Katedra Informatiky
Fakulta Matematiky, Fyziky a Informatiky
Univerzita Komenského, Bratislava

Distribuované hashovacie tabuľky

(bakalárska práca)

György Tomcsányi

Odbor: Informatika
Vedúci: doc. RNDr. Rastislav Královič, PhD.

Bratislava, 2010

Čestne prehlasujem, že som túto bakalársku prácu vypracoval samostatne s použitím citovaných zdrojov.

. . . .

Ďakujem vedúcemu tejto bakalárskej práce doc. RNDr. Rastislavovi Královičovi, PhD. za cenné rady a pripomienky. Ďakujem tiež RNDr. Jaroslavovi Janáčkovi za pomoc pri realizácii simulácií.

Abstrakt

Autor:	György Tomcsányi
Názov bakalárskej práce:	Distribuované hashovacie tabuľky
Škola:	Univerzita Komenského v Bratislave
Fakulta:	Fakulta matematiky, fyziky a informatiky
Katedra:	Katedra informatiky
Vedúci bakalárskej práce:	doc. RNDr. Rastislav Královič, PhD.
Rozsah práce:	42 strán
Bratislava, jún 2010	

Táto bakalárska práca podáva prehľad o princípoch fungovania distribuovaných hashovacích tabuliek. Jej cieľom je načrtnúť hlavné problémy pri tvorbe týchto systémov a poukázať na možné riešenia. Práca ďalej prezentuje podrobný popis dvoch implementácií. Praktická časť štúdie predloží výsledky simulácií, ktorých účelom je demonštrovanie vlastností protokolov a ich vzájomné porovnanie.

Kľúčové slová: distribuované algoritmy, hashovanie, chord, kademlia, simulácie

Obsah

1 Úvod **1**

1.1 Motivácia . . . 1

1.2 Základné pojmy . . . 2

1.2.1 Hashovacie tabuľky . . . 2

1.2.2 Distribuované hashovacie tabuľky . . . 3

1.2.3 Konzistentné hashovanie . . . 4

1.3 Praktické aplikácie . . . 5

2 Implementácie **8**

2.1 Chord . . . 8

2.1.1 Prehľad protokolu . . . 8

2.1.2 Štruktúra . . . 9

2.1.3 Pripojenie uzla . . . 12

2.1.4 Stabilizácia . . . 14

2.1.5 Zlyhanie, odpojenie a replikácia . . . 17

2.2 Kademlia . . . 17

2.2.1 Prehľad protokolu . . . 17

2.2.2 Štruktúra . . . 18

2.2.3 Routovacia tabuľka . . . 20

2.2.4 Funkcie protokolu . . . 22

2.2.5 Perzistencia dát . . . 24

3 Simulácie **25**

3.1 Popis simulácie . . . 25

3.2 Bandwidth . . . 27

3.3 Čas odozvy . . . 28

3.4 Počet krokov pri routovaní . . . 29

3.5 Paralelné dotazy . . . 29

3.6 Úspešnosť vyhľadávania 30
3.7 Function fit 31
3.8 Skúsenosti a porovnanie 32

Záver **34**

A Vstupný súbor omnetpp.ini **38**

Zoznam obrázkov

2.1 Rozdelenie adresného priestoru 10
2.2 Routovacia tabuľka – finger table 11
2.3 Rozdelenie siete 20
2.4 Budovanie routovacej tabuľky 22

3.1 Priebeh merania 26
3.2 Vplyv veľkosti siete na vyťaženie siete 27
3.3 Vplyv životnosti uzlov na vyťaženie siete 28
3.4 Čas odozvy 28
3.5 Priemerný počet krokov 29
3.6 Vplyv počtu paralelných správ 30
3.7 Úpešnost vyhľadávania (počet, životnosť) 30
3.8 Úspešnosť vyhľadávania (paralelné správy) 31
3.9 Škálovanie algoritmov 31
3.10 Odhad počtu krokov pre veľké siete 32

Zoznam tabuliek

3.1 Najdôležitejšie nastavenia 26

Kapitola 1

Úvod

Distribuované hashovacie tabuľky (DHT) sú skupina distribuovaných systémov, ktoré poskytujú služby podobné hashovacím tabuľkám. Cieľom tejto práce je poskytnúť prehľad o princípoch fungovania týchto algoritmov. Praktická časť štúdie skúma vlastnosti DHT v reálnych sieťach.

Táto kapitola dáva prehľad základných pojmov, ktoré sú nevyhnutné pre porozumenie fungovania DHT. Uvádza tiež príklady praktických aplikácií týchto systémov. Zvyšok práce je organizovaní nasledovne: Kapitola 2 predstaví dve implementácie DHT. Po technických informáciach Kapitola 3 prezentuje simulačné prostredie a výsledky meraní. Záverečná Kapitola 3.8 zhrnie hlavné výsledky práce.

1.1 Motivácia

Peer-to-peer (P2P) aplikácie sa dostali do povedomia širšej verejnosti službami na zdieľanie súborov. Prvá z nich bola Napster. Ona ešte nebola úplne decentralizovaná, lebo používala centrálny server na vyhľadávanie. Kvôli tomuto komponentu bola celá sieť zraniteľná. Prvá distribuovaná a decentralizovaná sieť bola Gnutella. Ona síce vyriešila problém „single point of failure", ale mala veľmi neefektívne vyhľadávanie, totiž musela dotazy rozposlať všetkým počítačom v sieti. Tieto služby ukázali, že je nárok na distribuované dátové štruktúry, ktoré vedia obslúžiť milióny počítačov. DHT mali spojiť decentralizovanosť Gnutelly a efektivitu Napsteru a umožniť tak vývojárom písať zložitejšie P2P aplikácie.

1.2 Základné pojmy

1.2.1 Hashovacie tabuľky

Hashovacie tabuľky sú v praxi často používané na reprezentáciu dynamických množín. Základným kameňom tejto problematiky je *hashovacia funkcia*. Je to funkcia ktorá mapuje veľkú množinu dát na malé identifikátory - obvykle čísla. Toto číslo môže slúžiť ako index prvkov poľa, na tomto princípe sú založené hashovacie tabuľky, ktoré popíšeme detailnejšie.

Hashovacia tabuľka je dátová štruktúra ktorá asociuje *kľúče* s *hodnotami* a podporuje základné slovníkové operácie:

- Vlož
- Nájdi
- Vymaž

Je to zovšeobecnená verzia klasického poľa. Priama adresácia prvkov poľa zaistí rýchly prístup k prvkom, ale potrebuje k tomu pamäť o veľkosti celého univerza kľúčov. Použitie hashovacej tabuľky sa oplatí v prípade ak počet uložených kľúčov bude výrazne menší ako univerzum. Kým pri použití priamej adresácii je adresa prvku rovná kľúču, pri hashovaní sa adresa získa aplikovaním hashovacej funkcie na kľúč.

Definujme si teraz hashovacie tabuľky trocha formálnejšie. Majme univerzum kľúčov U. Hashovacia tabuľka je dynamická množina v ktorej ku každej uloženej hodnote prislúcha kľúč z U. Prvky sú uložené v poli $P[0..n-1]$. Nech $h : U \mapsto \{0, 1, \ldots, n-1\}$ je hashovacie funkcia. Hodnota prislúchajúca kľúču k je na adrese $P[h(k)]$.

Môže sa stať, že dva rôzne kľúče sa zobrazia na tú istú pozíciu. Tento prípad sa nazýva *kolízia*. Existuje viac prístupov riešenia tohto problému. Pri *hashovaní zreťazením* sa v $P[i]$ nachádza ukazovateľ na hlavu spájaného zoznamu prvkov, ktoré sa hashujú na pozíciu i. Vyššie uvedené operácie sa dajú jednoducho implementovať na týchto zoznamoch. Ďalšou možnosťou je *otvorená adresácia*. Pri nej sú všetky prvky uložené priamo v tabuľke. Ak hľadáme daný prvok, postupne preskúšame (sondujeme) rôzne pozície, pokiaľ nenarazíme na správny index. Postupnosť pokusov závisí od konkrétnej implementácie. Pri *lineárnom sondovaní* sú vzdialenosti medzi skúškami konštantné, a pri *kvadratickom sondovaní* rastú lineárne. Ďalšou možnosťou je *dvojité hashovanie*, ktorá pre výpočet nasledujúcej skúšanej pozície použije

hashovaciu funkciu. Nevýhodou prvých dvoch stratégií je clustrovanie, keďže počiatočná pozícia určuje celú sondovaciu postupnosť. Z toho vyplýva, že ich suma je len $\Theta(n)$. Lepšou voľbou je dvojité hashovanie, lebo pri nej sondovaciu postupnosť určuje počiatočná pozícia a kľúč. To znamená $\Theta(n^2)$ rôznych postupností.

Väčšina programovacích jazykov poskytuje hashovacie tabuľky v nejakej podobe. Obvykle ich nájdeme v štandardnej knižnici daného jazyka, ako napríklad triedy *hash_map* a *unordered_map* v C++ [1] alebo *Hashtable* a *HashMap* v Jave. Iné jazyky ako napríklad PHP a Perl ich používajú na implementáciu asociatívneho poľa. Ďalšia praktická aplikácia je implementácia množín a vyrovnávacej pamäte.

1.2.2 Distribuované hashovacie tabuľky

P2P aplikácie ako Gnutella a Napster motivovali výskum, aby sa naďalej venovali distribuovaným dátovým štruktúram, a využili väčšie kapacity diskov a rýchlejšie pripojenie k internetu. Ak hashovacie tabuľky považujeme za vylepšenie polí, tak DHT sú ich ďalším rozšírením. Vlastnosti týchto štruktúrovaných systémov spĺňajú požiadavky praxe. Sú decentralizované, autonómne a vedia sa prispôsobiť zmenám bez centrálneho servra. Sú škálovateľné a robustné, teda dokážu reagovať na chyby v sieti. Funkčnosť uzlov je rovnaká, a zároveň sa sieť vie vysporiadať s konštantnou fluktuáciou počítačov.

Podobne ako hashovacie tabuľky aj DHT asociujú kľúče s hodnotami. Keďže všetky dáta sú rozdistribuované, hlavný problém je ako nájsť uzol obsahujúci dané informácie. Riešenie je rozdistribuovanie zodpovednosti za kľúče. Procedúra konkrétnej implementácie zaistí priradenie kľúča uzlu, každý uzol bude zodpovedný len za časť množiny všetkých kľúčov. Vyhľadávanie sa takto redukuje na nájdenie uzla ktorý je zodpovedný za kľúč. Rôzne imlementácie DHT majú rôzne stratégie, ale čas ktorý na to potrebujú je zväčša logaritmický. Ďalší problém je prichádzanie a odchádzanie uzlov. Po každej zmene v topológii siete treba zaistiť aby uzly mali správne routovacie informácie. Keďže konkrétne implementáce majú iné prístupy k tejto problematike, budeme sa jej venovať neskôr.

Štruktúra DHT sa delí na dve základné časti:

[1] hash_map nie je súčasťou štandardnej knižnice, ale GNU C++ Compiler a aj Microsoft Visual C++ ho implementuje. unordered_map je súčasťou C++ Technical Report 1, čo je dokument navrhujúci rozšírenie štandardnej knižnice.

Rozdelenie adresného priestoru (Keyspace partitioning)
Uzly pri vstupe do siete dostanú jedinečné identifikátory, čo je číslo z univerza kľúčov. Univerzum je rozdelené na niekoľko častí a každá bude priradená uzlu. Toto mapovanie sa deje na základe funkcie $\delta(k_1, k_2)$ ktorá definuje vzdialenosti medzi kľúčmi. Každý uzol je zodpovedný za kľúče ktorých vzdialenosť k jeho ID je najmenšia na základe metriky δ.

Logická sieť (Overlay network)
Každý uzol pozná len istú časť siete, túto informáciu si uchováva v routovacej tabuľke. Uzly ktoré pozná nazývame jeho susedmi. Tieto vzťahy definujú logickú sieť, ktorá je nezávislá na topológii fyzickej siete. Počet susedov uzla nazývame jeho stupňom. Táto hodnota určuje počet krokov potrebných na komunikáciu s ostatnými uzlami. Viac susedov znamená rýchlejšie routovanie, za cenu väčšej použitej pamäte. V praxi je bežné dovoliť $O(\log n)$ susedov, čo spôsobí routovanie v čase $O(\log n)$.

Implementácie DHT si ukážeme v ďalšej kapitole, kde popíšeme rôzne routovacie a vyhľadávacie protokoly.

1.2.3 Konzistentné hashovanie

Je prirodzené predpokladať, že v distribuovanom prostredí budú uzly priebežne prichádzať a odchádzať. Uzly DHT zistia kde je uložená hodnota prislúchajúca danému kľúču na základe výsledku hashovacej funkcie. Ukážeme, že výber vhodnej hashovacej funkcie je dôležitý pre správne fungovanie DHT.

Príklad 1. Majme hashovaciu tabuľku, ktorá na uloženie údajov používa pole o veľkosti n. Použime nasledujúcu hashovaciu funkciu:

$$x \mapsto (ax + b) \bmod n$$

Ak sa zmení dĺžka poľa, väčšina kľúčov by sa hashovala na iné miesto a preto by sme museli všetky už uložené prvky prehashovať.

Pre DHT to znamená, že rôzne uzly môžu mať rôzne informácie o svete. Chceme dosiahnuť, aby nezávisle od fluktuácie uzlov bolo možné udržiavať DHT v konzistentnom stave. Na problém rôznych „obrazov sveta“ narazili pri tvorbe distribuovanej cache aj autori [KLL+97]. Ich riešenie je *konzistentné hashovanie*, ktoré zaručí nasledujúce vlastnosti:

Plynulosť: ak sa do siete pripojí, alebo z nej odpojí uzol tak počet objektov ktoré treba preniesť na nový uzol bude minimálny.

Rozptyl: suma uzlov cez všetky „obrazy“ na ktoré sa mapuje ten istý objekt je minimálna.

Záťaž: suma objektov cez všetky „obrazy“ ktoré sa mapujú na ten istý uzol je minimálna.

Z vlastnosti *rozptyl* vyplýva, že napriek nekonzistentnosti sa jeden objekt mapuje len na málo uzlov. Rozdistribuovaním objektu na tieto uzly zaistíme, aby k nemu mali všetci prístup a zároveň použijeme pri tom len málo prídavnej kapacity. Jeden z kľúčových podmienok dobrých distribuovaných aplikácií je vyrovnávanie záťaže, vlastnosť *záťaž* zaistí aby ani jeden uzol nemal priradených príliš veľa objektov. Nakoniec vlastnosť *plynulosť* garantuje plynulé prispôsobenie sa zmenám v topológii DHT.

Vo vyššie uvedenej publikácií autori skonštruovali hashovaciu funkciu s týmito vlastnosťami. Väčšina implementácií DHT používa 128 a 160 bitové hashovacie funkcie, ako napríklad SHA-1.

1.3 Praktické aplikácie

Skôr než sa pozrieme na detaily protokolov DHT, ukážeme aké praktické aplikácie ich používajú. DHT podporujú uloženie dát v distribuovanom prostredí ktoré sú indexované kľúčami. Táto jednoduchá funkčnosť umožňuje programátorom vytvoriť rozsiahle distribuované systémy. Niekoľko z potenciálnych použití sú: distribuované file systémy, P2P zdielanie súborov, distribuované cachovanie, multicast a ďalšie.

Jedna z najznámejších aplikácií používajúcich DHT je protokol na zdielanie súborov BitTorrent. Je to síce P2P systém, ale potrebuje uložiť isté dáta na serveri. Protokol rozdelí zdielané dáta na niekoľko menších častí, ktoré sú identifikované ich hashovacou hodnotou. Aby sa dáta publikovali vytvorí sa súbor do ktorého sa tieto hodnoty jednotlivých častí zapíšu a zverejní sa na serveri. Ak si tento súbor niekto stiahne, nájde v ňom hashovacie hodnoty fragmentov, z ktorých vie poskladať požadované dáta. Aplikácia preto skontaktuje server, ktorý eviduje klientov zdieľajúce tieto časti. Aplikácia dostane zoznam kontaktov, od ktorých potom tie časti stiahne, a zároveň informuje o tom server a tak začne aj on zdieľať.

V [Loe08] je popísané rozšírenie tohto protokolu, ktoré umožní, aby sa informácie o dostupnosti dát nemuseli centralizovane manažovať. V súbore na serveri je potrebné uložiť len počiatočný kontakt, ktorý umožní klientom

pripojiť sa do siete. Základom tejto siete je protokol Kademlia, ktorému sa venujeme podrobnejšie v Kapitole 2.2. Do DHT sa ukladajú adresy počítačov, ktoré zdielajú časti podľa ich hashovacích hodnôt. Cieľom tejto modifikácie je umožniť klientom nájsť kontakty v sieti bez potreby centrálneho servera.

DHT pre BitTorrent sa v súčasnosti málo používa, lebo slúži hlavne ako záložný mechanizmus a existujú dve navzájom nekompatibilné implementácie. V [CW07] sú podrobne popísané problémy v implementácii. Hlavné chyby sú časté zlyhanie vyhľadávania, zlý výkon, zastaralé routovacie tabuľky, bezpečnostné nedostatky a rôzne bugy v programe. Tieto ťažkosti sú spôsobené zlou implementáciou DHT. V závere práce autori potvrdzujú, že DHT pre BitTorrent môže mať značné výhody. Ďalej poukazujú na to, že niektoré problémy sa nedajú napraviť bez porušenia spätnej kompatibility a preto navrhujú implementovať novú DHT.

Systém Scribe, ktorý publikovali pracovníci Microsoftu v [RKCD01], je ďalším príkladom použitia DHT. Je to rozsiahla plne decentralizovaná infraštruktúra na manažovanie publikovania a odberu informácií na základe tém. Umožňuje zverejňovať správy kategorizované podľa tém a registrovať sa na ich odber. Po pridaní nových informácií sa vygeneruje udalosť, ktorá na to upozorní odberateľov.

Kostru tejto služby tvorí protokol Pastry, jeden z prvých implementácií DHT. Na vytvorenie novej témy sa vypočíta hash názvu a mena autora a uzol zodpovedný za ten kľúc sa stane koreňom stromu odberateľov. Zverejnené správy v danej téme sa posielajú tomu uzlu. Na registrovanie sa na tému vypočíta uzol hash z názvu a mena autora a pošle koreňovému uzlu tej témy správu. Ak koreňový uzol dostane registračnú správu, pridá si odosielateľa do zoznamu svojich detí a bude mu preposielať všetky udalosti tej témy. Každý uzol sleduje správy, ktoré routuje. Ak medzi nimi nájde registračnú správu na tému, ktorej je odberateľom, tak ju nepošle ďalej. Pridá si odosielateľa do zoznamu detí a bude mu preposielať všetky správy ktoré dostane na danú tému. Takto sa vytvorí multicastový strom odberateľov, ktorý sa informujú navzájom o udalostiach danej témy. Tento prístup zaručí aby očakávaný počet odberateľov registrovaných na jednom uzle bol malý. Umožní, to dobrú škálovateľnosť, vie obslúžiť aj veľké skupiny odberateľov. Scribe má všetky dobré vlastnosti DHT, ako odolnosť voči chybám, samoorganizovanie uzlov a prispôsobenie sa zmenám v sieti. Keďže koreňové uzly sa vyberajú prakticky náhodne, zaručí sa tým vyrovnávanie záťaže.

Botnety sú veľké distribuované systémy a preto bolo len otázkou času kedy autori využijú DHT pri tvorbe takýchto sietí. Botnet je sieť počítačov

infikovaných škodlivým softvérom, ktorú riadi tzv. botmaster. Tieto počítače prijímajú rozkazy, ktoré nariaďujú infikovanie ďalších počítačov, rozosielanie spamu, DDoS útoky, phishing a ďalšie. Kľúčovým bodom návrhu botnetu je komunikácia jednotlivých uzlov a spôsob zadávania príkazov botmasterom. Jeden z obľúbených možností komunikácie je prostredníctvom siete IRC (Internet Relay Chat), čo je veľmi rozšírená forma okamžitej komunikácie cez internet. Používajú sa tiež ďalšie protokoly IM (Instant Messaging), alebo služby ako Twitter a niektorí autori si naprogramujú vlastné riešenia.

Začiatkom roku 2007 sa začal šíriť červ Storm, ktorý vytvoril botnet. Zaujímavosťou Stormu je, že na komunikáciu používa DHT (konkrétnejšie eDonkey/Overnet, čo je P2P sieť na zdieľanie súborov založená na protokole Kademlia). Táto voľba mu zabezpečila niekoľko dobrých vlastností. Sieť je veľmi robustná a dobre škálovateľná. Každý člen botnetu pozná len malú časť zvyšných počítačov, preto je ťažké narušiť sieť alebo odhadnúť jej veľkosť. Vďaka robustnej architektúre bola táto sieť veľmi aktívna a odborníkom sa ťažko robili protiopatrenia.

Kapitola 2

Implementácie

Táto kapitola prezentuje dve implementácie DHT. Sekcia 2.1 popisuje protokol Chord. Je to jeden z prvých algoritmov DHT, vyznačuje sa jednoduchou štruktúrou a dokázateľným dobrým výkonom aj v sieťach s veľkou fluktuáciou. Sekcia 2.2 predstaví protokol Kademlia. Pre ňu je charakteristická symetrická topológia a paralelné dotazovanie. Je to tiež prvý systém, ktorý využíva fakt, že staršie uzly zlyhajú s menšou pravdepodobnosťou.

2.1 Chord

2.1.1 Prehľad protokolu

Chord je jedným z prvých routovacích protokolov pre DHT, ktorý navrhli študenti University of California, Berkeley a MIT v [SMK+01]. Jeho hlavná funkcia je mapovanie kľúča na uzly v sieti. Pre programy používajúce Chord to môže znamenať, že ten uzol je zodpovedný za dáta prislúhajúce kľúču. Keďže v distribuovanom prostredí môžu uzly priebežne prichádzať a odchádzať, Chord sa musí vedieť efektívne prispôsobiť týmto zmenám a odpovedať na dotazy aj v takomto neustálenom prostredí.

Chord je určený pre rozsiahle P2P systémy a preto má niekoľko dobrých vlastností. Prvým z nich je jednoduchosť návrhu a nasadenia. Veľa akademických projektov je príliš komplikovaných na to, aby sa mohli v praxi použiť. Chord je ľahko rozšíriteľný a modifikovateľný, a preto našiel uplatnenie v niekoľkých praktických aplikáciách. Je plne decentralizovaný a jednotlivé uzly môžu byť geograficky distribuované. Keďže nemá žiadne centrálne servre a

každý uzol je rovnocenný celá sieť je robustná, dobre zvláda výpadok uzlov. Uzly majú rovnakú funkčnosť, sú samoorganizujúce. Ďalšia dôležitá vlastnosť je vyrovnávanie záťaže. Konzistentné hashovanie zaistí, aby sa zodpovednosť za kľúče rovnomerne rozdelila a aby pri zmenách v sieti nebolo treba prenášať veľa údajov na nové uzly. Pre takto veľké siete je nevyhnutná škálovateľnosť. Pri tejto implementácii rastie čas vyhľadávania logaritmicky, čo zabezpečí rýchle fungovanie aj v sieťach s miliónmi uzlov. Chord vie obslúžiť s rovnakými nastaveniami malé a veľké siete, nie je potrebné manuálne nastavovať škálovanie. Ďalšia kľúčová vlastnosť je dostupnosť. Sieť sa vie efektívne prispôsobiť zmenám v topológii. Uzol zodpovedný za daný kľúč sa dá vždy nájsť. Testy ukázali, že Chord dobre zvláda fluktuáciu uzlov a výpadky v sieti. Nekladú sa žiadne obmedzenia na použité kľúče, na hashovanie sa používa hashovacie funkcia SHA-1, teda univerzum je 160-bitové.

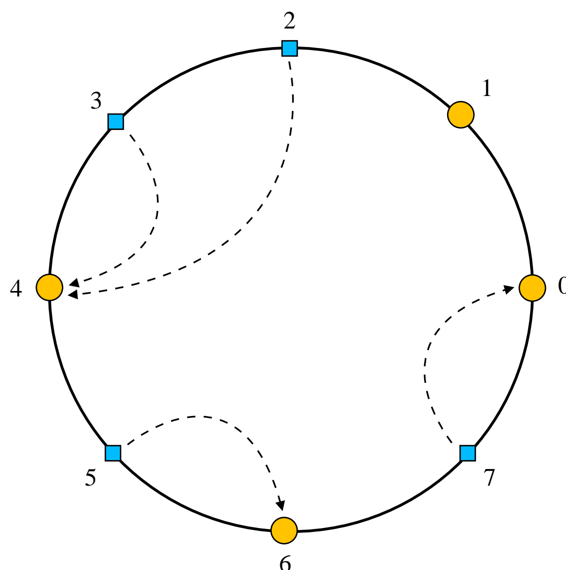
Pre úplnosť uvedieme aj za čo už Chord nie je zodpovedný. Riešenie týchto problémov prenechali autori protokolu programátorom aplikácií, čím zaistili väčšiu flexibilitu a širšiu možnosť využitia. Tieto úlohy sú autentifikácia, cachovanie, replikácia a používateľsky prívetivé pomenovanie dát. Chord principiálne nepodporuje anonymitu, kedže kľúče mapuje na uzly a nie na dáta. Je možné zistiť kto dané dáta do DHT vložil a kto ich vyžiadal. Autori [HWW02] pre P2P publikovacie systémy odolné voči cenzúre navrhli modifikáciu protokolu, aby zaistili anonymitu používateľov služby. Na ďalší problém poukázali simulácie: dotazy v Chord občas nedosiahnu cieľ. Stáva sa to hlavne ak sa dotaz pošle pred dokončením stabilizácie (proces stabilizácie si popíšeme neskôr).

2.1.2 Štruktúra

Ako sme si už vyššie popísali pri tvorbe DHT treba riešiť dva principiálne problémy: skonštruovať logickú sieť a rozdeliť adresný priestor. Protokol Chord priradí hodnotám a uzlom identifikátory z m-bitového univerza ($m = 160$). Identifikátory budeme ďalej volať ID alebo adresa hodnoty či uzla. Logická sieť je kruh v ktorom sú uzly cyklicky usporiadané podla ich ID. Každý uzol si pamätá nasledovníka - *successor pointer*, teda uzol s najmenším ID väčším ako jeho vlastný (mod 2^m). Adresný priestor sa rozdelí na časti medzi uzlami na kruhu. Každý uzol bude zodpovedný za dáta ktorých adresa je menšia ako jeho ID a zároveň väčšia ako ID ostatných uzlov (mod 2^m). Za dáta ktoré sa mapujú na adresu x je zodpovedný uzol priamo nasledujúci x, čo budeme označovať ako $successor(x)$. Obrázok 2.1 znázorňuje príklad

3-bitového univerza so štyrmi uzlami (0, 1, 4 a 6).

V takto definovanej sieti je vyhľadávanie priamočiare, dotaz sa posúva po kruhu až kým nenarazí na prvý uzol s väčším ID. Hoci tento algoritmus je jednoduchý na implementáciu je veľmi neefektívny, lebo správa v najhoršom prípade prejde cez všetkých n uzlov. Aby sa zaistilo rýchlejšie vyhľadávanie každý uzol si pamätá dodatočné informácie o sieti, ktoré nie sú potrebné na korektné fungovanie.

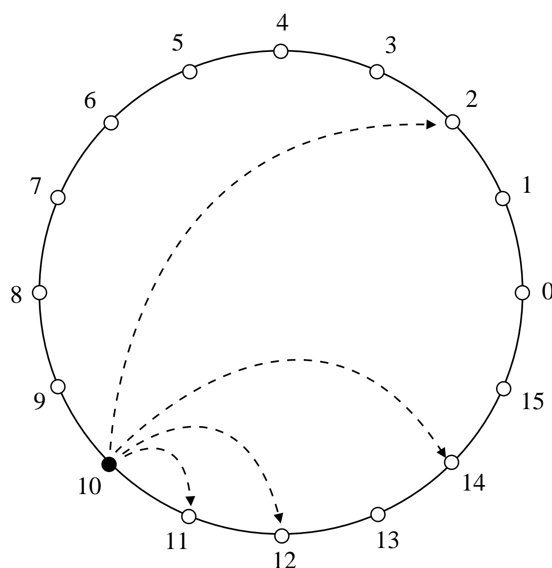


Obr. 2.1: Rozdelenie adresného priestoru

Pri použití m-bitovej hashovacej funkcie si každý uzol udržiava routovaciu tabuľku s maximálne m záznamami, ktorá sa nazýva *finger table*. i-ty záznam v tabuľke uzla u ukazuje na prvý uzol ktorého ID je aspon o 2^{i-1} viac ako ID u, teda $successor(u.id + 2^{i-1})$. Prvý záznam v tejto tabuľke je priamy nasledovník uzla. Záznamy obsahujú ID, IP adresu a port uzla. Časť protokolu - popísaný v nasledujúcej kapitole - je zodpovedný za udržiavanie finger table v konzistentnom stave.

Takto definovaná tabuľka má niekoľko vlastností, ktoré je dobré si uvedomiť. Každý uzol si uchováva informácie len o malej časti siete a zároveň pozná lepšie svoje blízke okolie než ďaleké. Z tabuľky sa obvykle nedá priamo vyčítať nasledovník ľubovoľnej adresy. Obrázok 2.2 znázorňuje jednoduchý príklad routovacej tabuľky v sieti so 16 uzlami. Je to idealizovaný prípad,

keď i-ty pointer ukazuje presne do vzdialenosti 2^{i-1}. V skutočnosti nemusí existovať uzol s takým ID a preto sa pamätá jeho najbližší nasledovník.



Obr. 2.2: Routovacia tabuľka – finger table

Pre získanie hodnoty na adrese x je potrebné nájsť uzol $successor(x)$. Uzol u obvykle nepozná nasledovníka ľubovoľnej adresy, ale vie nájsť vo svojej tabuľke uzol, ktorý je bližšie k x a preto vie viac o tej časti kruhu. u si preto nájde vo svojej finger table uzol ktorého ID je najtesnejšie pred u a pošle mu dotaz, aby našiel uzol zodpovedný za x. Keďže v každom kroku sa nájde uzol bližšie k x, postup sa opakuje kým sa nenájde priamy nasledovník x.

Tento proces sa dá implementovať iteratívne a rekurzívne. Pri iteratívnom prístupe riadi celú komunikáciu uzol, ktorý inicializoval dotaz. Spýta sa sériu uzlov na informácie z ich finger table a tak sa približuje k svojmu cieľu. Pri rekurzívnom prístupe iniciátor pošle svoj dotaz prvému uzlu, ktorý ho prepošle ďalej podľa svojho finger table. Každý vnútorný uzol počas tejto komunikácie routuje dotaz až po cieľ. Iniciátorovi sa vráti len jedna správa - odpoveď.

Ukážeme si pseudokód tohto algoritmu z pôvodnej publikácie protokolu. Notácia $u.foo()$ znamená, že funkcia $foo()$ sa zavolá a vykoná na uzle u. $u.bar$ je prístup k premennej bar uzla u. Funkcia $find_successor(id)$ nájde nasledovníka adresy id. Najprv mu funkcia $find_predecessor(id)$ vráti pria-

meho predchodcu id, jeho nasledovník potom musí byť priamy nasledovník id. Funkcia $find_predecessor(id)$ iteratívne skontaktuje uzly, ktoré mu vrátia ID uzla bližšie k id. Proces sa opakuje kým sa nenájde uzol ktorého nasledovník je už za id.

Funkcia 1 u.find_successor(id)

$u\prime \leftarrow find_predecessor(id)$
return $u\prime.successor$

Funkcia 2 u.find_predecessor(id)

$u\prime \leftarrow u$
while $id \notin (u\prime, u\prime.successor)$ **do**
  $u\prime \leftarrow u\prime.closest_preceding_finger(id)$
end while
return $u\prime$

Funkcia 3 u.closest_preceding_finger(id)

for $i = m$ downto 1 **do**
  if $finger[i].node \in (n, id)$ **then**
    return $finger[i].node$
  end if
end for

Keďže vzdialenosti vo finger table sa zdvojnásobujú, každým krokom iterácie sa skráti vzdialenosť k cieľu aspoň o polovicu. Po j krokoch je maximálna vzdialenosť k cieľu $2^m/2^j$, teda po $\log n$ krokoch $2^m/n$. Očakávaný počet uzlov v takto veľkej časti kruhu je 1 a zároveň $O(\log n)$ s veľkou pravdepodobnosťou. To znamená, že po $\log n$ krokoch potrebujeme maximálne ďalších $O(\log n)$ krokov k cieľu. Z toho vyplýva, že počet krokov na nájdenie uzla zodpovedného za dané dáta je $O(\log n)$. V skutočnosti testy ukázali, že priemerný počet krokov je $\frac{1}{2}\log n$.

2.1.3 Pripojenie uzla

Predpokladáme, že uzly v DHT priebežne prichádzajú a odchádzajú. Našim cieľom je udržiavať sieť v konzistentnom stave, aby sa dotazy mohli spra-

cúvať. Tejto požiadavke je možné vyhovieť ak dokážeme splniť nasledujúce invarianty:

1. každý uzol si korektne pamätá nasledovníka
2. pre každý kľúč k platí, uzol $successor(k)$ je zodpovedný za k.

Kvôli rýchlosti vyhľadávania je ešte žiadúce aby finger tables boli korektné.

Pre zjednodušenie funkcií na pripájanie uzlov si každý uzol pamätá jeho priameho predchodcu, nazývame ho *predecessor pointer*. Ak sa uzol u chce pripojiť do siete, tak predpokladáme, že pozná uzol u' pomocou ktorého sa nainicializuje a pridá do siete. Aby sa uzol u korektne pripojil do siete musí absolvovať tieto kroky:

Inicializovať vlastné pointre u požiada u' aby mu našiel nasledovníka. Túto informáciu si zapíše do finger table. Skontaktuje nasledovníka aby si ten mohol aktualizovať smerník na predchodcu. Potom si u začne napĺňať svoj finger table. Požiada u' aby mu našiel nasledovníkov. Kvôli efektivite si v každom kroku kontroluje, či i-ty záznam vo finger table nie je zároveň aj $(i+1)$-vý. Časová zložitosť celého procesu je $O(\log^2 n)$. Ďalšia optimalizácia je skonštruovať finger table na základe tabuľky nasledovníka a predchodcu, čo zníži zložitosť na $O(\log n)$

Aktualizovať pointre ostatných Keď už u má korektné informácie o sieti treba zaistiť aby aj uzly v sieti vedeli o ňom, to znamená aktualizovať finger table uzlov, ktorým sa u mohol stať fingerom. u bude i-ty finger niektorého uzla práva vtedy, ak ho u nasleduje aspoň o 2^{i-1} pozícií a i-ty finger toho uzla nasleduje za u. Prvý taký uzol, ktorý to môže splniť je predchodca $n - 2^{i-1}$. Uzol u prejde všetky uzly ktoré spĺňajú tieto podmienky a aktualizuje im finger table. Časová zložitosť procesu je $O(\log^2 n)$.

Preniesť dáta Keď si uzol u zaujme svoje miesto v sieti bude zodpovedný za dáta ktorým sa stal nasledovníkom. Čo sa má konkrétne udiať určuje softvér na vyššej úrovni. Implementácia tejto funkcionality je jednoduchá, lebo u sa mohol stať nasledovníkom len tých kľúčov za ktoré pred tým zodpovedal jeho nasledovník. Ak chce softvér preniesť tieto dáta zo starého vlastníka na nový, tak stačí skontaktovať jeden uzol.

2.1.4 Stabilizácia

Protokol na pripájanie uzlov popísaný v predošlej sekcii sa snaží udržiavať finger table všetkých uzlov v korektnom stave. Vo veľkej sieti s veľkou fluktuáciou uzlov je však náročné spĺňať stanovené invarianty a preto sa namiesto striktného aktualizovania všetkých uzlov pri pripojení používa periodická stabilizácia. Pri tomto prístupe sa vzdáme prísnych požiadaviek, ale - ako sa v praxi ukázalo - vieme udržiavať stále meniacu sa sieť v dostatočne konzistentnom stave. Základom stabilizácie je aktualizovať smerníky na nasledovníkov hneď pri pripojení nového uzla, čo nám zaistí korektné vyhľadávanie. Časť protokolu podľa týchto smerníkov periodicky aktualizuje finger table, čo zrýchli vyhľadávanie.

Pri jednoduchom pripájaní uzlov istý čas trvá kým sa aktualizujú pointre na nasledovníkov a finger table uzlov v sieti. Vyhľadávania bežiace v tom čase môžu zlyhať. Ideálny je prípad, keď sa všetky pointre stihnú aktualizovať, a tak sa dotaz spracuje bez problémov. Menej dobrý, ale stále akceptovateľný je stav, ak sú všetky smerníky na nasledovníkov korektné. V tom prípade sa dotaz správne vyhodnotí, ale potrebuje k tomu viac času. Problematický je stav, keď sa ešte ani smerníky na nasledovníkov nestihli aktualizovať. Vtedy dotazy neuspejú, a chyba je ohlásená softvéru vyššej úrovne. Ukážeme si algoritmus, ktorý rieši problém korektnosti a výkonu zvlášť. Najvyššiu prioritu má korektnosť, preto protokol pri pripájaní uzla v prvom rade zaistí správne fungovanie už existujúcej siete a novopripojenému uzlu umožní vyhľadávať. Ďalší krok je periodická stabilizácia, ktorá oznámi príchod nových uzlov, aby sa mohli aktualizovať pointre na nasledovníkov a predchodcov. Posledný krok je pravidelná kontrola záznamov vo finger table všetkých uzlov.

Funkcia 4 $u.join(u\prime)$

$predecessor \leftarrow nil$
$successor \leftarrow u\prime.find_successor(n)$

Ukážeme si pseudokódy funkcií na pripojenie uzla do siete a následnú stabilizáciu. Ak sa uzol u chce pripojiť skontaktuje uzol u', ktorý už v sieti je. Vo funkcii *join* u zavolá funkciu *find_successor* uzla u' a tá mu vráti jeho nasledovníka. Po uložení smerníka bude u vedieť vyhľadávať v sieti, ale sieť o ňom ešte nebude vedieť. Periodicky zavolaná funkcia *stabilize* aktualizuje informácie o nasledovníkoch. Zistí či predchodca nasledovníka nie je nový uzol. Ak áno tak si ho označí ako nasledovníka a informuje ho o zmene vola-

Funkcia 5 $u.stabilize()$

$x \leftarrow successor.predecessor$
if $x \in (u, successor)$ **then**
  $successor \leftarrow x$
end if
$successor.notify(u)$

Funkcia 6 $u.notify(u\prime)$

if $predecessor = nil$ **or** $u\prime \in (predecessor, u)$ **then**
  $predecessor \leftarrow u\prime$
end if

ním funkcie *notify*. V tomto kroku sa uzly dozvedia o novopripojených. Ak sa na uzle zavolá funkca *notifiy* tak si overí, či to môže byť jeho predchodca, ak áno tak si prepíše smerník. Po ukončení tejto fázy je nový uzol plnohodnotným členom siete, smerníky na nasledovníkov a predchodcov sú korektne nastavené, vyhľadávanie bude fungovať. Uzly ktoré ešte nie sú vo finger table ostatných, môžu spomaliť funkciu *find_predecessor*, ale po smerníkoch na nasledovníkov sa správy nakoniec doručia. Ďalšia periodicky volaná funkcia je *fix_fingers*, ktorá obnoví finger table. Vyberá si náhodné záznamy z tabuľky a volaním funkcie *find_successor* overí, či sú ešte aktuálne. Na príklade si ukážeme fungovanie procesu pripájania.

Príklad 2. Uvažujme situáciu keď uzol u_1 má nasledovníka u_2, zároven u_2 má predchodcu u_1. Ak sa do siete pripojí uzol u, ktorého ID padne medzi u_1 a u_2 tak sa udeje nasledovné:

1. vo funkcii *u.join* si nastaví *u.successor=*u_2.

2. po čase u zavolá u_2*.notify*, nastaví sa u_2*.predecessor=u*.

3. u_1 zavolá u_1*.stabilize*. Vypýta si u_2*.predecessor* (čo je u). Nastaví sa u_1*.successor=u*.

Funkcia 7 $u.fix_fingers()$

$i \leftarrow$ nahodny index do finger[]
$finger[i].node \leftarrow find_successor(finger[i].start)$

4. u_1 zavolá *u.notify*, nastaví sa *u.predecessor*=u_1

Po tomto kroku je u súčasťou siete. Časom každý uzol zavolá funkciu *fix_fingers*, čím sa u dostane do finger table príslušných uzlov.

Použitie tohto algoritmu zaistí dve dobré vlastnosti:

1. Ak uzol raz úspešne spracuje dotaz, bude to vedieť spraviť v ľubovoľnom čase v budúcnosti.

2. Po poslednom pripojení budú po istom čase všetky smerníky na nasledovníkov korektné.

Prvé tvrdenie vyplýva z toho, že ak vie uzol skontaktovať niektorý iný, tak to bude vedieť spraviť aj neskôr. Sieť sa vytvára postupne, začína len jedným uzlom, ku ktorému sa pridávajú ďalší. Indukciou ľahko dokážeme, že každý uzol bude vedieť skontaktovať toho prvého, lebo do siete sa môžu pripojiť len cez neho, alebo cez niekoho kto sa už do siete pripojil a teda vie skontaktovať ten pôvodný uzol. Každé volanie funkcií *stabilize* a *notify* vedie ku korektnejším smerníkom. Ak majú dva uzly rovnakého nasledovníka, tak si on vo funkcii *notify* vyberie z nich toho ktorý je bližšie. Z toho vyplýva, že ak do siete neprichádzajú ďalšie uzly, tak sa po čase dostaneme do stavu, kedy každý uzol má iného nasledovníka. Túto vlastnosť spĺňajú len kruhy. Keďže každý uzol vie skontaktovať prvého vieme, že to musí byť jeden súvislí kruh.

Ukázali sme si, že po pripojení uzla bude sieť schopná spracovávať dotazy korektne. Teraz si popíšeme aký dopad má pripájanie uzlov na výkon vyhľadávania. Ak má uzol dobre naplnenú finger table, tak ju môže použiť aj po pripojení uzlov. V časti 2.1.2 sme si ukázali, že pri routovaní sa po $\log n$ krokoch priblížime dosť blízko na to, aby sme zvyšok mohli lineárne prejsť a stále zachovať logaritmický čas. Tento argument môžme použiť aj tu, za predpokladu, že sa naraz nepripojí veľa uzlov. Je dokázané, že ak sa do siete o veľkosti n pripojí nanajvýš n nových uzlov, tak čas vyhľadávania bude stále $O(\log n)$ s veľkou pravdepodobnosťou. Vyplýva to z toho, že s použitím existujúcich finger table sa vieme dostať k pôvodnému predchodcu hľadaného uzla. Keďže medzi ľubovoľné dva uzly sa pripojí s veľkou pravdepodobnosťou maximálne $O(\log n)$ nových uzlov, bude treba lineárne prejsť len $O(\log n)$ uzlov.

2.1.5 Zlyhanie, odpojenie a replikácia

Zatiaľ sme skúmali vyhľadávanie kľúčov a pripájanie uzlov. Teraz sa pozrieme čo sa stane ak sa uzol dobrovoľne odpojí alebo zlyhá. Dôležitým bodom je aj replikácia dát, čomu sa tiež budeme venovať.

Nutnou podmienkou úspešného vyhľadávania sú korektné smerníky na nasledovníkov. Treba rátať s odpojením uzlov, kvôli čomu budú niektoré smerníky ukazovať na neexistujúce uzly. Riešenie tohto problému je namiesto jedného nasledovníka si pamätať zoznam nasledovníkov. Funkcia *stabilize* sa zmení nasledovne: pri aktualizácii zoznamu si uzol u vypýta zoznam svojho nasledovníka. u si potom na prvé miesto pridá $u.successor$ čím získa vlastný zoznam nasledovníkov. Routovanie bude fungovať rovnako, ak sa uzlu nepodarí skontaktovať nasledovníka, tak skúsi ďalšieho zo zoznamu.

Táto modifikácia protokolu má dobrý vplyv aj na bezpečnosť siete. Na narušenie siete by museli vypadnúť všetky uzly v zozname, čo je pre rozumne veľký zoznam málo pravdepodobné. Keďže protokol Chord priraduje identifikátory uzlom aplikovaním hashovacej funkcie na jeho ip adresu a port - teda „náhodne“ - na cielený útok proti sieti nestačí napadnúť počítače z jedného regiónu.

Vytváranie zoznamu nasledovníkov napomáha aj aplikácii používajúcej Chord pri riešení replikácie. Typicky sa vytvoria kópie dát na nasledovníkoch uzla, ktorý je za ne zodpovedný. Po zlyhaní uzla sa dáta nestratia a zároveň budú uložené na uzle, ktorý sa stal novým vlastníkom toho kľúča. Keďže uzol si sleduje zmeny vo svojom zozname vie informovať aplikáciu, kedy je potrebné vytvoriť ďalšie repliky.

Dobrovoľné odpojenie zo siete nie je potrebné riešiť zvlášť, lebo predošlý postup zaistí korektné nastavenie smerníkov, ale ak uzol oznámi svoje rozhodnutie môže tým zvýšiť výkon Chordu. Pred odpojením môže poslať svoje dáta jeho nasledovníkovi, čo zabezpečí dostupnosť aj tých dát, ktoré sa ešte nestihli replikovať. Môže tiež informovať svojho predchodcu a nasledovníka a pomôcť im nastaviť správne smerníky.

2.2 Kademlia

2.2.1 Prehľad protokolu

Kademlia je protokol pre DHT, ktorý publikovali pracovníci New York University v [MM02]. V súčasnosti ho používa viac veľkých P2P systémov ako

KAD a BitTorrent. Má niekoľko dobrých vlastností, ktoré iné implementácie nespĺňajú naraz. Cieľom protokolu je zaistiť konzistentnosť, vysoký výkon a routovanie s minimalizáciou latencie. Dôležitým prvkom návrhu je aj distribuovanie routovacích informácií, uzly získavajú nové informácie o sieti každou správou, ktorú dostanú.

Ako Chord aj Kademlia musí spĺňať základné požiadavky DHT ako napr. distribuovanosť, decentralizovanosť, škálovateľnosť, vyrovnávanie záťaže a dostupnosť. Tu si popíšeme vlastnosti, ktoré sú charakteristické pre protokol Kademlia. Prvá z nich je metóda manažovania routovacích informácií. Nie je potrebné posielať špeciálne správy na aktualizáciu tabuliek, namiesto toho získavajú uzly informácie o sieti z každej správy ktorú dostanú.

Adresný priestor je 160-bitový, ako pri väčšine DHT, ale vzdialenosti sa počítajú na základe metriky xor (výlučné alebo). Uzly si pamätajú informácie o častiach sieti, ktoré sa - podobne ako pri Chord - exponenciálne zväčšujú. Pre každú takúto časť si uzol pamätá nie jeden, ale zoznam kontaktov. Z toho vyplýva možnosť výberu počas routovania podľa stavu siete, čo umožní napr. minimalizovať latenciu. Zároveň existuje viac ciest k cieľu, preto je sieť viac odolná voči výpadkom a útočníkom. Uzol má možnosť poslať paralelné dotazy do časti, v ktorej je jeho cieľ. Opäť tým znížime odozvu, lebo nemusíme čakať na timeout vypadnutých uzlov. Počet týchto dotazov je nastaviteľný, čím sa vieme prispôsobiť rôznym požiadavkám. Routovacie zoznamy sú utriedené podľa času posledného kontaktu a zároveň sa preferujú uzly, ktoré sú v sieti dlhšie. Zaručí to väčšiu dostupnosť, lebo - ako štatistiky ukázali - čím dlhšie sa uzol nachádza v sieti, tým je pravdepodobnejšie, že ostane ďalšiu hodinu.

Funkcia xor je symetrická, čo umožní, aby uzol dostával dotazy od uzlov, ktorých má vo svojej routovacej tabuľke. Preto bude môcť na základe správ ktoré dostáva udržiavať svoju tabuľku aktuálnu. Zo symetrie ešte vyplýva, že uzly môžu poslať dotazy ľubovoľnému uzlu z cieľového intervalu. Ak sa použije asymetrická funkcia, tak sa cieľu môžme približovať len z jednej strany. Na kruhu Chord to znamená, že ľubovoľný uzol je príliš ďaleko od jeho predchodcov. Pri použítí xor je každý uzol z cieľového intervalu rovnako dobrým kandidátom na poslanie dotazu.

2.2.2 Štruktúra

Kademlia priraďuje uzlom a kľúčom 160-bitové identifikátory. Keďže štruktúra logickej siete a metóda rozdelenia adresného priestoru sú ovplyvnené výberom funkcie vzdialenosti, najprv si definujeme použítú metriku a ukážeme

jej vlastnosti. Vzdialenosť dvoch uzlov je bitový xor (bitová non-ekvivalencia) ich ID, čo budeme označovať ako $x \oplus y$. Je dobré si všimnúť, že xor je dobrá metrika, lebo spĺňa nasledujúce podmienky (pre všetky x, y, z):

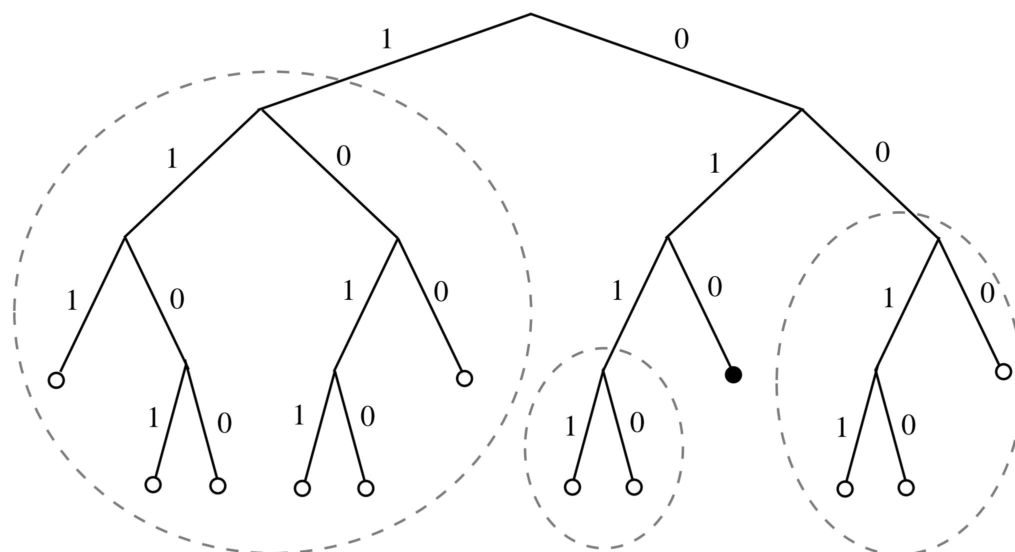
1. Nezápornosť: $x \oplus y \geq 0$
2. Totožnosť: $x \oplus y = 0 \iff x = y$
3. Symetria: $x \oplus y = y \oplus x$
4. Trojuholníková nerovnosť: $(x \oplus y) + (y \oplus x) \geq (x \oplus z)$

Posledná vlastnosť vyplýva z toho, že $(x \oplus y) \oplus (y \oplus z) = x \oplus z$ a $\forall a, b \geq 0 : a + b \geq a \oplus b$. Xor je tiež jednosmerný, pre dané x a d existuje práve jedno y také, že $x \oplus y = d$.

Logickú sieť si môžme predstaviť, ako binárny lexikografický strom nad abecedou $\{0, 1\}$, v ktorom listy reprezentujú uzly v sieti. Adresa uzla je jeho najdlhší jedinečný prefix. Ak na úplný binárny strom použijeme vyššie definovanú metriku, tak si všimneme, že vzdialenosť uzlov je úmerná výške najnižšieho podstromu do ktorého patria oba uzly. Inak povedané: vzdialenosť dvoch uzlov je rádovo počet bitov, ktoré nie sú v najdlhšom spoločnom prefixe.

Z pohľadu konkrétneho uzla je sieť rozdelená na časti podľa dĺžky najdlhšieho spoločného prefixu. Na binárnom strome to znamená, že uzol vidí sieť ako sériu podstromov, ktorých výška je maximálna a zároveň ten uzol tam nepatrí. Veľkosť týchto skupín rastie exponenciálne podľa vzdialenosti od uzla. Routovanie je založené na princípe, že každý uzol pozná aspoň jeden uzol zo všetkých podstromov (ak ten podstrom obsahuje nejaký uzol). Tá vlastnosť zaručí, aby vedel každý uzol nájsť ľubovoľný iný, na základe jeho ID. Na obrázku 2.3 je znázornení príklad rozdelenia logickej siete z pohľadu konkrétneho uzla. Označené časti sú podstromy, do ktorých musí mať uzol s jedinečným prefixom 010 kontakt. Rozdelenie adresného priestoru sa robí podľa metriky xor, každý uzol je zodpovedný za kľúče, ktoré sú k nemu najbližšie.

Jedna z dobrých vlastností protokolu Kademlia je schopnosť získavať routovacie informácie z obyčajných správ. Takto si vie udržiavať každý uzol svoju routovaciu tabuľku aktuálnu, bez potreby posielania špeciálnych dotazov. Každá správa obsahuje ID odosielateľa a tak si vie prijímateľ zaznamenať uzly, ktoré ho skontaktovali. Odosielateľ bude mať prehľad o stave siete a tak bude môcť s väčšou pravdepodobnosťou posielať správy živým uzlom.



Obr. 2.3: Rozdelenie siete

2.2.3 Routovacia tabuľka

Routovacia tabuľka obsahuje informácie o častiach siete, ktoré sme si vyššie popísali. Daná položka tabuľky na uzle u obsahuje zoznam uzlov, ktoré zdieľajú prefix istej dĺžky s u. Formálnejšie povedané, i-ty záznam v tabuľke obsahuje maximálne k uzlov, ktorých vzdialenosť k u je 2^i až 2^{i+1}. Tieto zoznamy voláme k-bucket. Kontakty v tabuľke sú reprezentované trojicou (IP adresa, port, ID). Veľkosť k-bucketov je nastaviteľná parametrom k. Zoznamy sú utriedené podľa času posledného kontaktu a aktualizujú sa každou prijatou správou.

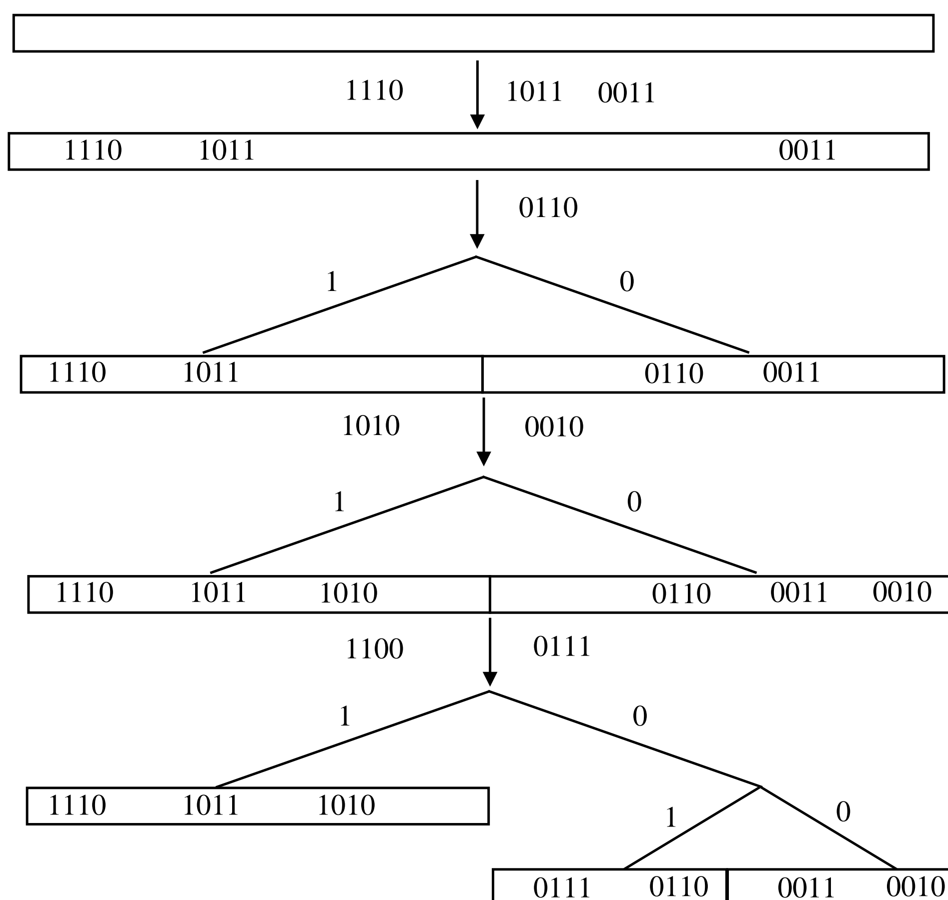
Z voľby metriky a štruktúry routovacej tabuľky vyplýva niekoľko vlastností. Každý uzol pozná lepšie svoje blízke okolie, ale má kontakt do každej časti siete. To zaručí, aby uzol vedel poslať správu ľubovoľnému inému. Xor je symetrická funkcia, preto ľubovoľný uzol z cieľového intervalu je dosť blízko k nemu. To je rozdiel oproti protokolu Chord, kde bolo treba hľadať predchodcu adresy a preto sa mohlo približovať len z jednej strany. Kademlia má väčší výber ciest k cieľu. Keďže záznamy v tabuľke sú zoznamy, uzol má možnosť posielať dotazy paralelne viacerým uzlom. Za cenu väčšej záťaže siete tak získame rýchlejšie vyhľadávanie. Zároveň to poskytuje - do istej miery - ochranu proti útokom založených na falšovaní routovacích informácií.

Ukážeme si štruktúru routovacej tabuľky a spôsob pridávania kontaktov. Tabuľku si môžeme predstaviť ako binárny strom, v ktorom listy sú k-buckety. Každý k-bucket obsahuje zoznam uzlov s nejakým spoločným prefixom a ten prefix určuje pozíciu k-bucketu v strome. Strom sa vytvára dynamicky, uzly sa pridávajú postupne. Na začiatku obsahuje tabuľka uzla u len jeden k-bucket, ktorý zahŕňa celý adresný priestor (prázdny prefix). Uzol tam pridáva kontakty, kým je veľkosť zoznamu menšia ako k. Ak je k-bucket - do ktorého chceme pridať ďalší záznam - plný, tak sa rozhodne podla toho, či u patrí do intervalu toho k-bucketu. Ak áno tak sa k-bucket rozdelí na dve časti a nový kontakt sa pridá na príslušné miesto. Ak to je k-bucket z iného intervalu, tak sa nový kontakt zahodí.

Tento proces demonštruje obrázok 2.4 z pohľadu uzla 0100 z obrázku 2.3. Môžeme sledovať postupné pridávanie uzlov a následné rozdeľovanie k-bucketov. Je dôležité všimnúť si, že kým na predošlom príklade sme používali ako adresu najkratší jedinečný prefix, v tomto prípade používame celé ID. Táto sieť používa 4-bitové ID a maximálna veľkosť k-bucketu je 3. Vidíme, že sa rozdelia len tie k-buckety, do ktorých patrí aj uzol 0100. V poslednom kroku sa pridáva uzol 1100 do plného k-bucketu, ale ten sa nerozdelí, lebo jeho interval je mimo uzla 0100.

Kvôli optimalizácii je ešte žiadúce, aby sa ošetrili prípady príliž nevyvážených stromov. Zavádza sa nová podmienka, podľa ktorej si uzly pamätajú všetky kontakty v najmenšom podstrome (v ktorom sú aj oni) o veľkosti aspoň k. To zaistí lepšiu informovanosť uzlov o zmenách v sieti.

Aby sa k-buckety udržiavali v korektnom stave, uzol si ich aktualizuje každou prijatou správou. Z doručenej správy sa dozvie, ktorý uzol ju odoslal. Pozrie sa do príslušného k-bucketu, či tam už ten uzol je. Ak áno, tak ho presunie na začitok zoznamu. Ak tam ten uzol ešte nie je a zoznam nie je plný, tak ho pridá na začiatok. Ak je plný, tak overí či je posledný uzol v zozname - o ktorom počul najdávnejšie - stále aktívny, pomocou funkcie *PING* (viď Kapitola 2.2.4). Ak áno, tak ho presunie na začiatok zoznamu a nový kontakt zahodí. Ak nie, tak ho zo zoznamu vyberie a nový uzol dá na začiatok. Je to stratégia least recently seen, pri ktorom sa živé uzly nikdy nevyhadzujú. Voľba tejto možnosti vyplýva zo štatistiky, podľa ktorej staré uzly opustia sieť s menšou pravdepodobnosťou. Uprednostnenie starých uzlov zvyšuje robustnosť siete a triedenie podľa času posledného kontaktu zaistí výber aktívnych uzlov. Ďalšia výhoda je istá odolnosť voči útokom typu DoS (Denial of Service), lebo k-buckety sa nedajú rýchlo zahltiť novými uzlami. Nevýhoda tohto prístupu je väčšia záťaž siete, kvôli častému overovaniu do-



Obr. 2.4: Budovanie routovacej tabuľky

stupnosti uzlov. Rieši sa to pozdržaním dotazov *PING*, kým má uzol aj niečo iné poslať cieľu. Kým sa nerozhodne o osude nových kontaktov uzol si ich pamätá v *replacement cache* a ak sa uvolní miesto v príslušnom k-buckete, tak ho tam pridá hneď.

2.2.4 Funkcie protokolu

V tejto časti popíšeme funkčnosť protokolu Kademlia. Budeme riešiť uloženie a vyhľadanie údajov, pripájanie uzlov a aktualizáciu routovacích informácií. Funkcie poskytované protokolom Kademlia sú implementované ako RPC (Remote procedure call), čo znamená poslanie špecifického dotazu pre uzol. Sú

to nasledujúce funkcie:

- *PING* - overí, či je uzol pripojený
- *STORE* - uloží dvojicu (kľúč, hodnota) na uzle
- *FIND_NODE* - vráti k uzlov, ktorých pozná najbližšie k danej adrese
- *FIND_VALUE* - to isté ako *FIND_NODE*, ale ak uzol má dáta, tak vráti tie

Kľúčovým bodom protokolu je nájdenie uzla najbližšieho k danej adrese. Ak uzol u chce nájsť uzol najbližšie k adrese id, tak vykoná nasledujúcu rekurzívnu funkciu. Z k-bucketu do ktorého patrí id vyberie α uzlov a pošle im paralelné a asynchrónne dotazy *FIND_NODE*. V nasledujúcom kroku u pošle ďalšie dotazy uzlom, o ktorých sa dozvedel z predošlých dotazov. Ak už nedostane žiadne bližšie kontakty, pošle dotazy *FIND_NODE* k najbližším uzlom, ktoré spoznal a ešte im neposielal dotaz. Funkcia skončí, ak dostane odpoveď od každého z nich. Výsledok je k uzlov, ktoré sú najbližšie k id. V každom kroku posiela nanajvýš α dotazov, čo je nastaviteľný parameter (napr. 3). Keďže táto funkcia je základom fungovania dotazov, treba si uvedomiť jej časovú zložitosť. Predpokladáme, že každý k-bucket obsahuje záznam, ak sa v tom intervale nachádza uzol. Vyhľadávanie v jednom kroku nájde uzol, ktorého ID je o jeden bit bližšie k cieľu, čo znamená, že je o polovicu bližšie. Z toho vyplýva, že počet uzlov cez ktoré sa správa routuje je $O(\log n)$.

Uloženie dát začína vyhľadaním k uzlov, ktoré sú najbližšie k danému kľúču. Potom sa každému z nich pošle správa *STORE* s dvojicou (kľúč, hodnota). Okrem tejto replikácie sa ešte dáta periodicky republikujú (viď Kapitola 2.2.5), aby sa zaistila väčšia dostupnosť dát. Vyhľadávanie dát funguje analogicky ako vyhľadávanie uzlov, len sa namiesto dotazov *FIND_NODE* posielajú *FIND_VALUE*. Proces končí, ak uzol dostane požadovanú hodnotu. Výsledok sa cachuje na najbližšom uzle, ktorý nevrátil dáta, čo zaistí rýchlejšie vyhľadávanie v budúcnosti.

Ako sme v Kapitole 2.2.3 popísali, poslanie každej správy spôsobí aktualizáciu routovacej tabuľky. Môže však nastať situácia, keď sa do niektorej časti adresného priestoru neposiela dotaz a preto záznamy v prislúchajúcom k-buckete zastarajú. Aby sme tomu predišli, uzol dá vyhľadať náhodnú

adresu zo sekcie, odkiaľ dlhší čas nedostal správu. To zaistí aktualizáciu k-bucketu a pridanie iniciátora do tabuliek tých uzlov. Tento proces nazývame obnovenie k-bucketu.

Aby sa uzol mohol pripojiť do siete musí poznať niekoho, kto už do nej patrí. Vyberie si pre seba náhodné ID a svoj kontakt pridá do k-bucketu. Následne dá vyhľadať svoje ID, čo mu vráti jeho k najbližších susedov. Ako vedľajší účinok vyhľadania sa aj on pridá do tabuliek svojich susedov. Potom dá obnoviť svoje prázdne k-buckety, čo opäť spôsobí jednak naplnenie vlastnej tabuľky a informovanie zvyšku siete.

2.2.5 Perzistencia dát

Protokol Kademlia kvôli perzistencii dát robí rôzne opatrenia. Pri ukladaní pomocou funkcie *STORE* sa hodnota uloží na k uzloch. Uzly blízko seba z pohľadu siete Kademlia v ideálnom prípade nemajú nič spoločné v reálnom svete. Táto replikácia je efektívna, lebo výpadky a útoky obvykle zasiahnu počítače z jednej lokality. Cachovanie vyhľadávania tiež zvyšuje dostupnosť.

Treba zaistiť, aby odchádzanie a prichádzanie uzlov neovplyvnilo vyhľadávanie. Každý uzol periodicky znovu publikuje dáta, ktoré sú na ňom uložené a dlhší čas neboli publikované. Keďže uzly zodpovedné za daný kľúč sú blízko, stačí jednému z nich republikovať a ostatné sa o tom dozvedia. Protokol zaistí, aby uzly vedeli o svojom okolí, preto ak sa pripojí nový uzol jeho susedia vedia naň preniesť potrebné dáta. Aby sa nové uzly nezahltili informáciami, kontaktuje ho len ten uzol, ktorý je k nemu najbližšie.

Kapitola 3

Simulácie

Táto kapitola opisuje priebeh simulácií. Sekcia 3.1 podáva prehľad o použitom prostredí a nastaveniach. Sekcie 3.2 až 3.7 prezentujú výsledky meraní. Sekcia 3.8 uvádza skúsenosti získané počas testov a porovnanie dosiahnutých výsledkov s inými prácami.

3.1 Popis simulácie

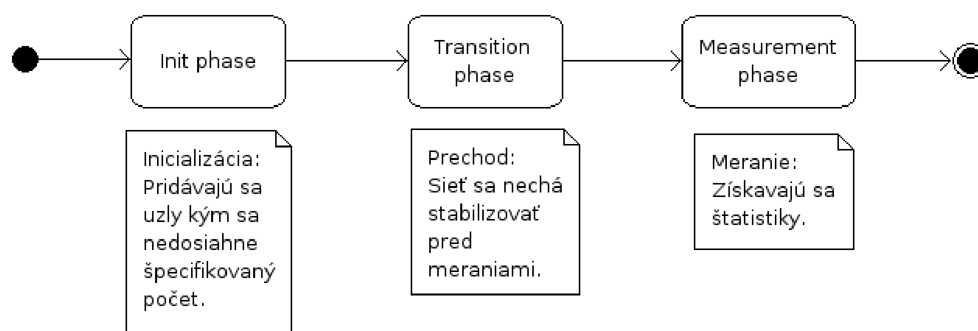
Simulácie sme realizovali v aplikácii OverSim [BHK07]. Je to open-source framework na simuláciu P2P sietí, v ktorom sú implementované protokoly Chord a Kademlia. Program ponúka širokú škálu nastavení a veľkú množinu štatistík. Parametre sa špecifikujú v vstupnom súbore *omnetpp.ini*. Výstupné súbory obsahujú dáta v skalárnej a vektorovej podobe, ktoré sa ďalej spracúvajú. Tabuľka 3.1 znázorňuje vybrané nastavenia s hodnotami, ktoré sme definovali ako predvolené pre simulácie. Pri konkrétnych meraniach je možné tieto hodnoty predefinovať. Použitý vstupný súbor so všetkými nastaveniami je v Dodatku A.

Beh aplikácie začína načítaním nastavení zo súboru. Samotná simulácia sa skladá z troch fáz (viď Obrázok 3.1). V inicializačnej fáze sa postupne pridávajú uzly do siete, kým sa nedosiahne počet *targetOverlayTerminalNum*. Potom sa začne prechodová fáza, ktorej dĺžka závisí od parametra *transitionTime* a slúži na stabilizáciu siete pred samotným meraním. Nakoniec sa spustí fáza merania, počas ktorej sa sieť simuluje a zbierajú sa štatistiky. Dĺžku poslednej fázy určuje parameter *measurementTime*.

Dôležitá je otázka simulácie fluktuácie uzlov, na čo OverSim ponúka viac

Parameter	Hodnota	Popis
overlayType	-	implementácia DHT
measurementTime	24h	dĺžka simulácie
targetOverlayTerminalNum	2000	počet uzlov po inicializácii
transitionTime	10min	dĺžka transition phase
churnGeneratorTypes	LifetimeChurn	typ generátora fluktuácie
lifetimeMean	5h	priemerná životnosť uzlov
routingType	-	typ routovania
numReplica	4	počet replík

Tabuľka 3.1: Najdôležitejšie nastavenia



Obr. 3.1: Priebeh merania

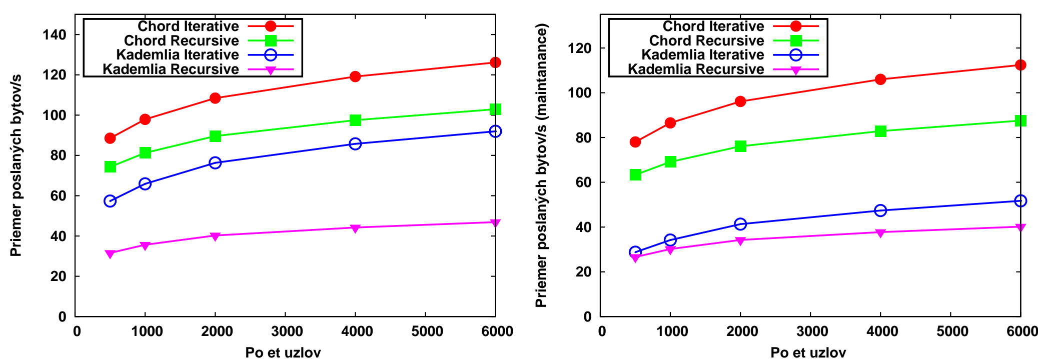
možností. V tejto práci sme použili generátor *LifetimeChurn*, ktorý funguje nasledovne: počas vytvorenia uzla sa náhodne vygeneruje jeho životnosť z pravdepodobnostnej funkcie. Po tom čase sa uzol odstráni. Nový uzol sa vytvorí po čase, ktorý sa vygeneruje z rovnakej funkcie. Priemerná životnosť sa nastavuje parametrom *lifetimeMean*.

Počas simulácií otestujeme aj rôzne typy routovania, ktoré sa nastavujú parametrom *routingType*. Prvá možnosť je iteratívne routovanie (nastavenie *iterative*), ktoré funguje nasledovne: iniciátor pošle dotaz najbližšiemu uzlu k cieľu z vlastnej routovacej tabuľky. Prijaté odpovede obsahujú bližšie uzly, ktorým iniciátor pošle v ďalšom kroku správy. Pri rekurzívnom routovaní (nastavenie *semi-recursive*) iniciátor pošle správu uzlu podobne ako pri iteratívnom, ale ten správu prepošle ďalej smerom k cieľu. Nakoniec hľadaný uzol pošle odpoveď iniciátorovi. Je dobré všimnúť si, že iteratívny prístup na nájdenie bližšieho uzla potrebuje dve správy a rekurzívny len jednu.

3.2 Bandwidth

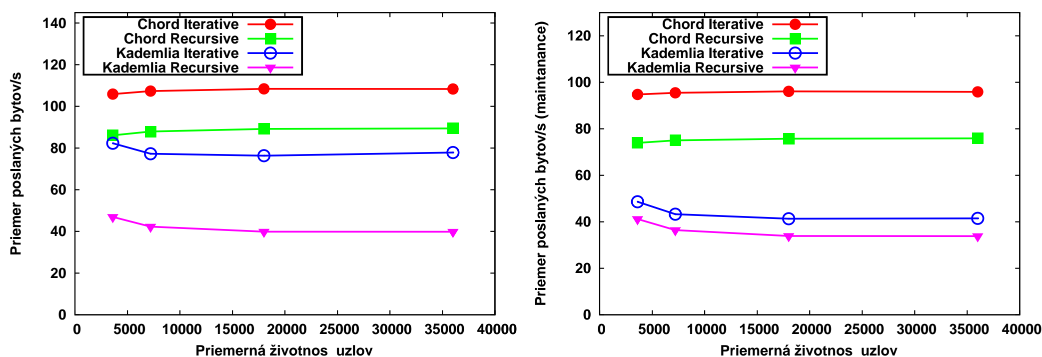
V prvých testoch sme skúmali vplyv veľkosti siete a životnosti uzlov na sieťovú komunikáciu. Merali sme zvlášť všetky správy a správy slúžiace na manažment siete (aktualizovanie routovacej tabuľky, pripojenie uzla atď). Počet uzlov v simuláciách sme zvolili na 500, 1000, 2000, 4000 a 6000. Priemerná životnosť uzlov nadobúdala nasledujúce hodnoty: 1h, 2h, 5h a 10h (menšia životnosť znamená väčšiu fluktuáciu).

Výsledky prezentované na obrázku 3.2 potvrdzujú teoretické výpočty: počet správ rastie logaritmicky od počtu uzlov (viď Kapitolu 3.7). Protokol Kademlia generuje väčšiu komunikáciu ako Chord. Iteratívny prístup routovania pre oba protokoly viac vyťažuje sieť. Tento výsledok vyplýva z toho, že - ako sme v Kapitole 3.1 ukázali - iteratívne routovanie potrebuje na každé priblíženie sa k cieľu dve správy, kým rekurzívne len jednu. Kademlia efektívne získava routovacie informácie z obyčajných správ a preto potrebuje len rádovo polovicu správ na manažovanie siete ako Chord.



Obr. 3.2: Vplyv veľkosti siete na vyťaženie siete

Obrázok 3.3 poukazuje na to, že životnosť uzlov ovplyvňuje počet správ len minimálne. Kademlia pošle viac správ pri väčšej fluktuácii a naopak Chord potrebuje viac správ pri väčšej životnosti.

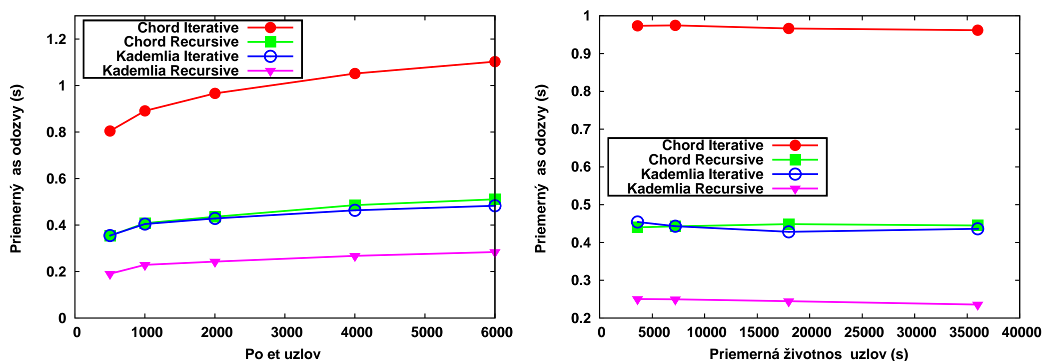


Obr. 3.3: Vplyv životnosti uzlov na vyťaženie siete

3.3 Čas odozvy

Výsledky simulácií zameraných na čas odozvy nám dávajú informácie o výkone algoritmov v praxi. Latenciu sme merali v sekundách a skúmali sme jej závislosť od počtu uzlov a veľkosti fluktuácie.

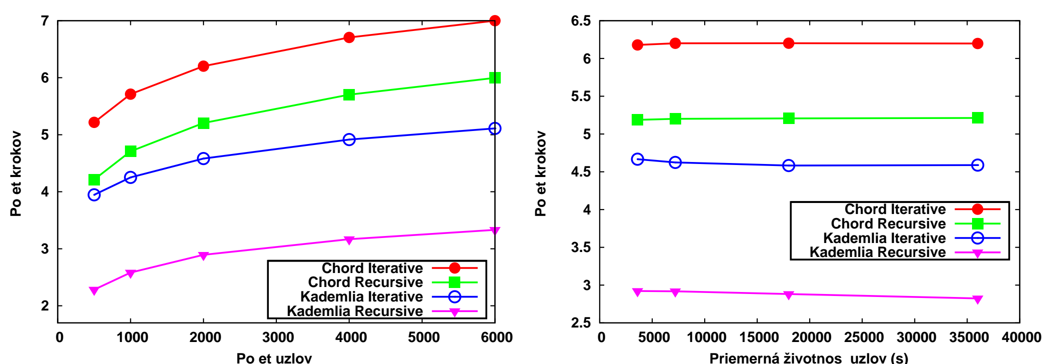
Obrázok 3.4 prezentuje výsledky testov. Protokol Kademlia potrebuje približne polovicu času na vyhľadávanie ako Chord. Rekurzívne routovanie je o polovicu rýchlejšie ako iteratívne, čo je spôsobené menším počtom správ. Výsledky tiež potvrdzujú, že čas odozvy sa škáluje logaritmicky (viď Kapitolu 3.7). Simulácie ukázali, že fluktuácia má len malý vplyv na oneskorenie správ. Pre všetky protokoly sa čas odozvy zmenšuje pri väčšej životnosti uzlov.



Obr. 3.4: Čas odozvy

3.4 Počet krokov pri routovaní

Počet krokov pri routovaní súvisí s odozvou, ale dáva nám informácie nezávislé od fyzickej topológie siete. Výsledky na obrázku 3.5 sú podobné ako výsledky časov odozvy. Kademlia potrebuje na vyhľadávanie menej krokov a rekurzívne routovanie má opäť lepšie výsledky ako iteratívne. Fluktuácia uzlov ovplyvňuje počet krokov minimálne.

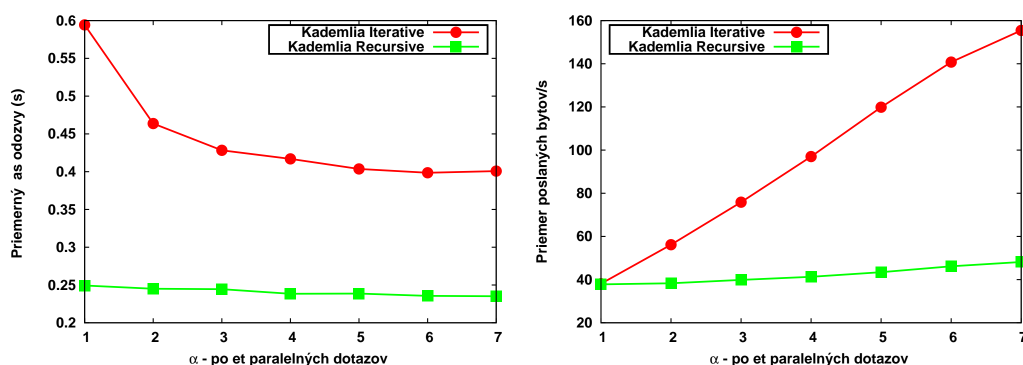


Obr. 3.5: Priemerný počet krokov

3.5 Paralelné dotazy

Protokol Kademlia umožňuje posielať paralelné, asynchrónne dotazy. Táto vlastnosť má zrýchliť routovanie, za cenu konštantného nárastu správ. Obrázok 3.6 ilustruje vplyv počtu paralelných dotazov na rýchlosť a vyťaženie siete. V simuláciach sme pre hodnotu parametra na nastavenie paralelizmu (α) skúšali interval $\langle 1, 7\rangle$.

Pri použití rekurzívneho routovania sme pozorovali len malé zmeny v správaní sa siete. Pri iteratívnom prístupe sme zaznamenali menšie časy odozvy pre väčšie hodnoty α. V praxi sa často používa nastavenie $\alpha = 3$. Vhodnosť tejto voľby potvrdzujú aj naše merania, keďže 3 paralelné dotazy spôsobia 30%-né zrýchlenie oproti jednému a väčšie hodnoty prinesú už len malé zlepšenie. Druhý graf ilustruje veľkosť sieťovej komunikácie. Pridanie jednej správy zapríčiní konštantný nárast vyťaženia.

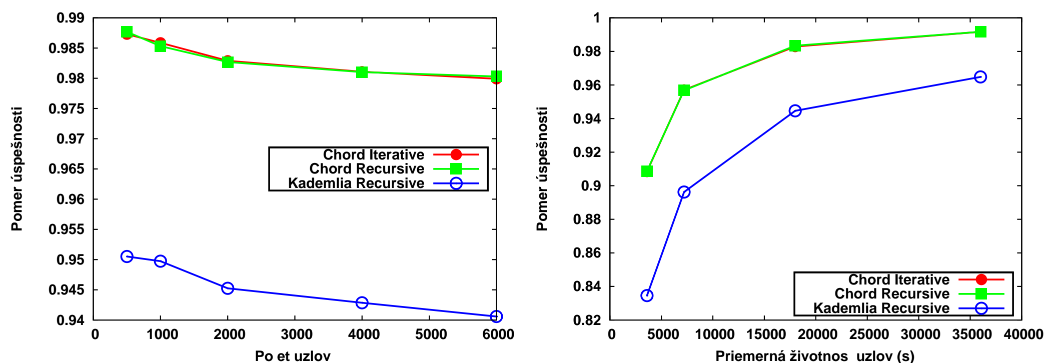


Obr. 3.6: Vplyv počtu paralelných správ

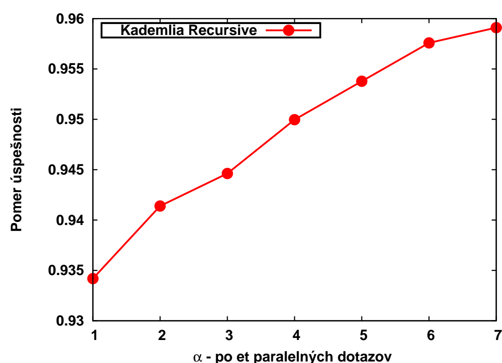
3.6 Úspešnosť vyhľadávania

Skúmali sme úspešnosť vyhľadávania v závislosti od počtu uzlov a od ich životnosti. Pre protokol Kademlia sme tiež testovali vplyv počtu paralelných správ. Obrázky 3.7 a 3.8 prezentujú výsledky. Protokol Kademlia pri použití iteratívneho routovania dosahoval úspešnosť skoro 100% pre všetky testy, preto sme tieto výsledky vynechali z grafov.

Úspešnosť vyhľadávania pre Chord ovplyvňuje voľba routovania len minimálne. Rekurzívna Kademlia má horšie výsledky ako Chord. Zvýšenie počtu uzlov v sieti spôsobuje pomalé klesanie úspešnosti. Úspešnosť najviac závisí od miery fluktuácie. Zvýšením životnosti sa zvýši aj úspešnosť. Pridávaním paralelných dotazov pre protokol Kademlia rastie úspešnost.

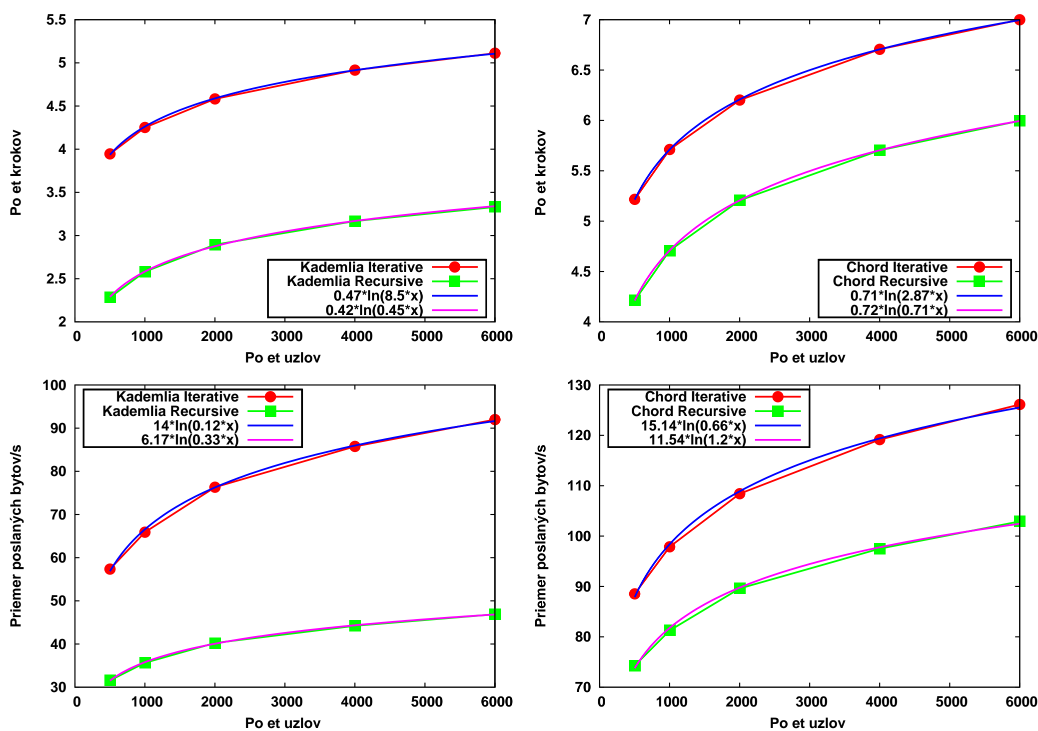


Obr. 3.7: Úpešnost vyhľadávania (počet, životnosť)



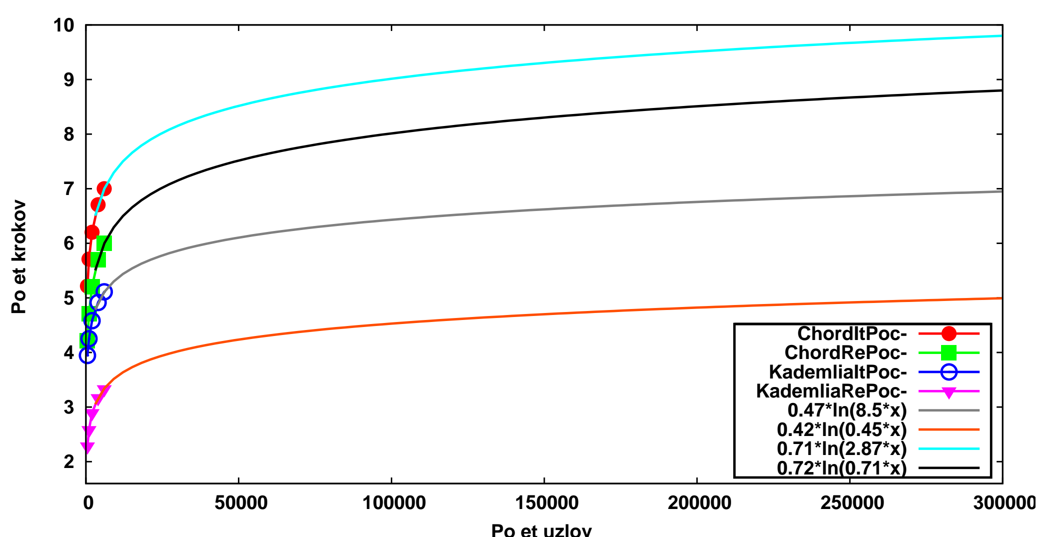
Obr. 3.8: Úspešnosť vyhľadávania (paralelné správy)

3.7 Function fit



Obr. 3.9: Škálovanie algoritmov

Jedným z cieľov simulácií bolo porovnať asymptotické odhady pre dĺžku routovania a počet správ s reálnymi hodnotami nameranými v praxi. Teoretické výpočty ukázali, že oba algoritmy pre spomenuté problémy majú zložitosť $O(\log n)$. Výsledky prezentované na obrázkoch 3.9. potvrdzujú tieto odhady. Na obrázku 3.10 sú načrtnuté odhady grafov pre väčšie siete. Tie nám dávajú informáciu o dobrej škálovateľnosti algoritmov aj pre státisíce uzlov.



Obr. 3.10: Odhad počtu krokov pre veľké siete

3.8 Skúsenosti a porovnanie

Cieľom simulácií bolo otestovať najdôležitejšie vlastnosti protokolov pre DHT. Zvolené prostredie bolo OverSim, ktoré vďaka širokej škále nastavení umožnilo pripraviť rôzne testy. Jediná nevýhoda programu vyplýva práve z vysokej konfigurovateľnosti, je náročné za obmedzený čas vyskúšať všetky možnosti. Dokumentácia projektu je síce použiteľná, ale nedefinuje presne všetky nastavenia a výsledné štatistiky.

Predchádzajúce štúdie sa venovali tejto problematike. V [Sib09] skúmali vhodnosť DHT pre mobilné siete. Výsledky škálovania a úspešnosti sú porovnateľné s výsledkami tejto práce. Medzi hlavné rozdiely v nastaveniach patrí dĺžka simulácií, v [Sib09] to je 14 dní, v tejto práci 24 hodín. Kvôli kratším simuláciam sme zvolili aj menšiu životnosť uzlov (5 hodín oproti 19,2 hodín). Naša snaha bola simulovať nastavenia algoritmov z praxe, preto sme použili replikáciu na štyroch uzloch a paralelné dotazy pre Kademlia. Spomínaná práca zakázala replikáciu a paralelizmus. Rozdiely medzi iteratívnym a rekurzívnym routovaním v tejto práci sú spôsobené použitím nastavenia *semi-recursive*, kým autori práce použili *recursive*. Naša voľba potrebuje menej správ a preto má rýchlejšie vyhľadávanie.

Záver

Cieľom práce bolo predstaviť princípy fungovania DHT. V úvode do problematiky sme popísali základné pojmy a kľúčové problémy, ktoré pri návrhu algoritmov treba riešiť. V Kapitole 2 sme predstavili dve konkrétne implementácie.

Protokol Chord sa vyznačuje jednoduchou štruktúrou a ľahkou implementáciou. Základom návrhu je topológia, v ktorej uzly sú usporiadané do kruhu. Predstavili sme triviálny prístup vyhľadávania, ktorý je pre prax kvôli nízkej efektivite nepoužiteľný. Venovali sme sa zlepšeniu výsledkov za použitia dodatočných routovacích informácií. Ďalší problém DHT je neustála fluktuácia uzlov. Uviedli sme agresívny prístup aktualizovania routovacích tabuliek hneď po zmenách v sieti. Táto voľba je však príliš striktná, preto sme sa ďalej venovali periodickej stabilizácii. Na záver sme spomenuli problém výpadku uzlov, ktorý sa rieši vytváraním replík a redundantnými routovacími informáciami.

Druhý prezentovaný protokol je Kademlia, ktorého hlavné výhody sú schopnosť získavať routovacie informácie z vyhľadávacích správ a možnosť paralelných dotazov. Novinka je tiež voľba operácie xor ako metrika logickej siete. Kľúčovým bodom protokolu je budovanie routovacej tabuľky, preto sme sa tomu problému podrobne venovali. Detailne sme popísali funkcie protokolu, ktoré používajú asynchrónne a paralelné dotazy. Spomenuli sme tiež problém replikácie a cacheovania.

V praktickej časti práce sme realizovali simulácie DHT. Cieľom bolo demonštrovať vlastnosti systémov a navzájom ich porovnať. Kapitola 3 prezentuje výsledky meraní, tu krátko zhrnieme hlavné výsledky. Podarilo sa nám ukázať, že oba protokoly sa škálujú dobre. Asymptotický odhad $O(\log n)$ pre prax znamená, že algoritmy DHT je možné nasadiť aj v sieťach s miliónmi uzlov. Zistili sme tiež, že vplyv fluktuácie na rýchlosť vyhľadávania je minimálny. Úspešnosť vyhľadávania je aj v nestálych sieťach akceptovateľná. Pre

protokol Kademlia sme skúmali výhody a nevýhody paralelného dotazovania. Pozorovali sme, že použitím len troch paralelných správ znížime čas odozvy o 30%. Usúdili sme, že pre dobre zvolené nastavenie je výhodné pre zníženie odozvy a zvýšenie úspešnosti povoliť väčšie vyťaženie siete.

Literatúra

[BHK07] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. OverSim: A Flexible Overlay Network Simulation Framework. In *Proceedings of 10th IEEE Global Internet Symposium (GI '07) in conjunction with IEEE INFOCOM 2007, Anchorage, AK, USA*, pages 79–84, May 2007.

[CW07] Scott A. Crosby and Dan S. Wallach. An analysis of bittorrents two kademlia-based dhts, 2007.

[HWW02] Steven Hazel, Brandon Wiley, and On Wiley. Achord: A variant of the chord lookup service for use in censorship resistant peer-to-peer publishing systems. In *In Proc. of the 1st International Peer To Peer Systems Workshop (IPTPS02)*, 2002.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, New York, NY, USA, 1997. ACM.

[Loe08] Andrew Loewenstern. Dht protocol, January 2008. `http://www.bittorrent.org/beps/bep_0005.html`.

[MM02] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65, London, UK, 2002. Springer-Verlag.

[RKCD01] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The design of a large-scale event notification infrastructure. In *In Networked Group Communication*, pages 30–43, 2001.

[Sib09] Simo Sibakov. Simulating a mobile peer-to-peer network. Master's thesis, Helsinki University Of Technology, November 2009.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

Dodatok A

Vstupný súbor omnetpp.ini

[General]
network = oversim.underlay.simpleunderlay.SimpleUnderlayNetwork

[Config ChordItPoc]
description = Chord (iterative, SimpleUnderlayNetwork, lifetimechurn)
**.measurementTime = 86400s
**.transitionTime = 600s
**.overlayType = "oversim.overlay.chord.ChordModules"
**.tier1Type = "oversim.applications.kbrtestapp.KBRTestAppModules"
**.routingType = "iterative"
**.targetOverlayTerminalNum = ${poc=500,1000,2000,4000,6000}
**.initPhaseCreationInterval = 0.1s
**.debugOutput = false
**.churnGeneratorTypes = "oversim.common.LifetimeChurn"
**.lifetimeMean = 18000s

[Config KademliaItPoc]
description = Kademlia (iterative, SimpleUnderlayNetwork, lifetimechurn)
**.measurementTime = 86400s
**.transitionTime = 600s
**.overlayType = "oversim.overlay.kademlia.KademliaModules"
**.tier1Type = "oversim.applications.kbrtestapp.KBRTestAppModules"
**.routingType = "iterative"
**.targetOverlayTerminalNum = ${poc=500,1000,2000,4000,6000}
**.initPhaseCreationInterval = 0.1s

**.debugOutput = false
**.churnGeneratorTypes = "oversim.common.LifetimeChurn"
**.lifetimeMean = 18000s
**.overlay.kademlia.lookupParallelRpcs = 3

[**Config ChordRePoc**]
description = Chord (semi-recursive, SimpleUnderlayNetwork, lifetimechurn)
**.measurementTime = 86400s
**.transitionTime = 600s
**.overlayType = "oversim.overlay.chord.ChordModules"
**.tier1Type = "oversim.applications.kbrtestapp.KBRTestAppModules"
**.routingType = "semi-recursive"
**.targetOverlayTerminalNum = ${poc=500,1000,2000,4000,6000}
**.initPhaseCreationInterval = 0.1s
**.debugOutput = false
**.churnGeneratorTypes = "oversim.common.LifetimeChurn"
**.lifetimeMean = 18000s

[**Config KademliaRePoc**]
description = Kademlia (semi-recursive, SimpleUnderlayNetwork, lifetime-churn)
**.measurementTime = 86400s
**.transitionTime = 600s
**.overlayType = "oversim.overlay.kademlia.KademliaModules"
**.tier1Type = "oversim.applications.kbrtestapp.KBRTestAppModules"
**.routingType = "semi-recursive"
**.targetOverlayTerminalNum = ${poc=500,1000,2000,4000,6000}
**.initPhaseCreationInterval = 0.1s
**.debugOutput = false
**.churnGeneratorTypes = "oversim.common.LifetimeChurn"
**.lifetimeMean = 18000s
**.overlay.kademlia.lookupParallelRpcs = 3

[**Config ChordItChurn**]
description = Chord (iterative, SimpleUnderlayNetwork, lifetimechurn)
**.measurementTime = 86400s
**.transitionTime = 600s
**.overlayType = "oversim.overlay.chord.ChordModules"

**.tier1Type = "oversim.applications.kbrtestapp.KBRTestAppModules"
**.routingType = "iterative"
**.targetOverlayTerminalNum = 2000
**.initPhaseCreationInterval = 0.1s
**.debugOutput = false
**.churnGeneratorTypes = "oversim.common.LifetimeChurn"
**.lifetimeMean = ${lif=3600s,7200s,18000s,36000s}

[**Config KademliaItChurn**]
description = Kademlia (iterative, SimpleUnderlayNetwork, lifetimechurn)
**.measurementTime = 86400s
**.transitionTime = 600s
**.overlayType = "oversim.overlay.kademlia.KademliaModules"
**.tier1Type = "oversim.applications.kbrtestapp.KBRTestAppModules"
**.routingType = "iterative"
**.targetOverlayTerminalNum = 2000
**.initPhaseCreationInterval = 0.1s
**.debugOutput = false
**.churnGeneratorTypes = "oversim.common.LifetimeChurn"
**.lifetimeMean = ${lif=3600s,7200s,18000s,36000s}
**.overlay.kademlia.lookupParallelRpcs = 3

[**Config ChordReChurn**]
description = Chord (semi-recursive, SimpleUnderlayNetwork, lifetimechurn)
**.measurementTime = 86400s
**.transitionTime = 600s
**.overlayType = "oversim.overlay.chord.ChordModules"
**.tier1Type = "oversim.applications.kbrtestapp.KBRTestAppModules"
**.routingType = "semi-recursive"
**.targetOverlayTerminalNum = 2000
**.initPhaseCreationInterval = 0.1s
**.debugOutput = false
**.churnGeneratorTypes = "oversim.common.LifetimeChurn"
**.lifetimeMean = ${lif=3600s,7200s,18000s,36000s}

[**Config KademliaReChurn**]
description = kademlia (semi-recursive, simpleunderlaynetwork, lifetimechurn)
**.measurementTime = 86400s

**.transitionTime = 600s
**.overlayType = "oversim.overlay.kademlia.KademliaModules"
**.tier1Type = "oversim.applications.kbrtestapp.KBRTestAppModules"
**.routingType = "semi-recursive"
**.targetOverlayTerminalNum = 2000
**.initPhaseCreationInterval = 0.1s
**.debugOutput = false
**.churnGeneratorTypes = "oversim.common.LifetimeChurn"
**.lifetimeMean = ${lif=3600s,7200s,18000s,36000s}
**.overlay.kademlia.lookupParallelRpcs = 3

[**Config KademliaRePar**]
description = kademlia (semi-recursive, simpleunderlaynetwork, lifetimechurn)
**.measurementTime = 86400s
**.transitionTime = 600s
**.overlayType = "oversim.overlay.kademlia.KademliaModules"
**.tier1Type = "oversim.applications.kbrtestapp.KBRTestAppModules"
**.routingType = "semi-recursive"
**.targetOverlayTerminalNum = 2000
**.initPhaseCreationInterval = 0.1s
**.debugOutput = false
**.churnGeneratorTypes = "oversim.common.LifetimeChurn"
**.lifetimeMean = 18000s
**.overlay.kademlia.lookupParallelRpcs = ${par=1,2,3,4,5,6,7}

[**Config KademliaItPar**]
description = kademlia (semi-recursive, simpleunderlaynetwork, lifetimechurn)
**.measurementTime = 86400s
**.transitiontime = 600s
**.overlayType = "oversim.overlay.kademlia.KademliaModules"
**.tier1Type = "oversim.applications.kbrtestapp.KBRTestAppModules"
**.routingType = "iterative"
**.targetOverlayTerminalNum = 2000
**.initPhaseCreationInterval = 0.1s
**.debugOutput = false
**.churnGeneratorTypes = "oversim.common.LifetimeChurn"
**.lifetimeMean = 18000s
**.overlay.kademlia.lookupParallelRpcs = ${par=1,2,3,4,5,6,7}

include ./default.ini