogy a kutatásaim során megtaláltam, ezek leginkább a routerek által alkalmazott NAT folyamat okozzák. Ezen a jó csomópont definíciójának kiegészítésével lehet drasztikusan javítani, miszerint megpróbálunk egy másik portról is kérést intézni a kérdéses csomópont felé, melyre ha válaszol, feltételezhetjük, hogy mások kéréseire is fog tudni válaszolni.
- Megvizsgáltam még egy hatékonyabb tárolási módot, mely bár egyértelműen jobb eredményeket adott az eredeti specifikációhoz képest, ennek hatása normális használat közben elenyésző a keresési időhöz képest. Speciális alkalmazásoknál, ahol elég csak a legközelebbi csomópontot megtalálni, hasznos lehet.

A dolgozatom végén leírtam, milyen jövőbeli területeken lehetne majd tovább hasznosítani az elosztott hash táblákat. Véleményem szerint a jövőben egyre inkább el fognak terjedni a weboldalak DHT-n keresztüli terjesztése, így ugyanis egy szinte megsemmisíthetetlen webhelyet kapunk.

Számomra jövőbeli lehetőségként pedig a privát DHT hálózatok lehetősége tűnt fel, melyet később remélhetőleg részletesebben is meg tudok vizsgálni.

9. Ábrajegyzék

10. Táblajegyzék

11. Hivatkozások

[1] „Peer-to-peer : Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/Peer-to-peer. [Hozzáférés dátuma: 26 október 2016].

[2] „How the Old Napster Worked : HowStuffWorks.com,” [Online]. Available: http://computer.howstuffworks.com/napster.htm. [Hozzáférés dátuma: 26 október 2016].

[3] „Gnutella : Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/Gnutella. [Hozzáférés dátuma: 26 október 2016].

[4] B. Cohen, „BitTorrent Specification : BitTorrent.org,” BitTorrent, 10 január 2008. [Online]. Available: http://www.bittorrent.org/beps/bep_0003.html. [Hozzáférés dátuma: 26 október 2016].

[5] B. Cohen, „DHT Specification : BitTorrent.org,” BitTorrent, 31 január 2008. [Online]. Available: http://www.bittorrent.org/beps/bep_0005.html. [Hozzáférés dátuma: 26 október 2016].

[6] M. J. Alberto Montresor, „PeerSim: A scalable P2P simulator,” *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09),* pp. 99-100, 2009.

[7] G. P. Jesi, „Build a new protocol for the PeerSim 1.0 simulator,” 24 december 2005. [Online]. Available: http://peersim.sourceforge.net/tutorial1/tutorial1.pdf. [Hozzáférés dátuma: 26 október 2016].

[8] M. Jelasity, „A Basic Event Driven Example for PeerSim 1.0,” 18 november 2006. [Online]. Available: http://peersim.sourceforge.net/tutorialed/tutorialed.pdf. [Hozzáférés dátuma: 26 október 2016].

[9] D. F. M Bonani, „A Kademlia module for Peersim,” [Online]. Available: http://peersim.sourceforge.net/code/kademlia.zip. [Hozzáférés dátuma: 26 október 2016].

[10] J. K. Liang Wang, „Real-World Sybil Attacks in BitTorrent Mainline DHT,” [Online]. Available: https://www.cl.cam.ac.uk/~lw525/publications/security.pdf. [Hozzáférés dátuma: 26 október 2016].

[11] F. O. B. K. Raul Jimenez, „Connectivity Properties of Mainline BitTorrent DHT Nodes,” [Online]. Available: https://people.kth.se/~rauljc/p2p09/jimenez2009connectivity.pdf. [Hozzáférés dátuma: 26 október 2016].

[12] F. O. B. K. Raul Jimenez, „Sub-Second Lookups on a Large-Scale Kademlia-Based Overlay,” *2011 IEEE International Conference on Peer-to-Peer Computing (P2P),* pp. 82-92, 2011.

[13] Wikipedia, „Kademlia : Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/Kademlia. [Hozzáférés dátuma: 26 október 2016].

[14] Realnets, „Massive DDoS attacks from lizardstresser : Realnets.com,” 1 július 2016. [Online]. Available: https://www.realnets.com/our-blog/massive-ddos-attacks-lizardstresser/. [Hozzáférés dátuma: 26 október 2016].

[15] C. Hockenberry, „Fear China : furbo.org,” 22 január 2015. [Online]. Available: http://furbo.org/2015/01/22/fear-china/. [Hozzáférés dátuma: 26 október 2016].

[16] A. Norberg, „DHT Security extension : BitTorrent.org,” 15 január 2014. [Online]. Available: http://www.bittorrent.org/beps/bep_0042.html. [Hozzáférés dátuma: 26 október 2016].

[17] „Distributed hash table : Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/Distributed_hash_table. [Hozzáférés dátuma: 26 október 2016].

12. Függelék

12.1. Egy PeerSim-ben használt konfigurációs fájl

```
# ::::::::::::::::::::::::::::::::::::::::::::::::::::::
# :: DHT NAT Configuration
# ::::::::::::::::::::::::::::::::::::::::::::::::::::::

# ::::: GLOBAL ::::::

# Network size
SIZE 50000
MINDELAY  30
MAXDELAY  400

SIM_TIME 1000*60*60*24*30*12

TRAFFIC_STEP  100
OBSERVER_STEP 1000*60

STARTUP_TIME 10 * 60 * 1000


# ::::: network :::::
random.seed 24680

simulation.experiments 1

simulation.endtime SIM_TIME


network.size SIZE


# ::::: LAYERS :::::
protocol.1uniftr peersim.transport.UniformRandomTransport
protocol.1uniftr.mindelay MINDELAY
protocol.1uniftr.maxdelay MAXDELAY

protocol.2unreltr peersim.transport.UnreliableTransport
protocol.2unreltr.drop 0
protocol.2unreltr.transport 1uniftr

protocol.3natlayer NATTransport
protocol.3natlayer.timeout 2 * 60 * 1000
protocol.3natlayer.transport 2unreltr
protocol.3natlayer.natratio 0.4
protocol.3natlayer.fwratio 0.2
protocol.3natlayer.fwtimeout 3 * 1000

protocol.4dht peersim.dht.core.DHTProtocol
protocol.4dht.transport 3natlayer
protocol.4dht.simultaniousQueries 4
protocol.4dht.stopAfterFound false
protocol.4dht.responseNodelistCount 8
protocol.4dht.searchNodeListCount 8
protocol.4dht.bucketLastChangedLimit 15 * 60 * 1000
protocol.4dht.usePerfectStoring false
protocol.4dht.returnQuestionableNode false
protocol.4dht.useNatFilter true
protocol.4dht.agressivePing true


# ::::: INITIALIZERS :::::
```

```
init.1uniqueNodeID CustomIDGenerator
init.1uniqueNodeID.protocol 4dht

init.2statebuilder BootstrapNodeInitializer
init.2statebuilder.protocol 4dht
init.2statebuilder.startuptime STARTUP_TIME
init.2statebuilder.connections 600



# ::::: CONTROLS :::::

# traffic generator
control.0traffic FindGenerator
control.0traffic.protocol 4dht
control.0traffic.step TRAFFIC_STEP
control.0traffic.from STARTUP_TIME
control.0traffic.afterBootstrap false

# turbolence
control.2turbulenceAdd peersim.dht.controls.Turbulence
control.2turbulenceAdd.protocol 4dht
control.2turbulenceAdd.deviation 10
control.2turbulenceAdd.step 1000 * 60
control.2turbulenceAdd.from STARTUP_TIME

control.1sinchurn SinChurn
control.1sinchurn.period 1000*60*60*24
control.1sinchurn.angle -3.1415
control.1sinchurn.amplitude 4000
control.1sinchurn.protocol 4dht
control.1sinchurn.step 1000 * 10



# ::::: OBSERVER :::::
control.3 SearchObserver
control.3.protocol 4dht
control.3.step OBSERVER_STEP
```