

FUSEX - A file system interface for XML Store

Bo Bendtsen
Christian Hemmingsen

19. maj 2004

Vejleder:
Fritz Henglein

Resumé

FUSEX er en implementation af et filsystem, der gør brug af XMLSTORE. XMLSTORE er et træstruktureret, værdiorienteret lager designet til opbevaring af XML dokumenter. At bruge det som basis for et filsystem giver nogle andre muligheder og begrænsninger for implementationens funktionalitet og semantik.

Udover at bruge XMLSTORE, har vi været nødt til at have en grænseflade til operativsystemkernen. Til dette har vi anvendt eksisterende software fra AVFS projektet[1].

Vi har implementeret et interface mellem en eksisterende XMLSTORE implementation og linux kernen og evalueret en passende semantik for dette.

Vi har vist at FUSEX via XMLSTORE er i stand til at tilbyde et distribueret filsystem med en komplet historik over alle filers tilstande, samt en semantik omkring opdatering, der som udgangspunkt er stringent single-copy, men også potentielt konfigurerbar.

Indhold

1	Indledning	4
1.1	Problemerne	4
1.1.1	Problemer med distribuerede filsystemer	4
1.1.2	Versionsstyringssystemer	4
1.2	Konklusion	5
2	UNIX filsystemer	6
2.1	Struktur — biblioteker og filer	6
2.2	Operationer til manipulation af filsystem	7
2.3	Håndtering af åbne filer	7
2.4	Information om filer og filsystemer	8
2.5	Rettigheder og ejerskab	8
2.6	Andre slags filer og operationer	9
2.7	Programmering med filsystemer	9
3	Implementation af filsystemer i LINUX	10
3.1	LINUX' VFS	10
3.1.1	(Af)montering – (un)mount	11
3.1.2	Superblock	11
3.1.3	Inodes	11
3.1.4	Åbne filer	12
3.1.5	Implementation af filsystemer med VFS	12
3.2	Implementation af filsystemer i user space	12
3.2.1	FUSE	13
3.2.2	FUSE' kernemodul	13
3.2.3	FUSE bruger-proces, grænseflade	13
3.2.4	Metoder i FUSE grænsefladen	14
3.2.5	Fuse-j	16
4	DVM og XMLSTORE	17
4.1	Værdier	17
4.1.1	Oprettelse og deling af værdier	18
4.1.2	Celler	19
4.2	DVM	19
4.2.1	Tilgang til træer	19
4.2.2	Oprettelse og ændring af dokumenter	21
4.2.3	Persistens af dokumenter	21
4.2.4	Symbolske navne for værdier	21
4.2.5	DVM i distribuerede miljøer	21
4.3	XMLSTORE	23
4.3.1	Reference service	23
4.3.2	Navne server	23

4.3.3	Disks	23
5	Et værdiorienteret filsystem	25
5.1	Krav til filsystemet	25
5.2	Generelt design af filsystemet	25
5.2.1	Filer, biblioteker og symlinks	26
5.2.2	Opdateringer	26
5.3	Distribuering	27
5.4	Replikation	28
5.5	Historik	28
5.6	Semantik for filkonsistens	29
5.6.1	Opdateringspolitik	29
5.7	Semantik for FUSEX	31
5.7.1	Inferenssystem for FUSEX	31
6	Programmeringsovervejelser	35
6.1	Kodeoversigt	35
6.2	Status på FUSEX	35
7	Brugervejledning	36
7.1	Installation	36
7.1.1	FUSE-1.1	36
7.1.2	FUSE-J	36
7.1.3	XMLSTORE	36
7.1.4	FUSEX	37
7.2	Brug	37
7.2.1	Lokalt	37
7.2.2	P2P	37
7.2.3	Afkobling	37
7.2.4	Formattering af FUSEX-disk	37
8	Afprøvning og test	38
8.1	Afprøvning	38
8.2	Ydelse	39
A	Projektforeslag (på engelsk)	41
B	Kildekode	42
B.1	XMLStoreFilesystem.java	42
B.2	XMLStorePeer.java	56
B.3	mount.fusex	58
B.4	xmlstorepeer	58

1 Indledning

“If the network is the computer, how do we program it?”

Dette spørgsmål er hvad motiverer PLAN-X projektet[2].

Inden for eksisterende populære softwareplatforme er der i høj grad en dominans af objektorienterede/imperative systemer (JAVA, .NET, etc.). Med internettets enorme popularitet og den almindelige udbredelse af højhastighedsforbindelse til selvsamme, bliver spørgsmålet hele tiden mere aktuelt. PLAN-X vil gerne udvikle en softwareplatform der er værdiorienteret. Dette projekt er et foreslået underprojekt¹ under PLAN-X, der har til formål at implementere et filsystem baseret på værdiorienterede principper til LINUX operativsystemet. Meningen er at det lager der skal ligge til grund for filsystemet skal være den del af PLAN-X projektet der er XMLSTORE.

1.1 Problemerne

Hvilke opgaver står vi overfor? Vi skal forstå hvad et filsystem - under linux - er. Dernæst skal vi lære hvordan man designer et, og hvilke værktøjer, som vi har til rådighed, vi vil benytte. Vi skal forstå vores medie - Document Value Model (DVM) i form af XMLSTORE - og indse hvilke muligheder en værdiorienteret platform vil give et filsystem. Denne rapport vil beskrive disse overvejelser, samt selve designet og afprøvning af FUSEX.

Navnet FUSEX er en sammenskrivning af FUSE og XMLSTORE. FUSE er det værktøj vi valgte at bruge til at skrive et filsystem der kørte i user-space.

1.1.1 Problemer med distribuerede filsystemer

Distribuerede filsystemer er særligt udsatte når det kommer til ydelse. Klient/server systemer, der er de mest udbredte, er eksponeret for flaskehalsproblemer på serversiden. Unødvendig trafik over netværk bliver helst undgået, da denne er meget dyr. Måden disse distribuerede filsystemer løser mange af de problemer med ydelse er ved at ændre kravene og semantikken omkring konsistens og samtidig tilgang.

Vi vil vise et distribueret filsystem der implementerer en sand single-copy semantik. Ved brug af DVM, et værdiorienteret API.

1.1.2 Versionsstyringssystemer

Versionsstyringssystemer bliver især brugt inden for programudvikling. De tjener oftest flere formål, først og fremmest holder det styr på hvordan filer ser ud over tid². De tjener som en backup af filerne, da den enkelte kun vil arbejde på en lokal kopi ogda versionstyringssystemet oftest gemmer filerne

¹se bilag A

²Dog kun de tidspunkter hvor ændringer er ført ind i systemet

på et “sikkert” sted. En tredje ting er dets evne til at påpege konflikter, som kan opstå når flere arbejder på samme fil. Dog er det nødvendigt for den enkelte bruger af systemet at bestemme hvornår en ændring skal indføres i systemer og i visse tilfælde at løse en konflikt.

FuseX er ikke et versionstyringssystem. Det har dog flere egenskaber der minder om dem man har for et versionsstyringssystem. Bl.a. historik og adgang til tidligere tilstande, men uden brug af speciele klienter.

1.2 Konklusion

Vi har vist at det er muligt at implementere et filsystem oven på XMLSTORE og (ad omveje) at integrere det med LINUX kernen, så det kunne monteres under LINUX og tilgås normalt via det systemkald til kernen.

Vi har implementeret historik for filsystemet og i realiteten lavet et filsystem med live backup.

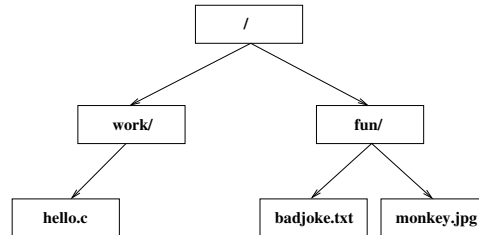
Vi har evalueret og implementeret en single-copy semantik for et distribueret filsystem, omend det ikke lykkedes os få det testet i et ægte distribueret miljø, grundet ydre faktorer vedrørende tredjeparts software.

2 UNIX filsystemer

Vi vil i dette afsnit se på hvordan et UNIX filsystem ser ud fra applikationsprogrammørens synspunkt. Hvordan den logiske opbygning er, hvilke operationer det skal understøtte og hvilke objekter det indeholder.

2.1 Struktur — biblioteker og filer

Strukturen i et UNIX filsystem er som et omvendt træ, roden i toppen og bladene i bunden. For at opnå denne struktur er der to slags objekter: Biblioteker og filer. Biblioteker er en samling af filer, biblioteker og andre slags filer³. Idet at et bibliotek kan indeholde et andet bibliotek gør at vi kan lave den omtalte træstruktur. Filer afbilder data og er blade i træet. Til hver fil eller bibliotek er der et navn tilknyttet. For at undgå tvetydighed kan to indgange i et bibliotek ikke have samme navn, heller ikke hvis de er af forskellig type. Figur 1 viser et eksempel på et simpelt lille filsystem.



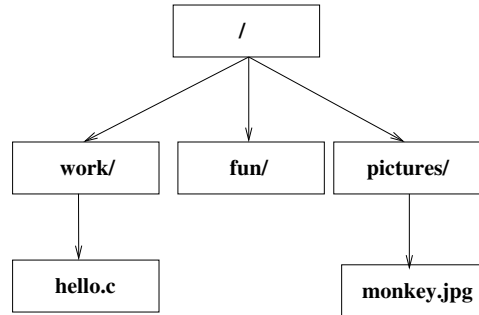
Figur 1: Simpelt filsystem. Bibliotekers navne er postfixet med “/”

Udover at en fil eller et bibliotek har et navn i det bibliotek det befinder sig i, opererer man ofte med “stier”. En sti er den information der skal til for at finde fra et bibliotek til et andet givent bibliotek eller fil. Stier kan være absolute eller relative. En absolut sti er vejen fra roden og en relativ sti er ens aktive arbejdsbibliotek⁴. Til at “gå opad” i et filsystem er der i hvert bibliotek en indgang med navnet “..”.

Stien bliver angivet som en streng, med de biblioteker man skal traversere på vejen spareret med “/”, en såkaldt sti-separator, for til sidst at ende med navnet på en fil eller et bibliotek. Betragter vi figur 1 igen, er den absolute sti til `hello.c` således være `/work/hello.c`⁵. Hvis arbejdskataloget var `/fun` ville den relative sti til `hello.c` være `../work/hello.c`.

³Ordet fil bliver ofte brugt om andre slags objekter end det vi typisk forstår ved en fil

⁴Kernen holder i så vid udstrækning det er muligt styr på ens aktive arbejdskatalog. Dette kan ændres med `chdir`. Det aktive arbejdsbibliotek er ikke noget vi vil beskrive yderligere, da det som sådan ikke har noget med filsystemer at gøre, men med fortolkning af relative stier



Figur 2: Filsystemet fra figur 1 efter diverse opdateringer

Vi vil ikke beskæftige os yderligere med stier, da de ikke er vigtige for vores implementation⁵.

2.2 Operationer til manipulation af filsystem

I det følgende vil ord i **fed** være navne på systemkald. Et filsystem er ikke meget beventt hvis ikke man kan manipulere indholdet af det. For at være brugbart skal man kunne oprette og fjerne biblioteker. Det gør man med **mkdir** og **rmdir**. Ligeledes med filer, de oprettes og fjernes med **open**⁶ og **unlink**. Til at ændre en fil eller et biblioteks navn bruges **rename**. En navneændring kan også involvere at en fil eller et bibliotek bliver flyttet til et andet sted i filsystemet. Antag følgende sekvens af systemkald:

```

mkdir("/pictures")

rename("/work/monkey.jpg","/pictures/monkey.jpg")

unlink("/fun/badjoke.txt")
  
```

Resultatet er ses i figur 2.

Disse fire systemkald er således nogle vi kan bruge til at organisere data med.

2.3 Håndtering af åbne filer

Vi har set hvordan vi kan slette og flytte filer i et filsystem. For at arbejde med indholdet af en fil, ændre det eller læse det, også kaldet Input/output — eller I/O — snakker man om at den skal “åbnes” først. Til dette bruges **open**. Når man åbner en fil får man en *filedescriptor*. Denne filedescriptor repræsenterer en tilstand for den I/O man er ved udføre på filen. I stedet

⁵Man kan kalde et systemkald med en relativ sti, men når et kernekald får en sti, er den altid gjort absolut

⁶**open** er meget alsidig og kan bl.a. bruges til at oprette filer, mere om det senere

for selv at holde styr på hvor man er i en fil, kan man blot bede om at læse fra eller skrive til en fil fra det sted hvor man var kommet til. Til dette bruger man **read** og **write**. Efter en af disse to er blevet kaldt, opdateres tilstanden. Det er dog muligt at ændre hvor i filen man gerne vil starte næste I/O operation fra, med **lseek**.

Når man er færdig med at lave I/O på en fil, kan man lukke den igen med **close**.

Det er muligt at trunkere en fil med **truncate**. Filens længde bliver sat til 0 og dens indhold vil være tomt.

2.4 Information om filer og filsystemer

Ofte vil man gerne have noget information før man går igang med at arbejde med filer og biblioteker. Typisk vil man gerne have en liste med hvilke filer og biblioteker der er i et bibliotek. Til det kan man bruge standardbiblioteksfunktionerne⁷ **opendir**, **readdir** og **closedir**. Ved at åbne et bibliotek med **opendir** får man en slags descriptor. Successive kald til **readdir** med denne descriptor som argument vil returnere et navn på en fil eller et bibliotek, indtil alle navne har været returneret. Når man er færdig med kan man kalde **closedir**.

Ud over at vide hvilke navne der er hvor, har man måske brug for yderligere information om en fil eller et bibliotek. Et filsystem holder styr på størrelsen (hvis det er en fil), hvornår der sidst var en ændring i indholdet, rettigheder og ejerskab (se næste afsnit) og mange andre ting. For at få fat i den information bruges **stat**.

2.5 Rettigheder og ejerskab

I et flerbruger system er det som regel ikke ønskeligt at alle har lige rettigheder til at ændre i filsystemet. UNIX specificerer ejerskab for filer og biblioteker. Samtidigt er der også rettigheder for filer og biblioteker. Rettigheder fortæller "hvem må hvad" baseret på en opdeling af brugere i tre grupper, ejeren, unixgruppemedlemmer og andre. En fils ejerskab er opdelt i to, en ejer og en unixejergruppe. Prøver en bruger at udføre en operation, der ikke er tilladt for den gruppe denne befinder sig i forhold til filens ejerskab, så vil det resultere i fejl og en uændret tilstand for filen eller biblioteket.

Ejerskab og rettigheder for filer kan modificeres med **chmod** og **chown**⁸. UNIX' sikkerhedsmodel med er simpel, men nem at implementere.

⁷Grunden til at det ikke er systemkald der beskrives her er, at de systemkald der er til denne slags opgave er meget upraktiske at bruge

⁸**chown** er typisk kun tilgængelig for systemets superbruger

2.6 Andre slags filer og operationer

Udover filer og biblioteker, understøtter UNIX filsystemer som regel flere andre slags filer. En anden slags fil er en “symbolsk lænke” (symlink). En symbolsk lænke indeholder kun en sti. Til at oprette symbolske lænker bruges **symlink** og til at læse den indeholdte sti **readlink**. Symbolske lænker bruges ofte til at skabe en anden logisk struktur over andre.

Ofte understøttes også “hårde lænker” (hard links). I forbindelse med hårde lænker er det relevant at komme ind på *inodes*. UNIX filsystemer definerer at der for hver objekt i filsystemet er en inode. En hver inode har et — for det filsystem den befinder sig på — unikt nummer. Da den traditionelle måde at implementere biblioteker på er at have en liste med navne og hvilken inode det svarer til, er det at lave en hård lænke simpelthen til en fil, simpelthen at tilføje et nyt navn til biblioteket med den anden fils inode nummer. Dette introducerer dog et behov internt i filsystemet for at holde styr på hvor mange navne der er tilknyttet til en inode.

Hårde lænker således anderledes er fra symbolske lænker på den måde at de ikke kræver nogle fortolkning, som **readlink** gør for symbolske lænker. En hård lænke er umuligt at skelne fra den originale fil. En hård lænke oprettes med **link**. Der kan som regel ikke oprettes hårde lænker til biblioteker⁹. At kende en fil eller biblioteks inode nummer kan være praktisk og brugbart i specielle tilfælde, men er generelt ikke nødvendigt.

2.7 Programmering med filsystemer

Dette var en kort og ikke specielt dybdegående beskrivelse af hvilke operationer der er tilgængelig for applikationsprogrammøren til håndtering af UNIX filsystemer. Der er ofte biblioteksfunktioner tilgængelige der “pakker” systemkaldene ind i et lettere tilgængeligt interface.

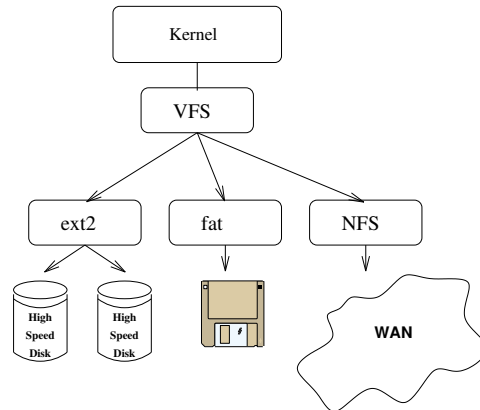
Dokumentation og nøjagtig beskrivelse af hvad de enkelte systemkald helt præcist gør, tager som argumenter og returnerer findes som regel on-site via man pages. Systemkald er dokumenteret under sektion 2 og biblioteksfunktioner under sektion 3. Vil man således se dokumentationen **read** skrives på systemet

```
$ man 2 read
```

og vil man se dokumentationen til biblioteksfunktionen **readdir**

```
$ man 3 readdir
```

⁹Det vil risikere at skabe cykler i filsystemet der kan være meget svære at opdage



Figur 3: VFS er det interface som alle filsystemer skal implementere, hvis de skal fungere transparent med UNIX' filtræ

3 Implementation af filsystemer i LINUX

I afsnit 2 så vi hvordan et filsystem ser ud fra applikationsprogrammørens synspunkt. Dette afsnit vil beskæftige sig med at beskrive hvad der skal til for at implementere et filsystem i LINUX. Valget af LINUX er gjort af flere årsager. For det første ville det være en vanskelig opgave at beskrive hvordan filsystemer er implementeret generelt, da de fleste operativsystemer er forskellige i deres interne virkemåde, og det alligevel ville ende med en masse platformspecifikke beskrivelser. For det andet er det i LINUX muligt at se kildekoden til kernen og diverse filsystemimplementationer og dermed se hvordan de fungerer sammen.

3.1 LINUX' VFS

VFS¹⁰ er i LINUX er det "lag" i kernen der tager sig af alle filsystemer. Det er designet på sådan , at de øvre lag i kernen kan behandle alle filsystemer på en ens måde. Et filsystem skal til gengæld en grænseflade defineret af VFS.

Grænsefladen indeholder alle de funktioner der er relateret til tilknytning og montering af et filsystem, caching af navne på inodes, håndtering af åbne filer og inodes, super-blocken og låsning. Ikke alle metoder i grænsefladen behøver at være understøttet, for at et filsystem kan være funktionelt, i hvilket tilfælde en generisk metode vil blive kaldt. Brown har[3] har skrevet en kort guide til VFS.

VFS definerer en grænseflade for et UNIX filsystem, samt stukturer for hvordan det er opbygget, f.eks. åbne filer og inodes.

¹⁰Virtual File System

Et filsystem behøver ikke at fysisk indeholde disse ting¹¹, men skal implementere deres funktionalitet til, omend det blot er på et niveau hvor det lader som at de eksisterer.

VFS definerer de operationer et filsystem kan implementere for de interne strukturer i filsystemet.

For mange strukturer i et filsystem, såsom inodes og åbne filer og allokering af plads på et filsystem, vil vi ikke gå i dybden med at beskrive hvordan de fungerer og hvad de burde gøre. Det er som vi vil se senere ikke interessant for dette projekt. Det er nok at vide at VFS eksisterer. Vi kunne have brugt VFS og det vil måske blive brugt mere direkte i en fremtidig iteration af dette projekt.

3.1.1 (Af)montering – (un)mount

Montering er den proces at få et filsystem gjort synligt i sin UNIX filstruktur. Kernen beder det modul der implementerer filsystemet om at indlæser superblocken fra et givent medie.

Ved afmontering beder kernen modulet om at gøre sig færdig, hvilket oftest inkluderer at frigøre de ressourcer denne instans af filsystemet har optaget.

3.1.2 Superblock

Super-blocken er “indgangen” til filsystemet. Det er den første der bliver indlæst og indeholder information om filsystemets tilstand og hvor roden i filsystemet er. Da et filsystem ikke kan monteres i tilfælde af en korruperet super-block, gør mange filsystemer det at de gemmer ekstra reserve super-blocks på forudbestemte steder i filsystemet.

3.1.3 Inodes

En inode er et basis-objekt i et filsystem. Den kan repræsentere en, et bibliotek, en lænke, eller noget helt andet. VFS behandler dem alle sammen ens og lader det være op til selve implementationen, eller andre lag af kernen at behandle dem forskelligt, alt efter hvad de repræsenterer. Et hvert objekt i et filsystem – repræsenteret ved en inode – har et – for filsystemet – unikt nummer. Inode strukturen som VFS arbejder med indeholder information om objektets type, ejerskab, rettigheder, oprettelsestidspunkt m.m. [3]

Et filsystem er selv ansvarlig for at generere og håndtere placering af inodes på medie, mens VFS tager sig af at udlede inodes ud fra stier (f.eks.:

¹¹Der findes adskillige eksempler på filsystemer der ikke implementerer superblocks og/eller inodes, især uden for “UNIX verden”. Et argument for at udelade det kan være pga. pladsovervejelser i forbindelse med filsystemer der skal være på medier med begrænset kapacitet, f.eks. floppy disketter, eller at det er unødvendigt da filsystemet er på et ikke modificerbart medie, f.eks. en CD-ROM

`/work/hello.c`), givet at de nødvendige dele af grænsefladen til denne operation er implementeret korrekt.

3.1.4 Åbne filer

De mange af de operationer man kan udføre på inodes er mere eller mindre “engangsoperationer”, og ikke indeholder nogen tilstand mellem kald. Åbne filer er anderledes. En fil er selvfølgelig angivet ved en inode i filsystemet, men data bliver typisk angivet via indirektion¹². En åben fil bliver ofte i computerlitteratur omtalt som en strøm af data. For at få den effekt, at man ved successive læsninger eller skrivinger har at gøre med en strøm, så er det nødvendigt at der gemmes en tilstand for den åbne fil, hvilket kernen så gør. Ideen med at gemme filens tilstand i kernen er bla. at den så kan deles mellem processer. Der er selvfølgelig også metoder til at manipulere tilstand.

Tilgangen til åbne filer foregår via en såkaldt “file descriptor”.

3.1.5 Implementation af filsystemer med VFS

Dette var en kort og på ingen måde komplet gennemgang af VFS og den grænseflade det præsenterer. At implementere et filsystem er ikke nogen lille opgave, og at arbejde med kerneprogrammering er altid farligt, i den forstand, at visse fejl der normalt (dvs. for en bruger proces) ikke er fatale kan gå hen og blive det. Vi vil ikke bruge tid og plads på detaljer om hvordan man bruger VFS, da vi ikke har brug for det i denne opgave. En overordnet ide om hvordan det fungerer er, som tidligere nævnt, nok.

3.2 Implementation af filsystemer i user space

Med en forståelse for hvordan VFS fungerer kunne man gå i gang med at implementere et filsystem. I vores tilfælde vil vi bruge XMLSTORE til at indeholde vores filsystem (ligesom et fastpladelager typisk indeholder et eller flere filsystemer). Problemet med det er at det API der eksisterer til XMLSTORE i sin nuværende form kun er implementeret i JAVA. Dette besværliggør direkte brug af LINUX’ VFS sammen med XMLSTORE, da det ikke umiddelbart er til at få JAVA kode til at køre i kernen. Der skal under alle omstændigheder laves en form for indirekte kobling mellem XMLSTORE og kernen. Vi vil her beskrive hvordan vi er i stand til at implementere et filsystem uden for kernen.

I LINUX kan man ikke umiddelbart lave en kobling fra kernen til user space¹³, og at lave så et stykke software ville være nok til en selvstændig

¹²I op til adskillige led

¹³I GNU/HURD projektet kan man gøre det med deres “translators”, som en del af dets mikrokernedesign, men det projekt har ikke produceret en version 1.0 siden dets start i 1991

opgave. Vi ledte derfor efter software der kunne hjælpe os dette problem. Vi fandt FUSE.

3.2.1 FUSE

FUSE er en del af AVFS¹⁴-projektet[1]. AVFS er et system der gør det muligt at “se ind” i f.eks. ziparkiver, eksterne ressourcer som FTPservere og montere dem som man normalt monterer et filsystem. Fordelen ved det er selvfølgelig at et program ikke behøver at skrives om for at tilgå en bestemt slags ressource, men kan gøre brug af standard operationer for filsystemer.

AVFS er den del af projektet som specifikt tager sig af at implementere den kode der skal til for at montere forskellige slags ressourcer, f.eks. komprimerede arkiver, og er som sådan ikke interessant for os som andet end inspiration til at implementere FUSE grænsefladen.

FUSE består af tre dele:

- Et kernemodul der implementerer VFS grænsefladen.
- Et bibliotek med en grænseflade til bruger-processer.
- Et program til montering af brugerens filsystem.

3.2.2 FUSE' kernemodul

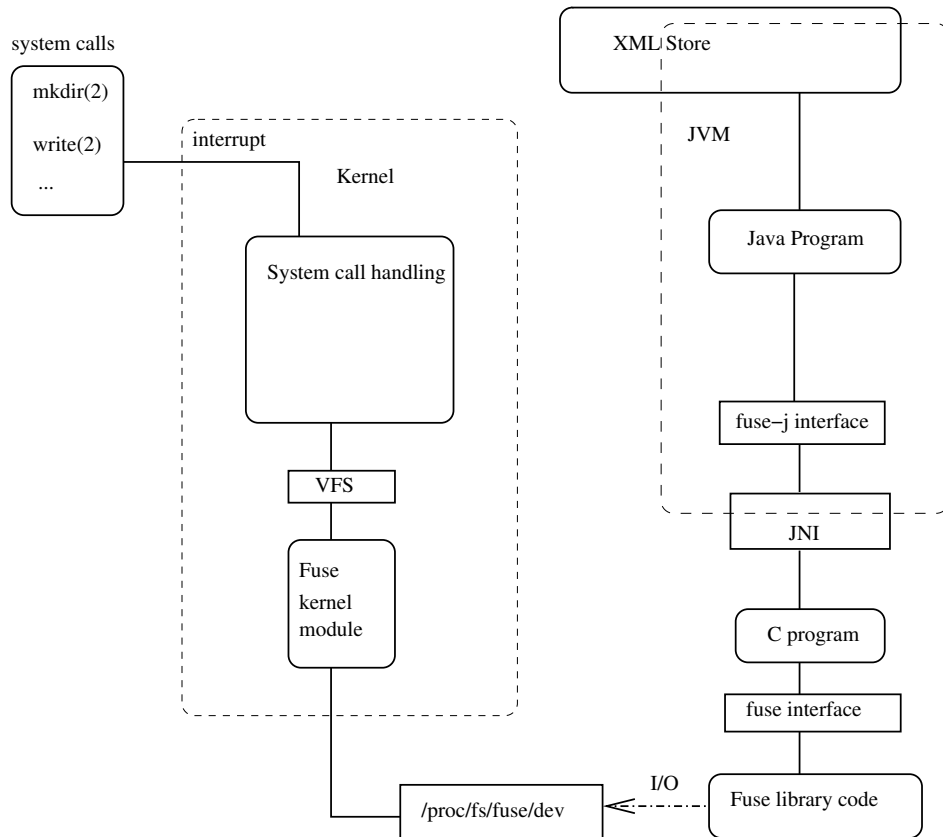
Kort sagt fungerer FUSE på den måde at det implementerer VFS grænsefladen i kernen, samt eksporterer et en speciel device fil `/proc/fs/fuse/dev`, som kommunikation med brugerprocesser skal gennem. Kernemodulet er designet sådan at det har en intern repræsentation og håndtering af inodes og tilstand for **åbne filer**. Hvad det betyder for den grænseflade som brugerprocessen skal implementere, vil vi se på i næste afsnit.

3.2.3 FUSE bruger-proces, grænseflade

Det man selv gør, er at implementere den grænseflade FUSE definerer. I programmet kalder man så et “main loop”, der sætter kommunikationen med kernen op. Det foregår på den måde at det åbner det device, der bliver eksporteret af kernemodulet, `/proc/fs/fuse/dev`.

Den grænseflade man skal implementere er lidt anderledes end den der er i VFS. Der er som tidligere nævnt ikke noget begreb om inodes og åbne filer, da det bliver håndteret af kernemodulet. Det resulterer i en meget simple grænseflade end den der er i VFS. I VFS har metoder der har med inodes at gøre, inodes som argumenter og metoder der har med åbne filer at gøre, filstrukturer som argument. FUSE grænsefladen behandler alle disse på samme måde, ved at tage stier som argument. F.eks oprettelse af bibliotek med **mkdir.:**

¹⁴A Virtual File System



Figur 4: Et system kalds vej gennem kernen til XMLSTORE og tilbage igen

```

VFS: int mkdir(struct inode * parent, struct dentry * name, int mode)
FUSE: int mkdir(const char * path, mode_t mode)

```

Som det ses er det i FUSE op til programmet at parse stien og finde ud af om det kan lade sig gøre. Man kan sige at FUSE grænsefladen minder mere om den for systemkald.

I figur 4 er der en oversigt over et systemkalds vej gennem kernen, via VFS, FUSE kernemodulet, via det specielle device `/proc/fs/fuse/dev`, til FUSE brugerprocessen.

, der så kommunikerer med XMLSTORE. I figuren optræder også FUSE-J, som er et JAVA interface til FUSE, som vi har valgt for at gøre implementationen nemmere, da API'et til XMLSTORE kun eksisterer i java.

3.2.4 Metoder i FUSE grænsefladen

Vi vil her lave en liste med de metoder fra FUSE grænsefladen som man skal/kan implementere. Hvis ikke andet er nævnt tager de alle en sti som

argument, som den fil operationen skal udføres på.

getattr() Svarer til hvad **stat** gør i UNIX.

readlink() Analog til UNIX

getdir() Gør hvad UNIX's **opendir**, **readdir** og **closedir** gør¹⁵ i en og samme operation. Til det har den bl.a. callback metode.

mknod() Bruges til at oprette objekter i filsystemet, filer, biblioteker, sym-links og specielle filer (hvis filsystemet tillader det). Den bliver kaldt hver gang en indgang i filsystemet skal oprettes.

mkdir() Opretter et bibliotek.

rmdir() Fjerner sletter et bibliotek.

unlink() Sletter en fil.

symlink() Opretter et symlink.

rename() Giver et objekt i filsystemet et nyt navn.

link() Opretter en hård lænke til en fil.

chmod() Ændrer rettigheder.

chown() Ændrer ejerskab.

truncate() Trunkerer en fil.

utime() Opdaterer tidsstempler på fil.

open() Åbner fil. Er en del anderledes end den **open** der kendes fra UNIX. Den returnerer ikke en ny filedescriptor, da det er noget kernemodulet i FUSE håndterer for os. I FUSE er **open** kun til for at kontrollere at en fil kan åbnes med de argumenter der gives med. Den opmærksomme læser vil selvfølgelig også bemærke fraværet af **close**.

read() Læser fra en fil angivet ved en sti, fra et bestemt afsæt, et bestemt antal bytes.

write() Skriver til en fil angivet ved en sti, fra et bestemt afsæt, et bestemt antal bytes.

statfs() Returnerer information om filsystemet. Tager ikke noget filargument¹⁶

¹⁵Hvis man vel og mærke vil have en liste med alle filer fra et bibliotek

¹⁶Selvom sjovt nok UNIX' gør det og returnerer statistik om det filesystem filen ligger på

release() En slags `close`. Bliver kaldt når enten

- Alle filedescriptors for en fil er blevet lukket (`closed`).
- Når alle memory mappings for filen er `unmapped`.

fsync() Beder om at synkronisere alle dele af en fil der er i hukommelse til ikke-sårbart lager.

3.2.5 Fuse-j

FUSE-J er fuse grænsefladen, men lavet i java. Det kalder ned i FUSE grænsefladen via JNI. Grunden til det er med, er for at lette integrationen med XMLSTORE der også er lavet i JAVA.

4 DVM og XMLSTORE

I det forgående afsnit blev FUSE og VFS gennemgået. Med de to ting, er kommunikationen med kernen på plads og vi er nu klar til at begynde at kigge på selve implementationen af filsystemet. Grundlaget for filsystemet er DVM og XMLSTORE som den implementation vi benytter os af. Pedersen og Pedersen[4] beskriver værdiorienteret programmering og introducerer DVM¹⁷ og XMLSTORE som en implementation i deres speciale. Se dette for en fuldstændig gennemgang af disse. Her vil vi komme med en begrænset gennemgang af DVM og XMLSTORE som en forberedelse til de næste afsnit. Har man læst Pedersen og Pedersens speciale kan dette afsnit springes over.

4.1 Værdier

Kender man til funktionsprogrammering og f.eks. SML, vil man vide at man ikke kan opdatere en værdi (value). Man kan binde en værdi til et navn, f.eks.:

```
- val t1 = (1,2); val t2 = (4,5); val t3 = (6,12);  
val t1 = (1,2) : int * int  
val t2 = (4,5) : int * int  
val t3 = (6,12) : int * int  
- val btree = (t1, (t2,t3));  
val btree = ((1,2),((4,5),(6,12)))  
: (int * int) * ((int * int) * (int * int))
```

Det ser jo oplagt ud at bruge “=” operatoren til at ændre på en værdi (hvilket den imperative programmør tror umiddelbart ville tro), men den udfører istedet sammenligning:

```
- btree = ((1,2),((4,5),(6,7)));  
val it = false : bool
```

og værdien kan ikke ændres på andre måder.

```
- val t3 = (6,7);  
val t3 = (6,7) : int * int  
- btree;  
val it = ((1,2),((4,5),(6,12)))  
: (int * int) * ((int * int) * (int * int))
```

Nye værdier kan konstrueres ud fra andre værdier, som det kan ses i eksemplet. Navne kan ændres og bindes til andre værdier, men en værdi er ikke-muterbar.

¹⁷Document Value Model

4.1.1 Oprettelse og deling af værdier

Oprettelse af værdier forgår i princippet som beskrevet i afsnit 4.1. Den måde at anvende værdier på siger ikke noget om hvordan den faktiske implementation er, men visse hvordan det fungerer rent konceptuelt. På grund af at en værdi aldrig *kan* ændre sig er der helt klart mulighed for at udnytte den egenskab med henblik på optimering. En optimering er at arbejde med “boxed” værdier. Dvs., at man i stedet for at arbejde direkte med værdier, arbejder med referencer til dem. Hvordan er det anderledes end f.eks. en pointer i C? Det er det på den måde at en reference er en funktion af en værdi, frem for en adresse på værdien. Referencer er sådan implementeret, at to forskellige værdier aldrig vil producere den samme reference, en bijektiv afbildning af værdier til referencer¹⁸.

Betragt SML eksempel koden i figur 5.

```
- val s = "world";  
val s = "world" : string  
- val s1 = "Hello";  
val s1 = "Hello" : string  
- val s2 = "Greetings";  
val s2 = "Greetings" : string  
- val t1 = (s1,s);  
val t1 = ("Hello","world") : string * string  
- val t2 = (s2,s);  
val t2 = ("Greetings","world") : string * string
```

Figur 5: SML eksempel 2

Antager man at man har en “boxed” repræsentation vil man have det repræsenteret som i figur 6, tabel 1 viser det mere formelt.

Bindinger		Boxed	
s	↦ vref1	vref1	↪ “world”
s1	↦ vref2	vref2	↪ “Hello”
s2	↦ vref3	vref3	↪ “Greetings”
t1	↦ vref4	vref4	↪ (vref2, vref1)
t2	↦ vref5	vref5	↪ (vref3, vref1)

Tabel 1: Boxed repræsentation

¹⁸Dette er selvfølgelig umuligt at garantere fuldstændigt med mindre at

$$\mathcal{D}(vref) \geq \mathcal{D}(v)$$

hvilket typisk ikke er tilfældet. I praksis er kollisioner dog meget sjældne.

I en “unboxed” repræsentation vil det se ud som i figur 7 og tabel 2

Bindinger	Unboxed
s ↦ val1	val1 = “world”
s1 ↦ val2	val2 = “Hello”
s2 ↦ val3	val3 = “Greetings”
t1 ↦ val4	val4 = (“Hello”, “world”)
t2 ↦ val5	val5 = (“Greetings”, “world”)

Tabel 2: Unboxed repræsentation

Denne egenskab gør det muligt at dele værdier på en effektiv måde uden at bryde med principperne i værdiorientering. Man kan så spørge, fungerer deling også på nyoprettede værdier, hvad hvis vi fortsatte eksemplet fra figur 5 med følgende linje:

```
- val t3 = (s1, "world");  
val t3 = ("Hello", "world") : string * string
```

Hvad vil der så ske? Ved oprettelse af en værdi vil der blive udregnet en reference til værdien, som så gemmes. Hvis der eksisterer en værdi for referencen er det unødvendigt at gemme, da den eksisterende værdi er identisk med den vi vil gemme jvf. bijektiviteten mellem værdier og referencer.

4.1.2 Celler

Celler er faktisk en imperativ struktur. En celle en opdaterbar værdi indeholdende en reference. Det er således muligt at opdatere hvilken reference den indeholder. Det er således stadig ikke muligt at ændre en værdi.

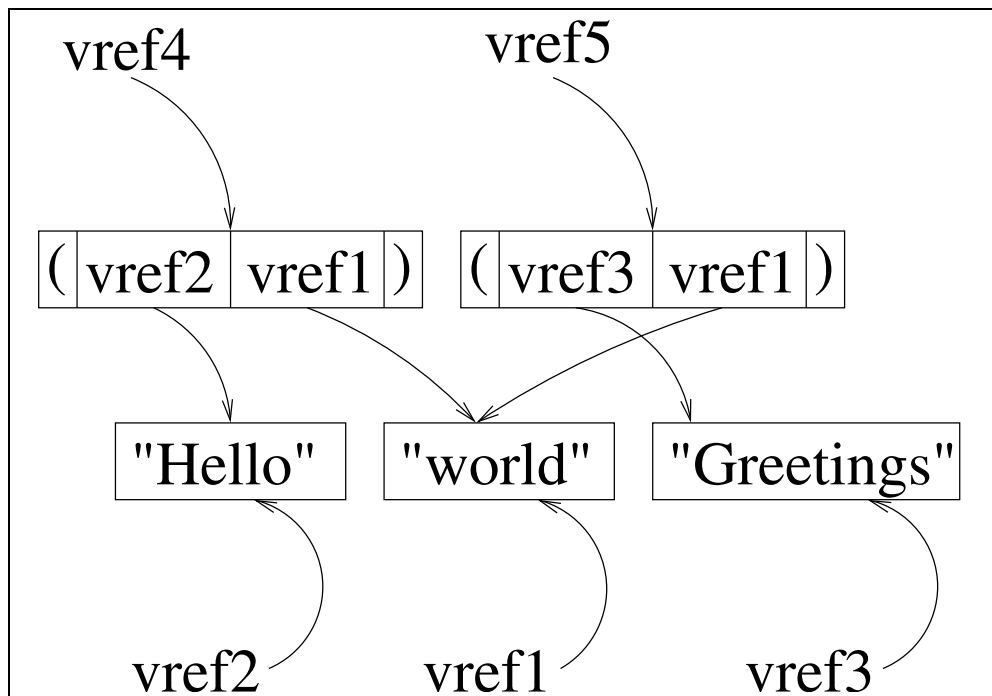
4.2 DVM

DVM er et træorienteret programmerings API med en værdiorienteret grænseflade, specielt designet med henblik på arbejde med XML-dokumenter[4][afsnit 4].

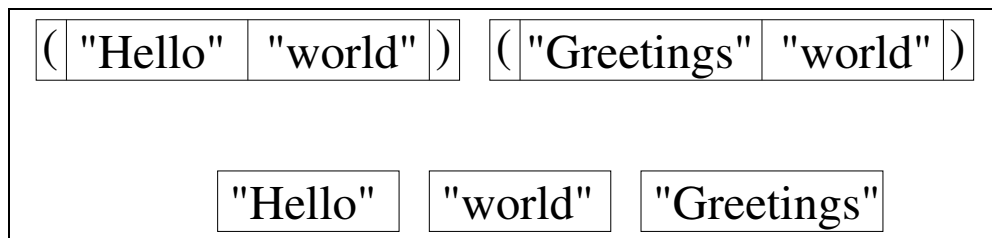
4.2.1 Tilgang til træer

API’et består af flere grænseflader, den vigtigste værende **Node** der repræsenterer dokumenter. Al tilgang til XML-dokumenter foregår via dette. Tilgang til attributer, data og anden information om en noder. “Oversat” til XML har en node den samme betydning som:

```
<tag attr1='val1' attr2='val2' ... attrn='valn'>  
... data og evt. mere XML ..  
</tag>
```



Figur 6: "Boxed" representation af eksemplet fra figur 5



Figur 7: "Unboxed" representation af eksemplet fra figur 5

4.2.2 Oprettelse og ændring af dokumenter

Det er selvfølgelig nødvendigt at have mulighed for at oprette dokumenter i DVM. Når det kommer til oprettelse, definerer DVM grænsefladen `XMLStoreFactory`. En implementation af DVM skal så implementere denne grænseflade, i vores tilfælde er det `XMLSTORE` der gør det. Via denne grænseflade er det så muligt at lave nye `Nodes`. Der er flere forskellige måder at oprette `Nodes` på, enten som en helt tom node eller med den data den skal indeholde givet på forhånd.

En anden vigtig funktion i grænsefladen som skal implementeres er `createXMLStore`, der returnerer en ny instans af et `Store`.

4.2.3 Persistens af dokumenter

Et `XMLSTORE` skal implementere funktionerne `load` og `save`. Disse bruges til at hente et dokument fra et `XMLSTORE`, henholdsvis gemme et dokument til et `XMLSTORE`. Når en nodes gemmes, returneres en `valuereference` til den. Når en node skal hentes ud fra et `XMLSTORE` bruges en `valuereference` som argument til `load` og en `Node` returneres. DVM har en doven politik mht. loading af børnenoder, dvs. at en node først hentes når den skal bruges. Det har visse fordele mht. til store dokumenter. Hvis man kun skal traversere en meget lille del af det samlede træ, er man fri for at hente det hele og hvis træet er meget stort er man fri for at have hele træet i hukommelsen.

4.2.4 Symbolske navne for værdier

I afsnit 4.1 så vi i hvordan SML binder en værdi til et navn. Den funktionalitet er også understøttet i `Dvm`. Et symbolsk navn er en streng, helst af en beskaffenhed der gør den forståelig for mennesker. Til at håndtere symbolske navne for `valuereferences` skal et `XMLSTORE` implementere en navne funktionalitet. Tre funktioner definerer denne funktionalitet.

bind binder en `valuereference` til et navn.

lookup Givet et navn, returneres den `valuereference` der er bundet til det navn. Hvis der ikke er bundet noget til det navn returneres ikke noget.

rebind Givent et navn og to `valuereferences`, den `valuereference` navnet er bundet til og den `valuereference` man vil ændre bindingen til. Hvis ikke den `valuereference` man giver som den "gamle" `valuereference` stemmer overens med den værdi den er bundet til i navne funktionaliteten (den er f.eks. opdateret af en anden peer), fejler operationen

4.2.5 DVM i distribuerede miljøer

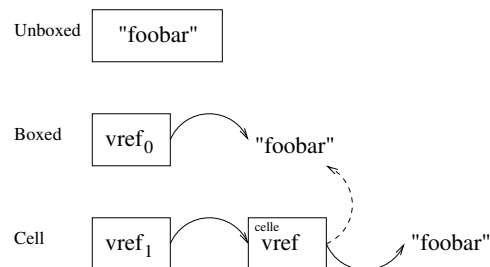
DVM er specielt designet til brug i distribuerede miljøer. Distribueret lager er oftest plaget af langsomme svartider. Modtrækket til dette er lokal caching.

Med caching følger problemer med “cache-coherence”, data i cache skal holdes opdateret i forhold til ændringer i den fil den cacher.

Distribueret lager også sårbart over for udfald af peers og for at undgå midlertidigt, eller i værste fald permanent, tab data af data replikeres det oftest mellem peers. Dette skaber ligeledes potentielle problemer med “data-coherence”, nemlig at data der er replikeret skal holdes opdateret i forhold til evt. ændringer i den logiske enkelte fil. At bruge imperative principper til distribuerede filsystemer vil derfor hurtigt resulterer i utrolig meget tid og computerkraft der bliver brugt på at sørge for at cachede og replikerede filer er konsistente.

I DVM er cache og data coherence aldrig et problem, da data er ikke-muterbart. En værdi kan på en sikker måde caches, da den reference der er til den aldrig vil komme til at referere til andet data. Ligeledes kan en værdi sikkert replikeres, den vil heller aldrig ændre sig. Med DVM kan man tage alle sine bekymringer mht. konsistens og smide dem væk, problemerne eksisterer slet ikke.

Der er dog visse steder hvor DVM træder lidt ved siden af sine egne principper, det er når det kommer til dens implementation af celler¹⁹. Celler er opdaterbare værdier, med visse begrænsninger. Begrænsningen er at værdien af en celle kun kan være en valuereference . Det er nærmest en dobbelt indirektion af en value, hvor det andet led kan ændres til at pege på andre værdier. Da det kun er en valuereference der skal opdateres, og disse er af en fast og relativt lille størrelse, ville man kunne implementere atomisk opdatering af celler relativt billigt.



Figur 8: Tre måder at gemme værdier på, Unboxed, boxed og i en celle. Bemærk, at den stiplede pil er hvad vref ville have peget på hvis alle repræsenterede værdier i det samme XMLSTORE. I det tilfælde ville $vref_0 = vref$

For en komplet beskrivelse af DVM, læs [4][afsnit 4].

¹⁹ironisk nok er celler p.t. ikke implementeret

4.3 XMLSTORE

XMLSTORE er PLAN-X's implementation af DVM. XMLSTORE arkitekturen er opdelt i flere komponenter

Reference service tager sig af afbilde referencer til værdier.

Navne server til at binde referencer til et navn.

Disks Lager for værdier.

4.3.1 Reference service

Reference servicen er en distribueret persistent hashtabel. Da en valuereference aldrig kan ændre hvad den refererer til, er der ingen opdateringer af i denne hashtabel. Når en værdi persisteres og der returneres en valuereference, gemmes der lokalt en afbildning af valuereference til en placering på en Disk (Disks bliver beskrevet i et lidt senere afsnit).

Når en applikation vil have værdien af en valuereference, slår man op i den lokale hash tabel først. Findes den der, returneres værdien. Er dette ikke tilfældet, spørges der ud på netværket om nogle har den afbildning i deres hash tabel. Er det tilfældet svarer en peer tilbage med værdien, der caches lokalt. Har ingen peers i netværket den afbildning, resulterer det i en time-out.

Denne noget simple måde at implementere en distribueret hashtabel fungerer, men har nogle punkter hvor forbedring er nødvendigt. F.eks. i et distribueret miljø vil en peer der bruger en værdi ofte altid skulle hente den fra en anden peer, når denne værdi ikke er tilstede i cache. Det kan resultere i unødigt meget trafik. Den eneste måde en værdi kan replikeres til en peer er hvis en peer ved et rent tilfælde gemmer en værdi identisk med der er på en anden peer.

4.3.2 Navne server

Navne serveren tager sig af afbildning af symbolske navne til valuereferences. Navneserveren er p.t. implementeret som en central server og er derfor det man kalder *a single point of failure*! Dette design skyldes den imperative natur **rebind()** har. I og med at rebind udfører den kontrol den gør, at se om der er lavet en anden binding siden man sidste lavede opslag på navnet, vil det betyde at den operation skulle kontrollere alle peers for en ny binding, hvis servicen var distribueret. Og det skulle selvfølgelig foregå atomisk. Det er tilsyneladende ikke en løsning man har ville binde an med, ikke engang på midlertidig basis.

4.3.3 Disks

En disk er det nederste lag i XMLSTORE arkitekturen. Det er her værdier i sidste ende bliver gemt til et persistent lager (typisk en harddisk). XML-

STORE definerer en grænseflade for Disks, sådan at ny funktionalitet og kan nemt kan implementeres. Det kan være alt fra caching til højt optimerede disks²⁰ specielt til brug for XMLSTORE.

Disks er ret simple af design, hvilket ses af at de kun per automatik tilbyder to funktioner: **save** og **load**. Disks understøtter sletning af filer²¹.

Disks implementerer deling af værdier, også på tværs af diske. Når en disk første gang gemmer en værdi, udregnes der en valuereference for den, der så gemmes i lokalt i reference servicen.

²⁰Et projektforslag om netop dette er at læse om på plan-x.org: Nativ XML Store disk implementation

²¹Det er et job for en distribueret spildopsamler. Da det i (praksis) er umuligt at vide om der et andet sted på netværket er en reference til en værdi i en disk, er det uforsvarlig at lade brugere af systemet slette værdier

5 Et værdiorienteret filsystem

I afsnit 4 har vi beskrevet principperne bag værdiorienteret programmering, beskrevet DVM og XMLSTORE. Ideen med denne opgave er at lave et filsystem, der bygger på de principper og udnytter de fordele det har over en traditionel imperativ model.

5.1 Krav til filsystemet

Filsystemet skal i så vidt det er muligt ligne et almindeligt UNIX filsystem og dermed være brugbart i de fleste “normale” situationer²². Målsætningen for det værdiorienterede filsystem er derudover at have følgende specielle træk:

Distribuering Data skal være distribueret ud på flere peers, og skal kunne tilgås på en uniform måde uafhængig af hvilken peer det er placeret.

Replikation Data er replikeret til flere steder på netværket, for dermed at imødekomme evt. tab af peers.

Værdiorienteret semantik At udnytte den semantik DVM tilbyder og udnytte den bedst muligt især til:

Persistens Data er persistent på tværs af sessions

Historik En hver version af filsystemets tilstand skal være tilgængelig. Dette vil muliggøre at filsystemet implementerer on-line backup og genskabelse af data.

Single copy semantik At implementere single-copy semantik for filsystemet, herunder mulighed for konfigurerbart håndtering opdateringskonflikter.

Dette er kravene til implementationen. Vi vil i de følgende afsnit se på hvordan filsystemet skal designes generelt og hvordan det ser ud når det skal modelleres vha. DVM og XMLSTORE. Derefter vil se på hvad der skal til for at denne implementation kan opfylde de ekstra krav vi har sat til det, samt hvordan XMLStore gør os i stand til det.

5.2 Generelt design af filsystemet

I afsnit 2 Den logiske opbygning af et UNIX filsystem er en træstruktur. DVM som vi har set i afsnit 4.2 er designet specielt til at arbejde med træstruktureret data, med tilhørende værktøjer og “byggeblokke”. Ved at udnytte DVM, il springet mellem den logiske og faktiske opbygning af filsystemet blive minimalt, og implementationen lettere.

²²Visse specielle funktionaliteter og features er ikke implementeret

5.2.1 Filer, biblioteker og symlinks

Med DVM er det nemt og oplagt at repræsentere og vedligeholde en træstruktur som den vi skal bruge til et filsystem.

I vores filsystem har vi valgt at have følgende objekter:

Filer Disse er noder, hvor filens indhold er `CHARDATA`.

Biblioteker De lavet som noder, hvor alle børnenoder er de filer, biblioteker eller symlinks det indeholder. I forhold til DVM er der lagt den begrænsning på børnenoderne, at der ikke må være to med samme navn, jvf. afsnit 2.1

Symlinks De er lavet som noder med en den sti den peger til gemt som `CHARDATA`.

Alle noder i filsystemet har en attribut der fortæller hvilken slags det, fil, bibliotek eller symlink.

Vi har valgt ikke at implementere:

Hard links Det har vi valgt fordi det er en utrolig imperativ struktur og fordi der ikke p.t. eksisterer en implementation af celler i `XMLSTORE`. Som nævnt i afsnit 2.6 kræver det at der holdes styr på hvor mange links der er til en fil. Skal det skulle implementeres uden celler ville man når man linkede til en fil ud over at lave et nyt træ der hvor man laver lænken fra, også skulle opdatere den node man lænker til og dermed skabe et til nyt træ. Det er ikke atomisk og vi fandt det ikke umagen værd at lave.

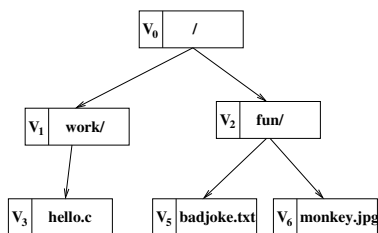
5.2.2 Opdateringer

I filsystemet vil den del af en fil der er data være gemt som en værdi i `XMLSTORE`. En værdi kan aldrig ændres og vil man ændre indholdet af en fil, må det nødvendigvis resultere i en ny værdi. I et traditionelt filsystem, er en ændring i en fil en ændring af indholdet af det lager som der er afsat til filen på et medie, evt. med tilføjelse af mere lager eller frigivelse alt afhængig om filens størrelse ændrer sig. I et værdiorienteret filsystem laves en helt ny fil. `XMLStore` er implementeret på en sådan måde at en værdi aldrig vil forsvinde, når den en gang er oprettet²³. `XMLStore` vil derfor altid indeholde en fils tidligere tilstand på et givent tidspunkt. Dette giver mulighed for at udvikle og integrere en historik funktion i filsystemet, hvilket vil blive omtalt senere.

Ved at bruge DVMs træer til at repræsentere filsystemer, kommer man ikke udenom effekterne af opdatering, nemlig det at en opdatering af en

²³Planer om at kunne slette værdier er vist nok i gang, men dette vil introducere problemer mht. til delte noder i `XMLStore`

enkelt node resulterer i et helt nyt træ. Opdatering af en node påvirker kun den direkte forældernode og dens forælder, indtil man når til rodnoden. Resultatet er en nye valuereference for rodnoden, for hver eneste opdatering. Antag at man ville tilføje en pointe til filen `/fun/badjoke.txt` fra filsystemet i figur 9.

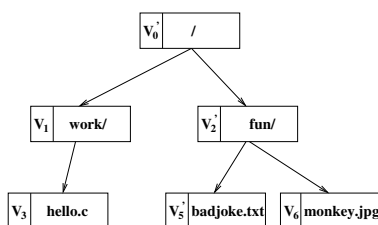


Figur 9: Et simpelt filsystem. Hver fil og bibliotek har en valuereference angivet som v_n

Det ville man gøre med et `write` systemkald. I FUSE grænsefladen vil det se sådan her ud.:

```
write("/fun/badjoke.txt", "Det sagde hun også igår", 24, 100)
```

Noden der repræsenterer filen bliver opdateret og filsystemet ser ud som i figur 10.



Figur 10: Ændring af `/fun/badjoke.txt` propagerer hele vejen til roden af filsystemet

Efter en opdatering er v'_0 den nye tilstand for filsystemet.

5.3 Distribuering

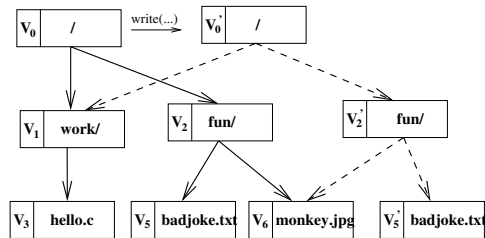
Et af kravene til filsystemet er at det er distribueret. Ved at bruge XML-STORE til at lave filsystemet i, er dette krav allerede adresseret, da det i forvejen understøtter distribuering. Det eneste krav for at få adgang til filsystemet er at man kan få kontakt til en anden peer der kører FUSEX.

5.4 Replikation

Et problem med at have et lager som XMLSTORE distribueret som et peer-to-peer netværk, er når en peer forsvinder. I det tilfælde vil det ikke være muligt at tilgå data lagret hos denne peer. For prøve at minimere dette problem er det ønskeligt at have en algoritme for replikation af data ud på flere forskellige peers. Dette er der p.t. kun begrænset understøttelse for i XMLSTORE, men er også et meget omfattende projekt, hvis det skulle laves. Vi antager at replikation ikke er noget vi skal tage os af så længe XMLSTORE bruges til at afbilde filsystemet.

5.5 Historik

Ser vi tilbage på eksemplet med opdatering af en fil fra afsnit 5.2.2, så er den faktiske tilstand for XMLSTORE som det ses i figur 11 (hvis vi antager at det kun indeholdt de filer og biblioteker man kan se i figur 9 til at starte med²⁴).



Figur 11: Opdatering af badjoke.txt propagerer til toppen af filsystemet, resulterende i en ny værdi for hele filsystemet. Noder der ikke er “på vejen” bliver delt med det “nye” filsystem

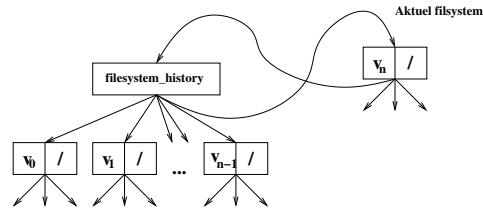
Som det ses af figuren, kan man ved ved at betragte v_0 som roden af filsystemet i stedet for den nye v_0' , se filsystemets tilstand som det var før vi ændrede på det.

Ved at have en datastruktur der holder styr på filsystemets tilstand over tid, har man muligheden for at gå tilbage i tiden og se hvordan filsystemet så ud med en tidligere rod og på den måde have en historik over filsystemets tilstand.

Tilgangen til historikken vil være via et specielt directory kaldet `history` der vil ligge i roden af filsystemet ²⁵. Via dette specielle bibliotek vil man få “read-only” adgang tidligere versioner af filsystemet, se figur 12.

²⁴For sjov kunne man antage at de andre filer der evt. var der, var blevet garbage collected

²⁵Lige som ext2 har sit `filesystem_history`



Figur 12: Adgang til gamle versioner af filsystemet sker via et specielt bibliotek i roden af filsystemet

5.6 Semantik for filkonsistens

At have en “single copy” semantik for et filsystem er noget der bliver interessant så snart en fil bliver tilgået af to eller flere forskellige processer/brugere samtidigt. Single copy semantik er konceptet om at der kun findes en kopi af en fil og at alle arbejder på den²⁶. Således vil en hver ændring foretaget af en bruger på en fil, være synlig for andre der har den åben.

Om end det ikke bliver kaldt single-copy semantik, så beskriver Silberschatz et al.[5] UNIX semantik således:

- Skrivninger til en åben fil er med det samme synlige for andre brugere der har denne fil åben på samme tid.
- ...

I UNIX semantik er en fil tilknyttet med en enkelt fysisk afbildning, der bliver tilgået som en eksklusiv ressource. Er der strid om denne resulterer den i at processer venter (på at den bliver fri).

Den måske nemmeste måde at forklare single copy semantik på, er ved at vise hvordan det kan gå galt. Problemet når et filsystem ikke implementerer single copy semantik er såkaldte “lost updates”. Problemet opstår når en opdatering af en fil “overskygger en anden”, oftest som resultat af en optimistisk caching-politik.

Det er klart at ikke er single copy semantik, men det er værd at overveje hvad implikationerne af at håndhæve den semantik er. Det kræver for det første at der er mere låsning omkring de operationer der modificerer en fil.

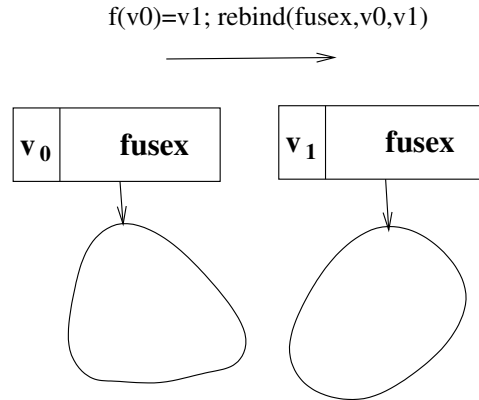
For det andet, hvad skal der så ske i en situation som den i eksemplet? Skal filsystemet returnere en fejl, fordi den ikke er up-to-date? Skal den lave ændringen i den, hvis det ellers ikke konflikter med den tidligere opdatering?

5.6.1 Opdateringspolitik

Vi hvilke valgmuligheder har af opdateringspolitikker? FUSEX bruger XML-STORES navneserver til lave den ene, opdaterbare, celle vi har brug for.

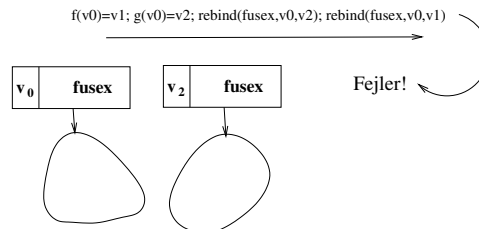
²⁶Hvilket i øvrigt gør det noget forvirrende at snakke om en kopi

Navneserveren kan kun binde et eksisterende navn til en ny værdi hvis man kender den gamle, derfor kan vi bruge den som mutex. Figure 13 viser hvordan en enkle proces “f” laver en ændring i filsystemet og succesfuldt genbinder “fusex” til den nye værdi. Hvis der er mere end en proces kan det gå galt på



Figur 13: En enkel proces “f” opdatere filsystemet

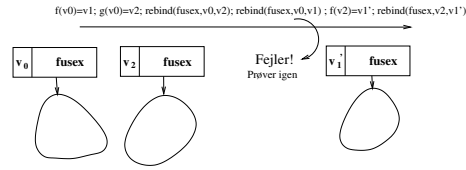
et tidspunkt. Figure 14 viser hvad der sker hvis proces “g” genbinder “fusex” mens “f” også er i gang med at ændre filsystem. Den strengest politik er som hvis at lade “f”’s kald fejle. Det er denne politik FUSEX benytter som standart.



Figur 14: To processer “f” og “g” opdatere filsystemet - striks

Alternativt kan man prøve at genudføre “f” på den nye filsystems tilstand og se om det går - se figur15.

Tilslut kan vi også overveje hvordan man kunne bruge historik-information til at lave smarte sammenfletnings regler, baseret enten på opdateringens alder eller de mellemliggende opdateringers indhold. Men det er helt anden opgave, så vi holder os i denne omgang til den strikse, altid korrekt for single-copy-semantik.



Figur 15: To processer “f” og “g” opdatere filsystemet - prøver igen

5.7 Semantik for FUSEX

Dette afsnit vil beskrive den semantik FUSEX implementerer. Vi tager udgangspunkt i det lidt simple interface som FUSE / FUSE-J definerer istedet for VFS', da det nok vil være lidt for omfattende og desuden laver FUSE ting i VFS som vi ikke har kontrol over.

5.7.1 Inferenssystem for FUSEX

Vi vil her præsentere et inferenssystem for FUSEX.

FS	\in	$FileSystems$
E	\in	$DirEntries$
D	\in	$Directories$
F	\in	$Files$
N	\in	$Names$
P	\in	$Paths$
L	\in	$Symlinks$
M	\in	$Metadata$
$FileSystem$	$::=$	$Directory$
$DirEntry$	$::=$	$File Directory Symlink$
$Directory$	$::=$	$(Name * DirEntry) list$
$File$	$::=$	$Data$
$Symlink$	$::=$	$Path$
$Name$	$::=$	$String$
$Path$	$::=$	$String list$

Figur 16: Abstrakt beskrivelse af FUSEX filsystem

Der skal knyttes et par kommentarer til nogle beskrivelsen af filsystemet i figur 16. I beskrivelsen optræder *Metadata*. *Metadata* er information om en indgang i filsystemet (en fils størrelse, hvornår den sidst blev ændret rettigheder, ejerskab m.m.). For at begrænse mængden af semantiske regler, af hvilke der er rigeligt af i forvejen, er *Metadata* ikke medtaget som et objekt

i filsystemet. Man skal dog være klar over at den er der. Alle operationer der resulterer i en ændring i den del af af objekt der ikke er metadata, ændrer også i metadata. Det er kun få operationer hvor det modsatte er tilfældet. Man skal således være opmærksom at inferenssystemet på dette punkt kan se udeeterministisk ud.

Hjælpfunktioner For at undgå unødvendigt mange regler på grund af deres rekursive virkemåde, defineres *lookup* og *bind*. Disse er ikke en del af det interfaces der skal implementeres, men repræsenterer to operationer der ofte bliver brugt i programmet²⁷

bind

$$\frac{P = [N], D' = D \cup (N \mapsto E)}{\langle D, \text{bind}(P, E) \rangle \rightarrow \langle D' \rangle}$$

$$\frac{P = [N_0, N_1, \dots, N_{n-1}, N_n], D = [\dots, (N_0 \mapsto D'), \dots], \langle D', \text{bind}(P \setminus N_0) \rangle \rightarrow \langle D'' \rangle, D''' = [\dots, (N_0 \mapsto D''), \dots], (n > 1)}{\langle D, \text{bind}(P, E) \rangle \rightarrow \langle D''' \rangle}$$

lookup

$$\frac{P = [N], D = [\dots, (N \mapsto E), \dots] \rightarrow \langle E \rangle}{\langle D, \text{lookup}(P) \rangle \rightarrow \langle E \rangle}$$

$$\frac{P = [N_0, N_1, \dots], D = [\dots, (N_0 \mapsto D'), \dots], \langle D', \text{lookup}(P \setminus N_0) \rangle \rightarrow \langle E \rangle}{\langle D, \text{lookup}(P) \rangle \rightarrow \langle E \rangle}$$

Semantiske regler Vi definerer her de semantiske regler for FUSEX:

getattr Denne funktion er speciel i og med den ikke returnerer en fil, men dens metadata. *M* er *E*'s metadata. Fungerer som **stat**

$$\frac{\langle D, \text{lookup}(P) \rangle \rightarrow \langle E \rangle, E \rightsquigarrow M}{\langle D, \text{getattr}(P) \rangle \rightarrow \langle M \rangle}$$

readlink

$$\frac{\langle D, \text{lookup}(P) \rangle \rightarrow \langle L \rangle}{\langle D, \text{readlink}(P) \rangle \rightarrow \langle L \rangle}$$

²⁷**bind** svarer lidt til det **link** ville gøre, men eftersom **link** bevidst ikke er implementeret i FUSEX er det udeladt for ikke at skabe forvirring.

getdir

$$\frac{\langle D, \text{lookup}(P) \rangle \rightarrow \langle D' \rangle, D' = [(N_0, E_0), (N_1, E_1), \dots, (N_{n-1}, E_{n-1}), (N_n, E_n)]}{\langle D, \text{getdir}(P) \rangle \rightarrow \langle [N_0, N_1, \dots, N_{n-1}, N_n] \rangle}$$

mkdir

$$\frac{P = [N], D = [\dots], D' = [], D'' = D \cup (N \mapsto D')}{\langle D, \text{mkdir}(P) \rangle \rightarrow \langle D'' \rangle}$$

$$\frac{P = [N_0, N_1, \dots, N_{n-1}, N_n], \langle \text{lookup}(P \setminus N_n), D \rangle \rightarrow \langle D' \rangle, \langle \text{mkdir}(N_n), D' \rangle \rightarrow \langle D'' \rangle, \langle D, \text{bind}(P, D'') \rangle \rightarrow \langle D''' \rangle, (n > 1)}{\langle D, \text{mkdir}(P) \rangle \rightarrow \langle D''' \rangle}$$

unlink

$$\frac{P = [N], D = [\dots, (N \mapsto E), \dots], E \notin \text{Directories}, D' = D \setminus (N \mapsto E)}{\langle D, \text{unlink}(P) \rangle \rightarrow \langle D' \rangle}$$

$$\frac{P = [N_0, N_1, \dots, N_{n-1}, N_n], \langle D, \text{lookup}(P \setminus N_n) \rangle \rightarrow \langle D' \rangle, \langle D', \text{unlink}(N_n) \rangle \rightarrow \langle D'' \rangle, \langle \text{bind}(P \setminus N_n, D''), D \rangle \rightarrow \langle D''' \rangle, (n > 1)}{\langle D, \text{unlink}(P) \rangle \rightarrow \langle D''' \rangle}$$

rmdir

$$\frac{P = [N], D = [\dots, (N \mapsto D'), \dots], D' = [], D'' = D \setminus (N \mapsto D')}{\langle D, \text{rmdir}(P) \rangle \rightarrow \langle D'' \rangle}$$

$$\frac{P = [N_0, N_1, \dots, N_{n-1}, N_n], \langle D, \text{lookup}(P \setminus N_n) \rangle \rightarrow \langle D' \rangle, \langle D', \text{rmdir}(N_n) \rangle \rightarrow \langle D'' \rangle, \langle D, \text{bind}(P \setminus N_n, D'') \rangle \rightarrow \langle D''' \rangle, (n > 1)}{\langle D, \text{rmdir}(P) \rangle \rightarrow \langle D''' \rangle}$$

symlink

$$\frac{P = [N], L = P', D' = D \cup (N \mapsto L)}{\langle D, \text{symlink}(P, P') \rangle \rightarrow \langle D' \rangle}$$

$$\frac{P = [N_0, N_1, \dots, N_{n-1}, N_n], \langle D, \text{lookup}(P \setminus N_n) \rangle \rightarrow \langle D' \rangle, \langle D', \text{symlink}(N_n, P') \rangle \rightarrow \langle D'' \rangle, \langle D, \text{bind}(P \setminus N_n, D'') \rangle \rightarrow \langle D''' \rangle, (n > 1)}{\langle D, \text{symlink}(P, P') \rangle \rightarrow \langle D''' \rangle}$$

rename

$$\frac{\langle D, lookup(P) \rangle \rightarrow \langle E \rangle, \langle D, unlink(P) \rangle \rightarrow \langle D' \rangle, \langle D', bind(P', E) \rangle \rightarrow \langle D'' \rangle}{\langle D, rename(P, P') \rangle \rightarrow \langle D'' \rangle}$$

write

$$\frac{\langle D, lookup(P) \rangle \rightarrow \langle F \rangle, F \rightarrow F', \langle D, bind(P, F) \rangle \rightarrow \langle D'' \rangle}{\langle D, write(P, data, offset) \rangle \rightarrow \langle D' \rangle}$$

read Filen opdateres ikke, heller ikke dens metadata²⁸

$$\frac{\langle D, lookup(P) \rangle \rightarrow \langle F \rangle, F \rightarrow data}{\langle D, read(P, offset) \rangle \rightarrow \langle data \rangle}$$

²⁸I LINUX er den semantik konfigurerbar, og bliver ofte brugt hvor I/O ønskes reduceret mest muligt

6 Programmeringsovervejelser

Dette afsnit vil beskæftige sig med de specifikke og interessante overvejelser der var omkring selve implementationen af FUSEX. Efter at vi gennemanalyseret vores problemstilling og valgt vores værktøjer - FUSE-J - var selve implementeringen trivielt programmeringsarbejde. Vi mener ikke vi laver nogle speciel knep der kræver ekstra ord med på vejen - koden (se bilag B) burde give sig selv sammen med kommentarerne.

6.1 Kodeoversigt

Her lille oversigt over kildekode, og hvad de enkelte dele er: XMLStoreFilesystem.java (bilag B.1) er selve FUSEX-programme, der implementere FUSE-J's FileSystem-klasse. XMLStorePeer.java (bilag B.2) er det lille program der starter vores "DistributedXMLStore"-noder. Da er skal ret mange java parametre til at starte de to programmer har vi lavet to små scripts til at starte vores programmer (bilag : B.3 og B.4).

6.2 Status på FUSEX

Undervejs i udviklingen løb vi ind flere forhentninger i vores omkringliggende miljø. Nogle fik vi selv løst, og andre blev rettet af ophavsmændene. Men helt til slut var der stadig ting vi ikke kunne nå at få i orden.

- "DistributedXMLStore" kunne ikke gemme noder der oversteg en bestemt størrelse (omkring 512-1024 bytes).
- Det netværkstemiljø vi havde til rådighed havde nogle problemer med UDP-trafik.

Begge punkter er ekstreme faktore på selve FUSEX programmet, og skulle kunne sættes direkte ind når de er blevet låst uden at ændre på FUSEX. Når det så sagt er der stadig massere af optimereing tilbage man kunne tage fat på i FUSEX. Nu når vi har set at konceptet *kan* laves, gælder det om at gøre overheadet at lille så muligt. Vi kan spare en del ved at fjerne nogle af lagene. Hvis vi gik over til at bruge XMLSTORES socket-interface kunne vi fx. spare FUSE-J-laget væk.

7 Brugervejledning

Vi vil nu beskrive brugen af FUSEX. Når det først er installeret og køre, vil det fungere som et almindelig filsystem, med nogle få begrænsninger, men også nogle ekstra finesser.

Du kan oprette biblioteker, filer og symbolske links. Dem kan du skrive, læse, slette, omdøbe, Ydermere har FuseX historik over filsystems tidligere tilstand. I biblioteket “history” har du read-only adgang til disse.

Som nævnt i programmeringsovervejelserne, kan du IKKE oprette hårde links eller bruge “access-time”.

7.1 Installation

For at bruge FUSEX skal du have følgende installeret på din maskine i forvejen: Linux, Fuse-1.1, Fuse-J og plan-x’s XMLSTORE. Den vejledning forudsætter at du har cvs-adgang til plan-x.org.

7.1.1 FUSE-1.1

Dette installere FUSE kerne-modulet.

```
# cd <path-to-plan-x>/projects/fusex
# tar xzf fuse-1.1.tar.gz
# cd fuse-1.1
# ./configure
# sudo make
# sudo make install
```

7.1.2 FUSE-J

Dette er java-bindingerne til FUSE. Der er en fejl med pakken, så husk at patche det med vores patch.

```
# cd <path-to-plan-x>/projects/fusex
# tar xzf fuse-j.tar.gz
# cd fuse-j
# patch jni/javafs.c ../javafs.c.patch
# <edit build.conf to match your java installation>
# make
```

7.1.3 XMLSTORE

```
# cd <path-to-plan-x>/xmlstores/bogebjerg/store
# ./make
```

7.1.4 FUSEX

```
# cd <path-to-plan-x>/projects/fusex
# make
# sudo mkdir /var/planx /mnt/fuse
# sudo chown <username>.<usergroup> /var/planx /mnt/fuse
# <add "/proc/fs/fuse/dev /mnt/fuse fuse noauto,user 0 0" to /etc/fstab>
# <edit "xmlstorepeer" and "mount.fusex" to match your system configuration>
```

FUSEX burde nu være installeret.

7.2 Brug

Du kan bruge FUSEX alene eller på en eller flere distribuerede XMLSTORES peers.

7.2.1 Lokalt

```
# <path-to-plan-x>/projects/fusex/mount.fusex /mnt/fuse
```

7.2.2 P2P

Start en eller flere xmlstore-peers. Du skal vælge portnumre der ledige både som UDP og TCP. Den første peer du starter skal ikke angive nogle boot-host, men de efterfølgende skal boot på en af de allerede kørende knuder.

```
# <path-to-plan-x>/projects/fusex/xmlstorepeer port-nummer \  
> [boot-host:boot-port]
```

Og så kan du tilkoble FUSEX

```
# <path-to-plan-x>/projects/fusex/mount.fusex /mnt/fuse \  
> port-nummer boot-host:boot-port
```

Du er nu klar til at bruge dit filsystem. Historiken findes i `history`-folderen.

7.2.3 Afkobling

Når du er færdig kan du afkoble det på selvvaneligvis.

```
# umount /mnt/fuse
```

7.2.4 Formattering af FUSEX-disk

For at starte med en blank disk skal du blot slette indholdet af `/var/planx/`

8 Afprøvning og test

Det var vores mål at teste FUSEX på flere XMLSTORE peers op imod NFS. Desværre spillede flere ydre faktorer i mod. Vi kunne ikke få adgang til et fungerende testmiljø hvor vi måtte installere kernemoduler. Endvidere viste det sig at DISTRIBUTEDXMLSTORE fejlede når man prøvede at gemme en data-knude knude med noget mere end 512-1024 bytes. Men funktionerne er ellers testet virker korrekt et p2p-XMLSTORE.

8.1 Afprøvning

Teststrategien at teste alle vores funktionerne i FUSEX gennem almindelige filsystems operationer. Vi testede funktionerne med et simpelt bash-script og en blank FUSEX-disk. Vi forudsætter at læser kan nok til unix kommander til at kunne læse dette script:

Test script af filesystems funktioner

```
#!/bin/bash
cd /mnt/fuse/
ls history
mkdir foo
ln -s foo bar
echo "vigtig" > foo/file.txt
cat bar/file.txt
echo "junk" > foo/file.txt
cat foo/file.txt
cat history/4*/foo/file.txt
mkdir delme
touch delme/1
touch delme/2
rm delme/2
rm -rf delme
touch baz
chown root.root baz
ls -ls
ls history
```

Resultater

```
0 - [00:32:11 19-05-2004] - created_new_disk
vigtig
junk
vigtig
total 2
```

```

    1 lrwxr-xr-x    1 bob      users      0 May 19 00:32 bar -> foo
    1 -rw-r--r--    1 root     root       0 May 19 00:32 baz
    1 drwxr-xr-x    1 bob      users      0 May 19 00:32 foo
    1 drwxr-xr-x    1 bob      users      0 May 19 00:32 history
0 - [00:32:11 19-05-2004] - created_new_disk
1 - [00:32:14 19-05-2004] - mkdir(_foo,493)
10 - [00:32:15 19-05-2004] - mknod(_delme_2,33188,0)
11 - [00:32:15 19-05-2004] - utime(_delme_2,1084919535,1084919535)
12 - [00:32:15 19-05-2004] - unlink(_delme_2)
13 - [00:32:15 19-05-2004] - unlink(_delme_1)
14 - [00:32:15 19-05-2004] - rmdir(_delme)
15 - [00:32:15 19-05-2004] - mknod(_baz,33188,0)
16 - [00:32:15 19-05-2004] - utime(_baz,1084919535,1084919535)
17 - [00:32:15 19-05-2004] - chown(_baz,0,0)
2 - [00:32:14 19-05-2004] - symlink(foo,_bar)
3 - [00:32:14 19-05-2004] - mknod(_foo_file.txt,33188,0)
4 - [00:32:14 19-05-2004] - write(_foo_file.txt,7,0)
5 - [00:32:14 19-05-2004] - truncate(_foo_file.txt,0)
6 - [00:32:14 19-05-2004] - write(_foo_file.txt,5,0)
7 - [00:32:15 19-05-2004] - mkdir(_delme,493)
8 - [00:32:15 19-05-2004] - mknod(_delme_1,33188,0)
9 - [00:32:15 19-05-2004] - utime(_delme_1,1084919535,1084919535)

```

Resultatet blev som vi forventede.

8.2 Ydelse

Vi ønsker også at måle hvor hurtigt FUSEX er i forhold til eksisterende filsystemer. Vi valgte at teste ext3, reiserfs og nfs udover FUSEX. Det viste sig hurtigt at den FUSEX i sin nuværende form var *langsomere* end modstanderne. Så vi valgte at skrive vores eget mini-performance-testprogram, hvor at køreslerne ikke skulle tage dagevis på FUSEX.

Test-script

```

#!/bin/env python
import time,sys,os
if len(sys.argv)<2:
    print "USAGE: test_performance.py <dir-to-test>"
    sys.exit(1)

print "Performance test...",sys.argv[1]
start = time.time()
for x in xrange(8):
    os.system("dd if=/dev/urandom of=%s/test count=128"%sys.argv[1])

```

```

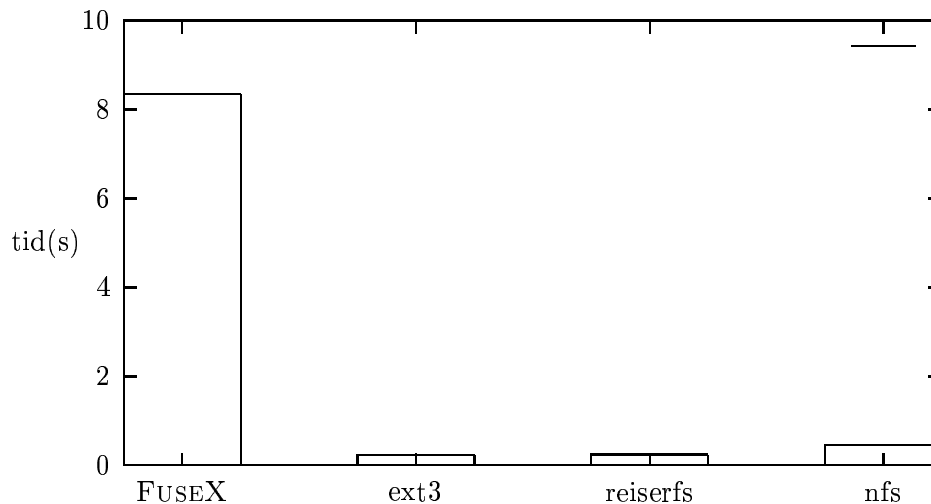
    os.system("cat %s/test > /dev/null"%sys.argv[1])
    os.system("rm %s/test"%sys.argv[1])
end = time.time()

print "Run time :", end-start

```

Resultater

Vi kørte FUSEX lokalt og uden debug-logging. Resultater blev som følger:



Filsystem	tid(s)
FUSEX	8.33837795258
ext3	0.231198072433
reiserfs	0.24388217926
NFS	0.463504076004

Tabel 3: Performance test resultater

FUSEX er 36 til 18 gange langsommere end konkurrenterne i denne simple test. Men den arbejder også igennem langt flere lag end konkurrenterne. Men det ville være interessant at se hvor dan FUSEX skalere i et P2P-miljø. Men det må blive i næste version.

Da FUSE udføre sine kommandore atomisk og da det ikke lykkes at få testet P2P-delen af FUSEX på andet en localhost har vi ikke i praksis dået testet om vores single-copy-semantik fungere korrekt, men det er selvfølgelig vores overbesvisning at det vil virke korrekt.

A Projektforeslag (på engelsk)

Unix file system interface for XML Store

This project is aimed at providing a Unix/Linux file system interface for mounting local or remote XML Stores. This makes it possible to provide a Unix file system interface to a distributed, replicated, mobile XML Store and provides backwards compatibility for applications operating on the Unix file I/O. Special emphasis should be placed on correctness, well-specified/predictable semantics and implementation efficiency, incl. comparison with NFS (both in terms of semantics and efficiency). It is suggested that the VFS (virtual file system) abstraction for implementing file systems in Linux is used.

Fritz Henglein

B Kildekode

B.1 XMLStoreFilesystem.java

```
/*
 * FUSEX:
 *
 * XMLStoreFilesystem
 *
 * Copyright (C) 2004 Christian Hemmingsen (kewl@diku.dk)
 * Bo Bendtsen (bobend@diku.dk)
 */

10 import fuse.*;
import fuse.util.Log;

import org.planx.xmlstore.*;
import org.planx.xmlstore.stores.*;
15 import org.planx.xmlstore.nodes.*;

import org.planx.xmlstore.input.*;
import org.planx.xmlstore.nameserver.*;

20 import java.net.*;
import java.io.*;
import java.util.*;
import java.nio.ByteBuffer;

25 import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;

public class XMLStoreFilesystem implements Filesystem {
30
    //
    // Private fields
    //
    private static final Log log = new Log(XMLStoreFilesystem.class);
35
    private XMLStore xmlstore;
    private NameServer ns;

    private Node root;
    private Node rootDir;
40 private ValueReference rootVref;
    private int rootState = -1;
    private boolean historyMode = false;
    private String rootLog;

45 private static final int ATTRS = 8;

    //
    // Customizable..
    //
50 private static final DateFormat DATEFORMAT =
    new SimpleDateFormat("HH:mm:ss_ddd-MM-yyyy");
    private static final String FUSEX_ROOT = "fusex";
    private static final int HISTORY = 100;
55 private static final String HISTORY_FOLDER = "history";
```

```

//
// Constructor
60 //
public XMLStoreFilesystem (XMLStore _xmlstore, NameServer _ns)
    throws FuseException {
    // creates the XMLStore filesystem interface...
    try {
65     xmlstore = _xmlstore;
        ns = _ns;
        rootVref = ns.lookup(FUSEX_ROOT);

        if (rootVref == null) {
70             // no fusexroot exists in the xmlstore,
            // creates a new fusex "disk"...
            log.info("creating_new_fusex_disk");
            rootLog = "created_new_disk";
            rootDir = NodeFactory.createElementNode
75             ("dir", XFCreateAttrs(XFNextRootName(), 493)); //493=rwxr-xr-x
            root = NodeFactory.createElementNode("fusex", rootDir);
            rootVref = xmlstore.save(root);

            try {
80                 ns.bind("fusex", rootVref);
            } catch (NameServerAlreadyBoundException e) {
                // another root has been created since the lookup,
                // use that instead...
                log.warn(e.toString());
85                 log.info("loading_exsiting_fusex_disk");
                rootVref = ns.lookup(FUSEX_ROOT);
                root = xmlstore.load(rootVref);
            }
        } else {
90             // loads an exsisting fusex "disk"...
            log.info("loading_exsiting_fusex_disk");
            root = xmlstore.load(rootVref);
        }
        rootDir = root.getChildNodes().getNode(0);
95     } catch (XMLStoreIOException e) {
        throw new FuseException(e.toString()).initErrno(FuseException.EIO);
    } catch (UnknownValueReferenceException e) {
        throw new FuseException(e.toString()).initErrno(FuseException.EIO);
    }
100 }

//
// Private metodes
//
105 private Attribute[] XFCreateAttrs (String name, int mode) {
    // create attributes...
    Attribute[] attrs = new Attribute[ATTRS];
    attrs[0] = NodeFactory.createAttribute("name", name);
    attrs[1] = NodeFactory.createAttribute("mode", Integer.toString(mode));
110     attrs[2] = NodeFactory.createAttribute("uid", "1000");
    attrs[3] = NodeFactory.createAttribute("gid", "100");
    attrs[4] = NodeFactory.createAttribute
        ("ctime", Integer.toString(((int)(System.currentTimeMillis() / 1000L)));
115     attrs[5] = NodeFactory.createAttribute
        ("mtime", Integer.toString(((int)(System.currentTimeMillis() / 1000L)));
    attrs[6] = NodeFactory.createAttribute("atime", "0"); //unsupported! NB!
    attrs[7] = NodeFactory.createAttribute("size", "0");

    return attrs;
}

```

```

120     }

private Attribute[] XFmtimeAttrs (Attribute[] old_attrs) throws FuseException {
    // update the mtime attribute to the current system time...
125     return XFModifyAttrs (old_attrs, "mtime", Integer.toString
        ((int)(System.currentTimeMillis() / 1000L));
    }

private Attribute[] XFModifyAttrs (Attribute[] old_attrs, String key,
130     String value) throws FuseException {
    // modify the <key> attribute to <vaule>...
    Attribute[] attrs = new Attribute[ATTRS];
    boolean found = false;
    for (int x=0; x<ATTRS; x++) {
135         if (!found && old_attrs[x].getName().equals(key)) {
            attrs[x] = NodeFactory.createAttribute(key, value);
            found = true;
        } else {
140             attrs[x] = old_attrs[x];
        }
    }
    if (!found)
        throw new FuseException("Unknow_attribute!").initErrno
145         (FuseException.EPERM);
    else
        return attrs;
    }

private String XFNextRootName() {
150     String date;

    synchronized (DATEFORMAT) {
        date = DATEFORMAT.format(new Date());
    }
155     return (++rootState) + "_-_" + date + "_-_" + rootLog.replace('/', '_'); // NB!
    }

private void XFGetCurrentRoot () throws FuseException {
160     // load the current root from the xmlstore...
    try {
        rootVref = ns.lookup(FUSEX_ROOT);
        root = xmlstore.load(rootVref);
165         rootDir = root.getChildNodes().getNode(0);
        String rootName = rootDir.getAttribute("name");
        rootState = Integer.parseInt(rootName.substring
            (0, rootName.indexOf("_-_")));
    } catch (XMLStoreIOException e) {
170         throw new FuseException(e.toString()).initErrno(FuseException.EIO);
    } catch (UnknownValueReferenceException e) {
        throw new FuseException(e.toString()).initErrno(FuseException.EIO);
    }
    }

175 private void XFSaveRoot () throws FuseException {
    // save the node in "root" too xmlstore as the new root...
    if (historyMode)
180         throw new FuseException("Can't write to the history folder").initErrno
            (FuseException.EROFS);
}

```

```

try {
    // adds the new historik...
    rootDir = NodeFactory.createElementNode
185     ("dir", rootDir.getChildNodes(),
        XFModifyAttrs(rootDir.getAttributes(), "name", XFNextRootName()));
    if (root.getChildNodes().getLength() == HISTORY) {
        root = DVMUtil.insertChild(DVMUtil.removeChild(root, HISTORY-1),
190         0, rootDir);
    } else {
        root = DVMUtil.insertChild(root, 0, rootDir);
    }

    ValueReference newRootVref = xmlstore.save(root);
195     ns.rebind(FUSEX_ROOT, rootVref, newRootVref);

    rootVref = newRootVref;

200     } catch (XMLStoreIOException e) {
        throw new FuseException(e.toString()).initErrno(FuseException.EIO);
    } catch (NameServerStaleOldReferenceException e) {
        // or should we try to merge here?
        throw new FuseException("STALE"+e.toString()).initErrno
205         (FuseException.EAGAIN);
    }
}

private FuseStat XFGetStat(Node node) throws FuseException {
210     // read the file metadata to a FuseStat structure...
    FuseStat stat = new FuseStat();
    String type = node.getNodeValue();

    if (type.equals("dir")) {
215         stat.mode = FuseFtype.TYPE_DIR | Integer.parseInt
            (node.getAttribute("mode"));
    } else if (type.equals("file")) {
        stat.mode = FuseFtype.TYPE_FILE | Integer.parseInt
            (node.getAttribute("mode"));
220     } else if (type.equals("symlink")) {
        stat.mode = FuseFtype.TYPE_SYMLINK | Integer.parseInt
            (node.getAttribute("mode"));
    } else {
225         throw new FuseException("Unsupported_type").initErrno
            (FuseException.EPERM);
    }

    stat.nlink = 1;
    stat.uid = Integer.parseInt(node.getAttribute("uid"));
230     stat.gid = Integer.parseInt(node.getAttribute("gid"));
    stat.size = Integer.parseInt(node.getAttribute("size"));
    stat.atime = Integer.parseInt(node.getAttribute("atime"));
    stat.mtime = Integer.parseInt(node.getAttribute("mtime"));
    stat.ctime = Integer.parseInt(node.getAttribute("size"));
235     stat.blocks = 1;
    return stat;
}

private void XFUpdateTree(Node leaf, LinkedList nodeList) throws FuseException {
240     // swap in the <leaf> into the <nodeList>, which is a path in the tree.
    // Returns a new root...

    // travel from the leaf to the root...

```

```

245     for (int x = nodeList.size() - 2; x >= 0; x--) {
        Node parrent = (Node) nodeList.get(x);
        ChildNodes siblings = parrent.getChildNodes();
        String name = leaf.getAttribute("name");
        Node[] newSiblings = new Node[siblings.getLength()];
        // find and update the affected node...
250     for (int y=0; y<siblings.getLength(); y++) {
            if (siblings.getNode(y).getAttribute("name").equals(name)) {
                newSiblings[y] = leaf;
            } else {
                newSiblings[y] = siblings.getNode(y);
255     }
        }
        leaf = NodeFactory.createElementNode
            ("dir", NodeFactory.createChildNodes(newSiblings),
            parrent.getAttributes());
260     //should mtime be updated in all the nodes?
    }
    rootDir = leaf; // the new root of nodeList
}

265 private void XFCreateNewElement (String path, String type, int mode, String data)
    throws FuseException {
    // create a new entry at <path> (w. <type> and <mode>)...
    // if data is not null, it will be added to the new leaf...

270     String parents = path.substring(0, path.lastIndexOf("/"));
    String name = path.substring(path.lastIndexOf("/") + 1);
    LinkedList nodeList = XFGetPath(parents);

    Node newElement =
275     (data == null?
        NodeFactory.createElementNode(type, XFCreateAttrs(name, mode)) :
        NodeFactory.createElementNode(type,
            NodeFactory.createCharDataNode(data),
            XFCreateAttrs(name, mode)));

280     Node parrent = (Node) nodeList.get(nodeList.size() - 1);
    ChildNodes siblings = parrent.getChildNodes();
    Node[] newSiblings = new Node[siblings.getLength() + 1];

285     for (int y=0; y<siblings.getLength(); y++) {
        newSiblings[y] = siblings.getNode(y);
    }
    newSiblings[siblings.getLength()] = newElement;

290     Node leaf = NodeFactory.createElementNode
        ("dir", NodeFactory.createChildNodes(newSiblings),
        XFmtimeAttrs(parrent.getAttributes()));

    XFUpdateTree(leaf, nodeList);
295 }

private void XFDeleteElement (String path) throws FuseException {
    // delete the node at <path>...
300     String parents = path.substring(0, path.lastIndexOf("/"));
    String name = path.substring(path.lastIndexOf("/") + 1);
    LinkedList nodeList = XFGetPath(parents);

    Node parrent = (Node) nodeList.get(nodeList.size() - 1);
    ChildNodes siblings = parrent.getChildNodes();
305     Node[] newSiblings = new Node[siblings.getLength() - 1];

```

```

    for (int y=0,z=0; y<siblings.getLength(); y++) {
        if (!siblings.getNode(y).getAttribute("name").equals(name)) {
            newSiblings[z++] = siblings.getNode(y);
310     }
    }

    Node leaf = NodeFactory.createElementNode
        ("dir",NodeFactory.createChildNodes(newSiblings),
315     XFmtimeAttrs(parrent.getAttributes()));

    XFUpdateTree(leaf,nodeList);
}

320
private Node XFFindChildNode (Node n, String name) {
    // finds the childnode from <node> with <name>...
    ChildNodes children = n.getChildNodes();
    for (int y = 0; y < children.getLength(); y++) {
325     n = children.getNode(y);
        if (n.getAttribute("name").equals(name))
            return n;
    }
    return null;
330 }

private LinkedList XFGetPath (String path) throws FuseException {
    // returns the nodes from the root to the leaf at <path>, if it exists...
335 String [] path_list = path.split("/");
    Node n = rootDir;
    LinkedList res = new LinkedList();

    res.add(n);
340
    historyMode = false;

    if (path.equals("")) {
        return res;
345 } else {
        int x;

        if (path_list.length > 1 && path_list[1].equals(HISTORY_FOLDER)) {
            historyMode = true;
350 n = NodeFactory.createElementNode("dir",root.getChildNodes(),
                XFCreateAttrs(HISTORY_FOLDER,493));

            res.add(n);
            x = 2;
        } else {
            x = 1;
355 }
    }

    for (; x < path_list.length; x++) {
360
        String type = n.getNodeValue();

        if (type.equals("dir")) {
            n = XFFindChildNode (n, path_list[x]);
        } else {
365 n = null;
        }
    }
}

```

```

        if (n == null)
            throw new FuseException("No_Such_Entry").initErrno
370             (FuseException.ENOENT);
        else
            res.add(n);
    }
    return res;
375 }
}

private void XFClose() throws FuseException {
380     // gracefully close down the xmlstore to make the changes persistent...
    try {
        xmlstore.close();
    } catch (XMLStoreIOException e) {
385         throw new FuseException(e.toString()).initErrno(FuseException.EIO);
    }
}

//
// Public metodes
390 //
public FuseStat getattr(String path) throws FuseException {
    log.debug(path);

    XFGetCurrentRoot();
395     return XFGetStat((Node) XFGetPath(path).getLast());
}

public FuseDirEnt[] getdir(String path) throws FuseException {
400     log.debug(path);

    XFGetCurrentRoot();

    boolean pathIsRoot = path.equals("/");
405     int offset = (pathIsRoot ? 3 : 2);

    Node node = (Node) XFGetPath(path).getLast();

    if (!node.getNodeValue().equals("dir"))
410         throw new FuseException("Not_A_Directory").initErrno
            (FuseException.ENOTDIR);

    ChildNodes children = node.getChildNodes();

415     FuseDirEnt[] dirEntries = new FuseDirEnt[offset+children.getLength()];

    dirEntries[0] = new FuseDirEnt();
    dirEntries[0].name = ".";
420     dirEntries[0].mode = FuseFtype.TYPE_DIR;

    dirEntries[1] = new FuseDirEnt();
    dirEntries[1].name = "..";
    dirEntries[1].mode = FuseFtype.TYPE_DIR;
425     if (pathIsRoot) {
        dirEntries[2] = new FuseDirEnt();
        dirEntries[2].name = HISTORY_FOLDER;
        dirEntries[2].mode = FuseFtype.TYPE_DIR;
    }
}

```

```

430     }

    for (int x = 0; x < children.getLength(); x++) {
        Node e = children.getNode(x);
        FuseDirEnt dirEntry = new FuseDirEnt();
435     dirEntries[x+offset] = dirEntry;
        dirEntry.name = e.getAttribute("name");

        String type = e.getNodeValue();
        if (type.equals("dir"))
440         dirEntry.mode = FuseFtype.TYPE_DIR;
        else if (type.equals("file"))
            dirEntry.mode = FuseFtype.TYPE_FILE;
        else if (type.equals("symlink"))
            dirEntry.mode = FuseFtype.TYPE_SYMLINK;
445     else
        throw new FuseException("Unsupported_type").initErrno
            (FuseException.EPERM);
    }

450     return dirEntries;
}

public void mknod(String path, int mode, int rdev) throws FuseException {
    log.debug(path + "_mode:_\" + mode + \"\"_rdev:_\" + rdev + \"\"");
455     rootLog = "mknod(\"+path+\",\"+mode+\",\"+rdev+)\";

    XFGetCurrentRoot();

    if ((mode & FuseFtype.TYPE_FILE) == FuseFtype.TYPE_FILE) {
460         XFCreateNewElement(path, "file", mode ^ FuseFtype.TYPE_FILE, "");
    } else {
        throw new FuseException("Unsupported_type").initErrno
            (FuseException.EPERM);
    }
465     XFSaveRoot();
}

public void mkdir(String path, int mode) throws FuseException {
470     log.debug(path + "_mode:_\" + mode + \"\"");

    rootLog = "mkdir(\"+path+\",\"+mode+)\";

    XFGetCurrentRoot();
475     XFCreateNewElement(path, "dir", mode, null);

    XFSaveRoot();
}
480     public void unlink(String path) throws FuseException {
        log.debug(path);

        rootLog = "unlink(\"+path+)\";
485         XFGetCurrentRoot();

        XFDeleteElement(path);

490         XFSaveRoot();
    }
}

```

```

public void rmdir(String path) throws FuseException {
    log.debug(path);
495     rootLog = "rmdir("+path+")";

    XFGetCurrentRoot ();

500     XFDeleteElement (path);

    XFSaveRoot ();
}

505 public void rename(String from, String to) throws FuseException {
    log.debug(from + "┘" + to);

    rootLog = "rename("+from+", "+to+")";

510     XFGetCurrentRoot ();

    Node node = (Node) XFGetPath(from).getLast ();

    String parents = to.substring(0, to.lastIndexOf("/"));
515     String name = to.substring(to.lastIndexOf("/") + 1);
    LinkedList nodeList = XFGetPath(parents);

    Node parent = (Node) nodeList.getLast ();
    ChildNodes siblings = parent.getChildNodes ();
520     Node[] newSiblings = new Node[siblings.getLength () + 1];

    for (int y=0; y<siblings.getLength (); y++) {
        newSiblings[y] = siblings.getNode (y);
525     }

    Attribute[] attrs = XFModifyAttrs (node.getAttributes (), "name", name);

    Node newNode = NodeFactory.createElementNode (node.getNodeValue (),
530         node.getChildNodes (),
            attrs);

    newSiblings[siblings.getLength ()] = newNode;

535     Node leaf = NodeFactory.createElementNode
        ("dir", NodeFactory.createChildNodes (newSiblings),
            XFmtimeAttrs (parent.getAttributes ()));

    XFUpdateTree (leaf, nodeList);

540     XFDeleteElement (from);

    XFSaveRoot ();
}

545 public void chmod(String path, int mode) throws FuseException {
    log.debug(path + "┘" + mode);

    rootLog = "chmod("+path+", "+mode+")";
550     XFGetCurrentRoot ();

    LinkedList nodeList = XFGetPath(path);

```

```

555     Node oldLeaf = (Node) nodeList.get(nodeList.size()-1);
        Attribute [] attrs = XFModifyAttrs
            (oldLeaf.getAttributes(), "mode", Integer.toString(mode));
560     Node newLeaf = NodeFactory.createElementNode(oldLeaf.getNodeValue(),
                                                    oldLeaf.getChildNodes(),
                                                    attrs);

        XFUpdateTree(newLeaf, nodeList);
565     XFSaveRoot();
    }

    public void chown(String path, int uid, int gid) throws FuseException {
570         log.debug(path + "_" + uid + "_" + gid);

        rootLog = "chown("+path+", "+uid+", "+gid+)";

        XFGetCurrentRoot();
575         LinkedList nodeList = XFGetPath(path);

        Node oldLeaf = (Node) nodeList.get(nodeList.size()-1);

580         Attribute [] attrs = XFModifyAttrs(XFModifyAttrs(oldLeaf.getAttributes(),
                                                            "uid", Integer.toString(uid)),
                                                            "gid", Integer.toString(gid));

        Node newLeaf = NodeFactory.createElementNode(oldLeaf.getNodeValue(),
                                                    oldLeaf.getChildNodes(),
                                                    attrs);

585         XFUpdateTree(newLeaf, nodeList);

        XFSaveRoot();
590     }

    public void utime(String path, int atime, int mtime) throws FuseException {
595         log.debug(path);

        rootLog = "utime("+path+", "+atime+", "+mtime+)";

        XFGetCurrentRoot();

600         LinkedList nodeList = XFGetPath(path);

        Node oldLeaf = (Node) nodeList.get(nodeList.size()-1);
        Attribute [] attrs = XFModifyAttrs(XFModifyAttrs
            (oldLeaf.getAttributes(),
605             "atime", Integer.toString(atime)),
            "mtime", Integer.toString(mtime));

        Node newLeaf = NodeFactory.createElementNode(oldLeaf.getNodeValue(),
                                                    oldLeaf.getChildNodes(),
                                                    attrs);

610         XFUpdateTree(newLeaf, nodeList);

        XFSaveRoot();
615     }

```

```

public void read(String path, ByteBuffer buf, long offset) throws FuseException {
    log.debug(path + "┐offset┐:┐" + offset);
620     XFGetCurrentRoot ();

    Node node = (Node) XFGetPath(path).getLast ();

    String data = node.getChildNodes ().getNode (0).getNodeValue ();
625     int start = (int)offset;
    int remaining = (data.length()-start);
    int end = (remaining <= buf.limit () ? data.length () : start + buf.limit ());

630     buf.put (data.substring (start ,end).getBytes ());
}

public void write(String path, ByteBuffer buf, long offset)
635     throws FuseException {
    log.debug(path + "┐offset┐:┐" + offset);

    rootLog = "write (" +path+", "+buf.remaining()+", "+offset+" )";
640     XFGetCurrentRoot ();

    byte [] barray = new byte [buf.remaining ()];

    buf.get (barray);
645     String newData = new String (barray);

    LinkedList nodeList = XFGetPath(path);

650     Node oldLeaf = (Node) nodeList.getLast ();

    String oldData = "";
    try {
        oldData = oldLeaf.getChildNodes ().getNode (0).getNodeValue ();
655     } catch (ArrayIndexOutOfBoundsException _) {
        // does nothing
    }

    int before = (int)offset;
660     int after = ((int)offset)+newData.length ();
    String data = oldData.substring (0 ,before) + newData +
        (after > oldData.length () ? "" :
        oldData.substring (((int)offset)+newData.length ()));

665     Attribute [] attrs = XFModifyAttrs (XFmtimeAttrs (oldLeaf.getAttributes ()),
        "size", String.valueOf (data.length ()));

    Node newLeaf = NodeFactory.createElementNode
        (oldLeaf.getNodeValue (), NodeFactory.createCharDataNode (data), attrs);
670     XFUpdateTree (newLeaf, nodeList);

    XFSaveRoot ();
}
675

public void truncate(String path, long size) throws FuseException {

```

```

        log.debug(path + "_" + size);
680    rootLog = "truncate("+path+", "+size+")";
        XFGetCurrentRoot();
        LinkedList nodeList = XFGetPath(path);
685    Node oldLeaf = (Node) nodeList.getLast();
        String data = oldLeaf.getChildNodes().getNode(0).getNodeValue();
690    Attribute[] attrs = XFModifyAttrs(XFmtimeAttrs(oldLeaf.getAttributes()),
        "size", String.valueOf(size));
        Node newLeaf = NodeFactory.createElementNode
695        (oldLeaf.getNodeValue(),
        NodeFactory.createCharDataNode(data.substring(0, (int) size)), attrs);
        XFUpdateTree(newLeaf, nodeList);
700    XFSaveRoot();
    }

    public void symlink(String from, String to) throws FuseException {
705        log.debug(from + "_" + to);
        rootLog = "symlink("+from+", "+to+")";
        XFGetCurrentRoot();
710    XFCreateNewElement(to, "symlink", 493, from); //493=rwxr-xr-x
        XFSaveRoot();
    }

    public String readlink(String path) throws FuseException {
715        log.debug(path);
        XFGetCurrentRoot();
720    Node node = (Node) XFGetPath(path).getLast();
        return node.getChildNodes().getNode(0).getNodeValue();
    }

    public void open(String path, int flags) throws FuseException {
725        log.debug(path + "_flags:_" + flags + "_ (NB!_Does_nothing_at_the_moment!)");
    }

    public void release(String path, int flags) throws FuseException {
730        log.debug(path + "_flags:_" + flags + "_ (NB!_Does_nothing_at_the_moment!)");
    }

    //
735    // Unimplemented !
    //
    public FuseStats statfs() throws FuseException {
        log.warn("Static_dummy_function._ (NB!_Does_nothing_at_the_moment!)");
    }

```

```

740     FuseStatfs statfs = new FuseStatfs ();

        statfs.blocks = Integer.MAX_VALUE;
        statfs.blocksFree = Integer.MAX_VALUE;
        statfs.files = 0;
745     statfs.filesFree = Integer.MAX_VALUE;
        statfs.blockSize = 1048576;
        statfs.namelen = 1024;

        return statfs;
750 }

public void link(String from, String to) throws FuseException {
    log.error(from + "_" + to);
    throw new FuseException("Unimplemented").initErrno(FuseException.EPERM);
755 }

//
// Java entry point
//
760 public static void main(String[] args) { //throws Exception
    log.info("entering");
    System.out.println("Starting_FuseX");

    XMLStoreFilesystem xmlstorefs = null;
765

    try {
        if (args.length == 3) {
            log.info("p2p_setup");

770             String[] bootArg = args[2].split(":");
            InetAddress bootHost = new InetAddress
                (bootArg[0], Integer.parseInt(bootArg[1]));
            int tcpPort = Integer.parseInt(args[1]);
            int udpPort = tcpPort;
775             String diskName = "/var/planx/client_"+tcpPort+"_disk";
            String routingName = "/var/planx/client_"+tcpPort;

            System.out.println("Using_"+bootHost+"_as_boot-peer.");
            XMLStore _xmlstore = new DistributedXMLStore
780                 (new WriteBufferXMLStore(diskName), routingName, udpPort, tcpPort,
                    bootHost, true);

            NameServer _ns = ((DistributedXMLStore)_xmlstore).getNameServer();
            xmlstorefs = new XMLStoreFilesystem (_xmlstore, _ns);
785         } else if (args.length == 1) {
            log.info("stand-alone_setup");
            String diskName = "/var/planx/fusex_disk";
            xmlstorefs = new XMLStoreFilesystem
790                 (new WriteBufferXMLStore(diskName), new LocalNameServer
                    (diskName+"_ns"));
        } else {
            System.err.println
800                 ("USAGE:_mount._fusex._mountpoint_[port_bootheostname:bootport]");
            System.exit(1);
        }
    }
    FuseMount.mount(new String[] { args[0] }, xmlstorefs);
    log.info("closing_xmlstore");
    xmlstorefs.XFclose();
}
catch (Exception e) {
    e.printStackTrace();
}

```

```
    }  
    finally {  
        log.info("exiting");  
    }  
805 }  
}
```

B.2 XMLStorePeer.java

```
/*
 * FUSEX:
 *
 * XMLStorePeer
 *
 * Copyright (C) 2004 Christian Hemmingsen (kewl@diku.dk)
 * Bo Bendtsen (bobend@diku.dk)
 */

10 import org.planx.xmlstore.*;
import org.planx.xmlstore.stores.*;
import org.planx.xmlstore.nameserver.*;
import org.planx.xmlstore.nodes.*;
import java.net.*;

15 public class XMLStorePeer {
    public static void main (String [] args) throws Exception {
        System.out.println("XMLStore_peer");

20        int tcpPort;
        int udpPort=0;
        String diskName;
        String routingName;
        InetAddress bootHost=null;

25        switch (args.length) {
            case 2:
                String [] bootArg = args [1].split (":");
                bootHost = new InetAddress (bootArg [0],
30                Integer.parseInt (bootArg [1]));
                System.out.println ("Using_" +bootHost +"_as_boot-peer.");
            case 1:
                udpPort = Integer.parseInt (args [0]);
                break;
35        default :
            System.err.println ("USAGE: _xmlstorepeer_port_[boothostname:bootport]");
            System.exit (1);
        }

40        System.out.println ("Starting_XMLStore_peer:");
        tcpPort = udpPort;
        //diskName="peer_"+InetAddress.getLocalHost().getHostName()+udpPort+"_disk";
        //routingName="peer_"+InetAddress.getLocalHost().getHostName()+udpPort;

45        diskName = "/var/planx/peer_" +InetAddress.getLocalHost().getHostName()+
            udpPort+"_disk";
        routingName = "/var/planx/peer_" +InetAddress.getLocalHost().getHostName()+
            udpPort;

50        DistributedXMLStore xmlstore = null;
        NameServer ns = null;

        try {
            xmlstore = new DistributedXMLStore
55            (new RawXMLStore (diskName), routingName, udpPort, tcpPort,
                bootHost, true);
            ns = xmlstore.getNameServer ();
        } catch (Exception e) {
            System.err.println ("ERR:\n"+e+"\n");
60            System.err.println ("USAGE: _xmlstorepeer_port_[boothostname:bootport]");
        }
    }
}
```

```

        System.exit(1);
    }

    System.out.println("Press \"q\" to shut down peer.");
65 System.out.println("Press \"d\" for debug info.");
    System.out.println("Press \"w\" for write test.");
    System.out.println("Press \"r\" for read test.");

    int command = 0;

70 while (command != 'q') {
        if (command == 'd') {
            try {
                System.out.println
75                 (xmlstore.load
                    (ns.lookup("fusex")).toString());
            } catch (Exception e) {
                System.err.println("NO_FUSEX_DATA_FOUND!_OR_EXCEPTION:\n"+e);
            }
        } else if (command == 'w') {
            try {
                ValueReference oldVr = ns.lookup("peertest");

                Node test = NodeFactory.createCharDataNode
85                 ("testdata_from_"+InetAddress.getLocalHost().getHostName()+
                    ":"+tcpPort + "_time_was_"+System.currentTimeMillis());
                ValueReference newVr = test.getValueReference();
                xmlstore.save(test);
                if (oldVr==null) {
                    ns.bind("peertest",newVr);
90                 } else {
                    ns.rebind("peertest",oldVr,newVr);
                }
                System.out.println("Write test done!");
            } catch (Exception e) {
                System.err.println("Write failed!\n"+e);
            }
        } else if (command == 'r') {
            try {
                System.out.println
100                 (xmlstore.load
                    (ns.lookup("peertest")).toString());

                System.out.println("Read test done!");
            } catch (Exception e) {
                System.err.println("Read failed!\n"+e);
            }
        }
        command = System.in.read();
110    }
    xmlstore.close();
    System.out.println("Closed down gracefully.");
}
}

```

B.3 mount.fusex

```
#!/bin/bash
JDK_HOME=/opt/sun-j2sdk-1.4.2
PLANX_PATH=/home/bob/datalogi/plan-x
FUSE_J_PATH=$PLANX_PATH/projects/fusex/fuse-j
XMLSTORE_JAR=$PLANX_PATH/xmlstores/bogebjerg/store/build/xmlstore.jar
FUSEX_PATH=$PLANX_PATH/projects/fusex/src/build

LD_LIBRARY_PATH=$FUSE_J_PATH/jni \
    $JDK_HOME/bin/java \
    -Dfuse.log=DEBUG \
    -classpath $FUSEX_PATH:$FUSE_J_PATH/build:$XMLSTORE_JAR \
    XMLStoreFilesystem $@
```

B.4 xmlstorepeer

```
#!/bin/bash
JDK_HOME=/opt/sun-j2sdk-1.4.2
PLANX_PATH=/home/bob/datalogi/plan-x
XMLSTORE_JAR=$PLANX_PATH/xmlstores/bogebjerg/store/build/xmlstore.jar
FUSEX_PATH=$PLANX_PATH/projects/fusex/src/build

$JDK_HOME/bin/java \
    -classpath $FUSEX_PATH:$XMLSTORE_JAR \
    XMLStorePeer $@
```

Litteratur

- [1] <http://sourceforge.net/projects/avf>.
- [2] <http://www.plan-x.org/>.
- [3] Neil Brown. *The Linux Virtual File-System Layer*. University of New South Wales, December 1999. <http://cgi.cse.unsw.edu.au/~neilb/oss/linux-commentary/vfs.html>.
- [4] Kasper Bøgebjerg Pedersen and Jesper Teglgard Pedersen. Value-oriented xml store. Master's thesis, ITU/DTU, 2002.
- [5] Silberschatz, Galvin, and Gagne. *Operating System Concepts*. Jonh Wiley and Sons inc., sixth edition, 2002.