

CHALMERS



Blockdrivrutiner i Linux

Implementering av en drivrutin utanför Linuxkärnan

HENRIK HUGO

ALEXANDER KURKIMÄKI

Examensarbete

Högskoleingenjörsprogrammet för datateknik

CHALMERS TEKNISKA HÖGSKOLA

Institutionen för data- och informationsteknik

Göteborg 2014

Innehållet i detta häfte är skyddat enligt Lagen om upphovsrätt, 1960:729, och får inte reproduceras eller spridas i någon form utan medgivande av författaren. Förbudet gäller hela verket såväl som delar av verket och inkluderar lagring i elektroniska och magnetiska media, visning på bildskärm samt bandupptagning.

© Henrik Hugo, Alexander Kurkimäki, Göteborg 2014

Förord

Detta examensarbete är utfört vid institutionen Data- och informationsteknik vid Chalmers tekniska högskola våren 2013. Momentet är den avslutande delen i utbildningen som utförs av studenter på dataingenjörsprogrammet och omfattar 15 högskolepoäng.

Dataingenjörsprogrammet vid Chalmers omfattar 180 HP och ger studenten både grundläggande teoretiska kunskaper samt praktiska tillämpningar inom data- och informationsteknik. Utbildningen innehåller kurser inom dator teknik, programutveckling (software engineering), inbyggda system, dynamiska system, algoritmer och datastrukturer men även matematik och elektronik, studenter har även möjlighet att fördjupa sig inom ämnen som datasäkerhet, nätverksteknik och visualiseringsteknik.

Rapporten riktar sig till studenter inom samma utbildningsområde samt andra med en teknisk bakgrund och läsaren bör ha kunskaper inom dator teknik och programmering.

Examensarbetet omfattar nätverks- och systemprogrammering i Linux med ett modernt programmeringsspråk samt efterforskningar om hur moderna drivrutiner utanför Linux-kärnan fungerar. Arbetet grundar sig i ett projektarbete, under läsperiod 3 våren 2013, vid Dataingenjörsprogrammet på Chalmers som undersökte möjligheten att implementera Ilait's blocklagringssystem IVBS för virtualiseringslösningen VMware vSphere.

Arbetet är utfört på Campus Lindholmen och på företaget Ilait AB i Kungälv. För att visa tacksamhet över att detta examensarbete blivit verklighet vill författarna tacka följande personer:

Mikael Grüner för handledning och rådgivning, programmerare på Ilait AB
Kim Lundgren, Chief Technical Officer på Ilait AB
Michael Abrahamsson, VD på Ilait AB
Christer Carlsson, handledare på Chalmers
Sakib Sisteck på Chalmers för vägledning och uppstart av examensarbetet

Abstract

Ilait, a hosting-company in Kungälv was searching for a driver in userspace in Linux with the ability to mount images from their own blockstorage-solution called IVBS with the use of a client-program. What is needed to make this work? Within this report the reader will find information on how with the help of Google's own programming-language called Go, NBD, a http-server that receives JSON and mounts images to an existing blockstorage-solution with the use of the driver is implemented. To make this work in Linux, knowledge about how the Linux-kernel is operating and having the driver run in userspace is needed.

Sammanfattning

Ett hostingföretag i Kungälv vid namn Ilait sökte en drivrutin i userspace i Linux som kunde hantera deras egna blocklagringssystem kallat IVBS för att montera avbilder på en dator via ett kontrollprogram. Hur går man tillväga för att få detta att fungera? Med hjälp utav Googles egna programmeringsspråk Go och NBD går det i denna rapport att läsa hur implementationen av en http-server som tar emot JSON och monterar avbilder till ett befintligt system med hjälp utav drivrutinen. För att kunna implementera detta i Linux krävs först förståelse hur kärnan i Linux hanteras samt hur det är möjligt att få drivrutinen att köras i userspace.

Innehållsförteckning

1 Inledning	2
1.1 Bakgrund	2
1.2 Problemformulering	2
1.3 Syfte	2
2 Metod	3
2.1 Arbetsmetod	3
2.2 Utvecklingsmiljö	3
2.3 Kravspecifikation	3
2.4 Kunskapsinhämtning	4
3 Teknisk bakgrund	5
3.1 Storage Area Network	5
3.2 Ilait Virtualized Block Storage	5
IVBS Paket	6
Skivavbilder	6
3.3 Linux	7
Linuxkärnan	7
Block-enheter	8
Kernelmoduler	8
3.4 Drivrutiner	9
3.5 Userspace I/O	10
3.6 Network Block Device	10
3.7 Programmeringsspråket Go	11
Paket	12
Kodformatering	12
Variabler och funktioner.....	12
goroutines	14
Kanaler.....	14
3.8 JavaScript Object Notation	14
4 Genomförande	16
4.1 UIO eller NBD	16
4.2 Val av programmeringsspråk	16
Tour of Go.....	16
4.3 Kommunikation med IVBS	16
4.4 Kontrollprogrammet netdclient	17
4.5 ivbsnetd	17
4.6 http-servern	17
4.7 Arkitektur	17
5 Resultat	19
5.1 Drivrutinen ivbsnetd	19
Kommunikation genom NBD	19
Kontrollprogrammet netdclient	20
5.2 Hastighetstest	21
Tester av Ilait	21
5.3 Buggar	22
Zombie-processer och låsning.....	22
6 Slutsats och diskussion	23
Referenser	24

Terminologi

Term	Beskrivning
avbild	Logisk representation av struktur och data från en lagringsenhet såsom en hårddisk. Eng. image.
branch	Användes inom Git för att beteckna en gren att arbeta med vid sidan av huvudkoden.
drivrutin	Kod som används utav operativsystem för att hantera hårdvara, även kallat drivare
hypervisor	Mjukvara som kan skapa och hantera virtuella maskiner.
IVBS	Ilait Virtualized Block Storage, Ilait's egna blocklagringslösning.
kernel	Kärnan i ett operativsystem som hanterar läs- och skrivoperationer mellan operativsystemet och hårdvaran.
NBD	Network Block Device, en TCP-lösning för att hantera virtuella enheter.
proxy	En mellanhand vid kommunikation i ett nätverk, exempelvis en server.
QEMU	En open-source hypervisor.
repository	Eng. repository. En plats där kod och andra filer lagras för bevaring.
SAN	Storage Area Network, nätverk utav lagringsenheter som är dedikerat för att hantera lagring och distribution av data.
UIO	Userspace I/O, en lösning för att utveckla drivrutiner till bl.a. PCI-enheter utanför kärnan.
userspace	Platsen i ett operativsystem där program körs på användarnivå

1 Inledning

1.1 Bakgrund

Molnlagring och "molnet" är tekniker som varit på stor uppgång de senaste åren och kan beskrivas som flera distribuerade resurser men som fortfarande agerar som en enda enhet. Dessa system kräver en avancerad mjukvarulösning som garanterar hastighet, redundans och därmed feltolerans.

Ilait, ett företag baserat i Kungälv som arbetar med "cloud computing" och hosting-tjänster, har själva utvecklat en egen molnlagrings-lösning kallad IVBS (Ilait Virtualized Block Storage). IVBS är byggd på en distribuerad modell och är tänkt att användas i produktionsmiljöer tillsammans med hypervisorn QEMU.

När IVBS endast kan utnyttjas tillsammans med hypervisorn QEMU, genom en egenutvecklad drivrutin, beslöt man att även utveckla en lösning för Linux.

Att utveckla drivrutiner i Linux kräver normalt att kod exekveras i Linuxkärnan, något som ger upphov till en rad problem. Ostabil kod i kärnan kan resultera i systemhaveri och möjligen dataförlust. Stabil kod som exekveras i en kernel-version kan dock vara ostabil i en annan eftersom Linuxkärnan konstant utvecklas och förändras.

Projekt som UIO (Userspace I/O) och NBD (Network Block Device) existerar för att tillåta att pseudo-drivrutiner körs utanför kärnan och därmed minimerar risk för systemhaveri och inkompatibilitet mellan kernel-versioner.

Detta tillåter också att drivrutinen använder sig av programmeringsspråk och bibliotek som inte finns tillgängliga i Linuxkärnan. Därför beslöts det att drivrutinen skall utvecklas i programmeringsspråket Google Go för att köras utanför kärnan.

1.2 Problemformulering

Hur implementerar man en drivrutin till IVBS i Linux som körs utanför kärnan och är programmerad i ett modernt programmeringsspråk.

1.3 Syfte

Syftet med detta arbete är att utveckla en drivrutin för Linux i userspace som kan representera skivavbilder i IVBS som block-enheter i filsystemet under */dev* i Linux.

För att kunna hantera drivrutinen ska ett kontrollprogram utvecklas som kan kommunicera med drivrutinen.

Slutligen är också syftet att få en inblick i de interna mekanismerna i Linux.

2 Metod

I detta kapitel redovisas och förklaras de metoder som använts under projektet

2.1 Arbetsmetod

Vid mjukvaruutveckling är det viktigt att ha en klar arbetsmetodik för att arbetet skall fortskrida så bra som möjligt. Därför valdes ett arbetssätt som är inspirerat av Scrum[8]. Då utvecklingsteamet endast bestod av två personer fanns inte motivation av att implementera alla aspekter av Scrum.

Kortare möten hölls dagligen för att stämma av och fördela uppgifter. Uppgifter skrevs upp på post-it lappar och tilldelades prioritet efter färg. På detta sätt kan utvecklarna ha klart för sig vad som krävs av programmet för att det ska vara funktionellt.

Då företaget bistod med kontorsutrymme och hårdvara bestämdes det att utvecklingen skulle ske på plats. Främst för att kunna hålla god kommunikation med företaget och för att få en god arbetsro.

Efter tidigare erfarenhet av versionshanteringssystem valdes Git för att hantera kodbasen och Github som, centralt repository. Github är ett gratis grafiskt verktyg för Git och är tillgängligt genom webbläsaren.

Med verktyg som Git kan flera personer arbeta på olika grenar eller "brancher" utan att störa varandra och skapa kaos. Även hantering av vilka funktioner som skall implementeras hanterades delvis via så kallade "issues" i GitHub. Dessa "issues" kan sedan länkas till en commit och kan på så sätt ge en bra överblick över när och var en ny funktionalitet implementerats eller en bugg felsökts.

2.2 Utvecklingsmiljö

Under projektet utnyttjades ingen specifik utvecklingsmiljö utan det beslutades att använda Sublime Text som texteditor för koden. Kompilering utfördes i en terminal.

För att hantera dokument användes Google Drive. Då inga dokument hade någon sekretess lämpar sig det väl för att kunna, såsom GitHub, vara oberoende av vilken dator och operativsystem som används. För att få tillgång till Google Drive krävs endast en modern webbläsare.

Vid val av programmeringsspråk beaktades Ilaitis preferens och expertis. Handledaren på företaget var kunnig i Go och rekommenderade att arbetet utfördes i detta. Go ansågs vara intressant men även nytt och blev valet av programmeringsspråk.

2.3 Kravspecifikation

Drivrutinen skall:

- ligga i userspace
- ha stöd för failover, d.v.s. kunna återansluta till en annan IVBS proxy vid fel
- kunna montera en avbild så att denna blir tillgänglig som block-enhet
- endast göra avbilder tillgängliga vid begäran av kontrollprogrammet
- använda sig av asynkron dataöverföring

2.4 Kunskapsinhämtning

För att kunna arbeta med Linux-kärnan krävs en grundläggande förståelse i hur Linux är uppbyggt samt kunskapen om moduler och drivrutiner. Under projektet utnyttjas en modul i Linux som heter NBD. För denna krävs ingen djupare förståelse av Linuxkärnan då utveckling framförallt kommer att ske i userspace.

Projektet innefattande primärt Gos officiella dokumentation [16] men även källkod till projekt som implementerar NBD i sina program. Några exempel på sådana projekt som var relevanta för denna rapport är BUSE [18] och den officiella NBD servern och klienten [17]. För att få en inblick i hur drivrutiner fungerar i Linux konsulterades även boken *Linux Device Drivers* [19]. Samtliga resurser fanns fritt tillgängliga på nätet.

För att kunna programmera effektivt i Go krävs att man skapar sig en förståelse hur Go är uppbyggt samt hur syntaxen är skriven. Processen att lära sig hur Go fungerar påskyndades genom att använda hemsidan *Tour of Go* som är Googles egna guide för att snabbt komma igång med Go. [2].

3 Teknisk bakgrund

I detta kapitel beskrivs vilka verktyg, programmeringsspråk, program, utvecklingsmiljöer samt bibliotek som använts under arbetet.

3.1 Storage Area Network

Storage Area Network (SAN) är ett nätverk bestående av servrar och lagringsenheter med huvuduppgift att distribuera och lagra stora mängder data. Med ett SAN får man tillgång till datalagring på blocknivå där all lagring är sammanfogad. Användningen av SAN är främst för att göra lagringsenheter som exempelvis diskarray eller bandmaskin tillgängliga till servrar där enheterna är tillsynes lokala hos operativsystemet.

Hastigheten på SAN är likvärdig med att hårddisken i själva verket skulle vara kopplad direkt till systemet, även om kommunikation faktiskt sker över ett nätverk. Kommunikationen mellan servrar och hårddiskar sker oftast via fiberkablar och speciella protokoll som iSCSI och Fibre Channel [24].

iSCSI är ett ålderdomligt protokoll och är en adaptation av SCSI-kommandon över TCP/IP. Ilait anser att det inte klarar av att prestera enligt den standard Ilait vill uppnå, vilket ledde till att företaget letade efter andra lösningar.

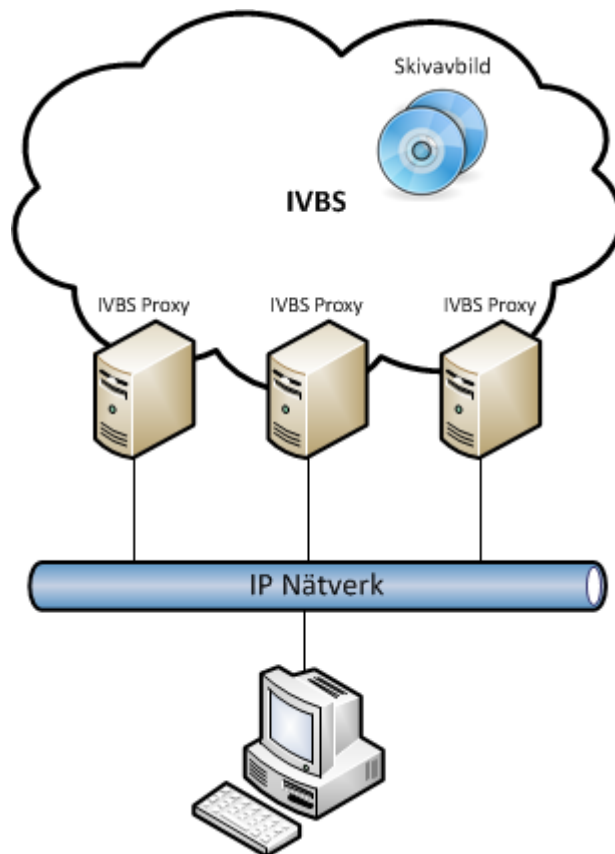
3.2 Ilait Virtualized Block Storage

Målsättningen med IVBS var att utveckla en egen blocklagringslösning med bättre funktionalitet, säkerhet och hastighet än det som fanns tillgänglig på marknaden. IVBS utvecklades av vår handledare på företaget år 2011 under hans examensarbete och har haft fortsatt utveckling sedan dess [23].

IVBS (se figur 3.1) kan ses som en moln-tjänst där man via en av flertalet proxies skickar kommandon till IVBS. Kommandon kan vara läs- och skrivoperationer som ger åtkomst till data på skivavbilder i IVBS. Det som skiljer IVBS från andra SAN-lösningar är att IVBS är en mjukvarulösning och kräver därför ingen specifik hårdvara. Detta gör att IVBS är en mer flexibel lösning jämfört med många kommersiella SAN-lösningar och är en av IVBS starka sidor.

När en läs- eller skrivoperation skickas till IVBS måste denna passera genom en proxy. Om denna proxy skulle utsättas för fel måste något göras för att upprätthålla anslutningen mot IVBS. Detta löses genom att IVBS, med jämna mellanrum, skickar ut en lista på proxies. Listan innehåller adresser till de proxies som är tillgängliga. Med denna lista går det snabbt och enkelt att ansluta direkt till en ny proxy utan större avbrott. Denna typ av feltolerans är något som är en annan av IVBS starka sidor.

I dagsläget används IVBS för skivavbilder på open-source hypervisorerna QEMU men i framtiden är det planerat att även utöka stödet till VMwares produkter.



Figur 3.1 Överblick över IVBS.

IVBS Paket

All kommunikation till IVBS sker via paket som skickas över ett TCP/IP nätverk. Paketerna talar om för IVBS vilka kommando klienten vill skall utföras. T.ex. att förbereda en avbild för läs- och skrivoperationer. Av dessa paket finner man några extra viktiga:

- **header**, en sektion med data som skickas med alla paket och ligger före datan. Beskriver vilken typ av paket som kommer och hur mycket data som följer.
- **greeting**, det första som skickas från IVBS när en TCP-anslutning etablerats och innehåller information om proxyn som används för anslutningen.
- **login**, detta paket skickas när klienten skall skapa en auktoriserad anslutning till IVBS. Det innehåller användarnamn och lösenord så att läs- och skrivoperationer kan utföras.
- **läs- och skriv**, dessa paket skickas till IVBS med information om vilken skivavbild som skall läsas ifrån eller skrivas till.
- **list-proxies**, ett paket som innehåller information om vilka proxies som finns tillgängliga för kommunikation till IVBS. Skickas med jämna mellanrum ut av IVBS till klienter.
- **mount**, används av klienter för att förbereda en avbild för läsning och skrivning.

Skivavbilder

En skivavbild är en datafil som innehåller t.ex. operativsystem, konfigurationer eller program vilka virtuella datorer använder för att kunna köras. Denna datafil är den information man finner hos de CD- och DVD-skivor som används vid installation av dylik programvara.

3.3 Linux

Detta avsnitt presenterar relevanta begrepp inom Linux och de delarna som rör detta projekt

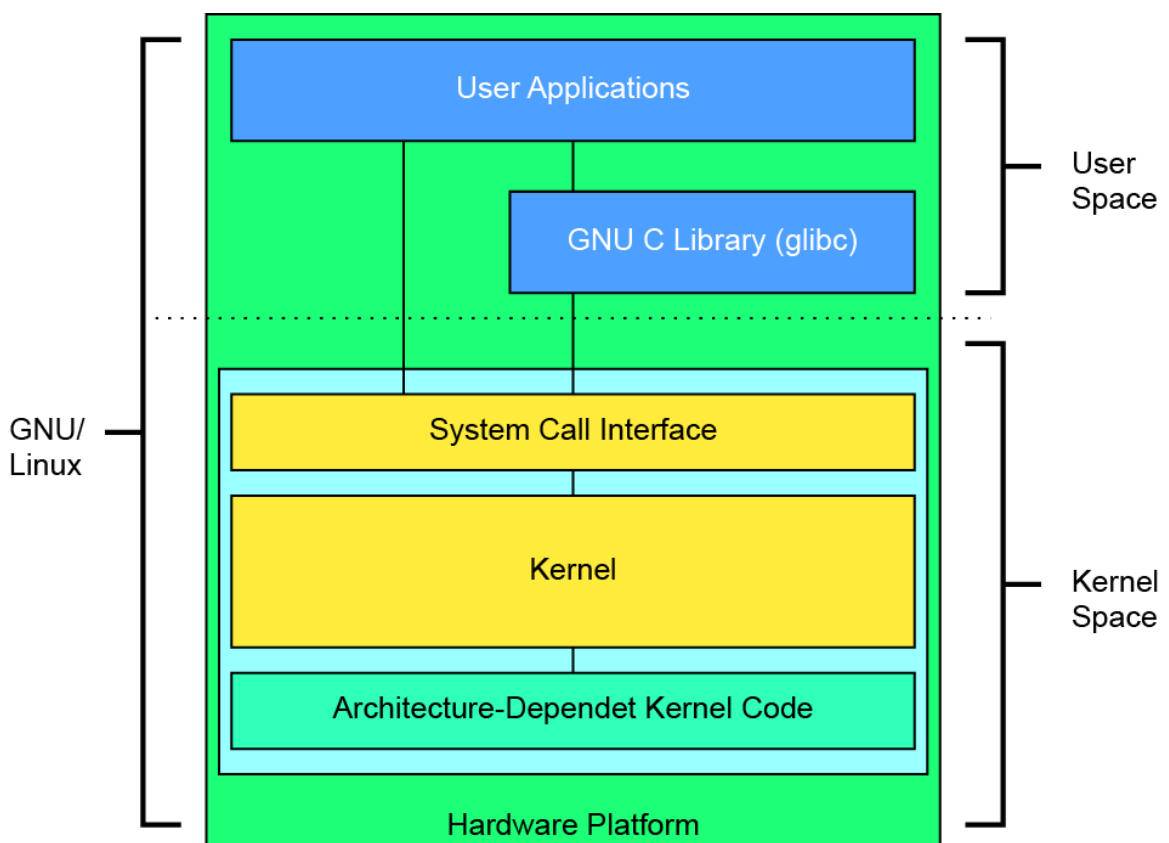
Linux är ett operativsystem som utvecklades från en operativsystemskärna skapad av finlandssvensken Linus Torvalds. Linux förknippas oftast med begrepp som öppen källkod och fri mjukvara (engelska *open-source* och *free software*). Torvalds ville ha ett fritt och öppet operativsystem där användaren skulle ha möjligheten att utveckla till kärnan och ändra om precis hur som helst. Linux liknar Unix och har blivit populärt bland utvecklare sedan det först släpptes.

Det finns många olika distributioner utav Linux som t.ex. Ubuntu, Xubuntu, Debian och Red Hat. En distribution är ett färdigt paket med kärna, valda bibliotek och mjukvara som distribueras ut till användarna, klar att installeras och köras.

Bibliotek och mjukvara som distribueras i paketen härrör ofta från open-source projektet GNU (GNU Not Unix). Därav det mer korrekta namnet GNU/Linux då Linux-kärnan är nästintill värdelös utan användarprogram.

Linuxkärnan

I alla operativsystem finns en kärna (kernel). Kärnan har till uppgift att hantera alla operationer mellan operativsystemet eller applikationer och hårdvaran. Namnet Linux syftar inte på operativsystemet utan själva kärnan.



Figur 3.2 Överblick över GNU/Linux.

Systemminnet i Linux kan delas upp i två distinkta regioner: kernel- och userspace (som visat i figur 3.2). Kernelspace är där kärnan och dess komponenter körs och userspace refererar till det minnesutrymme där alla användarprogram körs. Processer som körs i userspace hanteras av kärnan. Kärnan tilldelar resurser till processerna och

ser till att de inte stör varandra genom att isolera processernas minnesutrymme. Kernelpspace kan endast komma åt av användarprocesser genom systemanrop, t.ex. läs- och skrivoperationer (I/O) [5][10].

För att ett operativsystem skall kunna kommunicera med hårdvaruenheter krävs specifik kod. Denna kod kallar man för drivrutin, sådan kod används dagligen till möss, tangentbord, grafikkort, USB-minnen samt mycket mer.

Block-enheter

Enheter som CD-läsare och hårddiskar i Linux representeras som speciella filer i filsystemet och monteras under */dev* under ett speciellt filsystem, skiljt från resten av systemet. Exempelvis representeras alla partitioner på anslutna hårddiskar under */dev*.

Generellt sett finns det två typer av enheter, *character devices* och *block devices*. Med kommandot *ls* som listar filer och mappar i systemet kan man få fram information om en enhet under */dev* och med några extra parametrar även vilken typ. I figur 3.3 syns kommandot *ls* med parametern *-l* som listar filer och mappar under */dev*.

```
# ls -l /dev
brw-rw---- 1 root disk      8,  1 May 29 16:36 sda1
brw-rw---- 1 root disk      8,  2 May 29 16:36 sda2
brw-rw---- 1 root disk      8,  5 May 29 16:36 sda5
crw----- 1 root root       4,  0 May 29 16:36 tty0
crw----- 1 root tty        4,  1 May 30 13:15 tty1
```

Figur 3.3. Utdrag av utmatningen från *ls -l /dev*.

Första bokstaven på varje rad ovan visar om det är en block-enhet eller en tecken-enhet. Första raden visar då att *sda1* är en block-enhet. Det står också att filen (eller enheten) ägs av användare *root* och gruppen *disk*. Sist kommer namnet på enheten som det representeras i filsystemet och före det ett skapelsesdatum. Innan datumet kommer enheternas *major* och *minor numbers*. Dessa identifierar enheten med vilken drivrutin som den tillhör. För att hålla ordning på enheter finns en konvention som beskriver vilka *major* och *minor numbers* samt namn olika enheter bör tilldelas [13].

sda1 är här första och *sda2* den andra partitionen på första SATA-disken. Om det finns ytterligare en SATA-disk skulle dessa partitioner börja med *sdb*.

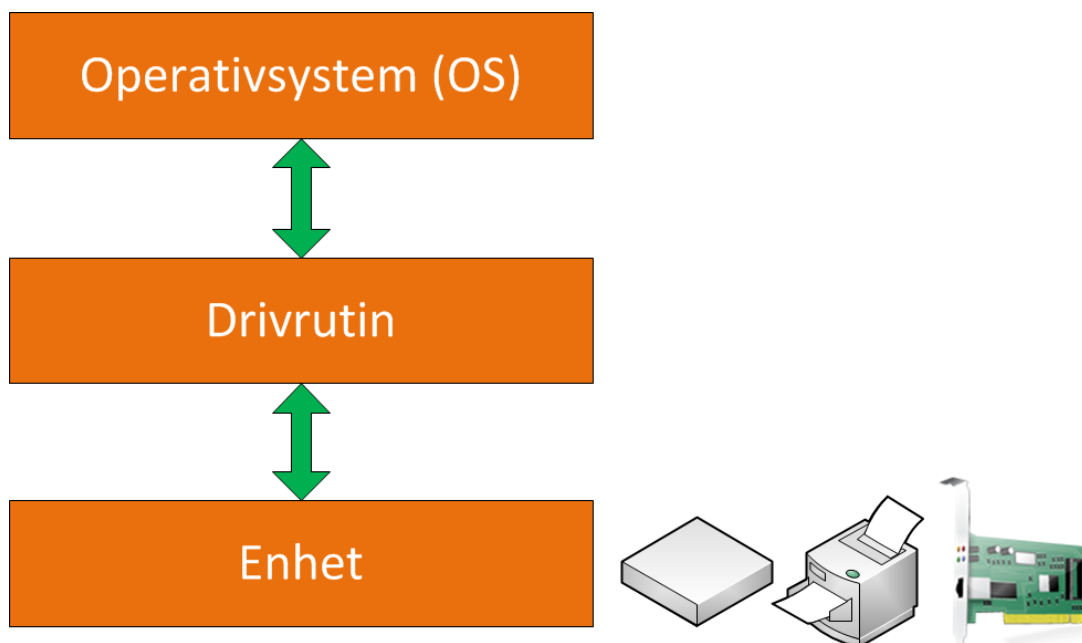
Drivrutinen i detta projekt kommer att använda sig av block-enheter i Linux. Dessa enheter kan vara hårddiskar eller DVD-läsare. Alla block i dessa enheter har en förutbestämd storlek och för att en enhet skall räknas som en block-enhet finns även kravet att kunna söka efter specifika block på enheten [9].

Kernelmoduler

En kernelmodul är ett program som laddas in i Linux-kärnan. Dessa moduler kan vara exempelvis drivrutiner vilket gör att kärnan kan kommunicera med hårdvara som är ansluten till datorn. Att utveckla en kernelmodul innebär risker. Datorn kan krascha och filsystemet kan raderas helt av att exempelvis en pekare har hamnat på fel adress. Det positiva med moduler är att man slipper kompilera om kärnan och starta om varje gång man vill lägga till funktionalitet [3].

3.4 Drivrutiner

En drivrutin är ett program som tar hand om systemanrop för en viss typ av hårdvaruenhet som är ansluten till datorn t.ex. skrivare, grafikkort och DVD-läsare. Drivrutinen ligger i ett lager mellan operativsystemet och hårdvaran, se figur 3.4. Systemanrop från användarprogram aktiverar kod i drivrutinen som kommunicerar direkt med enheten i fråga, ofta genom systembussen. Att program inte behöver känna till hårdvaran direkt underlättar för utvecklare, då de kan använda mer generella systemanrop för I/O till enheter.



Figur 3.4 Kommunikation med hårdvaruenheter genom drivrutiner.

Drivrutiner körs oftast i kärnan och koden beror på hur kärnan är uppbyggd. Det betyder att för kärnor som Linuxkärnan, vilka hela tiden förändras, måste drivrutiner ändras eller skrivas om för att fortsätta fungera med nyare versioner. Att sedan behöva underhålla olika versioner av en drivrutin för olika versioner av kärnor kan vara mycket tidskrävande. En lösning på detta problem är att istället köra drivrutiner utanför kärnan i det så kallade *userspace*.

Linuxkärnan har inte förändrats i någon större utsträckning på senare år. Majoriteten av drivrutiner har därför de facto inte varit i behov av större modifikationer.

Drivrutiner i *userspace* har flera fördelar:

- Använda bibliotek och verktyg som utvecklaren är van vid.
- Slipper utveckla och underhålla kernelmoduler.
- En bugg ger inte upphov till krasch i kärnan

Men även nackdelar:

- Inte direkt åtkomst till kärnans minne.
- Avbrott kommer inte att fungera.
- I vissa fall prestandaförlust

För att skriva en drivrutin i userspace som på något sätt kan kommunicera med hårdvaruenheter och operativsystemets mer integrerade delar måste man fortfarande ha en koppling till kärnan. Userspace I/O som beskrivs i kapitel 3.5 är en modul som försöker lösa just kopplingen till kärnan.

Det finns också ett flertal andra moduler tillgängliga som ger program i userspace ett interface till hårdvara eller system som inte annars är tillgängligt utanför kärnan. NBD som beskrivs i kapitel 3.6 är just en sådan lösning. Till skillnad från Userspace I/O, som är en generell lösning, implementerar NBD specifikt möjligheten att exponera virtuella block-enheter för operativsystemet och dirigera om I/O till dessa.

3.5 Userspace I/O

Userspace I/O (UIO) ger möjligheten att kunna ansluta egenutvecklad hårdvara och skriva drivrutiner utan att behöva lära sig och utveckla egna Linuxmoduler, eftersom UIO ligger som en modul i kärnan och skapar ett interface mot userspace. Varje UIO-enhet finns tillgänglig under `/dev` med namnet `uioX`, där X representerar ett nummer från 0 och uppåt beroende på antalet enheter.

Vid utveckling av en drivrutin till hårdvara som använder UIO kommer koden att exekveras i userspace. Därför resulterar det inte i att buggar i drivrutinen kan krascha kärnan. Utvecklaren behöver endast ändra i den kod som körs i userspace när en ny kernelversion släpps. Det skall dock noteras att alla nya kernelversioner ej behöver påverka drivrutinen funktionalitet. Det som gör detta möjligt är att UIO vidarebefordrar alla operationer från drivrutinen till kärnan, när UIO är inladdad i kärnan.

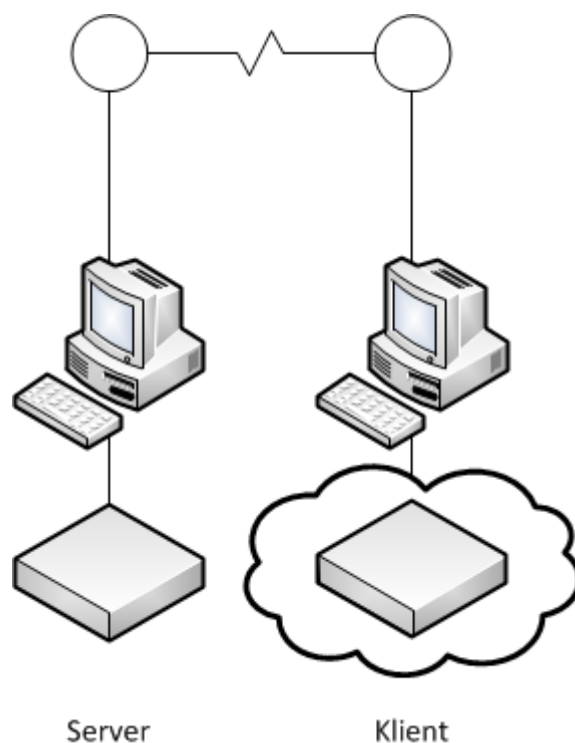
3.6 Network Block Device

Network Block Device (NBD) är en kernel-modul i Linux som tillsammans med NBD-protokollet är en lösning för att använda enheter och skivavbilder på fjärran servrar som sina egna block-enheter (se figur 3.6). NBD representerar virtuella block-enheter vid namn `NBDx`, där x är ett nummer. Varje gång klientdatorn vill läsa från t.ex. `/dev/nbd0`, som NBD har satt upp, kommer NBD att skicka en förfrågan till en server över TCP/IP som svarar med den efterfrågade datan [22].

Till skillnad från NFS (Network File System) går det att använda vilket filsystem som helst, då NBD endast hanterar block av data. För att NBD skall fungera måste NBD-modulen vara inladdad i kärnan, vilket enkelt kan göras via kommandot `modprobe`. `modprobe` kräver root-rättigheter, därav `sudo` i figur 3.5, om användaren inte är root.

```
# sudo modprobe nbd
/dev
  /nbd0
  /nbd1
  /nbd2
  /nbd3
  /nbd4
```

Figur 3.5 Mappstrukturen för `nbd`-enheter under `/dev` i Linux.



Figur 3.6 Illustration över NBD.

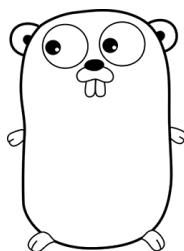
NBD ingår i Linux källkoden sedan version 2.1.101 [22] och finns därför redan med i de flesta distributioner av Linux. Det kan vara värt att nämna att endast klientsidan av NBD kräver kernel-modulen, detta innebär att implementationer av servern har skett på andra operativsystem, såsom Windows, BSD och Mac OS X.

Linux är inte heller ensam om idén och liknande implementationer finns som kanske t.o.m. är bättre än NBD. Men med NBD i Linux källkodsträd har projektet en stor fördel då koden alltid kommer att vara kompatibel med Linuxkärnan. En vidareutveckling av NBD är Enhanced Network Block Device (ENBD). ENBD har som mål att förbättra Linuxkärnans egen NBD-modul med stöd för Redundant Array of Independent Disks (RAID) över nätverket [21].

3.7 Programmeringsspråket Go

Detta delkapitel presenterar Go i korta drag och ger en insyn i varför Go existerar med redan så många andra populära språk.

Google har utvecklat och släppt sitt egna programmeringsspråk kallat Go där Go har målet att programmering skall vara enkelt och effektivt på samma gång. Grundarna Robert Griesemer, Rob Pike och Ken Thompson konstaterade att datorer blivit snabbare med åren men att mjukvaran inte fått samma utveckling vilket ledde till att Go skapades [7]. Vill läsaren sätta in sig mer i Go utanför denna rapport rekommenderas ett besök till programmeringsspråkets hemsida [2] med bland annat en interaktiv tur av språket [16].



Figur 3.7. Gos maskot, jordekorren Gordon.

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, 世界")
}
```

Figur 3.8 Det klassiska Hello World-program skrivet i Go med stöd för Unicode.

Paket

I många programmeringsspråk använder man sig utav paket för att hämta specifika funktioner som kan utföra diverse uppgifter. Detta för att inte programmeringsspråket skall bli tungt och belastat med funktioner som inte används. Genom funktionen `import` kan man i Go importera paket som innehåller diverse funktioner. I figur 3.8 importeras paketet `fmt` (förkortning av `format`) som innehåller funktioner som gör det möjligt att få utskrifter genom funktionen `Println`. När en funktion skall anropas skriver man först paketets namn följt av en punkt och slutligen namnet på funktionen. Funktioner som skall kunna användas av andra paket skrivs med stor bokstav i början. Kompilatorn känner automatiskt av detta och exporterar rätt symboler vid kompilering.

Många paket finns med i grunddistributionen av Go så som stöd för kryptering med paketet `crypto` [25]. Det finns även fler paket att hämta från externa källor som ger stöd för bland annat grafiska fönster [26].

Kodformatering

Vid skrivning av kod finns många riktlinjer som kan vara viktiga att följa. Att ha ordentliga kommentarer ovanför metoder kan vara viktigt. Detta om man vill att någon annan person efter skall få förståelse för koden som skrivits. Indentering kan även göra att koden blir mer lättläst. I många språk finns det ingen satt standard för hur koden skall se ut och många argumenterar för det som de tycker ser enklast ut.

I Go existerar inte detta problem då skaparna av Go har definierat en standard för hur koden skall vara formaterad och denna standard är den som anses vara korrekt. Tycker man att det är jobbigt att indentera koden kan man använda sig av det inbyggda verktyget `go fmt` som formaterar koden efter Gos standard. `go fmt` i detta sammanhang skall inte förvirras med paketet `fmt` för formaterad I/O.

Variabler och funktioner

Något varje programmeringsspråk besitter som gör att de skiljer sig från mängden är syntaxen för språket. I Go är inte detta något undantag då projektet startats från första början för att syntaxen i de stora språken, som exempelvis Java, hade krånglig syntax enligt skaparna [12]. Detta var bara en av anledningarna men med en lätt och enkel syntax kommer även smarta designval och dessa gör att Go skiljer sig från mängden.

```
i := 5
var j int = 12
```

Figur 3.9

I figur 3.9 deklarerar och initieras först en *integer* med värdet 5 med hjälp av operatoren `:=` utan att namna nyckelordet *int*, Go kan själv vid kompilering evaluera tilldelning av *i* till typen *int*. På nästa rad deklarerar variabeln *j* till en *int* med värdet 12, båda skrivsätten är helt korrekta dock är det andra sättet det säkraste. Uppmärksamma

läsare märker att typen skrivs efter variabelnamnet i Go. Det går även bra att deklarerar och initiera flera variabler på samma rad med operatoren := och då även med olika typer som demonstrerat i figur 3.10.

```
var x, y, z int           = 1, 2, 3
    c, python, java      := true, false, "no!"
```

Figur 3.10 Deklaration och initiering av flera variabler av olika typer på samma rad.

Att kunna deklarerar och initiera variabler på detta sätt finns till för att det är så pass vanligt att både deklarerar och initiera en variabel på samma rad.

```
func add(x int, y int) (int) {
    return x + y
}

func swap(x, y string) (string, string) {
    return y, x
}
```

Figur 3.11

Funktionsdeklaration i Go är i vissa drag liknande andra populära språk som Java och C, men skiljer sig också på andra punkter. I Go skrivs typen efter variabler och funktioner, returvärdet skrivs efter parameterlistan till en funktion, där man kan ha flera returvärden med olika typer. Typen skrivs efter variabler och funktioner som i figur 3.11. På samma sätt som med variabeldeklarationer kan man även korta ner deklaration av in-parametrar om två eller fler efterföljande in-parametrar delar typ genom att bara skriva typen en gång som visas i funktionen *swap* i figur 3.11 ovan.

En funktion kan också returnera flera resultat. Också visat i *swap* ovan, något som är lite ovanligare men effektivt. Detta tillåter funktioner att t.ex. returnera både resultat och eventuella fel på ett smidigt sätt. I figur 3.12 kombineras ett funktionsanrop med flera returvärden tillsammans med Gos kompakta variabeldeklaration.

```
x := "Hello"
y := "World"
fmt.Println(x,y)
x, y = swap(x,y)
fmt.Println(x,y)
-----
Hello World
World Hello
```

Figur 3.12

Fel i Go hanteras av den inbyggda typen *err*. Funktioner returnerar ofta en variabel av typen *err* som är nil om inget gick fel. *err* innehåller annars en textsträng som förslagsvis skrivs ut till terminalen eller en log. nil är Go's version av null.

goroutines

goroutines är Gos val av mekanism för att exekvera kod parallellt. Valet av namn kommer från att skaparna ansåg att namn som trådar och processer inte var lämpliga termer. Vad som gör goroutines speciella jämfört med hur andra programmeringsspråk hanterar trådar är att goroutines tar upp mycket lite minne. goroutines kan även köras i samma adressutrymme tillsammans med andra funktioner. När en goroutine blockeras för exempelvis väntan på en läs- eller skrivoperation körs de andra funktionerna vid sidan av, detta då goroutines är fördelade på flertal av operativsystemets trådar [11].

Med flertalet trådar som körs individuellt är det möjligt att få goroutines att kommunicera med varandra och dela information samt signalera att de är klara, genom att komplettera med kanaler.

Kanaler

Kanaler i Go är ett sätt för goroutines att kunna kommunicera med varandra och dela information. Det finns möjlighet att skapa två olika sorters kanaler, en obuffrad kanal och en buffrad kanal. En obuffrad kanal är som ett rör som går att skicka och läsa data ifrån. Vilken typ av data som får skickas i kanalen bestäms när kanalen initieras.

En buffrad kanal kan ses som en kö där man stoppar in värden och antalet värden som kan buffras måste specificeras. När man hämtar information från en buffrad kanal tas det första värdet som stoppats in ut, likt en vanlig kö. I programmeringstermer kallas en sådan kö för FIFO-kö (First In First Out). För att använda kanaler måste man först skapa en kanal, detta görs genom funktionen *make*.

```
ch1 := make(chan int)           // Initiera en obuffrad kanal för int
ch2 := make(chan int, 10)      // Initiera en buffrad kanal för int
                                // med 10 platser

ch1 <- 1 // Skicka värdet 1 på kanalen ch1
v := <- ch2 // Initiera variabeln v och ta emot data från kanal ch2
```

Figur 3.13 Syntax för kanaler.

3.8 JavaScript Object Notation

JavaScript Object Notation (JSON) är ett lättviktigt dataformat som användes mycket inom programmering för exempelvis C, HTML och Java. Då JSON är en enda sträng med tecken går det mycket enkelt för program att tolka strängen och utvinna informationen. JSON är inte bundet till ett visst programmeringsspråk vilket gör att många språk har stöd för JSON. XML har en mer komplex syntax då den bygger på HTML som alltid har inledande och avslutande taggar. JSON anses vara mer läsvänlig för människor jämfört med XML [14].

```
{"Mounted": [{"NbdDevice": "/dev/nbd0", "ImageName": "exjobb-test"}], "User": [{"Username": "exjobb", "Password": "chalmers"}]}
```

```
{  
  "Mounted": [  
    {  
      "NbdDevice": "/dev/nbd0",  
      "ImageName": "exjobb-test"  
    }  
  ],  
  "User": [  
    {  
      "Username": "exjobb",  
      "Password": "chalmers"  
    }  
  ]  
}
```

Figur 3.14 Syntaxen på JSON-fil, ovan är strängen i ett stycke, nedan en lättläst form.

JSON har delar av sitt ursprung från JavaScript med uppgift att kunna hantera enkla datastrukturer och associativa fält. De strukturer som JSON klarar av är nummer, strängar, booleaner, arrayer, objekt och null.

4 Genomförande

Detta kapitel beskriver genomförandet av arbetet, vilka val som gjorts och hur problem som uppstod löstes.

4.1 UIO eller NBD

Tidigt i arbetet gav Ilait på förslag att utnyttja UIO (se kap 3.5) för att kunna kommunicera med Linux-kärnan. Med UIO som utgångspunkt sattes målet till att skapa en uppfattning om hur UIO fungerade och hur lösningen skulle se ut. Det framkom då att UIO främst var menad för tecken-enheter (character devices). Mycket fokus låg på avbrott och minnesmappning för att anslutna hårdvaruenheter, något som inte var aktuellt för den tänkta lösningen.

Detta ledde till att valet föll på NBD (se kap 3.6). NBD är utvecklat för exakt det syfte som projektet behövde genom att kunna exponera virtuell hårdvara för systemet och omdirigera kommunikation med enheten till userspace.

4.2 Val av programmeringsspråk

Till val av programmeringsspråk fanns det två alternativ, C eller Go. C var ett bekant språk sedan tidigare och eftersom Linuxkärnan är programmerad i C hade språket en stark fördel. Med intresse för C som val upptäcktes NBD som redan låg i kärnan och endast krävde nätverkskommunikation från drivrutinen. Detta ledde till att Go blev ett mer intressant val. Då IVBS är utvecklat i Go och att Go var en rekommendation från företaget blev valet att utveckla drivrutinen i Go.

Tour of Go

Då Go har som mål att vara enkelt att skriva ser syntaxen lite annorlunda ut. För att nya användare skall lära sig Go har Google skapat en självlärnings-guide kallat Tour of Go som finns gratis att använda på deras sida där man går igenom struktur och funktionalitet. Av de 71 kapitel som finns tillgängliga i Tour of Go finns det ett antal övningar som är till för användaren att få utnyttja de kunskaper man lärt sig och för att skapa ett bra tänk för programmering. Varje kapitel består av text som förklarar funktionen som tas upp i kapitlet samt en textruta med kod som är möjlig att kompilera och köra, detta då Google uppmanar användaren att ändra i koden och på så vis skapa större förståelse för hur Go fungerar.

4.3 Kommunikation med IVBS

För att kommunicera med IVBS krävs funktionalitet för att hantera de paket som IVBS skickar. Greeting- och Login-paketet måste skickas och tas emot innan man får en anslutning till IVBS. Detta då Greeting-paketet innehåller ett sessions-id som IVBS använder sig utav för att identifiera paket från olika sessioner. Hantering av Login-paket krävs sedan av klienten för att kunna behålla anslutningen och inkluderar ett användarnamn och lösenord. När anslutningen är etablerad till IVBS kommer läs- och skrivoperationerna att accepteras, om ingen korrekt anslutning är skapad så kastar IVBS de paket som kommer från en då okänd källa.

All kommunikation mot IBVS genomfördes mot en testmiljö av IVBS för att inte riskera att känslig data skulle gå förlorad.

4.4 Kontrollprogrammet netdclient

För att kunna hantera drivrutinen med att ladda NBD enheter med specifika skivavbilder i IVBS krävs ett kontrollprogram. Med detta kontrollprogram ska en användare kunna starta upp och manuellt skicka kommandon till drivrutinen. Denna ska skicka information till en http-server i drivrutinen som tar emot dessa kommandon och utföra valda operationer. Kontrollprogrammet skall även kunna få ut information om statusen på drivrutinen och lista monterade avbilder.

4.5 ivbsnetd

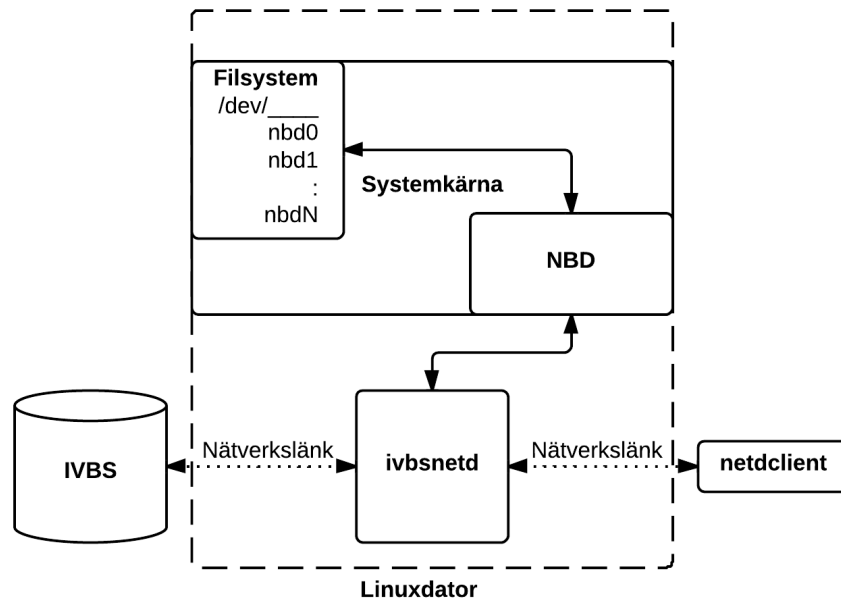
Själva drivrutinen kommer att bli den komponent som medlar mellan IVBS och NBD. ivbsnetd kommer som tidigare beskrivits inte fungera som en traditionell drivrutin utan kommer att köras som en tjänst eller daemon i userspace. Härifrån ska drivrutinen upprätta kommunikation med IVBS via tcp/ip och NBD via systemanrop. Uppgiften kommer sedan bli att medla förfrågningar från NBD till IVBS på ett efterfrågat format eftersom systemen implementerar olika protokoll.

4.6 http-servern

För att kunna hantera kommandon som skickas från kontrollprogrammet krävs någon form av server och valet blev att utnyttja http. Denna server har till uppgift att ta emot information via formatet JSON och efter önskade kommandon från kontrollprogrammet, utföra sina uppgifter med hjälp av drivrutinen. Denna http-server startas upp av drivrutinen och lyssnar på önskad port tills dess att den stängs av. Funktionalitet att kunna visa vilka NBD enheter som är monterade via en webbläsare valdes också till att implementeras.

4.7 Arkitektur

Efter att ha planerat vilka komponenter som krävs för att efterfrågad funktionalitet skall uppnås, behövs en arkitektur som kopplar samman allt. I figur 4.1 visas att NBD redan är inladdad i kärnan och har då redan en koppling till filsystemet, därför behövs en koppling mellan drivaren och NBD. Det krävs också en nätverkslänk mellan kontrollprogrammet netdclient och drivrutinen för att kunna skicka kommunicera med den inbyggda http-servern. Även koppling till IVBS är nödvändig och detta via vanlig tcp/ip-kommunikation.



Figur 4.1 Planerad arkitektur.

5 Resultat

I detta kapitel redovisas de resultat som uppnåtts under arbetet.

5.1 Drivrutinen ivbsnetd

Drivrutinen som utvecklats består av en http-server som svarar på kommandon och namnet till denna drivrutin blev "ivbsnetd" som agerar som en tjänst eller "daemon" som utför operationer mot en IVBS-proxy.

ivbsnetd medlar mellan förfrågningar från kärnan och IVBS med stöd för inloggning, montering av skivavbilder samt läs- och skrivoperationer. Instruktioner om vilka skivavbilder som skall monteras läses antingen in under start från en JSON-formaterad konfigurationsfil eller via http-servern under körning. http-servern lyssnar på port 8080 och behandlar JSON-paketerad data med kommandon.

Kommunikation genom NBD

Här följer beskrivningen av ett vanligt use case för drivrutinen när en ny avbild sätts upp för läsning och skrivning. Alltså flödet av händelser och utbyte av information. Texten följer flödet i figur 5.1.

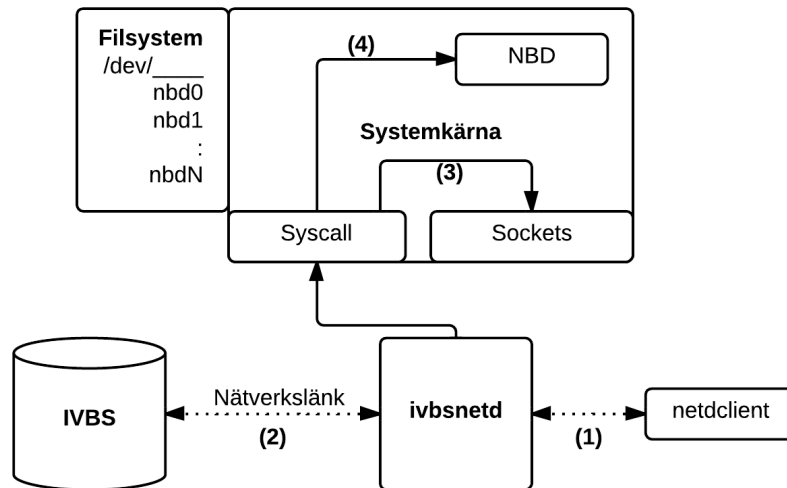
Klientprogrammet skickar en förfrågan om att montera en ny avbild(1). När ivbsnetd ska montera en avbild börjar den först med att be IVBS förbereda avbilden för läsning och skrivning (2). IVBS vill då ha namnet på avbilden som skall användas. Förutsatt att avbilden existerar svarar IVBS med avbildens storlek. Avbildens storlek sparas och ivbsnetd fortsätter med att sätta upp kommunikation med NBD i kärnan.

Fortsättningsvis utförs systemanrop för att sätta upp NBD.

Först görs ett systemanrop till funktionen socketpair för att skapa ett så kallat "socketpair" (3). Dessa agerar som en tvåvägskommunikation med NBD. ivbsnetd får två File Descriptors (FD) tillbaka från systemanropet. Dessa två FDs är integers som identifierar kommunikationsportar i systemet. Den ena behålls och den andra skickas till NBD genom ytterligare ett systemanrop via ioctl.

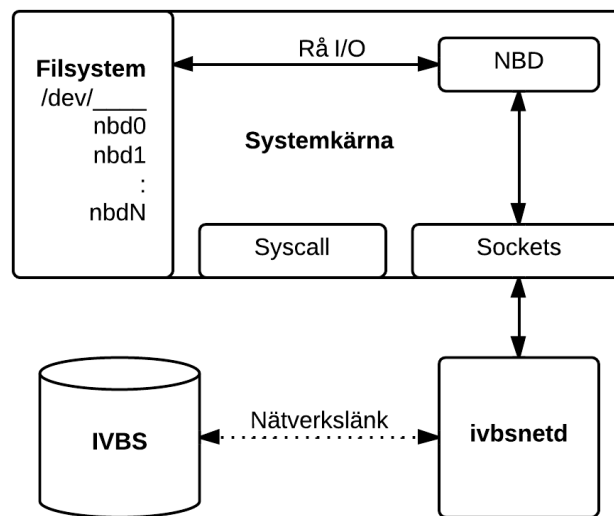
Sedan instrueras NBD om vilken storlek den virtuella blockenheten har samt vilken FD som skall användas med systemanrop via ioctl (input/output control) (4).

Slutligen görs ett ioctl-anrop till NBD om att ivbsnetd är redo att ta emot I/O. Detta systemanrop returneras först efter att NBD ombeds att stänga ner kommunikationen eller att fel uppstår som tvingar NBD att stänga ner.



Figur 5.1 Systemanrop under monterings-procedur.

Nu är NBD och ivbsnetd klara för att ta emot förfrågningar från andra program och operativsystemet som vill läsa och skriva till avbilden. I/O leds genom NBD som skickar förfrågan om en läs- eller skrivoperation till ivbsnetd. Förfrågan paketeras om till det format som IVBS vill ha och skickas därefter vidare ut på nätverket. När ett svar fås från IVBS skickar ivbsnetd i sin tur tillbaka svaret till NBD, se figur 5.2.



Figur 5.2 Kommunikation i systemet under körning.

Varje avbild som monteras via ivbsnetd använder en separat anslutning mot IVBS med sitt eget set av goroutines som sköter all kommunikation. Separationen av anslutningar avhjälper eventuella fel i en specifik anslutning att påverka andra anslutningar.

Det faktum att kommunikation med IVBS sker över tcp/ip genom systemkärnan ignoreras i figur 5.1 och figur 5.2.

Kontrollprogrammet netdclient

netdclient är ett program som körs från terminalen och kan kommunicera i JSON med http-servern i drivrutinen. Programmet tar in enkla textkommandon (tabell 5.1) och frågar efter mer indata om det krävs. Klienten formaterar då kommandon till JSON, skickar det över http och presenterar svar på ett läsligt format för användaren.

Check	Testar om http-servern är tillgänglig
Disc	Stänger ner http-servern.
Exit	Stänger av kontrollprogrammet.
Help	Visar möjliga kommandon
Lista	Ber http-servern om en lista av alla tillgängliga NBD enheter att kunna montera till.
Listm	Ber http-servern om en lista av alla NBD enheter som blivit monterade.
mount <device> <image> <user> <pass>	Monterar en avbild på en NBD-enhet med angiven användare och lösenord.
umount <device>	Stänger av sessionen på vald NBD enhet.

Tabell 5.1 Kommandon i kontrollprogrammet *netdclient*.

Innan varje kommando körs som skickar data till http-servern försäkras sig kontrollprogrammet om att http-servern är igång. Om inget svar fås anses http-servern vara otillgänglig. Då utförs ingen åtgärd hos kontrollprogrammet innan anslutningen till http-servern är etablerad igen.

5.2 Hastighetstest

När väl alla delarna av drivrutinen var färdig gjordes ett test att skriva 1000MB till en avbild i en testmiljö hos Ilait. Detta gav en hastighet på 4-5MB/s i genomsnitt. Testet kördes via trådlöst nätverk och inte via trådat nätverk. När testet kördes igen via trådat nätverk kom hastigheten upp i 10MB/s i genomsnitt. Detta gav oss en högre hastighet än förväntat. Datahastighet var inget krav då funktionalitet till NBD hos drivrutinen var av större prioritet.

Tester av Ilait

Efter arbetet hos Ilait har ytterligare tester utförts av vår handledare på företaget för att uppskatta *ivbsnetd* prestanda. Programmet kördes den gången på samma dator som en utvecklingsversion av IVBS för att minimera eventuella felkällor i kommunikationen mellan datorer. Figur 5.3, 5.4 och 5.5 visar vilka kommandon som kördes samt resultaten.

Testet använder sig av UNIX-kommandot *dd* som kopierar binär data mellan filer eller enheter.

Första försöket i figur 5.3 gav upp till 80 % snabbare resultat än tidigare mätningar men handledaren påpekade att slumpgeneratoren */dev/urandom* kan påverka resultatet. Generering av slumpmässig data i större mängder tar lång tid.

```
# sudo dd if=/dev/urandom of=/dev/nbd3 bs=1M
348127232 bytes (348 MB) copied, 18.7872 s, 18.5 MB/s
```

Figur 5.3 Skrivning av data från */dev/urandom*

Inför nästa test genererades data i förväg till en lokal fil som monterades i RAM för snabb åtkomst. Resultaten i figur 5.4 påvisar markant förbättring med hastigheter närmare tio gånger hastigheten från det tidigare testet.

```
# sudo dd if=/ramdisk/image.img of=/dev/nbd3 bs=1M  
1048576000 bytes (1.0 GB) copied, 6.77119 s, 155 MB/s
```

Figur 5.4 Skrivning av data från en fil i RAM

På grund av hur IVBS är designat rekommenderade handledaren på företaget att samma test kördes igen. Detta för att IVBS tar extra tid att allokeras utrymme första gången skrivningar sker på en avbild. Denna gång skrivs existerande data över av samma. Resultatet i figur 5.5 visar ännu en gång en markant förbättring med över 150 %.

```
# sudo dd if=/ramdisk/image.img of=/dev/nbd3 bs=1M  
1048576000 bytes (1.0 GB) copied, 4.39422 s, 239 MB/s
```

Figur 5.5 Skrivning av data från en fil i RAM andra gången

Testerna påvisar flera felkällor som påverkat resultaten i hög grad och allt eftersom eliminerats. Till slut ger testerna en uppfattning om vilken prestanda som kan förväntas av produkten. Dock har endast tester i form av sekventiella skrivningar genomförts och data för andra skrivmönster saknas. Det finns inte heller några tester där data läses tillbaka.

5.3 Buggar

ivbsnetd innehöll kända buggar och fel vid arbetets slut som inte hanterades eller löstes. Dels på grund av tidsbrist och dels på grund av prioritering av viktiga funktioner.

Zombie-processer och låsning

Programmet kunde under körning frysa och sluta svara under vissa förhållanden. Speciellt om ingen I/O hanterades under mer än 30 sekunder. Detta resulterar i att programmet genererar en eller flera zombie-processer som sitter kvar i systemet; ofta tills en omstart av operativsystemet sker. Problemet uppstår för att IVBS stänger anslutningen mot klienter som inte skickar keepalive-paket eller kommunikation inom 30 sekunder. Då *ivbsnetd* saknar funktionen att skicka keepalive-paket med jämna mellanrum måste t.ex. tester schemaläggas rätt så att anslutningen inte hinner stängas.

Git repo

Då GitHub använts under arbetet finns all källkod tillgänglig på ett repo under adressen: <https://github.com/demonicblue/netdriver>

6 Slutsats och diskussion

I projektets start gick tid åt till att göra efterforskning om huruvida UIO var den teknik som skulle användas för att lösa problemet. Dessvärre visade det sig att så var inte fallet och arbetet skiftade mot att söka efter nya alternativ. Efterforskningar utfördes främst på Internet för andra alternativ. NBD dök upp som ett intressant alternativ. Dessvärre saknade NBD detaljerad dokumentation om hur NBD faktiskt fungerade och vad som krävdes för att anropa NBD från ett annat språk än C. Innan inlärningsprocessen av Go påbörjades krävdes det studier av NBDs kod samt koden i projektet BUSE. BUSE är en enklare blocklagringsdrivrutin i userspace som utnyttjar NBD och är skriven i C. Inläring av Go var en viktig del av projektet då ingen erfarenhet av programmeringsspråket fanns sedan tidigare.

Att lära sig ett nytt språk är relativt enkelt om man kan grunderna i programmering och datastrukturer. Go har en egen självlärnings-guide, *Tour of Go*, och det blev lite som att få en intensivkurs. *Tour of Go* gav en snabb genomgång av språkets fundamentala struktur, syntaxen och språkets unika egenskaper. Genom att man själv kunde redigera och exekvera koden direkt från webbläsaren i *Tour of Go* var det mycket enkelt att komma igång med Go. Inlärningsprocessen av Go kunde förkortas då handledaren på företaget arbetat i Go de senaste två åren. Han kunde även förklara det som var svårt att förstå samt ge tips.

Under utvecklingen av drivrutinen och kontrollprogrammet försökte vi att dela upp arbetet på ett sådant sätt att man kunde arbeta individuellt och därmed kunde arbeta effektivare. Detta fungerade bra och utifall man själv inte visste vad som skulle göras kunde man snabbt ta ett möte och diskutera problemet. Ett bollplank var aldrig långt borta. Genom att utnyttja vissa idéer från projekthanteringsmetoden Scrum upprättades ett eget arbetssätt där vi skrev upp uppgifter på post-it lappar. Uppgifternas prioritet blev tilldelade olika färgkoder för att få en bättre översikt av vad som borde göras först.

Att arbeta på plats hos företaget gjorde det lätt att hålla god kommunikation med företaget och hela tiden diskutera frågor gällande projektet. Genom att arbeta på plats gavs en extra knuff att prestera bra gentemot om arbetet skulle skett någon annanstans.

När drivrutinens grundläggande funktionalitet var klar hade en större del av projektets tid redan blivit förbrukad. Det fanns ingen större tid att lägga på implementation av felsäkerhet, failover och testning. Ordentliga tester måste skrivas och sedan utföras av llait innan en implementation i företagets produktionsmiljö kan övervägas.

Ur en miljöaspekt kan detta examensarbete ha bidragit till en mindre förbrukning utav el. Detta då man utvecklat till en lösning i molnet. Istället för att användaren själv införskaffar hårdvara för lagring så utnyttjar man den redan tillgängliga hårdvaran hos företaget. Detta leder till att mindre antalet hårddiskar körs samtidigt och bidrar till en minskning av elförbrukning.

Projektet bidrar till en ökad användning av molnet. Företag behöver inte längre själva installera och underhålla en serverhall. Detta ansvar faller på hostingföretag, som llait. Något som bygger en helt ny bransch med nya möjligheter när allt fler företag väljer molnet.

Referenser

1. Chacon, Scott. 2009. *Pro Git. 3.2 Git Branching - Basic Branching and Merging*, 34-55. E-bok.
2. Google. Tour of Go. <http://tour.golang.org> (Acc 2013-05-17)
3. Salzman, Peter Jay. 2001. The Linux Kernel Programming Guide. *1.1 What Is A Kernel Module?*
<http://www.tldp.org/LDP/lkmpg/2.6/html/lkmpg.html> (Acc 2013-05-17)
4. Gerrand, Andrew. 2013. The Go Programming Language Blog. *Go 1.1 is released*, 13 maj.
<http://blog.golang.org/2013/05/go-11-is-released.html> (Acc 2013-05-17)
5. Jones, M. Tim. 2007. Anatomy of the Linux kernel. *developerWorks*. 6 juni.
<http://www.ibm.com/developerworks/library/l-linux-kernel/> (Acc 2013-05-15)
6. Google. FAQ. *Ancestors*. <http://golang.org/doc/faq#ancestors> (Acc 2013-06-03)
7. Google. FAQ. *Creating a new language*.
http://golang.org/doc/faq#creating_a_new_language (Acc 2013-06-03)
8. Scrum.org. What is Scrum? *A better way of working*.
<http://www.scrum.org/Resources/What-is-Scrum> (Acc 2013-06-03)
9. LinuxQuestions.org. 2009. *Block devices and block sizes*.
http://wiki.linuxquestions.org/wiki/Block_devices_and_block_sizes (Acc 2013-06-03)
10. Bellevue Linux. 2005. Kernel Space Definition.
http://www.linfo.org/kernel_space.html (Acc 2013-06-03)
11. Google. Effective Go. *Goroutines*.
http://golang.org/doc/effective_go.html#goroutines (Acc 2013-06-03)
12. Google. FAQ. *What are the guiding principles in the design?*
<http://golang.org/doc/faq#principles> (Acc 2013-06-03)
13. Cox, Alan. 2009. Linux Allocated Devices.
<https://www.kernel.org/doc/Documentation/devices.txt> (Acc 2013-06-03)
14. Json.org. *What is JSON?* <http://www.json.org> (Acc 2013-06-03)
15. Solie, Karl och Lynch, Leah. 2004. CCIE Practical Studies Volume II. *Chapter 6. QoS Rate Limiting and Queuing Traffic. The Basics: FIFO Queuing*. E-bok

16. Google. The Go Programming Language.
<http://www.golang.org> (Acc 2013-06-03)
17. NBD client and server source.
<https://github.com/yoe/nbd> (Acc 2013-06-03)
18. BUSE source code.
<https://github.com/acozzette/BUSE> (Acc 2013-06-03)
19. Corbet, Jonathan , Kroah-Hartman, Greg och Rubini, Alessandro. 2005. *Linux Device Drivers*. 3. uppl. Sebastopol: O'Reilly Media
20. Torvalds, Linus
<https://groups.google.com/forum/?hl=en&fromgroups#!msg/comp.os.minix/dINtH7RRrGA/SwRavCzVE7gJ> (Acc 2013-06-03)
21. The Enhanced Network Block Device.
<http://enbd.sourceforge.net/> (Acc 2013-06-03)
22. Network Block Device.
<http://nbd.sourceforge.net/> (Acc 2013-06-03)
23. Ilait Virtualized Block Storage.
Hansson, F., Gustafsson, A. och Grüner, M. (2011) *Ilait Virtualiserad blocklagring* .
Göteborg : Chalmers University of Technology
24. Storage area network (SAN) protocols: iSCSI and Fibre Channel
<http://searchitchannel.techtarget.com/tip/Storage-area-network-SAN-protocols-iSCSI-and-Fibre-Channel>
(Acc 2013-10-01)
25. crypto – The Go Programming Language
<http://golang.org/pkg/crypto/>
(Acc 2014-05-13)
26. mattn/go-gtk
<https://github.com/mattn/go-gtk>
(Acc 2014-05-13)