Examensarbete

Våren 2012

Sektionen för Hälsa och Samhälle

Avdelning Computer Science

Datasystemutveckling

3D-tabell för datalagring som molntjänst

Författare

Mikael Norrman

Handledare

Dawit Mengistu

Examinator

Eric Chen

Dokumentblad

Sektionen för Hälsa och Samhälle
Avdelning Computer Science
Högskolan Kristianstad
SE-291 88 Kristianstad
Sverige

School of Health and Society
Department Computer Science
Kristianstad University College
SE-291 88 Kristianstad
Sweden

Författare, program och år/Author, Program and Year:

Mikael Norrman, Datasystemutvecklingsprogrammet 2009

Handledare/Instructor:

Dawit Mengistu, Hkr

Examen/Examination:

Detta examensarbete på 15 högskolepoäng ingår i examenskraven för *kandidatexamen i datalogi.*

This graduation work on 15 higher educations credits is a part of the requirements for a *Degree of Bachelor with a major in Computer Science*.

Svensk titel:

3D-tabell för datalagring som molntjänst

English Title:

Cloud data storage using a 3D-table

Språk/Language:

Svenska
Swedish

Godkänd av/Approved by:

___

Eric Chen   Datum/Date
Examinator/Examiner

Sammanfattning

Projektet kretsar kring idén om en lagringsform, utformad som en tabell, som inte är bunden till en specifik plattform eller ett specifikt filsystem och som kan fungera som en molntjänst. Denna lagringsform är dessutom mer avancerad än en vanlig tabell då den innehåller tre dimensioner.
Idén kom till under en föreläsning om molntjänsten google app engine då det kort förklarades att de använder ett, av dem själva skapat, system som kallas för BigTable. Detta projekt liknar BigTable men skiljer sig då detta projekt är något enklare, öppen källkod samt att det är baserat på java vilket gör det plattformsoberoende.

Syftet med projektet är skapa en enkel plattform för datalagring som kan användas som molntjänst där den underliggande tekniken lämpar sig för användning på vitt skilda system och hårdvara.
Tanken med ett sådant system är att man med små medel ska kunna skapa en egen datalagring där den traditionella filservern suddas ut och det mer moderna molnkonceptet används. Genom att parkoppla servrar kan man i teorin skapa ett världsomspännande lagringsnätverk med enormt stor lagringskapacitet men är i praktiken mer inriktad på privata moln. Det spelar dock ingen roll om man enbart har en maskin eller flera hundra, för användaren kommer det alltid fungera likadant.

Problemet som detta projekt tar sig an är det om datalagring och hur den kan lösas på ett sätt som bidrar till ökad ekonomisk vinst utan att minska på skalbarhet och användbarhet.
Projektet löser även en annan viktig aspekt med datalagring vilket är versionshantering, något som andra tjänster inte erbjuder, tar bra betalt för eller som enbart finns i begränsad form.

Abstract

The project revolves around the idea of a storage format, shaped like a table, that's not bound to a specific platform or a specific file system and that has capabilities to work as a cloud service. This storage format is also a bit more advanced than a regular table because it has three dimensions.

The idea came from a lecture about Google app engine where it was explained that google uses a self made system called BigTable for storing large amounts of data. This project is similar to BigTable but is built from scratch and has key features such as being open source and is based on Java for platform independency.

The purpose of this project is to create a simple platform for data storage that can be used as a cloud service and where the underlying technology is able to utilize a wide variety of systems and hardware. By using limited computing resources you are supposed to be able to create a data storage where the traditional file server is phased out and the modern cloud concept is used in its place. It would in theory be possible to create a global storage network with almost unlimited storage capacity though in practice it may be more suitable for private clouds. Also it doesn't matter if you only have one server or several hundreds of them, to the user it will function just the same.

This project takes on the problem of solving data storage in an economical, scalable and usable way. It also solves the issue of versioning data, something that similar services don't offer, take good money for or only offer in a limited form.

Erkännande

Jag är mycket tacksam för framförallt två personers direkta och indirekta inblandning i detta examensarbete, två personer som med både kunskap och stöd har hjälpt mig i mitt arbete med hela projektet.
Den första av dessa två personer är min underbara fru, Madeleine Norrman, som med stort tålamod hjälpt mig och stöttat mig i mitt arbete från början till slut genom uppmuntran, förståelse och inte minst genom att ta hand om vår dotter i tid och otid.
Den andra personen är min handledare, Dawit Mengistu, som med sitt engagemang och sin kunskap hjälpt mig med viktiga val i projektet, att anordna utrustning för storskaliga tester och inte minst med den slutgiltiga examensrapporten. Han har stöttat projektet från början och har aldrig varit mer än ett e-postmeddelande bort.

Jag är också tacksam för min examinator, Eric Chen, som varit med redan från början och diskuterat projektet, som har haft höga förhoppningar på det och som också gett sina synpunkter på den slutgiltiga examensrapporten.

Ytterligare personer som har hjälpt mig genom sin förståelse, att agera bollplank, sitt stöd eller av andra orsaker är:
Peter Norrman (Far), Marcus Justesen (Klasskamrat), Fredrik Svensson (Vän), och Paul Tilderkvist (Tidigare arbetsgivare och vän).

Innehållsförteckning

Dokumentblad i

Sammanfattning ii

Abstract iii

Erkännande iv

Innehållsförteckning v

1 Introduktion 1

1.1 Bakgrund 1

1.2 Målsättning och syfte 1

1.3 Problemformulering 1

1.4 Jämförelse 2

1.4.1 Dropbox 2

1.4.2 BigTable 2

1.4.3 MongoDB 3

1.4.4 Likheter och skillnader 3

1.5 Metodik och resurser 5

1.5.1 Hårdvara 5

1.5.2 Mjukvara 5

1.5.3 Github 5

2 Metod 6

2.1 Arbetssätt 6

2.2 Utformning av mjukvara 6

2.2.1 Datahantering 6

2.2.2 Nätverk 7

2.2.3 Bibliotek 7

2.2.4 Grafiskt användargränssnitt 8

2.3 Alternativa lösningar 9

3 Implementation 10

3.1 Datahantering 10

3.1.1 BackStorage 10

3.1.2 DataProcessor 13

3.2 Nätverk, ursprunglig modell 15

3.2.1 Master 15

3.2.2 Slave 15
3.3 Nätverk, slutgiltig modell 16
3.3.1 Struktur 16
3.3.2 Kommunikation 17
3.3.3 Jobb 17
3.3.4 Nätverkshantering 19
4 Resultat 22
4.1 Funktion 22
4.2 Drift och underhåll 23
4.3 Test 24
4.3.1 Test av nätverksmodell 24
4.3.2 Test av överföringshastigheter 24
5 Slutsatser 26
6 Förslag för fortsatt arbete 27
7 Referenser 28
7.1 Källförteckning 28
7.2 Personliga kontakter 29
8 Bilaga 30
8.1 Bilaga 1, build_12_05_29 30
8.2 Bilaga 2, build_12_05_28 30
8.3 Bilaga 3, build_12_05_21 30
8.4 Bilaga 4, build_12_05_14 30
8.5 Bilaga 5, build_12_04_28 30
8.6 Bilaga 6, CustomRouter.zip 30

1 Introduktion

1.1 Bakgrund

Idén till projektet tillkom under en föreläsning om molntjänster och utveckling av applikationer för molnet. Under föreläsningen förklarades det vad Google app engine var för något och att det, liksom många andra Google-produkter och -tjänster, utnyttjade ett system de själva tillverkat som kallas för BigTable. BigTable fungerar som en extremt stor databas, uppbyggd som en tredimensionell tabell och kapabel att hantera datamängder upp över petabytes i realtid.
Google använder BigTable enbart för sina egna produkter med undantag för Google app engine[1] som använder BigTable som back-end för användarnas applikationer.

Förutom Googles BigTable kom det även inspiration från tjänster som t.ex. Dropbox som är en molnlagringstjänst för att kunna synkronisera filer över flera enheter. Dropbox erbjuder en viss lagringskapacitet gratis och har betalalternativ för högre lagringskapacitet.

1.2 Målsättning och syfte

Målet med projektet är att skapa en enkel datalagringsplattform som kan användas av både privatpersoner och företag för att skapa ett eget privat eller publikt datalagringsmoln. Projektet ska vara plattformsoberoende och ska kunna användas på både enkla hemdatorer och på superdatorer oavsett operativsystem och hårdvara. Det ska vara skalbart för olika behov och ska kunna fungera som ett sammansatt nätverk även över internet.
Kanske viktigast ändå är att mjukvaran skall ha öppen källkod så att användare kan ändra mjukvaran för att bättre passa sina behov.

Syftet är att skapa en fri programvara som ger likvärdig skalbarhet och lagringskapacitet som till exempel Googles BigTable och som ger företag och privatpersoner möjligheten att skapa sin egen tjänst liksom Dropbox som de kan utnyttja till alla möjliga olika lagringsbehov. Dessutom handlar det mycket om att jag som utvecklare ska utvecklas och lära mig nya tekniker och utvecklingssätt som ytterligare vidgar mina kunskaper.

1.3 Problemformulering

Detta projekt strävar efter att lösa frågan om datalagring som molntjänst på ett ekonomiskt och skalbart sätt. Projektet skall också vara:

- Enkelt att använda och inte kräva särskild hårdvara.
- Plattformsoberoende med låga ingångskrav för prestanda.
- Öppen källkod så att avancerade användare kan skräddarsy programvaran för deras specifika behov samt för att det ska vara gratis och fritt att använda.
- Anpassat för att kunna skapa ett lagringsnätverk globalt över internet.

Slutligen och kanske viktigast är att datalagringen ska lösas med hjälp av en 3D-tabell för att skapa versionshantering för datan.

1.4 Jämförelse

Detta projekt har många likheter med andra produkter och tjänster som finns men den löser också några av de problem eller brister som dessa andra produkter har. Jag kommer ta upp tre exempel på produkter och ge en jämförelse mellan dessa och mitt projekt.

1.4.1 Dropbox

Dropbox är en tjänst som tillåter en att synkronisera och dela filer över flera enheter. Filerna lagras i deras moln men kan också lagras lokalt på varje enhet för att förenkla tillgången.
De erbjuder tjänsten både i en gratis variant med kraftigt begränsat lagringsutrymme på runt 2-3 gigabyte och som betallösning med lagringskapacitet uppåt terabyte-storlek[3].

Dropbox	
Fördelar	Nackdelar
• Enkelt att använda • Synkroniserar filer över alla anslutna enheter • Viss versionshantering • Tillgängligt överallt	• Ingen möjlighet att själv serva molntjänst • Kostnad

Dropbox hade ca 50 miljoner användare 2011[2] varav ca 96 % var gratisanvändare[4].

1.4.2 BigTable

BigTable är, enligt deras egen beskrivning[5] ett ”distribuerat lagringssytem för att hantera strukturerad data och som är designad för att skala upp till väldigt stora storlekar; upp till petabytes över tusentals servrar”. Det används för att lagra data åt Googles många projekt med exempel som Google Maps, Google analytics m.fl.

BigTable	
Fördelar	Nackdelar
• Skalbart • Användbart inom många olika applikationer • 3D-tabell för versionshantering	• Inte tillgängligt för andra, varken som betald tjänst eller som öppen källkod

Enligt mätningar i augusti 2006[5] gjordes 1.2 miljoner förfrågningar per sekund till de 14 mest aktiva delarna av BigTable vilket genererade 741 megabyte per sekund in och 16 gigabyte per sekund ut i så kallade *remote procedure calls*.

1.4.3 MongoDB

MongoDB[6] är en högpresterande och skalbar databas som har öppen källkod. Mjukvaran finns i versioner för de fyra största operativsystemen (Linux, Windows, OS X, Solaris) och har drivrutiner för ännu fler system och språk. MongoDB började som en hemmabyggd databas som skulle vara distribuerbar istället för de traditionella relationsdatabaserna och slutade som en egen produkt som skaparna valde att släppa som öppen källkod.
MongoDB sparar data som dokument i JSON-format.

MongoDB	
Fördelar	Nackdelar
• Öppen källkod • Finns i versioner för de största operativ-systemen • Skalbart och snabbt	• Inte fullt plattforms-oberoende • Sparar data som dokument

1.4.4 Likheter och skillnader

Även om detta projekt är byggt helt och hållet från grunden delar det många koncept med dessa tre produkter/tjänster och använder dem för att skapa en ny unik lösning. De största likheterna finns hos BigTable och MongoDB där den sistnämnda är den mest snarlika produkten.

För att börja med Dropbox så är likheterna möjligheten till att lagra datan på en central plats och att kunna komma åt den från var som helst. Därefter slutar likheterna eftersom Dropbox inte säljer själva produkten utan endast tjänsten i motsats till detta projekt som ger användaren möjligheten att själv implementera sin molnlagring och att sedan kunna använda den som det passar i just det fallet. Även om Dropbox har en gratisversion av sin tjänst så är det också en stor skillnad eftersom detta projekt är inriktat på att vara öppen källkod och gratis.

När vi sedan går vidare till BigTable blir likheterna något fler men även här finns det skillnader. Först och främst så är det från BigTable som detta projekt hämtat inspirationen till sin 3D-tabell och, tillsammans med MongoDB, sin distribuerade lösning för att fungera som molntjänst. Även sättet som datan identifieras på är snarlikt men inte identiskt.
Den däremot största skillnaden mellan dessa två produkter, som detta projekt löser, är att BigTable är helt slutet och används enbart av Google internt för deras produkter, dock med undantag för Google app engine som ger användarna viss indirekt användning av BigTable. Detta projekt går tvärtemot den principen och är istället öppen källkod, fri för alla att ta del av och använda hur de än vill vare sig det är stora företag eller privatpersoner.

Vid en första anblick kan det verka som om detta projekt är likadant som MongoDB eftersom de båda är öppen källkod, distribuerade system och datalagring utan SQL men det finns en viktig skillnad och det är hur datan lagras. MongoDB lagrar data som dokument byggda i JSON som den sedan indexerar och lagrar medans detta projekt med sin 3D-tabell lagrar datan i celler bestående av ett kolumnvärde och ett radvärde samt en tidsstämpel som ger detta projekt sin tredje dimension. Tidsstämpeln används ifall det finns flera versioner av samma data.

Detta projekt försöker skapa en produkt som ger en datalagring snarlik den som BigTable använder men som är öppen och fri såsom MongoDB. Detta projekt strävar också efter att bli plattformsoberoende och användbart på så många olika enheter som möjligt tack vare Java.

1.5 Metodik och resurser

Arbetet med detta projekt har utförts enligt principerna för agil systemutveckling[9] med många kortare iterationer där fokus har legat på att skapa prototyper och att använda trial-and-error för att utveckla mjukvaran.
För varje del av projektet har en modell fastslagits i förväg för att ge en röd tråd igenom hela utvecklingsprocessen.
De resurser som använts beskrivs i mer detalj under egna underrubriker nedan.

1.5.1 Hårdvara

Under utvecklingens gång har en laptop modell Dell Latitude E6500 används för programmering och kontinuerliga småskaliga tester.
För längre tester som provar stabilitet har en specialbyggd dedikerad server använts.
Ytterligare en maskin har stått till förfogande för tester i samtida körning när så behövts.

I slutskedet av examensarbetet kommer även ett serverkluster bestående av sex noder finnas tillgängligt på Högskolan Kristianstad i Hässleholm.

1.5.2 Mjukvara

För utvecklingen av projektet används Netbeans IDE version 7.1.1 som plattform för java och webbutveckling. För analys och databehandling används Textpad version 5.3.1 och Notepad++ version 5.9.3.
Mjukvaran testas sedan under följande förhållanden:

- Windows 7 – JRE (Java runtime environment) 1.7
- Windows server 2008 – JRE 1.7
- Linux – JRE 1.6 och 1.5

Tester görs med interna funktioner samt i viss utsträckning med analysverktyget Wireshark version 1.4.4.

1.5.3 Github

Github är en versionshanteringstjänst baserat på Git[7] som jag valt att använda eftersom det har hög prestanda, är gratis och för att det är anpassat för öppen källkod vilket gör det enkelt att dela med sig av. Med över en miljon användare[8] är det dessutom en etablerad plattform för just denna uppgift.
Det är viktigt för detta projekt att vara öppet från början och github gör det väldigt enkelt eftersom vem som helst kan titta på innehållet. Däremot kan inte vem som helst göra ändringar i programmet utan de får i så fall ansöka om att få hjälpa till eller helt enkelt göra en kopia på koden och göra sin egen version.

2 Metod

2.1 Arbetssätt

Då detta examensarbete utförs av enbart en person sker allt arbete i hemmet med den utrustning som finns tillgänglig. En god kontakt med handledare ska upprätthållas för att få kontinuerliga bedömningar av arbetes gång.

Själva arbetet ska utföras enligt principerna för agil systemutveckling[9] med korta och täta iterationer. Detta ska användas för att underlätta implementationen av mjukvaran längre fram då mycket tester och prototyper kommer behöva göras.
Till skillnad från vattenfallsmodellen ger agil systemutveckling mer flexibilitet under utvecklingens gång och fokuserar mindre på dokumentation och mer på kommunikation för att komma framåt.

2.2 Utformning av mjukvara

Projektets mjukvara ska formas på ett sådant sätt att det löser de problem som formulerats tidigare i denna rapport samtidigt som det ska vara enkelt att tyda. Vissa aspekter har inte tagits med i planeringen av mjukvaran för att kunna skapa en rimlig tidsplanering. Saker som tagits bort är bland annat säkerhetskopiering och säkra dataöverföringar med kryptering.
Själva utformningen av detta projekt har gjorts i två huvuddelar och två mindre delar som är av mindre vikt för slutprodukten. Var del förklaras under egen underrubrik med de två huvuddelarna först.

2.2.1 Datahantering

Detta är den del som tar hand om datan som ska lagras på varje dator. All data lagras tillsammans i en enda stor fil, en datafil, som programmet kan hoppa runt i hursomhelst. Med varje datadel som lagras följer det viktig information som sparas tillsammans med datan i datafilen.
Denna del är designad för att kunna hantera samtida körning med många läs- och skriv-operationer samtidigt och har en del funktioner för att hantera fel och genomföra säkerhetskontroller. Dessa funktioner är:

- **Omindexering**
  Använder den information som lagras tillsammans med varje del data för att återskapa en katalog med samtliga datadelar som denna datafil innehåller.
- **Dataintegritetskontroll**
  Använder ett kontrollvärde hämtat från samma information som den som nämns i omindexering för att se om den data som finns lagrad just nu överrensstämmer med hur den såg ut när den först lagrades. Värdet får man genom att läsa igenom varje byte av datan och sedan använda dessa bytes i en algoritm som räknar ut ett kontrollvärde ur ordningsföljden. Om något kontrollvärde inte stämmer överens markeras den datan för borttagning. Inga försök att reparera datan kommer att göras utan den kommer enbart tas bort.

- **Upprensning**
  Dess enda uppgift är att kontrollera om en datadel är markerad för borttagning och sedan ta bort datan och dess medföljande information.

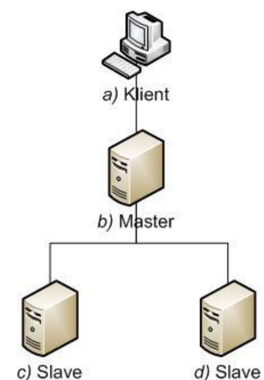
För att kunna hålla koll på hur många som har tillgång till datafilen samtidigt används ett biljettsystem. Var biljett innehåller en länk till datafilen och ett värde som bestämmer om biljetten är använd eller ej.
Biljetterna används sedan i ett verktyg som definierar hur dataöverföring ska ske mellan en yttre faktor och den inre datafilen. Allt för att minska risken för datafel i slutändan.

2.2.2 Nätverk

Den största delen av projektet är nätverksdelen. Denna del definierar hur servrar pratar med varandra inom sitt kluster men även hur en utomstående klient pratar med en server.
De punkter som är alla viktigast för denna del är:

- Möjligheten att kunna skapa ett sammankopplat lagringsnätverk även om enheterna är på vitt skilda fysiska platser.
- Att kunna prestera även om antalet anslutna klienter är stort.
- Att kunna ha ett kluster bestående av endast en dator upp till ett hundratal datorer.

Nätverket ska använda en modell *(Se figur 1)* där den första servern, även kallad master *(enhet b)*, är den enda server som har kontakt med klienterna *(enhet a)* och som vidarebefordrar förfrågningar till enheter inom klustret *(enhet c och d)*. Det innebär att om en eller fler slavar är anslutna till klustret så kommer master att minska sin egen lagringsaktivitet och istället fokusera på att skicka förfrågningarna vidare ut till sina anslutna slavar.
Detta är ett platt modell där alla slavar är direkt underställda mastern. För att hålla ordning på klustret är master försedd med en global katalog som registrerar alla slavars lokala kataloger. När en förfrågan kommer in så slås den eftersökta datan upp i den globala katalogen, slaven på vilken datan finns identifieras och förfrågan skickas vidare.
Detta innebär också att slavarna kontinuerligt får skicka uppdateringar till master när en post i dess lokala katalog ändras, läggs till eller tas bort.



Figur 1, enkel nätverksmodell 1

2.2.3 Bibliotek

Detta projekt ska sträva efter att vara så enkelt för utvecklare att använda som möjligt, oberoende av vilket språk man använder.
Tanken är att projektet ska ha ett flertal färdiga bibliotek eller drivrutiner för olika språk såsom C#, C++, Java, Python med flera. Ett bibliotek fungerar som en mellanhand mellan utvecklarens egen applikation och detta projekt för att underlätta implementationen och för att försöka eliminera risken för att felaktig data skickas i onödan.
På grund av den begränsade tid som examensarbetet innebär så är denna del lågt prioriterad och endast avsedd att bli implementerad om tid finns.

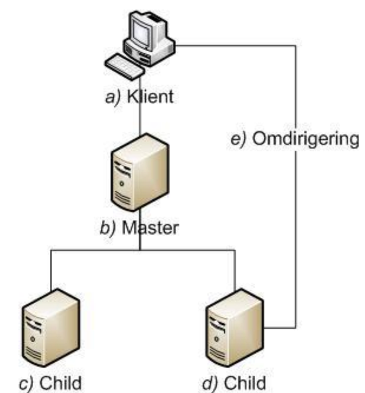
2.2.4 Grafiskt användargränssnitt

Som administratör vill man gärna ha någon form av kontroll över vad som sker i ens privata moln eller bara ha koll på hur systemet fungerar. Tanken är att projektet ska ha ett enklare gränssnitt för att kunna visa upp hur mycket data som finns lagrat på var nod i klustret och vad noden gör just nu.

Även vid installation vore ett grafiskt gränssnitt önskvärt för att underlätta installationen för administratören.

Denna del är precis som föregående del lågt prioriterad och endast avsedd att bli implementerad om tid finns.

2.3 Alternativa lösningar

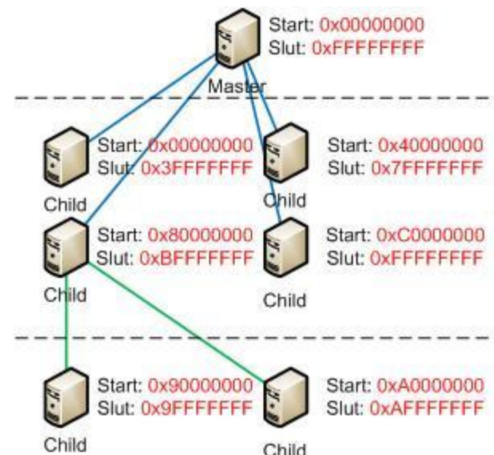


Figur 2, enkel nätverksmodell 2

Under arbetets gång började det bli mer och mer uppenbart hur den nätverksmodell som implementerats från början inte var en bra lösning för att kunna tillmötesgå de tre viktigaste punkterna för en nätverksmodell i detta projekt.
Därför diskuterades en ny modell *(Se figur 2)* fram som ska använda en trädstruktur för alla slavar, numera kallade barn *(enhet c och d)*, och som tar bort den globala katalogen hos master och istället bryter upp datan i områden. Varje barn i trädet (inklusive master själv) tilldelas ett område som den enheten styr över.

Master kan då dirigera om klienters anslutningar *(linje e)* till det barn som har korresponderande område i relation till begärd data. Områden delas upp utefter ett 8 tecken långt hexadecimalt id som genereras från kolumnvärdet för en datadel.

Delningen *(Se figur 3)* börjar hos master som maximalt kan ha 4 barn. Om master har tre barn eller färre kommer master själv också att hantera förfrågningar men om master har fyra direkt underställda barn så ger den helt upp sitt eget område och tar enbart hand om administrativa uppgifter. Därefter kommer ett barn endast kunna ha tre underställda barn till vilka det kan dela ut totalt tre fjärdedelar av sitt område.
I denna lösning har master information om vilka områden som tillhör vilka servrar och kan vid en förfrågan använda den informationen för att omdirigera en klient. Vilket område som tilldelas till ett barn bestäms alltid av det nya barnets förälder.



Figur 3, fördelningsdiagram

3 Implementation

Denna del beskriver hur systemet faktiskt är konstruerat och ger en mer detaljerad och teknisk beskrivning av det som tidigare beskrivits under rubrik 2.2 Utformning av mjukvara.

Genomgående för hela projektet är dock tekniken java NIO[10][11] och dess bufferts som används i nästan alla delar av mjukvaran. Java NIO ger många fördelar gentemot javas standard IO-bibliotek där framförallt dess bufferts med möjlighet till direkt minnesallokering, dataöverföring direkt mellan filer och dess icke-blockerande skriv och läsfunktioner spelar stor roll för mjukvarans utformning. Framförallt de icke-blockerande skriv och läsfunktionerna ger en helt annan konstruktion i jämförelse med standard IO. Tack vare det kan man använda en enda tråd för att hantera tusentals anslutningar.

3.1 Datahantering

När data lagras så följer även viss information *(Se figur 4)* med datan som gör den identifierbar och kontrollerbar. Denna information, kallad metadata, är alltid 512 bytes stor och innehåller information som under vilken kolumn och under vilken rad datan ligger, vilken version datan är och hur många bytes datan består utav.

Metadata - Definition

512 bytes

128 bytes Kolumn	128 bytes Rad	8 bytes Version	8 bytes Längd	8 bytes Checksum	128 bytes Ägare	8 bytes Förskjut-ning*	95 bytes <tom>	1 byte Städbyte**

* - Ett värde som endast används i LocalIndex.java. I övriga fall är dessa bytes också toma.
** - Ett värde som endast används i BackStorage.java för att kunna markera data för upprensning.

Figur 4, definition av metadata

Denna definition på bytenivå existerar för att förenkla lagringen och för att göra det enklare för nätverksdelen att skicka och ta emot denna typ av information. Anledningen till att det finns 95 bytes som är tomma är för att kunna ge utrymme åt framtida förändringar. Från allra första början var antalet tomma bytes 104.

När datan ska användas i mjukvaran används en särskild klass som byggts upp efter denna definition och som kan omvandla bytedatan till särskilda objekt. Totalt sett finns det två olika typer av dessa objekt, kallat *UnindexedDataObject* och *IndexedDataObject*, som är underställda den abstrakta klassen *DataObject*. Skillnaden mellan dessa två objekt är att *UnindexedDataObject* inte innehåller versions eller förskjutningsinformation.

Själva datahanteringen består utav sex klasser, två huvudklasser och fyra nyttoklasser. De två huvudklasserna är *BackStorage* och *DataProcessor* och de båda ger resten av mjukvaran dess nödvändiga tillgång till den datafil där all data lagras.

3.1.1 BackStorage

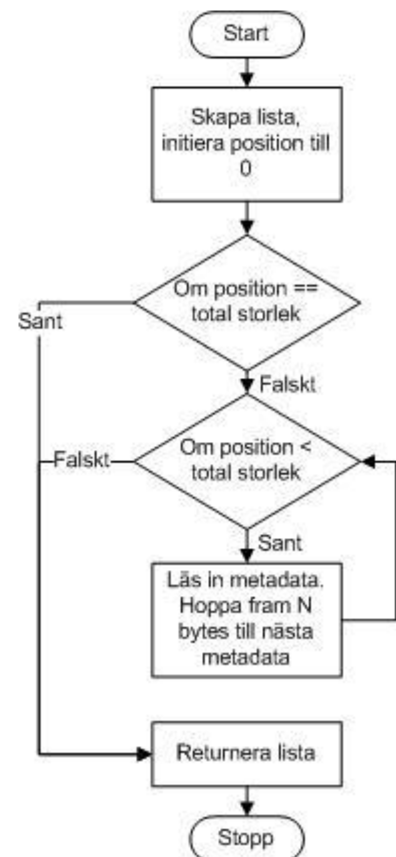
BackStorage är den klass som har absolut ensamrätt till en servers datafil. Den sköter om dataintegriteten, omindexering och upprensning av data när så krävs.
När en del av programmet vill komma åt datan så skapar den en ny instans av *DataProcessor*

som i sin tur kräver en biljett från *BackStorage*. Denna biljett innehåller ett *FileChannel*-objekt genom vilken åtkomst till datan sker. Biljetten har sedan en booleanvärde vilket ska sättas till sant när användandet av en biljett är klart.
Anledningarna till att använda ett biljettsystem är att kunna styra antalet samtida biljetter, även fast det inte är implementerat, och för att kunna ha koll på om det finns öppna dataanslutningar som behöver köras klart vid en eventuell avstängning. *BackStorage* initierar även en klass som körs i jämna intervaller och som aktivt rensar ut använda biljetter.

Huvudfunktionerna för denna klass är:

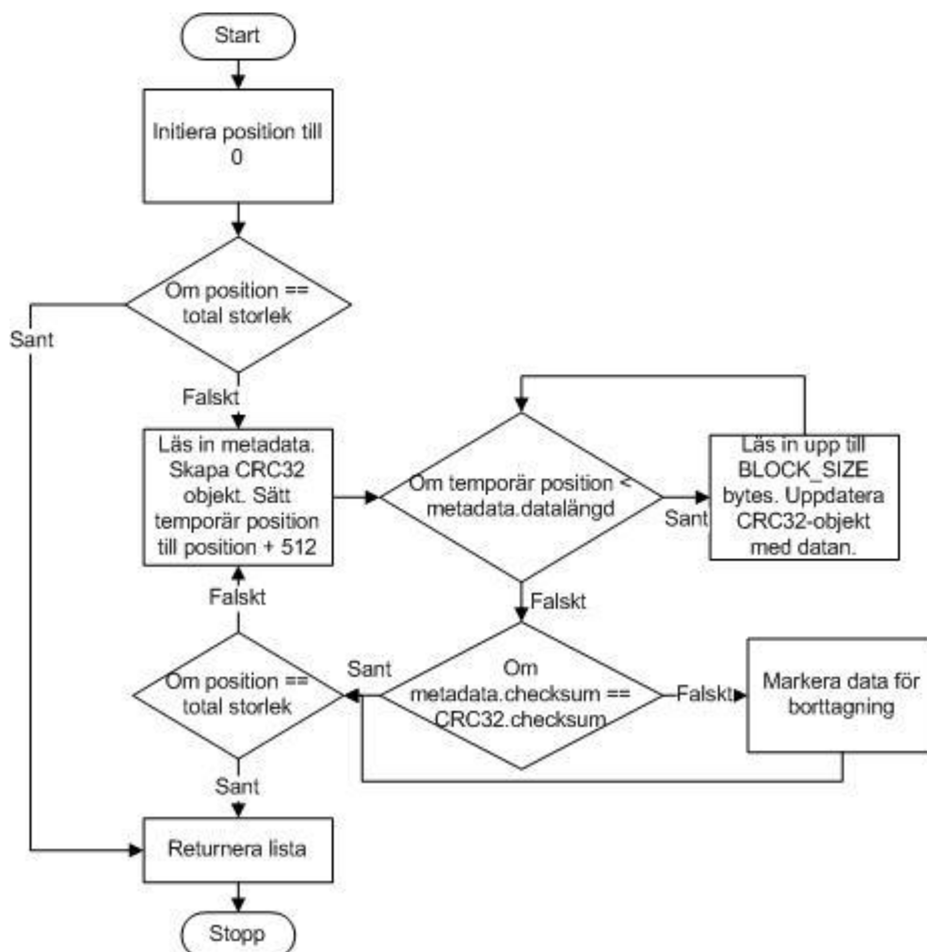
- **Omindexering**
  Att läsa in all indexeringsdata från datafilen och spara det i det lokala indexet.
  Processen *(Se figur 5)* skapar en tom lista av *IndexedDataObject* och sätter datafilens pekare (Indexvärdet för den byte i datafilen vi för närvarande kan läsa och skriva till) till noll. En kontroll görs om det finns data i filen och om inte så avslutas metoden direkt och returnerar således en tom lista.
  Om det finns data i filen så läser metoden in 512 bytes och omvandlar dem till ett *IndexedDataObject*. Detta metadataobjekt läggs sedan till i listan och pekaren flyttas fram det antalet bytes *(N bytes i flödesschemat)* som är specificerat som längden på datan i metadataobjektet.
  Denna läsning och framhoppning fortsätter sedan tills slutet av datafilen har nåtts då metoden returnerar den nu fyllda listan.

  Tack vare metoden *insertAll(List<IndexedDataObject>) i LocalIndex* kan denna process enkelt sammankopplas direkt till serverns lokala index.
  Omindexering används framförallt vid uppstart om servern upptäckt att den senaste avstängningen skedde felaktigt (Ej via kommandoraden).



Figur 5, omindexering flödesschema

- **Dataintegritetskontroll**



Figur 6, Flödesschema dataintegritetskontroll

En fullständig kontroll av den lagrade datan för att se så den fortfarande är korrekt. Kontrollen består av en jämförelse av kontrollvärden av typen CRC32[12] ("Cyclic redundancy code" med en polynomialstorlek på 33 bitar). CRC32 är en metod för att beräkna kontrollsummor och används populärt i ethernetnätverk och fillagring. Anledningen till dess popularitet är dess snabbhet och för att det är väldigt enkelt att implementera binärt i hårdvara.

När så data lagras för första gången skapas en ny CRC32-kontrollsumma baserat på dess bytedata. Denna kontrollsumma används sedan i integritetskontrollen för att se så den nuvarande bytedatan stämmer överens med när den lagrades första gången.
För att kunna genomföra denna kontroll *(Se figur 6)* sätts datafilens pekare till 0 och den första metadatainformationen läses in. Därefter sätts pekarens position till precis efter metadatans 512 bytes och datan tillhörande metadataobjektet börjar läsas in. All data som läses skickas till ett CRC32-objekt som räknar ut kontrollsumman för datan i dess nuvarande skick. Denna process med läsning av data och uppdatering av CRC32-objekt sker ända tills antalet lästa bytes överrensstämmer med datalängdsvärdet i metadataobjektet. Därefter jämförs kontrollsumman i metadatan med den nya kontrollsumman och om resultatet visar på olikheter markeras datan för borttagning

(Markeras genom att den sista byten, städbyten, i metadatan får värdet 1).
När hela datafilen genomsökts returnerar metoden ett booleanvärde som endast är sant om inga felaktiga kontrollsummor hittats.

- **Upprensning**
  En metod som genomsöker datafilen efter metadata markerat för borttagning. Metoden fungerar nästan likadant som omindexeringen *(Se figur 5)* men med skillnaden att den inte lagrar metadatan i en lista utan den letar rätt på metadata som har en städbyte med värdet 1.
  När den hittar metadata som uppfyller det kriteriet rensas metadatan och den tillhörande bytedatan bort. Borttagning sker genom att alla bytes direkt efter den markerade datan flyttas bakåt och skriver över den data som ska tas bort. Därefter minskas datafilens storlek ner med det antal bytes som var specificerat i den borttagna metadatan.
  Under hela processen har metoden exklusiv åtkomst att skriva till datafilen.

3.1.2 DataProcessor

Denna klass fungerar som mellanhand mellan *BackStorage* och övriga delar av programmet. Den är designad för att användas för en operation och sen avslutas.
Rent praktiskt hanterar den lagring, hämtning och borttagning av data via ett *FileChannel*-objekt vilket den får från *BackStorage* (I form av en biljett). Metoderna är ganska enkelt konstruerade och förutsätter antingen blockerande eller icke-blockerande körning.
Metoderna är som följer:

- **retrieveData**, icke-blockerande
  Metoden använder sig av tre parametrar, en buffert av typen *ByteBuffer*, ett 64-bitars integervärde som definierar läsposition i datan (ej i filen) samt ett *IndexedDataObject* som representerar den data som hämtas.
  Som icke-blockerande datahämtningsmetod läser den endast in så många bytes som får plats i buffertobjektet och returnerar sedan antalet lästa bytes som kontrollvärde.
- **retrieveData**, blockerande
  Denna metod använder sig av två parametrar, ett objekt av typen *OutputStream* och ett *IndexedDataObject* som representerar den data som hämtas.
  Olikt sin systermetod läser denna metod all data tillhörande metadataobjektet under en körning och skriver den direkt till dataströmmen som skickades med i metodanropet. Metoden är egentligen inte intressant för huvuddelen av projektet och används således inte aktivt men har viss användning i tester och liknande.
- **storeData**, blockerande
  Denna metod tar endast emot ett *UnindexedDataObject* som parameter och är blockerande eftersom den förutsätter att datan redan mottagits över nätverket och att datan redan existerar i en temporär fil.
  Anledningen till detta är för att undvika datakorruption i den mån det är möjligt och för att göra skrivoperationen till datafilen så kortvarig som möjligt.
  Metoden börjar med att registrera tiden då metoden startar (Vilket senare kommer

fungera som tidsstämpel i metadatan). Därefter skaffar metoden ensamrätt till att skriva till datafilen genom att använda en semafor i *BackStorage*-klassen. Detta är nödvändigt för att undvika att data överskrivs vid samtida skrivoperationer. Därefter läses datafilens nuvarande storlek av och blir det nya offsetvärdet för den datan som ska lagras. Det sista som sker innan datan kopieras in i datafilen är att datafilens storlek ökas med 512 bytes plus storleken på datan som ska lagras.
När så förberedelserna är gjorda kopieras datan från den temporära filen över till datafilen. Så fort kopieringen är avslutad tas den temporära filen bort och metoden returnerar ett *IndexedDataObject* som representerar den nyligen inlagda datan.

- **removeData**, blockerande
  Denna metod tar emot ett *IndexedDataObject* vilket representerar datan som ska tas bort. Själva borttagningen fungerar precis likadant som *BackStorage*-klassens upprensningsmetod. Borttagning sker genom att alla bytes direkt efter den specificerade datan flyttas bakåt och skriver över den data som ska tas bort. Därefter minskas datafilens storlek ner med det antal bytes som var specificerat i den borttagna metadatan.
  Returnerar sant om borttagningen var lyckad och falskt om inte.
- **removeData**, blockerande
  Denna metod möjliggör borttagning av flera dataobjekt samtidigt. Den tar emot en lista av *IndexedDataObject* och anropar den första removeData-metoden för vart objekt i listan. Efter att vart objekt tagits bort sparas det returnerade värdet från anropet i en temporär booleanvariabel. Har det värdet en gång blivit falskt så kan det inte bli sant igen.
  När alla objekt i listan tagits bort returneras sant om alla objekt togs bort utan problem eller falskt om minst ett av dem hade problem.

3.2 Nätverk, ursprunglig modell

Från början implementerades nätverksdelen enligt specifikationen beskriven under rubrik 2.2.2 Nätverk. Denna implementation innebar att en server utseddes till master och alla övriga servrar som skulle tillhöra lagringsnätverket fick bli slavar till master. Det innebar att samtliga slavar var direkt underställda master.
För att veta var all data fanns hade master en global indexeringskatalog i vilken alla slavars lokala indexkataloger registrerades.

3.2.1 Master

Själva skrivandet och läsandet av data sker i en enda tråd som använder icke-blockerande *SocketChannel*-objekt för att kommunicera med både klienter och slavar. Till detta hör en lyssnarklass som implementerar *ServerSocketChannel* för att vänta på nya anslutningar.
Problemet med ett icke-blockerande nätverkssystem är att inte all data kommer fram på en gång, något som innebär svårigheter om man är van vid strömbaserad dataöverföring.
Lösningen blev att implementera jobb för olika uppgifter som håller data nödvändig för just den uppgiften. Ett jobb är alltid en subklass av huvudtypen *AbstractJob*, som har två abstrakta metoder för läs och skrivoperationer, vilket gör att nätverkskoden inte behöver veta exakt vilken typ av jobb som genomförs utan den behöver bara anropa skrivmetoden i jobbet för att låta den skriva eller först läsa data till en buffert och anropa läsmetoden med den bufferten.
Som tidigare sagt så behandlar denna implementation både klienter och slavar i samma klass vilket innebär att de måste kunna separeras, något som lösts genom att tilldela id-nummer från 0 till 1 073 741 823 till klienter och id-nummer från 1 073 741 824 till 2 147 483 647 till slavar.
Master tar också hand om anslutningar mellan en klient och en slav genom att använda ett specifikt jobb kallat *RouteJob*. Med dess hjälp vidarebefordras nätverkstrafiken direkt till den slav som har den eftersökta datan.

3.2.2 Slave

En slav fungerar på samma sätt som master med sin icke-blockerande kommunikation i en enda tråd. Skillnaden är däremot att en slav endast har en anslutning och det är till master. Klienter når slaven genom att först kontakta master som sedan vidarebefordrar till slaven. När en klients förfrågan väl kommit så långt behandlas den med hjälp av jobb precis som på mastern.
Slaven har dessutom i uppgift att kontinuerligt uppdatera mastern med information om nya, uppdaterade eller borttagna index, något som den gör efter att en förfrågan har slutförts. Det sker även en komplett uppdatering av den lokala indexkatalogen vid uppstart av en slav.

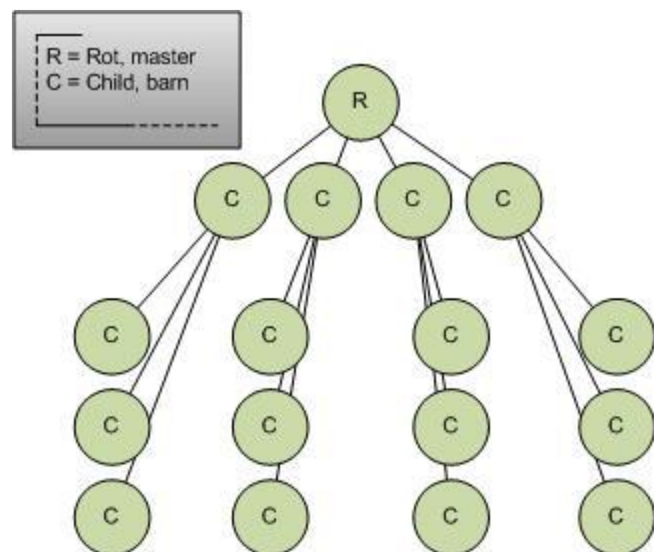
3.3 Nätverk, slutgiltig modell

Den slutgiltiga modellen skiljer sig en del från den förra modellen men ger också ett mer överskådligt system. De främsta skillnaderna gentemot den förra modellen är:

- Den platta hierarkin byts ut mot en trädstruktur med master som rot och alla slavar benämns istället som barn.
- Alla noder (Varje server i lagringsnätverket) har samma funktioner vilket gör att klienter inte längre behöver ansluta via rotservern utan kan ansluta direkt dit den ska.
- Den globala indexkatalogen är borttagen och endast de lokala indexkatalogerna finns kvar. Data identifieras genom att vart barn i trädstrukturen ges ett hexadecimalt område mellan 0x00000000 och 0xFFFFFFFF. Var data ska finnas i dessa områden bestäms av en hashningsalgoritm som körs på ett *IndexedDataObjects* kolumnvärde.
- Hanteringen av klienter och intern kommunikation mellan noderna är separerad och sköts i vars en tråd.

3.3.1 Struktur

Denna nätverks-modell använder en trädstruktur för att dela upp arbetsbördan i lagringsnätverket. Varje enhet i trädet kallas för nod men kan också delas upp som rot och barn. Det finns alltid en, och endast en, rot i ett träd medans det kan finnas obegränsat antal barn.
För detta träd gäller tumregeln att alla barn kan ha högst tre egna barn förutom roten som kan ha fyra *(Se figur 7)*. Anledningen till denna uppdelning är att ett barns omfång ska delas upp i fyra delar och delas lika över barnet själv och dess tre barn. På så sätt blir barnet inte själv utan arbete. Undantaget är såklart roten som helt överger sitt arbete med att lagra data om fyra eller fler barn finns i trädet.



Figur 7, trädstruktur

Området som delas upp är ett 32-bitars osignerat integervärde (För enkelhetens skulle omvandlat till hexadecimal form i både mjukvara och denna rapport). Uppdelningen av detta värde sker genom att en nod i trädet får ett område som tillhör just den noden *(Se figur 3)*.
Nodens förälder vet då att om data med hashvärde x ska hämtas så kan den bara skicka förfrågningen vidare till denna nod och veta att den kommer rätt i slutändan. När förfrågan kommer till denna nod kollar den upp om x är i det omfång som just denna nod har eller om den ska vidarebefordra förfrågan till ett av sina egna barn. Så fortsätter det tills förfrågan når den nod som är faktisk ägare till området som x finns i.
I praktiken fungerar det likadant med skillnaden att roten har en representation av hela trädet och kan därför genomföra denna kontroll utan att behöva omdirigera klientens anslutning hela

tiden. Istället omdirigeras klientens anslutning direkt till den nod som äger området som x tillhör.

Ett område tilldelas första gången ett nytt barn ansluter till lagringsnätverket och blir permanent för det barnet. Mjukvaran förutsätter att ett barn är online och är anslutet till lagringsnätverket.

3.3.2 Kommunikation

All kommunikation, både internt och externt, sker i byteformat och tolkas om på respektive nod eller klient. För klienter gäller följande:

- Om det är första meddelandet i en förfrågan så innehåller den första byten alltid protokollvärdet.
- Övrig data definieras av det jobb som initieras beroende på protokollvärdet.

För intern kommunikation mellan noder gäller följande:

- Ett meddelande börjar alltid med fyra bytes identifikationsvärde och ytterligare fyra bytes för längdvärde. Längdvärdet bestämmer hur mycket mer data som ska läsas av programmet.
- Nästa fyra bytes representerar ett ID för det jobb som skickade meddelandet. Används för att kunna svara till rätt jobb om svar krävs.
- Nästa byte representerar protokollvärdet för detta meddelande.
- Resterande bytes behandlas olika beroende på protokollvärdet.

3.3.3 Jobb

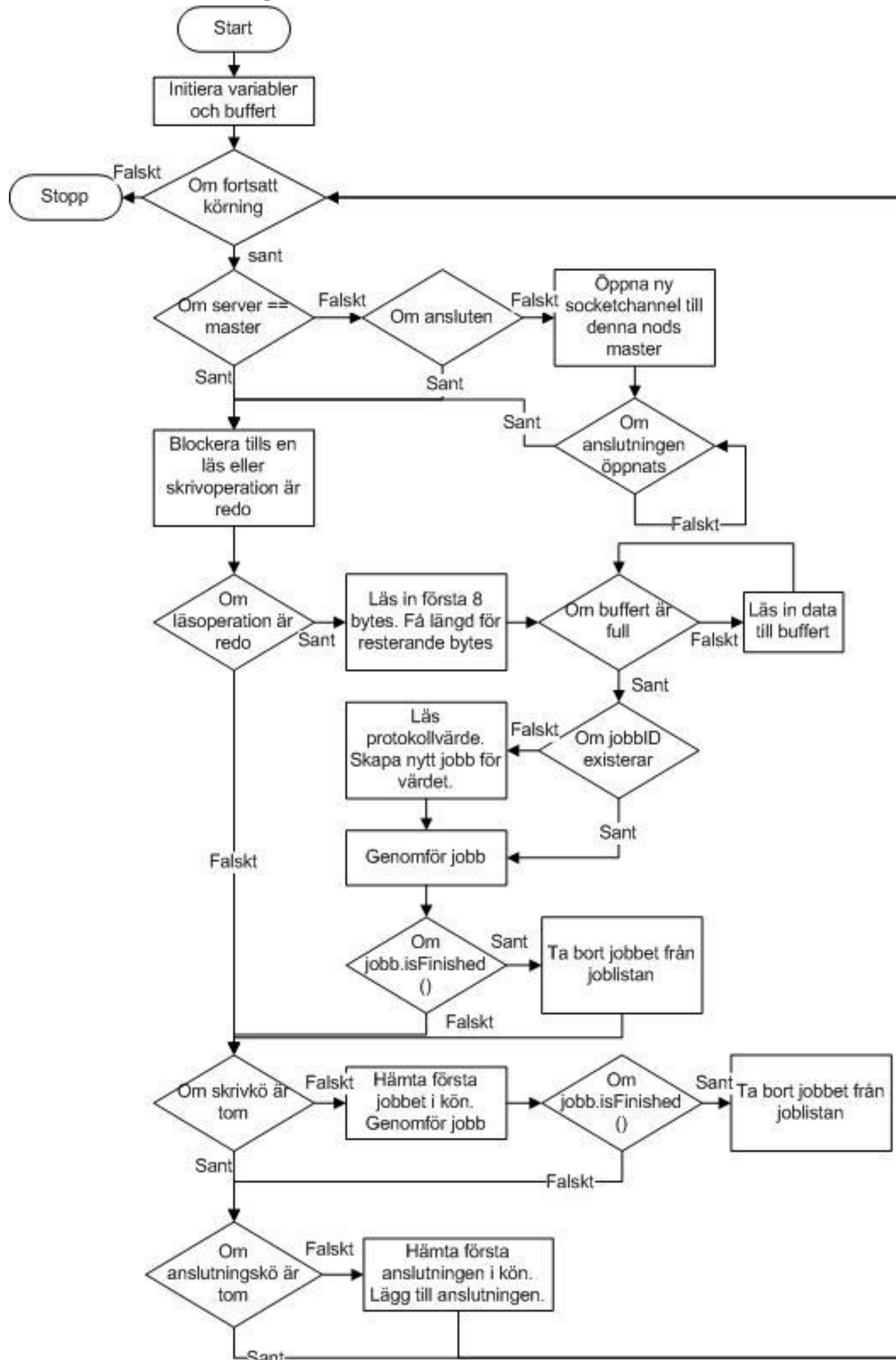
Precis som den ursprungliga nätverksmodellen använder denna nätverksmodell jobbklasser för att hantera den icke-blockerande kommunikationen. Grundprinciperna i alla jobb är definierade i klassen *AbstractJob*. Där definieras att varje jobb har ett booleanvärde som anger om jobbet är avslutat eller ej, en metod för att läsa och en metod för att skriva. Båda metoderna för in och utdata kräver en förinitierad buffert av typen *ByteBuffer* som parameter men skrivmetoden kräver också en referens till ett *SocketChannel*-objekt.

Därefter delas jobben in i ytterligare två abstrakta grupper kallade *InternalJob* och *ExternalJob* där *InternalJob* är för kommunikation inom lagringsnätverket och *ExternalJob* är för kommunikation mellan noder och klienter. Denna indelning finns eftersom ett jobb avsett för intern kommunikation inte har någon användning i extern kommunikation. Dessutom behöver interna jobb en annan typ av kontext än externa jobb.

Oavsett om ett jobb är internt eller externt följer det samma principer för hur det ska användas. För skrivoperationer gäller att jobbet använder det *SocketChannel*-objekt som inkommer som parameter vid metodanropet för att skriva data och att jobbet själv gör nödvändiga åtgärder för att datan ska bli korrekt. Ett jobb ska inte skriva mer data än vad som är den nuvarande begränsningen på bufferten som också den inkommer som parameter vid metodanropet. Vad som sedan sker är individuellt för vart jobb.
När jobbet är klart ska metoden *setFinished(boolean b)* användas för att säga åt programmet

att ta bort jobbet. Som extra kontroll ska jobbet även returnera ett booleanvärde som anger om jobbet ska fortsätta ligga i skrivkö eller ej.
För läsoperationer gäller att jobbet får sina indata från den buffert som inkommer som parameter vid metodanropet och att den kan manipulera bufferten hur den vill. Jobbet bör alltid återställa bufferten vid slutet av läsoperationen. Precis som för skrivoperationen måste metod *setFinished(boolean b)* användas för att meddela om jobbet är klart. Även denna metod returnerar ett booleanvärde vid avslut som, till skillnad från skrivoperationen, anger om jobbet behöver hamna i skrivkö eller ej.

3.3.4 Nätverkshantering



Figur 8, flödesschema nätverkshantering

Det är denna del som hanterar den grundläggande kommunikationen. Dess viktigaste syfte är att se till så att anslutningar läggs till och kopplas från på ett bra sätt samt att data läses in korrekt. Den ser även till så att de jobb som finns hanteras.

Logiken bakom denna del *(Se figur 8)* är väldigt enkel och är skapad för att hålla igång systemet och för att låta jobben sköta det finstilta arbetet.
För att den ska fungera krävs följande objekt:

- Buffert av typen *ByteBuffer*. Alla läs och skrivoperationer använder samma buffert för att minska på minnesanvändningen.
- En *Selector*[13]. Alla *SocketChannel*s är registrerade till detta objekt. Objektet håller koll på om en anslutning har data som behöver läsas in.
- En kö av typen *ConcurrentLinkedQueue* som används för inkommande anslutningar.
- En *HashMap* som tar *String*-objekt som nyckel och *InternalTrafficContext*-objekt som värde. Detta används för att hålla koll på samtliga anslutningar.
- Två objekt, *InternalTrafficHandlerChildProperties* och *InternalTrafficHandlerMasterProperties*, där endast ett kan vara aktivt under körning. Vilket bestäms av huruvida en server är master eller ej.
- En HashMap som tar *String*-objekt som nyckel och *InternalJob*-objekt som värde. Innehåller samtliga aktiva jobb.
- En kö av typen *Queue* som innehåller de jobb som behöver skriva data.

Anledningen till att *HashMap* används istället för vanliga listor är pga. att eftersom både jobb och anslutningar identifieras med hjälp av godtyckliga integervärden och eftersom jobb och anslutningar byts ut väldigt ofta så behövs en datastruktur där det kostar så lite tid som möjligt att hitta ett specifikt värde eller att lägga till och ta bort värden. Listor skulle snabbt bli ineffektiva eftersom de hade behövt genomsökas varenda gång ett värde skulle hittas. En *HashMap* kan hämta ett godtyckligt värde med en tidskomplexitet av O(1) mot en listas O(n)[14].

Klassen för nätverkshantering är en egen tråd som körs kontinuerligt tills antingen mjukvaran stängs av eller tills huvudklassen *Main* initierar *close()*-metoden hos denna klass.
Under körningens gång kan nya anslutningar läggas till i anslutningskön och nya jobb kan läggas till och hamnar då i jobbens *HashMap* samt jobbkön (Om jobb läggs till externt förutsätts det att dessa ska skriva data).

Själva genomkörningen av klassen *(Se figur 8)* börjar alltid med att en kontroll görs om den interna loopen ska fortsätta köras. Därefter, om denna nod inte är master, kontrolleras anslutningen till denna nods förälder och om anslutningen inte är upprättad så stannar tråden här tills en anslutning har upprättats.
När en anslutning väl existerar och är upprättad så går tråden vidare till en kontroll som ser om det antingen finns jobb i jobbkön i vilket fall tråden snabbt kollar om det finns några anslutningar som har läsbar data och sedan går vidare eller om det inte finns några jobb i jobbkön i vilket fall tråden stannar tills minst en anslutning har läsbar data.
I det fall att en ny anslutning eller ett nytt jobb skulle läggas till under tiden tråden väntar så

avbryter tråden sitt väntade och fortsätter körningen. Detta för att inte nya anslutningar eller jobb ska behöva vänta tills en annan anslutning har läsbar data innan de läggs till.

När den kontrollen väl är gjord går tråden vidare och börjar hantera eventuella läsoperationer den fått in. En läsoperation börjar med att 8 bytes läses in och behandlas. Anledningen är att dessa första 8 bytes (två 32 bitars integervärden) innehåller information om vilken nod som skickar data samt hur mycket data som ska skickas. När tråden vet om hur mycket data som skickas kan den läsa in precis så mycket data och sedan gå vidare till nästa steg.
Här i nästa steg genomförs en kontroll på de nästkommande fyra bytes som hämtats vilka innehåller ett jobbId. Om detta Id är noll eller om Id:t inte känns igen i jobbkatalogen läses en byte till in och behandlas som ett protokollvärde. Från detta värde bestäms sedan vilket jobb som ska användas, jobbet skapas och läggs till sist i jobbkatalogen.

Nästa steg är att de jobb som har inkommande data får behandla sin data och om de därefter anses färdiga tas de bort från jobbkatalogen.
Direkt därefter kontrolleras det om jobbkön är tom. Är den inte det så får jobbet först i kön genomföra sin skrivoperation och, om det returnerade värdet är sant, hamnar sedan längst bak i kön igen. Skulle det returnerade värdet vara falskt tas jobbet bort från jobbkön. Det ligger dock fortfarande kvar i jobbkatalogen såvida inte *setFinished()*-metoden har anropats i jobbet.

Till sist så kontrollerar tråden ifall anslutningskön är tom. Om inte så lägger den till alla anslutningar och tilldelar dem temporära Id-nummer. Dessa Id-nummer bytas ut sen när en nod har genomfört sin handskakning med sin master.

Detta förlopp beskriver hur den interna kommunikationen sker i *InternalTrafficHandler*-klassen. Den externa kommunikationen i *ExternalTrafficHandler*-klassen fungerar i stort sett likadant som den interna men med ett par skillnader:

- Bufferten har en mycket större maximal storlek. En algoritm som mäter belastningen justerar buffertstorleken efter varje iteration. Algoritmen är som följer:
  Total = 131072, a = jobbkatalog.antal, b = jobbkö.antal, c = anslutningar.antal
  $ny\ buffertstorlek = total / ((\log_2(a) + \log_2(b) + \log_2(c)) / 3)$
- Alla *InternalTrafficContext*-objekt är nu *ExternalTrafficContext*-objekt och alla *InternalJob*-objekt är nu *ExternalJob*-objekt.
- När en anslutning har data redo och inget jobb är registrerat på den anslutningen så läses endast en byte in först till skillnad från den interna kommunikationens 8 bytes. Den representerar protokollvärdet som sedan avgör vilket jobb som ska startas. Därefter behandlas övrigt data av jobbet i fråga.

4 Resultat

Resultaten för detta projekt baseras på koden för ”build_12_05_29” *(Se bilaga 1)* samt för tester utförda den 29:e maj 2012.

4.1 Funktion

Mjukvaran har under utvecklingens gång strävat efter att följa den modell som fastställdes redan tidigt i examensarbetet. Vägen till den senaste builden har däremot varit lång och krokig och har lett till en del förändringar. Nedan följer en punktlista på vad projektet är kapabelt till att göra:

- Datahanteringen kan med framgång omindexeras, integritetskontrolleras samt upprensas helt enligt tidigare beskrivningar.
- Programmet kan hantera samtida läsning. Skrivning sker under exklusivitet för att undvika datakorruption.
- Projektet kan sättas upp på olika operativsystem som t.ex. windows 7 och linux.
- Ett lagringsnätverk kan bildas, oavsett vart i världen noderna befinner sig.
- Arbetsbördan fördelas, om än ojämnt, över alla noder i lagringsnätverket.
- Klienter kan ansluta till rotservern och begära att få lagra eller hämta data. Rotservern kan omdirigera klienten till korrekt nod i lagringsnätverket.
- Lagringsnätverket kan hantera mer än en klient per nod åt gången.

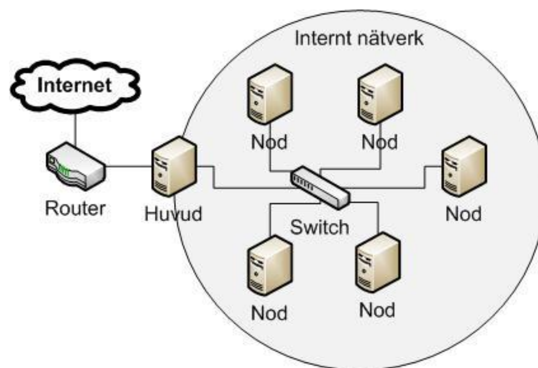
Sedan följer en lista med saker som projektet inte är kapabelt till att göra:

- Projektets nätverksdel är inte feltolerant och kan få stora delar av mjukvaran att låsa sig om ett fel uppstår.
- Klienter kan inte ta bort data eller hämta en specifik version av data (även om versionerna existerar).
- Projektet kan inte visa upp något grafiskt användargränssnitt för administratörer.
- Projektet kraschar om alldeles för många anslutningar från klienter görs samtidigt.

På det hela taget följer projektet sina mål väl men har på grund av den under omständigheterna korta tiden och omändringar i programkoden ej nått en sådan nivå att det kan kallas klart.

4.2 Drift och underhåll

Projektet har huvudsakligen körts på ett kluster installerat av handledare Dawit Mengistu på högskolan Kristianstads nätverk *(Se figur 9).* Klustret består av totalt sex datorer varav en står som huvudnod och resterande fem står som arbetsnoder. Den enda nod som har kontakt med omvärlden är huvudnoden. Detta har medfört vissa problem då mjukvaran kräver att alla noder i lagringsnätverket ska kunna nås direkt men att det i detta fall är omöjligt att nå arbetsnoderna på det interna nätverket från internet.



Figur 9, översiktsbild för datorkluster på högskolan Kristianstad.

Detta löstes genom att ett specialtillverkat routingprogram *(Se bilaga 6)* tillverkades i Java och kördes på huvudnoden i klustret. Programmet är skapat enbart för att kunna vidarebefordra anslutningar till det interna nätverket genom att lyssna på de portar som de interna arbetsnoderna lyssnar på. Skillnaden är att programmet ligger på huvudnoden och kan därför få kontakt med internet. När en anslutning väl kommer in skickar programmet den inkomna datan vidare till respektive nod på det interna nätverket.

För enkelhetens skull installerades rotnoden för lagringsnätverket på huvudnoden i datorklustret och arbetsnoderna i datorklustret blev barn i lagringsnätverket.
För att lagringsnätverket ska initieras korrekt måste rotservern startas först. Ordningsföljden därefter är oviktig då systemet delar ut indexeringsområden dynamiskt.

Under normal körning med få eller inga skriv och läsoperationer upprätthålls systemet väl. Processor och minnesanvändningen ligger på en låg och jämn nivå.
När systemet stressas med få anslutningar med mycket data blir överföringshastigheten märkbart lägre för var klient, processoranvändningen går upp kraftigt men minnesanvändningen förblir ungefär densamma. Om en klient avslutar överföringen i förtid kommer ett fel att registreras av den aktuella noden. Den temporära fil som skapas vid överföringen tas inte om hand och stannar kvar på hårddisken. I övrigt sköts alla överföringar om hand och avslutas korrekt.
Om systemet istället stressas med många anslutningar med lite data ökar processoranvändningen mycket kraftigt. Överföringarna tas om hand direkt och sparas i temporära filer som sedan behandlas i god takt. Problem uppstår då antalet anslutningar varit många och programmet inte hanterat alla filreferenser som blivit avklarade. Alla överföringar avbryts och mjukvaran stannar upp totalt.

Inga underhållsverktyg finns tillgängliga i den senaste builden av mjukvaran vilket gör en omstart av hela system nödvändig för att åtgärda ett större systemfel. En omstart måste börja med att rotservern startar, därefter är det oviktigt i vilken ordning noderna ansluter eftersom de redan blivit identifierade.

4.3 Test

Det är i huvudsak två tester som varit intressant för detta projekt. Först handlar det om att testa om nätverksmodellen fungerar och delar upp ansvaret över data över de olika noder som finns tillgängliga i lagringsnätverket. Därefter handlar det om att försöka mäta prestanda för mjukvaran och se så att det fungerar även under belastning. Dessa två tester beskrivs under vars en underrubrik.

För att genomföra testerna har en specialbyggd testklient använts. Testklienten har fyra lägen vilka är att lagra en fil, hämta en fil, lagra ordlista samt hämta ordlista. Ordlista är i detta fall en engelsk ordlista i formatet av en textfil.
Testklienten kan också mäta överföringshastighet när den skickar eller hämtar en enskild fil. Hastigheten uppmäts genom att antalet lästa eller skrivna bytes adderas ihop i en variabel som efter en sekund visas upp och nollställs igen. På så vis får programmet ett ungefärligt värde på överföringshastigheten per sekund.

4.3.1 Test av nätverksmodell

Detta test har genomförts på det datorkluster som tillhandahållits av handledare Dawit Mengistu på högskolan Kristianstads nätverk *(Se figur 9)* och via internet. Testet har bestått utav en rotserver, fyra barnnoder, den specialtillverkade routingprogramvaran samt testklienten. Testklienten ansluter i detta fall till lagringsnätverket via internet.

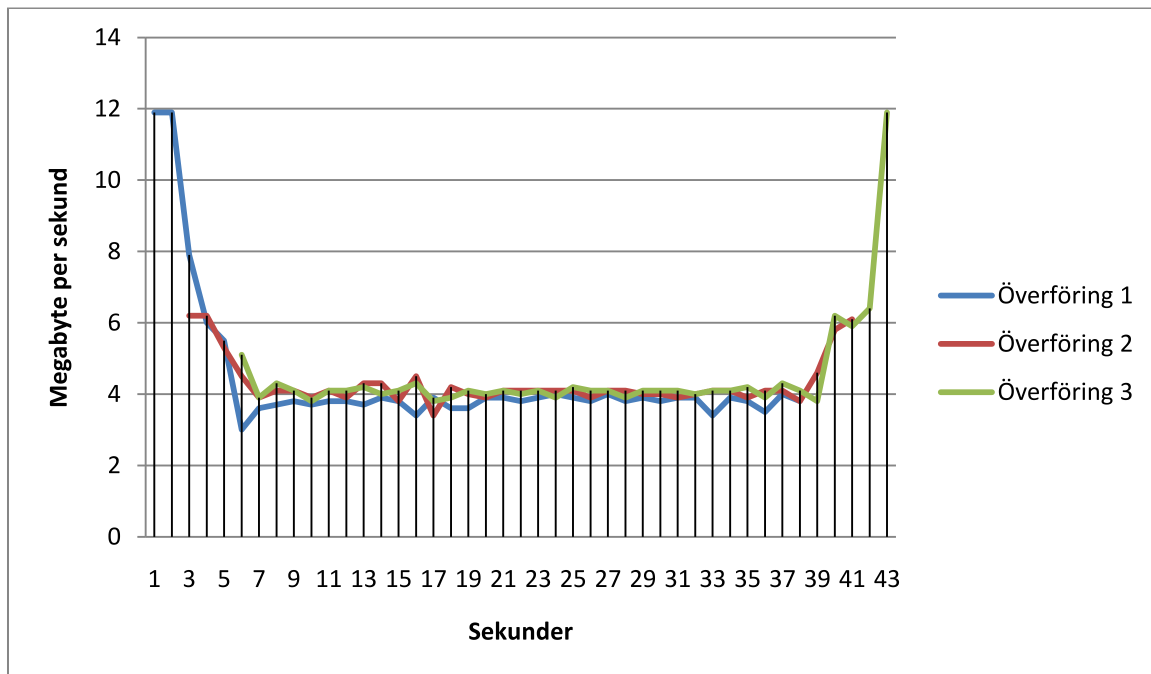
För detta test användes ordlistan i testklienten för att visa hur rotservern dirigerar om förfrågningarna om att lagra data till de noder som har rätt indexområde.
Eftersom det fanns fyra barn i lagringsnätverket lagrade inte rotservern själv någon data utan den omdirigerade alla förfrågningar till dess barn.
Genom att kontrollera varje barns lokala indexfil kunde det konstateras att nätverksmodellen fungerat alldeles utmärkt eftersom det fanns data på varje barn i lagringsnätverket samtidigt som inget barn hade kopior av samma data.

Problem som uppstod under testet var, som tidigare nämnt under rubrik 4.2 Drift och underhåll, att programmet till slut stannade upp eftersom antalet anslutningar och således antalet filreferenser blivit så många att programmet inte kunde öppna fler.

4.3.2 Test av överföringshastigheter

Detta test har genomförts på ett lokalt ethernetnätverk med en maximal överföringshastighet på 100 Mb/s (Megabit per sekund). En fil på cirka 170 megabyte användes för att genomföra mätningar. Testklienten använde nu sina funktioner för att lagra och hämta enskilda filer. Data avlästes var sekund som programmet kördes och visar överföringshastighet för var sekund. Vid detta tillfälle anslöt klienten till rotservern direkt över det lokala nätverket och inte genom internet.

För detta test användes tre instanser av testklienten. Samtliga tre körde exakt samma kommando för exakt likadana filer. En fördröjning på ca tre sekunder mellan varje start användes för att visa skillnader på överföringshastigheter vid olika belastning.



Figur 10, Diagram för test av överföringshastigheter

För den första överföringen *(Se figur 10)* började överföringshastigheten på ca 12 MB/s (Megabyte per sekund). Eftersom den teoretiska maxhastigheten för ett lokalt nätverk med en överföringshastighet på 100 Mb/s är 12.5 MB/s (100Mb/s / 8 = 12.5MB/s) så kommer överföringen upp i klart godkända hastigheter.
När så den andra överföringen slår igång så sjunker den första överföringens hastighet ner till först 8MB/s och sedan 5MB/s. Den andra överföringens hastighet börjar på strax över 6MB/s och sjunker sedan ner till den nivå runt 4MB/s som alla tre överföringar har när de körs samtidigt.
Av detta kan vi se att överföringshastigheten delas upp jämnt mellan de tre överföringarna och de går därför i samma hastighet. Så fort en överföring blev klar ökade hastigheten igen för de resterande överföringarna.

Hastighetsmässigt fungerar systemet mycket bra. Störst påverkan på den faktiska hasigheten har bandbredden till den nod som datan hämtas eller lagras på.

5 Slutsatser

Att skapa ett projekt som det här i Java har visat sig vara en hållbar idé. Tittar man på testerna för överföringshastighet så ser man att det inte är några problem att skriva till en nod i lagringsnätverket och i testerna för nätverksmodellen ser vi att det finns gott om utrymme för många anslutningar.
Däremot är det mycket som måste lösas för att mjukvaran ska bli praktiskt användbar. Framförallt måste det finnas felhantering i alla känsliga delar av mjukvaran. Som tidigare beskrivet så saknar nätverksmekanismen mycket av den nödvändiga felhanteringen vilket gör mjukvaran opålitlig.
Även vissa funktioner har ännu inte implementerats och kommer behöva bli det för att projektet ska bli komplett.

Så frågan är då, är projektet användbart i sitt nuvarande skick? Mitt svar är tyvärr nej. Det finns för många brister och en del funktioner saknas som är essentiella för att detta projekt ska vara en fullvärdig lagringslösning. Kan projektet nå dit? Ja det tror jag absolut. Projektet är helt och hållet öppen källkod och har dokumentation i form av javadoc och denna rapport. Vem som helst kan välja att göra en kopia av koden och bygga vidare på den. Även jag funderar på att fortsätta med det efter examensarbetets avslut. Det finns definitivt en marknad för den här typen av projekt och även om inte just detta projekt blir använt i slutändan så kommer kanske andra utvecklare kunna hämta inspiration från det.

6 Förslag för fortsatt arbete

Det som skulle behöva göras för att projektet ska bli en färdig lösning är:

- Skapa feltolerans och felhantering för nätverksklasserna.
- Implementera de funktioner som saknas t.ex. hämta en specifik version av data eller ta bort data.
- Skapa ett användbart användargränssnitt för att förenkla administration.
- Skapa bibliotek/drivrutiner för de mest intressanta språken.

Det finns även en del detaljarbete som kan behöva göras t.ex. bättre algoritmer för att justera buffertstorlek, mer jämlika icke-blockerande läs och skrivoperationer samt många mindre förbättringar för dataflöde och logik.

7 Referenser

7.1 Källförteckning

[1] Google. *Why app engine.* (Elektronisk) Tillgänglig: <https://developers.google.com/appengine/whyappengine>(2012-05-17)

[2] Dropbox. *About*. (Elektronisk) Tillgänglig: <www.dropbox.com/about>(2012-05-17)

[3] Dropbox. *Pricing*. (Elektronisk) Tillgänglig: <www.dropbox.com/pricing>(2012-05-17)

[4] Barret, V. (18 oktober 2011). *Dropbox: the inside story of techs hottest startup*. (Elektronisk) Tillgänglig: <http://www.forbes.com/sites/victoriabarret/2011/10/18/dropbox-the-inside-story-of-techs-hottest-startup/>(2012-15-17)

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. (2006). *Bigtable: A Distributed Storage System for Structured Data.* To appear in: OSDI'06: Seventh Symposium on Operating System Design and Implementation, Seattle, WA, November, 2006. (Elektronisk) Tillgänglig: <http://research.google.com/archive/bigtable.html>(2012-05-18)

[6] MongoDB (Elektronisk) Tillgänglig: <http://www.mongodb.org>(2012-05-17)

[7] Git software configuration management. *Git*. (Elektronisk) Tillgänglig: <http://git-scm.com>(2012-05-26)

[8] Github. *About*. (Elektronisk) Tillgänglig: <https://github.com/about>(2012-05-17)

[9] Manifesto for Agile Software Development. *Principles behind the Agile Manifesto*. (Elektronisk) Tillgänglig: < http://agilemanifesto.org>(2012-05-26)

[10] Jenkov, J. (27 augusti 2011). *Java NIO vs. IO*. (Elektronisk) Tillgänglig: <http://java.dzone.com/articles/java-nio-vs-io>(2012-05-22)

[11] Guarnacci, N. (18 juni 2009). *Java.nio vs Java.io*. (Elektronisk) Tillgänglig: <https://blogs.oracle.com/slc/entry/javanio_vs_javaio>(2012-05-22)

[12] Ritter, T. (Senast uppdaterad 30 april 1996). *The Great CRC Mystery*. (Elektronisk) Tillgänglig: < http://www.ciphersbyritter.com/ARTS/CRCMYST.HTM >(2012-05-26)

[13] Jenkov, J. *Java NIO Selector*. (Elektronisk) Tillgänglig: <http://tutorials.jenkov.com/java-nio/selectors.html>(2012-05-27)

[14] Coder-Friendly. (Senast uppdaterad 12 maj 2009). *Java collections cheatsheet*. (Elektronisk) Tillgänglig: <http://www.coderfriendly.com/2009/05/12/java-collections-cheatsheet>(2012-05-27)

7.2 Personliga kontakter

Dawit Mengistu, Universitetslektor i datorteknik, Högskolan Kristianstad

Eric Chen, Universitetslektor i datateknik/datakommunikation, Högskolan Kristianstad

8 Bilaga

8.1 Bilaga 1, build_12_05_29

Extern bilaga med filnamn build_12_05_29.zip. Ska medfölja denna rapport.
Senast skapade build-fil. Innehåller den slutgiltiga programkoden.

8.2 Bilaga 2, build_12_05_28

Extern bilaga med filnamn build_12_05_28.zip. Ska medfölja denna rapport.
Build från före linuxanpassningen.

8.3 Bilaga 3, build_12_05_21

Extern bilaga med filnamn build_12_05_21.zip. Ska medfölja denna rapport.
Första upplagan av projektet med den slutgiltiga nätverksmodellen.

8.4 Bilaga 4, build_12_05_14

Extern bilaga med filnamn build_12_05_14.zip. Ska medfölja denna rapport.
Sista builden där den ursprungliga nätverksmodellen användes.

8.5 Bilaga 5, build_12_04_28

Extern bilaga med filnamn build_12_04_28.zip. Ska medfölja denna rapport.
Projektets första riktiga build.

8.6 Bilaga 6, CustomRouter.zip

Extern bilaga med filnamn customrouter.zip. Ska medfölja denna rapport.
CustomRouter är den specialbyggda routingprogramvara som fungerar som en mellanhand mellan internet och ett internt nätverk. Det är mycket specifikt kodat och fungerar enbart i de sammanhang som är angivna i denna rapport.