CHALMERS

iSCSI-Server

Att utveckla en iSCSI-Target i Go

Examensarbete inom Högskoleingenjörsprogrammet i Datateknik

OSSIAN MADISSON

Institutionen för data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2015

iSCSI-Server
Att utveckla en iSCSI-Target i Go

Ossian Madisson

Institutionen för data- och informationsteknik
Chalmers tekniska högskola
412 96 Göteborg
Tel: 031-772 1000
Fax: 031-772 3663

Institutionen för data- och informationsteknik
Göteborg, 2015

Förord

Detta examensarbete utgör den avslutande delen av en högskoleingenjörsutbildning med inriktning på data vid Chalmers Tekniska Universitet i Göteborg. Examensarbetet utförs under 10 veckor och omfattar 15 högskolepoäng.

Dataingenjörsutbildningen är väldigt bred och ger grundläggande kunskaper inom flera områden, därför är det väldigt givande att avsluta hela utbildningen med ett större projekt där man får tillämpa en del av vad man lärt sig i skolbänken.

Examensarbetet är genomfört på ett företag som heter Ilait och är beläget i Kungälv. Jag vill tacka Ilait för möjligheten att ha fått genomföra mitt examensarbete på deras arbetsplats. Jag har lärt mig väldigt mycket under den här tiden.

Sammanfattning

Ilait är ett företag som man kan beskriva som servergrossist då de tillhandahåller diverse servertjänster, som virtuella servrar och backuptjänster, till återförsäljare. De bygger en stor del av sina tjänster på sitt eget system IVBS *(Ilait Virtualized Block Storage)* som är ett bra system men det saknar stöd för iSCSI *(Internet Small Computer System Interface).* De har haft kunder som har begärt stöd för just det och de vill därför undersöka möjligheten att erbjuda det.

IVBS är utvecklat i det ännu unga språket Go, därför är uppgiften med det här arbetet att genom att försöka utveckla ett iSCSI-system i Go undersöka om det är möjligt. För att genomföra detta arbetet behövdes kunskap om både iSCSI och Go. På vägen tillkom också ett behov av en ökad förståelse för både SCSI och Linux filsystem. Projektet resulterade i en iSCSI-server som det går att koppla upp sig mot, logga in på utan autentisering och använda för läsning och skrivning som om det var en lokal disk på klientsidan.

Abstract

Ilait is a company that provides various server-based services such as virutal server hosting and backup services. This is done exclusively for resellers. A large part of their services are built upon their own system IVBS (Ilait Virtualized Block Storage), a good system that unfortunately lack support for iSCSI (Internet Small Computer System Interface). This support is something that the company's customers have requested. Thus, Ilait wants to examine the possibilty to develop the technology to offer this.

Since IVBS is developed with the use of the still fairly young programming language Go, the main task of this project is to examine if it is possible to develop an iSCSI-system with the use of Go.

To tackle this project extensive knowledge of both both iSCSI and Go was required. Later during the process a need for extra knowledge about both SCSI and Linux filesystems arose. The project resulted in an iSCSI-server software that is possible to connect and login to without authentication. Once connected it is possible to read and write to it as if it were a local disk on the clientside.

Innehållsförteckning

Terminologi

iSCSI	*Internet Small Computer System Interface*, ett protokoll som använder SCSI-kommandon för att skicka data över ett nätverk.
SCSI	*Small Computer System Interface*, ett gränssnitt för anslutning av hårddiskar.
IVBS	*Ilait Virtualized Block Storage*, Ilaits eget lagringsystem.
GitHub	Versionshantering samt kan användas som backuptjänst, github.com.
RFC	*Request for Comments*, dokumentation om diverse protokoll och andra nätverksrelaterade objekt som hittas på ietf.org.
IETF	*Internet Engineering Task Force,* en organisation som "bestämmer" hur internet ska fungera.
PDU	*Protocol Data Unit,* namn för paketen som skickas med iSCSI.
AHS	*Additional Header Segment*, en del av en PDU.
LBA	*Logical Block Address*, en adress för att identifiera var en läsning eller skrivning ska påbörjas.
image	En avbild av t.ex. en diskenhet eller ett mjukvarusystem.
IET	*iSCSI Enterprise Target*, den populäraste öppna programvaruimplementationen av en *iSCSI-target*.
Wireshark	Wireshark är ett program som sniffar datatrafik som skickas över nätverket och med ett grafiskt gränssnitt visar upp datan för analys.
ITT	*Initiator Task Tag*. Ett unikt värde som en initierare använder för att kunna matcha respons mot *request*.
LUN	*Logical Unit Number*. Identifieringsnummer för en logisk enhet som används som lagringsenhet.
ANSI	*American National Standards Institute*. En organisation som hanterar standarder inom många områden [16].
CDB	*Command Descriptor Block*. Kan liknas vid en *header* som bär information om ett SCSI-kommando.
SSD	*Solid State Drive.* En typ av lagringsmedia.
R2T	*Ready To Transfer.* En *iSCSI-response*.

1 Inledning

1.1 Bakgrund

Ilait är ett företag som man kan beskriva som servergrossist då de tillhandahåller diverse servertjänster, som virtuella servrar och backuptjänster, till återförsäljare. De vill göra sina tjänster mer anpassningsbara och därför söker de att lägga till stöd för iSCSI i sitt lagringssystem. iSCSI är ett protokoll som skickar SCSI-kommandon mellan klient och en enhet.

1.2 Syfte

Syftet med projektet är att undersöka om företaget kan erbjuda en iSCSI-lösning utvecklad i Go som komplement till sitt nuvarande lagringsystem IVBS för att kunna tillgodose sina kunders önskemål. IVBS är en tjänst som tar emot kundens data och placerar den där det finns plats på ett sätt som optimerar serverutrymmet, men samtidigt håller datan lättillgänglig. Eftersom tjänsten IVBS är skriven i utvecklingsspråket Go så bör också detta komplement till tjänsten utvecklas i detta språk för att underlätta och effektivisera kommunikationen mellan systemen.

Projektet är en studie för att undersöka företagets möjlighet att utveckla den här kompletterande tjänsten. Studien genomförs genom att försöka utveckla ett program vars uppgift är att agera som server som åtminstone ska kunna hantera iSCSI över ett nätverk och skriva data till en lokal fil.

1.3 Avgränsningar

Projektet kommer inte att hantera någon hårdvara, endast mjukvara ska utvecklas. Eftersom projektet är ämnat för endast en student så är det inte tänkt att anslutningen till IVBS ska utvecklas. Istället ska det räcka med att via iSCSI kunna skriva till en lokal fil, men om det skulle visa sig gå oväntat snabbt och det finns tid över så är det en möjlig vidareutveckling att göra den kompatibel med IVBS.

2 Metod

2.1 Val av språk

Språket som är valt för utveckligen är Go. Detta valet grundas i att företaget var intresserade av att se en lösning i just Go, således har ingen vidare studie gjort huruvida ett annat språk hade varit ett bättre alternativ.

2.2 Arbetsmetod

Eftersom projektgruppen endast bestod av en person så behövdes inte någon daglig avstämning mellan medlemmarna i gruppen, vilket utesluter många arbetsmetoder. Tanken var från början att försöka arbeta till viss del efter metodiken Extrem Programmering, vilket innebär frekvent kommunikation med kunden, inte skapa onödiga dokument som inte leder projektet framåt, refaktorera (strukturera om och förbättra koden så den blir mer lättläst) koden ofta samt att testa varje steg på vägen innan man påbörjar nästa.

Testning har skett regelbundet på varje liten förändring i koden. Det var svårt att tillämpa automatiska tester på projektet, därför har varje del kontinuerligt testats manuellt. Om någon tidigare del av projektet kan ha påverkats av en senare förändring, så har denna tidigare del också testats. Detta har underlättat felsökning, då fel ramats in på ett oftast litet område av koden.

Arbetet har bedrivits delvis hemifrån och delvis ute hos företaget där de erbjöd kontorsplats. Ett *repository* på GitHub har använts för versionshantering och för att lätt kunna se när vissa ändringar blev användbara. Loggbok har skrivits var dag för att underlätta överblick och uppföljning utav utvecklingen under projektets gång.

2.3 Utvecklingsmiljö

Ett tips av handledaren på Ilait ledde till att utvecklingen skedde i Sublime Text 2 för att skriva koden som sedan kompilerades och kördes via terminal. Sublime Text 2 är en texteditor med en mängd bra funktioner som är användbara vid utveckling i Go.

Wireshark har använts mycket och ofta för att undersöka trafik som skickas över nätverk både för att se hur det ska se ut samt att lokalisera vad som ser inkorrekt ut. Detta har varit användbart för att bryta ner trafiken och analysera på bitnivå.

Google Drive har använts för att spara och hantera dokument som inte innehåller kod. Rapporten och andra dokument är skrivna i Google docs.

2.4 Kunskapsinhämtning

Eftersom iSCSI var okänt för mig vid projektets start så krävdes en hel del sökande på internet efter information om protokollet. Detta tog mycket tid i början. Utöver RFC-dokumentationen så är iSCSI ett ämne som är svårt att hitta mer än generell grundläggande information om.

Att programmera i Go var också något helt nytt och jag behövde spendera en del tid i början av projektet för att kunna komma igång med detta. Google har en bra basguide där man får lära sig grunderna som de kallar “A tour of Go”. Den var väldigt användbar för att kunna börja använda språket och språkets dokumentation har också blivit flitigt använd under projektet. Naturligtvis har också projektets handledare på Ilait varit till stor hjälp då internets alla källor har svikit.

Ungefär halvvägs in i projektet så visade det sig att de delar av SCSI som skulle användas av iSCSI också behövdes skrivas. Detta visade sig vara ännu ett ämne som var väldigt svårt att hitta någon relevant information om. Men så småningom hittades en SCSI-referensmanual från Seagate. Den har varit utgångspunkten för hela SCSI-delen av projektet.

3 Teknisk Bakgrund

3.1 iSCSI

3.1.1 Bakgrund

iSCSI står för *Internet Small Computer System Interface* och är ett protokoll för att hantera SCSI-kommunikation mellan enheter via ett nätverk.

Fibre Channel är en nätverksteknologi som började utvecklas i slutet på 80-talet och används för att koppla ihop och skicka data mellan olika enheter i diverse lokala nätverkslösningar [10].

I slutet på 90-talet dominerade Fibre Channel datalagring i större skala. Denna metod var snabb och effektiv men hade nackdelen att man var tvungen att fysiskt koppla ihop enheterna och den fungerade således bara på en lokal nivå. 1998 började Cisco och IBM att utveckla iSCSI som en alternativ metod för att hantera datalagring på större skala, 2003 blev iSCSI-protokollet sanktionerat av IETF *(Internet Engineering Task Force)* [1]. iSCSI var menat att bli en lättanvänd standard som utnyttjar ett vanligt IP-nätverk som kommunikationskanal. Den skulle därmed vara billig att använda och dessutom inte vara bunden till fysisk ihopkoppling utöver den vanliga nätverkskopplingen för att skapa ett nytt nätverk enbart för lagring. Men iSCSI hade nackdelar också. Fibre Channel var fortfarande snabbare än iSCSI på grund av begränsningen i hastighet för en TCP/IP anslutning, men med dagens teknologi så behöver inte detta längre vara ett problem. En annan nackdel var säkerheten vid användning av iSCSI, men ganska snabbt utvecklades stöd för flera olika autentiseringsmetoder och för kryptering så idag är som standard iSCSI säkrare än Fibre Channel.

Den faktiska datahanteringen görs genom SCSI-kommandon, vilket är en annan välkänd och lättillgänglig standard som redan var väletablerad.

3.1.2 En generell bild om hur iSCSI fungerar

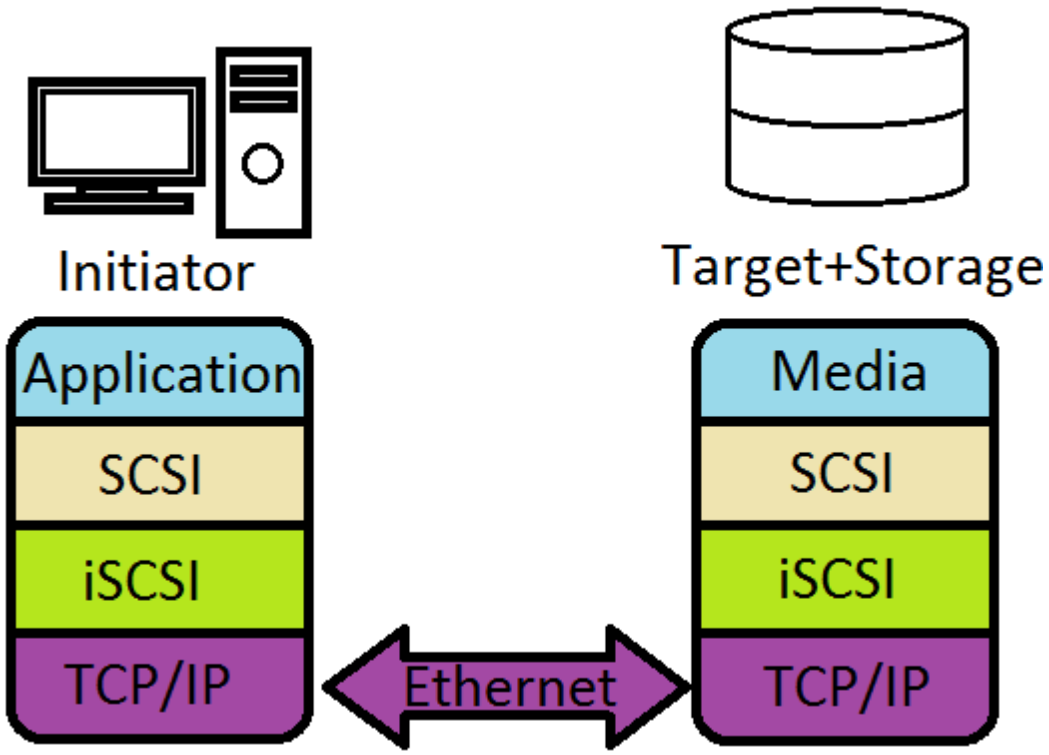
iSCSI-protokollet används genom att en initierare (på engelska *initiator)* tar kontakt med en mottagare (på engelska *target*).

En initierare är en klient som ska ansluta till en lagringsenhet och en mottagare är servern som agerar lagringsenhet. En initierare kan vara uppkopplad mot flera mottagare samtidigt och en mottagare kan vara uppkopplad mot flera initierare samtidigt. I en mottagare finns det ofta flera olika LUNs (*Logical Unit Number*) och var och en av dessa kan fungera som en disk (en lagringsenhet) för initieraren.

En initierare ansluter en session till en mottagare via båda enheternas iSCSI-system. När de är anslutna kan initieraren använda en eller flera LUNs från mottagaren genom att skicka SCSI-kommandon till den. SCSI-kommandot paketeras i iSCSI och skickas som en *iSCSI Request* via en IP-anslutning till mottagaren där ett iSCSI-system packar upp SCSI-kommandot och utför åtgärd. Antagligen så kommer mottagaren vilja skicka något tillbaka, antingen en *SCSI Respons* eller någon form av status, detta sker då som en *iSCSI Response* tillbaka till initieraren.

Enheterna kan också kommunicera med varandra utan att gå hela vägen ner till SCSI-nivån, iSCSI-protokollet har en del egna kommandon så som *Text Request* och *Text Response* [2].

Figur 3.1 beskriver hur de olika lagren hänger ihop vid användning av iSCSI.



Det finns två typer av sessioner en iSCSI-anslutning kan upprätta, *normal session* och *discovery session.* En *discovery session* används endast för att undersöka vilka mottagare som finns att ansluta till, inget annat går att göra på en sådan session.

3.1.3 PDU

Alla paket som skickas som antingen en *iSCSI-request* eller *responses* kallas för PDU *(Protocol Data Unit)*, och har en standardlängd på 48 bytes. Utöver dessa 48 bytes så kan det tillkomma AHS *(Additional Header Segment)*, *header digests, data segment* och *data digests.*

Tabell 3.1 - iSCSI-PDU (Protocol Data Unit)

Byte	**0**	**1**	**2**	**3**
Byte/Bits	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7
0	. |I| Opcode	F| Opcode-specific fields		
4	TotalAHSLength	DataSegmentLength		
8	LUN or Opcode-specific fields			
12				
16	Initiator Task Tag			
20	Opcode-specific fields			
-				
48				
-				

Tabell 3.1 beskriver hur ett *basic header segment* generellt ser ut, vilket alltså motsvarar de första 48 bytes som en PDU består av. Denna delen finns med i alla PDUs som skickas över iSCSI och i denna delen finns information om hur resten av paketet ska tolkas.

Här följer en förklaring av de olika delarna av ett *basic header segment*.

`Opcode` (kommer refereras till som svenskans opkod) är bit 3-7 av första byten i varje PDU och kan om PDUn i fråga skickas från en initierare anta värden enligt tabell 3.2, eller om PDUn i fråga skickas från en mottagare så kan den anta värden enligt tabell 3.3. Byte 2-4 innehåller extra information tillhörande respektive opkod, några bitar som agerar som flaggor för att framföra extra information eller reserverade till att alltid vara noll. Anledningen till att vissa är reserverade till att vara noll är att de inte behöver användas för den aktuella opkoden. `TotalAHSLength` beskriver om längden på en AHS om sådan existerar, annars är denna byten reserverad till noll. `DataSegmentLength` innehåller en längd på hur mycket data PDUn innehåller i ett eventuellt data segment, om data segment ej är med i aktuell PDU så är detta fältet reserverat till noll. `LUN or Opcode-specific fields` innehåller antingen den LUN som anslutningen berör eller i vissa fall något specifikt för opkoden, oftast är fältet då reserverat till noll. `Initiator Task Tag` detta är för den aktiva sessionen ett unikt nummer för att både initierare och mottagare ska kunna identifiera vilken uppgift som PDUn berör. `Opcode-specific fields` från byte 20-47 är ofta olika sekvensnummer som används för att hålla ordning på olika PDU så att de hanteras korrekt, i vissa fall också annan information så som SCSI-kommandot finns med här ifall PDUn är en *SCSI Command.*

Opkoderna är olika beroende på om de skickas från en initierare eller om de skickas från en mottagare. Här nedan följer två tabeller. Den första tabellen visar vilka opkoder som kan skickas från en initierare och den andra visar vilka som opkoder som kan skickas från en mottagare. Till vänster är hextalet som utläses från bit 3-7 av första byten i ett *basic header segment*. Till höger står kodens motsvarande kommando i klartext.

Tabell 3.2, opkoder som skickas från en initierare.	
`0x00`	NOP-out
`0x01`	SCSI Command
`0x02`	SCSI Task Management function request
`0x03`	Login Request
`0x04`	Text Request
`0x05`	SCSI Data-out
`0x06`	Logout Request
`0x10`	SNACK Request
`0x1c-0x1e`	Vendor Specific codes

Tabell 3.3, opkoder som skickas från en mottagare.	
0x20	NOP-In
0x21	SCSI Response
0x22	SCSI Task Management function response
0x23	Login Response
0x24	Text Response
0x25	SCSI Data-In
0x26	Logout Response
0x31	Ready To Transfer (R2T)
0x32	Asynchronous Message
0x3c-0x3e	Vendor Specific codes
0x3f	Reject

3.1.4 De vanligaste iSCSI-kommandona

I tabellerna ovan kan man se alla opkoder som kan skickas från båda hållen i en iSCSI-kommunikation.

Här följer en lista som beskriver de vanligaste av dessa kommandon.

- NOP-Out/In. *No Operation*, detta är ett kommando som oftast skickas för att kontrollera så att anslutningen fortfarande lever och fungerar.
- SCSI Command/Response. En av de huvudsakliga syftena med iSCSI, att kunna skicka SCSI-kommandon och responser, i en *SCSI Command*-PDU så är alltid byte 32-47 ett SCSI-CDB *(Command Descriptor Block).* Det är på dessa 16 bytes som det underliggande SCSI-kommandot identifieras och om kommandot använder sig av mer data utöver dessa 16 bytes så är det medskickat som data segment i iSCSI-PDUn.
- Login Request/Response. Detta är pecis vad det låter som, en initierare begär att få logga in på en mottagare och får ett svar.
- Text Request/Response. En PDU med ett data segment innehållande **"key=value"** par, flera par kan skickas med samma PDU. Detta används vanligen när man försöker etablera en anslutning och förhandlar om olika funktioner och värden som ska användas i anslutningen, t.ex. om man ska använda AHS eller ej.
- SCSI Data-Out/In. Kommandot som används för att läsa eller skriva data från eller till en LUN på mottagaren, mängden data som kan skickas på en gång är naturligtvis begränsad till ett värde som är överenskommet vid upprättning av anslutningen. Om mängden data som ska skrivas till en LUN överstiger gränsen så kommer mottagaren starta en sekvens där den skickar en R2T för att vara redo på att ta emot flera paket med data och skriva dessa på den i *Data-Out*-kommandot begärda adressen.
- Ready to Transfer (R2T). Detta används som beskrivet ovan i *SCSI Data-Out*, i R2T finns ett värde på hur många block av data mottagaren kan ta emot innan initieraren måste vänta på en ny R2T för att skicka mer data. När mottagaren har skrivit all data som den sa att den kunde ta emot i R2T så skickar den en ny R2T och säger att den kan ta emot mer och återupprepar denna processen tills den har skrivit all data som initieraren ville skriva i SCSI Data-Out PDUn.
- Vendor Specific codes. Olika implementationer av iSCSI kan lägga till extra funktionalitet med fler opkoder, då används dessa koder.

3.2 SCSI

3.2.1 Bakgrund

SCSI står för *Small Computer System Interface* och är ett gränssnitt som används för anslutning av hårdvara till datorer, t.ex. hårddiskar [15]. 1979 påbörjades utvecklingen av det som skulle komma att bli SCSI, det var företaget Shugart Associates som började utveckla det under namnet SASI *(Shugart Associates System Interface).* Utvecklingen tog olika vändningar under några år och till slut 1986 blev SCSI en godkänd standard av ANSI *(American National Standards Institute)* [3].

SCSI är ett numera ganska gammalt gränssnitt för att ansluta hårdvara så som hårddiskar och CD-läsare till datorer men det har vidareutvecklats med åren. Idag används inte gränsnittet längre så mycket i personligt bruk utan har till största del blivit ersatt av USB *(Universal Storage Bus)* [4] och SATA *(Serial Advanced Technical Attachment)*. Men gränssnittet används fortfarande vid serverhantering och datalagring på större skala.

3.2.2 CDB (Command Descriptor Block)

En SCSI-CDB kan liknas vid ett *basic header segment* för iSCSI som beskrivet ovan i 3.1.3. I en CDB finner man nödvändig information för att veta hur resten av datan ska hanteras och vilken typ av operation som ska genomföras. Här nedan följer en tabell som visar hur en CDB är konstruerad.

Tabell 3.3 - SCSI-CDB (Command Descriptor Block).

Byte	**0**	**1**
Byte/*Bits*	*0 1 2 3 4 5 6 7*	*0 1 2 3 4 5 6 7*
0	Opcode	Misc. CDB Info.
2	Logical Block Address (if required)	
4		
6	Additional CDB Data (if required)	
8		
10	Transfer Length, Parameter List Length and Allocation Length	
12		
14	Misc. CDB Info.	Control

Tabell 4 ger en generell bild av en 16 bytes CDB, det finns mindre CDB som också används men när man använder iSCSI så är det 16 bytes CDB som används.

De flesta fälten i CDB är beroende av vilken opkod som tillhör, om det är ett läs- eller skriv-kommando så måste en LBA *(Logical Block Address)* finnas med så att enheten vet var den ska läsa. Om man ser till rena SCSI-operationer, alltså opkoder till en SCSI-CDB, så finns det väldigt många fler operationer till SCSI än det finns till iSCSI [5].

3.3 Go

3.3.1 Bakgrund

Tre herrar vid namn Robert Griesemer, Rob Pike och Ken Thompson började i september 2007 att designa vad som idag är ett ungt och framgångsrikt programmeringspråk. Anledningen till att de började utveckla språket var att de ansåg att det saknades ett språk som var modernt nog att fungera effektivt med dagens teknologi. Enligt deras åsikt var man tvungen att välja mellan effektiv kompilering, effektiv körning eller att det skulle vara lätt att programmera. Det var inte hållbart att behöva välja endast en av dessa egenskaper när man valde ett språk att programmera i. Vid mitten av 2008 så blev projektet ett heltidsprojekt. Senare samma år tillkom en man vid namn Russ Cox till projektet och tillsammans tog de det vidare från prototyp till verklighet. Tionde november 2009 släpptes Go som ett publikt *open source* projekt och sedan dess har språket hela tiden utvecklats till vad det är idag [6].

3.3.2 Grundläggande funktionalitet

Vid en första anblick skiljer sig inte Go speciellt mycket från andra stora språk så som Java. Vilket innebär att om man sedan tidigare är erfaren med Java eller andra liknande språk så är det lätt att förstå hur Go fungerar. Precis som i andra språk kan man importera andra paket för att hämta funktioner som inte är skrivna i samma källkod. Visst finns det smådetaljer i syntaxen som skiljer sig från andra språk, t.ex. hur man deklarerar en variabel i Go. Deklarera en variabel i Go går att göra på flera sätt, här följer två exempel. Båda följande rader gör samma sak, alltså skapar en variabel i med typen int och ger den värdet 3.

```
Var i int = 3
i := 3
```

Förutom vissa skillnader i syntaxen så är Go ett språk som optimerat minneshantering på ett sätt som gör att det är lättare att skriva program som inte är krävande att köra. Ännu en skillnad mot andra språk hittas i hur man kan köra funktioner parallellt med varandra samt synkronisera kommunikation mellan dessa. Detta görs genom användande av `Go routine` samt `Go channel`. När man anropar en funktion som en `Go routine` så körs denna som en separat process och programmet exekverar det som kommer efter funktionen istället för att t.ex. vänta på ett resultat. Ibland kan det vara behändigt att invänta att en `Go routine` har genomfört något innan man kan vill gå vidare med någon annan funktion. Detta är något man kan åstadkomma genom att använda en `Go channel`. En `Go channel` är en kanal som man skapar som sedan går att skrivas till och läsas från. På så vis kan man vänta på att en `Go routine` ska skriva något på en given `Go channel` innan programmet exekveras vidare. Det kan då vara smart att sätta en maximal tid att vänta innan någon annan åtgärd tas, annars kan man riskera att sätta hela programmet i deadlock.

I Go finns också ett annorlunda sätt att hantera `arrays` på, nämligen med `slice`. En `slice` är som en `array` fast den får inte någon bestämd längd när den deklareras, istället får den en maximal kapacitet och kan då fyllas till alla tänkbara längder mellan 0 och den maximala kapaciteten. Vanliga `arrays` finns också i Go men eftersom `slice` är en mer flexibel variant så används den ofta istället för en `array`.

Detta var några av fördelarna och skillnaderna med Go, naturligtvis är det fler saker som skiljer sig mellan Go och andra språk. För den som är intresserad av att fördjupa sig mer så är Go ett mycket väldokumenterat språk med många bra introduktionsguider [7][8].

3.4 IVBS

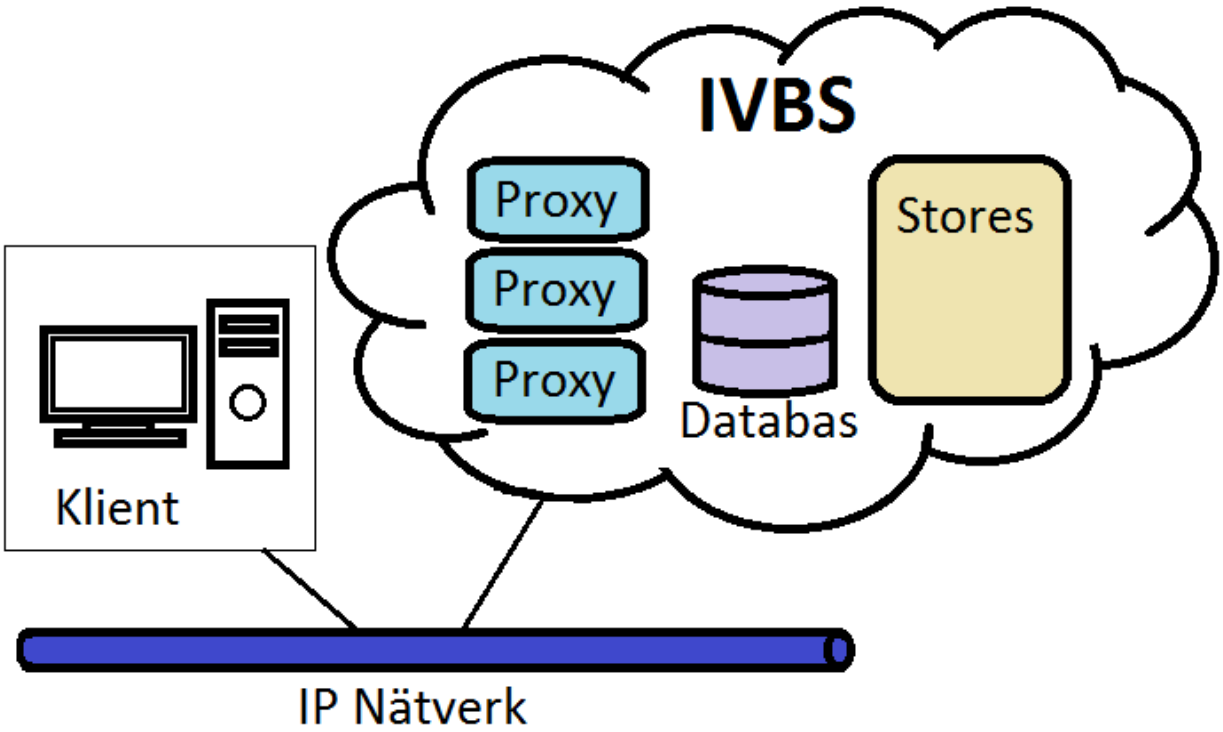
3.4.1 Bakgrund

IVBS står för *Ilait Virtualized Block Storage* och är ett lagringssystem som utvecklats av Ilait för att de var missnöjda med de alternativ som fanns på marknaden och de ville ha ett bättre och säkrare system. Utvecklingen började som ett examensarbete på Chalmers 2011 initierat av detta projekts handledare som sedan dess fortsatt utveckla systemet. IVBS används för att spara stora mängder data på lagringsmedia samtidigt som den ska förbli lättåtkomlig.

3.4.2 Struktur

Här följer en kortfattad beskrivning av hur IVBS fungerar. Följande bild är tänkt att göra det tydligare för läsaren att hänga med på beskrivningen och få en överblick av IVBS.

Figur 3.2. En överblick av IVBS.



Klienternas förfrågningar i IVBS behandlas i en av flera proxyservrar. I proxyservern sköts autentisering och kontroller både mot klient och mot databas. Om en proxyserver skulle gå ned av någon anledning så ansluter klienten automatiskt till en ny som om inget hade hänt. *Images* struktureras i databasen och proxyservern frågar databasen var den kan hitta datan som den söker och sedan kontaktar den lämplig lagringsserver för att utföra åtgärd på datan. Lagringsservrarna är den delen där den faktiska skrivningen och läsningen av data till fysisk media i form av mekaniska diskar eller SSD *(Solid State Drive)* beroende på hur frekvent datan behöver kommas åt.

Datadeduplicering är en metod för att lagra data effektivt, den fungerar genom hitta återkommande data och spara hur många gånger datan förekommer samt var istället för att skriva all data flera gånger [11]. IVBS delar upp all data i 4MB stora block, så kallade `extents`. Systemet använder sig av *datadeduplicering* för att att effektivisera lagring av data. Detta åstadkoms genom att identiska `extents` identifieras och tas bort för att spara utrymme men en referens till `extenten` sparas i databasen. På så vis kan flera olika *images* använda samma `extent` genom att de pekar på samma datablock. Alltså skrivs ingen mer data än en liten referens i databasen så länge det inte är ny unik data som ska lagras. Detta gör att man sparar mycket fysiskt minne samt att delar av data kommer att vara snabbare att komma åt då det lagras på snabbare media om det används ofta.

3.4.3 IVBS-Protokollet

IVBS struktur påminner lite om iSCSI men med något färre och annorlunda funktioner. Varje IVBS-paket börjar med en *header* med lite information om resten av paketet samt hur mycket data som medföljer.

Vanliga paket i IVBS är:

- *Greeting.* Detta används för att upprätta en anslutning mellan en klient och en proxy.
- *Login.* Används för att etablera en autentiserad anslutning och innehåller användarnamn och lösenord, denna processen behövs för att många andra funktioner i IVBS ska bli användbara.
- *List-Proxies.* Detta är ett paket som frekvent skickas från IVBS till dess klienter med en lista på tillgängliga proxies.
- *Mount*. Ett kommando som skickas från klienten för att göra en *image* redo för skrivning och läsning.
- *Läs och Skriv.* Dessa paket skickas med kommandon till IVBS med information om en aktuell *image* som ska läsas från eller skrivas till.

4 Genomförande

4.1 Uppstart

Innan projektet kunde starta på riktigt skapades en grov planering med ett preliminärt tidsschema och olika delar som behövde göras. Ilait bistod med kontorsplats ute hos företaget så en del av arbetet har tagit plats där.

Eftersom både iSCSI och Go var helt obekant innan projektets början så gick all tid under första veckan till att läsa på om desamma. E-boken *An Introduction to Programming in Go* var till stor hjälp för att komma igång med Go [7].

Information om iSCSI hämtades från olika artiklar och olika perspektiv för att få en klar helhetsbild av hur det används och kan användas. Med ökad förståelse för vad iSCSI är så var RFC-dokumentationen den klara fortsättningen för att få en djupare förståelse för hur det faktiskt fungerar [2].

Efter några dagar kunde den faktiska utvecklingen ta fart och ett grundprogram skapades delvis för att utforska Go lite mer men också för att lägga grunden till vad som skulle bli slutprodukten av arbetet. Det började som den simplaste av servrar som helt enkelt bara lyssnade efter en anslutning på en port.

4.2 Hjälpmedel

Utvecklingen har skett på en laptop med Ubuntu 14.04 LTS, detta valet baserades bara på tidigare preferenser. Som terminalhanterare i Ubuntu valdes Terminator istället för Ubuntus standardterminal för att Terminator har bättre möjligheter för anpassning av olika terminalfönster i samma process. Handledaren hos Ilait tipsade om Sublime Text 2 som textredigerare för att skriva kod. Go kod kompileras via terminal så ingen extra utvecklingsplatform har behövts.

För att använda sig av iSCSI krävs en mottagare och en initierare. Detta projekt fokuserade på utveckligen av mottagaren och har utvecklats mot den initierare som har blivit standard att använda i Ubuntu. Initieraren är en öppen programvara som heter *open-iscsi* [8]. Den var lätt att installera och konfigurera för att köra via terminal med ett fåtal kommandon [9].

Eftersom projektet var att utveckla något som redan fanns rent praktiskt men inte rätt anpassat efter behovet så var det naturligt att undersöka de produkter som redan fanns på marknaden för att hantera problemet. Således installerades IET (*iSCSI Enterprise Target)* för att testköras och analyseras. IET är en öppen programvara med mål att ha professionella funktioner och fungera bra som en riktig mjukvara i företagsmiljö.

Till en början analyserades paketen genom att skriva ner den mottagna datan i terminalen eller till en fil för att kunna se vad som skickats eller tagits emot mellan mottagare och initierare. Detta visade sig snabbt bli väldigt problematiskt då datan blev komplicerad och svårläst, vilket ledde till att Wireshark kom in i bilden. Wireshark är ett program som sniffar datatrafiken som skickas över nätverket och med ett användarvänligt gränssnitt visar upp datatrafiken och gör det lätt att på ett effektivt sätt kunna analysera de olika datapaketen. Det har många gånger under utvecklingen funnits ett behov av att analysera data som skickas

mellan IET och *open-iscsi* för att kunna se hur det skiljer sig mot det som skickades från den utvecklade programvaran. Med hjälp av Wireshark gick det många gånger bra att se vilken data som skickats som svar samt hur den var paketerad, vilket har varit mycket viktigt vid felsökning.

Värt att nämna är också att då utvecklingen har skett med både initierare och mottagare på samma enhet så har loopbackadressen *127.0.0.1* använts för kommunikation. På så vis har iSCSI-trafiken kunnat avläsas som om det skedde mellan två enheter över ett lokalt nätverk eller över internet.

4.3 Erfarenheter

Inte helt oväntat så har det under arbetets gång dykt upp väldigt många både små och stora problem.

Ett problem återkom tre gånger och krävde en ny lösning varje gång, problemet i fråga var att svaret som skickades på anslutningen inte togs emot eller ens dök upp i Wireshark. Om det inte dyker upp i Wireshark så beror det antingen på att det är av ett inkorrekt format och Wireshark känner inte igen att det tillhör något bekant protokoll. Första gången detta inträffade var ganska tidigt under utvecklingen innan stöd för utfyllnad hade lagts till så endast paket vars längd av en slump var jämnt delbara med 4 kunde identifieras och tas emot av initieraren. Den gången var lösningen helt enkelt att lägga till en supportfunktion som undersökte datalängden och justerade med att lägga till 1-3 bytes extra vid behov.

Den andra gången detta fel inträffade berodde det på att opkod-fältet fylldes med en inkorrekt kod som inte fungerade med hur resten av paketet var strukturerat. Detta ledde till att frågan inte kunde tolkas. Tredje gången samma problem uppstod så var det några bitar som hade fått fel värde i ett annorlunda svar på grund av en felaktigt skriven funktion.

Ett annat stort problem som hindrade utvecklingen ordentligt var att lösa hur SCSI-kommandon skulle tas om hand. Några dagar gick åt till att undersöka olika drivrutiner och standardprogram i Linux som skulle kunna ta emot SCSI-kommandon och utföra dessa. Men för att kunna använda dessa kommandon måste filen som iSCSI ska skriva till vara installerad som en disk redan och detta förstör lite syftet med att använda iSCSI eftersom mottagaren ska kunna använda en fil och hantera den som om det vore en disk. Lösningen för SCSI-kommandon visade sig behöva ett helt annat tillvägagångssätt, nämligen att skriva eget stöd för att hantera SCSI också. Vid den här situationen ökade arbetsbelastningen på projektet mer än vad som var tänkt från början då SCSI också var något nytt men som tur var så har iSCSI och SCSI många likheter i utformningen. De stora skillnaderna är att SCSI-kommandona är mycket mindre och mer lika varandra än iSCSI men däremot så finns det väldigt många flera än det gör till iSCSI. Men endast ett fåtal SCSI-kommandon har implementerats så att iSCSI ska kunna fungera och göra sina viktigaste funktioner.

Redan tidigt märktes det att vissa paket inte blev besvarade trots att de klart och tydligt fanns med i Wireshark och det visade sig då vara att programmet råkade läsa in två paket samtidigt och då hamnade det andra paketet i följden som data till det första och tolkades inte som en egen PDU. Till en början hjälpte det att låta inläsningen från anslutningen pausas i 10 millisekunder mellan varje paket, men när det var dags för större dataleveranser så gick inte detta längre. Detta ledde till en finurlig lösning som resulterade i att efter en inläsning så undersöks storleken av datalasten som hör till ett paket och så klipper den av den inlästa datan vid paketets slut och repeterar med nästa del.

Det sista riktigt stora problemet som satte stopp i utvecklingen var i slutet av arbetet när det var dags att lägga till funktionalitet för skrivning till fil genom mottagarmjukvaran. Implementeringen av SCSI för skrivning såg ut att fungera korrekt och R2T skickades för att be om mer data att skriva när det var läge för det. I Wireshark såg allt ut som det skulle och det blev inga fel när filen formaterades som en disk men den gick varken installera som en disk eller bygga filsystem på filen. Detta var ett problem som tog dagar att lösa men lösningen i sig var väldigt simpel och uppenbar när den väl var på plats. Programmet tolkade LBA som en adress i bytes och inte i block, vilket innebar att data skrevs över vid nästa skrivning för att skrivningen skedde på fel plats. T.ex. om 10 block ska skrivas med start på LBA 0 och 10 block till ska skrivas efter det med start på LBA 10. Som standard i iSCSI är varje block 512 bytes så motsvarande adresser att skriva på i bytes blir då 0 och 5120. Istället skrevs alla 10 blocken i princip på varandra och nästa block började också skriva på nästan samma ställe. Detta misstag ledde till mycket efterforskning i Linux filsystem och hur filer kan installeras som diskar. Men felet visade sig vara lättåtgärdat och efter det fungerade allting utomordentligt bra.

Naturligtvis var det en hel del mindre fel och buggar på vägen men de flesta åtgärdades ganska snabbt och orsakade inte någon större skada i längden eftersom manuell testning har genomförts frekvent på alla små och stora förändringar.

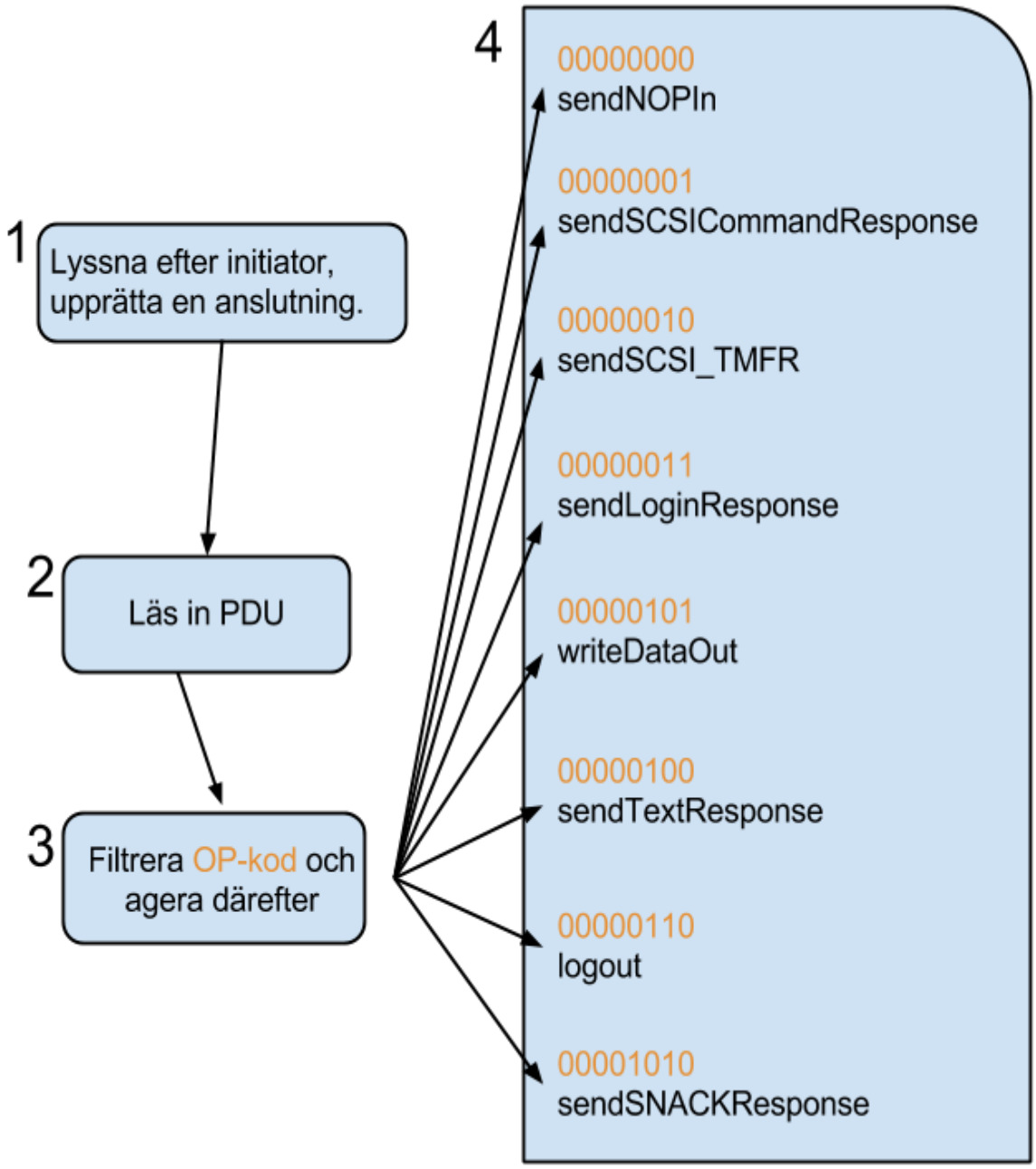
5 Beskrivning av programmet

5.1 Programstruktur

5.1.1 Överblick av hela programmet

Hela programmet som är utvecklad till att vara en iSCSI-mottagare har en grundstruktur som bygger på en lösning i 4 delar. På så vis att programmet tar emot PDUer från en initierare, hanterar datan, bygger ett svar och skickar tillbaka till initieraren.

Figur 5.1. Övergripande mottagarstruktur.



Figur 5.1 visar i tre steg hur programmet jobbar. När en anslutning till en initierare har upprättats så kommer programmet i steg 1 att lyssna efter en anslutning via TCP. När en

anslutning är upprättad så kommer programmet i steg 2 läsa in varje PDU, eventuellt formatera om längden om t.ex. flera paket läses in samtidigt. Efter detta skickas paketet vidare i steg 3 och första byten läses av och filtreras så att en opkod kan utläsas. För varje opkod anropas i steg 4 en egen funktion för att hantera datan i paketet enligt protokollets riktlinjer. Alla dessa funktioner är i grunden ganska lika varandra, men utför olika saker och hanterar inkommande eller utgående data på olika sätt. Varje funktion avslutas med att de skriver ett svar till anslutningen oavsett vad funktionen har genomfört för åtgärder med undantag för `writeDataOut` som är skrivfunktionen. Eftersom iSCSI använder sig av R2T och därför skriver flera paket efter varandra i en sekvens så skickar `writeDataOut` bara ett svar om all den totala datan för en sekvens är skriven eller om den är redo att ta emot en ny del av sekvensen av datapaket att skriva.

Steg 2 till 4 repeteras så länge en anslutning är aktiv.

5.1.2 Anslutningen

Mottagaren består av två delar, `main` och `target`. `Main` är delen som körs igång som ligger och väntar på en anslutning från en initierare. Vid exekvering så läses en konfigurationsfil där användaren har ställt in lämplig adress och port för att lyssna efter anslutningar på. I testfallet som körts under utvecklingen av programmet har detta då varit *localhost (127.0.0.1)* och port *3260* ty detta är standardporten för iSCSI-protokollet.

När en initierare ansluter på rätt port så accepteras anslutningen och en serverinstans startas, alltså en mottagare i detta fallet `target`. Mottagaren startas via en `Go routine` vilket innebär att den körs som en egen process [12] och `main` körs fortfarande och lyssnar efter anslutningar. Detta gör det möjligt för initieraren att logga ut eller bryta anslutningen och försöka ansluta igen utan att hela mottagarprogrammet måste startas om. Stöd för att ansluta flera initierare samtidigt är i skrivande stund ej testat men i teorin ska det fungera utan förhinder. Om man ansluter flera initierare samtidigt kommer andra delen av programmet få problem då läsning och skrivning inte hanterar möjligheten för flera initierare samtidigt än.

5.1.3 Inläsning av PDU från anslutning

När en serverinstans har startats genom att `target` har anropats så börjar också denna med att läsa konfigurationsfilen för att få fram diverse inställningar som användaren vill använda. Viktigaste i detta fallet är vilken eller vilka LUNs som ska användas. Det går också skriva in andra anslutningsspecifika saker t.ex. om data måste skickas i rätt ordning eller vilken autentiseringsmetod som ska användas. Detta är då saker som initierare och mottagare kommer överens om vid upprättningen av en session. I skrivande stund så accepterar mottagaren nästan alla initierarens inställningar och bryr sig i många fall inte om vad användaren har skrivit i konfigurationsfilen mer än vilken LUN som ska användas. Inställningarna som lästs in från konfigurationsfilen sparas i en `Go map` vilket är en standardtyp i Go som är dess implementation av hashtabeller [13].

Efter att konfigurationsfilen är inläst så skapas en `slice` vilket är som en mer flexibel `array` som inte behöver ha någon bestämd längd när den skapas. Den används ofta istället för `arrays` i Go [14]. En `slice` som heter `inpBuf` *(input buffer)* skapas med en kapacitet av 48 bytes eftersom headerlängden på en iSCSI-PDU är 48 bytes lång. Denna `slice` är tänkt att användas för att temporärt lagra alla inkommande PDUer. Sedan startas en loop som en `Go routine` där data läses in till `inpBuf` och eftersom denna är begränsad till 48 bytes så lagras bara headerdelen. För att gå vidare så undersöks segmentet av headern som kallas

`DataSegmentLength`, där det framkommer om det efter headern följer någon data och också hur mycket om så är fallet. Om det inte följer någon data efter headiern skrivs `inpBuf` till en `Go channel` som heter `inpCh` *(input channel)*. En `Go channel` är en kanal som kan skrivas till för att läsas från andra delar av programmet [12].

Om det visar sig att det följer en del data efter headern skapas en ny `slice bigInpBuf` *(big input buffer)* med kapaciteten 52+`DataSegmentLength`. Kapaciteten är bestämd av 48 för PDU-headerns längd plus 4 för att det ska finnas plats för eventuell utfyllnad plus längden av datasegementet givetvis. För att hämta datan så skapas ännu en `slice secondInpBuf` *(second input buffer)* med kapacitet motsvarande mängden data som ska läsas in, sedan läses resten av PDUn från anslutningen. `inpBuf` och `secondInpBuf` skrivs ihop efter varandra i den större `slicen bigInpBuf` och därefter skrivs `bigInpBuf` till samma `Go channel` som nämns ovan, `inpCh`.

Hela ovanstående process genomförs för att undvika att flera olika PDUer läses in tillsammans och på så vis förstör exekveringen.

5.1.4 Hantering av header

Som ovan nämnts i 5.1.1 så arbetar programmet med att hela tiden svara på PDUer från en initierare, så kallade *requests.* Inläsningen av nya PDUer sker som en egen process hela tiden när anslutningen är upprättad och huvuddelen av `target` hämtar data från `inpCh`. Man kan beskriva den här delen som att den lyssnar på `inpCh` och när något finns att hämta där så hämtas det och genom en simpel `switch` så kallas olika funktioner beroende på vilken opkod som ligger i headern.

En `slice` som heter `buf` skapas med kapacitet motsvarande längden på den hämtade PDUn. Därefter konverteras `buf[0]` (den första byten av PDUn) till `int64` för att sedan konverteras igen till en sträng som representerar värdet av första byten i binär form men av typen sträng, t.ex. **"101"**. För att få den korrekta opkoden så ska bara bit 3-7 användas och därför behöver strängar som **"10000011"** filtreras till **"00000011"** för att kunna jämföras korrekt, även **"11"** behöver ändras till **"00000011"**. Detta sker genom ett anrop av funktionen `func makeOpCodeFiltered(firstByteString string) string {}`, som returnerar den filtrerade strängen redo för att jämföras med de möjliga opkoderna.

När opkoden är filtrerad så återstår bara att jämföra den med alla de möjliga opkoderna som en initierare kan skicka och anropa lämplig funktion. I alla funktioner skickas anslutningen att skriva till samt `slicen buf` med som variabler. De flesta funktioner körs som vanliga funktioner och programmet väntar då på att ett svar har skickats innan den hämtar nästa *request*, undantagen är `sendSCSICommandResponse` och `writeDataOut` som körs som `Go routines` istället. Eventuellt skulle vissa andra kunna köras som `Go routines` också i en senare version.

5.1.5 Grunden i ett svar

Alla de olika funktionerna som anropas efter att opkoden tolkats bygger ett svar baserat på RFC dokumentationen [2] för iSCSI. För var och en av dessa funktioner finns det en `struct` med en `array` för varje fält med definierad längd och en `slice` för data. En av de kortare av dessa är till *Text Response* och ser ut som följer.

```
type textResponse struct{
      op [1]byte
```

```
        reserved1 [3]byte
        totalAHSLength [1]byte
        dataSegmentLength [3]byte
        lun [8]byte
        itt [4]byte
        ttt [4]byte
        statSN [4]byte
        expCmdSN [4]byte
        maxCmdSN [4]byte
        reserved2 [12]byte
        data []byte
}
```

Ovanstående utdrag ur koden är alltså grunden för ett *Text Response* som i funktionen `sendTextResponse` fylls med värden. `op` fylls med `0x24` som är opkoden som säger att det är ett *Text Response*, `reserved1` och `reserved2` är fält som alltid är tomma i den här typen av svar. `totalAHSLength` är också alltid tom eftersom programmet inte använder sig av *Additional Header Segment.*

`dataSegmentLength` är 3 bytes som fylls efter att datafältet har blivit fyllt, programmet undersöker längden på datan och skriver in den här för att initieraren ska veta hur mycket data som följer i slutet av paketet.

`lun` är ett värde som beskriver vilken LUN svaret tillhör, `itt` fylls med samma värde som i *requesten* för att initieraren ska kunna verifiera vilken *request* som svaret tillhör. `ttt` fylls alltid med `0xFFFFFFFF` vid den här typen av svar. `statSN`, `expCmdSN` och `maxCmdSN` står för *Status Sequence Number, Expected Command Sequence Number* och *Maximal Command Sequence Number.* Detta är iSCSIs sätt att sortera kommunikationen för att dels hantera att alla paket kommer fram på båda hållen samt att inte överösa mottagaren med för många kommandon samtidigt genom att mottagaren med varje svar skickar ett maximalt sekvens-nummer som ökar för varje gång mottagaren skickar ett svar. `data` är alltså datalasten som kan skickas med, i just detta fallet består datan av strängar på formen *“*`key=value`*”.*

En motsvarande `struct` finns för alla de andra svaren som kan skickas också med lite varierande fält beroende på svar, men de är i grunden märkbart lika. När dessa fälten är fyllda med lämpliga värden så skrivs de ihop till en `slice` och skickas ut på anslutningen till initieraren.

5.1.6 Att bygga ett svar

Alla funktioner bygger svaren på samma sätt, men de hanterar vad som placeras i de fält som är unika för svaren på olika sätt. Alla funktionerna börjar med att skapa en variabel av typen lämplig `struct` för just det svaret. Vissa fält är alltid samma för `structen` och fylls därför direkt t.ex. `op` och `itt`. Variablerna *`statSN`, `expCmdSN`, `maxCmdSN`* ligger som globala för anslutningen och ökar för varje skrivet svar.

I början av varje svarsfunktion skapas också en `slice` som heter `responseBuilder` för typen byte med kapacitet motsvarande det största möjliga svaret sessionen tillåts att skicka. Denna används i slutet av funktionen för skriva ihop alla fälten i `structen` till en `slice` som

sedan ska skrivas till anslutningen. För att fortsätta med ovanstående exempel i 5.1.5 kommer här en beskrivning av hur `sendTextResponse`-funktionen hanterar sin uppgift.

Utöver det som alla svaren gör så läses data in i en `slice` som heter `inData` av typen byte och med kapaciteten motsvarande mängden data. Sedan skapas en `Go map` som heter `keyValuesMap` som ska användas för att lagra och hantera den inkommande datan. Ett *Text Response* skickas alltid som svar till ett *Text Request* som är en PDU som alltid innehåller data på formen `"Key=Value"` strängar. Därför sparas dessa i en `Go map` av typen `["key"]="value"`. När den inkommande datan är lagrad i `keyValuesMap` så loopas denna över för att för varje `"key"` undersöka värdet. Ett exempel på `"Key=Value"`-par är `"SendTargets=All"`. När initieraren skickar detta så förväntar den sig att få tillbaka ett namn på en mottagare samt adress till mottagaren, detta är något som används vid en s.k. *Discovery session* som används för att se vilka mottagare som finns att logga in på. Vid denna kombinationen hämtas `"TargetName"` och `"TargetAddress"` ifrån `confKeyValuesMap` (`Go mapen` som håller datan som användaren har konfigurerat) och skriver dessa till `slicen data` i `structen`. När alla `"Key=Value"`-paren i den inkommande datan har hanterats och svar har skapats är det dags att pussla ihop hela svaret. Alla fälten i `structen` kopieras in på lämplig plats i `responseBuilder` och eventuell utfyllnad skrivs till för att svaret ska sluta på jämna 4 bytes. Sedan skrivs den använda delen av `responseBuilder` till anslutningen och svaret är färdigt. Nästa opkod läses in och en ny *request* ska hanteras.

5.1.7 Hantera SCSI-Kommando

När iSCSI-PDUn är ett SCSI-kommando så består alltid byte 32-47 av ett SCSI-CDB (*Command Descriptor Block*). Vissa av de andra fälten är specificerade som flaggor för att meddela något, t.ex. finns det en *Read*-flagga och en *Write*-flagga. Hela funktionen för att svara på ett SCSI-kommando är uppdelad i 3 delar genom 3 `if`-*satser*. Om bara *Read* är sann, om bara *Write* är sann eller om båda är osanna. I alla fallen så byggs ett svar på ett sätt som beskrivet ovan, men de kan variera mycket mer. När ingen av dessa är sanna så innebär det att en *SCSI Response* ska skickas till initieraren. Vanligtvis gäller det att skicka en *Test Unit Ready* som meddelar att LUNen i fråga finns.

När bara *Read* är sann så förväntar sig initieraren ett *SCSI DATA-In* som respons då detta innebär att data ska läsas någonstans. I den här PDUn kan en initierare skicka väldigt många olika SCSI-kommandon men en mottagare måste bara ha stöd för några av dem. Den som används flitigast och också den som är den mest uppenbara är *READ*. För att hantera detta SCSI-kommandot börjar funktionen med att spara adressen för var läsningen ska ske samt längden på hur många block av bytes som ska läsas. Sedan sker en uppslagning av LUN id i ovan nämna `Go map` som heter `confKeyValuesMap` där användarens inställningar är sparade. Här hittas filvägen till lagringsfilen och om filen existerar öppnas den och information så som dess storlek sparas i variabler. Eftersom en initierare kan begära att få läsa mer med en *READ* än vad som kan skickas som svar i en PDU måste flera svar med data kunna skickas vid behov. Om initieraren begär att få läsa mer än största möjliga datalast för ett svar läses så mycket som tillåtet och det skickas i ett svar. Därefter beräknas hur mycket mer data som ska läsas och samma jämförelse sker igen. När den totala datan som är kvar att skickas är mindre än största möjliga datalast läses denna och skickas som ett sista svar.

Även när bara *Write* är sann kan det vara många olika typer av SCSI-kommandon som ska behandlas. Den mest uppenbara och något avancerade är *WRITE*. Som respons förväntar sig initieraren antingen en *SCSI Response* eller *R2T (Ready To Transfer).* En mer detaljerad beskrivning av *WRITE*-implementationen följer i nästa stycke.

5.1.8 Att skriva data till fil

Den faktiska skrivningen till fil sker genom SCSI-kommandot *WRITE* samt iSCSI-kommandot *SCSI DATA-Out.* Precis som att en *READ* kan begära att få läsa mer än vad som kan skickas i en respons så kan en *WRITE* begära att få skriva mer än den data som kan skickas med i en *request*. Mängden data en initierare vill skriva sparas i ett fält som kallas *Expected Data Transfer Length* i SCSI-delen. Om detta fältet överstiger den maximala datalasten för en *request* ska mottagaren skriva den data som är medskickad för att sen skicka en *R2T Response* tillbaka.

Till varje *WRITE* följer svar i form av noll eller flera R2T beroende på hur mycket data som ska skrivas, när första R2T till en viss *WRITE* skickas som svar påbörjas en dataskrivningssekvens. Det finns en begränsning på hur mycket data som får skickas utan att initieraren behöver vänta på en ny R2T innan den skickar mer data tillhörande samma *WRITE*. Denna begränsning har initierare och mottagare kommit överens om vid upprättning av sessionen. När initieraren får en R2T respons kommer den att skicka tillbaka en eller flera *SCSI DATA-Out* PDUer med mer data.

Vid varje R2T medföljer ett *R2TSN (Ready To Transfer Sequence Number)*. Denna tillsammans med *Buffer Offset* och *Initiator Task Tag* gör det möjligt för mottagare och initierare att veta var och vilken data som ska skrivas. När första R2T skickas som svar på en *WRITE* skapas en `Go map` som sparar variabler och används för att hämta bland annat den totala mängden data som ska skrivas tillhörande det unika värdet för *Initiator Task Tag*. Om all data som kan tas emot för en R2T är mottagen och skriven så undersöker mottagaren om initieraren ville skriva mer data med samma *WRITE*-kommando. Visar det sig att så är fallet så skickas en ny R2T med ett R2TSN som är ett högre än det senaste i sekvensen. Är sekvensen däremot avslutad skickas ett *SCSI Response* som meddelar det, alltså är det bara när en ny R2T behövs eller när en sekvens är färdig som ett svar skickas mot *SCSI DATA-Out*.

5.2 Användande

Mottagarprogramvaran som den är idag är brukbar mot initieraren *open-iscsi*, detta är den enda initieraren som den är testad mot. Den har stöd för de viktigaste grundläggande funktionerna och lite till för en iSCSI-kommunikation.

Möjligheten att konfigurera mottagaren är begränsad men genom att redigera konfigurationsfilen som hittas i */etc/iscsit.conf* kan man lägga till en eller flera LUNs. Detta gör man genom att skapa en lämplig fil som ska agera lagringsenhet och lägger till en rad som länkar till den i stil med `“Lun 0 Path=/storage/lun0.bin”`. Det är viktigt här att högersidan av “=” länkar till filen i fråga och att det står *Lun N Path* till vänster där N är ett nummer för att identifiera vilken LUN det är.

För att ansluta till mottagaren kör man en *discovery-session* mot mottagaren från *open-iscsi* för att lokalisera vad mottagaren heter. Sedan kan man ansluta med en *normal-session* från *open-iscsi* utan autentisering och därefter går det bra att mounta med lämpligt verktyg. När den är mountad går den att hantera som en fysisk lagringsenhet som befinner sig på datorn som kör *open-iscsi.* För att logga ut kör man ett logout kommando från *open-iscsi* så kommer sessionen avslutas.

5.3 Alternativa lösningar

Ett sådant här program går naturligtvis att skriva på olika sätt, min lösning består av ett program i 4 delar som beskrivet i kap. 5.1. Del 1 är fristående från resten av programmet, den delen skapar bara en anslutning och sen körs del 2 till 4 som en separat process. Om inte del 1 hade varit en fristående del så hade inte flera initierare kunnat ansluta samtidigt. Detta är inget som stöds än men vid en full implementation är det något som behöver ses över.

Del 2 och 3 hänger ihop och kan inte köras parallellt. Eventuellt skulle de här två kunna köra parallellt med varandra för att höja prestandan och göra programmet snabbare, men då skulle också kontrollerna för hur vissa variabler anslutningsglobala variabler hanteras behöva skrivas om så att de kan hantera detta.

Del 2 och 3 hänger också delvis ihop med del 4 av samma anledning som de hänger ihop med varandra, men vissa funktioner i del 4 körs separat. Det beror på att de inte påverkar de anslutningsglobala variablerna på samma sätt och för att denna metod avsevärt förbättrar t.ex. skrivningshastigheten.

Man skulle kunna köra alla delar som en lång serie men då utesluter man möjligheten att flera initierare ska kunna ansluta samtidigt, dessutom hade programmet fått en mycket försämrad prestanda.

Ett alternativ som jag har övervägt är att lägga till ett femte steg som hanterar att bygga ihop ett svar och skriva den till anslutningen då det kan tyckas onödigt redundant att varje funktion i del 4 ska behöva göra detta på ett sätt som är ganska likt i alla funktionerna. I en sådan lösning skulle då del 4 bara hantera eventuella handlingar och sedan skicka vidare uppgiften att svara till del 5. Anledningen till att programmet inte fungerar så just nu är att jag inte från början var medveten om att flera delar skulle bli så pass lika. Detta handlar om en rätt stor refaktorering av koden som jag lämnat till en eventuell vidareutveckling av programmet.

6 Miljö och Etik

Molnlagring och virtualisering är något som ligger i tiden, en kritiker måste då fråga sig vad detta kan ha för effekter på samhället. Implementering av detta projektet kommer i längden utöka mängden användare som väljer att lagra data på virtualiserade servrar i molnet.

Ur ett etiskt perspektiv så är det viktigt att användarens integritet bevaras. När man lagrar sin data hos ett företag så är det viktigt att användaren kan lita på att företaget hanterar datan på ett sätt som inte försätter användaren i risk av att dess data går förlorad eller sprids. Därför ställs höga krav på den här typen av företag att både skydda de fysiska enheterna samt att förhindra att fel personer får tillgång till datan.

Att använda sig av virtualiserad lagring hos ett företag påverkar i längden konsumtionen av elektronik hos privatpersoner eller företag som väljer att använda sig av en sådan här lösning. T.ex. så skulle ett företag kunna hyra virtualiserade maskiner istället för att köpa kompletta datorer till sina anställda. Då minskar kraven för hur kraftfulla datorer de anställda kommer behöva eftersom att de endast kommer behöva kommunicera med en virtuell maskin i molnet. Att ett företag väljer att hyra servrar i molnet istället för att själva ha ett mindre datacenter innebär att energin som används till datacentret används på ett effektivare sätt, ty det är mer energieffektivt att ha ett stort datacenter än många små.

7 Resultat och Diskussion

Slutprodukten av det här arbetet är en iSCSI-mottagare som har stöd för grundläggande behov för en iSCSI-kommunikation. Programmet går att köras mot initieraren *open-iscsi* men endast utan autentisering. All skriven kod är välkommenterad och strukturerad för att vara lätt att förstå. Inga tester finns skrivna då all testning har skett kontinuerligt genom manuella tester med *open-iscsi.*

Slutprodukten av arbetet är begränsad på så vis att den endast är testad mot *open-iscsi* och den har inte stöd för mycket mer än grundläggande saker för en iSCSI-kommunikation men det som är implementerat fungerar. Projektets syfte var att undersöka om det är möjligt att genomföra en implementation av iSCSI-mottagare med hjälp av Go, vilket framgår av resultatet att det är.

För en vidareutveckling av programmet skulle nästa steg vara att lägga till autentisering för att kunna skapa en säker anslutning. Eftersom att inte så mycket information hämtas från konfigurationsfilen som potentiellt skulle kunna hämtas där så är detta också en given del som ligger tidigt i en eventuell vidareutveckling. Sen finns det vissa delar där programmet inte alltid tar hänsyn till alla eventuella flaggor i en PDU. Detta är flaggor som inte har behövts användas för att programmet ska fungera mot *open-iscsi* men för en full implementation skulle det vara nödvändigt att ta hänsyn till alla värden som finns där för att påverka något.

Eftersom det inte sedan tidigare finns någon för mig känd iSCSI-mottagare som är utvecklad i just Go så kan det vara värt att reflektera om varför det är så. Antagligen beror det inte på att Go skulle vara ett dåligt alternativ för att skriva ett sådant program i, utan snarare på att språket inte har funnits så länge och att alla iSCSI-mottagare som används idag påbörjades långt innan Go ens fanns som ett alternativ. IET som har använts för att jämföra med är skrivet i C. Det kanske inte är helt rättvist att jämföra mängden kod då IET är mycket mer komplett och har således mer funktioner, men jag är övertygad om att samma program i Go kommer kunna ha en effektivare grund som gör mer med mindre kod. Eftersom Ilait var intresserade av att se en lösning i just Go så har inte några vidare studier gjorts för att välja det optimala språket för uppgiften. Från egen erfarenhet tror jag att jag inte skulle få samma prestanda för den här typen av program med t.ex. Java. C är kanske ett utmanande alternativ och möjligheten till samma prestanda finns där, men det skulle vara en mer avancerad lösning för att uppnå den.

Arbetet har ganska väl följt den planering som var satt på förhand. Att lära sig hantera Go tog lite mindre tid än planerat men detta kompenseras med att det var väldigt svårt att hitta och ta in information om iSCSI. Vissa problem som går att läsa om i kap. 4.3 tog upp onödigt mycket tid men det rör sig om dagar inte veckor. En sak som påverkade planeringen var faktumet att SCSI också behövde hanteras av programmet, då det var tänkt att från början låta någon drivrutin i Linux hantera detta.

8 Referenser

1. TechTarget. 2008. [http://cdn.ttgtmedia.com/searchStorage/downloads/iSCSIeZineFINAL4.pdf] (Hämtad 2015-01-10)

2. The Internet Society. April 2004. *iSCSI Proposed Standard.* /http://tools.ietf.org/html/rfc3720] (Hämtad 2015-01-16)

3. John B. Lohmeyer. Juli 1990. *SCSI History.* [http://www.foo.be/docs-free/archive/historical-standard/ansi/x3t9/area01/history.txt] (Hämtad 2015-01-10)

4. Margaret Rouse. August 2014. *SCSI (Small Computer System Interface)* [http://searchstorage.techtarget.com/definition/SCSI] (Hämtad 2015-01-10)

5. INCITS Technical Committee T10. Januari 2015. *SCSI Operation Codes* [http://www.t10.org/lists/op-num.htm] (Hämtad 2015-01-10)

6. Google. FAQ. [http://golang.org/doc/faq#history] (Hämtad 2015-01-13)

7. Caleb Doxsey. 2012. *An Introduction to Programming in Go.* [http://www.golang-book.com/] E-Bok.

8. Alex Aizman & Dmitry Yusupov. Februari 2005. [http://www.open-iscsi.org/] (Hämtad 2015-01-16)

9. Ubuntu Community. [https://help.ubuntu.com/lts/serverguide/iscsi-initiator.html] (Hämtad 2015-01-16)

10. Wikipedia. Januari 2015. Fibre Channel. [http://en.wikipedia.org/wiki/Fibre_Channel] (Hämtad 2015-02-03)

11. Kent Olofsson. Så fungerar deduplicering. April 2007. [http://techworld.idg.se/2.2524/1.105215/guide-sa-fungerar-deduplicering] (Hämtad 2015-02-04)

12. Caleb Doxsey. 2012. *An Introduction to Programming in Go* [http://www.golang-book.com/10/index.htm] E-Bok

13. Andrew Gerrand. Februrari 2013. Go Maps in Action [https://blog.golang.org/go-maps-in-action] (Hämtad 2015-02-07)

14. Caleb Doxsey. 2012. *An Introduction to Programming in Go* [http://www.golang-book.com/6/index.htm] E-Bok

15. Wikipedia. Januari 2015. [http://sv.wikipedia.org/wiki/Scsi] (Hämtad 2015-02-10)

16. Wikipedia. Januari 2015.
[http://sv.wikipedia.org/wiki/American_National_Standards_Institute]
(Hämtad 2015-02-10)