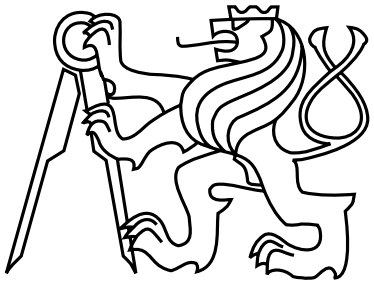
České vysoké učení technické v Praze
Fakulta elektrotechnická
Otvorená informatika



Diplomová práca

Veľkoobjemové úložisko emailov

Bc. Patrik Lenárt

Vedúci práce: Ing. Ján Šedivý, CSc.

Študijný program: Otvorená informatika, Magisterský

Odbor: Softwarové inžinierstvo

5. mája 2011

Poďakovanie

Prehlásenie

Prehlasujem, že som svoju diplomovú prácu vypracoval samostatne a použil som iba podklady uvedené v priloženom zozname.
Nemám závažný dôvod proti užitiu tohto školského diela v zmysle §60 Zákona č. 121/2000 Sb., o autorskom práve, o právach súvisiacich s právom autorským a o zmene niktorých zákonov (autorský zákon).

V Prahe dňa 1. 3. 2011 .....

Abstract

Abstrakt

Obsah

Zoznam obrázkov

Zoznam tabuliek

Kapitola 1

Úvod

S neustálym rozvojom informačných technológií súčasne narastá objem informácií, ktoré je potrebné spracúvať. Tento fakt podnietil vznik databázových systémov, ktoré slúžia na organizáciu, uchovávanie a prácu s veľkým objemom dát. V dnešnej dobe existuje množstvo databázových systémov, ktoré sa navzájom líšia svojou architektúrou, dátovým modelom, výrobcom atď.

Od začiatku sedemdesiatych rokov 20. storočia sú v tejto oblasti dominantou relačné databázové systémy (Relational Database Management Systems). Vďaka neustálemu prudkému rozvoju internetových technológií a rapídnemu rastu dát v digitálnom univerze [10] začínajú byť tieto systémy nepostačujúce. Medzi hlavné faktory pre výber relačného databázového systému doposiaľ patrili výrobca, cena a pod. V dnešnej dobe so vznikom moderných aplikácií (napríklad sociálne siete, dátové sklady, analytické aplikácie a iné), požadujeme od týchto systémov vlastnosti ako vysoká dostupnosť, horizotnálna rozšíriteľnosť a schopnosť pracovať s obrovským objemom dát (petabyte). Novo vznikajúce databázové systémy, spĺňajúce tieto požiadavky sa spoločne označujú pod názvom NoSQL (Not Only SQL). Pri ich výbere je v tomto prípade dôležité porozumenie architektúry, dátového modelu a dát, s ktorými budú tieto systémy pracovať.

Táto práca si kladie za cieľ viacero úloh, ktorými sú pochopenie a popis základných konceptov, ktoré tieto systémy využívajú, určenie kritérií vďaka ktorým môžeme tieto systémy navzájom porovnávať. Ďalej je úlohou analýzovať a popísať požiadavky pre systém veľkoobjemového úložiska elektronickej pošty, ktorý bude schopný spracovávať milióny emailov. Poslednou úlohou je na základe našich požiadavkov vybrať, čo najlepšie odpovedajúci NoSQL systém a s jeho použitím implementovať prototyp aplikácie.

1.1 Osnova

...

Kapitola 2

Databázové systémy

V tejto časti stručne popíšeme históriu vzniku databázových systémov, základné problémy pri tvorbe distribuovaných relačných databázových systémov a uvedieme možné spôsoby ich riešenia. Ďalej popíšeme základné koncepty, ktoré sa využívajú pri tvorbe distribuovaných databázových systémov a techniku MapReduce, ktorá slúži na prácu s veľkým objemom dát uloženým v systémoch NoSQL.

2.1 História

V polovici šesťdesiatych rokov 20. storočia bol spoločnosťou IBM vytvorený informačný systém IMS (Information Management System), využívajúci hierarchichký databázový model. IMS je po rokoch vývoja využívaný dodnes. Po krátkej dobe, v roku 1970, publikoval zamestnanec IBM, Dr. Edger F. Codd článok pod názvom „A Relational Model of Data for Large Shared Data Banks“ [6], ktorým uviedol relačný databázový model. Prvým databázovým systémom, ktorý tento model implementoval bol System R od IBM. Tento systém používal jazyk pod názvom SEQUEL, ktorý je predchodca dnešného SQL (Structured Query Language) slúžiacého na manipuláciu a definícu dát v relačných databázových systémoch. Tento koncept sa stal základom pre relačné databázové systémy, ktoré vďaka širokej škále vlastností ako napríklad podpora tranzakcií, dotazovací jazyk SQL, patria v dnešnej dobe medzi najpoužívanejšie riešenia na trhu.

V minulosti boli objem dát, s ktorým tieto systémy pracovali a výkon hardvéru mnohonásobne nižšie. Dnes napriek tomu, že výkon procesorov a veľkosť pamäťových zariadení rapídne stúpa, je najväčšou slabinou počítačových systémov rýchlosť prenosu dát medzi pevným diskom a hlavnou pamäťou. Ako príklad si vezmime bežnú konfiguráciu počítačového systému, ktorá obsahuje pevný disk o veľkosti 2TB a operačnú pamäť veľkosti 64Gb. Napriek týmto vysokým kapacitám tento systém bohužial dokáže v daný moment spracúvať maximálne 64Gb dát, čo je zlomok veľkosti v porovnaní s kapacitou pevného disku. Vznik nových webových aplikácií napr. sociálne siete, zavádzanie cloud computingu vyžadujú od systémov podporu škálovania, ktorá zabezpečuje vysokú dostupnosť, spoľahlivosť a ich nároky na spracovávaný objem dát sa neustále zvyšujú. Tieto nové požiadavky efektívne riešia distribuované systémy pod spoločným názvom NoSQL, ktoré popisuje následujúca kapitola.

2.2 ACID

Relačné databázové systémy poskytujú veľkú množinu operácií, ktoré sa vykonávajú nad ich dátami. Tranzakcie [7][8] sú zodpovedné za korektné vykonanie operácií v prípade, že spĺňajú množinu vlastností ACID. Význam jednotlivých vlastností akronýmu ACID je následovný:

- Atomicita (Atomicity) - zaisťuje, že sa daná tranzakcia vykoná celá, čo spôsobí korektný prechod systému do nového stavu. V prípade zlyhania tranzakcie nemá daná operácia žiaden vplyv na výsledný stav systému a prechod do nového stavu sa nevykoná.
- Konzistencia (Consistency) - každá tranzakcia po svojom úspešnom ukončení garantuje korektnosť svojho výsledku a zabezpečí, že systém prejde z jedného konzistentného stavu do druhého. Pojem konzistentný stav zaručuje, že dáta v systéme odpovedajú požadovanej hodnote. Systém sa musí nachádzať v konzistentnom stave aj v prípade zlyhania tranzakcie.
- Izolácia (Isolation) - operácie, ktoré prebiehajú počas vykonávania jednej tranzakcie nie sú viditeľné pre ostatné. Každá tranzakcia musí mať konzistentný prístup k dátam a to aj v prípade, že u inej tranzakcii dôjde k jej zlyhaniu.
- Trvácnosť (Durability) - v prípade, že bola tranzakcia úspešne ukončená, systém musí garantovať trvácnosť jej výsledku aj v prípade jeho zlyhania.

Implementácia vlastností ACID, ktoré zaručujú konzistenciu, zvyčajne využíva u relačných databázových systémoch metódu zamykania. Tranzakcia uzamkne dáta pred ich spracovaním a spôsobí ich nedostupnosť až do jej úspešného ukončenia, poprípade zlyhania. Pre databázový systém, od ktorého požadujeme vysokú dostupnosť alebo prácu pod zvýšenou záťažou, tento model nie je vyhovujúci. Zámky spôsobujú stavy, kedy ostatné operácie musia čakať na ich uvoľnenie. Jeho náhradou je Multiversion concurrency control, ktorý využívajú aj niektoré NoSQL databázové systémy.

Tranzakcie splňujúce vlastnosti ACID využívajú v distribuovaných databázových systémoch [1] dvojfázový potvrdzovací protokol (two-phase commit protocol). Distribuovaný databázový systém využívajúci tento protokol, ktorého tranzakcie splňujú vlasnosť ACID zaručuje konzistentnosť a je schopný odolávať čiastočným poruchám na sieti alebo v prípade čiastočnej poruchy systému. Vlastnosti ACID nekladú žiadnu záruku na dostupnosť systému. Takéto systémy su vhodné pre Internetové tranzakcie, aplikácie využívajúce platby apod. Existuje množstvo aplikácií, u ktorých má dostupnosť prednosť pred konzistenciou. Pri tvorbe distribuovaných databázových systémov je preto potrebné upustiť z niektorých ACID vlastností, čo spôsobilo vznik nového modelu pod názvom BASE.

2.3 Škálovanie databázového systému

Obecná definícia pojmu škálovateľnosť [3] je náročná bez vymedzenia kontextu, ku ktorému sa vzťahuje. V tejto práci budeme škálovateľnosť chápať v kontexte webových aplikácií,

[1] Distribuované databázové systémy sú tvorené pomocou viacerých samostatne operujúcich databázových systémov, ktoré nazývame uzly a môžu komunikovať pomocou sieti a užívateľovi alebo aplikácii sa javia ako jeden celok [ref].

ktorých dynamických vývoj kladie na databázové systémy viacero požiadavkov. Medzi hlavné patrí neustála potreba zvyšovania diskového priestoru a teda zvyšovanie veľkosti databáze alebo schopnosť obslúžiť čoraz vyšší počet užívateľov aplikácie (zvýšenie počtu operácií pre čítanie a zápis do databázového systému). V tomto prípade pod pojmom škálovatelnosť databázového systému rozumieme vlasnosť, vďaka ktorej je systém schopný spracúvať narastajúce požiadavky webovej aplikácie v definovanom čase intervale. Typicky pridaním nových systémov, čo spôsobuje vznik distribuovaného databázového systému.

Škálovateľnosť delíme na vertikálnu a horizontálnu. Táto metóda dodáva systému nasledujúce vlastnosti [5]:

- umožňuje zväčšiť veľkosť celkovej kapacity databáze a táto zmena by mala byť transparentná z pohľadu aplikácie na dáta.
- zvyšuje celkové množstvo operácií, pre čítanie a zápis dát, ktoré je systém schopný vykonať v danú časovú jednotku.
- v niektorých prípadoch môže zaručiť, že systém neobsahuje jednotku, ktorá by v prípade zlyhania spôsobila nedostupnosť celého systému (single point of failure).

Vertikálna škálovateľnosť je metóda, ktorá sa aplikuje pomocou zvýšovania výkonnosti hardvéru, tj. do systému sa pridáva operačná pamäť, rychlejšie viacjádrové procesory, zvyšuje sa kapacita diskov. Jednou z nevýhod tohoto riešenia je jeho vysoká cena a možná chvíľková nedostupnosť systému. Proces verktikálneho škálovania nad relačnou databázou obsahuje následujúce kroky:

- zámena hardvéru za výkonnejší
- úprava súborového systému (napr. zrušenie žurnálu)
- optimalizácia databázových dotazov, indexovanie
- pridanie vrsty pre kešovanie (memcached, EHCache, atď.)
- denormalizácia dát v databáze, porušenie normalizácie

V tomto prípade je možné naraziť na hranice Moorovho zákona [6] a na rad nastupuje horizontálna škálovateľnosť, ktorá je omnoho komplexnejšia. Horizontálnu škálovateľnosť je možné realizovať pomocou replikácie alebo metódou rozsekávania dát (sharding).

2.3.1 Replikácia

V distribuovaných systémoch sa pod pojmom replikácia rozumie vlastnosť, ktorá má za následok že sa daná informácia nachádza v konzistentnom stave na viacerých uzloch [2] tohoto systému. Táto vlastnosť zvyšuje dostupnosť, spoľahlivosť a odolnosť systému voči chybám.

V prípade distribuované databázového systému sa časť informácií uložených v databáze nachádza na viacerých uzloch. Táto vlasnosť môže napríklad zvýšiť výkonnosť operácií, ktoré

[2] Pod pojmom uzol v tomto prípade myslíme samostatný počitačový systém, ktorý je súčasťou distribuovaného systému

pristupujú k dátam a to tak, že dochádza k čítavaniu dát z databázy paralelne z viacerých uzlov. V systéme obsahujúcom repliku dát nedochádza k strate informácií v prípade poruchy uzlu. Replikácia a propagácia zmien v systéme sú z pohľadu aplikácie transparentné. Metóda replikácie nezvyšuje pridávaním nových uzlov celkovú kapacita databáze. Problémom tejto techniky je zápis dát, pri ktorom sa zmena musí prejaviť vo všetkých replikách. Existuje viacero metód pomocou, ktorých je možné zabezpečiť túto funkcionalitu:

- Read one - write all, u tejto metódy sa čítanie dát prevedie z ľubovolné uzlu obsahujúceho repliku. Zápis dát sa vykoná na všetky uzly obsahujúce repliku a v prípade, že každý z nich potvrdí úspech tejto operácie, zmena sa považuje úspešnú. Táto metóda nie je schopná pracovať v prípade, že dôjde k prerušeniu sieťového toku medzi uzlami (network partitioning) alebo v prípade poruchy uzlu.
- Quorum consensus - zápis na jeden uzol a následná asynchrónna propagácia repliky na ostatné uzly. Táto metóda je schopná zvládať stav pri ktorom dojde k prerušeniu sieťového toku alebo poruche uzlu. Implementácie využíva algoritmy pod názvom kôrum konsenzus (quorum consensus). ???

Výber metódy replikácie čiastočne popisuje dvojicu vlastností distribuovaného databázového systému a to dostupnosť a konzistenciu. Poďla teórie s názvom CAP (viď. nasledujúca kapitola) nie je možné aby systém disponoval súčasne vlastnosťami ako dostupnosť, konzistencia dát a schopnosť odolávať poruchám v prípade chyby v sieťovej komunikácii.

V relačných databázových systémoch sa replikácia rieši pomocou techniky Master-Slave. Uzol pod názvom master slúži ako jediný databázový stroj, na ktorom sa vykonáva zápis dát a replika týchto dát je následne distribuovaná na zvyšné uzly pod názvom slave. Touto metódou sme schopný mnohonásobne zvýšiť počet operácií, ktoré slúžia pre čítanie dát z databazového systému a v prípade zlyhania niektorého zo systémov máme neustále k dispozícii kópiu dát. Slabinou tejto techniky je uzol v roli master, ktorý znižuje výkonnosť v prípade operácií vykonavajúcich zápis a zároveň môže jeho porucha spôsobiť celkovú nedostupnosť systému.

Druhým riešením je technika Multi-master, kde každý uzol obsahujúci repliku je schopný zápisu dát a následne tieto preposiela zmeny ostatným. Tento mechanizmus predpokláda distribuovanú správu zamykania a vyžaduje algoritmy pre riešenie konfliktov spôsobujúcich nekonzistenciu dát.

2.3.2 Rozsekávanie dát (sharding)

Rozsekávanie dát je metóda založená na princípe, kde dáta obsiahnuté v databáze rozdeľujeme podľa stanovených pravidiel do menších celkov. Tieto celky môžeme následne umiestniť na navzájom rôzne uzly distribuovaného databázového systému. Táto metóda umožňuje zvýšiť výkonnosť operácií pre zápis a čítanie dát a zároveň pridávaním nových uzlov do systému sme schopný zvyšovať celkovú kapacitu databáze. V prípade, že architektúra distribuovaného databázového systému využíva túto techniku, zvýšenie výkonu jeho operácií a objemu uložených dát sa realizuje automaticky bez nutnosti zásahu do aplikácie.

Techniku rozsekávania môžeme považovať za architektúru známu pod názvom zdieľanie ničoho [12] (shared nothing). Táto architektúra sa používa pre návrh systémov využivajúcich multiprocesory. V takomto prípade sa medzi procesormi nezdieľa operačná ani disková

pamäť. Táto architektúra zabezpečuje takmer neobmedzenú škálovateľnosť systému a využíva ju mnoho NoSQL systémov ako napríklad Google Bigtable, Amazon Dynamo alebo technológia MapRreduce.

Pri návrhu distribuovaných databázových systémov, s využitím tejto techniky, patrí medzi kľúčový problém implementácia funkcie spojenia (join) nad dátami, ktorá sa radšej neimplementuje. V prípade, že sa dáta nad ktorými by sme chceli túto operáciu vykonať, nachádzajú na dvoch rozdielnych uzloch prepojených sieťou, takéto spojenie by značne znížilo celkovú výkonnosť systému a viedlo by k zvýšeniu sieťového toku a záťaži systémových zdrojov.

Keďže sa dáta nachádzajú na viacerých uzloch systému, hrozí zvýšená pravdepodobnosť hardverového zlyhania, poprípade prerušenie sieťového spojenia a preto sa táto technika často kombinuje s pomocou využitia replikácie.

V prípade použitia tejto techniky v relačných databázach, je nutný zásah do logiky aplikácie. Dáta uložené v tabuľkách relačnej databáze zachytávajú vzájomné relácie. Týmto spôsobom dochádza k celkovému narušeniu tohoto konceptu. Príkladom môže byť tabuľka obsahujúca zoznam zamestnancov, ktorú rozdelíme na samostatné celky. Každá tabuľka bude reprezentovať mená zamestnancov, ktorých priezvisko začína rovnakým písmenom abecedy a zároveň sa bude nachádzať na samostatnom databázovom systéme. Táto technika so sebou prináša problém, v ktorom je potrebné nájsť vhodný kľúč podľa, ktorého budeme dáta rozsekávať a zabezpečíme tak rovnomerné zaťaženie uzlov daného systému. Existuje viacero metód [7] a to:

- segmentácia dát poďla funkcionality - dáta, ktoré sme schopný popísať spoločnou vlastnosťou ukladáme do samostatných databáz a tieto umiestňujeme na rozdielné uzly systému. Príkladom môže byť samostatný uzol spravujúci databázu pre užívateľov a iný uzol s databázov pre produkty. Túto metódu spracoval Randy Shoup [3] [14], architekt spoločnosti eBay.

- rozsekávanie podľa kľúča - v datách hľadáme kľúč, podľa ktorého sme schopný ich rovnomernej distribúcie. Následne na tento kľúč aplikujeme hašovaciu funkciu a na základe je výsledku tieto dáta umiestňujeme na jednotlivé uzly.

- vyhľadávacia tabuľka - jeden uzlol v systéme slúži ako katalóg, ktorý určuje na ktorom uzle sa nachádzajú dané dáta. Tento uzol zároveň spôsobuje zníženie výkonu a v prípade jeho havárie spôsobuje nedostupnosť celého systému.

Replikácia a rozsekávanie dát patria medzi kľúčové vlastnosti využívané v NoSQL systémoch.

2.4 BASE

Akroným BASE [2] bol prykrát použitý v roku 1997 na sympóziu SOSP (ACM Symposium on Operating Systems Principles). BASE tvoria následujúce slovné spojenia:

[3] "If you can't split, you cant scale it." – Randy Shoup, Distinguished architect Ebay

- bežne dostupný (Basically Available) - systém je schopný zvládať jeho čiastočné zlyhanie za cenu nižšej komplexity.
- zmiernený stav (Soft State) - systém nezaručuje trvácnosť dát za cieľom zvýšenia výkonu.
- čiastočná konzistencia (Eventually Consistent) - je možné na určitú dobu tolerovať nekorektnosť dát, ktoré musia byť po určitom časovom intervale konzistentné.

Tento model poľavil na požiadavku zodpovednom za konzistenciu dát, s tým že dosiahol vyššiu dostupnosť v distribuovanom databázovom systéme aj v prípade čiastočného zlyhania. Prakticky môžeme každý systém klasifikovať ako systém spĺňajúci vlasnosti ACID alebo BASE.

BASE umožnuje horizontálne škálovanie relačných databázových systémov bez nutnosti použitia distribuovaných tranzakcií. Použitie tejto metódy je možné vďaka rozsekávaniu dát s využitím metódy segmentácie dát podľa funkcionality, ďalej sa využívajú perzistentné fronty a princípy událosťami riadenej architektúry (event driven architekture) [13]. Poľavením na požiadavku konzistencie dát sa v tomto prípade myslí stav, že dáta budú konzistentné po uplynutí určitého časového intervalu.

Systém obsahujúci čiastočnú konzistenciu dát je bankomatový systém. Po vybraní určitej čiastky z účtu, sa korektná informácia o aktuálnom zastatku účtu môže zobraziť až za niekoľko dní, kdežto tranzakcia ktorá túto zmenu vykonala musí spĺňať vlasnosti ACID. Medzi podobné webové aplikácie, u ktorých sa nepožadujú všetky vlasnosti ACID patria nákupný košík spoločnosti Amazon, zobrazovanie časovej osi aplikácií Twiter poprípade indexy Googlu. Ich nedostupnosť by znamenala obrovské finančné straty, napríkad v prípade, zlyhanie vyhľadávania pomocou Googlu by znamenalo zobrazenie nižšieho počtu reklám, nedstupnosť nákupného košíka v Amazone by spôsobila pokles predaja atď.

Aplikácia vyššie popísaných techník na relačné databázové systémy môže byť netrivialnou úlohou. Relačný model, je spôsob reprezentácie dát, ktorý umožnuje efektívne riešit určité typy problémov, preto snaha prispôsobiť tento model každému problému je nezmyselná. V tomto prípade, musíme uvažovať alternatívne riešenia, medzi ktoré patria systémy NoSQL.

2.5 CAP

Moderné webové aplikácie kladú na systémy požiadavky, medzi ktoré patrí vysoká dostupnosť, konzistencia dát a schopnosť odolávať chybám [4]. Dr. Brewerer v roku 2000 nastolil myšlienku dnes známu pod názvom teória CAP [4], ktorá tvrdí že je možné súčasne dosiahnúť len dvojicu z týchto vlastností. V roku 2002 platnosť tejto teórie pre asynchrónnu sieť matematicky dokázali Lynch a Gilbert [?]. Modelu asynchrónnej sieťe odpovedá svojimi vlasnosťami sieť Internet. Akroným CAP tvoria následujúce vlasnosti:

- Konzistencia (Consistency) - distribuovaný systém je v konzistentnom stave ak sa zmena aplikuje na všetky relevantné uzly systému v rovnakom čase.

[4] Chybou v tomto kontexte myslíme, prerušenie sieťovej komunikácie medzi uzlami daného systému, poprípade hardverová porucha uzlu???

- Dostupnosť (Availability) - distribuovaný systém je dostupný ak každý jeho uzol, ktorý pracuje korektne je schopný pri prijatí požiadavku zaslať odpoveď. V spojení s toleranciou chýb, tato vlastnosť hovorí, že prípade ak nastane sieťový problém [5] každý požiadavok bude vykonaný.

- Tolerancia chýb (Partition Tolerance) - uzly distribuovaného systému navzájom komunikujú pomocou siete, v ktorej hrozí stráta správ. V prípade vzniku sieťovej poruchy dané uzly medzi sebou navzájom nedokážu komunikovať. Táto vlasnosť podľa definície viď. Gilbert a Lynch tvrdí, že v prípade vzniku zlyhania sieťovej komunikácie medzi niektorými uzlami musí byť systém schopný naďalej pracovať korektne. Neexistuje reálny distribuovaný systém, ktorého uzly na vzájomnú komunikáciu využívajú sieť a nedochádza pri tom k stráte správ, teda k poruchám sieťovej komunikácie.

Pravdepodobnosť, že dôjde k zlyhaniu ľubovoľného uzlu v distribuovanom systéme exponenciálne narastá s počtom pribúdajúcich uzlov.

$$P(ubovonhozlyhania) = 1 - P(individulnyuzolnezlyh)^{poetuzlov}$$

2.5.1 Konzistencia verzus dostupnosť

V distribuovanom systéme nie je možné zaručiť súčasne vlasnosť konzistencie a dostupnosti. Ako príklad si predstavme distribuovaný systém, ktorý zaručuje obe vlasnosti aj v prípade sieťového prerušenia. Tento systém obsahuje tri uzly A,B,C, na ktorých sa nachádzajú identické (replikované) dáta. Ďalej uvažujme, že došlo k sieťovému prerušeniu, ktoré rozdelilo uzly na dva samostatné celky A,B a C. V prípade, že uzol C obdrži požiadavok pre zmenu dát má na výber dve možnosti:

1. vykonať zmenu dát, v tomto prípade sa uzly A a B o tejto zmene dozvedia až v prípade, že bude sieťové prerušenie odstranené

2. zamietnuť požiadavok na zmenu dát, z dôvodu že uzly A a B sa o tejto zmene nedozvedia až do jej odstránenia

V prípade výberu možnosti čislo 1 zabezpečíme dostunosť systému naopak v prípade možnosti číslo 2 jeho konzistenciu, avšak nie je možný súčasný výber oboch riešení.

Ak od daného systému tolerujúceho sieťove prerušenia požadujem konzistenciu na úkor dostupnosti jedná sa o alternatívu CP. Takýto systém zabezpečí atomickosť operácií ako zápis a čítanie a zároveň sa môže stať, že na určité požiadavky nebude schopný odpovedať. Medzi takéto systémy môžeme zaradiť distribuovaný databázový systém využívajúci dvojfázový potvrdzovací protokol (2PC).

V prípade, že poľavíme na požiadavku konzistencie tak takýto systém bude vždy dostupný aj napriek sieťovým prerušeniam. V tomto prípade sa jedná o model AP. Je možné, že v takomto systéme bude dochádzať ku konfliktným zápisom alebo operácie čítania budú po určitú dobu vraciať nekorektné výsledky. Tieto problémy s konzistenciou sa v distribuovaných databázových systémoch riešia napríklad pomocou metódy vektorový časovač (Vector

[5] týmto sa nemyslí porucha uzlu

clock) alebo na aplikačnej úrovni. Príkladom systému patriaceho do tejto kategórie je služba DNS.

V prípade, že systém nebude schopný zvládať sieťové prerušenia tak takýto systém bude spĺňovať požiadavok konzistencie a dostupnosti, varianta CA. Jedná sa o nedistribuované systémy pracujúce na jednom fyzickom hardvéri využivajúce tranzakcie.

Vyššie popísané vlasnosti nám umožnia vhodný výber distribuovaného databázového systému podľa požiadavkov našej aplikácie.

2.6 Eventuálna konzistencia

V ideálnom svete je predstava konzistencie v distribuovaných systémoch následovná: v prípade, že sa v systéme vykoná zmena (zápis dát), na všetkych uzloch sa táto zmena prejaví súčasne s rovnakým výsledkom. Konzistencia v distribuovanom databázovom systéme je úzko spojená s replikáciou. Keďže podľa CAP teórie distribuovaný systém nemôže súčastne spĺňať požiadavok dostupnosti, konzistencie v prostredí s možným sieťovým prerušením, je na našom zvážení ktorú z týchto vlasností uprednostníme pri návrhu a tvorbe aplikácií. Väčšina NoSQL systémov poskytuje eventuálnu konzistenciu. V následujúcej časti preto popíšeme rôzne typy konzistencie.

V predchádzajúcom texte sme už spomínali, že v dnešne dobe existuje mnoho aplikácií, u ktorých je možné poľaviť na požiadavku konzistencie a funkčnosť systému nebude v tomto prípade ohrozená, ak sa určitá zmena prejaví s miernym oneskorením. Takáto konzistencia je odlišná od definície vlastností ACID, kde ukončenie tranzakcie spôsobí, že systém sa nachádza v konzistentom stave. Na konzistenciu sa môžeme pozerať z dôch pohľadov. Prvým je klientský pohľad na strane zadávateľa problému resp. programátora, ktorý rozhodne aká je závažnosť zmien, ktoré sa budú vykonávať v systéme. Druhý pohľad je serverový, ktorý zabezpečuje technické riešenie a implementáciu techník zodpovedných za konzistenciu v distribuovaných databázových systémoch.

2.6.1 Konzistencia z pohľadu klienta

Pre potrebu následujúcich definíc uvažujme distribuovaný databázový systém, ktorý tvorí úložisko dát a tri nezávislé procesy A,B,C, ktoré možu v danom systéme zmeniť hodnotu dátovej jednotky, tj. vykonať zápis. Tieto procesy môžu zároveň zo systému hodnotu dátovej jednotku prečítať. Na základe toho ako dané procesy pozorujú nezávisle zmeny systému delíme konzistenciu [**?**] na:

Silná konzistencia (Strong consistency) - proces A vykoná zápis. Po jeho ukončení je nová hodnota dátovej jednotky dostupná všetkým procesom A, B, C, ktoré k nej následne pristúpia - vykonajú operáciu čítania. Túto konzistenciu zabezpečujú tranzakcie s vlasnosťami ACID.

Slabá konzistencia (Weak consistency) - proces A vykoná zápis novej hodnoty do dátovej jednotky. V takomto prípade systém negarantuje, že následne pristupujúce procesy A, B, C k tejto jednotke vrátia jej novú hodnotu. Definujeme pojem „nekonzistentné okno“ zabezpečujúci, že po uplynutí určitej doby sa táto nová hodnota dátovej jednotky prejaví vo všetkých procesoch, ktoré k nej pristúpia.

Eventuálna konzistencia (Eventual consistency) - je to špecifická forma slabej konzistencie. V tomto prípade systém garantuje, že ak sa nevykoná žiadná nová zmena hodnoty dátovej jednotky, po určitom čase budu všetky procesy pristupujúce k tejto jednotke schopné vrátiť jej korektnú hodnotu. Tento model ma viacero variacií, niektoré z nich popíšeme v nasledujúcej časti textu.

Read-your-write consistency - v prípade, že proces A zapíše novú hodnotu do dátovej jednotky, žiadny z jeho následujúcich prístupov k tejto jednotke nevráti staršiu hodnotu ako jeho posledný zápis.

Session consistency - v tomto prípade pristupuje proces k systému v kontexte relácií. Po dobu trvania relácie platí predchádzajúci typ konzistencie. V prípade zlyhania relácie sa vytvorí nová, v ktorej môže systém vraciať hodnotu dátovej jednotky, zápisanú pred vznikom predchádzajúcej relácie.

Monotonic read consistency - v prípade, že proces vrátil určitú hodnotu dátovej jednotky, tak v každom ďalšom prístupe, nemôže nastať situácia, kde by vrátil predchádzajúcu hodnotu dátovej jednotky.

Tieto typy konzistencie je možné navzájom kombinovať a ich hlavným cieľom je zvýšiť dostupnosť distribuovaného systému na úkor toho, že poľavíme na požiadavkoch konzistencie. Príkladom môže byť asynchrónna replikácia v modernom relačnom databázovom systéme, ktorá spôsobí, že systém bude eventuálne konzistentný.

2.6.2 Konzistencia na strane servera

Kôrum je minimálny počet hlasov, ktorý musí obdržať distribuovaná tranzakcia aby mohla následne vykonať operáciu v distribuovanom systéme. Technika založená na protokoloch kvôra (quorum-based protocols) je pužívaná na vykonávanie konzistentných operácií v distribuovaných databázových systémoch.

Definujme nasledujúcu terminológiu:

- N - počet uzlov, ktoré obsahujú repliku dát
- W - počet uzlov obsahujúcich repliku, na ktorých sa musí vykonať zápis, aby bola zmena úspešne potvrdená
- R - počet uzlov s replikov, ktoré musia vrátiť hodnotu dátového objektu v prípade operácie čitanie

Rôzna konfigurácia týchto parametrov zabezpečí rozdielnu výkonnosť a dostupnosť distribuovaného systému. Uvažujme následujúce príklady, kde N = 3.

1. R = 1 a W = N, v takomto prípade zabezpečíme že systém bude optimalizovaný pre operácie čítania dát. Operácie budú konzistentné, pretože uzol z ktorého dáta čítame sa prekrýva s uzlami na ktorých vykonávame zápis. Nevýhodou tohoto modelu je, že v prípade nedostupnosti všetkých replík nebude možné do systému zapisovať. V prípade systémov, kde vyžadujeme rýchle čítanie a na systém je obrovský počet požiadavok čítania sa môže hodnota N pohybovať v stovkách až tisícoch, závisí to od počtu uzlov v systéme.

2. W = 1 a R = N, tento prípad je vhodný pre systémy u ktorých požadujeme rýchly zápis. Tento model môže spôsobiť stratu dát v prípade, že systém s replikou na ktorú sa vykoná zápis zlyhá.

3. W + R <= N, tento model spôsobí, že uzly, na ktoré sa vykonáva zápis a čítanie sa neprekrývajú, z čoho vyplýva u distribuovaného databázového systému vlasnosť eventuálnej konzistencie.

Nekonzistentnosť dát, môže byť tolerovaná v distribuovaných systémoch , ktoré sú vysoko škálovateľné za cieľom dosiahnutia lepšieho výkonu operácií, ktoré slúžia pre zápis a čítanie dát, celkovej výkonnosti a dostupnosti systému. Hranica do akej miery je možné dovoliť nekonzistenciu je určená požiadavkom klientskej aplikácie a vyššie spomínané modely sa ju snažia riešiť. Väčšinu z týchto modelov implementujú NoSQL systémy.

2.7 MapReduce

Nárast diskových kapacít a množstva dát, ktoré na nich ukladáme spôsobuje jeden z ďalších problémov, ktorým je analýza a spracovanie dát. Kapacita pevných diskov sa za posledné roky mnohonásobne zvýšila v porovnaní s dobou prístupu a prenosových rýchlosti pre čítanie a zápis dát na tieto zariadenia.

Pre jednoduchosť uvažujme nasledujúci príklad, v ktorom chceme spracovať pomocou jedného počítačového systému 1TB dát uložených na lokálnom súborovom systéme, pri priemernej prenosovej rýchlosti diskových zariadení 100Mb/s. Za ideálnych podmienok by čas na prečítanie týchto dát presahoval dve a pol hodiny. Tento čas je z praktických dôvodov neprípustný. V prípade, že by sme tento 1TB dát vhodne rozdelili na sto počítačov a na každom z nich tento úsek spracovali, celková doba spracovania by sa znížila za ideálnych podmienok na necelé tri minúty. Spoločnosť Google v roku 2004 zverejnila programovací model pod názvom MapReduce [**?**], ktorý rieši tento problém pomocou paralelizácie výpočtu.

MapReduce je programovací model, ktorý slúži na paralelne spracovanie dát (PB). Model využíva vlastnosti paralelných a distribuovaných systémov, je optimalizovaný pre beh na klastri, tvorenom vysokým počtom (tisícky) spotrebných počítačov. Jeho cieľom je pre programátora zastrieť všetky problémy, ktoré spôsobuje paralelizácia, poruchovosť systémov, distribúcia dát vzhľadom na ich lokalitu a rovnomerne rozvhovanie záťaže medzi systémami. Poskytuje rozhranie pre automatickú paralelizáciu a rozsiahly distribuovaný výpočet.

Pre použitie tohoto nástroja musí programátor zadefinovať dve funkcie pod názvom map a reduce. Funkcia map na jednotlivých uzloch systému, transformuje vstupné data na základe zadefinovaného kľúča (k1) na medzivýsledok, ktorý obsahuje nové kľúče (g1,...,gn) a k ním odpovedajúce hodnoty. Tieto hodnoty sa zoradia podľa ich prislušnoti ku kľúčom (g1,...,gn), následne sa odošlú na uzly s funkciou reduce. Užívateľom definovaná funkcia reduce nad hodnotami priradenými pre kľúč (g1,...,gn) prevedie operáciu, ktorej typickým výsledkom je jedna výsledná hodnota.

map(kľuč k1, hodnota) -> list(kľuč(z g1,...,gn), hodnota2)

reduce(kľúč(z g1,...,gn), list(hodnota2)) -> list(hodnota3)

2.7.1 Príklad

? histogram slov

2.7.2 Architektúra

Obrázok

2.7.3 Použitie

MapReduce nie je vhodný na spracovanie dát v reálnom čase, online spracovanie dát, ktoré sú citlivé na latenciu a to z dôvodu jeho optimalizácie pre dávkový beh.

Výhodou tohoto modelu je, že dokáže pracovať s neštrukturovanými dátami. Jeho implementácia spoločnosťou Google, ktorá zároveň využíva distribuovaný súborový systém GFS nie je verejna. V rámci hnutia NoSQL vzniklo open-source riešenie pod názvom Hadoop, ktoré implementuje tento model na vlastnom distribuovanom súborovom systéme HDFS. Zároveň vznikli frameworky ako HIVE alebo PIG, ktoré sú nadstavbou modelu MapReduce, majú syntax podobnú jazyku SQL a využívajú ich NoSql databázové systémy pre spracovanie dát.

2.8 Zhrnutie kapitoly

Cieľom tejto časti bolo pochopiť základné princípy, ktoré platia v distribuovaných databázových systémoch a sú súčasťou systémov NoSQL. Taktiež sme identifikovali problémy, ktoré môžu nastať v prípade tvorby distribuovaných relačných databázových systémov a stručne popísali možnosti ich riešení.

Kapitola 3

Definícia problému

Množstvo digitálnych informácii, každým rokom prudko narastá. Podľa štatistík spoločnosti IDC [10] v roku 2006 dosahovala kapacita digitálneho univerza veľkosť 161 exabytov. Podiel elektronickej pošty bez spamu, tvoril 3% z tohoto objemu. Podľa odhadov na rok 2010 mala kapacita digitálneho univerza dosiahnuť veľkosť 988 EB (exabyte), čo je takmer šesťnásobok nárastu pôvodnej kapacity v období štyroch rokov. V rozmedzí rokov 1998 až 2006 sa mal počet schránok elektronickej pošty zvýšiť z 253 miliónov na 1.6 miliardy. Predpoveď IDC ďalej uvádzala, že po ukončení roku 2010 tento počet presiahne hodnotu dvoch miliard. Počas obdobia medzi rokmi 1998 až 2006 celkový počet odoslaných správ elektronickej pošty rástol trikrát rýchlejšie ako počet jej užívateľov, dôvodom tohoto prudkého nárastu bola nevyžiadaná elektronická pošta, nazývaná spam. Predpokladá sa, že až 85% dát z celkového odhadovaného objemu 988 EB budú spracovávať, prenášať alebo zabezpečovať organizácie. Napriek tejto explózii digitálnych informácii je potrebné správne porozumieť hodnote týchto dát, nájsť vhodné metódy pre ich ukladanie do pamäti počítačových systémov, ich archiváciu a to tak aby sme ich mohli ďalej spracúvať a efektívne využiť. Táto časť práce si preto kladie za cieľ pochopiť štruktúru dát, ktoré reprezentujú elektronickú poštu tj. emaily a následne pomocou využitia vhodných databázových technológií popísať a navrhnúť model systému pre ich archiváciu.

3.1 Archivácia elektronickej pošty

S neustálym nárastom informácii obsiahnutých v digitálnom univerze, sa zároveň zvyšuje objem dát, ktorý reprezentuje elektronickú poštu. Je preto potrebné porozumieť štruktúre emailových správ a následne ich vhodne spracovať. Tieto dáta je potrebné uložiť tak aby sme dosiahli úsporu diskového priestoru, boli sme nad nimi schopný vykonávať operácie ako fultextové vyhľadávanie, zber údajov pre tvorbu štatistík alebo ich opätovné sprístupnenie. Emaily obsahujú čoraz viac podnikových obchodných infomácií a iný dôležitý obsah, z tohoto dôvodu musia byť organizácie všetkých rozmerov schopné uchovávať tento obsah pomocou vhodných archivačných nástrojov. S problémom archivácie zároveň úzko súvisí problém bezpečnosti. Pod pojmom bezpečnosti v tejto oblasti máme na mysli hlavne ochranu proti nevyžiadanej pošte tj. spamu, spyware, malware a phishingu. Na boj proti týmto hrozbám využívajú organizácie anti-spamové a anti-vírusové systémy. Možné dôvody prečo archivovať elektronickú poštu sú následovne: ??? strucne popisat jednotlive pojmy

- súkromie dát
- regulatory compliance
- e-discovery
- knowledge management
- self-service for end users
- disaster recovery
- legislatíva (zákon ukladá povninnosť niektorým subjektom archiváciu)
- corporate policy (nutnost skladovat skutecnou kopii emailove komunikace mimo pusobnost uzivatele pripadne technicky prostredku) ???
- datamining

3.2 Požiadavky na systém

V následujúcej časti popíšeme vlastnosti systému, ktorý bude slúžiť na archiváciu veľkého objemu emailových správ. Primárnym požiadavkom na systém je jeho rozšíriteľnosť, dostupnosť a nízkonakladová administrácia. Predpokladané množstvo uložených dát v tomto systéme bude dosahovať desiatky až stovky TB (terabyte). Takúto kapacitu dát nie je možne uchovať na bežne prístupnom spotrebnom hardvéri (commodity hardware). Systém musí byť neustále dostupný a dáta zalohované v prípade vzniku havárie niektorej z jeho časti. Nad uloženými dátami, je potrebné vykonávať výpočtovo náročné operácie ako generovanie štatistík. Tieto požiadavky prirodzene implikujú využitie distribuovaného databázového systému. Medzi hlavných kandidátov, vďaka ktorým sme tieto požiadavky schopný vyriešiť patria NoSQL databázové systémy, ktorých zrovnanie popíšeme v následujúcej kapitole.

3.2.1 Nefunkčné požiadavky

3.2.1.1 Rozšíriteľnosť

Predpokladáme použitie bežne dostupného spotrebného hardvéru, namiesto superpočítačov. Z dôvodu neustalého nárastu objemu elektronickej pošty, musí byť systém schopný horizontálneho škálovania, ktoré využijeme pre zvýšenie celkovej kapacity dátového úložiska (stovky terabajtov). Pridávaním nových uzlov do systému zároveň získame vyšší vypočetný výkon, ktorý využijeme na spracovanie dát pomocou techniky MapReduce. U databazového systému je nutná podpora replikácie, ktorá zvýši výkonnosť operácií pre čítanie a zápis do systému a vďaka nej nebude potrebné riešiť externé zálohovanie systému.

Požiadavok	Popis požiadavku	Riziko	Priorita

3.2.1.2 Dostupnosť

Systém musí byť neustále dostupný a schopný odolávať poruchám v sieťovej komunikácií (network partitions), krátkodobej neodstupnosti uzlov, úplným zlyhaniam jednotlivých uzlov a umožnovať spracúvať tok v rozmedzí 10Mbps až 1Gbps. Je vhodné aby sa dáta replikovali vo vnútry datacentra na dva uzly a tretia replika bola umiestnená v datacentre, ktoré sa bude nachádzať na geograficky odlišnom mieste. Vyžadujeme aby systém neobsahoval bod, ktorého porucha by spôsobila celkovú nedostupnosť systému (ang. single point of failure). Tento problém rieši vlasnosť decentrelizácie, ktorá zabezpečuje, že každá jednotka systému vykonáva rovnakú funkciu a je kedykoľvek nahraditeľná.

3.2.1.3 Nízkonákladová administrácia

Prevádzkovanie systému a jeho administrácia by mala byť čo najmenej náročná. Výmena alebo pridanie uzlu by mali trvať maximálne 24hod.

3.2.1.4 Bezpečnosť

Osoby s oprávnením pre prístup k systému budú schopné oprerovať s jeho celým obsahom. Nekladieme žiadne požiadavky na uživateľské role.

3.2.1.5 Implementačné požiadavky

Cieľom je implementácia systému s využitím open source technologií.

3.2.2 Funkčné požiadavky

3.2.2.1 Ukladanie emailov

Systém musí umožnovať uloženie emailu a jeho obálky do databáze v pôvodnej podobe. Klúčovým požiadavkom je ukladanie príloh emailov, kde požadujeme aby sa unikátne prílohy ukladali do systému len raz. Dôvodom je vysoká úspora dátového priestoru. Požadujeme aby bolo možné globálne nastaviť pre emaily patriace do špecifickej domény dobu počas, ktorej budú v systéme archivované.

???

- čas prijatia emailu
- obálka emailu
- predmet správy
- identifikátor emailu (message ID)
- názvy príloh a ich veľkosti

- celková veľkosť emailu
- doba spracovania emailu pomocou antivýrových a antispamových aplikácií

3.2.2.2 Export emailov

Systém musí umožnovať prístup k ľubovoľnému uloženému emailu v jeho pôvodnej podobe.

3.2.2.3 Vyhľadávanie emailov

Vyžadujeme fultextové vyhľadávanie emailov podľa údajov nachádzajúcich sa v obálke, podľa hlavičiek (predmet správy, odosielateľ, príjemca), podľa textu obsiahnutom v tele správy a podľa názvu prílohy.

Hľadanie je potrebné realizovať nad všetkými emailovými správami uloženými v systéme, nad správami podľa danej domény a nad správami, ktoré prináležia danému uživateľovi.

3.2.2.4 Štatistiky

Nad uloženými správami budeme počítať štatistiky pomocou využitia metodiky MapReduce. Pre emaily patriace do danej domény budeme v mesačných intervaloch sledovať následovné ukazatele:

- počet všetkých emailov
- počet jedinečných príjemcov
- počet emailov označených ako spam
- celková kapacita, ktorú zaberajú emaily
- celková kapacita, ktorú zaberajú emaily s príznakom spam
- ???
- ???

Poďla popisu databázových syštémov v prechádzajúcej kapitole vidíme, že použitie relačnej databáze nie je vhodné pre riešenie nášho problému. Medzi základné problémy patrí náročné horizontálne škalovanie a problémy s dostupnosťou. V nasledujúcej časti sa budeme preto zaoberať popisom NoSQL a po ich analýze vyberieme dvoch kandidátov, ktorých použijeme pre implementáciu riešenia.

Kapitola 4

NoSQL

Názov NoSQL bol prvykrát použitý v roku 1998 ako názov relačnej databázy, ktorej implementácia bola prevažne v interpretovaných programovacích jazykoch a neobsahovala jazyk SQL. V druhej polovici roku 2009 [?] sa názov NoSQL začal používať v spojení s databázovými systémami, ktoré nepoužívajú SQL dotazovací jazyk a tradičný relačný model, sú schopné horizontálneho škálovania pracujú na bežných spotrebných počítačoch, vyznačujú sa vysokou dostupnosťou, odolávajú chybám (hw sw siet) a používajú jednoduchý alebo bezschémový dátový model.

Novo vznikajúce webové aplikácie ako napríklad sociálne siete spracúvajú čoraz väčší objem dát, musia byť schopné v daný moment obslúžiť čoraz väčší počet užívateľov a neustále dostupné. Pôvodným cieľom hnutia NoSql, bolo vytvoriť koncept, pre tvorbu moderných databáz, ktoré by boli schopné riešiť tieto nové požiadavky. Idea týchto systémov je založená na filozofii, ktorá tvrdí, že nemá zmysel sa za každú cenu snažiť prispôsobiť dáta modelu relačnej databáze. Cieľom je vybrať systém, ktorý bude čo najvhodnejšie opovedať požiadavkom na uloženie a spracovanie našich dát. NoSQL obecne nepopisuje, žiaden konkrétny databázový systém, namiesto toho je to obecný názov pre nerelačné (non-relational) databázové systémy, ktoré majú odlišné vlastnosti a umožnujú prácu s rôznými dátovými modelmi. Medzi ich ďalšie znaky patrí slabá konzistencia, možnosť spracúvať obrovské objemy dát (PB), jednoduché API a možnosť asynchrónného zápisu dát. Tieto systémy nepodporujú operáciu databázového spojenia z dôvodu, že znižuje výkonnosť a zvýšuje zaťaženie siete (v prípade, že by sa táto operácia mala vykonať nad dátami, ktoré sa nachádzajú na rôznych uzloch). Pre tieto databázove systémy ďalej platí, že sú distribuované, podporujú automatickú replikáciu a rozsekávanie dát. Kedže sa jedná o relatívne mladé systémy, jedným z ich nedostatkov je malá podpora frameworkov, neustále sa meniace API a taktiež u mnohých chýbajúce rôzne grafické utility pre ich správu a monitoring. Medzi dátami, ktoré do nich ukladáme, je možné vytvárať vzájomné závislosti až na aplikačnej vrstve. Cieľom tohoto konceptu je riešiť spomínané novo vznikajúce problémy a zároveň koexistovať s relačnými databázovými systémami.

4.1 Typy NoSQL databázových systémov

Medzi nerelačné databázové systémy patria objektové, dokumentové, grafové, stĺpcové a databázové systémy s dátovým modelom typu kľúč-hodnota. V následujúcej časti stručne

popíšeme štvoricu najpopulárnejších.

4.1.1 Kľúč-hodnota (Key-value)

Tento model využíva pre ukládanie dát jednoduchý princíp. Blok dát, ktorý môže mať ľubovoľnú štruktúru je v databáze uložený pod názvom kľúča, ktorý zvykne byť reprezentovaný ako textový reťazec. Databázové systémy, využívajúce tento model majú jednoduché API rozhranie:

```
void Put(string kluc, byte[] data);
byte[] Get(string key);
void Remove(string key);
```

Výhodou tohoto modelu je, že databázový systém je možné ľahko škálovať. Bohužial v tomto prípade sa o štruktúru uložených dát musí starať klient, čo umožnuje dosahovať vysokú výkonnosť na strane databázového systému. Tento model existuje v mnohých modifikáciach.

Relačný databázový model reprezentuje dáta pomocou tabuliek, pre ktoré definujeme ich štruktúru a ktoré sú normalizované aby sme predišli duplikácii dát. Pre zabezpečenie integrity jednotlivých entít a referenčnej integrity využívame primárne a cudzie kľúče. Tabuľky s popisom názvov ich stĺpcov a vzťahy medzi nimi nazývame databázovou schémou.

Najväčšou nevýhodou je, že databázový systém nie je schopný medzi uloženými dátami zachytiť ich vzájomné relácie, čo patrí medzi základné požiadavky pri modelovaní dát. Úložiško typu kľúč-hodnota nevyužíva normalizáciu dát, dáta sú často duplikované, vzťahy a integrita medzi nimi sa riešia až na aplikačnej úrovni. Pre vkladané dáta a k nim asociované kľúče sa nedefinujú žiadne obmedzenia.

Medzi databázové systémy využívajúce model kľúč-hodnota patria: Dynamo, Tokyo Cabinet, Voldemort, Redis a iné.

4.1.2 Stĺpcovo orientovaný model (Column [Family] Oriented)

Množstvo databázových systémov využíva pre reprezentáciu dát tabuľky, ktoré sú tvorené stĺpcami a popísané schémou (tj. názvy stĺpcov, tabuliek). Každý riadok tabuľky reprezentuje dáta, ktoré sa nazývajú záznamy a tieto sú následne sekvenčne ukladané na pevný disk. Tento model, nazývaný riadkový, je vhodný pre systémy, u ktorých sú dominantou operácie vykonávajúce zápis. Relačné databázové systémy, využívajúce tento model sú teda optimalizované pre zápis. Pre efektívny prístup k dátam môže tento model využívať indexy.

V dnešnej dobe existuje veľký počet aplikácií, u ktorých prevládajú operácie čítania nad zápisom. Patria sem dátové sklady, customer relationship management (CRM) systémy, systémy pre vyťažovanie dát alebo analytické aplikácie pracujúce s obrovským objemom dát. Pre potreby týchto aplikácií a ich reprezentáciu dát je vhodné použiť stĺpcový model [13][14], ktorý je zároveň optimalizovaný pre operácie čítanie dát. Data reprezentujúce stĺpce sú uložené na pevnom disku v samostatných a súvislých blokoch. Načítanie dát do pamäti a následná práca s nimi je efektivnejšia ako u riadkových databáz, kde je potrebné načítať celý záznam obsahujúci hodnoty stĺpcov, ktoré sú pre nás v daný moment irelevantné.

Riadkový model obsahuje v jednom zázname dáta z rôznych domén, čo spôsobuje vyššiu entropiu v porovnaní so stĺpcovým modelom, kde sa predpokláda, že dáta v danom stĺpci pochádzajú z totožnej domény a môžu si byť podobné. Táto vlasnosť umožnuje efektívnu komprimáciu dát, ktorá znižuje počet diskových operácií. Nevýdou tohoto modelu je zápis dát, ktorý spôsobuje vyššiu záťaž diskových operácií. Optimalizáciou môže byť dávkový zápis dát.

4.1.2.1 Stĺpcovo orientovaný model v NoSQL

Predchodcom tohto nového prístupu k stĺpcovému modelu v NoSQL systémoch je databázový systém od Google - Bigtable. V tomto prípade sa využíva kombinácia modelu kľúč-hodnota so stĺpcovo orientovaným modelom. Na takýto model sa môžeme pozerať ako na viacdimenzionálne úložisko typu kľúč-hodnota. Detailnejšie tento model popíšeme v následujúcej kapitole.

Tento model sa používa v databázových systémoch ako Google BigTable, Hbase, Hypertable alebo Cassandra.

4.1.3 Dokumentový model

Dokumentové databáze sú založené na predchádzajúcom modeli typu kľúč-hodnota. Požiadavkom na ukládané data je, že musia byť v tvare ktorému rozumie databázový systém. Štruktúra vkládaných dát môže byť popísaná napríklad pomocou XML, JSON, YAML. Táto štruktúra nám následne umožnuje okrem jednoduchého vyhľadávania pomocou kľúč-hodnota vytvárať s využitím indexov zložitejšie dotazy nad dátami na strane databázového systému.

Medzi databáze reprezentujúce tento typ úložiska patrí napríklad CouchDB a MongoDB.

4.1.4 Grafový model

Tento typ databáz využíva pre prácu s dátami matematickú štruktúru - graf. Dáta sú reprezentované pomocou uzlov, hran a ich atribútov. Uzol je základný samostatný a nezávislý objekt. Pomocou hran medzi uzlami modelujeme závislosti, ktoré popisujeme pomocou atribútov. Nad uzlami a hranami sa využíva model kľúč-hodnota. Medzi hlavné výhody patrí možnosť prechádzania týchto štruktúr s využitím známych grafových algoritmov.

Tento model sa napríklad využíva v aplikáciach sociálnych sieti alebo pre sémantický web. Patria sem databáze ako Neo4j alebo FlockDB.

4.2 Porovnanie NoSQL systémov

V dnešnej dobe existuje veľké množstvo NoSQL databázových systémov, ktoré majú odlišné vlasnosti, komplexitu a vďaka tomu ich môžeme použiť pre rôzne účely. Snaha porovnať tieto systémy na globálnej úrovni je nerealizovateľná a často vedie k omylu. Cieľom tejto sekcie je definovať základné body vďaka, ktorým je možné tieto systémy kategorizovať a vrámci danej kategórizácie porovánať. Tieto zistenia nám následne môžu pomôcť pri výbere vhodného systému odpovedajúceho našim požiadavkom.

Následujúce body patria medzi hlavné kritéria pri porovnávaní týchto systémov:

1. dátový model
2. dotazovací model
3. škálovateľnosť
4. schopnosť odolávať chýbam (failure handling)
5. elastickosť
6. konzistencia dát
7. typ perzistentného úložiska
8.

4.2.1 Dátový a dotazovací model

Dátový model definuje štruktúru, ktorá slúži na ukladanie dát v databázovom systéme. Dotazovací model následne definuje obmedzenia a operácie, ktoré sme schopný nad uloženými dátami vykonávať. V predchádzajúcej sekcii sme popísali základnú kategorizáciu NoSQL systémov poďla dátového modelu. Dátový a dotazovací model do určitej miery popisuje výkonnosť, komplexnosť a vyjadrovaciu silu databázového systému. Dotazovací model popisuje API.

Pre výber vhodného dátového modelu pre našu aplikáciu je dôležité porozumieť štruktúre dát a definovať operácie, ktoré nad týmito dátami budeme vykonávať.

4.2.2 Škálovateľnosť a schopnosť odolávať chybám

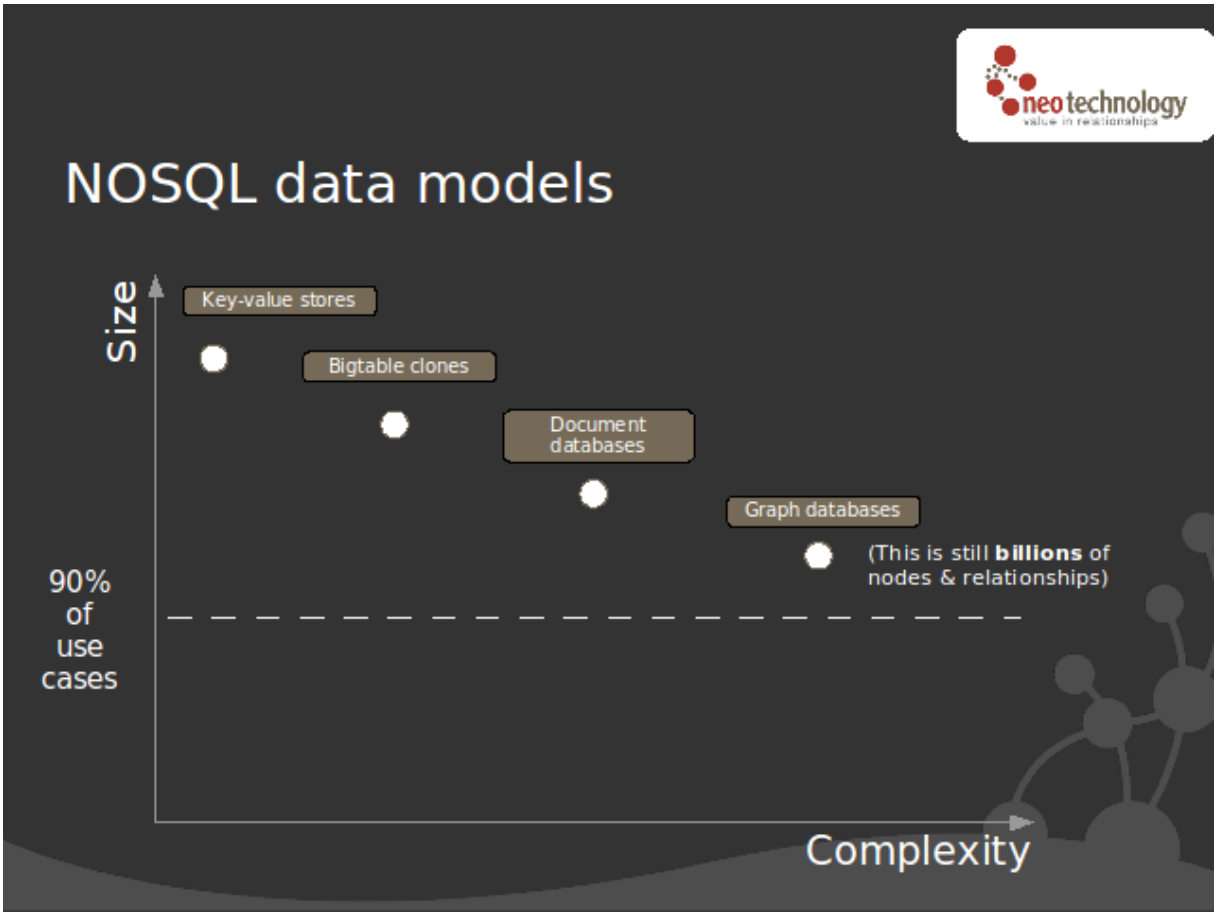
Táto vlasnosť kladie na systém požiadavky ako podpora replikácie a rozsekávania dát. Distribuované databázové systémy implementujú tieto techniky na systémovej úrovni. V prípade ich podpory, môžeme od systému požadovať:

- podporu replikcácie medzi geograficky oddelenými dátovými centrami
- možnosť pridania nového uzlu do distribuovaného databázového systému, bez nutnosti zásahu do aplikácie

Počet uzlov, na ktorých sa vykonáva replika dát a konfigurácia databázového systému, ktorá podporuje geograficky oddelené dátové centrá zároveň určujú akej miery je systém schopný odolávať chybám, medzi ktoré môžeme zaradiť poruchu uzlu alebo sieťové prerušenia.

Častým požiadavkom webových aplikácií, z dôvodu neustáleho nárastu dát, na databázový systém je podpora škálovania s cieľom zvýšenia veľkosti databáze. S neustálym vývojom nových aplikácií musíme uvažovať potrebu škálovania z pohľadu komplexnosti. V tomto prípade predpokladáme, že štruktúra dát ktoré do databázového systému ukladáme sa môže s postupom času meniť. Pojem škálovanie z pohľadu komplexnosti je popísaný v knihe... Schopnosť škálovania z pohľadu komplexnosti ovplyvňuje výber dátového modelu.

Obrázok XY zachytáva pozíciu NoSQL dátových modelov z pohľadu škálovania komplexnosti a veľkosti dát.



Obr. 4.1: Pozícia dátového modelu z pohľadu jeho škálovania podľa veľkosti a komplexnosti. Zdroj: Neo4J a NOSQL overview and the benefits of graph databases, Emil Eifrem, prezentacia.

Dátový model typu kľuč-hodnota a stĺpcovo orientovaný model (Bigtable clones) majú jednoduchú štruktúru, ktorá sa dá horizontálne škálovať. Nevýhodou tohoto prístupu je naopak to, že všetká práca s dátami a ich štruktúrou sa prenáša do vyšších vrstiev, o ktoré sa musí starať programátor. Naopak dokumentový a grafový model poskytuje bohatšiu štruktúru na prácu s dátami, ktorá spôsobuje komplikovanejšie škálovanie vhľadom na veľkosť dát. Podľa odhadov spoločnosti Neotechnology až 90% aplikáci, v prípade že sa nejedná o projekty spoločností Google, Amazon atď., spadá do rozmedia kde sa veľkosť záznamov pohybuje rádovo v miliardách. Za zmienku stojí fakt, že aj napriek tomu, že tieto dátové modely sú si navzájom izomorfné, vhodnosť ich použitia závisí na konkrétnom príklade a požiadavkoch na aplikáciu.

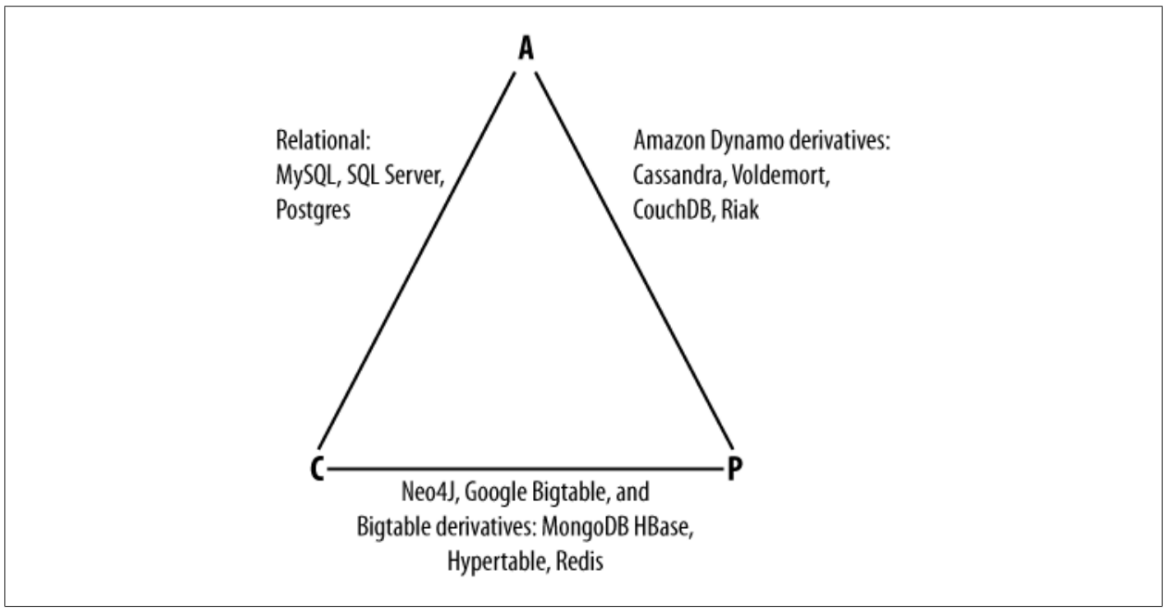
4.2.3 Elastickosť

Vďaka horizontálnemu škálovaniu sa snažime o zýšenie kapacity celkového dátového úložiska. Elastickosť škálovania popisuje ako sa daný systém dokáže vysporiadať s pridaním alebo

odobraním uzlu. U tejto vlastnosti sledujeme či je potrebné manuálne rebalancovanie dát, reštartovanie celého systému alebo zmena v uživateľskej aplikácii. Táto vlastnosť by ideálne mala zabezpečiť lineárne zvyšovanie výkonu u operácií ako je čítanie alebo zapisovanie dát.

4.2.4 Konzistencia dát

Poďla teórie CAP platí, že v prípade výskytu sieťových prerušení, ktoré sú súčasťou distribuovaného databázového systému nie je možné súčasne zaručiť vlasnosť konzistencie a dostupnosti. NoSQL systémy preto môžeme rozdeliť podľa tohoto modelu.



Obr. 4.2:

Umiestnenie niektorých NoSQL datázových systémvo sa môže meniť podľa ich konfigurácie.

Podpora replikácie, rozsekávania dát a zaradenie systému podľa modelu CAP určuje jeho dostupnosť.

4.2.5 Perzistentné úložisko

Typ perzistentného úložiska popisuje interný spôsob ukládania dát v databázovom systéme. NoSQL systémy môžu používať na ukládanie dát následujúce štruktúry:

- B-stromy
- distribuované hešové tabuľky
- Memtable / SSTable [1]

[1] Tieto štruktúry popíšeme v následujúcej sekcii

- spojové zoznamy
- operačná pämeť, ktorej obsah je pravidelných intervaloch ukládaný na pevný disk

Podľa požiadavkov na našu aplikáciu sme čiastočne schopný odhadnúť pomocou, ktorej štruktúry by sme mohli dosiahnúť čo najefektívnejšiu výkonnosť.

4.3 Výber NoSQL systémov

V predchádzajúcej sekcii sme tieto systémy rozdelili do štyroch hlavných kategórií podľa ich dátového modelu, ktorý je kľúčový pri výbere vhodného databázového systému podľa požiadavkov aplikácie. Popis a výkonnostné porovnanie NoSQL systémov, ktoré reprezentujú jednotlivé kategórie by boli nad rámec tejto práce. Paralelne s touto prácou vzniká diplomová práca, ktorá rieši podobný problém s využitím dokumentových databázových systémov **??**, preto túto kategóriu vynecháme.

NoSQL systémy môžeme rozdeliť podľa toho v akom prostredí pracujú. Väčšina týchto systémov vyžaduje ich inštaláciu na počítačové systémy, patria sem napríklad Voldemort, Cassandra, Riak, Hbase. Tieto systémy sú open-source. Okrem nich existujú distribuované databázové systémy, ktoré sú poskytované ako cloudové riešenie a to Amazon SimpleDB, Microsoft Azure SQL, Yahoo! YQL a prostredie od spoločnosti Google AppEngine. Tieto systémy poskytujú priamo rozhranie na prácu s dátami a funkčnosť perzistentného úložiska zabezpečujú poskytovatelia týchto služieb. Ich hlavným cieľom je zefektívnenie vývoja a nasadzovania aplikácií.

Podľa analýzy požiadavkov na našu aplikáciu a štruktúry dát, ktoré budeme do databázového systému ukládať nie je vhodné použitie databázových systémov s grafovým modelom a modelom kľúč-hodnota. Systémy s grafovým modelom sú určené na diametrálne odlišnú úlohu problémov naopak v prípade, použita systémov kľúč-hodnota by bola práca na strane aplikácie zbytočne náročna. Našim požiadavkom najlepšie vyhovuje stĺpcovo orientovaný model, ktorý sme sa rozhodli použiť pre návrh našej aplikácie.

V tejto časti práce sa zameriame na stručný prehľad a vzájomné porovnanie systémov, ktoré poskytujú stĺpcovo orientovaný model. Medzi tieto voľne dostupné (open-source) systémy patria HBase, Cassandra a Hypertable. Aj napriek totožnému dátovemu modelu, majú tieto systémy odlišné vlasnosti. Následujúca tabuľka zobrazuje popis vlasností, na ktoré sme sa zamerali pri výbere víťaznej dvojice.

Z porovania je vidieť, že systémy obsahuju množstvo spoločných vlastností. Pri výbere systémov sme zohľadnili aj ich praktické využitie spoločnostiami pôsobiacimi na trhu v produkčných podmienkách. Systém Cassandra je používaný spoločnosťou Facebook v aplikácii na súkromnú poštu. Medzi ďalšie požiadavky patrili podpora komunity, dokumentácia a vývojový cyklus týchto systémov. Z týchto systémov sme následne vybrali dva a to HBase a Cassandru. Dôvodom prečo sme zavrhli systém Hypertable je nepostačujúca dokumentácia, málo aktívna komunita a pomalý vývojový cyklus. V následujúcej sekcii popíšeme ich detaily.

Vlastnosti/Databázový systém	Hbase	Cassandra	Hypertable
Distribuovaný systém	áno	áno	áno
Dátový model	Bigtable [2]	Bigtable	Bigtable
Dotazovací model			
Klient	Thrift, REST	Thrift, Avro	Thrift, C++
Perzistentné úložisko	HDFS	LSS[3]	HDFS, KFS, LSS
SPOF [4]	áno	nie	áno
Pridanie uzlu do živého systému	áno	áno	
Podpora viacerých datacentier	áno	áno	áno
Rozsekávanie dát	áno	áno	áno
Replikácia	pomocou HDFS	áno	áno
Elastickosť	áno	áno	
Konzistencia	CP	AP	
Programovací jazyk	Java	Java	C++
MapReduce	áno	áno	áno
Komunita	+	+	-

Kapitola 5

Cassandra

Distribuovaný databázový systém Cassandra bol vytvorený pre interné účely spoločnosti Facebook v roku 2007. Cassandra slúžila na vyhľadávanie v súkromnej pošte, poskytovala úložisko pre indexy. Hlavnými požiadavkami na tento systém bolo zvládať miliardu zápisov denne, schopnosť škálovania podľa narástajúceho počtu používateľov, beh na spotrebných počítačoch a podpora replikácie medzi geograficky oddelenými dátovými centrami. Ďalším požiadavkom bola vysoká dostupnosť, teda aby chyba žiadného uzlu nespôsobila celkovú nedostupnosť systému. Cassandra je teda decentralizovaný systém, kde každý uzol vykonáva tie isté operácie. V roku 2008 bola zverejnená ako open source projekt a je neustále vyvíjaná mnohými spoločnosťami a vývojármi. Tento systém využíva architektonické princípy distribuovaného databázového systému Dynamo od spoločnosti Amazon a zároveň ich kombinuje s dátovým modelom distribuovaného databázového systému Bigtable vytvoreného spoločnosťou Google. V následujúcom texte popíšeme hlavne princípy, na ktorých je tento systém založený.

5.0.1 Dátový model

Cassandra k dátovému modelu systému Bigtable pridala štruktúru pod názvom „super stĺpec“ (angl. super column). Základnou jednotkou dátového modelu je stĺpec. Stĺpec je tvorený názvom, hodnotou a časovým odtlačkom, ktorý využíva Cassandra pri riešení konfliktov. Skupina stĺpcov je identifikovná pomocou unikátneho kľúča a predstavuje riadok, avšak počet a názvy stĺpcov nie je potrebné vopred definovať. Zoradené riadky podľa hodnoty kľučov a v nich zoradené stĺpce obaľuje štruktúra pod názvom „rodina stĺpcov“ (ang. column family). Kľúče sú interne reprezentované ako reťazec znakov a zároveň zotriedené. Názvy stĺpcov možu byť viacerých typov ako napríklad Ascii, Utf-8, Byte podľa, ktorých sú zotriedené. Je možné implementovať vlastnú metódu pre triedenie. Riadky obsiahnuté v jednej rodine stĺpcov sú na pevnom disku fyzicky umiestnené v jednom súbore typu SSTable. Je vhodné do rodiny stĺpcov ukládať relevantné záznamy, ku ktorým budeme pristupovať spoločne, čim sa vyhneme zbytočným diskovým operáciam. Operácie nad stĺpcami, ktoré identifikuje daný kľúč sú atomické v rámci repliky. Operácie nad daným riadkom nevyužívajú zamykanie. Voliteľným príznakom, ktorý môžeme u stĺpcu nastaviť je parameter TTL (angl. time to live), ktorý po uplynutí časového intervalu označí dáta za zmazané.

Štruktúra super stĺpec je špecialny typ stĺpca, ktorý je tvorenými obyčajnými stĺpcami. Stĺpec typu super má názov a jeho hodnota je tvorená zoznamom názvov obyčajných stĺpcov. Tento prístup pridáva ďalšiu úroveň v štruktúre. Super stĺpce obaľuje podobná štruktúra pod názvom super-rodina stĺpcov (angl. super column family).

Keyspace definuje faktor replikácie a jej metódu, ktorá môže byť závyslá poprípade nezávislá na sieťovej topológii. Na keyspace sa môžeme pozerať ako na databázu v relačných databázových systémov obsahujúcu rodiny stĺpcov, ktoré môžeme prirovnať k tabuľkám v relačných databázach.

Aktuálna verzia Cassandri definuje maximálnu veľkosť dát 2GB, ktoré je možné uložit do jedného stĺpca a stanovuje limit dve miliardy pre maximálny počet stĺpcov v jednom riadku.

5.0.2 Rozdeľovanie dát

Kľúčovým požiadavkom systému Cassandra je jeho schopnosť škálovania do šírky, čo vyžaduje pridávanie nových uzlov. Tento požiadavok vyžaduje mechanizmus, ktorý zabezpečí dynamické rozdeľovanie dát medzi uzlami systému. Uvažujme príklad, kde máme k dispozícii jeden server obsahujúci veľké množstvo objektov, ku ktorým pristupujú klienti. Medzi server a klientov vložíme vrstvu kešovacích systémov, kde každý z týchto systémov bude zodpovedný pre rýchly prístup k danej časti objektov nachádzajúcich sa na serveroch. Klient teda musí byť schopný určiť, ku ktorém kešovaciemu systému musí pristúpiť v prípade, že chce daný objekt. Predpokladajme, že klientom zabezpečíme výber jednotlivých kešovacím systémom pomocou hešovania s využitím lineárnej hešovacej funkciek (x -> ax + b (mod p), kde p je počet kešovacích systémov). Pridanie nového kešovacieho systému alebo jeho zlyhanie bude mať katastrofálny dopad na funkčnosť systému. V prípade, že sa zmení parameter p, teda počet kešovacích systémov každá položka bude odpovedať novej a zároveň chybnej lokácii. Tento problém rieši elegantne technika pod názvom úplné hašovanie (angl. consistent hashing) [11], ktorá sa využíva v distribuovaných systémoch pre prácu s distribuovanými hašovacími tabuľkami. Túto techniku taktiež využíva systém Cassandra.

Výstup hašovacej funkcie MD5 reprezentujeme pomocou „kruhu“, kde v smere hodinových ručičiek postupujeme od minimálnej hodnoty hešovacej funkcie (tj. 0) k maximálnej. Každý uzlol v systéme má pridelenú náhodnu hodnotu z tohoto rozsahu, ktorá určí jeho jednoznačnú pozíciu. Identifikácia uzlu v systéme, na ktorý sa uložia dáta reprezentované hodnotou kľúča sa vykoná aplikáciou hešovacej funkcie na dáta reprezentujúce kľúč. Na základe tejto hodnoty je jednoznačne určená pozícia v kruhu a v smere hodinových ručiek je vyhľadaný najbližší uzol. Výhodou tejto metódy je, že každý uzol je zodpovedný za hodnotu kľúčov, ktorých poloha sa nachádza medzi ním a jeho predchodcom. V prípade pridania nového uzlu alebo jeho odobratia, sa zmena mapovania kľúčov v kruhu prejaví len u jeho susedov. Táto technika zároveň prináša nevýhody, medzi ktoré patrí rovnomerná distribúcia dát a vyváženie záťaže. Dynamo tento problém rieši spôsobom kde každý uzol je zodpovedný za viacero pozícií na kruhu, takzvané virtuálne pozície. Cassandra využíva vlastné mechanizmi na monitorovanie záťaže a automaticky presúva pozície uzlov. Taktiež je možné explicitne u každého uzlu stanoviť jeho polohu v kruhu pomocou zadania jeho identifikátora. Tento spôsob je vhodný v prípade, ak vieme predom určiť koľko uzlov bude obsahovať náš systém. V prípade zvyšovania počtu uzlov je možné tieto identifikátory a teda ich polohu v kruhu zmeniť za chodu systému,

čím sme schopný opäť dosiahnúť jeho rovnomerné vyváženie. Identifikátor polohy uzlov vieme určiť pomocou následujúceho programu, kde K je počet uzlov v systéme.

```
RING_SIZE = 2**127
def tokens(n):
  rv = []
  for x in xrange(n):
    rv.append(RING_SIZE / n * x)
  return rv

print tokens(K)
```

5.0.3 Replikácia

S úplným hašovaním úzko súvisý replikácia, ktorá zabezpečuje vysokú dostupnosť a odolnosť dát proti ich stráte (angl. durability). Každá dátová jednotka vložená do systému je replikovaná na N uzlov, kde počet N je voliteľne nastaviteľný pre daný keyspace. Každý uzol sa v prípade replikácie $N > 1$ stáva koordinátorom, ktorý je zodpovedný za replikáciu dát, ktorých kľúč spadá do jeho rozsahu na kruhu. V prípade zápisu koordinátor replikuje dáta na ďalších N - 1 uzlov. Cassandra podporuje viacero spôsobov pre umiestňovanie replík.

Jednoduchá stratégia

Táto stratégia umiestňuje replikú dát bez ohľadu na umiestnenie serverov v datacentre. Replika dát uzlu je uložená na jeho N-1 susedov v smere hodinových ručičiek. Z toho vyplýva, že každý uzol je zodpovendný za dáta, ktorých kľúče spadajú do jeho rozsahu a taktiež dáta, ktorých kľúče spravuje jeho N predchodcov.

Sieťová stratégia

Pri tejto metóde a úrovni replikácie s hodnotou aspoň tri, sme schopný zabezpečiť umiestnenie dvoch replík v rozdielnych rackoch, tretia replika bude umiestnená do iného datacentra. Táto stratégia je výhodna v prípade ak chceme použiť časť serverov na výpočty pomocou Mapreduce a zvyšné dve repliky budú slúžiť na obsluhu reálnej prevádzky.

5.0.4 Členstvo uzlov v systéme

Distribuovaný systém musí byť schopný odolávať chybám ako porucha uzlov alebo sieťové prerušenia. Podpora decentralizácie a detekcia chýb využíva mechanizmi založené na gosship protokoloch. Tieto protokoly slúžia pre vzájomnú komunikáciu uzlov vymnieňajúcich si navzájom doležité informácie o svojom stave. Periodicky v sekundových intervaloch každý uzol kontaktuje náhodne vybraný uzol, kde si overí či je tento uzol dostupný. Detekcia možného zlyhanai uzlu je realizovaná algoritmom s názvom Accrual Failure Detector [17].

Pridávanie novýchu uzlov, presun uzlov v rámci kruhu a iné operácie sa taktiež dejú pomocou Gosship protokolu. Tento protokol zabezpečuje, že každý uzol obsahuje informácie

o tom, ktorý uzol je zodpovedný za daný rozsah kľúčov v kruhu. Ak sa vykonáva operácia čítania alebo zápisu dát na uzol, ktorý nie je zodpovedný za tento kľúč, dáta sú automaticky preposlané na správny uzol s časovou zložitosťou O(1).

5.0.5 Perzistentné úložisko

Tento systém bol primárne navrhnutý tak aby spracúval vysoký tok dát pre zápis, s tým že čo najmenej ovlplyvní efektívnosť operácií na čítanie. Cassandra využíva ako perzistentné úložisko dát lokálny súborový systém.

5.0.6 Konzistencia

Konzistencia systému je maximálne konfigurovateľná a využíva princípy techník založených na protokoch kôra. Klient si môže nastaviť hodnotu R určujúcu koľko replík musí potvrdiť úspešnosť operácie čítania dát. Hodnota W určuje na koľko replík je potrebné vykonať zápis a následne vrátiť potvrdenie o jeho úspešnosti klientovi. V prípade, že platí vzťah $R + W > N$, kde N je počet replík tak sa jedná o silne konzistentný systém, naopak v prípade voľby klienta, kde $R + W < N$, sa jedná o slabú konzistenciu čím zaručíme vysokú dostupnosť.

5.0.7 Zápis dát

Ak uzol obrží požiadaok pre zápis, dáta sú zapísané do štruktúry pod názvom commit log, ktorá je uložená na lokálnom súborovom systéme a zabezpečí trvácnosť dát. Zápis do tejto štruktúry je vykonávaný sekvenčne čo umožnuje dosiahnúť vysokú priepusnosť. Dáta sú následne nahrané do štruktúry pod názvom memtable, ktorá sa nachádza v operačnej pamati a v prípade, že by tento zápis zlyhal alebo by došlo k neočakávanému zlyhaniu inštancie Cassandri je možné ich obnovenie z commit logu. Po dosiahnutí určitéh prahu, tj. počtu dát uložených v memtable sú tieto štruktúry asynchrónne zapísané do štruktúr pod názvom SSTable (Sorted String Tables), ktoré už nie je možné modifikovať pomocou aplikácie. Štruktúry SSTables sa následne zlievajú v pravidelných intervaloch na pozadí počas behu, táto operácia je neblokujúca. Počas zlievania SSTables dochádza k zotriedenému zlievaniu kľúčov, k nim prinaležiacím dát, odtraňovaniu dát určených na vymazanie a generovaniu nových indexov. Taktiež dochádza ku generovaniu štruktúr pod názvom Bloom filters[1] pre každú SSTable.

Zápis dát nevykonáva žiadne diskové operácie, ktoré by potrebovali čítať dáta, je atomický pre danú ColumnFamily a v prípade, že systém Cassandra beží je stále dostupný a rýchly.

5.0.8 Čítanie dát

V prípade požiadavku na načitanie dát, sa požadované dáta najprv hľadajú v štruktúrach memtable, ktoré sú uložené v operačnej pamati. Ak sa dané data nenachádzajú v operačnej pamati, vyhľadávanie sa uskutočnuje podľa kľúča v diskových štruktúrach SSTable. Kedže snahou systému je čo najefektívnejšie vyhľadávanie, využívajú sa bloom filtre. Bloom filtre sú nedeterministické algoritmy, ktoré dokážu otestovať či element patrí do množiny. Napriek

[1] http://en.wikipedia.org/wiki/Bloom_filter

ich nedeterminizmu generujú len falošné pozitíva. Pomocou nich je možné namapovať kľúče zo štruktúr SSTables do bitových polí, ktoré je možné uchovať v operačnej pamati. Vďaka tomu sa redukuje prístup na disk, keď hľadáme súbor, ktorý obsahuje dáta odpovedajúce hľadanému kľúču. Požiadavok pre čítanie dát možeme zaslať na ľubovoľný uzol.

5.0.9 Zmazanie dát

Keď vykonáme operáciu reprezentujúcu zmazanie dát, tieto dáta sa nevymažú okamžite. Namiesto toho sa vykoná operácia, ktorá dané dáta označkuje príznakom pod názvom tombstone. Po uplynutí doby, ktorá je štandardne nastavená na desať dní, sa tieto dáta odstránia pri procese zlievajúcom štruktúry SSTables.

5.0.10 Bezpečnosť

Implicitne Cassandra nevyužíva žiadne prvky, ktoré by poskytovali možnosť znemožnenia prístupu k dátam v nej uložených. K dispozícii je modul, ktorý umožnuje nastavenie auntentizácie na úrovni Keyspace-u pomocou textových hesiel alebo ich MD5 odtlačkov.

Obmedzovanie prístupu k datám je preto potrebné zabezpečiť na aplikačnej úrovni.

obmedzenie MYSQL = +papier 5 mysqlscalling and high avail.pdf -

Musime predpokladat ze zlyhania v tak velkej infrastrukture su bezna vec a nie ojedinela zalezitost = HW a siet zlyhania Zapis nie je nikdy rejectnuty(ani v pripade hw poruchy ani v pripade concurent writes), z tho odovodu je to alway writtable system = co ale implikuje ze konflikty sa riesia pocas operacie READ riesenie konfliktu na strane db vyuziva jednoduche techniky = last write win, kdezto na strane klienta napriklad mozeme pouzit merge na 2 rozne verzie dat (nakupny kosik) bigtable = multi dimensional sorted map Zero hop DHS = kazdy uzol obsahuje dostatok informacii na priame presmerovanie poziadavku na cielovy uzol

kluc je array of bytes na neho sa aplikuje mdť hash ktory vygeneruje +ľábit identifikator na zaklade ktore sa urci uzol kde budu ulozene data pre dany kluc

partitioning = consistent hashing = najvacsia hodnota has funkcie v spojeni s najnizsou (0) vytvori kruh na ktory sa umiestnia jednotlive uzly.... data sa potom podla md5(key) umiestnia na dany uzol (v smere hod ruciciek ktoreho pozicia ja vacsia ako pozicia md5(kluca) Kazdy uzol je takto zodpovedny za svoj region (uzol => predchodca) ... vyhodou konzistent hashovania je ze havaria/pridanie noveho uzla ovplyvňuje len susedov. Tento koncept obsahuje viacero nedostatkov ako napriklad nerovnomerna distribucia zataze.... preto sa aplikuje upraveny consistent hasing a to tak ze kazdy uzol obsahuje viacero virtualnych pozicii na ringu

Replikacia zabezpecuje hlavne dostupnost a stalost dat durability

214: W a R impact object availability durability and consistency N R W - 3,2,2 podla dynamo paper 215

distirbucia klucov pri malej zatazi - horsie to loadbaloncovalo ako pri velkej zatazi 215 209 tabulka + obrazok

diskusia nad n,r,w 214

pouzili sme osobitny disk pre commitlog

detekcia konfliktov pri upgrade pomocou vector clock - ale kedze mi len WRITE nezaujima nas to v pripade ze preda dany stlpec nemame hodnotu nic sa nikam nezapisuje

kapitola impl – v pripade ze nejaka polozka neexistuje nic sa nedeje ... neukladame ... handlujeme na strane klienta

CAS je AP HBASE CP

Hardware failure is the norm rather than the exception.

Kapitola 6

HBase

V tejto kapitole stručne popíšeme distribuovaný súborový systém, ktorý je súčasťou projektu Hadoop a zároveň slúži ako perzistentné úložisko pre distribuovanú databázu HBase. Následne popíšeme základné princípy fungovania tohoto databázového systému.

Hadoop

je v jave

Hadoop[1] je open source projekt, ktorého hlavne cieľe sú vysoká dostupnosť, škálovateľnosť a distribuovaný výpočet. Základ tvorý distribuovaný súborový systém HDFS (Hadoop Distributed Filesystem) a framework MapReduce pre spracúvanie objemu dát v desiatkách PB [15]. Architektúra HDFS vychádza z princípov distribuovaného súborového systému GFS (Google File System) vytvoreného spoločnosťou Google[9], framework mapreduce bol inspirovany ggl mapreduce =ref clanok

kto ho pouziva a na ake typy appl

Súborový systém využíva architektúru master-slave. Uzol master, pod názvom Namenode, udržiava v pamäti RAM metadata, ktoré popisujú štruktúru súborov, adresárov, reprezentujú mapovanie súborov na bloky a určujú ich umiestnenie na uzloch Datanode. HDFS predpokláda prácu so súbormi rádovo v desiatkách gigabajtov, ktoré sú interne reprezentované dátovými blokmi o veľkosti 64-128MB. Tieto bloky sú uložené v uzloch typu slave, ktoré sa nazývajú Datanode. Klient v prípade načítania súboru kontaktuje Namenode, ktorý mu poskytne informácie, na ktorých uzloch typu Datanode sa nachádzajú bloky reprezentujúce súbor a dátová komunikácia následne prebehne medzi klientom a daným Datanode. HDFS je optimalizovaný pre jednorázový zápis dát a ich následné mnohonásobné čítanie. Bloky sa replikujú na uzly Datanode. Štandardne je nastavená úroveň replikácie na hodnotu tri, teda každý blok je uložený trikrát.

Hlavným nedostatkom tejto infraštruktúry je fakt, že uzlol Namenode tvorí kritický bod systému, v prípade jeho nedostupnosti nie je možné pracovať s HDFS a prípadná strata dát na tomto uzle spôsobí totálne zlyhanie súborového systému bez možnosti jeho obnovy. Súborový systém nie je vhodný pre ukladanie veľkého počtu malých súborov. Uzol Namenode alokuje 150B pre objekt typu blok a 150B pre objekt typu súbor. V prípade uloženia súboru,

[1] http://hadoop.apache.org/

ktorý nepresahuje veľkosť jedného bloku je potrebné alokovať 300B dát. Ak uložíme 10 000000 takýchto súborov veľkosť metadát, ktoré udržiava Namenode v operačnej pamati zaberie 3GB. Celkový počet uložených súborov je obmedzený veľkosťou pamati RAM, ktorou disponuje uzol Namenode.

Z týchto pozorovaní vyplýva fakt, že distribuovaný súborový systému HDFS nemá praktické využitie ako úložisko dát slúžiace k archivácii emailových správ, pre ktoré sme zadefinovali požiadavky v kapitole XY.

SYNC!!! a zapis pipeline style page 69 hadoop book

HBase

replikovanie in pipeline

Kapitola 7

Testovanie výkonnosti

V následujúcej kapitole sa zameriame na popis výkonnostných testov, ktoré sme vykonali v reálnych podmienkach.

7.1 Testovacie prostredie

Pre výkonnostné testovanie sme mali k dispozícii 9 počítačov s rovnakou hardverovou a softvérovou konfiguráciou, ktoré boli navzájom prepojené pomocou 10Gbit switchu a komunikovali po 1Gbit linke. Komkrétnu softvarovú konfiguráciu testovaných aplikácií popíšeme jednotlivo v nasledujúcich podkapitolách.

Hardverová konfigurácia

- 4 jádrový procesor Intel, 5506@2.13Ghz
- 4 GB RAM
- 5 pevných diskov (SATA, 7200RPM) o veľkosti 1TB - RAID0
- 1Gbit sieťová karta

Softvérová konfigurácia

Každý uzol obsahoval inštaláciu operačného systém Debian GNU/Linux Lenny x64, Sun Java JDK 1.6.0_+88. Na každom uzle bol deaktivovaný odkladací priestor (angl. swap). Za účelom monitorovania bol použitý softvér Zabbix, VisualVM a dstat.

Sieťová konfigurácia

Hodnota maximálnej reálnej sieťovej priepusnosti medzi dvoma uzlami bola zmerná pomocou aplikácie XY s výslednou hodnotou 940Mbit.

7.2 HDFS

Nad distribuovaným súborovým systémom HDFS sme vykonali testy určujúce maximálnu hodnotu priepusnosti pri zápise dát, z dôvodu aby sme vylúčili možné úzke hrdlo v jeho prepojení s databázovým systémom HBase. Pre účely testovania sme použili verziu Hadoop-0.20.2, veľkosť haldy pre JVM (parameter -Xmx, JVM Heap) sme nastavili na 1GB.

Meranie maximálnej rýchlosi zápisu sme testovali v troch konfiguráciach. Každá konfigurácia obsahovala jeden uzol v role master, na ktorom bežali Namenode a JobTracker. Na ostatných uzloch typu slave bežali Datanode a Tasktracker. Konfigurácia klastrov bola následovná:

- A - tri uzly slave s faktorom replikácie jedna
- B - tri uzly slave s faktorom replikacie tri
- C - šesť uzlov slave s faktorom replikácie tri

Počas testu sme na súborový systém zapisovali tri rôzne veľkosti súborov, kde v HDFS bola zachovaná štandardná veľkosť bloku tj. 64MB. Každý test bol vykonnaný trikrát a výsledná hodnota bola určená ako aritmetický priemer. Výsledky testu, ktoré zobrazuje tabuľka 7.1 ukazujú, že zvýšenie faktoru replikácie ma zásadný vplyv na celkový výkon. Dôležitý fakt, ktorý vyplynul z výsledkov testovania je, že v prípade ak zvýšime dvojnásobne počet uzlov v klastri (prípad B, C) jeho výkonnosť vzrastie lineárne, čo potvrdzuje vysokú škálovateľnosť daného systému. Veľkosti zapisovaných súborov sme volili s ohľadom na konfiguráciu systému HBase, kde veľkosť súborov Memstore, ktoré sa budú zapisovať z operačnej pamati na HDFS je 128MB.

Klaster			
Súbor	A	B	C
128 MB	287 MB/s	102 MB/s	190 MB/s
812 MB	371 MB/s	85 MB/s	162 MB/s
4 GB	433 MB/s	85 MB/s	163 MB/s

Tabuľka 7.1: Výkonnosť HDFS pre zápis dát

vytazenie CPU 30%

Popisat detailne test?

7.3 HBase

7.4 Cassandra

There's no clear test plan for distributed systems, no failover benchmarking tool, nothing on reliability, availability or data consistenc

Počet uzlov				
Veľkost riadku	3	4	5	6
1 KB	18182	28397	31132	31847 ???
10 KB	6473	7857	9574	12120
100 KB	726	941	1137	1374
512 KB	134	173	218	253
1 MB	62	92	100	126

Tabuľka 7.2: Počet zapísaných riadkov s jedným stĺpcom za sekundu

Počet uzlov	Replikácia	Konzistencia	Čas	Riadok/sek	Priepustnosť
1	1	One	132	22805	22
3	1	One	67	41074	44
3	3	One	115	26176	26
3	3	Quorum	165	18182	17
6	3	One	87	34483	34
6	3	Quorum	103	29126	28

Tabuľka 7.3: Zápis riadkov o veľkosti 1000 B

Počet uzlov	Replikácia	Konzistencia	Čas	Riadok/sek	Priepustnosť
1	1	One	419	261	261
3	1	One	104	971	971
3	3	One	58	1741	1741
3	3	Quorum	327	307	307
6	3	One	23	4555	4555
6	3	Quorum	66	1515	1515

Tabuľka 7.4: Čítanie riadkov o veľkosti 1000 B

TODO: do slovnik slov = def. klaster Commodity hw = komponenty sú štandardizované, bežne dostupné a ich cenu neurčuje samostatný výrobca (IBM, DELL,...) ale trh. Príkladom takéhoto HW môže byť nasledujúca konfigurácia 2 x 250GB SATA drives, 4-12GB RAM, dual x dual core CPU's. Snahou je nájsť vhodný balanc medzi cenou a výkonom.

=== Porovnanie vykonnosti systemov

nastroj na naplnenie systemov

meranie

write - read

Kapitola 8

Návrh systému

V tejto kapitole popíšeme návrh systému, ktorý bude slúžiť na archiváciu emailov a spĺňať požiadavky, ktoré sme pre tento systém definovali. V prvej časti sa zameriame na výber vhodných open source nástrojov pre implementáciu prototypu a následne popíšeme dosiahnuté výsledky v testovacom prostredí, ktoré preukážu vhodnosť využitia NoSQL systému Cassandra pre riešenie tejto úlohy.

8.1 Zdroj dát

Základným prvkom, ktorý budeme v našom systéme archivovať je emailový objekt, ktorý definuje dokument RFC 2821 []. Tento objekt pozostáva z SMTP obálky a samotnej emailovej správy. Obálka obsahuje informácie, ktoré sú potrebné pre korektné doručenie správy pomocou emailového servera a patria tam napríklad odosielateľ emailového objektu a jeden alebo viacerý príjemcovia. Emailová správa predstavuje semištrukturovaný dokument [16] v textovej podobe, ktorého syntax popisuje štandard RFC 2822 [1] z roku 2001 pod názvom Formát Internetovej správy (angl. Internet Message Format). Dokument RFC 2822 nahradzuje a upravuje pôvodné RFC 822 pod názvom Štandard pre formát Internetových textových správ ARPA z roku 1982 (angl. Standard for the Fromat of ARPA Internet Text Messages). Obsah emailovej správy delíme na hlavičku a telo, ktoré sú od seba oddelené znakom reprezentujúcim prázdny riadok. Telo správy nie je povinné. Štruktúru tela správy a polia v hlavičke rožšírujú štandardy, pod názvom MIME (ang. Multipurpose Internet Mail Extensions), RFC 2045, RFC 2046, RFC 2047, RFC 2048 a RFC 2049 []. Tieto štandardy pridávajú možnosť použitia iných znakových sád ako US-ASCII, ďalej umožnujú štruktúrovať telo správy (vnorené správy rfc822), definujú formát a typy pre zasielanie príloh atď.

Zber emailových objektov je realizovaný na unixových serveroch, ktoré budú používať emailový server QMAIL[1]. Tento server bude zároveň realizovať antispamovú kontrolu kontrolu pomocou modulu Qmail-scanner[2], ktorý je naprogramovaný v jazyku Perl[3]. Po doručení emailového objektu na server je emailový objekt spracovaný QMAIL-om, ktorý volá modul

[1] http://cr.yp.to/qmail.html
[2] http://qmail-scanner.sourceforge.net
[3] http://www.perl.org

qmail-scanner a následne dokončí obsluhu doručenia. Tento modul sme vhodne modifikovali pre potreby nášho systému. Modifikovaný súbor je súčasťou zdrojových kódov tejto práce. Výstupom je dvojica súborov a to obálka, ktorá obsahuje viacero štatistických údajov a samotný textový súbor reprezentujúci emailovú správu v pôvodnej podobe. Príklad obálky znázorňuje obrázok 8.1. Detailný popis tejto štruktúry sa nachádza na webovej adrese http://qmail-scanner.sourceforge.net/.

```
Tue, 15 Mar 2011 10:12:09 CET Clear:RC:1(88.208.65.55):SA:1 0.007811 9508
odosielatel@server prinemca@server2 predmet <13001802281039145 46@aq>
1300180329.16836-0.forid1:5987 priloha1:134
```

Obr. 8.1: Obsah obálky z programu qmail-scanner

8.2 Analýza dát

Jedný z hlavých požiadavkov systému je deduplikácia príloh emailových správ z dôvodu úspory diskovej kapacity. Hlavička s názvom „Content-Type“, ktorú definuje RFC 2045 špecifikuje typ dát v tele MIME správy. Jej hodnota je tvorená z dvoch častí a to názov typu média (angl. media type) a bližšie špecifikovaný podtyp, napríklad „image/gif“. Norma definuje základných päť typov médií a to text, image, audio, video a application. V prípade, našej aplikácie má zmysel využiť deduplikáciu na všetky tieto typ s výnimkou typu „text/plain“, kde predpokladáme, že sa jedná o bežnú textovú správu napísanú užívateľom.

Program pre analýzu a deduplikáciu emailovej správy bol napísaný v programovacom jazyku Python[4]. Tento program dodržiaval špecifikáciu RFC 2822 a RFC2045. Medzi povinné polia hlavičky emailu patria pole „From“ a Date. Aj napriek tomu, že tieto polia sú definované už od roku 1982 v RFC 1982 analýza nášho datasetu ukázala, že XY % emailov tento požiadavok nespĺňa. Z celkovej množiny emailov o veľkosti 98000 bolo 0.022% emailov, ktoré nespĺňali štruktúru definovanú normou RFC 2045. Metódu deduplikácie sme riešili nasledujúcim spôsobom:

- analýzou emailu sme určili časti, v ktorých sa nachádzajú prílohy
- nad dátami reprezentujúcimi prílohu sme pomocou kryptografickej funkcie SHA2-256 spočítali heš (H), ktorú sme použili ako unikátny identifikátor prílohy
- dáta reprezentujúce prílohu v emaile sme nahradili značkou v tvare MARK:H
- dáta reprezentujúce prílohu sme do databáze uložili pod kľúčom H

8.3 Databázová schéma

Databázovú schému sme navrhli s ohľadom na to aké operácie nad danými dátami budeme vykonávať a taktiež sme pri návhru využili poznatky získané štúdiom architektúry tejto databáze. Schéma je tvorená pomocou štyroch Column families a to:

[4] http://python.org

- messagesMetaData obsahuje meta informácie identifikované v obálke emailového objektu a emailovej správy, nad ktorými budeme vykonávať štatistické výpočty pomocou metódy Mapreduce
- messagesContent obsahuje obálku, hlavičku a telo správy
- messagesAttachment slúži na ukladanie deduplikovaných príloh emailov
- lastInbox v chronologickom časovom poradí, podľa hodnoty Date v hlavičke emailu, zaznamenáva emaily daného užívateľa

Tradičné techniky pre popis databázových schémat nie je možné aplikovať na databázové systémy ako napríklad Bigtable alebo Dynamo. Jedným z dôvodom je, že na tieto schémy sa aplikuje denormalizácia, duplikácia dát a kľúče sú často komplexného charakteru. Dodnes neexistuje, žiadny štandard, ktorý by definoval popis týchto schémat. Článok pod názvom techniky pre definíciu štruktúr pomocou diagramov v cloude a návrhové vzory (angl. Cloud data structure diagramming techniques and design patterns [5]) navrhuje stereotypy pre diagramy v jazyku UML a obsahuje vzory pre popis štruktúry týchto dát. Obrázok 8.2 znázorňujúci databázovú schému nášho modelu aplikuje tieto techniky.

Ako jedinečný identifikátor emailovej správy v databáze využivame nasledujúcu schému:

```
emailID = sha256(uid + MessageId + date)
```

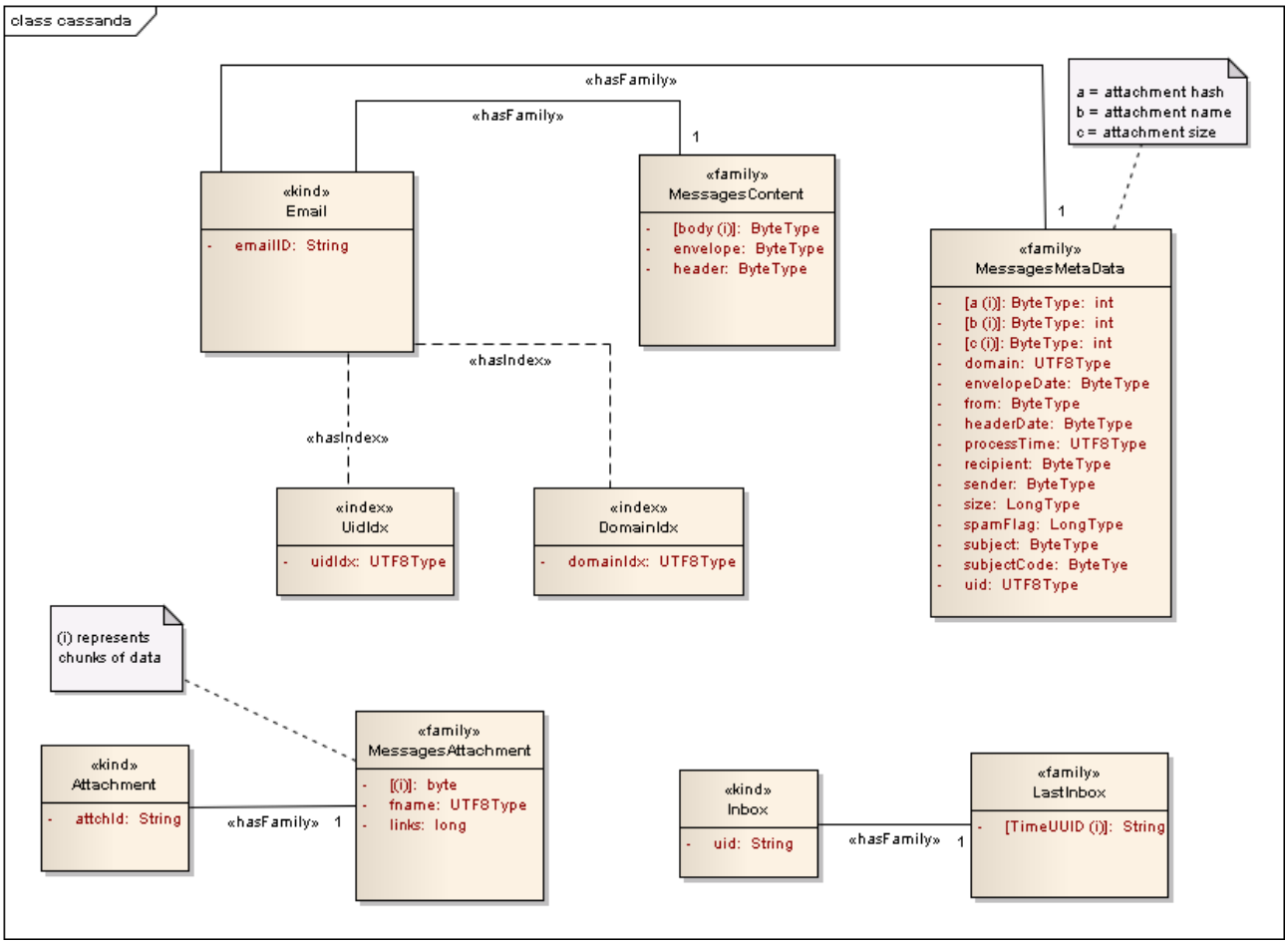
V tejto schéme reprezentuje:

- uid emailovú adresu príjemcu, v tvare jan@mak.com
- MessageId je ide identifikátor z hlavičky daného emailu
- date je časová značka reprezentujúca čas kedy bol email prijatý emailovým serverom (formát: rok, mesiac, deň, hodina, minúta, sekunda)
- emailID je heš funkcie SHA2 v hexadecimálnom tvare

V predchádzajúcej kapitole sme zistili, že Cassandra nie je optimalizovaná pre zápis blokov dát (ang. blob), ktorých veľkosť prevyšuje 5MB avšak optimálne výsledky pre zápis dosahuje pri veľkosti blokov 512KB. Preto prílohy a samotné dáta emailu v prípade, že ich veľkosť presahuje 1MB zapisujeme do samostatných stĺpcov o veľkosti 512KB na aplikačnej úrovni. Názvy stĺpcov číslujeme vzostupne v rozmedzí 0 až n. Spätnú rekoštrukciu pri načitaní dát je potrebné zabezpečiť taktiež pomocou klienta.

8.4 Fultextové vyhľadávanie

Fultextové vyhľadávanie realizujeme pomocou samostatného NoSQL systému Elasticsearch. Archív reprezentujeme pomocou jedného indexu s názvom emailArchive, ktorý obsahuje dva typy s názvom email a envelope. Schéma týchto typov obsahuje polia poďla, ktorých chceme v emailov vyhľadávať a jej reprezentáciu zapísanú vo formáte JSON znázorňuje obrázok 8.3.



Obr. 8.2: Databázová schéma

8.5 Implementácia

V programovacom jazyku Python sme implementovali klienta pre zápis dát do databáze Cassandra a ElasticSearch. Jednou z najdoležitejších vlastností týchto klientov je voľba úrovne konzistencie pri zápise. Našou prioritou je integrita dát a od databáze požadujeme silnú konzistenciu. Zvolili sme úroveň QUORUM, ktorá zabezpečí zápis dát na N / 2 + 1 replík a klient následne obdrží potvrdenie o úspešnosti zápisu, inak zápis opakujeme. Klient, ktorý slúži na čítanie dát z databáze využíva taktiež úroveň QUORUM. Tieto vlastnosti nám zabezpečujú, v prípade použitia faktoru replikácie tri (dáta sa v databázovom systéme nachádzajú trikrát), silnú úroveň konzistencie na strane klienta a databáze. Analýza emailovej správy a jej deduplikácia spotrebúva hlavne CPU zdroje. Moderné procesory obsahuju viacero jadier, tento fakt môžeme využiť pre paralelizované spracúvanie emailov, teda každé jadro CPU bude spracúvať súčasne jednu emailovú správu.

```
mappingsEmail = {
  "inbox": {"type": "string"},
  "from": {"type": "string"},
  "subject": {"type": "string"},
  "date" :{"type": "date"},
  "messageID" :{"type": "string", "index": "not_analyzed"},
  "attachments":{"type": "string"},
  "size": {"type": "long", "index": "not_analyzed"},
  "body": {"type": "string"}
}

mappingsEnvelope = {
  "sender": {"type": "string"},
  "recipient": {"type": "string"},
  "ip": {"type": "ip"},
  "date": {"type": "date"}
}
```

Obr. 8.3: JSON schéma pre fultextové vyhľadávanie

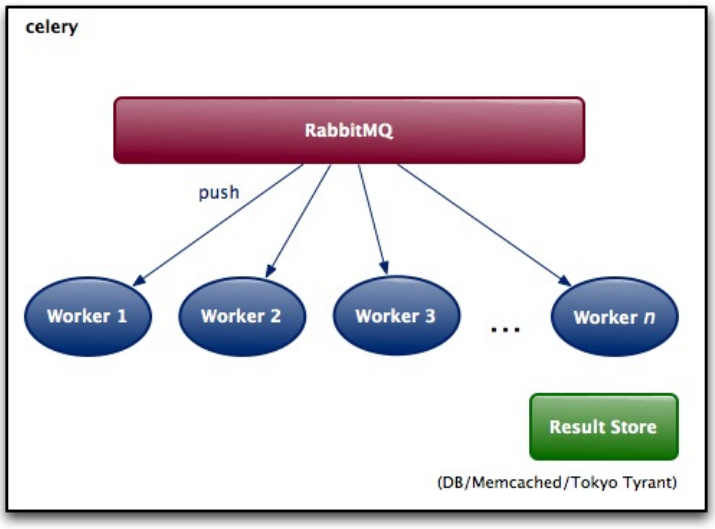
Celery

Paralelizáciu našej aplikácie sme zabezpečili pomocou využitia asynchrónnej fronty úloh pod názvom Celery[5], ktorá využíva architektúru distribuovaného predávania správ (ang. distributed message passing). Architektúru znázorňuje obrázok 8.4. „Pracovníci“ (angl. workers) reprezentujú samostatné procesy v našom prípade proces pre analýzu a deduplikáciu emailu, ktoré môžu bežať paralelne. Broker v našom prípade aplikácia RabbitMQ[6] obdrží správu, ktorá sa uloží do fronty. Táto správa je následne zaslaná ľubovoľnému pracovníkovy (v našom prípade proces pre analýzu a deduplikáciu emailu), ktorý ju spracuje. Táto architektúra je plne distribuovaná, dokáže odolávať chybám (napr. v prípade výpadku elektrickej energie správy nadalej pretrvávajú vo fronte).

8.5.1 Klient

Proces spracovania nového emailu je znázornený pomocou sekvenčného diagramu8.5. Pri príchode nového emailu, ktorý je spracovaný emailovým serverom Qmail, je vytvorená nová úloha pomocou aplikácie Celery. Táto úloha uloží do brokera identifikátor emailu, v našom prípade cesta k súboru, ktorý reprezentuje emailovú spravú. V prípade, že je v daný okamžik k dispozícií ľubovoľný pracovník, je emailová správa spracovaná pomocou nášho analyzátora a následne zapísaná do databáze Cassandra a fultextového systému Elasticsearch. Na testovacie účely sme nemali k dispozícii reálny dátový tok emailov. Vrstvu reprezentujúcu Qmail sme nahradili modulom, vytvájúcim nové úlohy prechádzaním lokálneho súborového systému, ktorý obsahoval testovaciu množinu emailový správ.

[5] http://celeryproject.org
[6] http://www.rabbitmq.com



Obr. 8.4: Architektúra Celery, zdroj: http://ask.github.com/celery/getting-started/introduction.html

Návrh realizácie deployment diagram (oddelene datacentra - na tretej replikacii pocitat MR)ty?

8.5.2 Výpočet štatistík

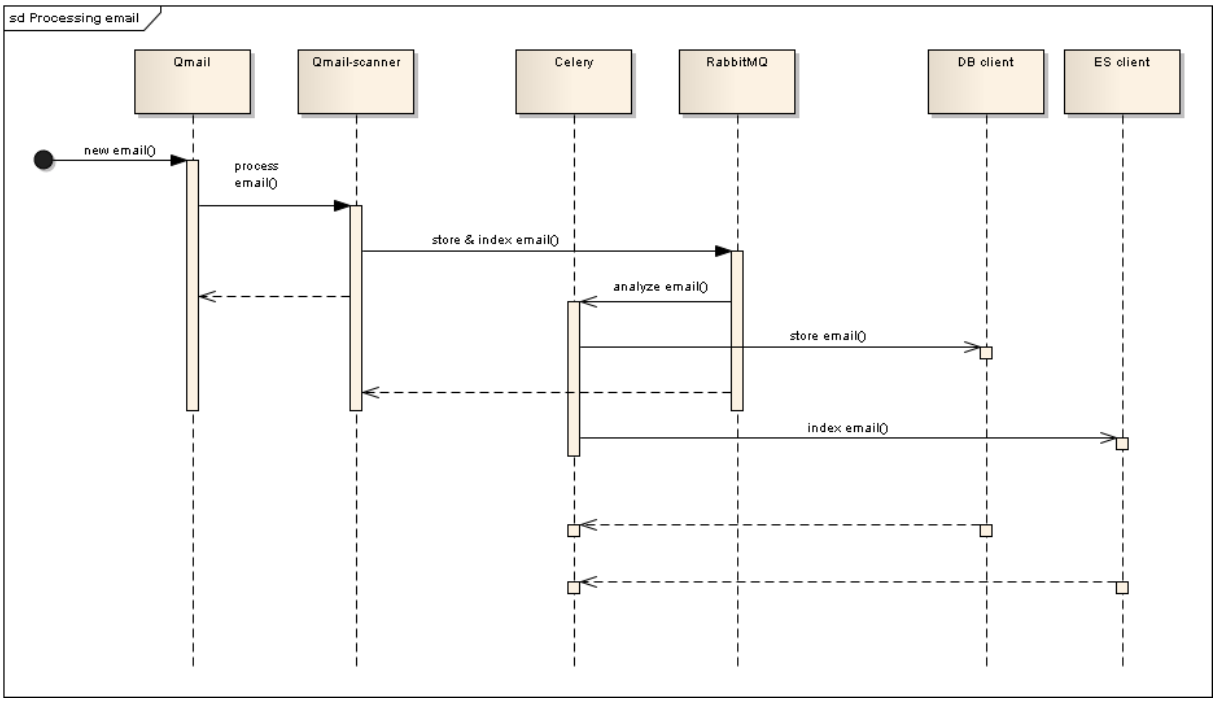
Cassandra podporuje spoluprácu so systémom Hadoop, čo nám dáva do rúk mocný nástroj na masívne paralelné spracovanie dát pomocou techniky Mapreduce. Samotné písanie aplikácii v je náročné a okrem toho programový model Mapreduce obsahuje viaceo problémov. Model napríklad neobsahuje primitíva na filtrovanie, agregáciu, join (je potrebná ich vlastná implementácia) a nepodporuje možnosť definície viackrokového dátového toku [8]. Tieto nedostatky rieši nástroj Pig vďaka, ktorému sme boli schopný spracúvať meta dáta uložené v databáze. Obrázok 8.6 zobrazuje programovú ukážku, ktorá slúži na výpočet najvačsieho emailu pre každú doménu, pre jednoduchosť sme vynechali časti, ktoré slúžia na načítanie dát z databáze a obsahujú uživateľsky definovanú funkciu v programovacom jazyku Java, ktorá slúži na predspracovanie vstupných dát do vhodného formátu. Podpora uživateľom definovaných funkcií je jednou z ďalších výhod nástroja Pig.

8.5.3 Webové rozhranie

TODO web IFACE – pristup koncovych uservo k archivu a diskusia ohladne bezpecnosti

8.6 Overenie návrhu

Pomocou vyšie popísaného návrhu a implementovaných nástrojov sme overili funkčnosť nami navrhovaného modelu. Vhodná voľba daných verzií u aplikácií Celery a RabbiMQ vyplynula



Obr. 8.5: Spracovanie emailu

```
notSpam = FILTER grp BY group.spam == 1;
maxSize = foreach grp {
  size = rows.size;
  generate group, MAX(size);
};
STORE maxSize into 'biggestEmailPerDomainDomain' using PigStorage(',');
```

Obr. 8.6: Programová ukážka v jazyku Pig

počas písania a ladenia samotnej aplikácie. Všetky tieto aplikácie sú neustále vo vývoji, to isté platí pre databázu Cassandra a systém Hadoop. Počas písania tejto práce prebehlo viacero rozhovorov so samotnými autormi týchto aplikácii. Konkrétne databáza Cassandra na začiatku práce neobsahovala takmer žiadnu ucelenú dokumentáciu, počas začiatkov experimentov sme začínali s verziou 0.7.0. Počas ukončovania tejto práce je aktuálna verzia 0.7.5 a medzitým vznikala kvalitná dokumentácia od spoločnosti Datastax[7].

Konfigurácia

Hardverová konfigurácia obsahovala 9 serverov, kde konfigurácia každého servera bola totožná a obsahovala štvorjádrový procesor o frekvencii 2Ghz, 4 GB RAM, pevné disky 5x1TB (RAID0) 7200 RPM SATA. Všetky serveri obsahovali inštaláciu Linux Debian Lenny x64,

[7] https://datastax.com

Sun Java 1.6.0_+88. Na šiestich serveroch bola nainštalovaná databáza Cassandra 0.7.3, Hadoop 0.20.2, dvojica serverov obsahovala klientskú aplikáciu, Celery 2.6 a posledný server obsahoval inštaláciu RabbitMQ 2.1.1.

Overenie integrity dát

Databázový kluster sme naplnili testovacími dátami obsahujúcimi emaily o objeme 300GB. Následne sme nasimulovali prípad obnovy dát z archívu, kde sme všetky emaily v náhodnom poradí z databázy načítali, zostavili ich do pôvodného tvaru (klientskou aplikáciou) a porovnali sme ich odtlačok pomocou hešovacej funkcie MD5 s odtlačkom pôvodných dáta. Tento test prebehol bez akejkoľvek chyby.

Pozorovanie

Zaujímavým pozorovaním bol samotný fakt, že z celkového objemu emailových správ 300GB sa po deduplikácií príloh tento objem znížil na 99GB, teda došlo k 30% úspore diskovej kapacity. Štruktúry do ktorých sme ukladali dáta pre potrebu štatistík zaberali 0,4% z celkového objemu dát, čo je zanedbateľná položka.

Kapitola 9

Záver

mapreduce - PIG popis vysledkov kolko tvorili indexy ES rychlost

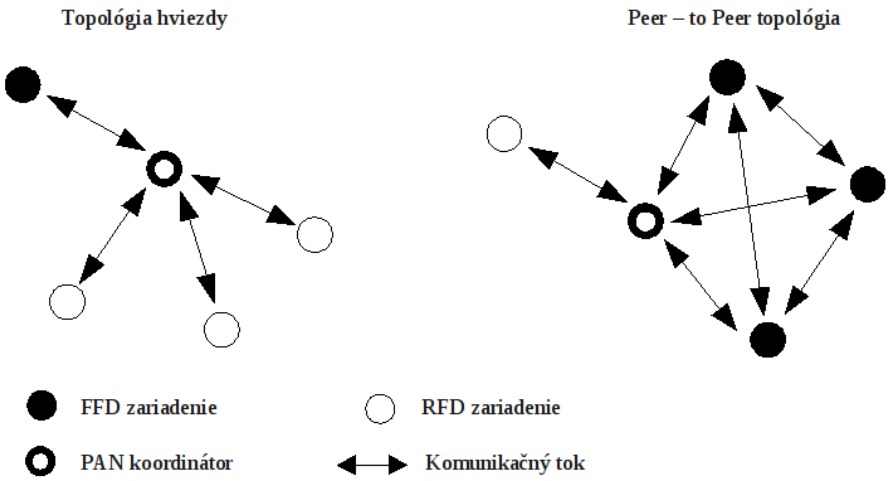
Kapitola 10

ZMAZAT

10.1 Komponenty IEEE 802.15.4

Zariadenia delíme na dva druhy a to zariadenie poskytujúce úplnú funkčnosť FFD (Full-function device) a redukované zariadenia RFD (Reduced-function device). Zariadenie FFD môže operovať v troch módoch, ktorými sú PAN (Personal area network) koordinátor, koordinátor, alebo koncové zriadenie. Zariadenie FFD ďalej dokáže komunikovať so zariadeniami typu RFD alebo FFD a zariadenie RFD je schopné komunikácie len so zariadením FFD. Výhodou RFD zariadení, je že neposielajú veľké objemy dát a teda na ich realizáciu je potrebná minimálna pamäťová kapacita. Samotná WPAN je tvorená dvoma alebo viacerými zariadeniami, ktoré komunikujú na tom istom fyzickom kanály.

10.2 Sieťová topológia



Obr. 10.1: Topológie štandardu 802.15.4

10.2.1 Technické parametre

Modul popisujú následujúce parametre[**?**].

Dosah vnútri	30m
Dosah vonku	90m
Sila výstupného signálu	1mW (0 dBm)
Rýchlosť prenosu dát	250 000 bps
Rýchlosť sériového rozhrania	1200 bps - 250 kbps
Citlivosť pri príjme	-92 dBm

Tabuľka 10.1: Špecifikácia výkonu a rýchlosti

Frekvenčné pásmo	ISM 2.4 GHz
Rozmery	2.438cm x 2.761cm
Operačná teplota	-40 až 85C

Tabuľka 10.2: Obecné parametre

Podporované sieťové technológie	Point-to-point, Point-to-multipoint, Peer-to-peer
Počet kanálov	16
Anténa	Whip
Adresácia	PAN ID

Tabuľka 10.3: Sieťové parametre

Kapitola 11

Teória antén

Anténa je zariadenie, ktoré slúži na vysielanie a príjem rádiových signálov. Toto zariadenie konvertuje elektromagnetické vlny na elektrickú energiu a opačne. Podľa toho ako sú signály vysielané ich delíme na všesmerové (vysielanie vo všetkých smeroch) a smerové (vysielajú len v danom smere). Medzi chovaním sa vysielacej a prijímacej antény nepozorujeme žiadne rozdiely.

Následujúce parametre slúžia na popis základných vlastností antén:

- smerovosť, určuje v akom smere sú elektromagnetické vlny vysielané. Je posudzovaná na základe vyžarovaných charakteristík, ktoré delíme na vertikálne a horizontálne. Meria sa pomocou parametrov zisk antény a vyžarovací uhol.
- vyžarovací uhol
- impedancia antény
- zisk antény
- frekvenčná šírka prenášaného pásma
- polarizácia
- účinnosť

H rovina, je rovina v ktorej sa šíri vektor magnetického poľa a sleduje sa v nej ako sa mení intenzita elektrického poľa. Naopak v E rovine sa šíri vektor elektrického poľa a sleduje sa zmena intenzity magnetického poľa.

11.0.2 Definícia pojmov

Účinnosť je pomer vyžarovaného výkonu k výkonu, ktorý privádzame na vstup antény.

Zisk určuje mieru smerovosti antény. Definujeme ho ako pomer vyžarovanej intenzity antény v danom smere k intenzite, ktorá je vyprodukovaná ideálnou anténou vyžarujúcou do všetkých smerov rovnomerne, bez strát a obe antény majú na vstupe rovnaký výkon. Zisk berie v úvahu

okrem smerovosti aj účinnosť antény. Pre zisk ďalej platí, ak ma anténa pre daný smer väčší zisk ako je celkový zisk antény, tak v nejakom inom smere musí byť zasa zisk menší aby bola zachovaná celková energia. Tohoto faktu si je možno všimnúť u grafov popisujúcich zisk antén, ktoré boli vytvorené meraním v anténnej komore viď obrázok **??**.

11.0.2.1 Pojmy

- dB - je skratka pre decibel. Je to matematické vyjadrenie používané na zobrazenie závislosti medzi dvoma hodnotami.

- Rádiofrekvenčný výkon - je buď výkon vysielača alebo prijímača vyjadrený vo Wattoch. Taktiež môže byť vyjadrený v dBm. Vzťah medzi dBm a Wattmi je vyjadrený nasledovne $P_{dBm} = 10 * \log P_{mW}$

- Zoslabenie signálu modeluje nasledujúci obrázok **??**. Kde Pin je vstupný výkon a Pout je hodnota výstupného výkonu.

  Zoslabenie je vyjadrené v dB podľa nasledujúceho vzťahu: $P_{dB} = 10 * \log(Pout/Pin)$. Napríklad predpokladajme, že dôjde k strate 1/3 vysielaného signálu (Pout/Pin = 2/3), potom hodnota zoslabenia v dB je 10 * log (2/3) = -4.05 dB

- Citlivosť prijímača je minimálna hodnota výkonu radio frekvenčného signálu potrebná na vstupe prijímača, aby bol signál ďalej spracovaný.

- Stratovosť je oslabenie výkonu radiofrekvenčneho signálu, ktorý je šírený v priestore. Je vyjadrená v dB a ďalej závisí na vzdialenosti medzi vysielacou a prijímacou anténou, na viditeľnosti medzi vysielacou a prijímacou anténou a na veľkosti antén.

11.0.3 Typy antén

Izotropická anténa sa používa pre teoretické účely, vysielané vlny majú rovnaké parametre, ktoré popisujú anténu vo všetkých smeroch. Používa sa hlavne pri popisovaní a porovnávaní vlastností reálnych antén.

Horn anténa sa používa v situácia, kde je potrebné dosiahnuť vysokého zisku, vlnová dĺžka je krátka, môže byť širokopásmová alebo úzko-pásmová, čo záleží na jej tvare. Taktiež dokáže pracovať s akoukoľvek frekvenciou. Keďže charakteristiky tejto antény sú známe a dobre matematicky popísané používa sa táto anténa ku kalibrácii iných systémov, tento fakt bol využitý aj počas meraní v anténnej komore. Hlavným parametrom, ktorý bol podstatný u meraní s touto anténou je jej zisk.

Whip anténa, je model antény, ktorý používajú ZigBee zariadenia, ktoré som simuloval. Anténa je poväčšine vertikálna a u ZigBee zariadení upevnená na doštičke plošného spoja. Je to anténa, ktorá vysiela horizontálne do všetkých smerov a hluché zóny sú vertikálne v bode upevnenia a ukončenia.

11.0.4 Stratovosť voľného priestoru (Free-space path loss)

Stratovosť signálu vo voľnom priestore sa používa k predikcii sily rádiového signálu. Napriek tomu, že nemodeluje dôveryhodne realitu obsahujúcu prekážky, odrazy atď, má veľký význam pre základné pochopenie šírenia sa signálu v reálnych podmienkach. Využíva sa taktiež pri tvorbe simulačných modelov a pri vývoji v oblasti bezdrôtových systémov.

Definujeme ju ako stratu sily signálu elektromagnetickej vlny, ktorá vzniká medzi dvoma priamo viditeľnými bodmi vo voľnom priestore, kde nie sú žiadne prekážky, odrazy a nedochádza k ohybu vĺn.

Formula pre výpočet stratovosti je nasledovná:

$$FSPL = \left(\frac{4\pi d}{\lambda}\right)^2 = \left(\frac{4\pi df}{c}\right)^2,$$

kde:

- λ je vlnová dĺžka (m)
- f je frekvencia signálu (Hz)
- d je vzdialenosť od vysielača (m)
- c je rýchlosť svetla vo vákuu 2.99792458 * 10^8 m/s

Daná formula vyjadrená v dB:

$$\begin{aligned} FSPL(dB) &= 10\log_{10}\left(\frac{4\pi df}{c}\right)^2 \\ &= 20\log_{10}\left(\frac{4\pi df}{c}\right) \\ &= 20\log_{10}(d) + 20\log_{10}(f) + 20\log_{10}\left(\frac{4\pi}{c}\right) \end{aligned}$$

Kapitola 12

Analýza a návrh riešenia

12.0.4.1 Výkonová závislosť

Následujúci obrázok 12.1 zobrazuje vysielač (T), u ktorého vystupuje dvojica výkonov P_1, P_2 a hodnota zisku pre daný smer $G_T(\alpha_T)$, ďalej u prijímača (R) vystupujú výkony P_3, P_4 a zisk v danom smere $G_R(\alpha_R)$. V simulácii bude zohrávať hlavnú úlohu hodnota výkonu P_4, hodnota výkonu P_1 je pred vyslaním rámcu známa, je to hodnota špecifikovaná výrobcom zariadenia. V následujúcej časti odvodím vzťah pomocou, ktorého určím hodnotu výkonu P_4, na základe ktorej sa prijímač rozhodne či sa naozaj jedná o prijímané dáta alebo sa vysielaný signál zoslabil na takú úroveň, kedy bude považovaný za šum na kanály.

Nasleduje formula pre výpočet hodnoty výkonu P_4, pomocou FSPL:

$$\left[\frac{P_3}{P_2}\right]_W = \left(\frac{\lambda}{4\pi}\right)^2 \frac{1}{d^\alpha}$$

$$\begin{aligned}\left[\frac{P_3}{P_2}\right]_{dB} &= 10\log_{10}\left(\frac{\lambda}{4\pi}\right)^2 \frac{1}{d^\alpha} \\ &= 10\log_{10}\left(\frac{c}{4\pi f}\right)^2 \frac{1}{d^\alpha} \\ &= 20\log_{10}\left(\frac{c}{4\pi f}\right) - 10\alpha\log_{10} d \\ &= -40.2251 - 10\alpha\log_{10} d,\end{aligned}$$

kde:

- λ je vlnová dĺžka (m), $\lambda = \frac{c}{f}$
- f je frekvencia signálu (Hz), pre použité zariadenia XBee f = 2450 Mhz
- d je vzdialenosť od vysielača (m)
- c je rýchlosť svetla vo vákuu 2.99792458 * 10^8 m/s

- α je koeficient útlmu prostredia (pre vzduch $\alpha = 2$)

Zavediem nasledujúce označenie:

$$L = -40.2251 - 10\alpha \log_{10} d$$

Ďalej platí:

$$P_2 = P_1 + G_T(\alpha_T)$$

$$P_3 = P_2 + L$$

$$P_4 = P_3 + G_R(\alpha_R),$$

z čoho následne vyplýva následujúca rovnosť:

$$P_4 = P_1 + G_T(\alpha_T) + G_R(\alpha_R) + L + K,$$

kde:

- P_1 - výkon privedený do antény vysielača (dBm)
- P_2 - výkon vysielaný vysielacou anténou (dBm)
- P_3 - hodnota výkonu prijatého prijímacou anténou (dBm)
- P_4 - výkon vystupujúci z káblu prijímacej antény a vstupujúci do prijímača (dBm)
- $G_T(\alpha_T)$ - zisk vysielacej antény v danom smere (dBi)
- $G_R(\alpha_R)$ - zisk prijímacej antény v danom smere (dBi)
- K - konštanta, ktorá spôsobuje ďalšie straty existujúce v reálnom prostredí (odrazy vo vodičoch, konektoroch, atď.)

Taktiež zároveň platí:

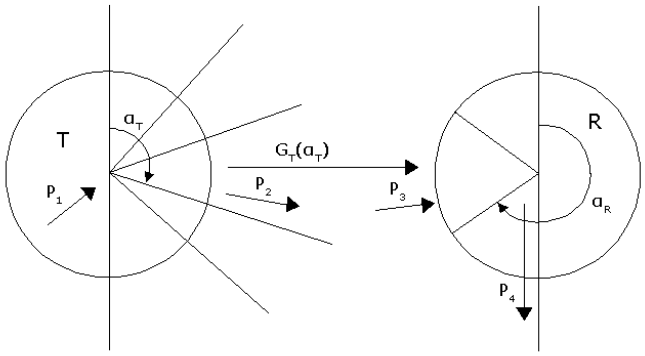
$$P_4^{'} < P_4^{''},$$

kde:

- $P_4^{'}$ - hodnota výkonu v reálnom prostredí (W)
- $P_4^{''}$ - spočítaná hodnota výkonu P_4 (W)

$$[P_3]_W < [P_2]_W => \left[\frac{P_3}{P_2}\right]_{dB} < 0.$$

Daný vzťah uvažuje straty, ktoré vznikajú na konektoroch antén, prepojovacím káblom medzi anténou a zariadením, vlastným odporom vodiča antény a iné. v podobe konštanty K.



Obr. 12.1: Popis výkonov a zisku u antén pre vysielač a prijímač

12.0.4.2 Štruktúra modulu NIC

NIC (Network card interface) je časť sieťového adaptéru zodpovedná za fyzický prístup k médiu a adresácii na úrovni MAC vrstvy, popisuje ju fyzická a linková vrstva ISO-OSI modelu. Štruktúra tohoto modulu je znázornená na obrázku 12.2. V mojom prípade je teda fyzická vrstva tvorená modulom *snrEval*, *decider* a vrstvu MAC tvorí modul *mac*. Vzájomná úzka kooperácia medzi týmito vrstvami je dôvodom prečo sú zapuzdrené v jednom module. Všetky submoduly modulu NIC, sú zdedené z triedy *ChannelAcces*, ktorá je ďalej odvodená z triedy *BasicModule*. Táto trieda poskytuje funkcionalitu umožňujúcu komunikáciu jednotlivých staníc, ktoré sú vo vzájomnom dosahu. Na úrovni fyzickej vrstvy ma hlavne zaujíma modelovanie oslabenia signálu a výpočet chybovosti na kanále. Tým, že je fyzická vrstva tvorená osobitne modulom *snrEval*, som schopný modelovať výpočet stratovosti na kanále pomocou rôznych metód a na základe výsledku sa v module *decider* rozhodnem pomocou akého kritéria tieto dáta vyhodnotím. Napríklad modul *decider* bude rozhodovať o tom či dané dáta príjme na základe porovnania hodnoty SNR, spočítanej z modulu *snrEval*, s definovanou hraničnou hodnotou, alebo sa môže rozhodovať na základe počítania pomocou formúl pre výpočet chybovosti na kanále (napr. BER). Vďaka tomu môžem tieto moduly navzájom rôzne kombinovať. V následujúcej časti detailnejšie priblížim štruktúru modulov *snrEval* a *decider*.

Modul snrEval

Tento modul zabezpečuje príjem a vysielanie dát na kanál. Ďalej vytvára správy typu *AirFrame* z *MacFramu* a opačne, počas toho ako odpočúva kanál zároveň mení stavy rádia, ktoré sú reprezentované stavovým automatom a taktiež zabezpečuje simuláciu oneskorenia vo vysielaní alebo príjme pomocou pomocných funkcií (*bufferMsg, unbuefferMsg*). V mojom modele ma zaujímala jedna z jeho ďalších vlastností a to ukladanie a spracúvanie SNR hodnôt pri prijímaní rámcov. Rámec fyzickej vrstvy (*AirFrame*) obsahuje pomocnú štruktúru *SnrList*, ktorú reprezentuje štruktúra *List* programovacieho jazyka C++ a záznam tejto štruktúry obsahuje dve položky a to časovú značku prijatia rámcu a k nej odpovedajúcu hodnotu SNR. V mojom prípade je hodnota SNR spočítaná na základe vzťahu [hodnota

výkonu vstupujúca do prijímača (P_4) / hodnota šumu na kanále], kde hodnotu P_4 počítam za využitia modifikovanej formule FSPL. Túto hodnotu následne predávam do modulu *Decider*, kde je ďalej spracovaná.

Práca modulu snrEval je znázornená vývojovým diagramom na obrázku 12.3. Keď sa nachádza modul v stave SYNC prijíma správu. Pred jej prijatím sa najskôr vykoná kontrola, či nedošlo k poškodeniu SFD (Start frame delimiter), následne sa spracuje zvyšok správy a spočíta sa hodnota SNR. V prípade, že sa modul nenachádza v stave SYNC a obdrží ďalšiu správu je tato správa považovaná za šum, hodnota šumu sa zvýši o hodnotu výkonu, ktorým bola táto správa prijatá a spočíta sa nová hodnota SNR, ku ktorej je pripojená časová značka. Detailnejšie sa budem zaoberať modelom kolízie v následujúcej kapitole.

V tomto module je metóda *handleLowerMsg* rozdelená na dve časti a to:

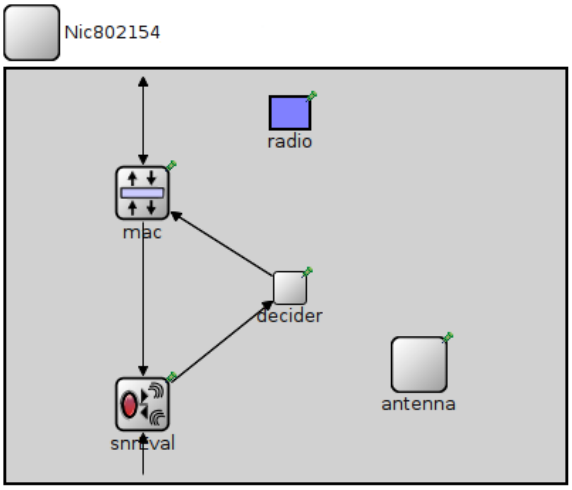
- *handleLowerMsgStart* - volá sa hneď po prijatí správy, volá metódy na výpočet hodnoty prijatého výkonu (P_4) a následne predáva spracovanie ďalším metódam na základe stavu rádia
- *handleLowerMsgEnd* - slúži na samotné odoslanie správy vyššej vrstve a zároveň pripája *SnrList* ako parameter.

Modul decider

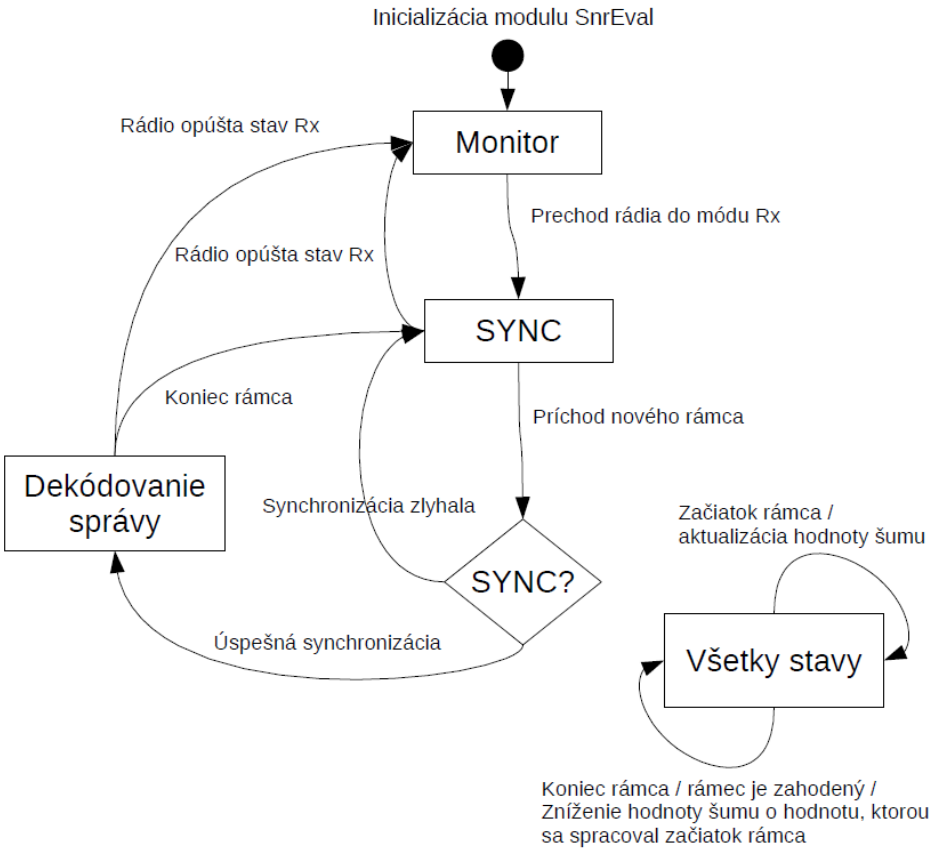
Modul spracúva len správy, ktoré prichádzajú z kanálu cez modul *SnrEval*. Správy z vyšších vrstiev, ktoré sa posielajú na kanál neprechádzajú týmto modulom a to z dôvodu, že tento modul len rozhoduje, či sa daná správa zahodí alebo prepošle vyššej vrstve. Rozhodovanie je založené na základe výpočtov ako napríklad chybovosť bitov alebo sa rozhoduje či sa má správa zahodiť na základe vysokej hodnoty šumu na kanále porovnaním s hraničnou hodnotou. Tieto vlastnosti však sledujem len u správ prichádzajúcich z kanálu. *Decider* teda slúži na výpočet chybných bitov (BER) v správe, čo počíta pomocou hodnôt uložených v štruktúre *SnrList*. Ďalej je v ňom možné implementovať rôzne opravné kódy. V simulácii je použitý vzorec na výpočet chybných bitov (BER) pre moduláciu MSK.

$$BER = 0.5 * exp(-0.5 * SNR)$$

Modul *mac* ďalej poskytuje funkcionalitu metódy na riadenie prístupu k médiu CSMA. Modul *radio* je centrálne zdieľaný a moduly ako *snrEval* alebo *mac* prepínajú jeho stavy (napr. RX, TX) podľa potreby. Posledným modulom, je modul *antenna* ktorý popisuje parametre antény a je využívaný modulom *snrEval*.



Obr. 12.2: Štruktúra NIC v OMNeT++



Obr. 12.3: Prechody medzi stavmi v module snrEval

Kapitola 13

Realizácia

Pre potrebu simulácie som teda ako prvý vytvoril modul *Antenna*, ktorý popisuje anténu ZigBee zariadení nasledujúcimi parametrami:

- zisk - ideálny zisk antény, ktorý nájdeme v popise antény (dBi)
- impedancia (Ω)
- config - odkazuje na xml súbor popisujúci zisk antény (*.xml)

Modul sa snaží byť čo najobecnejší pre prípadne využitie aj v iných modeloch a pridal som ho medzi ostatné moduly MF. Z predošlých parametrov je najdôležitejší parameter *config*. Pomocou tohoto parametru sa odkazujem na súbor, ktorým popisujem zisk antény pre daný uhol, v ktorom anténa prijíma alebo vysiela. Hodnoty tohoto súboru sú z meraní, ktoré boli vykonané v anténnej komore. Nasledujúce riadky zachytávajú príklad popisu konkrétnej antény.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE antenna SYSTEM "Antenna.dtd">
<antenna>
        <angle min="0" max="0" gain="0.3" />
        <angle min="0" max="30" gain="0.4" />
        <angle min="30" max="360" gain="0.9" />
<antenna>
```

Jednotlivé záznamy v elementoch <angle ...>, reprezentujú hodnotu zisku pre uhol z intervalu (min, max>. Jednotlivé intervaly musia byť zoradené vzostupne, bez prekrývania sa. V prípade, že je rámec odoslaný pod uhlom ktorý, sa v popisnom xml súbore nenachádza, je použitá hodnota ideálneho zisku, ktorá je zapísaná v .ned súbore popisujúcom štruktúru modulu antény. Táto hodnota môže byť prepísaná v konfiguračnom súbore danej simulácie omnetpp.ini pomocou nasledujúceho zápisu: sim.host[*].nic.antenna.gain = 2.1dBi

Parametru *config*, ktorý odkazuje na xml súbor je potrebné priradiť xml súbor aj v prípade ak chcem v simulácii použiť len hodnotu teoretického zisku antény. V aktuálnej verzii Omnetu nie je možné priradiť tomuto parametru napríklad hodnotu „false“ a zabezpečiť

týmto neodkazovanie sa na xml súbor. Túto funkcionalitu som ďalej do Omnetu neimplementoval, pretože po konzultácii s autorom Omnetu, som sa dozvedel, že táto možnosť pribudne v jeho novej verzii. Tento prípad riešim tak, že parametru config priradím xml súbor obsahujúci samotný koreňový element <root />, ktorý vyhodnotím pri samotnom spracúvaní xml súboru.

Samotný xml súbor popisujúci konkrétny model antény priradím danej stanici následovne: sim.host[*].nic.antenna.config = xmldoc(„antenna1.xml“)

Po spustení simulácie sa v inicializačnej časti modulu validuje daný xml súbor pomocou súboru Antenna.dtd, ďalej sa načíta do pamäti, z dôvodu, že simulátor poskytuje len DOM parsér a sprístupním si odkaz na jeho prvý element <angle ...>. Modul ďalej obsahuje metódu *findGainValue*, ktorá v danom xml súbore vyhľadá hodnotu zisku pre daný uhol. Z dôvodu optimalizácie som pre vyhľadávanie v štruktúre xml súboru použil binárne vyhľadávanie.

Ďalej som do modulu *SnrEvalRadioAccNoise3* implementoval výpočet modifikovanej formule FSPL. Samotný priestor, takzvaný playground, v ktorom sa odohráva simulácia som rozdelil na štyri kvadranty, vďaka čomu dokážem veľmi efektívne počítať uhol, pod ktorým bol rámec vyslaný z vysielacej stanice a uhol, pod ktorým bol rámec prijatý na prijímacej stanici. Tieto uhly sú prepočítané na strane príjemcu, z ich hodnôt zistím pomocou modulu antény, konkrétne hodnoty daných ziskov $G_T(\alpha_T)$ a $G_R(\alpha_R)$. Hodnotu zisku $G_R(\alpha_R)$ určím priamo pomocou metódy *faindGainValue* modulu *antenna*, ktorý obsahuje prijímač. Prijatý rámec obsahuje hodnotu jedinečného identifikátoru modulu (moduleId), z ktorého bol vyslaný. Pomocou neho sprístupním odkaz na modul vysielača a taktiež zavolám jeho metódu *findGainValue*, ktorá mi vráti hodnotu zisku $G_T(\alpha_T)$. Následne môžem spočítať hodnotu výkonu P_4, z tejto hodnoty sa ďalej spočíta hodnota SNR pomocou vzťahu SNR = P_4/[hodnota šumu prostredia], kde hodnota šumu prostredia je rovná -100dBm. SNR sa ďalej pripojí ako kontrolná informácia k rámcu a odošle o úroveň vyššie vrstve *decider*. Táto vrstva následne spočíta hodnotu BER pre daný rámec a v prípade, že nedošlo k poškodeniu rámca je tento rámec predaný opäť vyššej vrstve a to vrstve MAC.

Pre potreby vyhodnocovania modelov, som ďalej upravil modul *ChannelControl*, kde bol pridaný parameter ratio, pomocou, ktorého je prepočítavaná vzdialenosť v modeli na reálnu vzdialenosť.

Počas implementácie a ladenia modelu som objavil v produkčnom kóde MF, dve chyby, konkrétne pri výpočte hodnoty BER v module *decider*, druhá chyba bola v module *snrEval*. Po upozornení autora boli obe chyby opravené.

13.0.5 Model kolízie

Pri komunikácii ZigBee zariadení v reálnom prostredí môže dochádzať k ich vzájomnému rušeniu. Takáto situácia môže nastať napríklad v prípade, že nastane kolízia v mechanizme, ktorý riadi prístup k médiu (CSMA-CA) alebo ak máme dve siete, kde prijímač z prvej siete príme súčasne v jednom okamihu počas svojho stavu Rx, dva rámce. Súčasné prijatie dvoch rámcov na anténe zanesie do komunikácie šum (čo je vlastne prídavný signál), ktorý môže ďalej spôsobiť chybovosť (BER). Keďže, som chcel modelovať aj situácie, u ktorých by dochádzalo k rušeniu, musel som pre tieto potreby model čiastočne modifikovať. Daný model neposkytuje vrstvy štandardu ZigBee preto nie je možné modelovať dve nezávislé siete. Danú kolíziu som preto vytvoril pomocou modifikácie MAC vrstvy, čím som dosiahol,

že dané zariadenie sa chovalo ako generátor šumu, tj. periodicky vysiela rámce, s tým, že dochádza ku kolízii a prijímač príjme viacero rámcov súčasne.

V simulácii je používaný diskrétny simulátor, počas simulačného času prebiehajú udalosti. Model súčasného prijatia viacerých rámcov vyzerá tak, že v čase keď prijímač príjme rámce simulačný čas sa zastaví a samotné prijatie rámcov považujeme za udalosti v rovnakom simulačnom čase. Procesorový čas však neustále beží a teda program obsluhujúci súčasne prijatie viacerých rámcov príjme tieto rámce v skutočnosti s určitým časovým odstupom, čo je v simulácii reprezentované pomocou udalosti. Najprv je prijatý prvý rámec, je spracovaná jeho obsluha, následne sa spracuje druhý rámec atď. Tomuto popisu odpovedá nasledujúci obrázok 13.2, ktorý zachytáva situáciu pri ktorej došlo ku kolízii v modifikovanej vrstve MAC (modul *mac*).

13.0.5.1 Popis kolízie

Obrázok 13.2 je výstup z nástroja Sequence chart a detailný popis jednotlivých udalosti je možné analyzovať pomocou nástroja Event log. Oba tieto nástroje sú vhodné pre analýzu simulácie poprípade jej ladenie a pribudli vo verzii OMNeT++ 4.0. V mojom modely 13.1 som simuloval vznik kolízie na module host[0], ktorý periodicky prijímal rámce z modulu host[1]. Modul host[2] som použil ako generátor rámcov, ktoré budú spôsobovať kolízie. Na obrázku reprezentuje sivý úsek zastavenie simulačného času. Modul host[0] prijal v rovnaký čas dva rámce, prvý od modulu host[1], čomu odpovedá číslo udalosti 41, druhý od modulu host[2] s číslom udalosti 43. Oba tieto rámce boli prijaté modulom *snrEval*. Na základe predchádzajúceho popisu modulu *snrEval*, prebehne následujúca obsluha:

1. rámec z udalosti 41, vstupuje do modulu *snrEval*, ktorý sa sa prepne do stavu SYNC, keďže sa jedná o nový rámec, je spočítaná hodnota SNR1, nasleduje prepnutie do stavu Dekódovanie správy

   SNR1 = (výkon, ktorým bol rámec prijatý / šum)

2. následne je prijatý rámec z udalosti 43, tento rámec bude spracúvaný ako šum, spočíta sa nová hodnota SNR2, rámec sa následne zahodí

   SNR2 = (hodnota výkonu, ktorým bol prijatý rámec z udalosti 41) / (šum + prijatý výkon rámca z udalosti 43)

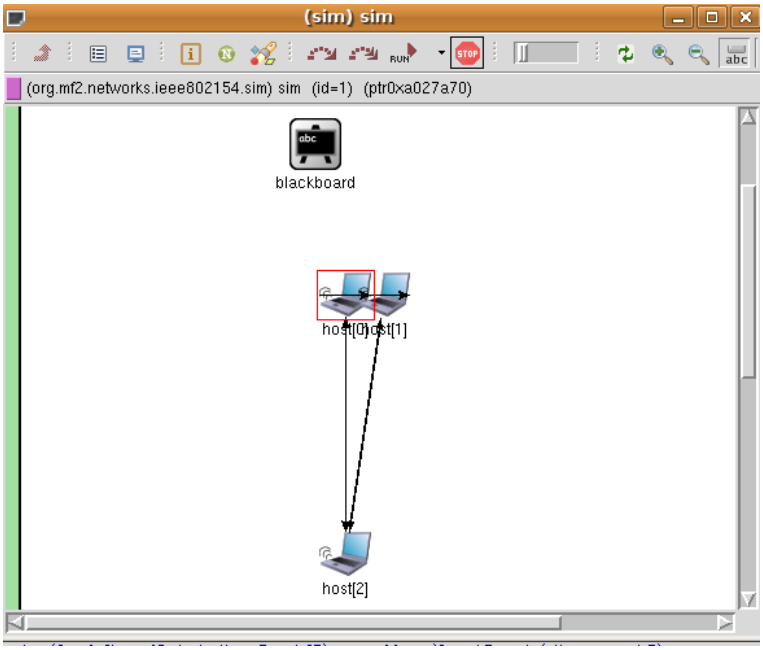
3. ukončí sa spracovanie rámca z udalosti 41, hodnoty SNR1 a SNR2 sa pripoja ako kontrolné informácie k správe, ktorá sa následne prepošle modulu *decider*

4. *decider* na základe hodnôt SNR1, SNR2 spočíta BER a podľa jeho hodnoty sa rozhodne či sa rámec zahodí (tj. rámec obsahuje chybné bity) alebo pošle modulu *mac*

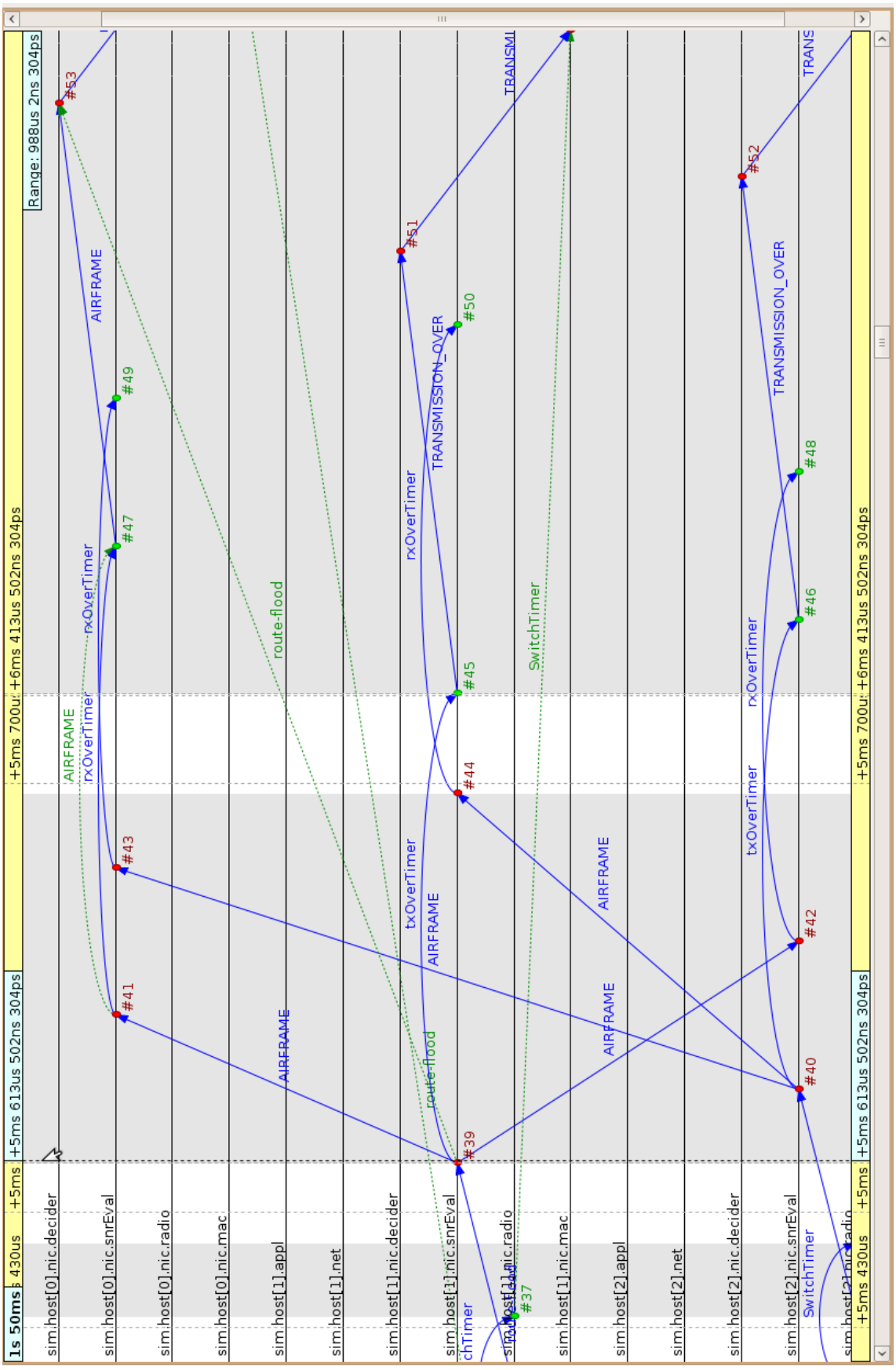
Čím je nižšia hodnota SNR, tým je vyššia pravdepodobnosť, že prijatý rámec bude obsahovať chyby. Tento fakt vychádza zo Shannonovej vety, danej nasledujúcou formulou, ktorá udáva max. teoretický limit prenosovej rýchlosti C kanálu s pásmom o šírke W a odstupom signálu od šumu (SNR).

$$C = W * \log_2(1 + SNR)[b/s, Hz]$$

V prípade, že sa hodnota SNR blíži k nule, prenosová rýchlosť kanálu sa taktiež blíži k nule, z čoho vyplýva, že dochádza k veľkej strate prenášaných dát, čo zapríčiní veľkú chybovosť na prenášaných dátach.



Obr. 13.1: Model kolízie



Obr. 13.2: Kolízia na MAC vrstve

Kapitola 14

Testovanie

V tejto časti popíšem testy, ktoré som uskutočnil pomocou daného modelu a porovnám výsledky týchto testov s reálnym meraním.

14.0.6 Testy zamerané na pohyb XBee zariadení

Pri vykonávaní týchto testov, som modeloval komunikáciu dvoch XBee zariadení, z ktorých jedno bolo v pozícii príjemcu a vysielač sa pohyboval. Hlavný faktor, na ktorý som kládol dôraz bolo pozorovanie ako sa mení hodnota výkonu na strane príjemcu s narastajúcou vzdialenosťou. Taktiež som sledoval počet zahodených rámcov, takzvanú stratovosť rámcov (na základe výpočtu chybovosti BER na prijímači) s narastajúcou vzdialenosťou a oslabovaním sa signálu, viď tabuľka 14.1

Vykonal som následujúce testy:

1. vzdiaľovanie sa vysielača (s výkonom vysielača 1mW a 10mW) po kroku 0.6cm (po každom kroku bol vyslaný rámec) do vzdialenosti 5m, viď. grafy 14.1 a 14.2
2. vzdiaľovanie sa vysielača (s výkonom vysielania 1mW a 10mW) po kroku 0.6cm (po každom kroku bol vyslaný rámec) do vzdialenosti 5m a súčasná náhodná rotácia oboch zariadení okolo vlastnej osi
3. vzdiaľovanie sa vysielača (s výkonom vysielania 1mW a 10mW) po kroku 0.6m (po každom kroku bol vyslaný rámec) do vzdialenosti 250m, viď. graf 14.3
4. vzdiaľovanie sa vysielača (s výkonom vysielania 1mW a 10mW) po kroku 0.6m (po každom kroku bol vyslaný rámec) do vzdialenosti 250m a súčasná náhodná rotácia oboch zariadení okolo vlastnej osi, viď. graf 14.4
5. náhodná rotácia vysielača okolo prijímača vo fixnej vzdialenosti 10m

Detailné výstupy z týchto testov sa nachádzajú na priloženom CD vo forme spracovaných grafov a taktiež vo forme súboru s príponou .sca, čo je jeden z výstupných formátov simulátoru Omnet, ktorý sa dá ďalej vhodne spracúvať pomocou nástroja Scave.

Analýzou týchto meraní je vidno, že krivky grafov sa takmer zhodujú s reálnymi meraniami a to aj napriek tomu, že reálne prostredie obsahuje množstvo faktorov spôsobujúcich odrazy atď. Daným modelom som schopný modelovať reálne podmienky s vysokou presnosťou aj napriek tomu, že zanedbám straty, ktoré v nich vznikajú.

Vzdialenosť [m]	Stratovosť [%]	Výkon vysielača [mW]
100	0	1
150	0.12	1
200	11.87	1
220	30.42	1
250	76.38	1
300	49.65	10

Tabuľka 14.1: Stratovosť rámcov

14.0.7 Testy s kolíziou

V týchto testoch bola poloha zariadení XBee stacionárna. Model uvažoval dve zariadenia prijímač a vysielač, kde vysielač vysielal rámce. Ďalej som do simulácie zapojil generátor šumu (zariadenie periodicky generujúce rámce, ktoré spôsobovali kolíziu). V tejto simulácii som sledoval koľko rámcov bolo zahodených (stratovosť) z dôvodu šumu spôsobeného generátormi na prijímači.

Vykonal som následujúce testy:

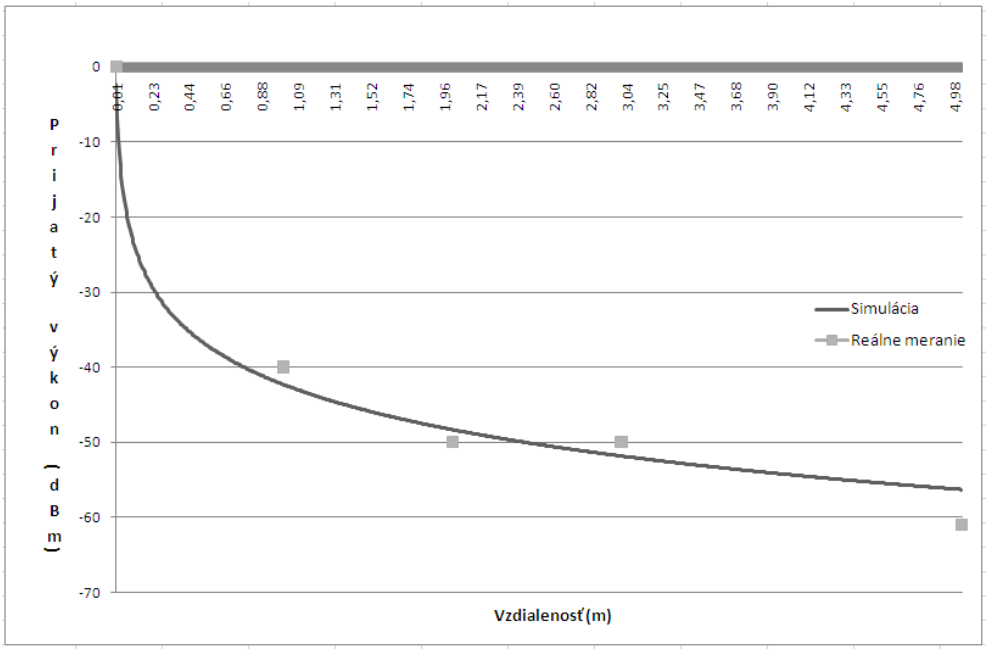
1. model viď. obrázok 13.1 prijímač - host[0], vysielač - host[1] boli umiestnené vo vzdialenosti 30cm, generátor šumu - host[2] (vysielací výkon 10mW) vo vzdialenosti 3m a 4m
2. model totožný s predchádzajúcim no bol pridaný druhý generátor šumu, jeho umiestnenie bolo x = host[2].x - 0.7m, y = host[2].y. Vysielací výkon oboch generátorov šumu bol 10mW.

Vzdialenosť generátora šumu [m]	Stratovosť [%]	Stratovosť v reálnych podmienkach [%]
3	9.8	15
4	0	nebolo merané

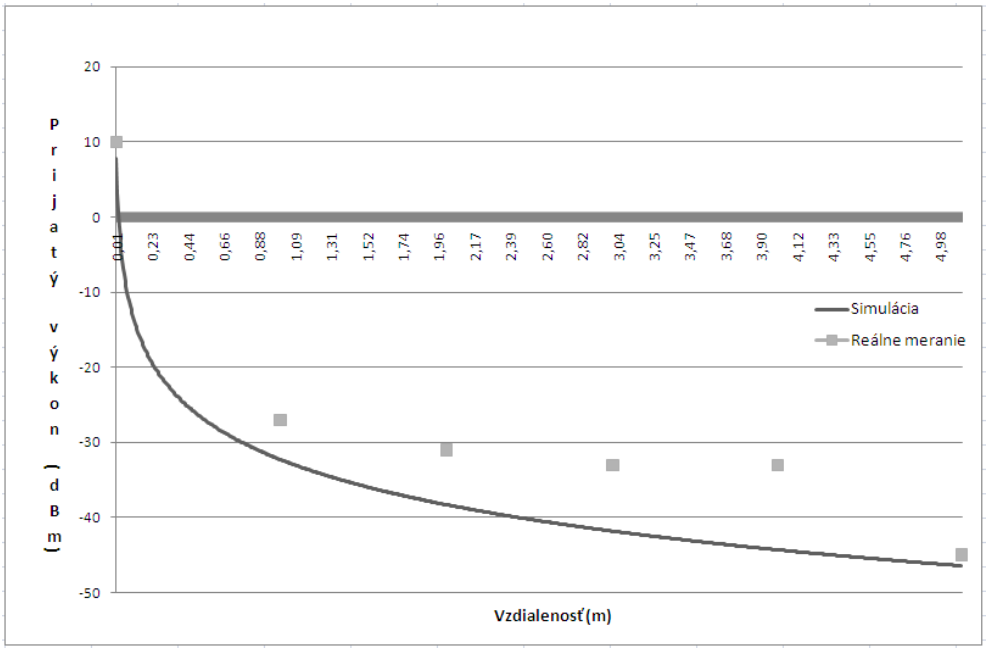
Tabuľka 14.2: Prípad č. 1

Vzdialenosť generátorov šumu [m]	Stratovosť [%]
3	98
4	80

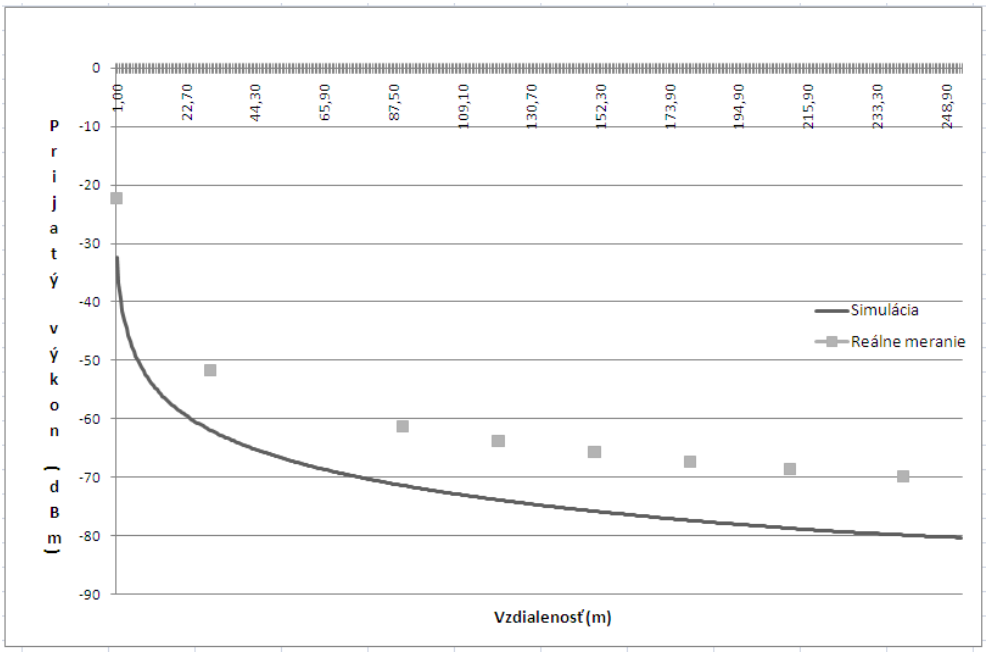
Tabuľka 14.3: Prípad č. 2



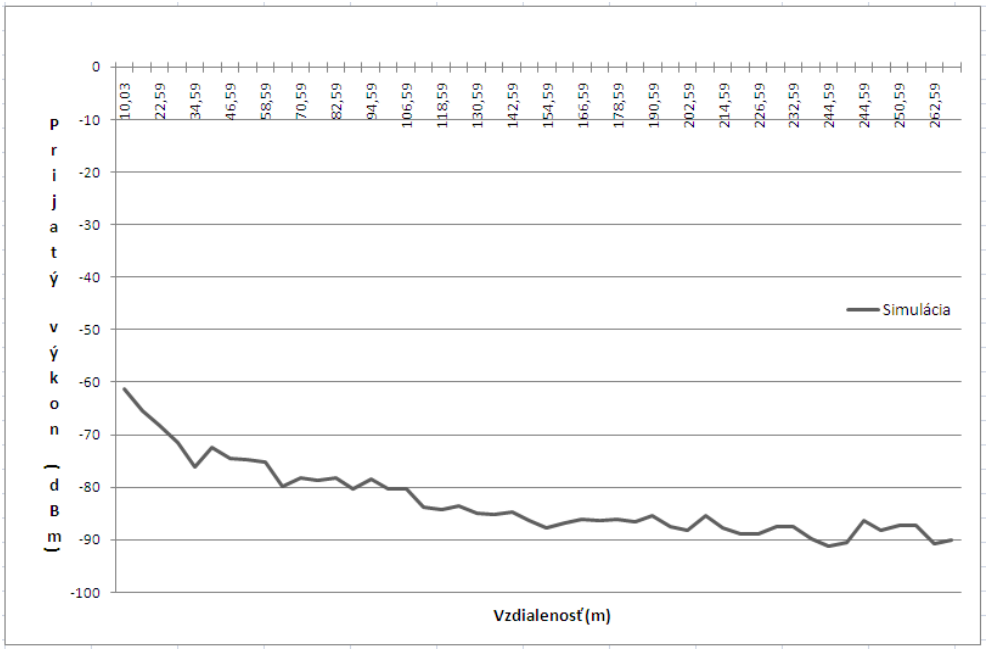
Obr. 14.1: Pohyb na vzdialenosť 5m, vysielací výkon 1mW



Obr. 14.2: Pohyb na vzdialenosť 5m, vysielací výkon 10mW



Obr. 14.3: Pohyb na vzdialenosť 250m, vysielací výkon 10mW



Obr. 14.4: Pohyb na vzdialenosť 250m, vysielací výkon 1mW, rotácia okolo vlastnej osi

Kapitola 15

Záver

Literatúra

[1] Internet message format, 2001.

[2] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, Paul Gauthier. Cluster-base scalable network services, 1997.

[3] A. B. Bondi. Characteristics of Scalability and Their Impact on Performance, 2000.

[4] Brewer. Towards robust distributed systems., 2000.

[5] David Salmen, Tatiana Malyuta, Rhonda Fetters, Normert antunes. Cloud Data Structure Diagramming Techniques and Design Patterns, 2010.

[6] E. F. Codd. A Relational Model of Data for Large Shared Data Banks, Jún, 1970. `www.seas.upenn.edu/~zives/03f/cis550/codd.pdf`.

[7] Eben Hewitt. Cassandra: The Definitive Guide, November 2010.

[8] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proc. VLDB Endow.*, 2:1414–1425, August 2009.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[10] John F. Gantz, David Reinsel, Christopher Chute, Wolfgang Schlichting, John McArthur, Stephen Minton, Irida Xheneti, Anna Toncheva, Alex Manfrediz. The Expanding Digital Universe, A Forecast of Worldwide Information Growth Throught 2010, Marec 2007.
`http://www.emc.com/`, stav z 28. 2. 2011.

[11] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[12] Michael Stonebraker. The Case for Shared Nothing Architecture, 1986.
`http://db.cs.berkeley.edu/papers/hpts85-nothing.pdf`, stav z 28. 2. 2011.

[13] D. Pritchet. BASE: An Acid Alternative.

[14] R. Shoup. The eBay Architecture, Striking a balance between site stability, feature velocity, performance, and cost, November 2006.
`www.addsimplicity.com/downloads/eBaySDForum2006-11-29.pdf`, stav z 28. 2. 2011.

[15] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 1013–1020, New York, NY, USA, 2010. ACM.

[16] J. Udell. *Practical Internet Group Ware*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 1999.

[17] Xavier Défago, Péter Urbán, Naohiro Hayashibara, and Takuya Katayama. The Phi accrual failure detector., 2004.

Dodatok A

Zoznam použitých skratiek

APL Aplication Layer

APS Aplication Support Sub-layer

CSMA-CA Carrier Sense Multiple Access - Collision Avoidance

FFD Full Functionality DEvice

FEL Future Event List

FES Future Event Set

FSPL Free-space path loss

GTS Guaranteed Time Slot

GUI Graphical User Interface

IDE Integrated Development Environment

LQI Link Quality Indicator

LR-WPAN Low-Rate Wireless Personal Area Network

MAC Medium Access Control

MF Mobility Framework

NED Network Description

NIC Network Card Interface

NWK Network

RFD Reduced Functionality Device

PAN Person Area Network

PHY Physical

SAP Service Access Point

SFD Start Frame Delimiter

SNR Signal to noise ratio

ZDO ZigBee Device Object

WLAN Wireless Local Area Network

WPAN Wireless Personal Area Network

Dodatok B

Inštalačná a užívateľská príručka

B.0.8 Inštalácia simulátoru OMNeT++ pre platformu Linux

1. Stiahnutie archívu obsahujúceho zdrojový kód zo stránok
   http://www.omnetpp.org/omnetpp
2. Prekopírovanie archívu do adresára /usr/local/
3. Rozbalenie archívu pomocou príkazu tar zxvf omnetpp-4.0.src.tgz
4. Do užívateľského profilu .bash_profile alebo .profile pridáme riadok
   export PATH=$PATH:/usr/local/omnetpp-4.0/bin
5. Je potreba zabezpečiť prítomnosť následujúcich balíkov v systéme

   sudo apt-get install build-essential gcc g++ bison flex perl tcl8.4 tcl8.4-dev
   tk8.4 tk8.4-dev blt blt-dev libxml2 libxml2-dev
   zlib1g zlib1g-dev libx11-dev

6. Prevedieme následujúce príkazy:
   cd /usr/local/ometpp-4.0
   ./configure
   ./make
7. Spustenie OMNeT++ s IDE pomocou príkazu omnetpp

B.0.9 Inštalácia mnou modifikovaného Mobility Frameworku

1. Stiahnutie súborov Mobility frameworku z svn http://my-svn.assembla.com/svn/mframework/, poprípade prekopírovanie adresára mf2o4 z priloženého CD do adresára /usr/local/
2. Import MF do aplikácie OMNeT++
   (a) Po spustení aplikácie Omnet, klikneme na oblasť „Project explorer“, pravým tlačítkom a zvolíme položku „Import...“
   (b) Zvolíme „General->Existing project into Workspace“

(c) V položke „Select root directory“, zvolíme cestu k adresáru mf2o4, tj. /usr/local/mf2o4

(d) Pomocou CTRL+B, preložíme zdrojové súbory

B.0.10 Práca s modelom IEEE 802.15.4

Vo vývojom prostredí Omnetu si otvoríme v oblasti „Project explorer“ adresárovú štruktúru mf2o4, kde si následne otvoríme adresár networks a v ňom adresár ieee802.15.4. V tomto adresári sa nachádzajú aj xml súbory popisujúce antény. Otvoríme si súbor omnetpp.ini, tento súbor je hlavným konfiguračným súborom modelu simulácie. Zahrnul som do neho ukážkové nastavenia viacerých modelov, ktoré som simuloval. Samotná simulácia sa potom spustí otvorením súboru omnetpp.ini a následným kliknutím na tlačítko „Run“ z menu aplikácie.

Dodatok C

Obsah priloženého CD

Následujúci obrázok C.1 zobrazuje štruktúru priloženého CD.

```
.
|-- data
|   |-- XBee.xlsx                    - popis charakteristik antén z meraní
|   |-- Graphs.xls                   - grafy z realnych meraní
|   `-- Graphs2.xls                  - grafy zo simulácie
|-- mf2o4                            - modifikovaný Mobility framework
|   `-- networks
|       `-- ieee802154               - model IEEE 802.15.4
|           `-- results              - výsledky zo všetkých uskutočnených simulácii
|-- readme.txt                       - popis adresárovej štruktúry
|-- text
|   `-- Lenart-thesis-2009.pdf       - text bakalárskej práce vo formáte PDF
`-- zoznamCD
```

Obr. C.1: Výpis priloženého CD