Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE

David Brodský

Distribuovaná hašovací tabulka pro klienta protokolu BitTorrent

Katedra softwarového inženýrství

Vedoucí bakalářské práce: Mgr. Pavel Parízek,
Studijní program: Informatika, obecná informatika

2007

Rád bych poděkoval svým rodičům za podporu, jíž se mi od nich neustále dostává. Dík samozřejmě patří i mému vedoucímu Mgr. Pavlu Parízkovi, který měl velmi cenné komentáře, a Jitce Juříčkové za pomoc s kontrolou textu bakalářské práce. V neposlední řadě také musím zmínit vývojáře programu Azureus za poskytnuté informace.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 2.8.2007 David Brodský

Obsah

1 Úvod **6**

2 DHT **9**
2.1 Jemný úvod do Kademlie . . . 9
2.2 Základní parametry . . . 10
2.3 Diverzifikace . . . 12
2.4 Protokol . . . 13
2.5 Bootstrap uzel . . . 14
2.6 Uložení hodnoty do DHT . . . 15
2.7 Vyhledání hodnoty v DHT . . . 16

3 Formát paketů **18**
3.1 Klíče a hodnoty . . . 18
3.2 Síťové souřadnice . . . 18
3.3 Hlavička požadavku . . . 19
3.4 Hlavička odpovědi . . . 20
3.5 Požadavek `PING` . . . 21
3.6 Odpověď `PING` . . . 21
3.7 Požadavek `STORE` . . . 21
3.8 Odpověď `STORE` . . . 21
3.9 Požadavek `FIND_NODE` . . . 22
3.10 Odpověď `FIND_NODE` . . . 22
3.11 Požadavek `FIND_VALUE` . . . 22
3.12 Odpověď `FIND_VALUE` . . . 23
3.13 Odpověď `ERROR` . . . 23
3.14 Požadavek `KEY_BLOCK` . . . 24
3.15 Odpověď `KEY_BLOCK` . . . 24

4 Implementace **25**
4.1 AzDHT . . . 25
4.2 Moduly klienta Tairent . . . 27
4.3 Protokol mezi AzDHT a klientem . . . 28

5 Obsah přiloženého CD **30**
5.1 Instalace 30
5.2 Konfigurace a spuštění 30
5.3 Popis nejčastějších problémů 32

6 Porovnání Kademlie, Azureusovy DHT a BitTorrent DHT **33**

7 Závěr **35**

Literatura **36**

Název práce: Distribuovaná hašovací tabulka pro klienta protokolu BitTorrent
Autor: David Brodský
Katedra: Katedra softwarového inženýrství
Vedoucí bakalářské práce: Mgr. Pavel Parízek
E-mail vedoucího: pavel.parizek@mff.cuni.cz

Abstrakt: Obsahem této práce je přesný popis distribuované hašovací tabulky (DHT) implementované v BitTorrent klientovi Azureus, její použití pro účely distribuovaného trackeru a porovnání s jinou DHT sloužící ke stejnému účelu. Součástí této práce je také implementace rozhraní na bázi TCP serveru umožňujícího snadný přístup k DHT bez nutnosti ji znovu implementovat a specifikace použitého komunikačního protokolu.

Klíčová slova: DHT, Azureus, BitTorrent

Title: Distributed hash table for BitTorrent client
Author: David Brodský
Department: Department of software engineering
Supervisor: Mgr. Pavel Parízek
Supervisor's e-mail address: pavel.parizek@mff.cuni.cz

Abstract: The goal of this study is exact description of distributed hash table (DHT) that is implemented in BitTorrent client Azureus, its usage as a distributed tracker and comparison with another DHT that is used for the same purpose. This study also includes implementation of interface based on a TCP server that provides easy access to the DHT without need to reimplement it. Specification of communication protocol is included as well.

Keywords: DHT, Azureus, BitTorrent

Kapitola 1

Úvod

V současné době vzrůstá potřeba distribuovat velké množství dat od jejich zdroje až ke klientům. Existující systémy můžeme rozdělit na typ klient-server (např. FTP, HTTP...), kdy klient přímo „požádá“ server o data a ten mu je pošle. Celá komunikace tak probíhá výhradně mezi klientem a serverem. Druhým typem jsou peer-to-peer (P2P) sítě, ať již čistě P2P (např. Gnutella, Freenet...) nebo hybridní (např. Napster), kdy existuje centrální server spravující informace potřebné pro distribuci dat.

Výhoda modelu klient-server spočívá obvykle v dosažení vyšších rychlostí přenosu dat, snadného spuštění stahování, menší režie a rozhodující může být i centrální správa. Na druhou stranu klade vyšší nároky na přenosovou kapacitu, neboť data se distribuují z jednoho místa. Naproti tomu P2P přenáší část zátěže na klienty, kteří se tak sami podílejí na distribuci dat a tím šetří prostředky původního zdroje. Pokud je P2P správně navržené, může minimalizovat útoky DoS a ani při výpadku centrálního serveru nedojde k znemožnění fungování sítě a tím i distribuce souborů. Mezi nevýhody P2P pak většinou patří nižší přenosové rychlosti způsobené menší kapacitou linky klientů, kteří již data distribuují, složitější začátek stahování a celkově vyšší režie.

BitTorrent je P2P protokol pro distribuci souborů. Byl navržen a implementován Bramem Cohenem a jeho specifikace je veřejně přístupná [2]. Tento protokol umožňuje distribuovat ohromné množství dat velkému počtu zájemců bez výrazné zátěže na původní zdroj. Oproti jiným P2P systémům se vyznačuje snadným začátkem distribuce a vysokou mírou podílení se každého klienta na distribuci. Mezi další vlastnosti patří velká rychlost transportu souborů a, pokud existuje alespoň jeden zdroj dat, tak i dostupností těchto dat.

Původní protokol definuje dva typy programů, které se podílejí na fungování. Klient je program, který implementuje BitTorrent protokol a pomocí něj je schopen přenášet jakýkoliv typ souborů od ostatních klientů. Aby věděl, jaká data má přenášet, je nejdříve nutné vytvořit soubor s metainformacemi (torrent). V něm se nachází data o názvech distribuovaných souborů, jejich délce a také informace zajišťující integritu přenesených dat. Navíc ještě obsahuje adresu trackeru, což je druhý typ programů podílejících se na funkci systému.

Tracker spravuje informace o tom, kdo daný torrent právě stahuje a tento seznam

poskytuje klientům. To je právě nejslabší místo celého systému. Pro zamezení stahování stačí vypnout tracker určitého torrentu a celý systém přestává fungovat. Tento problém se nyní řeší pomocí distribuované hašovací tabulky (DHT).

Každý klient zapojený do DHT (uzel) uchovává informace o ostatních uzlech tak, aby s nimi mohl efektivně komunikovat, a navíc obsahuje část distribuované databáze. Pro potřeby sdílení dat jsou v ní uloženy informace o klientech, kteří tato data nabízejí nebo stahují. Tak se každý uzel v DHT chová jako tracker pro určitou skupinu torrentů. Je vhodné poznamenat, že DHT obecně umožňuje ukládání libovolných dat a není omezena pouze na torrenty.

Mezi klíčové vlastnosti distribuovaných hašovacích tabulek patří decentralizace, škálovatelnost a tolerance vůči chybám. Tyto vlastnosti zaručují, že je taková síť odolná proti napadení a znemožnění fungování.

V současné době existují dvě implementace DHT, které se používají pro BitTorrent. Obě dvě jsou založeny na DHT se jménem Kademlia, kterou navrhli Petar Maymounkov a David Mazières [4]. První fungující DHT použitelnou pro BitTorrent měl Azureus [1], jeden z nejznámějších klientů pro tento protokol.

Druhá přišla později, ale dostupnost specifikace a její jednoduchost přispěly k tomu, že se používají kromě originálního BitTorrent klienta také např. v klientech KTorrent nebo μTorrent (pro další reference ji nazvěme jako BitTorrent DHT, viz [3]). Použitím DHT je původně hybridní síti umožněno pracovat jako čistě P2P.

Petar Maymounkov a David Mazières popisují Kademlii jako peer-to-peer systém pro ukládání a vyhledávání dvojic ⟨klíč, hodnota⟩. Minimalizuje množství zpráv pro konfiguraci, které musí uzly v systému posílat, aby se o sobě navzájem „dozvěděly“, neboť se tyto informace rozšiřují automaticky jako vedlejší efekt vyhledávání dat.

Klíče mají délku 160 bitů, lze si je tedy snadno představit jako SHA1 haš nějakých dat. To se výborně hodí pro použití právě s torrenty, neboť každý torrent je identifikován SHA1 hašem jedné části jeho metainformací (podrobnosti viz [2]). Pro potřeby BitTorrentu se jako klíč používá tento identifikátor, hodnota obsahuje kontaktní informace o klientech stahujících tato data.

Každý uzel v Kademlii má ID o délce 160 bitů. Dvojice ⟨klíč, hodnota⟩ jsou pak ukládány v uzlech, jejichž ID se nějakým způsobem nachází „blízko“ hodnotě klíče. Algoritmus směrování založený na ID uzlů dovoluje efektivně lokalizovat stroje blízko libovolnému klíči. Vzdálenost mezi identifikátory (buď uzel–uzel, nebo uzel–klíč) se měří pomocí metriky založené na operaci XOR[1].

Cílem této práce je vytvořit specifikaci DHT v klientovi Azureus. Tato specifikace by měla být vhodným doplněním a rozšířením popisu Kademlie, jejíž principy Azureusova DHT využívá, tak, aby mohla sloužit jako reference pro další studium Azureusovy DHT bez nutnosti zkoumat zdrojové kódy. Součástí práce je také implementace BitTorrent klienta Tairent využívajícího Azureusovu DHT pro ověření funkčnosti popsaného řešení. Užitím DHT se v následujícím textu bude myslet právě Azureusova DHT, pokud nebude uvedeno jinak.

Zbytek textu má následující strukturu: V kapitole 2 se nachází upřesněný popis fungování DHT v Kademlii, od základních vlastností, přes rozšíření typických právě

[1] Logická operace exclusive or, podrobnosti viz `http://en.wikipedia.org/wiki/Exclusive_or`.

pro Azureusovu DHT až po přesný popis procesu uložení a vyhledání hodnoty v DHT. Kapitola 3 přesně specifikuje obsah paketů použitých pro komunikaci mezi uzly DHT. Následující kapitola 4 obsahuje popis implementace obecného serveru poskytujícího snadný přístup k DHT bez nutnosti ji psát znovu, specifikaci komunikačního protokolu mezi tímto serverem a klientem a také využití této služby v klientovi Tairent. Obsah přiloženého CD, instalace a použití je popsán v kapitole 5. V kapitole 6 je stručné porovnání Kademlie, Azureusovy DHT a BitTorrent DHT. Poslední kapitola 7 krátce shrnuje získané poznatky.

Kapitola 2

DHT

Kademlia je obecný popis funkce DHT založené na metrice počítané pomocí operace XOR. Cílem této kapitoly není opakování pasáží již popsaných v práci [4], ale upřesnění fungování Azureusovy DHT tam, kde se od původní verze liší.

V rámci zjednodušení je zde proto obsažen pouze jemný úvod do principů funkce Kademlie a použitých pojmů. Přibyly části popisující přidání a vyhledání klíče a jeho hodnot v DHT. Tyto sekce upřesňují algoritmus uvedený v popisu Kademlie vzhledem k rozšířením implementovaných v Azureusově DHT. Této kapitole, vzhledem k jejímu technickému zaměření, může být bez dřívějšího studia [4] těžší porozumět.

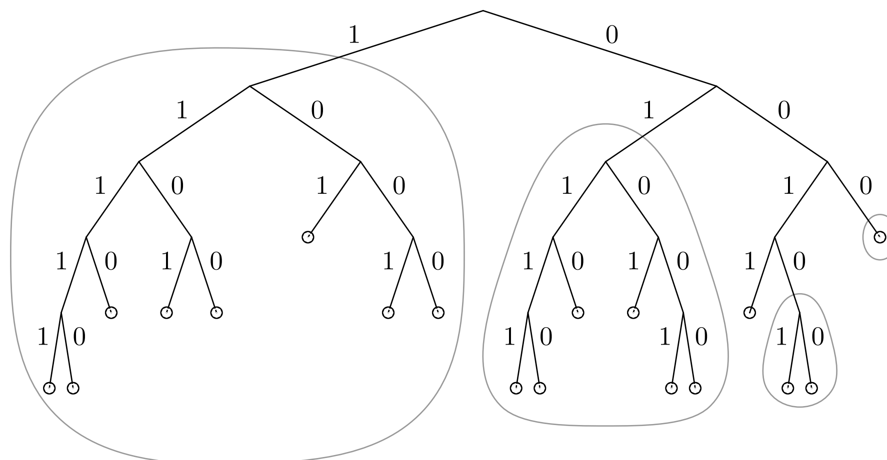
Protokol pro komunikaci mezi uzly se postupně vyvíjel, proto se měnil i formát předávaných zpráv. V následujícím textu se mohou objevit odkazy na konstanty, které udávají chování DHT. Tyto konstanty jsou definovány v tabulce 2.1. Současná minimální akceptovaná verze protokolu je 14.

2.1 Jemný úvod do Kademlie

Kademlia zachází s uzly jako s listy v binárním stromě, kdy pozice každého uzlu je určena nejkratším unikátním prefixem jeho ID. Obrázek 2.1 ilustruje situaci na ukázkovém stromě pro uzel s unikátním prefixem 0011. Pro libovolný zadaný uzel se strom dělí na menší podstromy, které tento uzel neobsahují (v obrázku naznačené šedým oválem).

Název	**Hodnota**	**Název**	**Hodnota**
DIV_AND_CONT	6	XFER_STATUS	12
ANTI_SPOOF	7	SIZE_ESTIMATE	13
ANTI_SPOOF2	8	VENDOR_ID	14
FIX_ORIGINATOR	9	BLOCK_KEYS	14
NETWORKS	9	GENERIC_NETPOS	15
VIVALDI	10	VIVALDI_FINDVALUE	16
REMOVE_DIST_ADD_VER	11	RESTRICT_ID_PORTS	32

Tabulka 2.1: Konstanty pro verze protokolu



Obrázek 2.1: Binární strom Kademlie. Černý bod označuje ve stromě pozici uzlu $0011\cdots$. Šedé ovály vyznačují k-buckety, ve kterých musí uzel $0011\cdots$ znát nějaký kontakt.

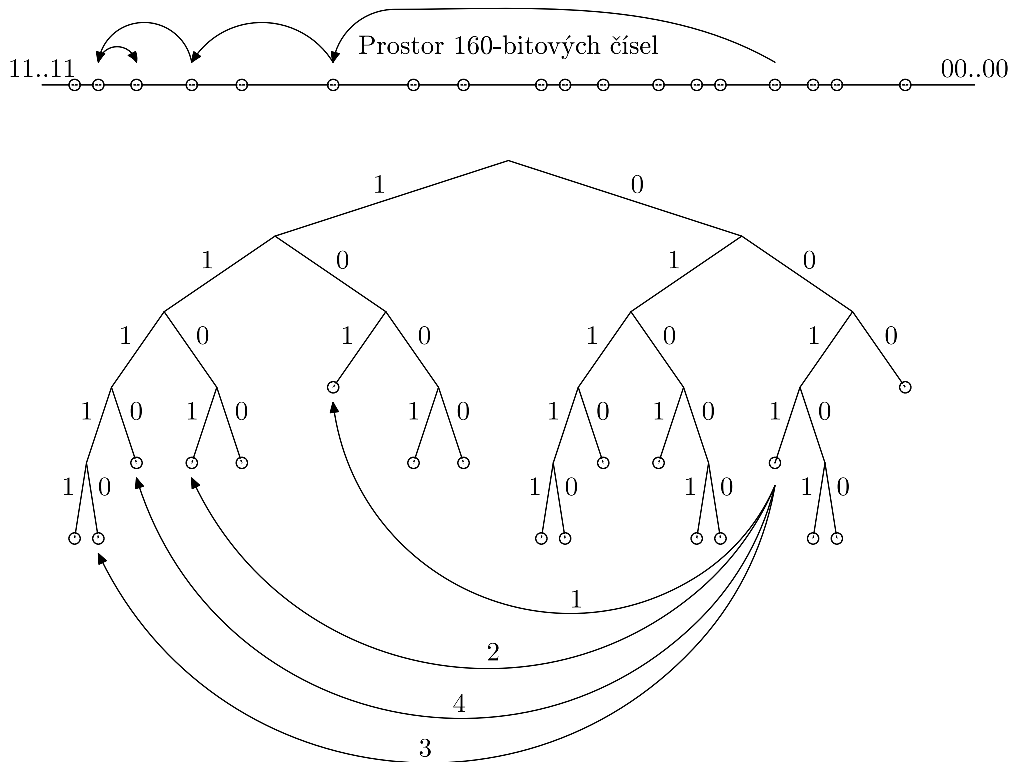
Uzly v Kademlii si ukládají kontaktní informace o ostatních uzlech v systému. Pro každé $0 \leq i < 160$ si každý uzel uchovává seznam kontaktů na ostatní uzly, které se od něj nachází ve vzdálenosti 2^i až 2^{i+1}. Tyto seznamy se nazývají k-buckety. Při nízkých hodnotách i budou k-buckety obecně prázdné, naopak pro velké hodnoty i mohou obsahovat až k kontaktů, kde k je parametr celého systému.

Nejdůležitějším algoritmem v Kademlii je vyhledání k nejbližších uzlů podle zadaného ID. Iniciátor (uzel, který s operací začal) tohoto vyhledávání nejdříve vybere α uzlů ze svého k-bucketu, který je nejblíže zadanému ID, a paralelně jim pošle žádost o navrácení nejbližších kontaktů. Jak se vrací odpovědi s uzly, jejichž ID je blíže hledanému ID, začne jim iniciátor posílat stejnou žádost. Tak se postupně dozvídá o stále bližších uzlech, až žádné další nemůže najít. Obrázek 2.2 ilustruje situaci, kdy uzel s prefixem 0011 vyhledává jiný s ID začínajícím na 1110.

Při tomto procesu se může stát, že některý uzel neodpoví. V takovém případě se označí jako selhávající a odstraní se z uvažovaného seznamu dokud se nevrátí odpověď na dotaz. Pokud neodpoví na více dotazů v řadě, může být v příslušném k-bucketu nahrazen jiným uzlem ze seznamu náhradních, ale pouze v případě, že náhradní uzel odpoví na nějaký dotaz. Tím se zabrání vymazání validních informací při výpadku připojení.

2.2 Základní parametry

DHT umožňuje ukládat libovolné hodnoty asociované libovolnému klíči (tzn. bez omezení na délku nebo tvar klíče). Protože Kademlia definuje, že klíč musí mít délku 160 bitů, je třeba jej před použitím převést tak, aby jeho zakódovaná forma měla



Obrázek 2.2: Vyhledání uzlu podle jeho ID. Uzel 0011··· nachází uzel 1110··· tím, že se úspěšně dozvídá o bližších uzlech. Horní reprezentuje prostor 160-bitových čísel a ukazuje, jak vyhledávání postupně konverguje k cílovému uzlu. První RPC je určeno již známému uzlu 101···, další pak uzlům vraceným z předcházejících RPC.

patřičnou délku. Proto je na klíč před dalším použitím aplikován SHA1 haš. Nadále se již bude předpokládat tato zakódovaná forma.

Azureusova DHT definuje konstantu k rovnu 20. Parametr b je roven 4. Tento parametr způsobuje, že podstrom reprezentující plný k-bucket (viz obrázek 2.1) může být dále rozdělen až tolikrát, kolikrát určuje tento parametr. Uzel si tak může zvýšit znalost systému pro vzdálená ID, čímž se sníží počet kroků nutných při jejich vyhledávání.

Protože hodnoty expirují, je nutné je po určitém čase obnovovat. Doba pro obnovení klíče a jeho hodnot uzlem, který je jejich původním zdrojem, je nastavena na 8 hodin. Pokud ještě hodinu po uplynutí této doby není klíč a jeho hodnota obnoven, vyprší jeho platnost. Pro ostatní uzly je doba obnovení nastavena na 30 minut. Interval pro obnovu k-bucketů, s nimiž neproběhla žádná komunikace, je rovněž nastaven na 30 minut.

ID uzlu se vytváří v závislosti na verzi protokolu, který uzel podporuje. Zkonstruuje se jako SHA1 haš z řetězce ⟨IP adresa uzlu⟩:⟨port⟩, kde adresa je v čitelné podobě. Navíc, je-li verze podporovaného protokolu $\geq$ `RESTRICT_ID_PORTS`, číslo portu se před transformací použije jako zbytek po dělení 1999.

2.3 Diverzifikace

Diverzifikace je technika, kterou Kademlia nepopisuje. Jedná se o způsob, jak zabránit, aby některý uzel byl přetížený, ať již z důvodu uložení příliš velkého množství hodnot v databázi, nebo značného počtu dotazů na určitý klíč [1]. Pokud taková situace nastane, tak dotyčný uzel vydá pokyn k diverzifikaci klíče (obvykle v odpovědi na požadavek) s daným typem diverzifikace. Příjemce pak vytvoří seznam s diverzifikacemi v závislosti na původním klíči a operaci, která se s ním prováděla. Použití této techniky je popsáno v sekcích 2.6 a 2.7.

Tato vlastnost umožňuje rozprostřít velice vytížené klíče mezi uzly, jejichž ID se nenachází blízko sebe (myslí se tím vzdálenost ve stromě dle obrázku 2.1, nikoliv fyzická vzdálenost mezi uzly). Tím se odlehčí nejen uzlu, který leží nejblíže původnímu klíči, ale i uzlům, přes které se provádí vyhledávání.

Rozeznáváme 3 typy možných diverzifikací klíče:

- `DT_NONE = 1` — klíč se nediverzifikuje, vrací se v původním tvaru,
- `DT_FREQUENCY = 2` — klíč se diverzifikuje, pokud je moc dotazů na jeho hodnotu,
- `DT_SIZE = 3` — diverzifikace se provádí, je-li již v uzlu uloženo mnoho hodnot.

Pro jeden klíč existuje 10 možných slotů pro diverzifikace označených od 0 do 9. Sloty slouží pro uložení hodnoty na náhradním místě, pokud ji původní uzel odmítne uložit. Výsledek diverzifikace kromě původního klíče závisí i na použitém slotu, není však závislý jejím typu (od něj se odvíjí, který slot se použije). Diverzifikuje se tak, že se za původní klíč připojí jeden bajt s číslem slotu, do kterého diverzifikace spadá, a na výsledek se aplikuje SHA1 haš.

Vzejde-li požadavek na novou diverzifikaci klíče, která ještě není uložena v seznamu s diverzifikacemi, pak se vytvoří podmnožina slotů, která se použije později, a nastaví se čas, kdy diverzifikace expiruje. Tento čas je určen v náhodném rozmezí od dvou do tří dnů. Všechny tyto informace (klíč, typ, podmnožina slotů a expirace) se uloží do databáze pro vyhledávání již existujících diverzifikací.

Použité sloty pro jednu diverzifikaci jsou závislé na operaci, pro kterou se diverzifikace požaduje, která ještě dále má jeden příznak. Označme tyto parametry jako `put` pro určení zda jde o operaci vložení hodnoty do DHT a `exhaustive` jako další příznak.

Nazvěme operaci zjištění seznamu diverzifikovaných klíčů (jehož použití je popsáno v sekcích 2.6 a 2.7) jako `getKeys`. Tato operace má (kromě klíče, pro který se diverzifikace provádí, a typu diverzifikace) dva parametry `put` a `exhaustive` a jejím výsledkem je seznam s diverzifikovanými klíči. Je-li nastaven příznak `put` a typ je roven `DT_FREQUENCY`, pak výsledný seznam obsahuje diverzifikace, kdy jsou použité všechny sloty od 0 do 9. V případě nastavení parametru `exhaustive` se do výsledného seznamu rovněž přidá původní klíč.

[1] Není přesně specifikováno, kde je nastaven limit pro takovou situaci. Uzel se může rozhodnout, že má dostatek prostředků, a tak nikdy nebude odpovídat s žádostí o diverzifikaci.

Při stejně nastaveném příznaku put a typu diverzifikace rovným DT_SIZE se ze slotů použije pouze jejich podmnožina určená při vytvoření diverzifikace. V případě nastavení příznaku exhaustive se opět do výsledného seznamu přidá původní klíč.

Není-li příznak put nastaven a typ diverzifikace je roven DT_FREQUENCY, pak se pro diverzifikaci použije náhodně vybraný slot[2].

Jestliže je typ diverzifikace rovný DT_SIZE a příznak put není nastaven, tak se pro výsledný seznam použijí buď všechny sloty, a to v případě nastavení příznaku exhaustive, nebo se náhodně zvolí dva různé sloty.

Seznam s diverzifikacemi klíče se konstruuje rekurzivně. Nejprve se pro zadaný klíč, typ diverzifikace a parametry put a exhaustive zjistí, zda je v databázi pro tento klíč uložena diverzifikace. Není-li nalezena, klíč se přidá do výsledného seznamu a rekurze nepokračuje.

V opačném případě se zjišťuje, zda byl již klíč zpracován. Pokud ano a výsledný seznam jej ještě neobsahuje, přidá se do něj. Není-li klíč zpracován, zjistí se seznam jeho diverzifikovaných klíčů pomocí operace getKeys aplikované na diverzifikaci z databáze se zadanými parametry put a exhaustive. Na každý prvek tohoto seznamu se použije rekurze, kdy se kromě klíče ze seznamu předávají také parametry put a exhaustive. Každý prvek seznamu získaného rekurzí se do výsledného přidá pouze když v něm ještě není zahrnut. Pro celou rekurzi je použit jeden seznam se zpracovanými klíči, který se postupně může rozrůstat.

Rekurze se zastavuje při nenalezení diverzifikace v databázi, nebo když již byl daný klíč zpracován.

Je-li nastaven neplatný typ diverzifikace, vrací se původní klíč jako jediný prvek seznamu při požadavku na nalezení existujících diverzifikací. Když je požadavek na vytvoření nové diverzifikace, vrací se prázdný seznam. Za neplatný typ je považovaný rozdílný od DT_NONE, DT_FREQUENCY nebo DT_SIZE v případě požadavku na nalezení seznamu diverzifikací, a různý od DT_FREQUENCY nebo DT_SIZE, jestliže se má vytvořit nová diverzifikace.

2.4 Protokol

Komunikační protokol použitý v Azureusově DHT se skládá z pěti RPC: PING, STORE, FIND_NODE, FIND_VALUE a KEY_BLOCK[3]. První čtyři uvedené RPC jsou popsány v [4], proto jen ve zkratce.

PING se používá pro zjištění, zda je uzel dostupný. RPC STORE slouží k uložení páru ⟨klíč, hodnota⟩ v cílovém uzlu. FIND_NODE používá 160-bitový identifikátor jako parametr. Příjemce tohoto RPC vrací seznam s k uzly, o kterých ví, že se nachází nejblíže zadanému ID. Podobné chování má i RPC FIND_VALUE, pouze v případě, že uzel již dříve obdržel STORE se stejným klíčem, vrátí místo seznamu s uzly uloženou

[2]Diverzifikace je použita pro zjištění hodnoty pro nějaký klíč, který již byl diverzifikován. Při tomto typu diverzifikace již byla hodnota uložena na všech deseti slotech, proto stačí vybrat pouze jednu z nich.

[3]Ve skutečnosti existuje ještě RPC STATS, které slouží pouze pro statistické účely a nemá na funkci žádný vliv.

hodnotu. KEY_BLOCK je opět rozšíření Kademlie ze strany Azureuse.

Obsah a formát paketů použitých pro jednotlivé RPC je popsaný v kapitole 3.

RPC KEY_BLOCK slouží k zablokování, popř. odblokování nějakého klíče. Operace vyhledání nebo uložení hodnoty, která by měla jako parametr zablokovaný klíč, okamžitě ukončí svoji činnost. To znamená, že není možné do DHT vložit hodnotu pro zablokovaný klíč a ani není možné zjistit pro takový klíč přidruženou hodnotu. KEY_BLOCK se posílá pro zablokované klíče při přidání nového kontaktu, nebo pokud se provádí obnova databáze klíčů a jejich hodnot.

Zablokování může být přímé, nebo nepřímé. Při přímém uzel obdrží žádost o zablokování pomocí RPC KEY_BLOCK, nepřímé je způsobeno vyhledáním, nebo uložením zablokovaného klíče. Platnost přímého zablokování klíče nikdy nevyprší. Naopak platnost nepřímé blokace je omezena na 1 týden od jejího vytvoření.

Údaje o blokaci jsou uloženy v jejím požadavku. První bajt udává, zda se jedná o blokaci, nebo její zrušení. Následující tři bajty nemají žádný speciální význam. Bajty na páté až osmé pozici od začátku se interpretují jako celočíselná hodnota s pořadím big-endian. Tato hodnota označuje čas vytvoření blokace (ne nutně vztaženém k nějakému konkrétnímu okamžiku).

Pokud přijde požadavek na blokaci a je již známa nějaká žádost se stejným klíčem, musí se před jeho akceptováním zjistit, zda je možné starý přepsat. V případě, že staré zablokování klíče je přímé a nové nikoliv, je nová žádost zamítnuta. Blokace se rovněž neakceptuje v situacích, kdy je starší než dosavadní žádost.

Jediný, kdo v současné době může nějaký klíč zablokovat nebo odblokovat, je firma Aelitis inc[4]. Každý takový požadavek nese informaci o tom, zda se má provést blokace nebo odblokování, čas tohoto požadavku a blokovaný klíč. Tento požadavek je pak digitálně podepsán[5].

2.5 Bootstrap uzel

Bootstrap uzel je speciální v tom, že jej ostatní uzly využívají při inicializaci k začlenění do systému. Když se chce nový uzel připojit do systému, obvykle zná pouze adresu bootstrap uzlu. V prvním kroku mu pošle požadavek FIND_NODE na nalezení sebe sama. Postupně pak obnovuje k-buckety, které jsou dále než jejich nejbližší soused.

Z uvedeného postupu vyplývá, že bootstrap uzel je zatížen dotazy od mnohem většího počtu uzlů. Aby nebyl ještě více zneužíván, ignoruje všechny RPC kromě

[4]Firma, která vyvíjí Azureuse.

[5]Podepisuje se pomocí RSA. Veřejný klíč, kterým se podpis kontroluje, se vytváří z modula b8 a4 40 c7 64 05 b2 17 5a 24 c8 6d 70 f2 c7 19 29 67 3a 31 04 57 91 d8 bd 84 22 0a 48 72 99 98 90 0d 22 7b 56 0e 88 35 70 74 fa 53 4c cc cc 69 44 72 9b fd da 54 13 62 2f 06 8e 79 26 17 6a 8a fc 8b 75 d4 ba 6c de 76 00 96 62 44 15 b5 44 f7 36 77 e8 09 3d db a4 67 23 cb 97 3b 4d 55 f6 1c 20 03 b7 3f 52 58 28 94 c0 18 e1 41 e8 d0 10 bb 61 5c db bf ae b9 7a 7a f6 ce 1a 5a 20 a6 29 94 da 81 bd e6 48 7e 8a 39 e6 6c 8d f0 cf d9 d7 63 c2 da 47 29 cb f5 42 78 ea 49 12 16 9e db 0a 33 a exponentu 10001h.

jediného: `FIND_NODE` na nalezení odesílatele. Z uvedeného dále vyplývá, že po počátečním začlenění je odstraněn ze směrovací tabulky, neboť již na další dotazy nebude reagovat.

2.6 Uložení hodnoty do DHT

Při vložení hodnoty do DHT se nejprve vytvoří seznam s existujícími diverzifikacemi (viz sekce 2.3) s nastaveným příznakem `put`. Příznak `exhaustive` se nastavuje pouze pokud je lokální uzel původcem klíče a příslušné hodnoty.

Pro každý z těchto klíčů se vyhledá k uzlů, jejichž ID je nejblíže klíči. Jakmile jsou tyto uzly nalezeny, proběhne odstranění zablokovaných klíčů. Tato filtrace je nutná až v této fázi, neboť některý uzel mohl poslat žádost o blokaci příslušného klíče.

Následně se každému z těchto k uzlů pošle žádost o uložení hodnoty. Příjemce může odpovědět s žádostí o diverzifikaci příslušného klíče, nebo s žádostí o blokaci klíče (jedná se o nepřímou blokaci). V prvním případě se opět vytvoří seznam s diverzifikacemi klíče, tentokrát se však vytváří nová diverzifikace. Příznak `put` je při generování seznamu nastaven, příznak `exhaustive` se nenastavuje. Pro každý klíč z tohoto seznamu se opět spustí procedura vložení klíče do DHT. U tohoto nového vložení již nejde o původní klíč, tedy se při příslušné diverzifikaci nenastavuje příznak `exhaustive`.

Pokud odpověď od cílového uzlu obsahuje žádost o zablokování klíče, která je označena jako validní, je nutné ukončit činnost ukládání hodnoty do DHT.[6]

Příjemce paketu s žádostí o uložení klíčů a hodnot (nazvěme ho pro tuto část jako lokální uzel) nejprve zkontroluje, zda mají oba seznamy shodnou délku. Pokud tomu tak není, vrátí odpověď, kdy jsou typy diverzifikací všech klíčů nastaveny na `DT_NONE`.

Pro každý klíč a jeho hodnoty pak následuje operace jejich vložení do databáze. Tato operace vrací typ diverzifikace, která se má s klíčem provést. Je také možné, že byl klíč zablokován. V takovém případě se podle verze použitého protokolu buď vrací chyba `KEY_BLOCKED` (viz sekce 3.13), nebo odpověď `STORE` s diverzifikacemi, jejichž typ je nastaven na 0 (jedná se tedy o neplatný typ, příjemce odpovědi se již nesnaží takový klíč uložit). První typ odpovědi se použije při verzi použitého protokolu $\geq$ `BLOCK_KEYS`, jinak se pošle druhý typ.

První kontrola, která se provádí, zamezuje přeplnění databáze. Pokud je již databáze plná, hodnoty se neuloží a vrací se typ diverzifikace `DT_SIZE`.

Před možným uložením do databáze se nejprve zjišťuje, zda je klíč dostatečně blízko ID lokálního uzlu. Vytvoří se seznam s k aktivními uzly, jejichž ID je nejblíže ukládanému klíči. Jestliže uzel, který žádost přijal, není v tomto seznamu, hodnoty se neuloží a pro příslušný klíč se vrátí typ diverzifikace `DT_NONE`.

[6] Jak je toho dosaženo není definováno. Např. Azureus původní klíč přepíše novým náhodným. Ačkoliv cílový uzel může dostat žádost o uložení hodnoty pro tento náhodný klíč, bude odmítnuta, neboť klíč pravděpodobně nebude dostatečně blízko k ID uzlu.

Následně se pro každou hodnotu testuje, zda je její původce totožný s uzlem, který odeslal žádost o uložení (v takovém případě se jedná o přímé vložení). Pokud je mezi hodnotami taková, kde tato podmínka neplatí, jedná se obnovení hodnoty uzlem, který není jejím původcem. V takovém případě se zjistí, zda je nejvzdálenější z k nejbližších uzlů blíže lokálnímu uzlu, než odesílatel žádosti. Je-li tomu tak, hodnota se neuloží a opět se vrátí `DT_NONE`.

V této fázi se z databáze podle klíče zjistí přidružené hodnoty a typ jeho diverzifikace. Ten se pak vrátí jako výsledek operace, ať se již hodnota přidá, nebo ne.

Nakonec se kontroluje, zda uzel, který požadavek poslal, je skutečně tím, za který se vydává. To je zajištěno porovnáním spoof ID z požadavku s vygenerovaným ID. Toto ID musí být rekonstruovatelné z adresy odesílajícího uzlu a zároveň by jej nikdo cizí neměl uhádnout.[7] Odesílající uzel se jej dozví z odpovědi na požadavek `FIND_NODE`. Jestliže se ID liší od očekávaného, hodnota se nepřidá.

Při přidání hodnoty k příslušnému klíči se možný postup dále dělí podle toho, zda se jedná o přímé nebo nepřímé vložení. Při přímém je situace jednoduchá. V seznamu již může pro původce hodnoty nějaká existovat. Pokud je její verze menší než nově vkládané, stará se odstraní a vloží se nová. Toto porovnání je nutné dělat s ohledem na hodnotu vydanou uzlem, který nenastavuje její verzi a ta je implicitně rovna -1. Při shodné verzi se pouze aktualizuje čas vložení hodnoty, neboť to znamená, že právě byla obnovena a není nutné ji proto znovu obnovovat. Když je verze již uložené hodnoty větší než nově vkládané, nic se neděje a vložení se ignoruje. Nakonec je ještě nutné odstranit všechny hodnoty od stejného původce, které byly vloženy nepřímo (ať již jejich verze byla jakákoliv).

V případě, že se jedná o nepřímé vložení, je nejprve nutné zkontrolovat, zda by se tím nepřepsala hodnota vložená přímo. Kdyby k tomu mělo dojít, vložení se ignoruje. Pokud pro klíč ještě není vložena hodnota se stejným původcem a typ diverzifikace tohoto klíče není roven `DT_NONE`, pak se hodnota neuloží. I u nepřímého vložení se porovnávají verze vkládané hodnoty a hodnoty již uložené v databázi. Pokud ani jedna verze není implicitně nastavena, pak se vložení chová stejně jako přímé. V případě, že některá hodnota má verzi rovnou -1, porovnává se čas vytvoření nové a staré hodnoty. Nová se uloží pouze pokud je vytvořena později, než stará. Jestliže v databázi ještě neexistuje hodnota vložená stejným původcem, není potřeba provádět žádné kontroly přepsání a nová hodnota se vloží.

2.7 Vyhledání hodnoty v DHT

Stejně jako u vkládání hodnoty se i při jejím vyhledávání nejprve vytvoří seznam s diverzifikacemi klíče. Příznak `put` není nastaven a `exhaustive` se bere jako parametr operace.

Každý klíč ze seznamu se vyhledává mezi k uzly, jejichž ID je nejblíže hledanému

[7] Azureus nejdříve zakóduje adresu uzlu, pro který se toto ID vytváří, DES šifrou s náhodným klíčem vygenerovaným na začátku běhu aplikace. Z takto vytvořeného řetězce se poté použijí první 4 bajty.

klíči. Odpověď od cílového uzlu může klíč zablokovat, v takovém případě nebyla nalezena žádná hodnota. Uzel kromě výsledné hodnoty mohl také vrátit typ diverzifikace pro hledaný klíč různý od `DT_NONE`. Nastala-li taková situace a ještě se nevyhledávaly hodnoty pro nějaký diverzifikovaný klíč, vytvoří se nová diverzifikace pro tento klíč a pro každý klíč z výsledného seznamu se spustí nové vyhledávání. Příznak `exhaustive` pro vytvoření seznamu diverzifikovaných klíčů je stále shodný s počátečním parametrem operace.

Vlastnost popsaná v Kademlii, kdy se po každém úspěšném vyhledání hodnoty provede její uložení v uzlech, které tuto hodnotu nevrátily, není v Azureusově DHT použita. I kdyby se o takové chování nějaký uzel pokusil, jeho snaha by byla pravděpodobně odmítnuta, neboť cílový uzel by hodnotu neuložil, protože by byl od ní příliš vzdálený.

Příjemce žádosti o vyhledání hodnoty může odpovědět třemi různými způsoby:

1. Vyhledávanými hodnotami — takový případ nastává, jestliže jsou hodnoty uloženy v databázi a jejich klíč není zablokovaný. Hodnoty se nemusí vejít do jednoho paketu s odpovědí, potom se seznam rozdělí do více paketů a odpovídajícím způsobem se nastaví příznak o pokračování (viz sekce 3.12). Typ diverzifikace u každého odeslaného paketu je stejný a nastavený na typ uložený v databázi u příslušného klíče.

2. Seznamem s k uzly, jejichž ID je nejblíže hledanému klíči, pokud uzel nemá hodnotu uloženou.

3. Chybou obsahující požadavek na blokaci, jestliže je klíč zablokovaný. Když je verze použitého protokolu $\geq$ `BLOCK_KEYS`, pak se vrátí chyba s příslušnou žádostí a jejím podpisem. V opačném případě se vrací prázdný seznam hodnot a typ diverzifikace nastaven na `DT_NONE`. Příznak o pokračování není nastaven.

Kapitola 3

Formát paketů

Tato kapitola se zabývá přesným popisem formátu paketů použitých pro komunikaci. UDP datagramy použité pro přenos zpráv mezi uzly DHT obsahují různé informace v závislosti na typu zprávy, kterou přenášejí. Každý paket se skládá z hlavičky a těla. Rozlišují se dva typy paketů: požadavek a odpověď na něj.

Informace je před posláním třeba serializovat, aby protistrana mohla rekonstruovat původní informaci. Čísla jsou v paketech uložena binárně jako big endian[1]. Rozsah hodnot pro jednotlivé typy je shodný se základními typy v jazyce Java[2].

Hodnoty typu `boolean` se ukládají stejně jako `byte`; `false` má hodnotu 0 a `true` je rovno 1. Při ukládání adresy uzlu se nejprve uloží délka adresy pod typem `byte` (4 pro IPv4, 16 pro IPv6), následuje adresa, kdy nejvýznamnější bajt adresy je uložen jako první. Číslo portu je uloženo jako hodnota typu `short`.

Časový údaj typu `long` se ukládá jako počet milisekund od začátku Epochy[3].

3.1 Klíče a hodnoty

Serializace klíče je triviální podle tabulky 3.1.

Skupiny hodnot se serializují dle tabulky 3.2. Počet hodnot v jedné skupině je určen položkou `VALUES_COUNT`. S každou hodnotou se kromě ní přenáší i další informace: verze (`VALUE_VERSION`), čas vytvoření (`VALUE_CREATION_TIME`), délka hodnoty (`VALUE_LENGTH`), adresa uzlu, který hodnotu poprvé uložil (`ORIGINATOR_ADDRESS`) a příznaky (`FLAGS`). Verze hodnoty je rovna 0, pokud je číslo použitého protokolu $<$ `REMOVE_DIST_ADD_VER`.

3.2 Síťové souřadnice

Síťové souřadnice jsou obsaženy v různých paketech. Mohou sloužit pro vhodnější výběr kandidátů na začátku procesu vyhledávání k uzlů nejbližších zadanému ID. Ukládají se dle tabulky 3.3.

[1] Viz např. `http://http://en.wikipedia.org/wiki/Endianess`.

[2] Viz `http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html`.

[3] 1. 1. 1970 00:00:00 GMT.

Název	**Typ**
KEY_LENGTH	byte
KEY	byte[]

Tabulka 3.1: Položky jednoho klíče

Název	**Typ**
VALUES_COUNT	short
VALUE_VERSION	int
VALUE_CREATION_TIME	čas
VALUE_LENGTH	short
VALUE	byte[]
ORIGINATOR_ADDRESS	adresa
FLAGS	byte

Tabulka 3.2: Skupina hodnot

Název	**Typ**	**Verze protokolu**
COORDINATES_COUNT	byte	≥ GENERIC_NETPOS
COORDINATE_TYPE	byte	≥ GENERIC_NETPOS
COORDINATE_LENGTH	byte	≥ GENERIC_NETPOS
COORDINATE	souřadnice (závisí na typu)	vždy

Tabulka 3.3: Síťové souřadnice

Počet obsažených síťových souřadnic udává položka COORDINATES_COUNT. Typ jedné souřadnice se určuje podle COORDINATE_TYPE. Možné hodnoty jsou 0 pro žádný, 1 pro Vivaldi verze 1 a 5 pro Vivaldi verze 2). COORDINATE_LENGTH značí délku serializované souřadnice.

Popis principu Vivaldi v1 je uveden v [5] a přesahuje rámec této práce. Pokud jsou již v paketu souřadnice uloženy, pak musí obsahovat tento typ. Jeho délka je vždy rovna 16, neboť obsahuje 4 hodnoty typu float s následujícím významem: složka X, složka Y, výška a chyba.

Vivaldi v2 je volitelné a Azureus využívá knihovnu Pyxida[4]. Souřadnice pro tento typ má pět rozměrů typu float. Za nimi je uložena hodnota typu long udávající stáří souřadnice.

3.3 Hlavička požadavku

Položky hlavičky požadavku jsou popsány v tabulce 3.4. CONNECTION_ID značí číslo spojení. Vytváří se jako náhodné číslo s nastaveným nejvýznamnějším bitem. Údaj MESSAGE_TYPE udává typ zprávy, kterou paket přenáší. Položka TRANSACTION_ID je číslo unikátní pro celou komunikaci, které se při spuštění programu náhodně vygeneruje a následně se použije vždy zvětšené o 1.

[4]Domovská stránka projektu je na adrese http://pyxida.sourceforge.net.

Název	**Typ**	**Verze protokolu**
CONNECTION_ID	long	vždy
MESSAGE_TYPE	int	vždy
TRANSACTION_ID	int	vždy
PROTOCOL_VERSION	byte	vždy
VENDOR_ID	byte	$\geq$ VENDOR_ID
NETWORK_ID	int	$\geq$ NETWORKS
LOCAL_PROTOCOL_VERSION	byte	$\geq$ FIX_ORIGINATOR
NODE_ADDRESS	adresa	vždy
INSTANCE_ID	int	vždy
LOCAL_TIME	čas	vždy

Tabulka 3.4: Hlavička požadavku

Název	**Typ**	**Verze protokolu**
MESSAGE_TYPE	int	vždy
TRANSACTION_ID	int	vždy
CONNECTION_ID	long	vždy
PROTOCOL_VERSION	byte	vždy
VENDOR_ID	byte	$\geq$ VENDOR_ID
NETWORK_ID	int	$\geq$ NETWORKS
INSTANCE_ID	int	vždy

Tabulka 3.5: Hlavička odpovědi

Verze protokolu se ukládá pod položkou PROTOCOL_VERSION. Obvykle je rovna nejvyšší verzi, o které se ví, že ji adresát podporuje. VENDOR_ID obsahuje ID autora implementace protokolu (v současné době 00h pro Aelitis inc, 01h pro ShareNet[5] a ffh pro neznámého autora).

Položka NETWORK_ID je rovna číslu sítě, pro kterou je paket určen. Možné hodnoty jsou 0 pro stabilní verzi Azureuse (tedy síť, která je určena pro veřejné použití) a 1 pro CVS verzi (testovací účely).

LOCAL_PROTOCOL_VERSION označuje verzi protokolu, kterou podporuje odesílatel. Při nesplněném požadavku na minimální verzi protokolu použitou v paketu se tato hodnota ukládá až za tělem požadavku.

NODE_ADDRESS obsahuje adresu odesílatele, INSTANCE_ID pomocný údaj běhu programu (vytvořené jako náhodné číslo). Poslední položka (LOCAL_TIME) hlavičky požadavku je čas, kdy byl paket poslán.

3.4 Hlavička odpovědi

Strukturu hlavičky odpovědi udává tabulka 3.5. MESSAGE_TYPE stejně jako u požadavku označuje typ zprávy. Položky TRANSACTION_ID, CONNECTION_ID a INSTANCE_ID

[5] http://www.sharep2p.net/, v době psaní nedostupný server.

Název	Typ	Verze protokolu
SPOOF_ID	int	≥ ANTI_SPOOF
KEYS_COUNT	byte	vždy
KEYS	klíče	vždy
VALUE_GROUPS_COUNT	byte	vždy
VALUES	skupiny hodnot	vždy

Tabulka 3.6: Požadavek STORE

Název	Typ	Verze protokolu
DIVERSIFICATIONS_LENGTH	byte	≥ DIV_AND_CONT
DIVERSIFICATIONS	byte[]	≥ DIV_AND_CONT

Tabulka 3.7: Odpověď STORE

musí mít hodnotu shodnou s příslušnými položkami v požadavku, na který se odpovídá. Význam ostatních položek je stejný jako u požadavku (viz sekce 3.3).

3.5 Požadavek PING

PING má číslo typu zprávy 1024, tělo požadavku je prázdné.

3.6 Odpověď PING

Odpověď PING má typ zprávy roven 1025. Síťové souřadnice (viz sekce 3.2) se v paketu ukládají pouze když je verze použitého protokolu ≥ VIVALDI.

3.7 Požadavek STORE

Číslo typu zprávy STORE požadavku je 1026. Obsah paketu je určen tabulkou 3.6.

SPOOF_ID slouží pro kontrolu odesílatele požadavku. Má zabránit tomu, aby se uzel vydával za nějaký cizí a pod změněnou identitou ukládal falešné hodnoty. Toto ID musí být shodné s číslem dříve obdrženým přes odpověď FIND_NODE.

Počet obsažených klíčů je určen položkou KEYS_COUNT, která musí být shodná s VALUE_GROUPS_COUNT. Popis serializace klíčů a hodnot je uveden v sekci 3.1. Skupiny hodnot jsou uloženy ve stejném pořadí jako klíče. Podle této vlastnosti je možné přiřadit konkretní skupinu hodnot jejich klíči.

3.8 Odpověď STORE

Odpověď STORE má číslo typu zprávy 1027 a její obsah je určen tabulkou 3.7. Ke každé hodnotě uložené v požadavku STORE je možné vrátit typ diverzifikace, která se má s klíčem provést. DIVERSIFICATIONS_LENGTH udává jejich počet a uloženy

Název	Typ
ID_LENGTH	byte
ID	byte[]

Tabulka 3.8: Požadavek FIND_NODE

Název	Typ	Verze protokolu
SPOOF_ID	int	≥ ANTI_SPOOF
NODE_TYPE	int	≥ XFER_STATUS
DHT_SIZE	int	≥ SIZE_ESTIMATE
NETWORK_COORDINATES	síťové souřadnice	≥ VIVALDI
CONTACTS_COUNT	short	vždy
CONTACTS	adresy	vždy

Tabulka 3.9: Odpověď FIND_NODE

Název	Typ
KEY	klíč
FLAGS	byte
MAX_VALUES	byte

Tabulka 3.10: Požadavek FIND_VALUE

jsou pod položkou DIVERSIFICATIONS. Pořadí typů je shodné s pořadím klíčů v požadavku.

3.9 Požadavek FIND_NODE

Tabulka 3.8 udává obsah tohoto požadavku. Číslo typu zprávy je rovno 1028.

3.10 Odpověď FIND_NODE

Tato odpověď má číslo typu zprávy rovno 1029. Obsah paketu je určen tabulkou 3.9.

Význam SPOOF_ID je popsán v sekci 3.7 a použití v sekci 2.6. NODE_TYPE označuje typ uzlu, který odpovídá. Možné hodnoty jsou 0 pro bootstrap uzel (viz sekce 2.5), 1 pro běžný uzel a ffffffffh pro neznámý stav.

DHT_SIZE udává odhadovanou velikost DHT (lze použít 0 pro neznámou velikost). Síťové souřadnice (NETWORK_COORDINATES) jsou popsány v sekci 3.2.

Poslední částí těla paketu jsou uložené kontakty. Jejich počet je specifikován položkou CONTACTS_COUNT. U každého kontaktu je vždy uložena pouze adresa.

3.11 Požadavek FIND_VALUE

Číslo typu zprávy tohoto požadavku má hodnotu 1030, jeho obsah udává tabulka 3.10. Serializace klíče je uvedena v sekci 3.1. Položka FLAGS udává příznaky vyhle-

Název	**Typ**	**Podmínka**
HAS_CONTINUATION	boolean	protokol $\geq$ DIV_AND_CONT
HAS_VALUES	boolean	vždy
CONTACTS_COUNT	short	HAS_VALUES == false
CONTACTS	adresy	HAS_VALUES == false
DIVERSIFICATION_TYPE	byte	HAS_VALUES == true a protokol $\geq$ DIV_AND_CONT
VALUES	skupina hodnot	HAS_VALUES == true

Tabulka 3.11: Opověď FIND_VALUE

Název	**Typ**	**Podmínka**
ERROR_TYPE	int	vždy
SENDER_ADDRESS	adresa	ERROR_TYPE == WRONG_ADDRESS
KEY_BLOCK_REQUEST_LENGTH	byte	ERROR_TYPE == KEY_BLOCKED
KEY_BLOCK_REQUEST	byte[]	ERROR_TYPE == KEY_BLOCKED
SIGNATURE_LENGTH	short	ERROR_TYPE == KEY_BLOCKED
SIGNATURE	byte[]	ERROR_TYPE == KEY_BLOCKED

Tabulka 3.12: Odpověď ERROR

dávání a maximální počet vrácených hodnot je roven MAX_VALUES.

3.12 Odpověď FIND_VALUE

Tabulka 3.11 udává obsah odpovědi FIND_VALUE, jejíž číslo typu zprávy je rovno 1031.

Položka HAS_CONTINUATION udává, zda se hodnoty nevešly do tohoto paketu a jsou uloženy v dalším (který opět může být rozdělen). Zda paket obsahuje hodnoty, nebo seznam s k kontakty, je určeno položkou HAS_VALUES.

Počet uložených kontaktů je roven CONTACTS_COUNT, CONTACTS obsahuje jejich seznam.

Typ diverzifikace klíče vrácené hodnoty udává položka DIVERSIFICATION_TYPE. Způsob serializace hodnot je uveden v sekci 3.1.

3.13 Odpověď ERROR

Pro tuto zprávu bylo vyhrazeno číslo typu rovno 1032, její obsah je určen tabulkou 3.12.

ERROR_TYPE označuje číslo chyby. Možné hodnoty jsou UNKNOWN = 0 pro neznámou chybu, WRONG_ADDRESS = 1 pro chybu způsobenou špatnou adresou uzlu uvedenou v hlavičce zprávy a KEY_BLOCKED = 2 pro chybu signalizující zablokovaný klíč. Poslední se posílá v situacích, kdy uzel dostal požadavek na uložení nebo vyhledání zablokovaného klíče.

Název	Typ
SPOOF_ID	int
KEY_BLOCK_REQUEST_LENGTH	byte
KEY_BLOCK_REQUEST	byte[]
SIGNATURE_LENGTH	short
SIGNATURE	byte[]

Tabulka 3.13: Položky požadavku KEY_BLOCK

V případě chyby WRONG_ADDRESS obsahuje položka SENDER_ADDRESS skutečnou adresu odesílatele paketu tak, jak ji příjemce rozpoznal z UDP hlavičky paketu.

Jestliže je typ chyby nastaven na KEY_BLOCKED, pak paket obsahuje délku požadavku na blokaci (KEY_BLOCK_REQUEST_LENGTH), požadavek (KEY_BLOCK_REQUEST), délku podpisu požadavku (SIGNATURE_LENGTH) a podpis (SIGNATURE).

3.14 Požadavek KEY_BLOCK

Tento požadavek má číslo typu zprávy rovno 1036 a jeho obsah je určen tabulkou 3.13. Posílá se pouze v případě, že cílový uzel podporuje blokování klíče (verze protokolu je $\geq$ BLOCK_KEYS).

SPOOF_ID se používá stejně jako u požadavku STORE (viz sekce 3.7). Následující položky mají shodný význam s příslušnými položkami v paketu ERROR.

3.15 Odpověď KEY_BLOCK

Číslo typu zprávy pro tento paket je rovno 1037. Tělo zprávy je prázdné.

Kapitola 4

Implementace

Implementace DHT pro klienta Tairent[1] se skládá ze dvou samostatných částí. První tvoří upravená verze Azureuse s názvem AzDHT. Modifikace spočívá v zablokování nepoužívaných částí (jako je BitTorrent klient, uživatelské rozhraní a pluginy) a naopak vytvoření serveru, který zpracovává požadavky od klientů a předává je DHT. Druhou částí jsou dva moduly pro Tairent. Nejprve modul `azdht` vytváří rozhraní pro komunikaci s DHT a poskytuje příslušné metody. Druhým modulem je `azdhttracker`, který využívá modulu `azdht` pro implementaci distribuovaného trackeru.

Původní záměr napsat DHT kompatibilní s Azureusovou byl shledán nevhodným a místo něj se zvolilo a implementovalo výše uvedené řešení, které přináší řadu výhod a odstraňuje nevýhody původního návrhu, mezi které patří obrovská duplikace existujícího kódu, možnost zanesení nekompatibilit a v neposlední řadě také nutnost implementovat další distribuovaný systém [5]. Mezi výhody můžeme zařadit 100% kompatibilitu s Azureusem, snadnou udržovatelnost a stabilitu již prověřeného řešení.

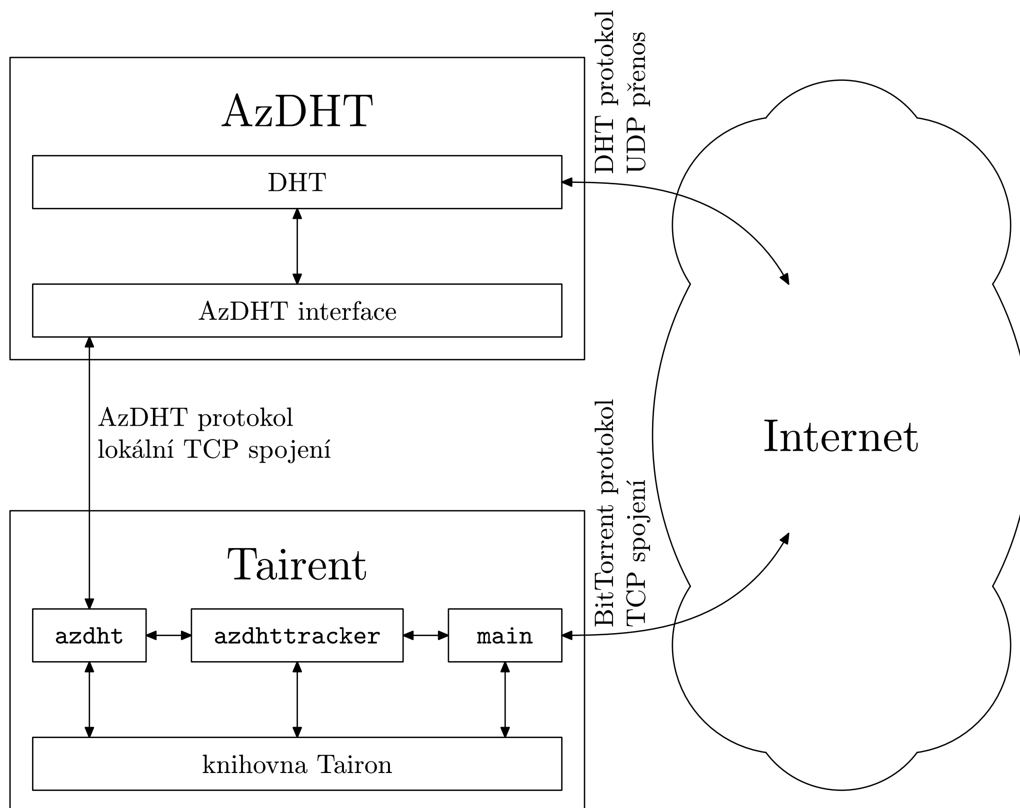
Schéma implementace systému je na obrázku 4.1. Program Tairent pro svou funkci využívá knihovnu Tairon[2], která zapouzdřuje systémová primitiva (jako jsou např. funkce pro práci se sítí, vlákny, nebo regulárními výrazy).

4.1 AzDHT

AzDHT je, stejně jako Azureus, implementován v Javě. Nové třídy byly umístěny do balíku `cz.sinister.trekie.tairent.dht`. Hlavní nese jméno `AzDHT`, další dvě pak `Server` a `Connection`. Přeložený program je distribuován jako jeden JAR archív.

Třída `AzDHT` implementuje metodu `main`, zpracovává parametry předané programu a vytváří potřebné objekty: `DHT` (včetně podpůrných objektů jako je logger) a `Server`. Konfigurace AzDHT se provádí předáním příslušných parametrů přes příkazovou řádku (podrobnosti viz sekce 5.2.

[1] Domovská stránka projektu je na adrese `http://trekie.sinister.cz/tairent`.
[2] Domovská stránka knihovny je na adrese `http://trekie.sinister.cz/tairon`.



Obrázek 4.1: Schéma implementace systému

Stavové informace DHT, které se předávají mezi jednotlivými spuštěními programu, se ukládají do adresáře `.azdht/` v domovském adresáři uživatele, který program spustil. Zahrnují mimo jiné kontakty DHT z předchozího běhu, které se využijí pro rychlejší začlenění uzlu do DHT. V případě neexistence tohoto adresáře je při novém spuštění vytvořen s výchozími hodnotami a uzel se do DHT začlení pomocí bootstrap uzlu (viz sekce 2.5), který je dostupný na adrese `dht.aelitis.com` (s IP 85.31.105.2) a portu 6881.

`Server` implementuje jednoduchý TCP server naslouchajícím na adrese a portu předané jako parametr konstruktoru. Nekonečná smyčka, která běží v novém threadu, akceptuje nová spojení a pro každé vytvoří nový objekt typu `Connection`. Činnost smyčky je možno přerušit zvenčí zavoláním metody `close()`, která uzavře serverový soket a také všechna aktivní spojení.

Činnost objektu `Connection` reprezentující jedno spojení je taktéž založena na nekonečné smyčce, ve které se přijímají požadavky od klienta, předávají se DHT a v případě potřeby se zpět pošle odpověď. Jestliže nastane jakékoliv chyba, je navázané spojení ukončeno. Komunikační protokol je popsán v sekci 4.3.

4.2 Moduly klienta Tairent

Modul azdht sloužící jako rozhraní pro komunikaci s DHT implementuje pouze jedinou třídu AzDHTModule ve jmenném prostoru Tairent::AzDHT. Ukazatel na instanci této třídy je možné získat voláním statické metody self(). Modul se při načtení pokusí spojit s AzDHT na adrese definované konfigurační volbou azdht-address a portu určeném hodnotou azdht-port. Pokud není některá volba nastavena, použije se 127.0.0.1 pro adresu a 7994 pro port. V případě neúspěchu nebo ztráty spojení se modul znovu pokouší spojit jednou za 60 sekund. Zda je modul připojen je možné zjistit pomocí metody isConnected().

Pro základní operace s DHT slouží metody get(), put() a remove(). Všechny mají jako první parametr klíč typu String. V případě metody remove() nejsou potřeba žádné další parametry. Metoda put() vyžaduje kromě klíče a hodnoty (typ String) také příznaky (typ char), které se v DHT ukládají spolu s hodnotou. Poslední metoda get() používá navíc pouze příznaky.

Operace zjištění hodnot pro nějaký klíč pomocí metody get() je jediná, od které se očekává, že vrací nějaké hodnoty. Asynchronní povaha všech operací způsobuje, že není možné použít standardního mechanismu návratové hodnoty pro předání výsledku zpět zájemci, a je proto nutné použít jiný způsob. To je zajištěno zaregistrováním funktoru pomocí metody registerHandler(), který je zavolán jakmile AzDHT vrátí odpověď na požadavek. Funktoru se předává klíč typu String a objekt typu Tairent::Core::BEncode obsahující seznam s vrácenými hodnotami (který může být i prázdný).

Každá vrácená položka v seznamu je typu BEncode::MAP[3]. Samotná řetězcová hodnota je uložena pod klíčem value. K dalším vlastnostem je možné přistupovat pomocí klíčů originatorIP (obsahuje řetězec s IP adresou původce hodnoty), originatorPort (s číslem UDP portu, který využívá DHT původce hodnoty) a flags, kde jsou uloženy číselné příznaky hodnoty.

Všechny požadavky jsou ignorovány, pokud modul není připojen k AzDHT.

Modul azdhttracker využívá azdht k implementaci distribuovaného trackeru. Jeho jediná třída se jmenuje Tairent::AzDHTTracker::AzDHTTrackerModule. Po prvním připojení k AzDHT modul zaregistruje obsluhované torrenty a zároveň si vyžádá seznam klientů, kteří dané torrenty stahují. Zažádání o seznam klientů pak probíhá každých 10 minut.

Identifikátor torrentu (podrobnosti viz [2]) se používá jako klíč pro všechny operace s DHT.

Hodnota asociovaná s identifikátorem torrentu je středníkem nepovinně rozdělena na několik částí. První údaj definuje IP adresu a TCP port klienta (zapsaný ve formátu čitelném pro člověka: aaa.bbb.ccc.ddd:port). IP adresa může chybět, v takovém případě první část tvoří pouze číslo portu a adresa se použije shodná s adresou původce hodnoty. Další části (které jsou klientem Tairent ignorovány) mohou obsahovat příznaky (např. zda klient podporuje šifrovaný protokol) nebo UDP port klienta.

[3]BEncode::MAP je pouze alias pro typ std::map<String, BEncode>.

Pro získání seznamu klientů z DHT si modul vyžádá hodnoty asociované s identifikátorem torrentu. Příznaky dotazu se nastavují v závislosti na tom, zda se torrent ještě stahuje, nebo už je kompletní. V prvním případě má hodnotu 1 (konstanta `AzDHTModule::DOWNLOADING`), ve druhém 2 (konstanta `AzDHTModule::SEEDING`).

Při registraci torrentu se do DHT pouze vloží příslušný pár ⟨identifikátor torrentu, číslo portu⟩. Protože se v hodnotě neukládá IP adresa, je nutné spouštět AzDHT na stejném počítači jako Tairent, nebo použít např. překlad adres. Příznaky se nastavují stejně jako u získávání seznamu klientů. Pro zrušení registrace se zažádá o vymazání hodnoty pro příslušný klíč.

Činnost distribuovaného trackeru pro určitý torrent (ať již z jakýchkoliv důvodů) je možné zakázat nastavením hodnoty 1 pro klíč `private` v `info` části torrentu. DHT se pro takový torrent nikdy nepoužije.

4.3 Protokol mezi AzDHT a klientem

Po navázání spojení mezi klientem a AzDHT serverem může začít symetrický přenos požadavků a možných odpovědí na ně. To znamená, že formát zpráv je v obou směrech shodný. Každá zpráva začíná celočíselnou hodnotu definující délku následující zprávy. Toto číslo má vždy 4 bajty v pořadí big-endian. Samotná zpráva je slovník serializovaný pomocí formátu bencode (popis viz [2]).

Každý požadavek klienta povinně obsahuje položku `request`, která označuje typ požadavku. Možné hodnoty jsou `get`, `put` a `remove`. V případě chybějící položky `request` nebo špatného typu její hodnoty se klientovi pošle chybová zpráva `request is not of type string` (o chybových zprávách viz dále). Při neznámém požadavku se vrací chyba `unknown request`.

Každý typ požadavku navíc povinně obsahuje položku `key` obsahující klíč, jehož se požadavek týká. Pokud položka chybí, nebo když má špatný typ, pošle se klientovi chyba `key is not of type string`.

Požadavek `get` slouží pro zjištění hodnoty (nebo hodnot) asociovaných s klíčem. Tato zpráva musí obsahovat položku `flags`, jež značí příznaky požadavku. Při chybějící položce, nebo pokud není celočíselného typu, se posílá chybová zpráva `flags are not of type integer`.

Jediná situace, kdy server posílá klientovi zprávu (pomineme-li chyby), nastává po dokončení vyhledávání hodnot pro nějaký klíč. Maximální počet těchto hodnot je nastaven na 30, aby se seznam poslaný nějakým uzlem v DHT vešel do jednoho paketu. Odpověď obsahuje položku `response` nastavenou na `get`. Klíč, pro který bylo prováděno vyhledávání, se ukládá do položky `key`. Položka `values` pak obsahuje seznam se zjištěnými hodnotami. Každý element tohoto seznamu je slovník s informacemi o hodnotě. Samotná hodnota je uložena pod klíčem `value`. Další položky obsahují celočíselné příznaky hodnoty pod klíčem `flags`, IP adresu původce hodnoty (klíč `originatorIP`) a konečně položka `originatorPort` obsahuje číslo UDP portu, který používá uzel původce hodnoty.

Požadavek `put` slouží pro uložení hodnoty asociované s daným klíčem do DHT. Povinné položky tohoto požadavku jsou celočíselné příznaky uložené pod klíčem

`flags` a řetězcová hodnota, která se má do DHT vložit. Podmínky a případná chybová zpráva pro příznaky jsou stejné jako u požadavku `get`. V případě chybějící hodnoty nebo jejího špatného typu se klientovi posílá chybová zpráva `value is not of type string`.

Hodnotu je z DHT možno vymazat požadavkem `remove`. Kromě již zmíněného klíče nevyžaduje žádné další parametry.

Chybová zpráva má nastavenou položku `response` na `failed`. Popis chyby se ukládá pod klíčem `error`. Poslední položkou, kterou chybová zpráva obsahuje, je požadavek, který chybu způsobil, dostupný pod klíčem `request`. Po odeslání chybové zprávy se vždy uzavře spojení.

Kapitola 5

Obsah přiloženého CD

5.1 Instalace

Na CD přiloženém k této práci se nachází JAR archív pro program AzDHT, zdrojové kódy pro Tairent a jeho knihovnu Tairon a také Jam[1], nástroj, který se používá pro přeložení knihovny Tairon a programu Tairent. Zdrojové kódy všech částí systému jsou umístěny v adresáři `src/`, JAR archív pro AzDHT pak v adresáři `jar/`.

Kromě již zmíněných částí také CD obsahuje skript `install.sh`, pomocné skripty v adresáři `scripts/`, text bakalářské práce v adresáři `doc/` a další soubory v adresáři `data/`.

Spuštěním `install.sh` z kořenového adresáře CD se v domovském adresáři aktuálního uživatele vytvoří nový adresář `Tairent/`, pokud ještě neexistuje. Na toto místo se pak nakopírují zdrojové kódy programu a knihovny, JAR archív, pomocné skripty pro přeložení a spuštění programu a vytvoří další nezbytná adresářová struktura.

Ke kompilaci je nutné mít v systému nainstalované GCC[2] (podporovaná řada verzí má číslo 4.1) a Python[3] (podporovaná je řada 2.4).

Skript `build.sh` ve vytvořeném adresáři `Tairent/` slouží ke kompilaci zdrojových kódů. Jeho spuštěním se přeloží knihovna a program a vše je tak připraveno ke konfiguraci a spuštění.

5.2 Konfigurace a spuštění

Pro potřeby této sekce se jako kořenový adresář instalace rozumí `Tairent/` v domovském adresáři uživatele. Všechny skripty je nutné spouštět z tohoto adresáře.

AzDHT při spuštění akceptuje volby (předané např. přes příkazovou řádku) pro nastavení UDP portu pro DHT (`-d`⟨`port`⟩ nebo `--dht-port=`⟨`port`⟩), adresy, na které bude server naslouchat (`-l`⟨`adresa`⟩ nebo `--listen`⟨`adresa`⟩), a TCP portu (`-s`⟨`port`⟩ nebo `--server-port=`⟨`port`⟩) a zda se má zapnout logování DHT na

[1] Nástroj je dostupný na adrese `http://www.perforce.com/jam/jam.html`.

[2] GNU Compiler Collection, `http://gcc.gnu.org/`.

[3] Domovská stránka tohoto jazyka je na adrese `http://www.python.org/`.

standardní výstup (přepínač `-v` nebo `--verbose`). Poslední parametr, který je rozeznáván, slouží pro výpis nápovědy (přepínač `-h` nebo `--help`). V případě spuštění programu bez parametrů se použijí přednastavené hodnoty, které by měly ve většině případů vyhovovat: 7995 pro port DHT, `localhost`:7994 jako adresa a TCP port serveru a vypnuté logování.

Samotné spuštění se provede příkazem

```
$ java -jar azdht.jar
```

v adresáři `Tairent/`. Inicializace DHT, která předchází vytvoření lokálního serveru pro komunikaci s rozhraním AzDHT, může chvíli trvat, a proto je po tuto dobu interface nedostupný.

Před spuštěním Tairentu je nutné provést jeho nastavení. Hlavní konfigurační volby jsou umístěny v souboru `Tairent/run/tairent.xml`. Předdefinované hodnoty není nutné ve většině případů měnit:

```
<?xml version="1.0" encoding="utf-8"?>
<config>
   <!-- Adresář, ve kterém jsou umístěny moduly pro
        Tairent -->
   <modules-directory>../tairent/modules</modules>
   <!-- Adresář se souborem torrents.xml -->
   <torrents-directory>../torrents</torrents-directory>
   <!-- TCP port, na kterém Tairent naslouchá -->
   <torrent-server-port>7996</torrent-server-port>
   <!-- Adresa pro spojení s AzDHT -->
   <azdht-address>127.0.0.1</azdht-address>
   <!-- TCP port pro spojení s AzDHT -->
   <azdht-port>7994</azdht-port>
   <!-- Maximální počet navázaných spojení na jeden
        torrent -->
   <max-connections>50</max-connections>
   <!-- Maximální počet slotů pro upload na jeden torrent -->
   <max-uploads>7</max-uploads>
   <!-- Minimální počet slotů pro upload na jeden torrent -->
   <min-uploads>1</min-uploads>
   <!-- Umístění unix soketu pro ovládání -->
   <control-socket>/tmp/tairent.sock</control-socket>
</config>
```

Nastavení obsluhovaných torrentů se nachází v souboru `torrents.xml` v adresáři `Tairent/torrents/` a obsahuje ukázkovou konfiguraci pro torrent Fedora Core 7 na architektuře i386:

```
<torrents>
    <torrent file="Fedora-7-i386.torrent"/>
</torrents>
```

Atribut `file` u elementu `torrent` určuje soubor, ve kterém se nachází informace o torrentu. Pokud cesta neobsahuje oddělovač adresáře (/), musí být soubor umístěný ve stejném adresáři jako soubor `torrents.xml`. Je možné použít atribut `complete="1"` pro nastavení torrentu, který je již celý stažený, a má se proto pouze distribuovat. Všechny torrenty se ukládají do adresáře `run/`.

Po tomto nastavení je možné spustit skript `run.sh`, který nastaví proměnnou prostředí `LD_LIBRARY_PATH` tak, aby linker našel knihovnu Tairon na správném místě, a spustí Tairent. Program AzDHT není tímto skriptem spuštěn.

Skript `tairentctl` (nacházející se v adresáři `tairent/utils/`) slouží ke komunikaci s již běžícím Tairentem. Pokud se spustí s parametrem `torrents`, vypíše informace o torrentech (mezi jinými také kolik již bylo staženo dat), které klient obsluhuje.

5.3 Popis nejčastějších problémů

Na standardní výstup se při běhu programu vypisují informace o jeho stavu. Je tak možné diagnostikovat problémy vzniklé při jeho běhu:

`Info (src/azdht/azdhtmodule.cpp:...) Cannot connect to AzDHT program, retrying`

Nepodařilo se vytvořit spojení s AzDHT. Pokud program běží, adresa, porty a firewall jsou správně nastaveny, je možné, že ještě nebyl vytvořen server pro lokální spojení. Modul `azdht` se pokusí o nové spojení přibližně za 1 minutu.

`Error (src/main/torrentmanager.cpp:248) Cannot load informations about torrents`

Soubor `torrents.xml` nelze načíst. Může být poškozen, chybět, nebo má špatně nastavena přístupová práva.

`Warning (src/main/connection.cpp:282) Invalid protocol length`

Vzdálený klient poslal špatnou úvodní sekvenci při navázání spojení. Jedná se běžné oznámení.

`Info (src/main/torrentmanager.cpp:327) No fast-resume data for torrent ...`

Nejsou dostupné informace z předešlého stahování torrentu. Budou vytvořeny při ukončení běhu aplikace.

`Warning (src/httptrackerclient/client.cpp:80) Tracker failure: ...`

HTTP tracker pravděpodobně vyžaduje rozšíření, které není uvedeno ve specifikaci [2]. Pokud je k dispozici AzDHT, použije se pro získání seznamu klientů tato varianta.

`Caught exception in main: Cannot bind the server socket`

Tato situace obvykle nastává, jestliže byl Tairent spuštěný dříve, než systém ukončil všechna spojení z předchozího běhu. Řešením je počkat, až systém taková spojení odstraní, nebo akci opakovat za několik minut.

Kapitola 6

Porovnání Kademlie, Azureusovy DHT a BitTorrent DHT

Zatímco Azureus implementuje plnohodnotnou DHT umožňující uchovávat klíče i hodnoty libovolného tvaru, BitTorrent DHT je určena čistě pro účel distribuovaného trackeru. Tomu také odpovídají názvy použitých RPC: `GET_PEERS` a `ANNOUNCE_PEER`. První je ekvivalent RPC `FIND_VALUE`, druhé pak `STORE`. `PING` a `FIND_NODE` zůstávají shodné pro všechny tři DHT.

Hodnoty v BitTorrent DHT jsou vázány na protokol IPv4. V budoucnu může být obtížnější přejít na novější protokol IPv6.

ID uzlu se v BitTorrent DHT konstruuje stejně jako v Kademlii, tedy náhodně. To přináší nutnost přenášet toto ID v rámci RPC a navyšuje tak režii přenosu.

V Kademlii i Azureusově DHT hodnoty expirují, což vyžaduje jejich obnovování po určitém čase. Ve specifikaci BitTorrent DHT není toto chování popsáno, což ponechává prostor pro různé chování implementací. Naopak obnovování hodnot v DHT pro tento účel nemá smysl, neboť pokud původce hodnoty přestane být dostupný, není vhodné, aby někdo cizí obnovoval kontakt na něj. To se také projevuje ve způsobu určení IP adresy uzlu obsluhujícího příslušný torrent. V Azureusově DHT je možné ji explicitně specifikovat, navíc se ještě ukládá IP adresa původce hodnoty; naproti tomu v BitTorrent DHT se IP adresa určuje implicitně z paketu nesoucího RPC `ANNOUNCE_PEER`.

Práce [4] se zmiňuje o technice ukládání hodnoty podél vyhledávací cesty. Při úspěšném nalezení hodnoty pro nějaký klíč ji dotyčný uzel uloží v nejbližším uzlu, který ji ještě nemá uloženou. To může umožnit zahlcení uzlu hodnotami, neboť je nemůže odmítnout jako Azureusova DHT z důvodu přílišné vzdálenosti. Azureusova DHT toto chování z principu nepodporuje, specifikace BitTorrent DHT se o ní vůbec nezmiňuje.

V Kademlii nejsou popsány žádné mechanismy pro ochranu před tím, aby se nějaký uzel vydával za úplně jiný (pro potřeby distribuovaného trackeru by tak bylo umožněno do seznamu klientů stahujících torrent přidat uzel, který s ním nemá nic společného). BitTorrent DHT specifikuje předávání tzv. tokenu při RPC `FIND_NODE` a `ANNOUNCE_PEER`, který však nemá přesně stanovený formát. Azureusova DHT od verze `ANTI_SPOOF` používá celočíselné spoof ID.

Azureusova DHT oproti Kademlii i BitTorrent DHT používá nové techniky, zejména diverzifikaci a blokování klíče. První slouží jako ochrana proti přetížení uzlu, zatímco druhá jako ochrana autorských práv.

BitTorrent DHT má maximální velikost k-bucketu nastavenou na 8 (20 u Azureusovy DHT), o parametru b se vůbec nezmiňuje, což zapříčiňuje menší znalost sítě. To se podepisuje na větším počtu kroků nutných pro nalezení cílového uzlu.

Celkově je znát, že Azureusova DHT je oproti BitTorrent DHT vyspělejší. Její složitost a chybějící specifikace může být jeden z důvodů, proč kromě Azureuse není implementována v žádném dalším klientovi. BitTorrent DHT je oproti ní navržena specificky pro účely distribuovaného trackeru, což se podepisuje na zvoleném přístupu.

Kapitola 7

Závěr

Specifikace Azureusovy DHT uvedená v této práci je doplnění popisu Kademlie tak, aby bylo možné vytvořit implementaci kompatibilní s Azureusem. Dalším krokem bude přeložení specifikace do angličtiny a její zveřejnění na AzureusWiki[1], tak, aby byla přístupná všem.

Vytvořením programu AzDHT byla otevřena cesta pro další projekty, které chtějí Azureusovu DHT využívat bez nutnosti její implementace. Tím se mohou ostatní vývojáři zaměřit na své programy a ne na duplikaci existujícího kódu. Funkčnost AzDHT pro účely distribuovaného trackeru je ověřena jejím použitím ve spojení s BitTorrent klientem Tairent.

Porovnání Azureusovy DHT a BitTorrent DHT v kapitole 6 ukazuje odlišný přístup k implementaci velice podobných systémů. Toto porovnání je vhodným doplněním této práce, i když není jejím primárním účelem. Celkově se dá říct, že cíle práce vytyčené v úvodu se podařilo naplnit.

[1] Stránky jsou dostupné na adrese `http://www.azureuswiki.com`.

Literatura

[1] Azureus – Java BitTorrent Client, http://azureus.sourceforge.net.

[2] BitTorrent protocol specification v1.0,
http://www.bittorrent.org/protocol.html.

[3] BitTorrent Trackerless DHT Protocol Specification v1.0,
http://www.bittorrent.org/Draft_DHT_protocol.html.

[4] Petar Maymounkov, David Mazières: Kademlia: A Peer-to-peer Information System Based on the XOR Metric,
http://www.cs.rice.edu/Conferences/IPTPS02/109.pdf.

[5] Frank Dabek, Russ Cox, Frans Kaahoek, Robert Morris: Vivaldi: A Decentralized Network Coordinate System,
http://pdos.csail.mit.edu/papers/vivaldi:sigcomm/paper.pdf.