Masteroppgave:
Remmen Streaming Server
Et distribuert videostreamingsprogram

Andreas Bergstrøm
Høgskolen i Østfold
Avdeling for Informatikk

5. oktober 2010

Opphavsrett

Sammendrag

Oppgaven omhandler utviklingen av Remmen Streaming System (RStS), underliggende teknologier og deres rolle i utviklingen av systemet.

Bakgrunnen for oppgaven er Høgskolen i Østfolds utbygging av bygningsmassen på Remmen i Halden (se side 11). Designet for dette bygget var å gå bort ifra koaksialkabling for distribusjon av TV og video, med et ønske om å benytte kun nettverkskabel for all distribusjon av TV og video.

Mulighetene som ble vurdert var innkjøp av en allerede eksisterende kommersiell løsning, egenutvikling av en softwarebasert løsning eller å beholde tradisjonell koaksialkabling.

Resultatet av denne prosessen ble RStS, nå i sin andre versjon ved navn Alt Når Det Passer (ANDP), men det vil være arbeidet med den første versjon som diskuteres i denne oppgaven.

RStS er en distribuert løsning for å ta opp og streame satellitt-TV-kanaler. Den skiller seg ut fra andre tilsvarende løsninger ved å være skrevet i åpen kildekode basert på åpne standarder uten bruk av noen proprietære løsninger. Løsningen er tatt i bruk ved Høgskolen i Østfold og er også planlagt pakket inn for distribusjon til alle Uninetts medlemsinstitusjoner for bruk i undervisningen.

Løsningen er basert på Python, XML-RPC for distribuert programmering, DreamBox linux-baserte satellitt-mottakere samt Hauppauge PVR 250 og 350 MPEG-2 videodigitaliseringskort.

Takk til

Jeg vil gjerne takke Audun Vaaler for hans hjelp under arbeidet med denne oppgaven. Hans innspill og innsigelser har vært nyttige og nødvendige verktøy, og hans arbeid med ANDP har tatt resultatene ett skritt videre.

Jeg vil også takke min veileder, Børre Ludvigsen, for hans hjelp med denne oppgaven.

Innhold

Figurer

Innledning

Denne oppgaven vil omhandle Remmen Streaming System (RStS), dets design og teknologi. RStS var ment å erstatte et tradisjonelt koaksialbasert video-distribusjonssystem, med et IP basert system. Dette for å slippe å legge koaksial-kabler i Høgskolen i Østfolds nye høgskolebygg i Halden, og dermed spare de kostnader det ville ha medført. Samtidig skulle løsningen være basert på åpen kildekode og åpne standarder.

Del I

Referansemateriell

Kapittel 0

Bakgrunnsstoff

Dette kapittel gir en innføring i de teknologier som diskuteres i denne oppgaven. Hensikten med dette er å gi en kort innføring i teknologien, samt gi en enkel analyse av de faktorer som førte til valget av disse teknologier.

0.1 Python

Python er et fritt, åpent og plattformuavhengig programmeringsspråk[1] uten lisensproblematikk. Det finnes i dag implementasjoner for de fleste UNIX-plattformer, Mac Os, MS-DOS, Windows familien og OS/2.

Python bruker indentering for syntaks-parsing, som tvinger frem velstrukturert kode. Python har et standardbibliotek av funksjoner for bl.a. strengprossessering, internettprotokoller, software engineering og debugging, og operativsystem-interfacing. I tillegg finnes også et antall tredjeparts utvidelser fritt og åpent tilgjengelig.[2]

[1] Med 'fritt og åpent' menes at tilgang til å forandre og publisere forandringene av språket er tillatt. Videre kan en fritt få tilgang til spesifikasjon, og i henhold til EUs standard er det ikke patentbelagt, eller patentene er lisensiert uten kostnad til evig tid.

[2] Python[3] ble opprinnelig skrevet av Guido van Rossum, som leder Python utviklingen også i dag, mens han jobbet ved CWI. Guido arbeidet der med det interpreterte programmeringsspråket ABC, og lærte fra det en god del om design av programmeringsspråk. Mange av Pythons grunnleggende trekk stammer derved fra ABC. Dette inkluderer indentering for syntaks interpreting og muligheten for veldig-høynivå datatyper.

Van Rossum merket seg mangler ved ABC, slik som mangel på utvidbarhet, men fant også styrker han ville ha med seg videre. Han tok videre med seg syntaks og semantikk for exception-handling fra Modula.

Van Rossum arbeidet ved Amoeba distribuert operativsystemgruppen ved CWI, og trengte noe bedre for systemadministrasjon enn C-programmering og bourne-shell. Guido kom da til den konklusjonen at et skriptingspråk med syntaks som ABC, men med tilgang til systemkall ville være meget passende.

Van Rossum arbeidet med å lage Python for Amoeba prosjektet, primært på sin egen fritid, men ble inspirert av de gode tilbakemeldingen han fikk fra sine kolleger.

Python har følgende fordeler som er vesentlige for dette prosjektet: Det er et høynivå generisk programmerings- og skriptingspråk, som er interpretert, interaktivt, og objektorientert. Det har støtte for moduler, exceptions, dynamisk typing, ekstrem høynivå dynamiske datatyper og klasser, alt med en klar og entydig syntaks.

Python er et interpretert programmeringsspråk, og vil derfor ikke ha samme ytelsen som et kompilert programmeringsspråk som f.eks. C. Python interpreteren er skrevet og kompilert i C, og det er mulig å interface direkte mot C-bibliotek samt egenskrevet C kode, slik at denne ulempen i de fleste tilfeller ikke et problem. For RStS var rask utvikling og vedlikehold viktigere enn ytelse.

Python ble valgt som programmeringsspråk utifra en totalvurdering som la vekt på anvendbarhet, utviklingstid og hvilke språk som en allerede behersket. Se også side 18 for en nærmere diskusjon om valg av programmeringsspråk.

0.2 XML-RPC

XML-RPC er en protokoll for distribuert databehandling, en *Remote Procedure Calling* protokoll [4]. Den er designet for å brukes over internett og andre nett som benytter seg av HTTP.

XML-RPC er en XML basert RPC protokoll. RPC er en teknikk for å lage distribuerte, klient-server baserte applikasjoner. Det er i prinsippet en utvidelse av de tilgjengelige programkall til benyttelse over flere maskiner [5]. Tradisjonelt har program kun hatt tilgang til programkall lokalt på maskinen, primært kun internt i programmet. RPC lar programmet gjøre programkall mot andre prosesser, gjerne på det samme datasystemet, eller på et annet system, tilkoblet med et datanettverk. Ved å bruke RPC kan en lage en distribuert applikasjon uten å ta hensyn til den underliggende nettverksprotokoll.

RPC er et rammeverk for å gjøre klient/server programmering mer kraftig og enklere å programmere.

Et RPC kall fungerer som et vanlig funksjonskall. Når et RPC kall blir sendt, sendes argumentene videre til prosedyren som kalles og prosedyren som gjorde dette, venter på et svar. For RPC innebærer dette at klient sender kallet til serveren, og venter. Tråden er blokkert for videre prosessering frem til et svar mottas eller kallet timer ut. Når et kall ankommer, kaller serveren en prosedyre som utfører kallet, og sender svaret til klienten. Når RPC kallet er ferdig, fortsetter klientprosedyren.

Prosessen rundt valget av XML-RPC er nærmere diskutert på side 30.

Python ble offentliggjort da Guido postet kildekoden på USENET i februar 1991, og har siden da utviklet seg til et godt, moderne og høyt anvendelig programmeringsspråk.

Utvikling av applikasjoner med RPC

RPC egner seg for å forenkle applikasjoner som må kjøre over flere maskiner, men det finnes flere alternativer, slik som å bruke telnet eller SSH til å logge på den eksterne maskinen, eksekvere et program, og så benytte seg av output fra det eksekverte program.

De fleste Unix baserte operativsystemer, og også nyere Windowsbaserte operativsystemer støtter telnet, shell og SSH basert innlogging og kjøring av programmer. En kunne da se for seg en applikasjon som benytter seg av et slikt remote shell, kjører et program på en annen maskin, og kopierer over resultatet. Dette gir ikke alltid et pent resultat, det krever et login per servermaskin, og det kan bli tregt.

Med en RPC basert applikasjon vil man isteden spesifisere en protokoll for RPC kommunikasjon, utvikle en klientapplikasjon og en serverapplikasjon som snakker sammen med RPC kall, som regel enklere, raskere og programmeringsmessig "renere" enn bruk av shell, telnet eller SSH.

XML og RPC

XML-RPC er en standardisert RPC protokoll basert på XML.

XML-RPC kall sendes som HTTP POST kall, i kroppen til kallet sendes en XML-streng istedenfor HTML. Prosedyren eksekveres på serveren, og verdien som returneres formatteres som XML.

Formålet med en XML basert RPC protokoll er å ha en enkelt implementerbar protokoll for RPC over internett. Ved å basere seg på HTTP protokollen, vil kallene gå enkelt igjennom alle brannmurer. Ved samtidig å ha en nærmest selvdokumenterende XML innkapsling av programmkallene, kan protokollen enkelt tas i bruk av alle med grunnleggende programmerings og XML kunnskaper.

0.3 Streaming og videodistribusjon

Videolan og VLC

VLC [3] er en fri og åpen mediespiller. Den støtter de vanligste lyd- og bildekodeker som en vanlig bruker kan møte. VLC har også støtte for å enkode lyd-

[3] Videolan[6] var opprinnelig et studentprosjekt ved Ecole Centrale Paris, der en gruppe studenter ønsket å kunne se på TV på deres datamaskiner. I 1998 hadde deres arbeid med VideoLAN Server (VLS) og VideoLAN Client (VLC) nådd det punktet der de kunne sende og motta strømmer. Begge applikasjoner ble designet for å være modulære med en kommunikasjonskjerne i midten.

De første VLC og VLS versjonene var proprietære og støttet kun MPEG2. I 2001 fikk studentene tillatelse av Ecole til å frigjøre koden under GNU Public Licence (GPL), noe som økte antall utviklere raskt.

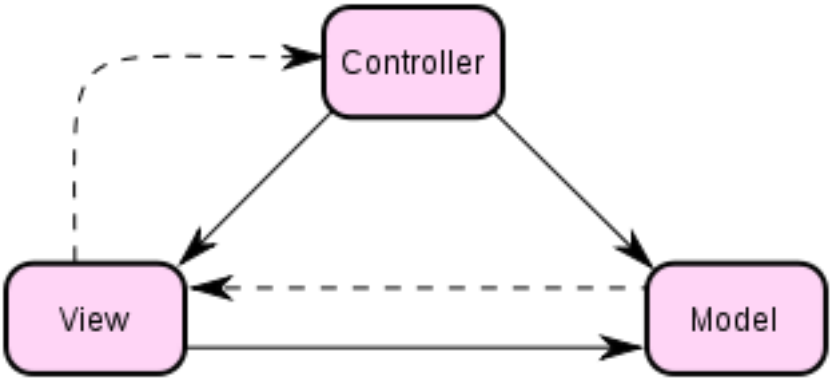
og bildedata i de fleste av de kodeker den kan lese, slik som H.264, Theora, MPEG-4 og andre.

VLC kan kontrolleres fra flere grafiske brukergrensesnitt, kommandolinjen og også fra en innebygd telnet-server. Denne fleksibiliten ble vurdert som sentral i valget av teknologi for mediaspiller og enkoder, samtidig som dette var teknologi som en allerede var godt kjent med.

0.4 Designmetodikk

Model-view-controller

MVC (Model - view - controller) [7] er en designmodell ment å forene brukerens mentale modell av et system, og den digitale modellen som eksisterer i data-maskinen. En ideell MVC løsning skaper illusjonen av at brukeren manipulerer data direkte.



Figur 0.1: Model view controller diagram. [2]

Model
Modellen definerer dataene i systemet. En modell kan være et enkelt objekt eller en objektstruktur. Modellen bør ha en en-til-en relasjon mellom enkelt-objektene i modellen og verden slik designeren ser det.

Videre arbeid med VLC og VLS muliggjorde storskala multicasting med VLS, og i 2003 ble også MPEG4 støtte lagt til.

VideoLAN-prosjektet er i dag det ledende open-source prosjektet for visning og streaming av multimedia. VLS serverens funksjonalitet finnes i dag bygd inn i VLC, som fortsatt har styrken av å være en totalt modulær applikasjon. VLC kan i dag spille av de fleste vanlige multimediaformater, slik som de forskjellige MPEG standarder, Windows Media, Quicktime og Ogg-filer.

View
Et view er en representasjon eller presentasjon av en modell. Et view vil normalt vise et subsett av modellens attributter arrangert på en spesiell måte. Et view kan derfor defineres som et presentasjonsfilter.

Controller
En controller er linken mellom brukeren og systemet. All interaksjon foregår via denne. Den gir brukeren data ved å vise views, som regel basert på brukerens input. Kontroller behandler all input, og oppdaterer modellen, men vil aldri supplementere viewet. På samme måte vil aldri viewet vite om inndata.

0.5 PostgreSQL

PostgreSQL [8] er en åpen og fri databasemotor. Dette gir PostgreSQL noen fordeler over de tradisjonelle kommersielle databasemotorer, slik som lisensproblematikk i forhold til antall brukere av systemet. Der kommersielle systemer er lisensiert per bruker, har PostgreSQL ingen begrensninger. Kommersielle databaser kommer dog med bedre supportløsninger, men selv om PostgreSQL ikke kan tilby dette, så tilbyr både utviklere og andre sluttbrukere gjerne hjelp med problemer. PostgreSQL kan normalt kjøres i flere år mellom uten i databasemotoren, på like linje med de kommersielle. Det er designet for et minimum av vedlikehold og opplæringskostnader, også ved overgang fre et kommersielt system, slik som Oracle. Det at det er fritt og åpent lar en enkelt skreddersy og utvide databasemotoren ved behov. Databasemotoren er også kryssplattform, og kjører på de fleste moderne operativsystemer, slik som Unix, Linux og Windows. Databasedesign og administrasjonsverktøy finnes både åpent og fritt, men også som produkter fra kommersielle aktører.

Valget av PostgreSQL fremfor andre databasemotorer, slik som bl.a. MySQL (for dypere diskusjon, se side 34), var i hovedsak begrunnet i to elementer. Primært fordi jeg allerede hadde erfaring med motoren, men også fordi PostgreSQL ble oppfattet som mer skalertbart en MySQL, som hovedsakelig benyttes til LAMP (Linux, Apache, MySQL, PHP) for enkle og kompliserte websider.

4

[4]PostgreSQL startet som mange andre tunge open-source prosjekter ved University of California at Berkeley (UCB). Professor Stonebraker ved UCB startet et databaseprosjekt kalt Postgres, som en oppfølger til Ingres, som hadde vært et eksperiment med klassisk relasjonsdatabaseteori.

Postgres ble utviklet i akademia fra 1986 til 1996 med det mål å teste ut nye databasekonsepter, slik som objektrelasjonsteorier. Under denne tiden ble regler/beskrankninger, prosedyrer, dato/tid-datatyper, utvidbare typer med indekser samt objektrelasjonell funksjonalitet utprøvd og implementert.

I 1995 arbeidet to av Stonebrakers Ph.D. studenter med å bytte ut Postgres egenutviklede spørrespråk med et utvidet utvalg fra SQL-standarden.

I 1996 tok Postgres skrittet ut i det alminnelige open-source miljøet, hvor en gruppe utviklere så potensialet i databasemotoren. Over det neste tiåret ble koden arbeidet med slik at den ble konsistent og uniform, utallige regresjonstester ble utviklet for kvalitetssikring, et allment kvalitetssikringssystem ble tatt i bruk å sikre at bugs ble fanget opp, prioritert og fikset, nye

muligheter ble lagt til og dokumentasjon skrevet.

Postgres regnes i dag for å være en godt fundamentert og bunnsolid databasemotor, og det er uvanlig å finne en større bedrift eller offentlig institusjon som ikke bruker PostgreSQL til noen applikasjoner.

Del II

Remmen Streaming Server

Kapittel 1

Design av RStS

Bakgrunnen for Remmen Streaming Server er Høgskolen i Østfolds nye høgskolebygg i Halden. I den sammenheng ønsket man at videodistribusjon ble flyttet fra de tradisjonelle koaksialkabler til å være standard IP trafikk over høgskolens nettverk på lik linje med all annen trafikk.

Løsningen skulle være en kombinasjon av video-on-demand og sanntids streaming, der en startet med en live streaming del, og skulle legge til funksjonalitet over tid.

1.1 Problemstillingen

Problemstillingen for RStS var å få laget et IP basert multimediesystem. Det skulle dekke funksjonaliteten til et tradisjonelt sanntids koaksialsystem, men også kunne støtte andre kilder.

En så for seg at det kunne støtte satellittbaserte kilder, digitalisere og lagre gamle VHS kassetter, og kanskje også CD og DVDer. Videre skulle det være modulbasert og generelt nok til å kunne støtte andre kilder når de kom til, slik som f.eks. opptak av forelesninger.

Designet av administrasjonsgrensesnittet er beskrevet i gjennomføringskapitelet (se side 15).

Det vil her bli sett nærmere på prosessene bak designet av det overordnede systemet, samt de biter som ikke dekkes av gjennomføringskapitelet.

1.2 Overordnet design

RStS hadde ingen designspesifikasjon fra oppdragsgiver utover å erstatte koaksialkabling med TV over IP. Det ble valgt å bruke grunnleggende designspesifikasjoner i tillegg til designspesifikasjonen, utifra en antakelse om hva en best mulig løsning ville innebære.

- Systemet skulle være objektorientert.
  Ut i fra dette ble det valgt å bruke en objektorientert designmetodikk for å sikre en programmatisk modul/objektbasert løsning med svake bindinger mellom moduler for å sikre lett utskiftbarhet av enkeltmoduler.
- Systemet skulle være eksekverbart modulært og kunne kjøre over flere fysiske datamaskiner.
  Ut i fra dette ble det valgt å finne en metodikk som tillot en form for RPC (remote procedure calls).
- Systemet skulle være datamessig modulært og data skulle være frakoblet kode.
  Ut i fra dette er en metodikk som i etterkant kan minne om MVC (Model - View - Controller) valgt.

Selv om metodikken bak MVC ikke var kjent under designprosessen, er det resulterende designet i stor grad kompatibelt med MVC. Visning ble adskilt fra data, og data ble adskilt fra kode. Som beskrevet i gjennomføringskapitelet er systemet delt i tre: Frontend/GUI, Logic-modulene og Manager-modulene.

Frontend/GUI-modulene inneholder kun visningslogikk. De benytter seg av kall mot ApplicationLogic modulen for å sende brukerens oppdateringer inn i systemet, og viser bare resultatet av disse oppdateringer. Koden er også modulær og utskiftbar, da den kun baserer seg på kjente kall inn i Application-Logic, og kan derfor byttes ut med annen visningslogikk skulle det bli nødvendig. Denne delen av RStS er dermed kompatibel med et view i MVC metodikken.

Logic- og manager-modulene inneholder både kode for henting og manipulering av data, samt kode for å kontrollere hele systemet. Denne blandingen av model og controller fra MVC metodikken gjør den ikke helt kompatibel med MVC, men det vil allikevel være relativt enkelt å bytte ut enkelt-moduler, noe som også fant sted under utviklingsprosessen og som er beskrevet i gjennomførings- kapitelet.

I tillegg er manipulasjon av data begrenset til en modul, mens alle andre sender manipulerte data inn, og videresender eller bruker resultatet uten å manipulere eller prosessere dataene.

Kapittel 2

Sammenlignbare systemer

Det ble gjort lite systematisk innhenting av data om sammenlignbare systemer under selve prosessen for design av systemet. Det fantes erfaring som ble benyttet for selve designet i Uninett-systemet, samt hos Børre Ludvigsen og Audun Vaaler, etter mange års FOU arbeid med multimediesystemer over nett.

Et kursorisk nettsøk etter tilsvarende systemer fant noen få kommersielle systemer.

I 2010 finnes det mange leverandører av video over IP/satelitt over IP systemer, slik som Adtec, Cisco, ViewCast, Fujitsu, Digital Rapids, Teradek, Inlet og andre kommersielle leverandører. Disse systemene er dog alle basert på lukket teknologi.

Det finnes en løsning for privatpersoner, MythTV [9], som gir samme funksjonalitet som RStS, men for privatpersoner og kun en bruker. Dette prosjektet ble i liten grad vurdert da det var lite utbredt når arbeidet med RStS ble startet, og det ble vurdert slik at det var mer arbeid å gjøre daværende MythTV multibruker enn å starte arbeidet fra bunnen av, samt problemed med krypterte kanaler og fraværet av open source drivere gjorde det umulig å se på majoriteten av de kanaler som oppdragsgiver ønsket.

Kapittel 3

Gjennomføring

3.1 Remmen Streaming Server (RStS)

Bakgrunn

I første fase av prosjektet ble administrasjonsbrukergrensesnittet og den bakenforliggende koden laget. Senere ble realtime videostreamingsbiten og til slutt opptak for video-on-demand koden programmert.

Konsept

Remmen Streaming Server er et video over IP distribusjonssystem for TV og radiokanaler tatt imot over satellitt, og for å kunne støtte flest mulig samtidige strømmer er det grunnleggende designet distribuert.

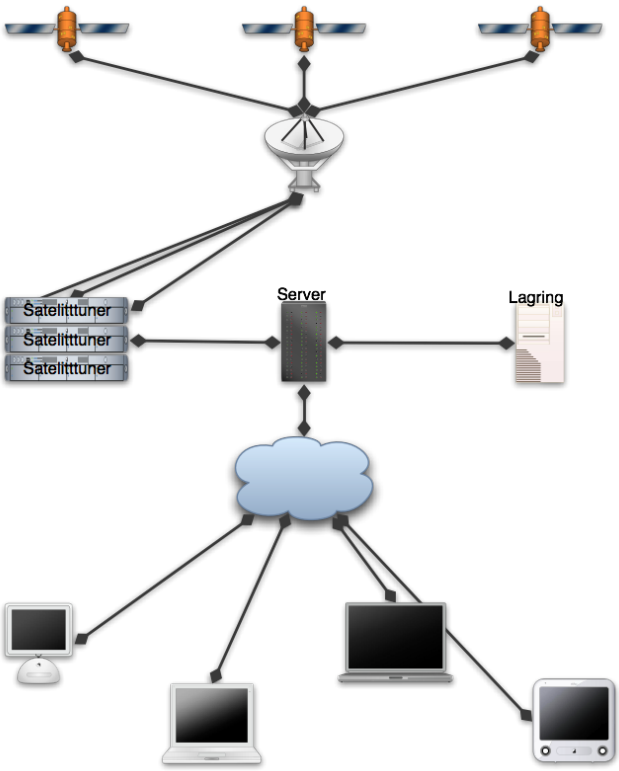
Programmer har tre lag av moduler.

På øverste lag finnes brukergrensesnittene, både for administratorer av systemet, og for brukerne av systemet.

På neste lag finnes kjernen i Remmen Streaming Server, kontroll-modulen. Alle programkall som krever informasjon utover hva den enkelte modul kjenner til, går gjennom denne modulen, og det er denne modulen som styrer de underliggende streamingmoduler.

På nederste lag finnes applikasjonsmodulene. Disse modulene gjør de underliggende jobbene i systemet, slik som realtimestreaming, opptak, og godkjenning av reservering.

All kommunikasjon mellom modulene foregår over XML-RPC og tillater



Figur 3.1: En konseptuell skisse av et satellittstreamingsystem.

en distribuert løsning som kan gå over flere servere, eller som kan samles på en server.

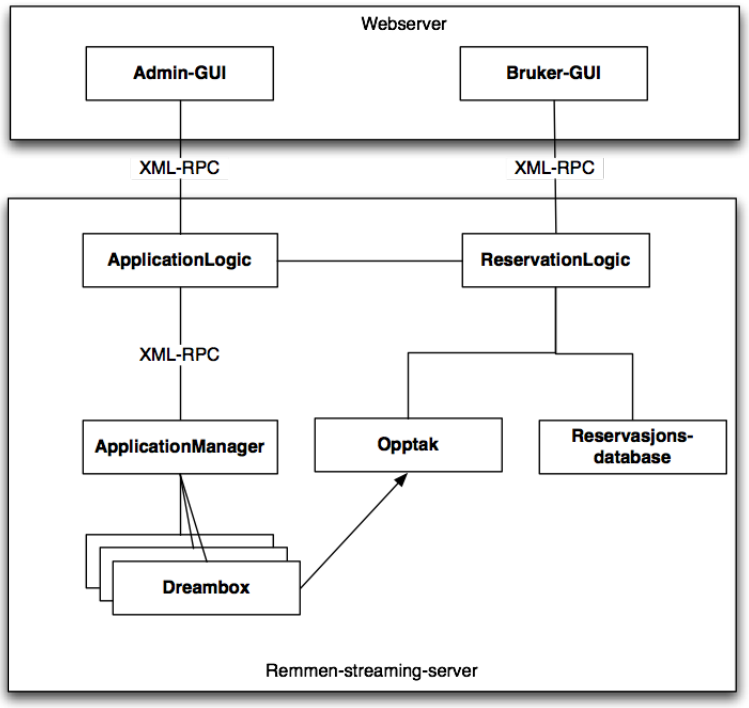
Brukergrensenittet

Administrasjonsbrukergrensesnittet

Det første som ble laget var administrasjonsbrukergrensesnittet og dets kommunikasjon med kjernemodulen ApplicationLogic.

Den første designutfordringen var hva som skulle lages først. Administrasjonsbrukergrensesnittet eller kjernemodulen ApplicationLogic.

Begge alternativ hadde sine fordeler og ulemper. Ved å lage ApplicationLogic først ville man sikre at den var internt meget effektiv, og i stand til å gjøre sine oppgaver enkelt og sikkert. ApplicationLogic var ment å være selve kjernen i programmet, og skulle være den delen som alle andre moduler kommuniserte gjennom eller med. Problemet her var at brukergrensesnittet ble begrenset av



Figur 3.2: Skisse over RStSs moduloppbygning.

hvordan ApplicationLogic var implementert, og løsninger som var bedre for administratorbrukeren kunne bli vanskeligere å implementere.

Alternativet, som også ble valgt, var å utvikle disse to modulene i parallell, med felles kravspesifikasjon der ApplicationLogic utvikles til å enkelt kunne gi administrasjonsgrensesnittet tilgang til den informasjon og de kommandoer som må til for å gi et best mulig brukergrensesnitt.

Selve administrasjonsbrukergrensesnittet ble utviklet som et prosjekt i kurset Grensesnittdesign, mens ApplicationLogic modulen ble utviklet som en del av Uninett prosjektet Video over IP.

Kravene til grensesnittet var følgende:

- Alle tilgjengelige strømmer skulle vises for administratoren.
- All tilgjengelig informasjon om de enkelte strømmer skulle vises
- La administratoren forandre identifiseringen av en strøm
- La administratoren lage nye strømmer
- La administratoren re-konfigurere eksisterende strømmer

- La administratoren bestemme dekningen til den enkelte strøm
- La administratoren stoppe og starte enkeltstrømmer
- All HTML kode skulle kunne valideres som XHTML 1.0 Strict
- All CSS kode skulle være validerbar i henhold til CSS 1.2
- All kommunikasjon mellom moduler skulle foregå over XML-RPC

I og med at det skulle bli utviklet både programmoduler og et webgrensesnitt med CGI funksjonalitet, måtte et passende programmeringsspråk bli valgt. For CGI grensesnitt er det tre programmeringsspråk som er de dominerende.

PHP er et rent skriptingspråk, det er veldig populært til denne typen oppgaver og det lar utvikleren raskt utvikle CGI grensesnitt og HTML-sider. Problemet med PHP var att det ikke egnet seg til de bakenforliggende modulene i RStS.

C er en gammel traver. Det er et lavnivå programmeringsspråk som kan gjøre stort sett det meste, og med en god programmerer og nok tid, så og si alt. Det er dog ikke et raskt språk for utvikling, men et kraftig et.

Python er både et programmeringsspråk, og et skriptingspråk. Det har fordelen av å raskt komme igang som PHP, og samtidig mye av kraften til C.

Da målet var å få opp en prototype raskt, og Python var kraftig nok til å lett kunne tilfredsstille de krav som var satt, ble Python valgt som programmeringsspråk.

Den neste problemstillingen var hvordan grensesnittet skulle se ut. Da det var flere strømmer som skulle vises samtidig, krevde det et designparadigme som lot en vise flere logiske enheter med informasjon samtidig.

Det var flere muligheter som ble vurdert, men disse var alle variasjoner over en "kortstokk" lagt ut over ett bord.

Se skisse prototype 1, 2, 3, 4.

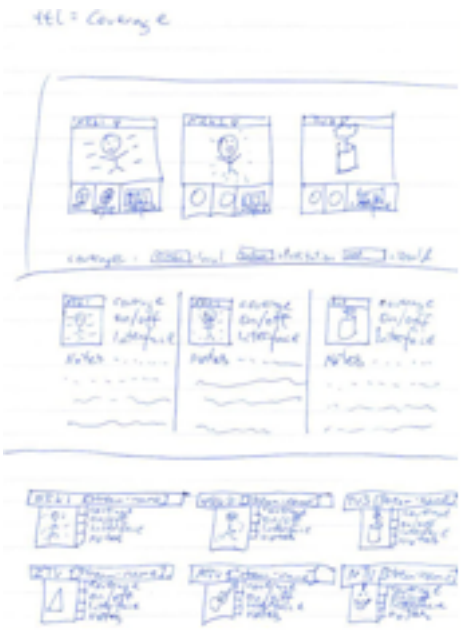
En dropdownboks kan inneholde mange kanaler uten at de tar opp skjermplass, og ble valgt uten vurdering av alternativer.

Neste punkt var å vise streamingstatus for administratoren.

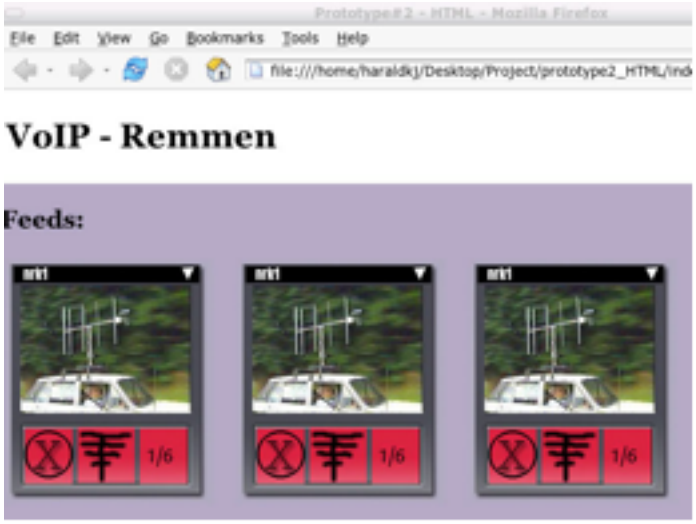
Det var her flere muligheter, en er en rød/grønn lampe som viser om den er av eller på, en annen er et snapshot av streamen slik den er i øyeblikket og den tredje var å vise streamen slik den er i øyeblikket, altså spille av alle strømmene i systemet samtidig.

Se skisse prototype 5.

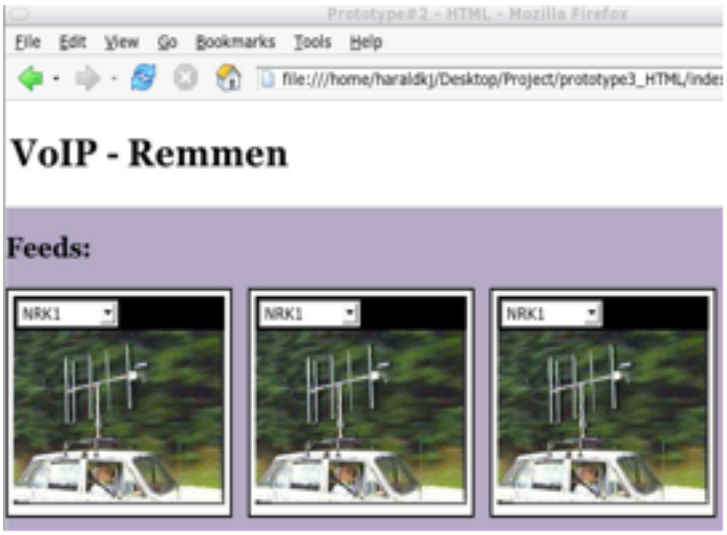
Det å spille av alle strømmene samtidig viste seg å være meget ressurskrevende, og viste seg under testing å være veldig forvirrende..



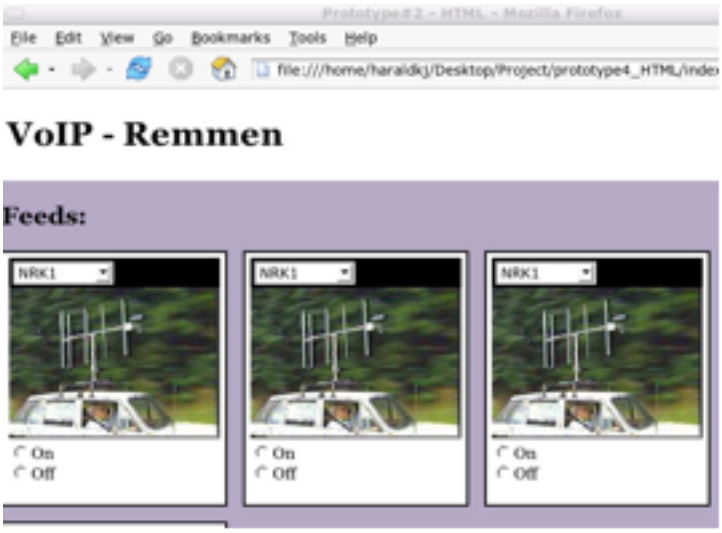
Figur 3.3: Administratorgrensesnitt prototype 1.



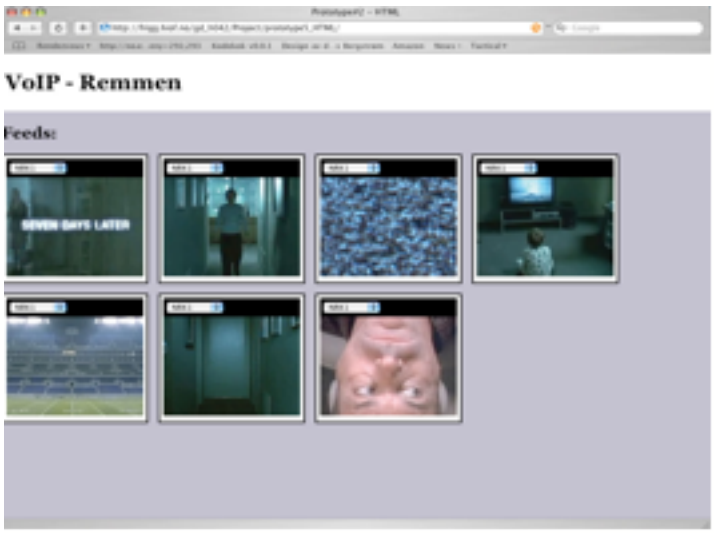
Figur 3.4: Administratorgrensesnitt prototype 2.



Figur 3.5: Administratorgrensesnitt prototype 3.



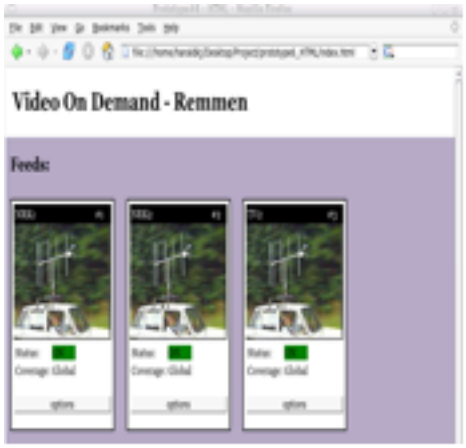
Figur 3.6: Administratorgrensesnitt prototype 4.



Figur 3.7: Administratorgrensesnitt prototype 5.

Dette ble aldri brukertestet, da utviklingsgruppen ble svimmel av å se på skjermbildet, og konkluderte med at dette var noe brukerene heller ikke ville kunne bruke.

Se skisse prototype 6 og 7.



Figur 3.8: Administratorgrensesnitt prototype 6.

To designvalg var nå tatt. Dropdownboks for kanalvalg og stillbilde/snapshot for statusinformasjon.

Oversiktsbildet skulle vise full status for alle strømmene, status og dekning (coverage) måtte fortsatt legges til. Med status menes her om strømmer sendes

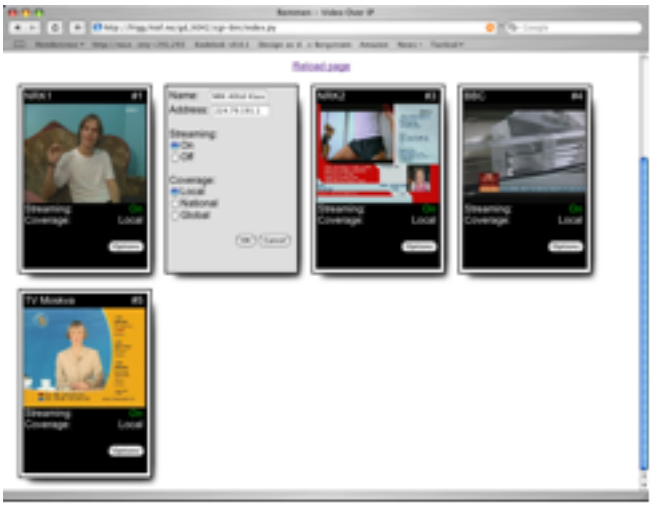
Figur 3.9: Administratorgrensesnitt prototype 7.

eller ikke, og det ble valgt en rød/grønn bakgrunn for teksten for å sikre rask og enkel oversikt.

Dekning ble i første omgang vist som den faktiske time to live/hop limit.

Kortstokkanalogien gav også en enkel løsning på hvordan forandre konfigurasjon av systemet. Når brukeren trykker på valg (options) knappen, "snus" det "kortet" rundt og "stream options" (strømvalg) vises isteden.

Her kan brukeren velge å sette kanal, multicastaddresse for livestreaming, om strømmen skal livestreames eller ikke, samt dekningsområde for strømmen.



Figur 3.10: Administratorgrensesnitt endelig design.

Designvalg

Hvorfor akkurat dette designet? Følgende retningslinjer ble brukt:

Enkel synlighet av systemstatus.
Den primære rollen til administrasjonsgrensesnittet var å raskt vise status til alle strømmer for å kontrollere at alle strømmer var operative. Sekundært og unntaksvis ville det bli brukt for å konfigurere strømmene.

Alternativene var da å vise all informasjon til enhver tid, eller begrense informasjon i overblikksbildet til akkurat den brukeren trenger for å få en oversikt over status. Da oversikt ville være den viktigste funksjonen, ble derfor all informasjon som ikke trengtes for oversikt flyttet over til selve "konfigurasjons-kortet".

Likhet mellom systemet og den virkelige verden.
Da systemet skal overta for fysiske satellitttunere og TVer, så vil det være enklere for brukeren om systemet minner om slik systemet fungerte rent fysisk, enn en komplett ny digital løsning.

Minimalt behov for brukermanual.
Dette skulle være et grensesnitt som ikke ble brukt ofte, men raskt kunne gi en enkel og god statusoversikt, samt muligheten til å forandre innstillinger enkelt. For å få til dette, ble det valgt å ha et krav om at det ikke skulle være nødvendig med en manual i designspesifikasjonen.

Brukergrensesnittet

Det første valget for brukergrensesnittet var, som for administrasjons-brukergrensesnittet, programmeringsspråk. Da administrasjonsbrukergrensesnittet og kjernemodulene allerede var implementert i Python, ville det å velge et annet programmeringsspråk, slik som PHP, medføre økt kompleksitet i forhold til gjenbruk av kode og oversikt over kildekoden, og Python ble derfor valgt til programmeringsspråk.

For brukergrensesnittet ble det ikke laget de omstendelige "use cases" som var laget for administrasjonsgrensesnittet, ei heller noen kravspesifikasjon, utover det at brukergrensesnittet skulle la brukeren reservere, ta opp og se på strømmer. Denne mer ad-hoc utviklingsmetoden ble valgt delvis for å spare tid, og delvis for å prøve alternativet til hvordan administrasjonsbrukergrensesnittet ble utviklet.

Det første problemstillingen var det overordnede design, både kodemessig og grensesnittmessig.

Kodemessig kunne en da velge å prøve å gjenbruke mest mulig kode ifra administrasjonsgrensesnittet, eller en kunne velge å starte fra bunnen opp. Dette ville være relativt avhengig av hvordan brukergrensensesnittet ble seende

ut rent grafisk, ville det likne på administrasjonsgrensesnittet, eller ville det ha et selvstendig design, basert på liknende retningslinjer og former som adminstrasjonsgrensesnittet.

Som utgangspunkt ble det da valgt å benytte seg av koden fra adminstrasjonsgrensesnittet, samt et av "kortene" fra kortstokkanalogien.

Brukergrensesnittet hadde flere oppgaver som skulle løses. Det skulle la brukeren reserve og ta opp strømmer. Det skulle la brukeren se på strømmen i realtime, og det skulle la brukeren se på tidligere opptak.

For starten av implementeringen av brukergrensesnittet var det flere muligheter. En kunne ha valgt å lage en implementasjon som tillot alle oppgavene samtidig, eller en kunne ha valgt å først løse en oppgave, for å bygge på de resterende over.

Det å lage alt samtidig ble vurdert som en mye mer kompleks fremgangsmåte enn å først løse en oppgave, og ble derfor valgt som løsning, selv om det kunne medføre visse bearbeidelser av implementeringen av den første oppgave når resten skulle bygges på.

Med disse to problemstillinger avklart, tok arbeidet med brukergrensesnittet fast. Parallelt med utviklingen av selve brukergrensesnittet ble det da arbeidet med implementeringen av reservasjonskoden i modulen kalt "ReservationLogic", dette arbeidet er beskrevet under ReservationLogic (se side 34).

Den neste problemstillingen var det grensesnittmessige design. Brukergrensesnittet hadde to primæroppgaver det skulle kunne utføre for reservering av programmer. Det måtte tillate reservering, og kunne gi en liste over de eksisterende reserveringer.

Det var her to mulige løsninger. Den ene løsningen innebar at listen over de allerede eksisterende reservasjoner ble vist på en side, mens en annen side viste grensesnittet for reservering av programmer. Det andre alternativet var å vise både grensesnittet for reservering av programmer og listen over eksisterende reservasjoner på samme nettside.

Det første alternativet var det mer tradisjonelle grensesnittdesignet. Det har en klar adskillelse mellom inntastingen av data, og resultatet av operasjonen, og gir normalt minimalt med forvirring for brukeren. Men i og med at dette var et reservasjonssystem med et lite antall plasser å reservere på, vil informasjonen om de allerede eksisterende reservasjoner være til stor hjelp for brukeren. Brukeren kan da raskt få en oversikt, og kan da med relativt sikkert forutse om brukerens reservasjon vil være mulig eller ikke.

Den typen grensesnitt som viser både operasjon og resultat på samme side, er dog ikke like vanlig for nettsider, som det adskilte alternativet, men da behovet for en forhåndsoversikt for brukeren ble vurdert til å være tungtveiende, ble alternativet med all informasjon på samme nettside valgt.

Det opprinnelige "kortet" utviklet for administrasjonsgrensesnittet viste

seg snart å være litt for lite for all den informasjon brukeren trengte å fylle inn for å foreta en reservasjon.

En reservasjon kan defineres unikt ut ifra kanalnavn, starttidspunkt og -dato, samt slutttidspunkt og -dato. I tillegg hadde brukeren et behov for å velge om reservasjon skulle være et opptak eller en live reservasjon.

En senere del av brukergrensesnittet skulle tillate søk i tidligere reservasjoner, det ble derfort vurdert slik at det var et behov for metadata om reservasjon. Brukeren fikk derfor også muligheten til å legge til en tittel og beskrivelse på reservasjonen.

Selv om da det opprinnelige "kortet" viste seg å være for lite, ble rammen gjort større, og ikke lenger en analogi for et "kort" i en kortstokk.

Det neste designproblemet lå i forholdet mellom reservasjonsboksen og listen over de eksisterende reservasjoner. Skulle reservasjonsboksen ligge på topp med resultatene under, eller skulle reservasjonsboksen og listen over reservasjoner ligge ved siden av hverandre?

Ved å ha reservasjonsboken øverst og de eksisterende resultater under ville funksjonaliteten minne om en standard søkemotor på web. Dette vil gi brukeren visse forventninger til funksjonaliteten, samtidig som brukeren vil få en gjenkjennelseseffekt av layouten. Problemet med dette var at det ikke var søkeresultater som skulle vises, men derimot data over allerede eksisterende reservasjoner.

Alternativet med at reservasjonsboksen og listen over allerede eksisterende reservasjoner var da det som ble valgt, det ble vurdert som enklere å forstå for brukeren da det ikke ville ligge noen implisitte forventninger om søkeresultater.

Den neste problemstillingen var den kodemessige løsningen for grensesnittet. Det fantes i hovedsak to forskjellige alternativer.

Det første alternativet var å stille reservasjonsboksen og listen over allerede eksisterende reservasjoner ved siden av hverandre ved å bygge de inn i en HTML ¡table¿ tag, alternativet var å få dette til ved hjelp av CSS.

Bruke HTML for å definere layout, regnes av mange som dårlig websidedesign da det blander sammen struktur og design. Fordelen med å bruke table-taggen er det å sikre at innholdet vises på nøyaktig den måten som designeren/koderen ønsker, med minimal mulighet for at nettleseren eller brukeren får forandret på dette.

CSS er laget for å kunne skille mellom struktur og design i HTML-koden, og gir det mange mener er en renere og bedre HTML-kode. CSS er mye mer avhengig av nettleseren, og flytter en del kontroll fra koderen/designeren til brukeren og nettleseren. Ved å bruke CSS kunne man også få til at listen over allerede eksisterende reservasjoner ble flyttet under reservasjonsboksen hvis brukeren skulle gjøre nettleservinduet tilstrekkelig smalt.

Større fleksibilitet for brukeren, samt "riktig" bruk av HTML tagger ble vurdert til mer tungtveiende enn muligheten for milimeterdesign av grensesnittet, og ble således valgt som løsning.

Videre skulle designet av selve listen over allerede eksisterende reservasjoner lages. Dette var av natur en relativt lang eller kort liste over data, og den tidligere forkastede ¡table¿ taggen ville her kunne gjøre den jobben den er designet for; vise informasjon i tabellarisk form.

Men hvilken informasjon skulle gis til brukeren? For alle reservasjoner var den følgende informasjonen tilgjengelig: tunernummer, kanalnavn, starttid, sluttid, brukernavn, når reservasjonen ble foretatt, reservasjonstittel og reservasjonskommentarer.

Den informasjonen som brukeren trenger, er å få en rask oversikt over eksisterende reservasjoner slik at brukeren kan unngå å prøve å reservere på et tidspunkt hvor det allerede finnes reservasjoner på samtlige tunere.

Den første biten av informasjon som brukeren da trenger, ble da vurdert til å være dato og klokkeslett en reservasjon startet på. Listen ble også sortert etter denne opplysningen, med de reservasjonene nærmest i tid satt øverst i lista.

Da en reservasjon ble satt til å ha en maks varighet på fem timer under utviklingen av selve reservasjons-backenden, ReservationLogic-modulen, kunne reservert til begrenses til å være klokkeslettet reservasjonen slutter på.

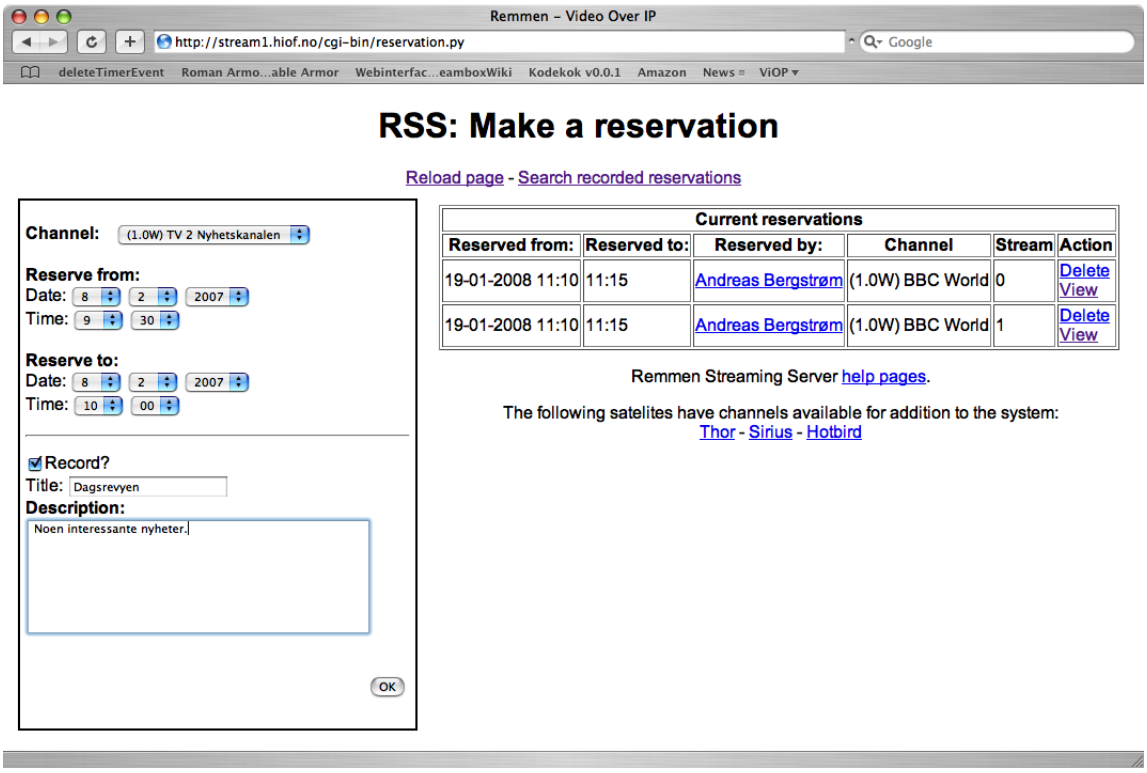
Skulle det da oppstå kollisjoner, eller en bruker finner ut at en annen bruker har reservert veldig mye kolliderende i en periode, ble det vurdert til at muligheten for å kunne kontakte denne andre brukeren burde være tilstede. Med utgangspunkt i brukernavnet som blir lagret ved en reservasjon, slår grensesnittet opp fullt navn og e-post adresse i institusjonens LDAP database og lager en e-post link som lar brukeren kontakte den som har foretatt reservasjonen.

Et annet scenario som kunne tenkes er at flere brukere vil ta opp samme kanal på samme tidspunkt for samme program, kanalnavnet for reservasjonen ble tatt med.

Det ville også være interessant for brukeren å se om det er noen tilgjengelige tunere i det tidsrommet brukeren ønsker å reservere, tunernummeret ble tatt med. Internt i systemet nummereres dog tunerene med 0 som første tuner, noe som er helt vanlig internt i programmer. Skulle da brukeren få informasjonen som om tunerene ble nummerert fra en, eller fra null?

Hvis nummereringen internt i systemet og det som ble vist til brukeren skulle være forskjellige, ville muligheten for off-by-one feil være sterkt tilstede under debugging når brukeren melder inn feil. Det ble derfor vurdert dithen at det ville gi mindre forvirring å vise brukeren en teller som begynte på null enn å øke alle verdier med en for å gi en nummerering som brukeren var mer vant til.

Men hva nå hvis brukeren skulle ønske å slette sin reservasjon? En siste kolonne ble lagt til brukergrensesnittet for å utføre en handling. For alle de reservasjoner som brukeren selv har lagt inn har dermed brukeren en mulighet til å slette reservasjonen, mens brukeren ikke kan slette reservasjoner som tilhører andre brukere.



Figur 3.11: Skjermbilde av reservasjonsgrensesnittet.

Med selve reservasjonsgrensesnittet ferdig designet og klar for brukertesting, var tiden moden for å utvikle søkegrensesnittet.

Også for denne delen av brukergrensesnittet var den første problemstillingen det overordnede grafiske design.

Ved hjelp av dette grensesnittet skulle brukerene kunne søke igjennom de tidligere reservasjoner der hvor det hadde blitt foretatt opptak, for å kunne spille av det opptaket som brukeren ønsker å se.

Som for reservasjonsgrensesnittet fantes det her primært to muligheter for designet av grensesnittet.

En kunne velge å benytte seg av det eksisterende grensesnittet for reservasjon, men tilpasse det for søk i opptakene. Fordelene med denne løsningen var at brukeren da ville kjenne seg igjen fra da reservasjonen ble gjort, problemet var

at grensesnittet ikke minnet om tradisjonelle websøkgrensenitt.

Alternativet var å lage et mer tradisjonelt websøkgrensesnitt med et søkefelt på toppen og vise resultatene under. Fordelen med dette alternativet var det at dette var funksjonalitet brukeren var vant til, men en slik løsning ville da bli annerledes enn reservasjonsgrensesnittet, og kanskje litt vanskeligere å søke i den spesifikke datamengden som opptak er.

Med hovedvekt på gjenkjennelseseffekten fra det eksisterende reservasjonsgrensesnittet og muligheten for raskere utvikling av søkefunksjonaliteten vis a vis applikasjonsmodulen som utfører søket, ble et design som minnet om reservasjonsgrensesnittet valgt.

Søkeboksen ble dermed designet helt likt reservasjonsboksen, slik at brukeren skulle mest mulig kunne kjenne seg igjen, men samtidig ha muligheten til å lett kunne søke i alle attributter.

Den siste designutfordringen lå i hvilken informasjon som søkevinduet skulle vise.

Etter en nøye vurdering ble konklusjonen at den viktigste informasjonen om et opptak var når opptaket ble gjort, når den ble reservert fra og når den ble reservert til.

Kanalen som opptaket ble gjort på ble også vurdert til å være nødvendig informasjon, sammen med navnet som ble gitt reservasjonen.

Beskrivelsen av opptaket ble også vurdert som nødvendig informasjon og lagt til.

Til sist ble applikasjonsmodulen brukt til å gi en last ned link. Direkte streaming ble vurdert, men krevde et annet videoformat enn det som opptakene blir gjort i, det ble derfor ingen “stream” link.

Som standard visning ble det valgt å vise de femten nyeste opptakene i databasen.

Kjernemodulen ApplicationLogic

Designet for Remmen Streaming Server forutsatte en distribuert løsning. Systemet måtte spres over flere maskiner, og det ville da trenges et sentralt kontaktpunkt for de forskjellige modulene.

I det opprinnelige designet skulle systemet bruke flere standard DVB satellitttunere, og enkodes digitalt over flere maskiner.

ApplicationLogics hovedoppgave var derfor å være det sentrale samlingspunkt i Remmen Streaming Server.

Den første designutfordringen for ApplicationLogic var på hvilket grunnlag

skulle APIet for modulene rettet mot brukerene defineres. Det fantes her to muligheter. Den første muligheten lå i å designe ApplicationLogic frittstående utifra hva Remmen Streaming Server som system skulle gjøre. Slikt som som brukergrensesnitt og reservasjonssystem ville derfor måtte tilpasses dette APIet, uten muligheter for å påvirke designet av APIet. Det andre alternativet var å designe APIet og ApplicationLogic ut i fra de behov som brukergrensesnitt og reservasjonssystem måtte ha.

Da adminstrasjonsgrensesnittet ble utviklet samtidig som ApplicationLogic, og jobben til ApplicationLogic var å være knutepunkt for systemet, ble det vurdert slik at det var bedre å tilpasse ApplicationLogics funksjonalitet og API til brukergrensesnittet enn omvendt.

Remmen Streaming Server skulle være et distribuert, modulbasert system. På bakgrunn av dette ble det vurdert til at en objektorientert fremgangsmåte ville være den optimale for ApplicationLogic.

En av ApplicationLogics oppgaver ville være å holde oversikten over de kanaler og strømmer som RStS kunne tilby.

En kanal i Remmen Streaming Server er definert som en fysisk TV/radio kanal, som på en vanlig satellitttuner, for eksempel NRK 1 og NRK 2.

En strøm er en logisk enhet som omfatter en streamingenhet. Dette vil si en satellitttuner, videodigitalisering, opptak og kringkasting av en kanal.

Det var derfor naturlig å starte med å lage to klasser, en for kanalinformasjon og en for strøminformasjon.

Videre var det raskt klart fra administrasjonsgrensesnittutviklerenes side at vi trengte funksjonalitet for å starte og stoppe strømmer, få og sette informasjon om strømmer, få og sette kanalinformasjon, og å hente ut strømstatus.

Et designspørsmål her var om en skulle bruke binære programkall for slikt som å starte og stoppe en strøm, eller om en skulle lage to programkall, en for å starte, og en for å stoppe strømmene.

Fordelen med et binært programkall for både å starte og stoppe strømmer er et mindre antall programkall, og en strukturell "renere" løsning, men en rask overfladisk lesning av kildekoden vil det ikke umiddelbart være klart om et programkall skal starte eller stoppe en strøm. Med to programkall, blir det totalt sett flere programkall, litt mindre strukturelt "ren" kode, men en rask skumlesning av koden vil umiddelbart vise om kallet skal starte eller stoppe strømmen.

Bruken av separate start og stopp kall er relativt vanlig, og når det i tillegg gav muligheten for å raskere få en enkel oversikt over koden når en skumleser den, ble alternativet med start og stoppkall valgt.

Den neste utfordringen var lagring av programdata mellom sesjoner, og lagring av kanal- og strøminformasjon.

ApplicationLogic skulle lagre informasjon om strømmer og deres tilhørende kanaler, spesifikk statusinformasjon for strømmene. Denne informasjon måtte ikke gå tapt selv om ApplicationLogic skulle krasje.

Det fantes flere muligheter for lagring av disse dataene.

Den første var bruken av en flat tekstfil med all informasjon lagret i et semikolonseparert format. Dette ville være enkelt menneskelesbart, men ville kreve reparsing av klassene ved oppdatering, og en del ekstra kode for å skrive det ut og inn av ApplicationLogic.

Det andre alternativet var bruken av et XML-format, både internt i programmet og for lagring eksternt. Dette ville kreve mindre koding av parsing, men ville kreve en del flere ressurser enn å pakke informasjon inn i objekter definert av klasser i Python.

Det tredje alternativet var å lagre selve objektenes binærdata direkte til disk. Python har en modul som heter pickle, som lar en skrive et pythonobjekt til disk, for å senere hente det opp fra disk. Dette var raskt og enkelt, krevde minimalt med programmering, og gav størst effektivitet internt i programmet.

XML ville gitt en ren, menneskelesbar løsning, med et minimum av koding sett i forhold til en flat tekstfil. Allikevel ble det siste alternativet valgt, da den antatte høyere effektiviteten fra å holde informasjonen i pythonklasser istedenfor et pythoninterpretert XML-tre ble vurdert som bedre for prosjektet, samtidig som det minimaliserte kode-arbeidet for lagring av statusdata.

Det neste skrittet i implementeringen var kommunikasjon mot den bakenforliggende ApplicationManager. Selv om XML-RPC var satt som kommunikasjonsprotokoll, ønsker oppdragsgiver noe enklere, flere alternativer ble prøvd ut.

Det første alternativet var bruk av de totalt åpne, ukrypterte og relativt usikre r-kommandoene under Linux/Unix for å kommunisere med ApplicationManager. Dette vil være raskt og enkelt, men med behov for å kalle eksterne programmer, og de er også ekstremt usikre

Det andre alternativet var bruken av telnet protokollen. Dette er noe sikrere enn r-kommandoene da det krever brukernavn og passord, men det sendes fortsatt ukryptert over nettet.

Det tredje alternativet som ble vurdert var bruken av SSH. SSH muliggjorde kryptering av all trafikk, samt bruker- og passordautentisering.

Det fjerde alternativet var bruken av XML-RPC. XML-RPC gjør det veldig enkelt å programmere distribuert, da hver server kan pakkes inn i et objekt og programkallene kan kjøres som om det var til et objekt som eksisterte kun lokalt i programmet. Oppdragsgiver syntes midlertidig at dette ble for komplisert.

Den første versjonen brukte derfor SSH for kommunikasjon med

ApplicationManager.

Etter å ha ha implementert funksjonaliteten for start- stopping av strømmer ut i fra administrasjonsbrukergrensesnittets behov, kjørte vi en første runde med brukertesting av selve administrasjonsbrukergrensesnittet.

Dette viste seg fort å være et veldig tregt grensesnitt som gav brukeren god tid til å drikke kaffe mellom hver gang programmet svarte. Etter litt debugging, viste det seg at ssh-kallene brukte veldig lang tid når de ble kallet fra Python.

Denne forsinkelsen mellom hvert kall til ApplicationManager ble fra testbrukerenes side vurdert som for langt, noe som prosjektgruppa sa seg enige om.

ApplicationManager ble reimplementert som et Python program basert på XML-RPC, og ApplicationLogic forandret sine kall til ApplicationManager til å basere seg på XML-RPC isteden.

En annen problemstilling for ApplicationLogic var hvorvidt den skulle multi-threades eller ikke. En mulighet som ble vurdert, og testet, var hvorvidt kommunikasjonen for hver enkelt strøm skulle kjøres gjennom sin egen tråd. Dette viste seg dog å ikke være nødvendig, det viste seg å ikke være et problem med samtidig kommunikasjon om flere strømmer, samt at all informasjon som ikke lå i ApplicationManager, lå mer effektivt i den sentrale ApplicationLogic-tråden.

Multithreading og Python kan også være en dårlig kombinasjon, da det å stoppe en multithreaded Python prosess krever veldig eksakt koding, av en type som ikke alltid er mulig i en applikasjon. ApplicationLogic ble derfor singlethreaded. (Dog multithreaded i den forstand at XML-RPC modulen kan ta imot flere requests samtidig.)

Senere utvikling av koden viste seg å gi et behov for en tråd som sjekket om streaming var oppe, og som kunne restarte prosesser når en backendmaskin hadde restartet. Klasses CheckApplicationManager ble opprettet for å sjekke status mot backendmaskiner og restarte streaming når den var av, men skulle være på, og omvendt.

Denne klassen ble først satt inn som en ekstra tråd i ApplicationLogic, men ble senere skilt ut som en separat modul for et renere design, og for å enklere terminere ApplicationLogic grunnet problemet med Python, multithreading og terminering av prosesser.

Et streamingsystem for satellittkanaler trenger en metode for å velge kanal på. Dette var den neste store utfordringen. De fleste tilgjengelige satellitttunere var ikke designet for å kunne styres fra en PC, og da en fant noen som tilbød dette, ble de funnet for dyre for prosjektets oppdragsgiver.

En måtte altså finne en måte å fjernstyre tunere på. Det ble lagt opp til at det skulle være seks tunere fra oppdragsgivers side.

Etter en del søk på internett ble IRTrans IR [10] sender funnet. Dette var en USB boks med muligheter for to IRsendere, samt en innebygget IR mottaker. Tre slike bokser ble bestilt for å fjernstyre satellitttunerene.

Dette gjorde det mulig, gjennom ett enkelt kommandolinjekall å fjernstyre de forskjellige tunerene. API kall for dette ble derfor bygget inn i ApplicationLogic.

Spørsmålet var, hvilke muligheter skulle brukeren ha når det gjaldt styring av boksen? Muligheten for å taste inn sifferene 0 - 9 ble sett på som en selvfølge, samt muligheten for å kjøre én kanal opp eller ned, samt volum opp eller ned. Litt videre testing viste at det mest sannsynlig dekket det behovet systemet hadde for fjernstyring av DVB tunerene.

ApplicationLogic modulen var nå klar til bruk, og arbeidet med ReservationLogic for reservasjonskoden og VODLogic for opptakskoden ble påbegynt.

Bruken av irsender viste seg å fungere godt, og kanalbytte fungerte som det skulle. Opptak ble gjort på bakgrunn av livestreamingen, og fungerte, men ikke optimalt.

I et forsøk på å gjøre systemet enklere, samt å fremskynde en tidlig betatesting, ble en satellitttuner kalt DreamBox [11] tatt i bruk.

DreamBox er en satelitttuner basert på PC-teknologi. Den kjører et linuxbasert operativsystem og kommer med full nettverksstøtte, samt muligheten til å installere en harddisk. Dette gjorde det mulig å fjernstyre DreamBoxen over nettverket, og den hadde innebygget opptaksskeduleringssytem, noe som gjorde opptaksdelen til RStS betraktelig mer pålitelig og kodemessig enklere.

ApplicationLogic ble da splittet opp i TunerLogic og DreamBoxLogic, TunerLogic ble lagt til side, og DreamBoxLogic ble utviklet videre for bruk av DreamBoxer som satellitttunere.

DreamBox har et ikke fullt ut offisielt dokumentert HTTP basert skriptsystem som tillater enkelt å fjernstyre tuneren ved sende argumenter til webscript. Heldigvis har DreamBox dog en god del uoffisiell dokumentasjon da det også er et open source program, og etter noe prøving og feiling var funksjonaliteten for kommunikasjon med DreamBoxen for opptak og kanalbytte implementert.

Applikasjonsmodulene

Applikasjonsmodul - ApplicationManager

Da RStS var laget for å være et distribuert system grunnet behovet for flere enkodermaskiner, trengtes det en kontrollapplikasjon per enkodernode.

I første versjon var ApplicationManager et shellskript som startet VLC prosesser, og som ApplicationLogic startet over SSH. Dette viste seg fort å

bli for tregt, og det ble da valgt å bruke XML-RPC som i resten av systemet.

Funksjonaliteten som ApplicationManager trengte var i stor grad styrt av ApplicationLogic. Det skulle kjøre en ApplicationManager per streamingnode, og den skulle kunne starte og stoppe strømmene, samt rapportere på strømstatus.

En problemstilling lå i hvordan start-kallene skulle utformes. Et alternativ var å sette informasjon for de enkelte strømmene i et SetStreamData kall, og bare ha behov for å kalle StartStream() når en strøm skulle startes. Det andre alternativet var å gi alle disse variablene når en strøm skulle startes, og dermed ikke har noen lokal cache over strøminformasjonen.

RStS var designet for å være modulært på en slik måte at enkeltmodulene skulle fortsette å kjøre selv om andre moduler hadde gått ned, samtidig skulle streamingnodene være tynne som mulig, slik at en tjenesteovervåkningsmodul, CheckApplicationStreamer ble implementert på servernoden. Det var derfor naturlig å ikke lagre strøminfo i ApplicationManager og derimot gi all nødvendig informasjon til startkallet.

I første iterasjon skulle ApplicationManager styre VLC prosesser som tok en digitalisert lyd/bilde strøm fra et Hauppauge PVR-250 kort, og multicastet dette ut på nettet. Rekkevidden til multicastingen ble satt i administrasjons-brukergrensesnittet.

Når de vanlige satellitttunerene ble byttet ut med DreamBoxer, var det ikke lenger nødvendig å re-digitalisere kanalene og deres lyd/bilde strømmer, da dette kunne hentes ut av DreamBoxen. ApplicationManager ble derfor raskt konvertert til å hente strømmene fra DreamBoxene for å multicaste disse ut på nettet.

Det ble her vurdert hvorvidt ApplicationManager skulle fullstendig baseres på DreamBoxstreaming, eller om den også skulle kunne støtte streaming fra digitaliseringskort slik som Hauppauge PVR-250 kortene.

Da valget av DreamBox over vanlige satellitttunere enda ikke var endelig, valgte man å kunne støtte begge alternativ. Det ble derfor implementert støtte for multicasting av videostreamen fra en DreamBox i ApplicationManager som kunne sameksistere med den opprinnelige streamingen av digitalisert videostrøm fra Hauppauge PVR kortene.

Selv om DreamBoxene i utgangspunktet så ut til å være ideelle for oppgaven, viste de seg etter hvert å kunne passe bedre for sitt egentlige design, et enbruker satellittsystem. Den digitale videostrømmen som DreamBoxene sendte ut kunne bare leses av en bruker av gangen, og opptaksfunksjonaliteten i DreamBoxen fungerte her som en bruker. Det var dermed ikke mulig å se på en strøm og ta den opp samtidig, da to samtidige brukere av den digitale strømmen av ukjente grunner korrumperte strømmen for samtlige brukere.

Det fantes her flere mulige løsninger på dette problemet.

Den første mulige løsningen var å redefinere funksjonaliteten slik at en

enten så på en strøm, eller tok den opp. Dette ville omgå problemet med korrumpering, men denne typen funksjonalitet ble vurdert til å ikke være lett forståelig for sluttbrukeren. Det kunne også tenkes at en isteden økte antallet DreamBoxer slik at en kunne reservere enten opptak, livestreaming eller livestreaming og opptak slik at en benyttet seg av to DreamBoxer når behovet for livestreaming og samtidig opptak meldte seg.

Det andre alternativet var å forsøke å oppdatere koden i selve DreamBoxen. Den kjører en optimalisert og spesialisert utgave av linuxoperativsystemet, og kunne derfor skreddersys ved behov. Programmet som benyttes internt i boksen for streaming, og mest sannsynlig også for opptak er VLC, som normalt ikke har korrumpsjonsproblemer ved flere samtidige brukere. Dette ville kreve noe arbeid med Dreamboxens meget dårlig dokumenterte API, samt gjøre oppgradering av boksens "firmware"/os mye vanskeligere på sikt.

Det tredje alternativet var å gå tilbake til redigitalisering av livestreamingen ved hjelp av Hauppauge PVR kortene. Koden for denne typen livestreaming var skrevet for analoge tunere og hardwaren kjøpt inn. Redigitalisering ville medføre et visst kvalitetstap, men med den planlagte båndbreddebruken så ble det vurdert atkvalitetstapet ikke ville være merkbart eller synlig.

Det første alternativet gav en løsningen som krevde minimalt med implementasjon og videre testing. Da reservasjonssystemet allerede krevde enten opptak eller livestreaming, kunne en raskt ha oppdatert reserveringskoden til å reservere to eller ingen tunere hvis en bruker ba om opptak og livestreaming. For resten av systemet ville ikke denne forandringen ha noen betydning, da hver tuner måtte behandles separat. Det andre alternativet var implementasjons-tungt. Det ville være nødvendig å sette seg inn i DreamBox APIet, samt VLC APIet for å forandre koden, skrevet i C, slik at korrupsjon ikke lenger forekom når multiple klienter ba om den digitale strømmen. Det tredje alternativet krevde minimalt med implementasjon, da koden allerede var skrevet, og også testet en del, men ville krever noe videre brukertesting for å få alt til å virke etter spesifikasjonene.

Da prosjekter på dette tidspunktet lå bak skjema, ble tidsbruk vurdert som en viktig faktor, funksjonalitet ble vurdert som den sekundære faktor. Dette eliminerte umiddelbart alternativ 2. Alternativ 1 og 3 ble vurdert til å kreve tilnærmet samme tid for implementering, men alternativ 3 ville kreve noe mer testing. Alternativ 3 tilbød samme funksjonalitet som alternativ 1, men med færre DreamBoxer. Da oppdragsgiver ønsket å minimalisere mengen av maskiner i datamaskinracket ble alternativ 3 valgt, og ApplicationManagers kode ble rullet tilbake til siste versjon før implementeringen av DreamBox funksjonaliteten ble lagt til.

Applikasjonsmodul - ReservationLogic

Da funksjonaliteten for grensesnitt, kjernemoduler og livestreaming var implementert, var det neste skrittet i prosessen med RStS muligheten for reservasjoner av strømmer for livestreaming eller opptak.

Den første problemstillingen var her; hvordan skulle reservasjonsinformasjonen lagres. Det fantes mange alternativer.

Det tradisjonelt enkleste i kodekompleksitet var en flat tekstfil, tokenseparert som inneholdt all informasjon. En tokenseparert liste kan enkelt dokumenteres og implementeres i de fleste programmeringsspråk, den ville også være menneskelesbar. Problemet med en tokenseparert flat tekstfil er at den er ikke optimalisert for søking og ville bli eksponentielt tregere jo større databasen ble. I tillegg måtte all søk, insert og remove funksjonalitet implementeres fra bunnen av.

Alternativ to var å lagre informasjonen i en Pythonstruktur, slik som konfigurasjonsinformasjonen for ApplicationLogic. Dette ville enkelt og raskt kunne implementeres, men ville ha samme problemer med store databaser og søk som den flate tekstfila.

Det tredje alternativet var å bruke en SQLbasert database, slik som PostgreSQL. Det eksisterer Python-PostgreSQL interface som gjorde det like enkelt å implementere reservasjonsdatabasen som en fullstendig SQL database som bruken av pickle og pythonstrukturer. PostgreSQL har i tillegg god funksjonalitet for manipulering og søk i dato-data og store databaser, og dette gav større skalerbarhet og enklere implementasjon av dato-søkefunksjonaliteten som de litt mer avanserte søkene i reservasjoner gav et behov for.

Behovet for god dato søke- og håndteringsfunksjonalitet, samt skalerbarhet for databasen ble vurdert som mer tungtveiende enn en kodemessige enklere implementasjon av en flat database, I tillegg hadde vi allerede god erfaring med PostgreSQL og dets Python interface.

Databasen

Den neste utfordringen var å designe databasen som reserveringsinformasjon skulle lagres i, og databasens struktur.

Det første og viktigste attributtet til en relasjonsdatabase er primærnøkkelen som unikt identifiserer en databaseentry.

Det fantes to alternativer for en primærnøkkel:

Det første alternativet var å bruke en teller internt i databasen som økte med en for hver databaseentry som ble lagt til, og som kun ble brukt internt i databasen for å skille mellom entries. Ulempen med dette var at det genererte ekstra data som det egentlig ikke var behov for.

Det andre alternativet var å bruke en eller flere felt i reservasjonsdataene som primærnøkkel. Hvis det finnes noe i selve dataene som unikt identifiserte hver reservasjon, ville databasen ikke behøve å lagre ekstra informasjon, og den kunne raskt gi tilbakemelding om reservasjonen var mulig.

Konklusjonen var at strømid, dvs. tuneren som reservasjonen skal foretas på, sammen med start og slutt-tidspunkt unikt identifiserte en reservasjon, ble alternativ to valgt, og strømid, sammen med start- og slutt-tidspunkt ble satt som primærnøkkel.

I første iterasjon av streamingkoden ble det benyttet tunere istedenfor DreamBoxer, et felt med kanalnavn var nødvendig for å tune tuneren til den riktige kanalen.

I tillegg ble det vurdert som fornuftig å lagre brukernavnet til den som foretok reservasjon, dette gjorde det mulig å kontakte personen hvis den skulle reservere for mye i et tidsrom, eller skulle finne på å reservere et upassende program.

For å dokumentere når ting ble reservert, og kunne sjekke dette på senere tidspunkt, ble også tidspunktet hvor reservasjonen ble foretatt lagt til i databasen.

I tillegg ble det lagt til et booleansk flagg for å markere om det skulle tas opp eller kun reserveres for live streaming, samt to felt for tittel på reservasjonen, samt notater om reservasjonen.

Det trengtes også et system for å validere reservasjonsdata mot tidsrommet reservasjonen ble foretatt i. Her viste det å velge en SQL databasemotor seg å være en styrke.

En kunne enkelt i databasen legge til beskrankninger, begrensninger i hvilke data som var gyldige, for å sikre at dataene som ble lagt til var passende for RStS.

Det første restriksjon (beskrankning) som ble lagt til var en sjekk som sikret at både ønsket reservasjonsstart og -slutt tid var i fremtiden, samt en sjekk for å sikre at sluttiden kom etter starttiden. Disse kontroller av dataene ble lagt på databasenivå istedenfor programnivå da det var mye raskere å implementere de i PostgreSQL datasen enn i Python.

Etter noe brukertesting ble det oppdaget at designet tillot reserveringer å løpe over flere år. Dette var ikke tilsiktet, og en restriksjon ble lagt til i databasen for å sikre at ingen reservasjoner kunne være lenger enn fem timer. Fem timer ble vurdert som mer enn langt nok til å ikke forstyrre vanlige reservasjoner, da de aller færreste TV-programmer varer lenger enn 2 timer. Skulle det være nødvendig, kan flere konsekutive reserveringer reserveres, men ville kreve merarbeid skulle det være ønskelig å misbruke reservasjonsmuligheten.

Koden

I sin første iterasjon skulle ReservationLogic gi funksjonaliteten for reservering av opptak og livestrømmer, samt sørge for at reservasjoner ble gjeninnført på de reserverte tidene.

Et valg for å sikre dette var å dele koden i modulen i to. En tråd som

arbeidet med å sjekke om det var noen reservasjoner de to neste minutter, og en annen tråd som svarte på XML-RPC forespørsler om reserveringer.

Tråden som sjekket om det var noen reservasjoner sjekket hvert hele minutt om det var noen reservasjoner, og brukte ApplicationLogics API for å bytte kanal og eventuelt starte et opptak, alternativt avslutte et kjørende opptak.

Hvor ofte det skulle sjekkes om det var noen reservasjoner var en problemstilling, men hvert minutt ble vurdert til å være ofte nok. Det var en mulighet for at brukeren reserverte noe som skulle tas opp med en gang, men denne problemstillingen var ikke relevant, da det var et kravat reservasjoner måtte starte minimum fem minutter frem i tid. Dette gav også en ekstra sikkerhet i tilfelle at reservasjonsmaskinen og opptaksenheten skulle få tidssynkroniseringsproblemer.

Det ble også lagt til et API-kall i ReservationLogic som lot brukeren sjekke om noen opptak eller reservasjoner skulle starte det neste minuttet, selv om det fantes et mer generelt kall som også kunne brukes. Selv om dette ga noe unødvendig kode, ble det vurdert som nødvendig, det det forenklet koding, og gjorde det mulig å spesialtilpasse koden for opptakssjekking skulle det bli nødvendig.

I den neste iterasjon av ReservationLogic ble den branchet ut til DreamBoxReservationLogic, som ble tilpasset DreamBoxene for opptak. Dette gav en del fordeler over den forrige opptaksløsningen:

I den opprinnelige løsningen ble bildet fra tunerene digitalisert og multicastet ved hjelp av ApplicationManager og VLC til enhver tid. Opptakskoden skulle når opptak var reservert starte en ny VLC prosess som tok opp multicaststrømmen og lagret den til disk. Dette gav muligheten for noe pakketap i opptaket, da multicast må benytte UDP, og garanterer dermed ikke for at alle pakker når frem. En annen modul, VODLogic, ble også skrevet for å håndtere henting av disse opptakene.

Det at DreamBoxene overtok opptaksjobben gjorde det enklere. Det var nå mulig å ta opp reservasjonen ved å dumpe den digitale sendingen direkte til disk, noe som reduserte mulig pakketap til å være de som skapes av atmosfæriske forstyrrelser. DVB standarden har bygd inn feilkorrigering, som reduserte problemer skapt av atmosfæriske forstyrrelser til et minimum.

En DreamBox kan også bruke tidssignalet i en TV-strøm til tidssynkronisering, noe som kan sikrer tidssynkronisering av selve opptaksenheten. Videre ble selve kodejobben betraktelig forenklet da DreamBoxen har all den funksjonalitet som trengs innebygd, både for reservering av opptak og reservering av kanal for visning, noe som gav en ekstra mulighet for feilsjekking av input.

Med dette, var ReservationLogic klar for videre bruk og testing.

3.2 Metoder

Grensesnittdesign

Deler av RStS ble utviklet som en del av et gruppeprosjekt i kurset Grensesnittdesign ved HiØ. I den forbindelse ble det benyttet metodikk for designet brukergrensesnittet til RStS.

For å skape en systematisk oversikt over hva adminstrasjons- og brukergrensesnittene skulle gi brukeren, ble en rekke use cases (se vedlagt CD, project.report.doc) laget. Disse dokumenterte oppgavene som skulle løses, hvordan de skulle løses og resultatet av en slik utført oppgave.

Disse ble så brukt når grensesnittene var ferdig kodet for å verifisere at funksjonaliteten som opprinnelig tenkt var med, eller dokumentert forandret der det viste seg nødvendig.

Videre ble det laget et systemdesigndokument som gav grunnlaget for designet av de forskjellige moduler i RStS som beskrevet under gjennføring. Dette dokument beskrev hvordan modulene skulle samhanadle, og gav også en konseptuell skisse av modulene.

Et foreløbig programdesign med beskrivelse av funksjonskall for APIet ble også laget som en del av denne prosessen.

På bakgrunn av disse dokumentene ble så arbeidet med utviklingen av RStS påbegynt som beskrevet i forrige kapitel.

Del III

Diskusjon

Kapittel 4

Muligheter og hindringer

4.1 Innledende - Muligheter i HiØs lokalnett

Nettverksinfrastrukturen har for ethernettbaserte nettverkene hatt en stødig utvikling fra 10 mbps koaksialbaserte nettverk til dagens 1 gbps switchede nettverk. 10 mbit koaksialnettverkene og 10 mbit hub-baserte nettverkene hadde 10 mbit som en teoretisk øvre hastighet, men hadde store problemer med kollisjoner og svekket ytelse grunnet kollisjoner og med det, behovet for å sende data på nytt.

Med switchede nettverk ble 10 mbps en mer sannsynlig overføringshastighet for nettverksnodene, da switchene skapte mange små en til en virtuelle nettverk, som sammen med full dupleks tillot begge noder å sende i opptil 10 mbps uten den samme kollisjonsproblematikken.

Switchede nettverk ble dog ikke vanlig før skiftet til 100 mbps, noe som er dagens standard for de fleste sluttbrukernettverk.

Uninett [12] har som mål å få 1000 mbps/1 gbps på alle høgskole- og universitetsnett, dette programmet kalles Gigacampus [13]. Gjennom Remmenutbyggingen og totalrenoveringen av den eksisterende bygningsmassen på Remmen, var også muligheten tilstede for å basere HiØs nettinfrastruktur på remmen på 1 gbps. Dette skapte en del muligheter som et 100 mbps nettverk ikke har.

Switchteknologi og kabling tillater 1 gbps full dupleks kommunikasjon mellom sluttnode og sluttnode.

HiØ har, i likhet med alle andre Uninett institusjoner også full multikaststøtte i sine nett.

Kombinasjonen av høyere generell overføringshastighet, høy ende-til-ende

overføringshastighet og lav risiko for nedsatt overføringshastighet grunnet kollisjoner/"network congestion" gir noen veldig spennende muligheter.

Sanntidsvideo

Streaming

Sanntidsvideo i kringkastingskvalitet krever en god del båndbredde. Den mest brukte standarden for dette er MPEG2 for DVB [14] og DVD [15]. For DVB brukes også H.264/AVC [16] i økende grad grunnet mindre båndbreddebehov for same kvalitet.

For fullkvalitets sanntidsvideo krever MPEG-2 10 mbps (DVD-kvalitet), mens DVB krever 5 mbps for MPEG 2 og 2.5 for H.264/AVC, tilsvarende for HDTV er 10 mbps MPEG2 og 5 mbps H.264/AVC.

Som eksempel vil det her tas utgangspunkt i streaming av fullkvalitet HDTV tvideo etter DVB standarden på lokalnettet. For en bruker vil dette innebære fem eller ti Mbps båndbredde mellom videoserver og brukeren. Det forutsettes også at det ikke brukes feilkorrigering utover det som allerede finnes i videosignalet.

I For et 10 mbps nettverk vil MPEG-2 ikke være et alternativ, da all annen trafikk vil føre til degradering av videokvaliteten. En 5 mbps H.264 strøm kan støttes frem til en bruker.

II For et førstegenerasjons switched 100 mbps nettverk vil den totale datamengden som en switch kan sende være 100 mbps totalt, og ikke 100 mbps per port. Hvis en ta kunne benytte denne båndbredden fullt ut, vil det være mulig for å sende ti 10 mbps MPEG-2 strømmer eller alternativt 20 H.264 5 mbps strømmer. Da en må gå ut ifra at annen trafikk også vil gå over nettverket, vil en i praksis kunne bruke maksimalt en 80 til 90 mbps, noe som reduserer antall tilgjengelige strømmer.

III For et switched 1000 mbps nettverk kan hver enkel port takle 1000 mbps trafikk. Det vil da være båndbredden ut ifra server som er definerer begrensningen for antall strømmer. Ved full utnyttelse av båndbredden vil en da kunne sendet 100 10 mbps strømmer eller 200 5 mbps H.264 strømmer, men sett bort ifra serveren, vil de enkelte klienter fortsatt ha 995 mbps tilgjengelig båndbredde. En må dog gå ut ifra at serveren vil ha noe annen trafikk, i praksis vil rundt 95% av den tilgjengelige båndbredden kunne utnyttes.

Diskusjon - Remmen leveringsmodell

Problemet med modellen beskrevet her, basert på unicastmodellen, er skalerbarhet.

Utviklingen fra 10 mbps nettverk til 1000 mbps nettverk har muliggjort sendingen av høykvalitetsvideo over et datanettverk, men det finnes flere problemer, som hindrer storskalabruk i forbindelse med tilgjengelig båndbredde.

Unicastmodellen er en videreføring av telefonlinjemodellen, hvor det er en dedikert forbindelse mellom sender og mottaker. Over et pakkesvitsjet nettverk er dette riktignok en virtuell dedikert forbindelse, men likefullt er det et behov for en dedikert menge båndbredde for alle punkter mellom tjeneren og serveren, og denne mengden båndbredde må være tilgjengelig under hele sesjonen for å unngå forstyrrelser i sanntidsdata.

Hva er konsekvensene av at båndbredden ikke er tilgjengelig under kortere eller lengre perioder?

Overføringsprotokoller for unicast og klientsiden er basert rundt tre muligheter:

Den første og mest vanlige er å buffre en femten til tjue sekunder med video, rør å avspilling startes. Fordelen med denne metoden er at den tåler at det er noe ustabilitet i selve nettverket uten at det påvirker avspilling av materialet, problemet er at blir ustabilitetene for store, vil videoen stoppe opp og en må re-buffre. Dette gir ikke en spesielt god opplevelse for sluttbrukeren. En kan jo tenke seg her at en bruker prioritering i nettverket for å dempe ustabiliteter og sikre båndbredden mellom klient og tjener, for eldre og mye av dagens nettverkinfrastruktur er dette dog et problem da prioritering av pakker krever ekstra prosessering av alle pakker, og denne prosessorkraften har ikke alltid vært tilgjengelig, slik at nettverket som helhet kan risikere å få redusert hastighet.

Den andre muligheten er å laste ned hele materialet til disk, for å spille det av lokalt. Fordelen med en slik modell er at det, forutsatt at maskinen hardwaremessig kan spille av videomaterialt, sikres at avspilling kan foregå uten avbrytelser eller kvalitetstap grunnet problemer eller midlertidig/varig kapasitetsmangel i nettverket. Det finnes dog noen problemer med denne modellen. Det primære problemet ligger i det at brukeren må laste ned hele videofilen til disk først. Avhengig av tilgjengelig kapasitet i nettverket og på tjenersiden kan dette ta fra noen få minutter til et par timer, avhengig av lengden og kvaliteten på filmen. I motsetning til buffring vil brukeren dermed ikke oppleve en tilnærmet øyeblikkelig oppstart, men kan risikere å måtte vente i et lite til stort tidsrom. Hvis en legger til noen sluttnoder, slik som f.eks. hjemmekontorer og liknende, vil de ha en mindre båndbredde enn 1000 mbps, kanskje lavt som oppring samband på 56 kbps, men mer sannsynlig 768 kbps ADSL forbindelse, vil denne modellen dog ha en styrke. Når en antar at maks tilgjengelig båndbredde er mindre enn den båndbredden som er nødvendig for å vise videoen i realtime, vil det ikke være noe annet alternativ

for brukeren. Selve om en skulle buffre noe video, vil en relativt raskt gå slutt på buffer og måtte re-buffre. En kan tenke seg at en tjeneste basert på total nedlasting på materiale fungerer slik at en starter nedlasting en dag og at nedlastingen er ferdig neste dag.

Den tredje muligheten er en variasjon av den første, og det er å starte avspilling øyeblikkelig. Forutsatt at tilgjengelig båndbredde i nettverket fra ende til ende er større enn nødvendig båndbredde for å spille av videofilen, vil avspilling kunne begynne umiddelbart. En har da muligheten til å sakte opparbeide et buffer for å takle midlertidige båndbreddeproblemer, samt strupe hastigheten til nødvendig båndbredde pluss en liten overhead for å tillate flest mulig tilkoblinger til denne videostrømmen. I motsetning til løsningen med å laste ned hele filmen først, vil det ikke være nødvendig for serveren å prøve å gi hver klient maksimalt med båndbredde, men derimot kun gi den nødvendige. Dette vil også lette arbeidet med å maksimalisere antall klienter som kan tilkobles serveren. Problemet her, er at streamingen er avhengig av at tilgjengelig båndbredde er større enn nødvendig båndbredde under hele streamingen hvis en ikke lager en buffring/cache mekanisme, samt at denne modellen ikke takler nettverksproblemer spesielt godt.

Konklusjon

Det største problemet med en unicastbasert streamingmodell er skalerbarheten.

En mulig generell formell for skalerbarhet der skalerbarhet er antall samtidige klienter kan se slik ut:

$$skalerbarhet = \frac{maksimal\ bandbredde}{bandbredde\ per\ strøm} \tag{4.1}$$

En slik formell tar dog ikke høyde for noen forsinkelser og problemer i nettverket, slik som problemer med kabelkvalitet, dårlig terminering og liknende.

En eksakt vurdering av andelen av teoretisk maksimal båndbredde som vil være tilgjengelig i praksis ligger utenfor denne oppgaven, men vil her være satt til 95 prosent. Dette er basert på en tommelfingerregel som setter av båndbredde til out-of-band signalering og annen nødvendig trafikk på nettverket, men som ikke har noen empirisk støtte.

Formelen vil da se slik ut:

$$skalerbarhet = \frac{teoretisk\ maksimal\ bandbredde}{bandbredde\ per\ strøm} * 0,95 \tag{4.2}$$

For en 5 mbps HDTV-kvalitets strøm på et 1 Gbps nettverk vil en da kunne sende $\frac{1000}{5} * 0,95 = 190$ strømmer.

De fleste nettverk vil dog ikke kun være for videostreaming, men en må også anta at annen trafikk vil gå gjennom nettet, noe som ytterligere reduserer antall tilgjengelige samtidige strømmer.

Ved en antakelse om 50 prosent av tilgjengelig båndbredde til annen trafikk, vil antallet tilgjengelige samtidige strømmer være 85. Dette antallet kan økes ved å gå ned på oppløsning og kvalitet, men det grunnleggende problemet gjenstår. Ved et fullstendig switchet moderne nett, vil behovet for å sende annen trafikk være redusert til det streamer-maskinen har behov for, noe som øker den tilgjengelige båndbredden.

Men uavhengig av dette, og at tilgjengelig båndbredde for det resterende av nettverket teoretisk ikke skulle lide av bruken av båndbredde til streaming, gjenstår det et problem med skalerbarheten av streamingen.

Den primære flaskehalsen er tilgjengelig båndbredde mellom streaming-maskinen og dens nettverksswitch. Det finnes dog alternativer til unicast.

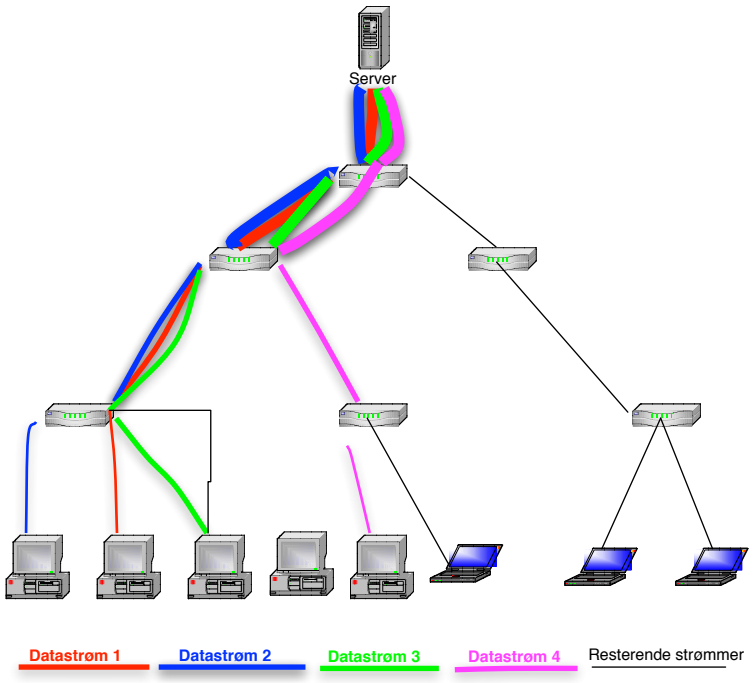
Unicast vs multicast

En god analogi for unicastdistribusjonsmodellen er det fysiske telefon-nettet. For hver telefon (slutt-node), finnes det en fysisk kabel frem til en telefonsentral (switch). Tidligere benyttet en seg også av dedikerte fysiske kabler, en for hver samtale, som ble koblet sammen ved en telefonsamtale, og var dedikert til denne samtalen under hele samtalen. Dagens moderne telefon-nett multiplekser mange samtaler inn på samme fysiske kabel, men prinsippet om en ubrutt kjede frem til den andre parten eksisterer fortsatt, den er bare virtualisert bort ifra fysiske enkeltkabler. På samme måte foregår en unicastoverføring, bare med en enda høyere grad av virtualisering.

For ikkesanntidsoverføringer, som filnedlastning, er ikke varierende båndbredde et problem. Når for stor trafikk på nettverket fører til pakketap, kan de delene av overføringen som er tapt sendes på nytt, uten at det påvirker nedlastingens kvalitet. Klientnoden får fila i sin helhet.

Behovet for den virtuelle dedikerte forbindelsen er fortsatt til stede, men annen trafikk er ikke et problem.

Overføring av lyd, video og annen sanntidsinformasjon er dog et problem. Sanntidsinformasjon er kun interessant for klientnoden i den utstrekning den når klientnoden i sanntid eller nær sanntid. Dette skaper et behov for ikke bare en dedikert virtuell forbindelse, men et behov for en dedikert båndbredde under hele oversendelen av sanntidsdataene. Samtidig vil det ofte kunne være mange klienter som ønsker de samme sanntidsdataene, men med unicast-modellen, vil disse kreve klienter gange strømmens båndbredde i båndbredde ifra nettverket.



Figur 4.1: En konseptuell skisse av unicast til flere klienter

Multicast

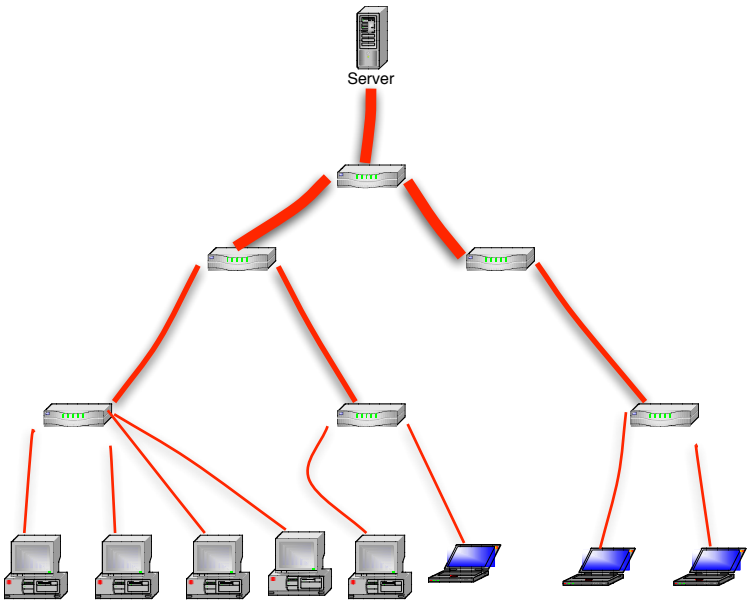
Løsningen på denne typen sanntidsdata til multiple klientnoder er multicast.

En god analogi her er vannforsyningsrør. Vann kan også ses på som en sanntidskrevende ressurs, da en trenger vann når en åpner krana, og ikke fem minutter senere. Vann distribueres uten dedikerte linjer mellom kran og vannforsyningen, men derimot ved hjelp av åpne forgreininger uten styring. Tappingen av vann ved vannkrana skaper et behov for mer vann som automatisk dekkes uten behov for detaljstyring.

Den store fordelen med multicast ligger i besparing av båndbreddebruk. Ved at båndbreddebehovet mellom to rutingnoder kun er båndbredden til strømmen som overføres, og ikke båndbredden til strømmen som overføres ganger antall klienter strømmen overføres til, skalerer denne løsningen veldig godt.

Multicast er enda ikke vanlig utenfor de akademiske miljøer og andre som arbeider mye med nettverk, men da HiØs nettverk støttet multicast, var ikke dette en problematisk faktor.

Selv om antall klienter ikke lenger skaper et skaleringsbehov, finnes det dog noen andre utfordringer. Denne artikkelen[17] gir en god diskusjon av



Figur 4.2: En konseptuell skisse av multicast

dette.

Pakketap: Unicast benytter primært TCP for overføring, men kan også bruke UDP. Fordelen med TCP er at det er en protokoll som garanterer at pakken når mottakeren, og resender tapte pakker ved behov. Multicast, som en mange til mange protokoll, kan kun benytte seg av UDP, som ikke kan garantere at alle pakker når mottakeren. For strømmer der integriteten av innholdet er kritisk, slik som filoverføring, kan ikke pakketap tolereres, og TCP og unicast er dermed en opplagt metode. Men for sanntidsdata er det andre kriterier som er viktigere. Den viktigste egenskapen til sanntidsdata er at de kan sendes og mottas i sanntid. Der unicast kan godta at samme segment av strømmen blir sendt flere ganger frem til den når mottakeren, kan ikke sanntidsdata godta det samme. Det at pakkene blir sendt i sanntid uten gjensending blir en del av bildet en må tolerere og en styrke, da en kan garantere at de data som når frem er i sanntid. De fleste sanntidsprotokoller for multicast tar høyde for noe pakketap i form av forward error correction hvor data spres ut over flere pakker og også sendes i form av sjekksummer og i noen protokoller to ganger slik at tapte pakker kan gjenskapes på andre siden. De fleste MPEG standarder støtter også dette.

Network congestion: Mangelen på garantert mottak av datapakkene skaper et problem i forbindelse med høy trafikk på nettverket. Ved høy trafikk on congestion problematikk vil noen ruter-noder starte med å droppe UDP trafikk som default, mens andre vil måtte droppe vilkårlige pakker. For TCP er ikke dette et problem, da de blir gjensendt når det er mulig, og gjensendingsraten

reduseres frem til ruternoden igjen kan prosessere all trafikk. Denne typen mekanismer eksisterer ikke for multicast.

En konsekvens av dette er at skulle for mange multicaststrømmer gå gjennom en ruternode, vil den oppleve varig network congestion, og kvaliteten på samtlige multicaststrømmer vil oppleve pakketap.

Den andre konsekvens er nært beslektet med den første, og innebærer at både kortvarig og langvarig network congestion kan skape problemer for en multicast strøm.

Det finnes dog også mulige løsninger på disse utfordringene. En kan tenke seg at en ved hjelp av ressursprioritering i rutingnodene setter av en del av den tilgjengelige båndbredden til multicast, og mer spesifikt til de multicast-strømmer som en ønsker at skal være tilgjengelig med god kvalitet i nettverket.

Streaming av åpne formater på en Windowsplattform

Innledning

En av de viktigste utfordringene for et streamingsystem for HiØ var brukervennlighet. De fleste antatte brukere var ikke IT-kyndige, og burde derfor helst ikke behøve å gjøre stort mer enn å trykke på en link.

HiØ hadde valgt Windows, versjon XP, som sin primærplattform, og 99.9 prosent av mulige brukere ville derfor sitte på denne plattformen. De resterende brukere ville sitte på en standard linux løsning, eller en selvoppsatt Mac Os X løsning, men vil generelt ha god nok IT-kompetanse til å få enhver standard-kompatibel løsningen til å fungere knirkefritt uten store problemer.

Windowsplattformen, hadde dog visse problemer med frie og åpne standarder.

Windows Media

Windows Media [18] er standardformat for bilde og video i Microsoft Windows. Windows media er en lukket og proprietær standard, og Microsoft har blitt dømt til å fjerne Windows Media fra Windows solgt i Europa [19].

På tross av dette er Windows Media fortsatt godt integrert i Microsoft Windows, ikke minst grunnet noen gode sider sett fra et rent brukerperspektiv.

Windows Media, både avspiller og format, er dypt integrert inn i Microsoft Windows, og dets nettleser, Internet Explorer.

Dette muliggjør dyp integrering og enkelt brukergrensesnitt, slik som f.eks. NRKs video on Demand tjeneste, og Regjeringen.no. Problemene med en slik løsning, er dog at den kun fungerer godt på Windowsplattformen, og ikke på Mac Os X, linux-baserte OS og andre plattformer.

Den gode integreringen med OSet og internetbrowseren skaper allikevel en forventning til hvordan andre løsninger skal virke.

MPEG-2 og QuickTime

QuickTime [20] er blant de mest installerte mediespillerene på Windows-plattformen etter Windows Media. QuickTime har god støtte for flere MPEG formater, men krever en lisensiert ekstra-modul for å spille av MPEG-2.

QuickTime tilbyr dog en god integrering med nettleserene på sine plattformer (Windows og Mac OS X), noe som for de fleste tilfeller gir en god og enkel avspilling av media for sluttbrukeren.

Problemet med QuickTime er at den blant annet overtar avspilling av såkalte .sdp filer [21], og det viste seg å være nærmest umulig å la andre mediaspillere overta .sdp i MS Internet Explorer. MS Internet Explorer var dog den nettleseren som en stor del av de ikke IT-kyndige i brukergruppen ville bruke, ofte med QuickTime installert.

Konflikten lå her i QuickTimes forandring av nettleseren for å gi sluttbrukeren en mest mulig automatisk og arbeidsfri avspilling av media, og problemene som oppstår når QuickTime ikke kan spille av mediet fullt ut. Spilleren ville starte i nettleseren, lyd ville komme fra høyttalerene, men det ville ikke være noe bilde.

VLC og automatisk avspilling

Da det ikke var et alternativ å betale QuickTime lisensen for MPEG-2, og bruk av åpen kildekode og åpne formater var et mål for prosjektet, var VLC utpekt som medieavspiller for prosjektet.

Problemet med VLC lå i integreringen med MS Internet Explorer. Der Windows Media og Quicktime integrerer godt med Internet Explorer, har VLC kun støtte for direkte integrasjon i nettleseren FireFox.

Målet om åpne standarder og åpen kildekode lå her i direkte konflikt med usabilityhensyn. Der de proprietære løsningene Windows Media og QuickTime kunne tilby direkte intergrering i nettleseren for avspilling av media, kunne fri programvare kun tilby integrering i FireFox. Bruken av Internet Explorer i en majoritet av den tenkte brukergruppen gjorde behovet for en løsning absolutt.

En god nok løsning ville være å få VLC til å bli startet av nettleseren, men QuickTimes absolutte kapring av filtypen for den typen video som ble streamet, .sdp, gjorde dette vanskelig.

VBscripting og andre løsninger

Av hensyn til brukervennlighet og sluttbrukeren var det nødvendig å finne en løsning som tillot bruken av VLC og åpne standarder og som samtidig tillot sluttbrukeren å bruke Internet Explor og/eller QuickTime.

Filendelsen .remmen

Det primære problemet lå i å få startet VLC automatisk selv når QuickTime var installert hos sluttbrukerene. QuickTime har i utgangspunktet god støtte for de strømmer som bruker Session Description Protocol for å sende data over multicast, men MPEG-2 som vi brukte til video, krevde som tidligere nevnt en ekstra modul.

Problemet med Internet Explorer og QuickTime var at selv om fil og MIME-typen for SDP-filer var satt til å åpnes med VLC, ville QuickTime forsøke å spille de av direkte i nettleseren. Dette ville gi lyd, uten bilde.

Det fantes sikkert en mulighet for å fjerne denne koblingen og fortsatt kunne ha QuickTime installert, men problemet lå i å ha en løsning som selv ikke IT-kyndige enkelt kunne bruke.

Problemet lå i filendelsen .sdp, en løsning som ble vurdert var det enkle grepet å forandre filendelsen og dens tilhørende MIME-type. Nye linker ble generert med filendelsen .remmen og en eksperimentell MIME-type ble laget kallt "application/x-remmen". Problemet med denne løsningen var at den var i det store og hele et hack, og det gav problemer på andre operativsystemer, men lot brukeren velge å bruke VLC for .remmen filer uten at QuickTime skapte problemer for brukeren.

VBscripting for Internet Explorer

Problemene med andre OS og behovet for noe konfigurasjon av klienten for å få .remmen filene til å virke skapte et behov for andre løsninger. et forslag fra IT-drift ved HiØ som skulle overta løsningen var bruken av VBScript.

VBScript fungerer kun med Internet Explorer, dette ville være en løsning rettet direkte mot Internet Explorer, og ville dermed kunne omgå hele QuickTime problematikken.

Sikkerhetsmessig var dog denne løsningen et problem. Løsningen med VBscripting krevde at websiden som leverte linkene ba nettleseren og dets OS om å starte et program lokalt på maskinen, noe som er mer vanlig for virus og andre angrep enn legitime applikasjoner. I et standardoppsett kunne dog serveren som leverer Remmen Streaming Server konfigureres til å være en del av Intranettet for Internet Explorer, og IE ville da tillate at VBskriptet

eksekverte og startet VLC.

Problemet var å skaffe en løsning som også virket for de Windowsmaskiner som ikke stolte på nettsiden og dermed ikke ville kjøre VBscriptet. Det var her to mulige løsninger. Det første alternativet var å kun sende VBscript linker til Internet Explorer klienter, uten noen annen link. Dette muliggjorde god brukervennlighet for sluttbrukerene, men ville skape problemer hvis Internet Explorer ikke var satt opp til å stole på systemet. Det andre alternativet var å sende VBscript og en normal link til .remmen filen til alle klienter. Nettlesere som ikke er Internet Explorer ville ignorere VBscriptet, mens Internet Explorer ville først kjøre VBscriptet og gi den vanlige linken. Problemet var at Internet Explorer da ville gi den vanlige linken uavhengig av om VBscriptet hadde eksekvert normalt eller ikke.

Det var ikke den beste løsningen, men det var det minste av to onder.

Oppsummering

Streaming i åpne formater og med fri kildekode konkurrerende med de eksisterende proprietære formater (f.eks. Windows Media, QuickTime, RealPlayer), skaper visse problemstillinger.

Den gode integreringen mellom nettleser og medieavspiller som de proprietære formater kan monopolisere for den mest dominerende nettleseren, Internet Explorer, skaper forventninger hos sluttbrukeren. Disse forventningene til god integrasjon og såkalt "automagisk" avspilling vil kunne gi sluttbrukeren et dårlig inntrykk av alternativene.

Men er forventningene til god integrasjon og "automagisk" avspilling i seg selv et problem? For de fleste brukere er et godt brukergrensesnitt med små problemer en fordel, og ofte en nødvendighet for att de skal kunne bruke applikasjonen.

Der de proprietære løsninger baserer seg delvis på åpne og delvis på lukkede standarder, må de åpne og frie løsninger basere seg på åpne og frie standarder. Problemet ligger vel kanskje da i mangelen på åpne og frie standarder for integrering i nettlesere?

For de proprietære løsninger gir muligheten til å lukke de åpne løsninger ute et konkurransefortrinn lenge de proprietære løsninger er markedsledende, dette problemet eksister ikke for de åpne løsningene som alltid vil konkurrere på en lik basis uavhengig av markedslederskap.

For bruk på web så er nå arbeides det nå med HTML 5 som skal inneholde video og audio tagger [22]. Det primære problemet her har vært valget mellom åpne og propietære standarder. I det opprinnelige utkastet så var Ogg/Theora valgt som videostandard, men det har senere blitt trukket grunnet innsigelser fra flere av H.264s patentholdere som også er medlem av W3C som lager HTML standardene. Microsoft og Apple har valgt å støtte H.264 som videotag

codec i sine HTML 5 implementasjoner, der Opera, Mozilla Firefox og andre har valgt den frie og åpne Ogg/Theora. Et mulig kompromiss ser ut til å kunne bli Googles WebM [23] prosjekt som kvalitetsmessig er likeverdig med H.264, men som Google har sluppet fritt og åpent, og samtidig gitt en evig, gratis lisens til å bruke de patenter som omfatter WebM. WebM vil dog nok ikke utgjøre en kritisk menge webvideo før 2011/2012.

CAM, DVB og DreamBox

Innledning

Streaming av TV sendt over satellitt er i dag vanlig for kabelselskaper og borettslag. Disse løsninger er dog baserte på koaksialkabler og vanlige TV signaler frem til sluttbrukeren. En av utfordringene i dette prosjektet var å finne TV tunere som tillot fjernstyring over IP og en viderelevering av den digitale MPEG-2 strømmen som TV over satellitt sendes i.

Det finnes flere mulige løsninger på problemet, men alle har sine egne problemer.

Den første løsningen baserer seg på PCI utvidelseskort for en standard IBM-kompatibel PC, hvor en kobler til koaksialkabelen fra satellittantennen. En kan da ta MPEG-2 strømmen direkte, og videresende denne til sluttbrukeren hvis en ønsker.

Den andre løsningen baserer seg på en standard satellittmottaker, hvor en re-digitaliserer det analoge bildet som kommer ut fra mottakeren.

Den tredje løsningen er på mange måter en hybrid av de to første, hvor en har en standard satellittmottaker som kjører et tilpasset standard operativsystem og en kan få den digitale strømmen direkte ut.

Den første løsningen skulle i utgangspunktet være den beste, problemet ligger i krypteringen som foregår av mange TV-kanaler. I utgangspunktet har DVB-standarden en god løsning på dette, såkalte CAM-moduler, Conditional Access Modules, men satellittselskapene er ikke glad i at disse brukes i satellitt-mottakere de ikke har godkjent, og grunnet frykten for satellittpirater, er ingen PCI DVB mottakere godkjent av noe satellittselskap. Det er dermed noen tekniske og juridiske barrierer for å få dette til å virke tilfredsstillende.

Satellittselskapene og parabol-pirater

For mange av satellitt TV-kanalene som finnes kommer inntektene i fra betalende kunder. Levering av video over satellitt har dog hatt store problemer med såkalte satellitt-pirater [24] siden dag èn.

Historisk sett har det vært en kamp mellom satellitt-leverandørene på den

ene siden og parabol-piratene på den andre siden, med et hardwaregrensesnitt definert av en åpen standard som slagfeltet.

I et forsøk på å redusere piratvirksomheten har satellittselskapene i økende grad begynt å ta i bruk delvis proprietære løsninger, eller å redusere tilgjengeligheten av lovlig hardware som kan kombineres med pirat-dekodere.

Den vanligste metoden i dag for å hindre piratvirksomheten er såkalt "double-binding" hvor en ikke bare må ha en dekoder som støtter dekrypteringskortet, men kortet bindes til serienummeret til den bestemte dekoderen.

Lovligheten av å binde kundene på en slik måte og omgåelse av slike bindinger ble ikke vurdert under prosjektet.

Kapittel 5

Konklusjon

5.1 Resultat

Hva ble resultatet?

Kort oppsummert ble resultatet et fullstendig modulært, fullt andvendbart generisk streamingsystem for opptak og livestreaming av satellittkanaler.

Kildekoden lå lenge ute på SourceForge.net og var tilgjengelig for de som skulle ønske å bruke systemet. Systemet ligger fortsatt på SourceForge, men er i dag ikke lett tilgjengelig for offentligheten, da versjon 2 foretrekkes. Allerede versjon 1 var ønsket av Uninett tilgjengeligjort for deres medlemsinstitusjoner som et generelt video over IP system som kunne erstatte koaksialbaserte systemer.

En god menge erfaring med bruk av DreamBoxer, programmering mot deres grensesnitt og bruk av CAM-moduler i et slik system gav et veldig godt fundament for arbeidet med versjon 2 av systemet, der rammeverket ble designet av Audun Vaaler.

Systemet har et enkelt og minimalistisk webgrensesnitt, men som enkelt kan forbedres gjennom å bytte ut koden som skaper webgrensesnittet.

Hva ble annerledes?

Systemet var designet som et fullstendig modulært og lett utbyttbart system. Forandringene av kravspesifikasjonene underveis førte dog til at mye kode er forandret så mye at det gikk ut over modulariteten og enkelheten som var et mål for systemet.

Systemet er fullt ut anvendbart, men har blant annet problemer med opptak

gjort på Dreamboksene, og filstørrelse når opptak blir avspilt. Systemet vil heller aldri bli tatt i bruk slik det er beskrevet her, men erfaringene fra arbeidet med systemet gir et godt fundament for versjon 2, som har det foreløbige navnet Alt Når Det Passer (ANDP).

5.2 Hva gikk galt/hva kan forbedres?

Som i alle prosjekter var det aspekter som gikk galt og aspekter som kan forbedres. Det primære problemet i å stabilisere koden for finpussing hadde sin bakgrunn i overgangen fra de ir-styrte tunere til DreamBoxer. Fremdriften var ikke good nok til tider. Et annet problem var ikke nok brukertesting av brukergrensesnittet.

Planlegging

Prosjektets form og grunnleggende struktur og API ble tidlig definert, og fulgt i den videre utvikling. Den grunnleggende struktur ble laget med utgangspunkt i behovet for en distribuert applikasjon minimum spredd over to til tre maskiner. Dette skapte et behov for en modularisert tilnærming til problemet, og et fast API mellom de forskjellige moduler.

Dette skapte en viss kompleksitet i koden, men gav også muligheten for å raskt kunne forandre deler av designet uten en fullstendig omskrivning av koden.

Det er godt mulig at forandringer av slik som overgangen fra ir-styrte tunere til DreamBoxer burde ha ført til en re-evaluering av det overordnede designet for RStS, istedenfor å bytte ut deler av to moduler uten å evaluere følgene for helheten av systemet. Dette ble ikke gjort, og det er mulig at hadde det blitt gjort da, kunne de ha ført til en bedre overordnet design en det som er i dag.

Realtimestreaming

Realtimestreamingen skapte også enkelte problemer kombinert med avgjørelsen om å la DreamBoxen stå for opptaksfunksjonaliteten. Da DreamBoxen ikke viste seg å være i stand til å både levere en transportstrøm digitalt, og ta den opp, var den enkleste løsningen å bruke de allerede innkjøpte PVR-350 kort, men og det var den beste løsningen, det er mindre sikkert. et viktig aspekt ved den avgjørelsen var tidsaspektet, målet var å få systemet ferdig så raskt som overhodet mulig, og bruken av videodigitaliseringskortene hjalp nok med til å få dette til.

Det er dog relativt store kostnader forbundet med å kjøpe inn et kort per livesstrøm, direkte i innkjøpspris for kortene, og inndirekte ved at PCer må

kjøpes inn for å drive kortene. En løsning som tok den digitale transportstrømmen og delte den i to, en for opptak og en for livestreaming ble nok ikke vurdert godt nok, og er den løsningen som ville blitt valgt utifra erfaringen med dette prosjeket.

Brukertesting

Brukertesting skulle i utgangspunktet vært brukt mye i Remmen Streaming Server. Og som den vedlagte rapporten viser, var det opprinnelige adminstrasjonsgrensesnittet godt brukertestet slik at det skulle kunne brukes av de fleste.

For utviklingen av selve brukergrensenittet for reservasjon av kanaler, gjenhenting av opptak og for å se på live streaming, var tilgangen på brukere dog ikke like god. Det var i utgangspunktet oppdragsgiver som skulle spre Remmen Streaming Server i et stadig større utvalg av sluttbrukerene, men dette ble desverre aldri gjort. Det er her fullt mulig at det burde ha blitt funnet en mindre testgruppe for testing av brukergrensenittet, men dette ble da ikke gjort, og brukergrensesnittet ble derfor heller ikke testet mye av andre enn Audun Vaaler.

Kode-entropi?

Kode med stor grad av gjenbrukbarhet og modularitet blir som hovedregel sett på som god kode, men vil gjenbruk av kode med for liten tilpasning til den nye funksjonaliteten gi et godt nok sluttresultat.

Da brukergrensesnittet for vanlige brukere ble utviklet ble det tatt utgangspunkt i den eksisterende koden for administrasjonsbrukergrensesnittet. Denne koden var enkel og modulær, samt tilpasset å lage et antall bokser tilsvarende antall kanaler i en webside.

Problemet med gjenbruket av koden i dette tilfellet var at den nye koden ble lagt til inne i koden som genererte innholdet i en boks, slik at mye unødvendig kode i forhold til den funksjonaliteten som skulle lages ble beholdt. Dette førte også til at det ble mye vanskeligere å få en oversikt over hvordan koden i dette tilfellet fungerte da det var flere funksjonskall og if-løkker som bare kallet andre funksjoner èn gang.

Fordelen med gjenbruket av koden for administrasjonsbrukergrensesnittet var tidsbruk for å få ting opp og gå, det gikk atskillig raskere med den koden i bunn enn alternativet med å skrive alt på nytt. Problemet lå i videreutvikling av koden etter hvert som grensesnittet utviklet seg, og i de senere versjoner gikk helt bort ifra kortstokkmetaforen.

Det hadde her vært en fordel å ha skrevet koden for brukergrensesnittet på bunnen av når selve utseendet var ferdig forandret etter brukertesting for å lette videre utvikling og vedlikehold av koden i bunnen av. Med utgangspunkt

i de erfaringer som da er gjort i løpet av prosjektet, burde dette da resultere i vesentlig bedre og mer effektiv kode.

Windows Media Player og QuickTime - alternative løsninger

Windows Media Player og QuickTime Player har enkelte store fordeler med tanke på brukergrensesnitt for den jevne sluttbruker, som primært vil sitte på en Windows XP maskin med Internet Explorere som nettleser, selv om andre operativsystemer og nettlesere (slik som Firefox), gjør stadige innhugg. Begge disse, lukkede og proprietære løsninger, har god integrering inn i Internet Explorer, og gir dermed sluttbrukeren en følelse av at alt "bare virker". For sluttbrukeren er det viktig at ting nærmest bare virker uten problemer, og at det dermed ikke er mye oppsett av maskinen som må gjøres før ting begynner å virke.

Løsningen med MPEG-2 i multicast skapte vesentlige problemer med å få ting til å fungere "automagisk", og det er mulig at en burde ha sett nærmere på alternative løsninger, selv om disse i hovedsak ville være i strid med de grunnleggende prinsippene som ble valgt for prosjektet.

En vanlig måte å embedde video i websider på, er Flash-formatet, som blant annet Youtube [25] benytter seg av, baserer seg på embedding i websider som krever en flash leser, noe de fleste har installert på maskinene sine, og som enkelt kan installeres på de Windows maskiner som ikke har det installert. Dette er en delvis lukket og proprietær løsning, selv om selve flash-formatet er åpent dokumentert, kreves det spesielle løsninger for å lage selve innholdet.

Ett annet alternativ er Ogg/Theora med Ogg/Vorbis lyd. For å integrere Ogg i nettleserene, finnes det en java applet med navn Cortado [26] som kan benyttes for å embedde lyd og bilde i en webside.

Da Ogg, Theora og Vorbis er åpne og patentfrie formater, og Sun arbeider med å frigi hele java-APIet og den bakenforliggende kildekoden, er dette en løsning som i hovedsak ville vært i tråd med de grunnleggende prinsippene for prosjektet.

Problemet med disse alternative løsningene er den manglende støtten for multicast. Avveiningen for videre utvikling vil da her ligge mellom brukervennlighet og skalerbarhet, grunnet mangelen på støtte av åpne og frie formater og standarder. Dette diskuteres nærmere litt senere i konklusjonen.

5.3 Veien videre

Det arbeides nå med versjon 2 av Remmen Streaming Server. Denne tar utgangspunkt i de erfaringer, problemstillinger og det arbeid som er gjort med Remmen Streaming Server, men forsøker en litt annen tilnærming for å løse noen av de grunnleggende problemer som blant annet Windows Media Player og QuickTime

skapte, samt problemet med å redigitalisere videostrømmene for realtimestreaming. Audun Vaaler er den primære koderen bak versjon 2 slik den foreligger i dag, med sitt nye navn Alt Når Det Passer.

Tilleggsstoff og vedlegg

Ordliste

A
ANDP *Alt Når Det Passer* iii,v,56

D
DreamBox iii,32-35,37, 55,56
En linux-basert PVR satelitt-mottaker

H
HTTP *HypterText Transfer Protocol* 4,5,32

L
LAMP 7
Linux, Apache, Mysql, PHP - Webutvikling hvor en benytter disse fire teknologier.

M
multicast 19,33,37, 45-48,50,58
nettverksbasert en-til-mange eller mange-til-mange kommunikasjon

P
Python iii,3,4,18, 22,23,29-31,34-36
Programmerings- og skriptingspråk

R
RPC *Remote Procedure Call* 4
RStS iii,xi,4, 11-13,15,17,18, 29,32,34, 36-38,56
Remmen Streaming Server

U
unicast *nettverksbasert en-til-en kommunikasjon* 42-47

V
VLC 5,6,32-34,37, 49,50
VideoLan Client - en multimedia avspiller og enkoder

Vedlegg

Kildekode - Remmen Streaming Server v1 - CD Ytterligere skjermbilder av Remmen Streaming Server. - CD Grensesnittdesignrapport for administrasjonsbrukergrensesnittet ligger på CD som bakgrunnsmateriale. - CD

Bibliografi

[1] Creative Commons
URL: http://creativecommons.org/about/licenses/
(lesedato: 05.10.2010)

[2] Wikipedia Commons. Model view controller diagram
URL: http://en.wikipedia.org/wiki/File:ModelViewControllerDiagram2.svg
(Lesedato: 05.10.2010)

[3] G. van Rossum. General python faq
URL: `http://www.python.org/doc/faq/general/`
(lesedato: 05.10.2010)

[4] XML-RPC
URL: http://www.xmlrpc.com
(lesedato: 05.10.2010)

[5] D. Marshall. Remote procedure calls (RPC)
URL: `http://www.cs.cf.ac.uk/Dave/C/node33.html`
(lesedato: 05.10.2010)

[6] Videolan project
URL: `http://www.videolan.org`
(lesedato: 05.10.2010)

[7] T. Reenskaug. Model View Controller
`http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html`
(lesedato: 05.10.2010)

[8] Postgresql website
URL: `http://www.postgresql.org`
(lesedato: 05.10.2010)

[9] MythTV. Open source dvr
URL: http://www.mythtv.org/
(lesedato: 05.10.2010)

[10] IRTrans Infrared Control System
URL: http://www.irtrans.de/en/
(lesedato: 05.10.2010)

[11] Dream Multimedia
URL: http://www.dream-multimedia-tv.de/en
(lesedato: 05.10.2010)

[12] Uninett AS
URL: http://www.uninett.no
(lesedato: 05.10.2010)

[13] Uninett Gigacampus
URL: http://www.gigacampus.no
(lesedato: 15.04.2009)

[14] Digital Video Broadcasting Project
http://www.dvb.org/technology/fact_sheets/DVB-S2_Factsheet.pdf
(lesedato: 04.10.2010)

[15] ISO/IEC Moving Pictures Experts Group (MPEG)
URL: http://mpeg.chiariglione.org/
(lesedato: 04.10.2010)

[16] ETSI - Digital video broadcasting (dvb) - specification for the use of video and audio coding in broadcasting applications based on the mpeg-2 transport stream
URL: http://www.etsi.org/deliver/etsi_ts/101100_101199/101154/01.09.01_60/ts_101154v010901p.pdf
(lesedato: 04.10.2010)

[17] P. Arberg. High availability multicast delivery in iptv networks
URL: http://www.nanog.org/meetings/nanog40/abstracts.php?pt=MjMzJm5hbm9nNDA=&nm=nanog40
(lesedato: 05.10.2010)

[18] Microsoft windows media
URL: http://www.microsoft.com/windows/windowsmedia/default.mspx
(lesedato: 02.08.2008)

[19] Eu press release - windows media
URL: http://europa.eu/rapid/pressReleasesAction.do?reference=IP/04/382&format=HTML&aged=1&language=EN&guiLanguage=en
(lesedato: 05.10.2010)

[20] Apple Quicktime
URL: http://www.apple.com/quicktime/
(lesedato: 05.10.2010)

[21] IETF. Session description protocol
URL: http://www.faqs.org/rfcs/rfc2327.html
(lesedato: 05.10.2010)

[22] W3C consortium. HTML 5 W3C editor's draft
URL: http://dev.w3.org/html5/spec/spec.html
(lesedato: 05.10.2010)

[23] WebM Project
URL: http://www.webmproject.org/
(lesedato: 05.10.2010)

[24] Wikipedia: Pirate decryption
URL: http://en.wikipedia.org/wiki/Pirate_decryption
(lesedato: 05.10.2010)

[25] Youtube - broadcast yourself URL: http://www.youtube.com
(lesedato: 05.10.2010)

[26] Flumotion cortado
URL: http://www.flumotion.net/cortado/
(lesedato: 05.10.2010)