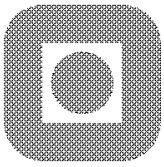
NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET

FAKULTET FOR FYSIKK , INFORMATIKK OG MATEMATIKK



HOVEDOPPGAVE

Kandidatens navn:	Daniel Remmem
Fag:	Datateknikk
Oppgavens tittel (norsk):	VCR-funksjonalitet for MPEG2-video levert fra en videotjener
Oppgavens tittel (engelsk):	VCR-functionality for MPEG2-video delivered by a video server

Oppgavens tekst:

MPEG-2 er i dag den mest aktuelle standarden for leveranse av høykvalitets video over nettverk. I utviklingen av standarden er det primært tatt hensyn til avspilling i normal hastighet i foroverretning. Oppgaven går ut på å studere MPEG-2-standarden for å finne strategier for å oppnå fleksibel avspilling (VCR-funksjonalitet) av MPEG-2-video levert over nettverk fra en videotjener. Viktig funksjonalitet er støtte for normal avspilling, posisjonering (hopp), avspilling i høy/lav hastighet og baklengs avspilling. I vurderingen av ulike strategier er det viktig å ta hensyn til ressursforbruk i videotjeneren. Et utvalg av de foreslåtte løsningene skal implementeres og testes ut i praktiske forsøk med videotjeneren Elvira II.

Oppgaven gitt:	28. August 1997
Besvarelsen leveres innen:	5. Januar 1998
Besvarelsen levert:	22. Desember 1997
Utført ved:	Institutt for datateknikk og informasjonsvitenskap
Veileder:	Olav Sandstå

Trondheim, 22.12.97

Roger Midtstraum
Faglærer

Sammendrag

Denne rapporten presenterer resultatet av en hovedoppgave utført ved Institutt for datateknikk og informasjonsvitenskap, Norges teknisk-naturvitenskapelige universitet.

Oppgaven har omhandlet bruk av videostandarden MPEG-2 i en videotjener. Målet har vært å finne ut hvordan VCR-funksjonalitet kan oppnås i en film av dette formatet. Med VCR-funksjonalitet menes tilfeldige hopp i videoen (tilfeldig aksess), og avspilling i ulike hastigheter både forlengs og baklengs. Hovedvekten skulle legges på å oppnå spoling (høye hastigheter) uten for stort bruk av ressurser. Metoder for å oppnå dette skulle implementeres og testes, disse skulle rette seg mot integrasjon i videotjeneren Elvira II. Denne tjeneren er i utvikling ved databasegruppa ved instituttet.

MPEG-2 standarden tilbyr to ulike typer strømmer for lagring av lyd og video, programstrøm og transportstrøm. For Elvira II er det valgt å bruke transportstrømmen.

En rekke metoder for å oppnå spoling og tilfeldig aksess er beskrevet og vurdert. Den beste metoden for å oppnå spoling ble vurdert til å være bruk av *spolefiler*. En spolefil er en fil som konstrueres fra originalfila, og som ved avspilling gir en ulik hastighet og/eller retning sammenlignet med avspilling fra originalfila. For å oppnå tilfeldig aksess i en film er det funnet en metode som baserer seg på identifisering av et felt i transportstrømmen.

Det er konstruert og implementert en framgangsmåte for å lage spolefiler med ulike hastigheter, og som tilbyr tilfeldig aksess i disse. Denne framgangsmåten baserer seg på bruk av både ferdig og egenutviklet programvare. Videre er det implementert programvare som integrerer både originalfiler og spolefiler i Elvira II. Denne integrasjonen er gjort slik at Elvira II kan spille av filene på en korrekt måte, og slik at overganger mellom avspillingshastigheter kan gjøres best mulig.

Vi har gjort målinger for å vurdere tidsforbruk for de ulike stegene i prosessen. Disse viser at det for et fullskala-system ikke er aktuelt å bruke programvare for hele konstruksjonen av spolefiler. Dette tar for lang tid, her er det nødvendig å benytte spesiell maskinvare for enkelte steg.

Det er foretatt testing som viser at Elvira II ved hjelp av de integrerte filene klarer å levere en strøm som gir en bra avspilling. Bruk av spolefilene og overgangen mellom de ulike avspillingshastighetene fungerte også som det skulle. Målinger viser at vi enkelt kan konstruere spolefiler slik at ressursbruken ikke blir for stor.

Det konkluderes med at bruk av spolefiler er en fornuftig framgangsmåte for å oppnå ulike avspillingshastigheter for MPEG-2 i en videotjener.

Forord

Denne rapporten presenterer resultatet av en hovedoppgave utført ved institutt for datateknikk og informasjonsvitenskap, NTNU. Oppgaven er tilknyttet Elvira-prosjektet ved gruppen for databaseteknikk.

Oppgaven går ut på å undersøke hvordan VCR-funksjonalitet kan oppnås ved bruk av MPEG-2 i en videotjener. Forslag til løsninger beskrives, og disse vurderes med hensyn på bruk i databasegruppens videotjener Elvira II. Det implementeres en prototype av et system som kan brukes til å innføre VCR-funksjonalitet i denne tjeneren.

Jeg vil takke faglærer Roger Midtstraum og veileder Olav Sandstå for ypperlig råd og veiledning under arbeidet med denne oppgaven.

Trondheim, 22 desember 1997

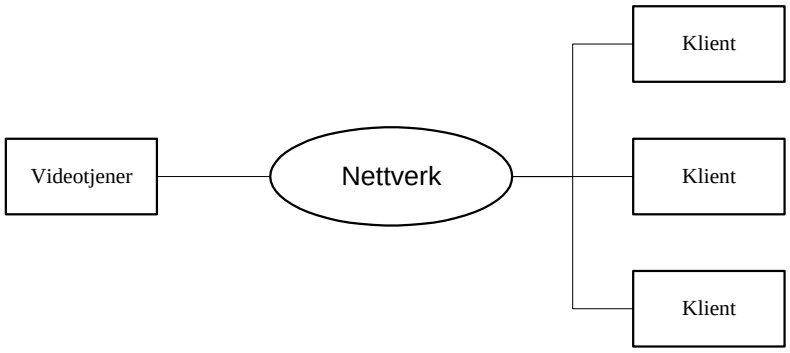
Daniel Remmem

Innholdsfortegnelse

Innledning

Det er i dag et stort behov for overføring av digital video over nettverk. Det finnes for eksempel allerede mange applikasjoner for levering av video over Internet. Felles for de fleste av disse er at kvaliteten på både lyd og bilde er relativt lav på grunn av begrenset nettverkskapasitet. Ny nettverksteknologi, som for eksempel ATM, gjør at det blir mulig å overføre video med langt bedre kvalitet. Dette åpner for muligheter til å lage digitale systemer som tilbyr levering av video med TV-kvalitet og enda bedre.

Video-on-demand (VOD) er et konsept som går ut på at en bruker har muligheten til selv å velge når og hvilken film han/hun vil se. Dette konseptet kan for digital video realiseres ved å lage en *videotjener*. En videotjener er et system som administrerer og leverer video til klienter (brukere). Figur 1 viser en prinsippskisse av hvordan en videotjener kan brukes til å kommunisere med og levere video til flere klienter samtidig over et nettverk.



Figur 1 Videotjener og klienter

For å belaste nettverket så lite som mulig vil det være nødvendig for videotjeneren å bruke komprimert video. Det finnes i dag flere standarder for å utføre denne komprimeringen. For høykvalitetsvideo er det imidlertid en standard som skiller seg ut, MPEG-2. Denne standarden er spesielt utviklet for komprimering av video med høy kvalitet og vil i framtiden kunne bli brukt i mange applikasjoner, blant annet Digital TV, video på DVD-format og annet.

En viktig egenskap for en videotjener er at brukeren har muligheten til å være interaktiv, dvs. spole og hoppe i videoen. Denne funksjonaliteten kan sammenlignes med spoling for en vanlig videomaskin og kalles derfor *VCR-funksjonalitet*. Bruk av en standard som MPEG-2 gjør at dette blir vanskelig. Dette kommer av at formatet på videoen er vanskelig å manipulere, det blir benyttet relativt kompliserte teknikker for å komprimere videoen så sterkt som mulig.

Elvira er et videotjener-prosjekt som databasegruppa har arbeidet med de siste årene. Disse videotjenerene har hovedsakelig brukt MJPEG (Motion-JPEG) som videoformat (men støtter også MPEG-1). MJPEG er et relativt enkelt format, og det kreves ikke kompliserte teknikker for å oppnå VCR-funksjonalitet i en video av dette formatet. Ulempen er at formatet har lav komprimeringsgrad, noe som fører til krav om stor lagringskapasitet og høy nettverkskapasitet for video av god kvalitet.

Databasegruppa utvikler nå en ny utgave av denne tjeneren, Elvira II. Denne skal i tillegg til MJPEG også støtte MPEG-2. For å oppnå VCR-funksjonalitet for MPEG-2 filmer må det på grunn av det kompliserte formatet brukes spesielle teknikker. Denne rapporten omhandler slike teknikker, og beskriver et forslag til hvordan det kan gjøres på best mulig måte. I begynnelsen av rapporten vil det gis generelt bakgrunnstoff om digital video og komprimering. MPEG-2 standarden vil også bli beskrevet nærmere.

Problemformulering

I dette avsnittet vil det bli gitt en beskrivelse av hva som skal gjøres i denne hovedoppgaven. Det vil først bli gitt litt informasjon vedrørende bakgrunnen for dette prosjektet. Deretter følger en mer konkret beskrivelse av hva som skal gjøres. Tilslutt vil det bli sagt noe om hvorfor det ikke er trivielt å utføre disse oppgavene, hva består vanskelighetene i ?

Bakgrunn

Databasegruppa utvikler for tiden en ny videotjener kalt Elvira II. Både denne og forløperen Elvira støtter Motion-JPEG (MJPEG). Dette formatet er populært i videotjenere fordi det er et enkelt format, det består av enkeltkodete bilder som legges sekvensielt etter hverandre. Dette gjør at man uten større vanskeligheter kan manipulere rekkefølgen og utvalget av disse bildene. Dermed blir det enkelt å oppnå spoling og annen VCR-funksjonalitet i dette formatet. Ulempen er at komprimeringen blir lav, vi må bruke en stor datamengde for å lagre video av en gitt kvalitet.

MPEG-komiteen (Moving Picture Expert Group) har utarbeidet standarder som i dag er ledende når det gjelder komprimering av digital video. MPEG-2 er den eneste av disse som er spesielt konstruert for video av høy kvalitet og avspilling med høy båndbredde. Standarden gir høy komprimering, men dette går på bekostning av enkelheten. Formatet er bygd opp på en slik måte at det blir mye vanskeligere å manipulere avspillingen, sammenlignet med MJPEG.

Det er dette som er utgangspunktet for oppgaven. Det er ønskelig at man i Elvira II skal kunne ha tilgang til VCR-funksjonalitet, også ved bruk av MPEG-2. Oppgaven går i hovedtrekk derfor ut på å undersøke hvordan MPEG-2 best mulig kan integreres i Elvira II på en måte som muliggjør VCR-funksjonalitet.

Hva skal gjøres?

Det skal finnes fram til en eller flere metoder som muliggjør VCR-funksjonalitet for MPEG-2. VCR-funksjonalitet innebærer at en bruker (klient) har mulighet til å foreta følgende operasjoner på en video:

- Vanlig avspilling av videoen.
- Hopp til et tilfeldig sted i videoen (tilfeldig aksess).
- Spol videoen *fremover* med en hastighet *lavere* enn normal avspilling (sakte film).
- Spol videoen *fremover* med en hastighet *høyere* enn normal avspilling (vanlig spoling).
- Spol videoen *bakover* med en hastighet *lavere* enn normal avspilling.
- Spol videoen *bakover* med en hastighet *høyere* enn normal avspilling.
- Vis et stillbilde.

En begrensning som legges på spolingen (alle typer) i Elvira II er at ikke *lyd* skal spilles av samtidig. Dette vil være en forenklende faktor.

Metodene som blir funnet skal vurderes med hensyn til kriterier som er viktige for Elvira II. Dette kan være kriterier som visuell kvalitet på spolingen, gjennomførbarhet (arbeidsmengde), nettverksbelastning, diskbelastning, og CPU/minne forbruk. Basert på denne vurderingen skal det velges en metode som anses som mest lovende. Denne metoden skal implementeres og testes ut i Elvira II.

Tilslutt skal det foretas målinger som undersøker hvordan denne metoden egner seg for bruk i Elvira II. Disse skal være basert på de ovennevnte kriteriene. Dersom metoden kan gjennomføres i forskjellige varianter, bør slike målinger danne grunnlaget for en sammenligning mellom de ulike variantene.

Vanskelighetene

Hva er så vanskelighetene med å få til VCR-funksjonalitet for MPEG-2? Som nevnt så er MPEG-2 konstruert for å komprimere dataene i størst mulig grad. Dette blir gjort ved teknikker som vanskeliggjør en senere manipulering av videoen. Komprimeringen har hovedsakelig bare en funksjon: å få datamengden for *normal avspilling* redusert i størst mulig grad.

Noen egenskaper for MPEG-2 som vanskeliggjør spoling og tilfeldig aksess sammenlignet med f.eks MJPEG er:

- **Bildeavhengigheter**
  For enkle formater kan man dekode et hvilket som helst bilde i en video uten å ha dekodet noe på forhånd. Dette er ikke tilfelle for MPEG-standardene. For å redusere datamengden maksimalt, utnytter man det faktum at etterfølgende bilder ofte er svært lik de foregående. Ved å kode bare forskjellene fra det forrige bildet trenger man mye mindre datamengde for å representere det nye bildet. Dette medfører imidlertid at man ikke kan dekode slike bilder alene, man er nødt til å dekode det forrige bildet først. Vi ser at vi får avhengigheter mellom de ulike bildene og ting som spoling og tilfeldig aksess kompliseres.

- **Innpakkingsformatet (systemlaget)**
  MPEG-2 innebefatter bruk av to forskjellige formater for innpakking av lyd og bilde: programstrømmen og transportstrømmen. Disse sørger for en sikker lagring, oversendelse og synkronisering av lyd og bilde. Transportstrømmen støtter også bruk av flere programmer (lyd og bilde). Denne innpakkingen fører til at man ikke uten videre kan manipulere/aksessere videoen som ligger skjult i formatet.

- **Synkroniseringsmekanismene**
  MPEG-2's systemlag synkroniserer bilder og lyd ved å bruke tidsstempler. Tidsstemplene styrer også hastigheten filmen skal vises med. Dette vanskeliggjør derfor spoling. De samme tidsstemplene som finnes for normal avspilling kan sannsynligvis ikke direkte brukes på samme måte under spoling.

Bakgrunn

Dette kapittelet er laget for å gi leseren en bakgrunn i de emner som denne oppgaven bygger på.

Først skrives det litt om hvordan selve videoen med lyd og bilde blir representert på en digital måte. Deretter gir vi en introduksjon til komprimering av data generelt, men med spesiell vekt på komprimering av lyd og bilde. Så skrives det litt om dagens nettverksteknologi som kan brukes til å overføre video.

Til slutt kommer det viktigste i dette kapittelet, en beskrivelse av MPEG-2 standarden. Det tas sikte på å gi leseren en god innføring i de viktigste prinsippene for MPEG-2. Mest vekt vil det bli lagt på oppbygging av systemstrømmen og synkroniseringsmekanismer.

Det meste av informasjonen i dette kapittelet er hentet fra Remmem/Østerhagen (1997). (Dette var et prosjekt ved IDI, der undertegnede var en av deltagerne)

Digital video

Denne rapporten vil i stor grad omhandle digital video og metoder tilknyttet dette emnet. Dette avsnittet tar sikte på å gi leseren en bakgrunn i hovedprinsippene som digitalisering av video bygger på.

Digitale Bilder og Bildesekvenser

Digitalisering av et bilde skjer ved at bildet deles opp i linjer, og at hver linje deles opp i punkter. Et slikt punkt kalles en *pixel* eller en *pel* (picture element). Hver pel representeres ved en bitkode. Denne koden forteller hvilken farge og intensitet punktet skal ha.

For å kunne representere et videobilde kodet ved hjelp av den europeiske TV-standarden, PAL, kreves det en oppløsning på rundt 720x576 pels og bitkoder på 24 bit. Dette betyr at *ett* bilde vil oppta rundt 1.2 MByte med data.

En digital videosekvens[1] vil bestå av en serie digitaliserte bilder. Vanligvis dreier det seg om 25 eller 30 bilder i sekundet. Om vi tar utgangspunkt i eksemplet over vil da et sekund med video bruke maksimalt 30 MByte. Dette tilsvarer 180 GByte for en sekvens på 100 minutter. Ved hjelp av forskjellige komprimeringsteknikker vil denne datamengden kunne reduseres med en faktor på 30 uten at kvaliteten på den rekonstruerte videosekvensen blir merkbart dårligere.

Det eksisterer i dag en rekke standarder for komprimering av digitale videosekvenser. En del av disse blir presentert under, men først gis det en kort innføring i fargerepresentasjon.

Fargerepresentasjon

Farge er det perseptuelle resultatet av lys med bølgelengde mellom 400nm og 700nm som treffer retinaen. På grunn av at det eksisterer tre forskjellige fargesensorer i øyet, er tre numeriske komponenter nok og nødvendig for å beskrive fargerommet. RGB (Red-Green-Blue) og CMY(Cyan-Magenta-Yellow) er begge modeller som baserer seg på bruk av tre fargekoeffisienter. RGB brukes for CRT baserte monitorer, mens CMY[2] benyttes for utskriftsenheter.

[1] *Videosekvens* henspeiler i denne sammenheng på en sekvens utelukkende bestående av bilder. Begrepet *video* vil i det følgende bli brukt for å indikere en sekvens som inkluderer både bilde og lyd.
[2] I dag er CMYK den mest brukte fargemodellen for utskriftsenheter. K står for svart.

Man kan adskille *luminanse* (eller intensitet) og *krominanse* (eller farge). Luminanse betegner den persepterte intensiteten til et objekt med egen utstråling av lys (for eksempel en lyspære eller solen). Krominanse brukes til å beskrive forskjellige farger som rød, blå, gul og grønn. Dette gjør det mulig å definere en en-til-en mapping mellom fargekoeffisienter og luminanse/krominanse koeffisienter. Både europeiske (PAL/SECAM) og amerikanske (NTSC) standarder for TV baserer seg på dette. I Europa opererer man med et fargerom som kalles YUV. Y står luminanse, mens krominansen er kodet inn i U og V koeffisientene. I Amerika benyttes fargerommet YIQ. Dette er identisk med YUV rommet, men med 33 graders rotasjon i UV.

For å forstå hvorfor YUV/YIQ benyttes fremfor fargekoeffisienter, er det viktig å være klar over at menneskets visuelle system er mer sensitivt ovenfor endringer i luminanse enn i krominanse. En følge av dette er at U og V kan representeres med mindre nøyaktighet enn Y. Dette er et viktig poeng for både analog og digital representasjon av farger.

Den digitale versjonen av YUV kalles YC_bC_r. C_b tilsvarer analog U, og C_r analog V. Det er dette formatet som brukes i MPEG standarden. Denne utnytter egenskapene til luminanse og krominanse, og gir mulighet til å kode mest informasjon inn i Y komponenten.[3]

Teknikken med å legge mest informasjon inn i Y kalles *subsampling.* Dette vil si at C_b og C_r komponentene måles med av lavere frekvens enn Y. Forholdet mellom Y og C_b/C_r skal være et helt tall. En standard måte å oppgi dette forholder er notasjonen:

Y målingssfrekvens : C_b *målingsfrekvens* : C_r *målingsfrekvens*

For eksempel vil 4:2:2 angi at C_b og C_r måles med halvparten av Y's målingsrate.

Videostandarder

Det eksisterer i dag en hel del standarder for koding og dekoding («kodeker») av digital video. Noen av disse er åpne, andre proprietære. F.Fluckiger (1995) beskriver følgende videostandarder:

H.261, en standard utviklet av ITU-T, er en standard for videokompremering som er utviklet for å muliggjøre videokonferanser over ISDN. Båndbredden til H.261 er et helt multiplum av 64 kbps, maksimum 1.92 Mbit/s. Spesifiserte oppløsninger er 360x180 og 180x90. Standarden er tilpasset sanntidskoding og dekoding. *H.263* er en forbedret versjon av H.261.

MPEG (moving pictures expert group) er en komité opprettet av ISO som utvikler standarden under samme navn. Det eksisterer tre ferdige MPEG standarder (Mer informasjon kan hentes fra WWW-siden «MPEG Moving Pictures Expert Group Information»).

> *MPEG-1* er konstruert for båndbredder opp til 1.5 Mbps. En vanlig bruk av denne båndbredden er videosekvenser med oppløsninger 352x240 med 30 bilder i sekundet, samt to lydkanaler à 125 Kbit/s. Standarden er optimalisert for avspilling fra CD.
>
> *MPEG-2* er optimalisert for høyere båndbredder enn MPEG-1. Standarden tilbyr et bredt utvalg av oppløsninger og støtter opp til 5 lydkanaler[4]. I tillegg er MPEG-2 tilpasset *mellomlinjert* video (interlacing), noe som i stor grad er benyttet i dagens TV standarder. Den har også støtte for leveranse av videostrømmer over nettverk.
> Se avsnittet om MPEG-2 for en utførlig beskrivelse av MPEG-2 spesifikasjonen.
>
> *MPEG-4* er tilpasset meget lave båndbredder - opp til 64 Kbit/s. Dette gir en oppløsning på 176x144 pels med 10 bilder per sekund. Lydsignalet kan ha en båndbredde fra 2 til 64 Kbit/s. Standarden er ment for bruk i videotelefoner og lignende utstyr.

Indeo er Intel's proprietære videostandard. Denne er beskrevet på WWW-siden «Indeo Video». Den tilbyr mange av de samme mulighetene som MPEG, men gir dårligere kompresjon. Med en oppløsning på 320x240 og 15 bilder i sekundet bruker Indeo 150 Kbit/s.

[3] Se diskusjonen om *kroma format* i avsnittet om MPEG-2.
[4] 5 kanaler er nødvendig for «surround sound».

Cinepak er en proprietær videostandard utviklet av Radius. Den er tilpasset avspilling fra CD-ROM. Koding av video med Cinepak er derfor en prosessorintensiv prosess, mens dekoding er en lettvektsprosess.

MOTION-JPEG benytter seg av JPEG spesifikasjonen for komprimering av enkeltbilder. MJPEG legger rett og slett sekvensielt kodede JPEG bilder etter hverandre. Strengt tatt er ikke dette en standard, men en teknikk.

Quicktime er en arktitektur for visning og editering av video, animasjon, tekst og annen dynamisk informasjon på flere plattformer. Den er utviklet av Apple og støtter flere kodeker, bl.a MJPEG og Cinepak.

Digital Lyd

Lyd digitaliseres ved at et innsignal måles (samples) med en gitt frekvens. De målte verdiene blir diskretisert og lagres fortløpende. Som nevnt over vil det være tettheten av måleverdiene og oppløsningen på diskretiseringen som bestemmer hvor nært opp til det analoge signalet man kommer.

Med *målefrekvens* menes antall målinger per tidsenhet, mens *kvantisering* er et annet uttrykk for diskretisering.

Den mest brukte målefrekvens i dag er 44.1kHz. Dette er frekvensen som anvendes i CD-standarden. Hver måleverdi bruker 16 bit. Dette gir et frekvensområde fra 0 til 22.5 kHz og en båndbredde på 88,2 kB/s per lydkanal. Lyden til en stereo langfilm på 100 minutter vil dermed kreve rundt 1GByte. Med komprimering vil denne datamengden kunne halveres uten at det perseptuelle resultatet forringes.

MPEG Audio er den vanligste standarden for komprimering av lyd i tilknytning til video. Den benytter seg av transformasjonskoding og ikke-lineær kvantisering. Se avsnittet MPEG-2, Audio for mer informasjon.

Komprimering

For å representere lyd og bilde i en video trengs det store mengder data. Som vist i forrige kapittel om digital video vil en spillefilm på 100 minutter av TV-kvalitet som ikke er komprimert kunne kreve en datamengde på nærmere 200 GByte. Vi ser at det er nødvendig å minske denne mengden betraktelig.

Det er to aspekter som gjør komprimering nødvendig; det ene er at vi vil selvfølgelig bruke minst mulig plass ved lagring av video på et digitalt medium. Det andre er nødvendigheten av rask overføring over nettverk. Dersom en videosekvens skal overføres "live" er det helt nødvendig å komprimere for at man skal ha mulighet til å overføre video med akseptabel kvalitet.

Å komprimere en datamengde vil si å kjøre denne gjennom en prosess som genererer en ny datamengde. Denne datamengden vil være mindre enn den originale. Ved å kjøre datamengden gjennom en motsatt prosess kan vi reprodusere den originale datamengden, evt konstruere en datamengde med tilnærmet samme egenskaper.

Vi skal i det følgende se litt på komprimering generelt; hva er det som gjør komprimering mulig, og hvilke egenskaper har komprimering ?

Generell informasjon om ulike komprimeringsmetoder er hentet fra F.Fluckiger (1995).

Tapsfri og Ikke-tapsfri komprimering

En viktig egenskap som skiller forskjellige komprimeringsmetoder er om den dekodede informasjonen er identisk med den originale eller ikke. Dersom den er det, kaller vi prosessen for en *tapsfri* komprimering, dvs. vi mister ikke noe informasjon ved komprimeringen. Denne form for komprimering blir brukt i tilfeller der det er kritisk at datamengden som kommer fram er korrekt, f.eks. i situasjoner der en datamaskin skal tolke et program eller data.

Ikke-tapsfri komprimering gjør at den kodede informasjonen ikke kan dekodes slik at vi får en datamengde identisk med den originale. Det betyr at informasjon har gått tapt ved komprimeringen. Vi kan også kalle en slik prosess irreversibel, det er ikke mulig å komme tilbake til utgangspunktet. Fordelen med denne type komprimering er at vi kan få en mye større grad av komprimering enn ved tapsfri komprimering.

Ikke-tapsfri komprimering blir først og fremst brukt i situasjoner der mennesker skal oppfatte ting, som i lyd og bilde. Dette kommer av at mennesker ikke trenger en perfekt gjengivelse av lyd og bilde for å få et tilnærmet like godt inntrykk av bildet / lyden. Ifølge F. Fluckiger er det gjort undersøkelser som viser at bilder kan komprimeres ganske mye vha. ikke-tapsfri komprimering før mennesker klarer å skille mellom det ukomprimerte bildet og det komprimerte. Dette faktum gjør at det så godt som alltid blir brukt ikke-tapsfri komprimering når det er snakk om lyd og bilder (videosekvenser).

Metoder for komprimering

Det finnes mange forskjellige metoder som blir brukt for å komprimere en strøm av data. Disse metodene har hver sine fordeler og bakdeler og det kan ofte lønne seg å benytte flere metoder under en komprimeringsprosess. Metodene kan plasseres i to hovedkategorier: *Mengdekoding* (Entropy encoding) og *Kildekoding* (Source encoding).

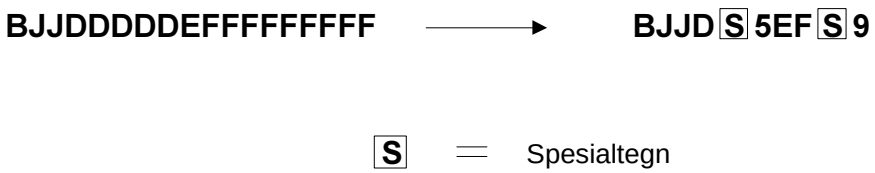
Mengdekoding

Dette er den enkleste formen for koding. Prinsippet for disse teknikkene er at man kun ser på datastrømmen som en rekke av byte, man ser ikke på hvilken rolle de ulike informasjonsbitene spiller i den store sammenhengen. Det gjelder kort sagt bare å erstatte lengre datasekvenser med kortere sekvenser som inneholder den samme informasjonen. Dette betyr at mengdekoding er tapsfri, noe som er en viktig egenskap ved denne formen for koding.

Vi skal nevne to metoder som faller inn under begrepet mengdekoding: *Løpelengdekoding* og *Huffmankoding*.

Løpelengdekoding

En viktig og enkel metode er løpelengdekoding (Run-length encoding). Den går ut på at en byte / et tegn som forekommer flere ganger blir erstattet av tegnet + et spesialtegn + et tall som angir antall forekomster av dette tegnet. Denne metoden vil egne seg godt i tilfeller der det er antatt at datastrømmen vil inneholde mange like tegn etter hverandre.



Figur 2 Et eksempel på løpelengdekoding.

Huffmankoding

Dette er en annen metode som er mye brukt. Dette er en statistisk metode som går ut på at man oppretter en tabell over frekvensene til bitsekvenser som blir brukt i en gitt datastrøm. De sekvensene som oftest blir brukt vil bli representert med en kort kode, mens sjeldnere brukte bitsekvenser vil få en lengre kode. På denne måten vil mengden data som må overføres være minimal. Et eksempel på et slikt system er kodesystemet *morse*.

Kildekoding

Denne formen for koding betyr at dataene blir omformet på en måte som er avhengig av semantikken til dataene, f.eks hvilken rolle de spiller i et bilde. Komprimeringsgraden[5] ved slike metoder kan variere sterkt avhengig av egenskapene til dataene, men ved gunstige tilfeller kan den bli meget høy sammenlignet med mengdekoding.

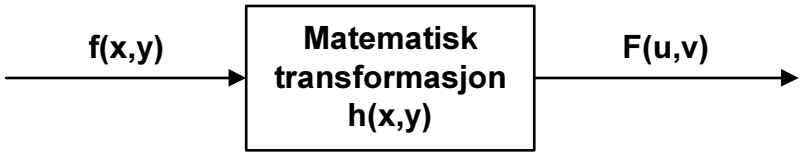
Det finnes tre hovedmetoder som kan kategoriseres som kildekodingsmetoder*: koding ved transformering, differansekoding* og koding ved *vektorkvantisering*.

Koding ved transformering

Ved denne typen koding blir det foretatt en matematisk transformasjon på datamengden som skal komprimeres. Se Figur 3 for en illustrasjon av en transformasjon. Transformasjonen som utføres bør velges slik at den på best mulig måte gjør datamengden egnet for senere komprimering. En ofte benyttet metode i forbindelse med komprimering går ut på at verdier i den opprinnelige datamengden bli transformert slik at de blir representert i et frekvensdomene. De mest signifikante verdiene vil kunne bli representert ved lave frekvenser, mens de mindre signifikante verdiene vil kunne representeres av høye frekvenser. Dette gjør at det blir lett å skille ut de ulike frekvensene, og dermed kan man behandle disse frekvensene på en slik måte at komprimering blir mer effektivt. Denne behandlingen vil som oftest medføre at informasjon går tapt.

Transformering er særlig egnet for bruk på bilder siden mye bildeinformasjon kan taes vekk uten at det går utover perseptuell kvalitet. Transformering i seg selv er en tapsfri prosess, men dersom den transformerte datamengden blir behandlet for å letteregjøre komprimering vil hele prosessen bli ikke-tapsfri.

[5] Komprimeringsgrad - et uttrykk for hvor mange ganger en datamengde kan bli komprimert / redusert.



Figur 3 Prinsippet ved transformasjon

Differensiell koding

Dette er metoder der bare differansen mellom en verdi og en antatt (predikert) verdi blir kodet. Disse metodene kalles derfor også prediktive metoder. Dette prinsippet er godt egnet for video, da det i videosekvenser ofte bare er små forskjeller fra et bilde til det neste. En enkel måte å implementere dette prinsippet på er å anta at det neste bildet er likt det nåværende, dvs. vi koder kun differansen.

Dette vil imidlertid ikke fungere særlig godt når videosekvensen består av mye bevegelse. I stedet brukes det da en teknikk som kalles *bevegelseskompensasjon* (motion compensation). Ideen her er å finne en bevegelse i videosekvensen for deretter å bruke denne bevegelsen til å predikere det neste bildet. Denne bevegelsen kan bli representert ved hjelp av vektorer. Deretter kan vi foreta en subtraksjon mellom det forrige bildet og det predikerte bildet. Nødvendig informasjon som må kodes blir da bare bevegelsesvektorene og prediksjonsfeilen (differansen).

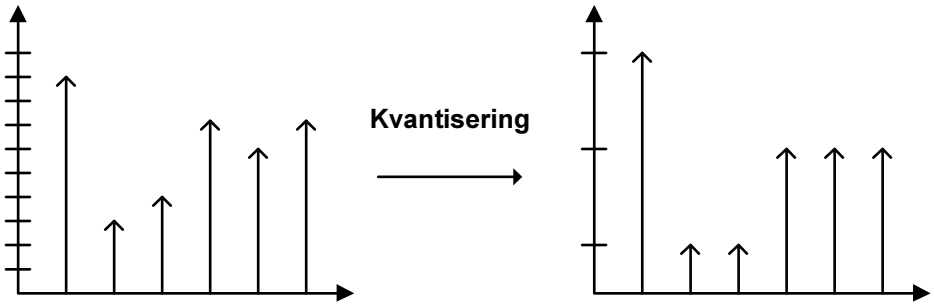
Koding ved vektorkvantisering

Denne metoden er relativt avansert, men dersom den blir bra utnyttet kan den gi en svært høy komprimeringsgrad. Prinsippet går ut på at datastrømmen blir delt opp i vektorer og det blir på forhånd konstruert en kodebok som inneholder mange mønster. For hver vektor gjelder det å finne det mønsteret som ligger nærmest vektoren. Det eneste som trengs å kodes er indeksen til mønsteret samt differansen mellom vektoren og mønsteret. Det er klart at jo nærmere vektoren ligger mønsteret jo mindre differansedata trengs å kodes.

Det er svært viktig å konstruere en kodebok der mønstrene ligger tett opp til vektorene i datastrømmen. For at denne metoden skal bli en suksess er det derfor en stor fordel å kjenne egenskapene til datastrømmen som vi skal komprimere.

Kvantisering

Kvantisering er en teknikk som ofte blir brukt i ulike sammenhenger, og særlig i forbindelse med koding av lyd og bilde. En datamengde kan inneholde målinger (funksjonsverdier), og disse målingene vil bruke en bestemt skala. Hovedprinsippet ved kvantisering er at nøyaktigheten på denne skalaen blir redusert . Det vil si at målingene blir mindre nøyaktige, mange målingsverdier vil bli flyttet til den nærmeste nye verdien på målingsskalaen. Dette betyr at flere målinger vil havne på samme verdi. Denne prosessen er derfor egnet for å gjøre datamengder mer egnet for komprimering, særlig mengdekoding vil bli mer effektivt. Prosessen er ikke-tapsfri, det er ikke mulig å komme tilbake til utgangspunktet. Likevel gir den bra resulater for lyd og bilder.



Figur 4 Prinsippet ved kvantisering.

Hvorfor fungerer kompresjon så bra?

Kompresjon vil som oftest gi svært gode resultater, men komprimeringsgraden vil i stor grad variere avhengig av ulike egenskaper til dataene. I dette kapittelet blir det sett på hvilke faktorer som ligger til grunn for at komprimeringen skal bli mest mulig vellykket. Det vil bli lagt vekt på komprimering av bilder og lyd.

Romlig redundans

At et bilde har høy grad av romlig redundans (spatial redundancy) betyr at det finnes mye lik informasjon i et bilde. For eksempel vil et bilde av et ensfarget objekt inneholde mye romlig redundans siden mange piksler vil bestå av den samme fargen. En høy grad av romlig redundans vil gjøre komprimering lettere og mer effektivt. Det er først og fremst mengdekoding som drar stor nytte av denne egenskapen til et bilde, de mange like verdiene vil kunne erstattes med kortere kodede sekvenser.

Temporær redundans

Dette er en egenskap som kan tilknyttes sekvenser av bilder. Som allerede nevnt vil vi i en videosekvens finne mange bilder som er veldig like. Vi sier at vi har høy temporær redundans (temporal redundancy) når etterfølgende bilder ikke inneholder mange endringer. Ved filming av et stillestående landskap vil dette i høy grad være tilfelle. Ved komprimering ved hjelp av bevegelseskompensasjon / differensiering vil høy temporal redundans være en viktig faktor når det gjelder å oppnå en god komprimering.

Psykovisuelle / Psykoakustiske aspekter

Ved presentasjon av lyd og bilde er det som regel et menneske som er mottakeren. Inntrykkene oppfattes og bearbeides av hjernen. Det er som nevnt ikke nødvendig med en perfekt gjengivelse av bilder og lyd for at inntrykkene allikevel skal bli gode. Det kan tillates en overraskende stor fjerning av informasjon uten at den oppfattede kvaliteten forringes nevneverdig. (Dette forutsetter selvsagt at den rette informasjonen fjernes!)

Dette betyr at vi kan tillate bruk av ikke-tapsfrie komprimeringsmetoder når resultatet skal presenteres for mennesker (noe lyd og bilde som oftest skal). Dette er et sentralt punkt når det gjelder komprimeringprosessen for lyd og bilder.

Ved komprimering av bilder og video blir dette best utnyttet ved å foreta en transformasjon (se «Koding ved transformering») slik at bildet blir overført til et frekvensplan. Ved god kjennskap til hvordan de ulike frekvensene påvirker menneskets oppfattelse av bildet (høye frekvenser er minst signifikant) kan det benyttes metoder som effektiviserer komprimeringen. *Kvantisering* er en mye brukt slik metode. (se «Kvantisering»)

Ved komprimering av lyd blir det ofte brukt *psykoakustiske modeller*. Disse beskriver hvordan lyden oppfattes av et menneske og hvilke deler av lyden som er lite betydelig for oppfattelsen og som derfor kan fjernes / reduseres. Se avsnittet «MPEG 2-Audio», side 34, for en nærmere beskrivelse.

Aktuell nettverksteknologi

Nettverket er blitt en viktigere og viktigere del av all datateknologi. Dette gjelder også for digital video. For å kunne tilby tjenester som video-on-demand, fjernlæring og videokonferanser er man avhengig av nettverksteknologi som har den båndbredde og funksjonalitet som kreves.
I dette kapittelet vil to av de viktigste nettverksteknologiene presentert, *ATM* (Asyncronous Transfer Mode) og *TCP/IP* (Transfer Control Protocol / Internet Protocol)

Informasjonen er hentet fra McDysan og Spohn (1995), og R.Stevens (1995).

ATM

ATM (Asynchronous Transfer Mode) er en teknologi som kan bli brukt både i WAN (Wide Area Networks) og LAN (Local Area Networks). Teknologien tar sikte på å tilby brukerene høy båndbredde samt overføringer som sikres en garantert kvalitet. Det siste prinsippet kalles Quality of Service og kommer av ulike krav som nettverksapplikasjoner i dag stiller. Video- og audio-applikasjoner vil gjerne ha en konstant båndbredde med minimale forsinkelser, mens i andre applikasjoner er det ikke kritisk med en jevn bitstrøm, det er bare den totale overføringstiden som teller.

ATM definerer 4 forskjellige klasser for kvalitetsservice. Klasse A tilbyr fast overføringstid med konstant bitrate. Dette kalles også kretsemulering (circuit emulation). Klasse B er mest aktuell for video og audio applikasjoner, den tilbyr også fast overføringstid, men med variabel bitrate. Klasse C har ingen krav om overføringstid og bruker variabel bitrate. Det samme gjelder for klasse D, men denne gjør ikke bruk av koblinger (connectionless service).

For å sikre mulighetene til å oppfylle ulike krav bruker ATM en spesiell pakketeknologi. En pakke i et ATM-nettverk kalles en celle og er av en fast størrelse på 53 byte. Denne lille, faste størrelsen gjør det mulig å foreta pakkesvitsjing som er enkel, og dermed rask og pålitelig. Den reduserer også forsinkelser, noe som er en fordel når det gjelder video og audioapplikasjoner. Man kan sette opp virtuelle koblinger og stier mellom to noder gjennom ATM-svitsjer, dette vil si at teknologien er forbindelsesorientert. Dette gjør det lettere å garantere gitte betingelser.

For å sende større pakker over nettverket kan applikasjoner benytte seg av ATM Adaption Layer, AAL. Dette finnes i fire forskjellige utgaver[6] med forskjellig funksjonalitet. Figur 5 viser plasseringen av AAL i et tilfelle hvor det benyttes en annen nettverksprotokoll(TCP/IP) over ATM.

ATM-nettverk har feildeteksjonsmekanismer, men tilbyr ikke oppretting av feil. Celler med feil blir ganske enkelt ignorert. Video og audio applikasjoner vil ha en fordel av dette, man har allikevel ofte ikke tid til å vente på retransmisjon.

Vi ser at ATM-nettverk egner seg svært godt for video og audioapplikasjoner der man benytter seg av «live» overføringer. Det største problemet er at den lille cellestørrelsen gjør det til en relativt tung prosess å dele opp videofilmer for sending av data i programvare. En løsning på dette problemet, ved siden av å benytte AAL, er å kjøre en nettverksprotokoll med større pakkestørrelse over ATM, gjerne TCP/IP. Oppdeligen av pakker til celler vil da skje i maskinvare. Det finnes også en del andre årsaker til at TCP/IP benyttes over ATM. Dette kommer vi tilbake til i neste avsnitt.

TCP/IP

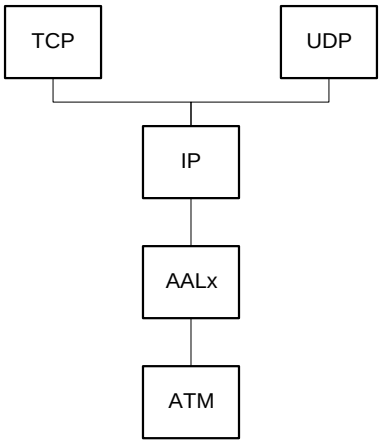
TCP/IP er det protokollsettet som er mest i bruk i nettverk i dag. Den tilbyr kommunikasjon mellom datamaskiner over LAN og WAN nettverk og danner basisen for Internett slik vi kjenner det i dag. Det er et åpent protokollsett som benytter seg av pakkesvitsjing. Pakkene i TCP/IP går vanligvis under betegnelsen *datagram.*

[6] AAL1, AAL2, AAL3/4, AAL5

TCP/IP er et sett med protokoller som tilbyr forskjellige lagdelte tjenester. TCP, UDP og IP er tre av disse protokollene.

IP står ansvarlig for transport av datagram over nettverket. Protokollen garanterer ikke at et datagram kommer frem til målet. Dette betyr at den er *usikker*. IP lagrer heller ikke noen tilstandsinformasjon om etterfølgende datagram. Dette gis betegnelsen *forbindelseløs*.

TCP sikrer flyt av data mellom to verter på nettverket. Ved hjelp av bl.a. segmentering av datagram og retransmisjon garanterer den at et datagram kommer frem til målet (om mulig). TCP er *forbindelseorientert.* For å overføre data med TCP må det først opprettes en forbindelse mellom de to maskinene som skal kommunisere. Over denne forbindelsen blir det overført en byteorientert strøm med data. Når begge parter er ferdige, tas forbindelsen ned.



Figur 5 TCP/IP over ATM

UDP er en protokoll som i motsetning til TCP ikke er forbindelseorientert. En skriveoperasjon fra et overliggende applikasjonslag fører til at et UDP datagram blir generert. Dette fører videre til at et IP datagram blir sendt ut på nettverket. Det finnes ingen garanti for at denne pakken når målet. Det at UDP ikke er forbindelseorientert gjør den i stand til å utføre såkalt *multicast*, dvs at en UDP pakke har flere maskiner som destinasjon.

Som nevnt tidligere brukes TCP/IP gjerne over et ATM nettverk. Årsaken til dette, ved siden av det rent ytelsemessige, er at TCP/IP er en protokoll som har oppnådd en mye bredere standardisering enn ATM. Dette betyr at applikasjoner konstruert for TCP/IP vil være mer portable.

TCP/IP kan ikke garantere en minimum overføringshastighet. Den gir en såkalt «best-effort» ytelse for levering av pakker. Det er ikke mulig å reservere ressurser, og som en følge finnes det intet QoS begrep for protokollen. Ved bruk av TCP/IP over ATM mister man altså denne funksjonaliteten.

I de tilfeller hvor det skal sendes video eller andre sanntids-strømmer over et nettverk, fortrekkes UDP. Grunnen til dette er at UDP ikke foretar retransmisjon av tapte pakker. I tillegg kan det i en del tilfeller være ønskelig å benytte seg av multicasting.

MPEG-2

«The Moving Pictures Expert Group» eller MPEG er en komite som ble opprettet av ISO (International Standards Organization) i 1988. Dens oppgave er å spesifisere standarder for kompresjon av audio og video.

I 1992 ble komiteens første resultat presentert under navnet MPEG-1 eller ISO/IEC 11172. Dette var en standard som var optimalisert for avspilling ved båndbredder opp til 1.5 Mbit/s, og var først og fremst rettet mot avspilling fra CD-ROM. Kvaliteten på den avspilte videoen lå på samme kvalitet som VHS standarden.

Så ble en ny fase i komiteens arbeid innledet. Målet var nå å spesifisere en standard som kunne kode TV signaler. Den største utfordringen var det faktum at vanlige TV signaler er *mellomlinjerte.* I tillegg skulle den nye standarden ha støtte for surround-lyd. Standarden ble kalt MPEG-2.

Dette kapittelet oppsummerer resultatet av dette arbeidet slik det er lagt frem i ISO/IEC 13818 spesifikasjonen. Denne spesifikasjonen er delt opp i følgende deler:

- ✓ Systemlaget («Systems»)
- ✓ Video
- ✓ Audio
- ✓ Overenstemmelse («Conformance»)
- ✓ Simulering i programvare
- ✓ Digital Storage Media - Control and Command («DSM-CC»)
- ✓ Real-Time Interface («RTI»)

I dette kapittelet vil det bli lagt vekt på de deler av spesifikasjonen som er interessante med henblikk på konstruksjon av en videotjener, nemlig systemlaget, video, audio og DSM-CC.

Video

ISO/IEC 13818-2 spesifiserer kodingen av MPEG-2 videostrømmer. Dette er en beskrivelse av hvordan en slik videostrøm skal *dekodes*. Følgelig blir det ikke oppgitt noen retningslinjer for hvordan en koder skal konstrueres.

I dette kapittelet blir det først gitt en fremstilling av oppbygningen til en MPEG-2 videostrøm. Deretter beskrives de forskjellige kompresjonsteknikkene som benyttes, og til slutt tar vi for oss begrepene skalerbarhet, profiler og nivåer.

Hvor ikke annet er nevnt er ISO/IEC 13818-2 brukt som referanse.

Videostrøm

En MPEG-2 videostrøm er hierarkisk oppbygd. Den er definert slik at det kan gjøres en entydig dekoding av strømmen. Følgende elementer er beskrevet i spesifikasjonen: sekvenser, gruppe-med-bilder, bilder, skiver, makroblokker og blokker. Figur 6, fra Koteng (1996), viser sammenhengen mellom de forskjellige elementene.

Sekvens

En sekvens defineres som en serie av gruppe-med-bilder (GOP) eller bilder. Den innkapsles med et sekvenshode og en sekvenshale. Sekvenshodet inneholder informasjon om hvordan de etterfølgende bildene skal dekodes. Dette er informasjon som dekoderen må ha for å dekode riktig. En strøm vil derfor alltid starte med et sekvenshode. Sekvenshoder kan gjentas utover i strømmen, dette vil tillate tilfeldig aksess inn i strømmen. Å plassere et sekvenshode foran hver GOP vil gjøre at strømmen kan aksesseres flest mulig steder.

Det er også mulig at informasjonen i sekvenshodet vil være ulikt for ulike deler av filmen, i en slik situasjon *må* sekvenshodet gjentas.

Gruppe-med-bilder (GOP for Group-Of-Pictures)

En GOP er en serie med bilder som kan tillate tilfeldig aksess inn i en videostrøm. Dette er ikke et påkrevet element i en MPEG-2 videostrøm. Første bilde i en GOP vil alltid være av rammetype I. En serie kan inneholde både *linje*- (field) og *ramme*- (frame) bilder. Se neste avsnitt for en beskrivelse av de forskjellige typene med bilder.

Bilde

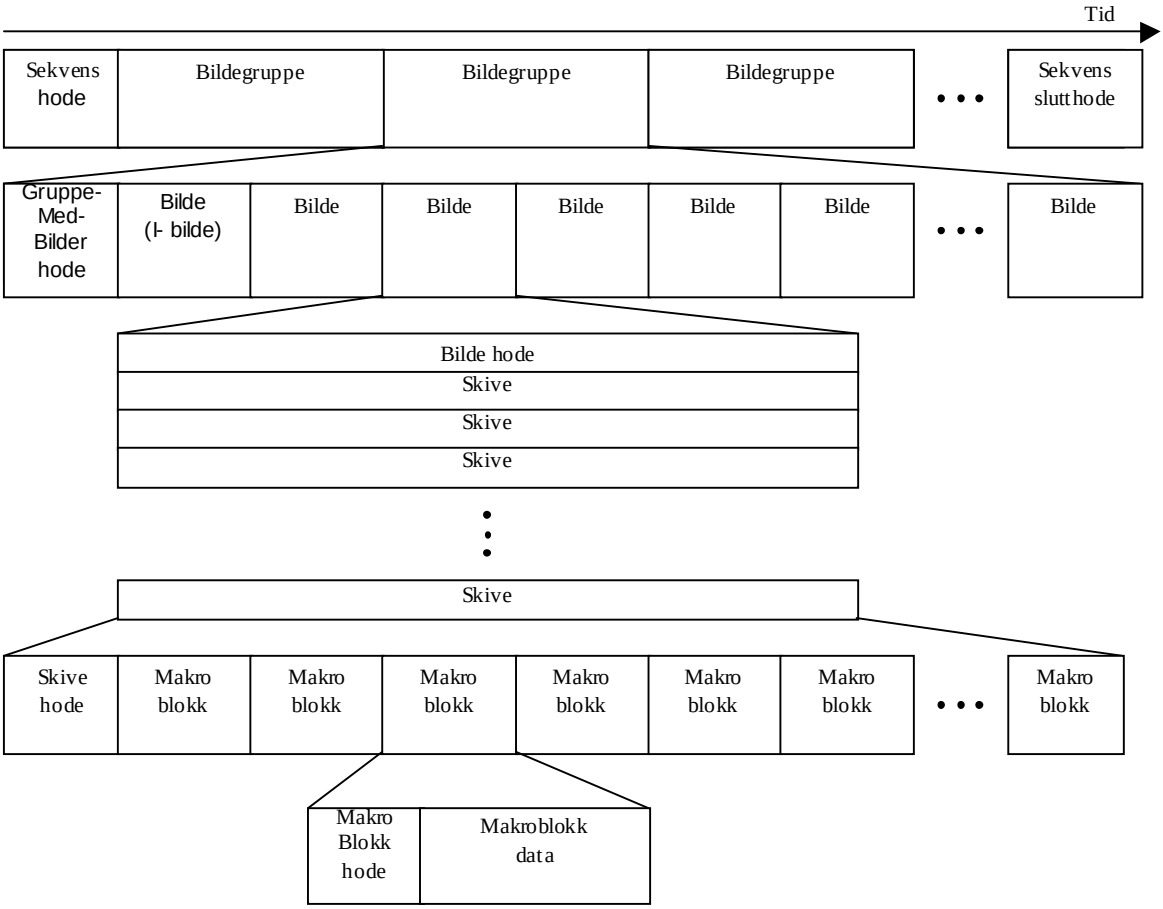
Et bilde eller en *ramme* er en mengde data som definerer et (eller halvparten av et) frittstående bilde i en videosekvens. En slik sekvens kan være progressiv eller mellomlinjert (interlacet). MPEG-2 støtter begge disse typene video. For mellomlinjert video kan et bilde være *linje*- eller *rammebasert*. Rammebaserte bilder er «hele» bilder hvor de odde og like linjene fra en mellomlinjert video er satt sammen. Linje-baserte bilder er «halve» bilder hvor odde og like linjer er satt sammen til hver sine bilder.

I tillegg kan bilder kodes på tre forskjellige måter; som I-rammer, P-rammer og B-rammer[7].

I-rammer (intra-ramme) er bilder hvor kun romlig redundans brukes for komprimering, dvs det brukes kun intra-ramme komprimering på disse bildene. På grunn av denne uavhengigheten fra andre bilder, brukes I-rammer som referanser ved tilfeldig aksess i en videostrøm. Hyppigheten av I-rammer avgjør den mulige aksessoppløsningen. Kompresjonen til slike bilder er dårligere enn P-rammer og B-rammer. For å lagre en pel kreves det i gjennomsnitt 2 bit.

P-rammer (predikativt bilde) er bilder som er kodet i forhold til nærmest foranliggende I- eller P-ramme. Dette er vist i Figur 7. De bruker bevegelsekompensasjon for å tilby bedre kompresjon enn I-rammer.

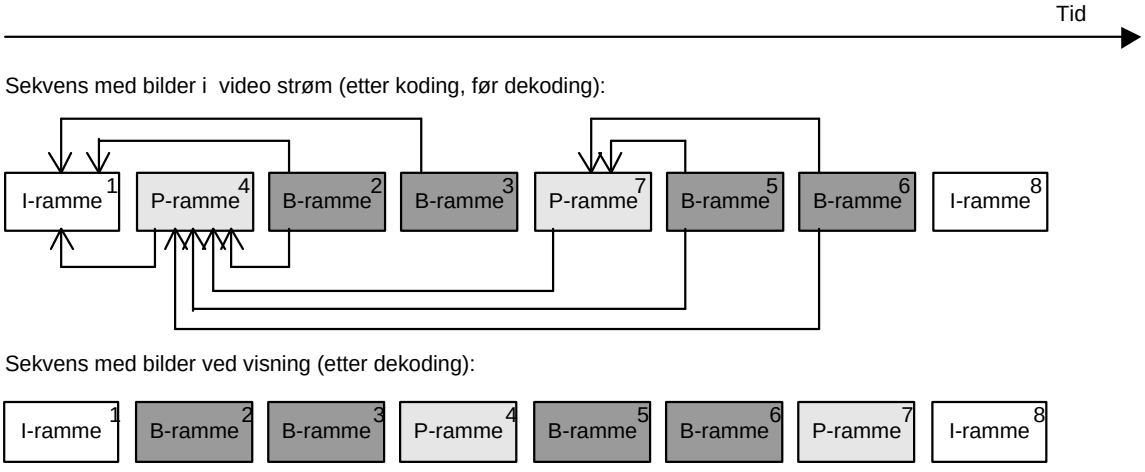
[7] I MPEG-1 var det definert en bildekoding til; D-ramme. Denne er ikke en del av MPEG-2 spesifikasjonen.



Figur 6 MPEG-2 elementer. Hentet fra Koteng (1996).

B-rammer (bi-direksjonal ramme) er bilder som kan bruke både foranliggende og etterliggende I og/eller P-rammer som referanse ved koding. Kompresjonen blir derfor bedre enn ved de andre alternativene, men krever større ressurser. Om en sekvens inneholder B-rammer, vil det være nødvendig at bildene sendes i en annen rekkefølge enn de vises. Årsaken til dette er at dekoderen krever kunnskap om etterliggende rammer for å kunne dekode B-rammer.

Det er MPEG-2 koderen som bestemmer hvilken sammensetning som utgjør en sekvens. I Figur 7 vises et eksempel på en slik sekvens. For å illustrere hvordan B-rammer fungerer er både videostrøm og visningsrekkefølge vist for en av sekvensene. Pilene viser avhengighetene mellom bildene.



Figur 7 Sekvens av I,P og B-rammer. Tallene viser presentasjonsrekkefølge. Pilene viser avhengigheter.

Skive (Slice)

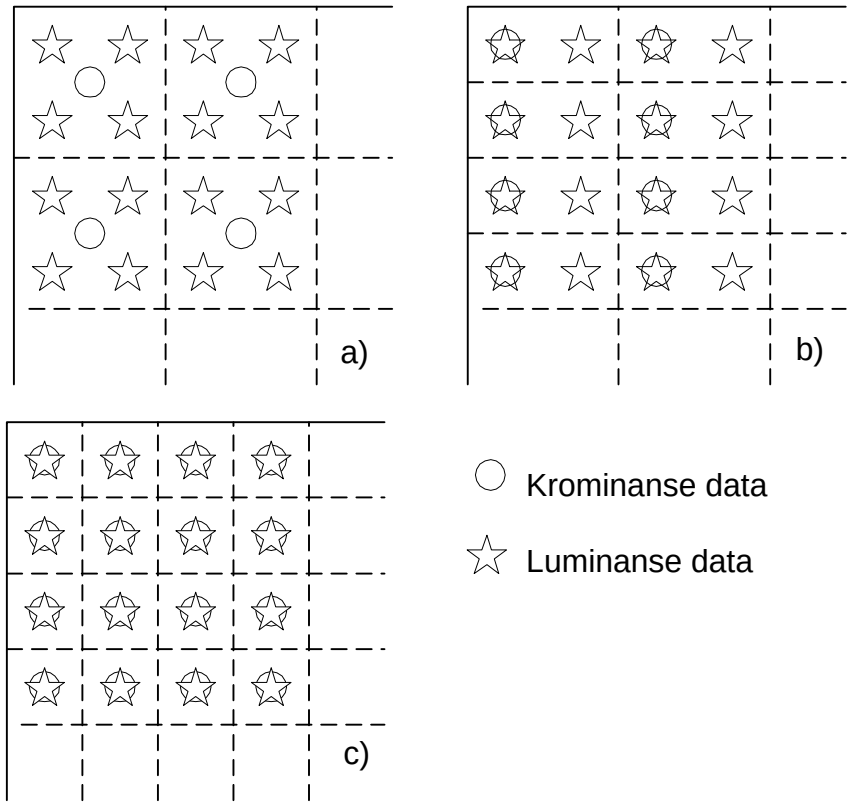
En skive er en fortløpende serie med makroblokker (se under) som ligger i samme horisontale rekke med makroblokker. Skivedata innkapsles med et skivehode. Se Figur 6.

Skiver brukes først og fremst til feilhåndtering. Om en bitstrøm inneholder feil, kan dekoderen gå til starten på neste skive. Mindre skriver gir bedre feilhåndtering, men minsker antall effektive bit som kan brukes til å sende data.

Makroblokk (MB)

Makroblokkene består av luminanse (lys) og krominansekomponenter (farge). Oppbygningen av en makroblokk er avhengig av hvilken subsampling som brukes. I MPEG-2 spesifiseres det tre formater; *4:2:0*, *4:2:2* og *4:4:4*. Oppløsningen på en makroblokk er fast; 16x16 pels.

En makroblokk i 4:2:0 formatet (Figur 8a) vil inneholde fire 8x8 pelmatriser med luminansedata (Y) og en 8x8 matrise for hver av krominansekomponentene (C_r og C_b).



Figur 8 Posisjonering av luminanse og krominanse data. Hentet og modifisert fra Koteng (1996)

4:2:2 formatet (Figur 8b) har C_b og C_r matriser som subsamples med en faktor på 2 i horisontal retning, og 1 (det vil si ingen subsampling i det hele tatt) i vertikal retning.

4:4:4 formatet (Figur 8c) måler C_b og C_r med samme oppløsning som Y. En slik makroblokk inneholder altså fire 8x8 luminanseblokker, og to ganger fire 8x8 krominanseblokker.

Blokk

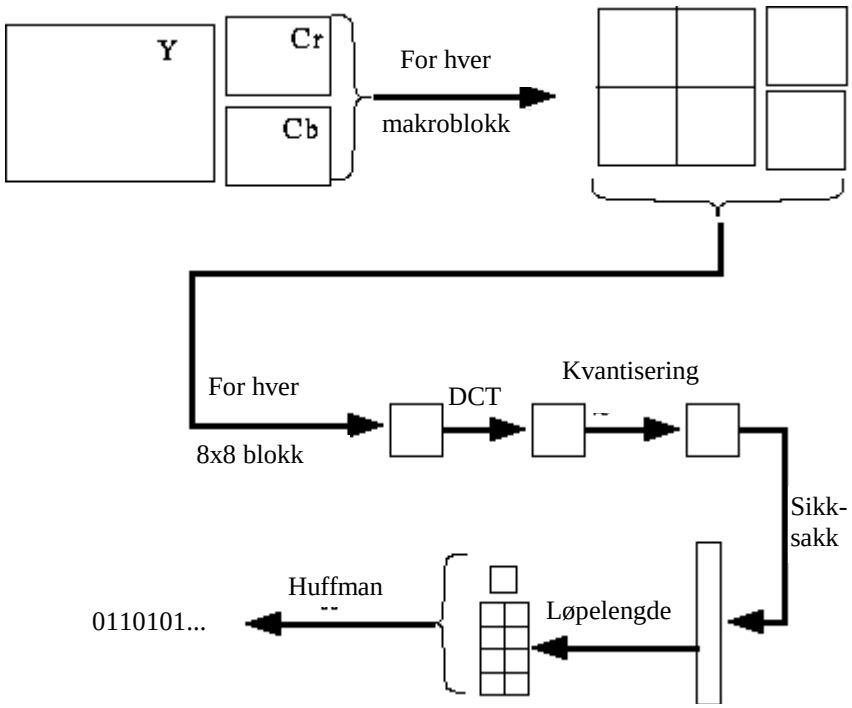
En blokk er den minste enheten i MPEG spesifikasjonen, og brukes til å bygge opp makroblokker. Den består av 8x8 pels. Det finnes tre typer blokker: luminanseblokker, røde krominanseblokker og blå krominanseblokker. Blokken er det grunnleggende elementet i intra-ramme komprimering.

Komprimering

MPEG spesifikasjonen benytter seg av to forskjellige klasser komprimering. Den ser på hvert bilde for seg (intra-ramme komprimering), og på bilder som følger hverandre (inter-ramme komprimering).

Intra-ramme komprimering

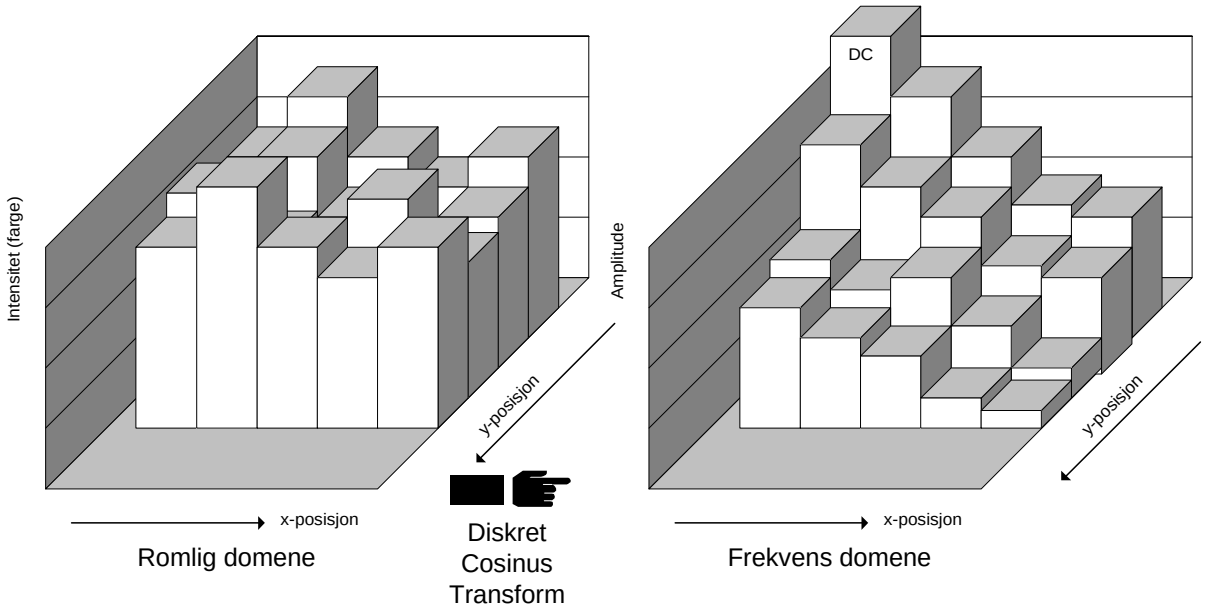
Ved intra-ramme komprimering blir makroblokker transformert til frekvensdomenet ved hjelp av en diskret cosinus transformasjon, kvantisert og ordnet ved hjelp av en sikk-sakk skanning. Til slutt blir det utført en løpelengde og Huffman-koding. Prosedyren illustreres for en makroblokk kodet i 4:2:0 formatet i Figur 9.



Figur 9 Intra kompresjon for bilder i 4:2:0 formatet. Hentet fra V. Lo (1996)

DCT

I de fleste bilder vil det eksistere grupper med pels som har mange av de samme egenskapene. I en MPEG-strøm vil dette gjelde både blokker med bilde- og feilkontroll-data. Ved en transformasjon over til et frekvensdomene vil dette vise seg ved at mesteparten av informasjonen vil bli liggende i lave frekvenskoeffisienter. I tillegg er den menneskelige persepsjon mest følsom for endringer i de lave frekvensdomener.



Figur 10 Diskret cosinus transformasjon

I MPEG utnyttes disse fakta ved at en blokk kjøres gjennom en diskret cosinus transform (DCT). Ut av denne transformen kommer det en ny 8x8 matrise som inneholder en komponent for ingen romfrekvens (DC) og andre elementer som representerer amplituden for de forskjellig frekvenser i blokken (AC verdier). Se Figur 10 for en illustrasjon av transformasjonen.

Ved mellomlinjert video er det i MPEG-2 for ramme-baserte bilder mulig å angi to forskjellige måter å utføre DCT på; en linje-optimalisert, og en ramme-optimalisert.

Kvantisering

DCT-transformasjonen er reversibel, dvs at ingen informasjon går tapt. Det blir heller ikke gjort noen komprimering av datamengden. For å få til dette utfører man en kvantisering. Denne kvantiseringen gjøres ved at man deler hvert element i den transformerte blokka på en tilsvarende verdi i en ferdigdefinert kvantiseringsmatrise. Tabellen under viser standardmatrisen for intra-ramme komprimering. Denne matrisen kan om nødvendig skaleres eller erstattes med en brukerdefinert kvantiseringsmatrise.

Etter kvantiseringen rundes den resulterende verdien av, og om den er større enn 1 beholdes den. Dersom dette ikke er tilfelle settes resultatet til 0.

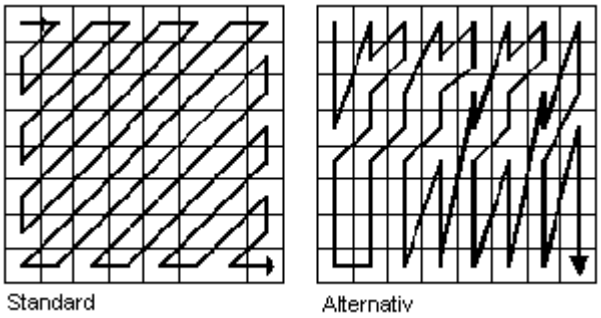
8	16	19	22	26	27	29	34
16	16	22	24	27	29	34	37
19	22	26	27	29	34	34	38
22	22	26	27	29	34	37	40
22	26	27	29	32	35	40	48
26	27	29	32	35	40	48	58
26	27	29	34	38	46	56	69
27	29	35	38	46	56	69	83

Kvantiseringsmatrise for intrakodede rammer. Hentet fra ISO/IEC 13818-2.

Verdiene i kvantiseringsmatrisen er minst for DC koeffisienten og de lave frekvensene, og øker med høyere frekvenser. Slik bevares den viktigste informasjonen, mens de høyere frekvenskoeffisienter blir 0 eller grovt kvantisert. Det må her nevnes at MPEG-2 har mulighet for en finere kvantisering enn MPEG-1. Det kan brukes 11 bit for å beskrive DC (8 i MPEG-1), samt at AC koeffisientene kan ha verdier i området [-2048,2048] ([-256,255] i MPEG-1).

Løpelengde- og Huffman-koding

Etter at kvantisering er gjort, vil man stå igjen med en matrise med mange like verdier, spesielt 0. For å utnytte dette utfører man en sikk-sakk skanning ved lesing av dataene. Dette fører til at like verdier kommer etter hverandre. Spesielt vil de siste dataene være rekker med 0'ere. I MPEG-2 beskrives det to måter å utføre sikk-sakk skanningen på, en for progressiv og en for mellomlinjert video. Figur 11 viser begge metodene. Standard skanning brukes gjerne for progressiv video, mens alternativ brukes for mellomlinjert



Figur 11 Sikk-Sakk skanning.

Etter at koeffisientene er ordnet etter sikk-sakk skanningen, blir de satt opp i serier med såkalte «løpelengde» par (run-amplitude pairs). Hvert av disse parene inneholder et tall som beskriver antall følgende 0 verdier, og et tall som gir amplituden til tallet som følger serien med 0-verdier.

Disse parene blir så kodet med Huffman-algoritmen. Her blir det brukt korte koder for par som opptrer ofte, og lengre koder for mer uvanlige par.

Inter-ramme komprimering

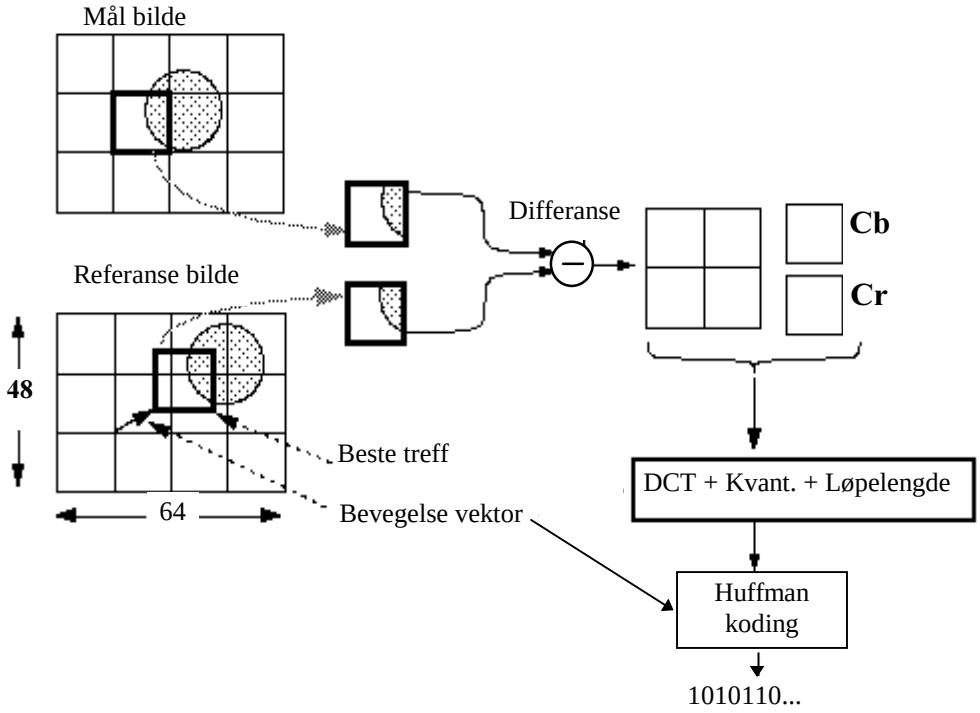
Mye av informasjonen i et bilde i en videosekvens vil finnes igjen i et foranliggende eller etterfølgende bilde. I MPEG-1 og MPEG-2 blir denne redundansen i tid utnyttet ved å representere bilder gjennom deres differanse med nabobilde. Dette kalles predikative bilder, og finnes i variantene P-rammer og B-rammer. Teknikken som brukes for å fjerne redundansen kalles *bevegelsekompensasjon* (motion compensation).

Bevegelsekompensasjon

Bevegelsekompensasjon (BK) brukes for å fjerne temporær redundans i P- og B-rammer. BK opererer på makroblokk-nivå.

En makroblokk som er komprimert ved hjelp av BK vil inneholde informasjon om:

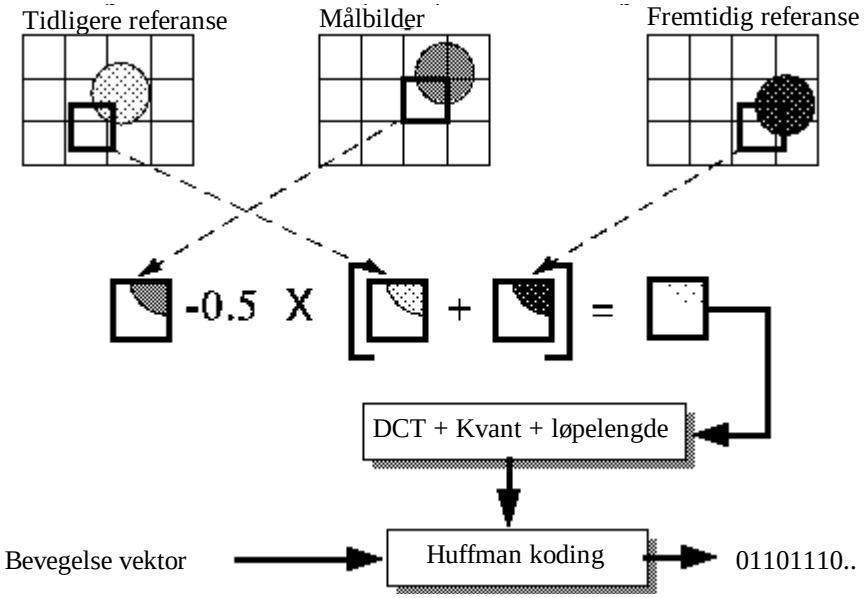
- ✓ En bevegelsevektor som beskriver den romlige forskjellen mellom referanseblokken og den kodede blokken.
- ✓ Forskjellene i innhold mellom referanseblokken og den kodede blokken, dvs den predikative feilen (prediction error).



Figur 12 Beregning av P-rammer. Hentet fra V. Lo (1996)

Figur 12 viser fremgangsmåten for beregning av en P-ramme. Først finnes den blokken i referansebildet som ligner mest på blokken som skal kodes. Deretter beregnes differansen mellom de to blokkene, samt en bevegelsevektor. Dataene komprimeres så ved hjelp av transformasjonskoding. Merk at det her benyttes en annen kvantiseringsmatrise enn ved koding av I-rammer.

I P-rammer vil makroblokker som ikke egner seg for BK komprimeres ved hjelp av de samme teknikken som brukes for I-rammer, dvs. transformasjonskoding. Om referanse-bildet for eksempel inneholder en bil i bevegelse, vil detaljene som var skjult bak denne bilen ikke kunne kodes ved hjelp av bevegelsevektor / prediksjonsfeil.



Figur 13 Beregning av B-rammer. Hentet fra V. Lo (1996)

Dette er ikke et problem for B-rammer, som kan bruke både fremtidige og tidligere bilder som referanse. De to referansene interpoleres for å beregne bevegelsevektoren og den predikative feilen. Det er også mulig å bare bruke fremtidige bilder som referanse (backward prediction). Dette brukes for å kode områder som ikke er vist i tidligere bilder.

Figur 13 viser hvordan prediksjonsfeilen beregnes for en B-ramme som benytter seg av både fremtidig og tidligere referansebilde. Denne komprimeres ved hjelp av transformasjonskoding. Deretter benyttes Huffman algoritmen på disse dataene og bevegelsevektorene.

Skalerbarhet, Profiler og Nivåer

Skalerbarhet

Med skalerbarhet menes det muligheten til å dekode et subsett av en videostrøm for å oppnå en ønskelig oppløsning. MPEG-2 spesifikasjonen støtter dette.

Skalerbarhet implementeres ved at videostrømmen bygges opp av et ordnet sett med bitstrømmer. Disse forskjellige bitstrømmene kalles *lag* (layers). Forskjellige dekodere av varierende kompleksitet vil derfor kunne dekode og vise video av forskjellig kvalitet fra samme videostrøm.

Det enkleste laget som kan dekodes og gi en brukbar videosekvens kalles *basislaget* (base layer). Alle andre lag i videostrømmen kalles *forbedringslag* (enhancement layers). I MPEG-2 er det definert fire verktøy som brukes i forbindelse med skalerbarhet, romlig skalerbarhet, SNR skalerbarhet, temporær skalerbarhet og data partisjonering.

Spatiell skalerbarhet gir mulighet til å generere to videolag med forskjellig romlig oppløsning. Basislaget tilbyr en versjon av videostrømmen med lav oppløsning, mens forbedringslaget inneholder forskjellen mellom den interpolerte versjonen av basislaget og innsignalet til videostrømmen. Forbedringslaget tilbyr derfor maksimal romlig oppløsning.

SNR skalerbarhet tilbyr et basislag som er kodet med en relativt grov kvantisering av DCT koeffisientene. Forbedringslaget inneholder *bare* data som gir en bedre kvantisering av koeffisientene. En fordel med SNR skalerbarhet er at det kan gi en høy feiltoleranse ettersom de viktige dataene i basislaget kan sendes over en sikker kanal, men forbedringslaget kan sendes over en kanal med større feilrate.

Temporær skalerbarhet vil si at basislaget inneholder bilder med en forholdsvis lav *rammerate*[8] (frame rate). Mellomliggende bilder kodes i forbedringslaget ved hjelp av prediksjon.

Datapartisjonering er en teknikk som benytter frekvensdomenet til å splitte en blokk med 64 kvantiserte transformasjons-koeffisienter opp i to bitstrømmer. Den ene, basislaget, vil inneholde viktig informasjon som lav-frekvens koeffisienter, DC verdier og bevegelsevektorer. Forbedringslaget vil inneholde høy-frekvens koeffisientene.

Hybrid skalerbarhet innebærer en kombinasjon av spatiell, SNR eller temporær skalerbarhet.

Profiler og nivåer

MPEG-2 spesifikasjonen er tilpasset en mengde forskjellige bruksområder, bl.a. lagring av digitale media, transmisjon av digital TV, og kommunikasjon. En praktisk implementasjon av standarden vil i de fleste tilfeller rette seg mot et subsett av disse bruksområdene. Dette er tatt hensyn til ved at MPEG-2 er delt opp i profiler og nivåer (levels).

En profil er et veldefinert subsett av bitstrømsyntaksen som er definert i spesifikasjonen. Det settes begrensinger på oppløsningen i fargerommet og skaleringen av bitstrømmen. Tabellen under viser de forskjellige profilene.

Profil	Beskrivelse
Simple	Som Main, men uten B-rammer. Støtter bare Main nivået
Main	Vil brukes i 95% av alle tilfeller. Ingen skalerbarhet. Støtter alle nivåene.
SNR Skalerbar	Som Main, men med SNR skalerbarhet.
Spatiell Skalerbar	Som SNR skalerbar,men med spatiell skalerbarhet.
High	Som spatiell skalerbar, men med 4:2:2 makroblokker.

Beskrivelse av de forskjellige profilene i MPEG-2.

Selv om en profil setter begrensinger på bitstrømmen, vil kodere og dekodere fortsatt kunne håndtere store variasjoner i parametre. For å gjøre slike implementasjoner håndterbare, blir det definert nivåer for hver profil. Et nivå er et sett begrensinger som settes på parametrene i bitstrømmen. Tabellen under, hentet fra Victor Lo (1996) viser de forskjellige nivåene. I kolonnen anvendelse blir det angitt hvilken videokvalitet nivået er tilpasset.

Nivå	Maks pels x linjer x fps	Max bitrate	Anvendelse
Low	352 x 240 x 30	4 Mbit/s	CIF
Main	720 x 480 x 30	15 Mbit/s	CCIR 601
High 1440	1440 x 1152 x 30	60 Mbit/s	HDTV
High	1920 x 1080 x 30	80 Mbit/s	SMPTE 240 std

Beskrivelse av de forskjellige nivåene i MPEG-2.

Audio

Lyden vil ofte være en viktig del av en videosekvens. Selv om lyd ikke tar like mye plass som video er det likevel viktig å komprimere lyden også.

Lyd er som vi vet trykkforskjeller som forplanter seg i et medium (for eksempel luft). Når en mikrofon fanger opp lyden blir den omdannet til elektriske spenningsnivåer. Disse spenningsnivåene blir målt (samplet) et visst antall ganger i sekundet av en datamaskin. *Målefrekvensen* angir hvor ofte det blir målt og blir angitt i enheten Hertz (Hz - antall ganger i sekundet). Hver måling blir representert ved et gitt antall bit, flere bit gir en nøyaktigere måling. For lyd av stereo CD-kvalitet kreves det en målefrekvens på 44.1kHz og 16-bit måling for hver av lydkanalene. For å kunne overføre lyd av denne kvaliteten kreves det 1.4 Mbit/s overføringshastighet. Dette betyr at en film på 100 minutter vil kreve omtrent 1 GByte ukomprimert lyddata. Vi ser at selv om dette bare er omtrent 1 % av ukomprimert videodata for samme film, vil vi også måtte komprimere lyden da den ellers ville utgjøre 10-30% av de komprimerte videodata.

[8] Rammerate angir antall bilder per sekund i en videosekvens.

Kompresjonsgraden for MPEG-audio kan varieres etter hvor strenge krav vi stiller til lydkvaliteten, men det vanlige er omtrent 6-7. Ved denne kompresjonsgraden vil man ha store vanskeligheter med å skille ukomprimert og komprimert lyd.

Informasjon om psykoakustiske modeller og de tre lagene som gies i dette kapittelet er hentet fra *Brandenburg/Stoll (1994)*, samt fra *The compatible multichannel sound system (Philips)*, *Sub-Band Coding (Otolith)*, samt *MPEG Moving Pictures Expert Group Information (Filippini)*.

Funksjonalitet

MPEG-1 Audio støtter mono- og stereolyd i frekvenser brukt for relativt høy kvalitet (44kHz). MPEG-2 Audio er toveis-kompatibel med MPEG-1, men har noen utvidelser når det gjelder funskjonalitet. Den støtter 5 kanaler med høy båndbredde. I tillegg har den en «forbedringskanal» (enhancement channel). Denne kanalen sikrer god kvalitet ved lave bitrater (under 64kbit/s for mono kanal). De 5 kanalene vil sørge for at det blir mulig å bruke MPEG-2 standarden ved bruk av surroundanlegg og lignende. Web-siden *MPEG 2 - the compatible multichannel sound system (Philips)* nevner at det er mulig å få hele 7 + 1 kanaler ved en utvidelse av bitstrømmen.

Psykoakustiske modeller

Ved bilder / videosekvenser kan vi redusere kvaliteten på bildene betraktelig uten at den oppfattede kvaliteten blir redusert i særlig grad. Dette er ikke tilfelle med lyd. Det skal ikke mye lydforstyrrelser til for at mennesker oppfatter lydkvaliteten som redusert. Hvordan får vi da en såpass god kompresjonsrate som 6-7 uten at kvaliteten blir dårligere? Svaret ligger i bruk av *psykoakustiske modeller*.

Psykoakustiske modeller er modeller som sier noe om hvordan den menneskelige hjerne oppfatter lyd. De sier noe om hvilke deler av lyden som legges best merke til og hvilke deler som «drukner» i annen lyd. Ved å bruke disse modellene ved kodingen av lyd kan man redusere kvaliteten på de enkelte deler av lyden som man allikevel ikke ville ha lagt merke til.

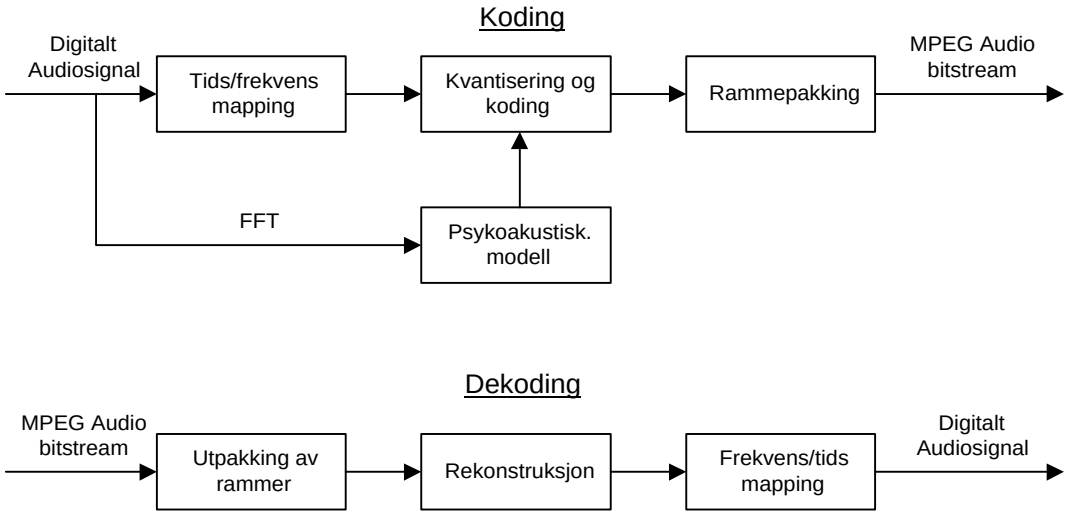
Dersom vi har to toner samtidig der den ene er 20 dB lavere enn den andre vil et menneske ikke kunne høre den laveste. Dette fenomenet kalles *maskering.* I en psykoakustisk modell vil dette prinsippet i sterk grad bli brukt. Dersom vi hadde hatt støy på den laveste tonen ville vi heller ikke ha hørt støyen. Dette kan vi overføre til normal lyd i en film; enkelte lyder vil til en viss grad maskeres. Avhengig av hvor sterkt lyden blir maskert kan vi redusere antall bit som brukes for å representere en måling. For en ukomprimert 16-bit lyd kan man for eksempel redusere til 4 bit for en sterkt maskert lyd. Dette vil medføre at færre bit trengs for å lagre lyden og vi får dermed en komprimeringseffekt.

I praksis blir dette gjort ved at spekteret fra 20 Hz til 20 kHz blir delt opp i flere sub-bånd. For hvert sub-bånd blir det kalkulert en maskeringseffekt for hver lyd og vi kan finne ut hvor mange bit vi trenger for å representere denne målingen. Maskeringeffekten blir beregnet ved hjelp av spektruminformasjon som modellen har fått gjennom en Fast Fourier Transform (FFT) som allerede er utført på det opprinnelige signalet. Vi velger antall bit slik at den støyen som blir innført ved bitreduksjon ligger under en grenseverdi for maskering. Denne maskingeffekten vil kunne ha en effekt også på nærliggende sub-bånd, men vil avta etterhvert som vi fjerner oss fra båndet der vi fant lyden. I en normal situasjon vil vi ha mange lyder spredt utover hele frekvensspekteret. Dette betyr at maskingeffekten blir addert sammen for alle sub-bånd. Reduseringen av antall bit kalles også kvantisering. Informasjon om kvantiseringen vil bli sendt over til dekoderen slik at den kan dekode signalet.

I tillegg til maskeringseffekten for samtidige lyder har vi også en maskeringseffekt foran og etter en lyd (Pre-masking og Post-masking). I et kort tidsrom før en sterk lyd (opptil 5 ms) og et litt lengre tidsrom etter en sterk lyd (100 ms) vil ikke hjernen oppfatte andre lyder så godt.

Et annet aspekt som en psykoakustisk modell kan ta for seg er prinsippet om at det menneskelige øret har lavere sensivitet i svært høye og svært lave frekvenser. Maksimal sensivitet finnes i området 2-4 kHz. Dette kan utnyttes ved å i større grad maskere sub-båndene for lave og høye frekvenser.

Koding og dekoding



Figur 14 Oversikt over koding og dekodingsprosessen for MPEG-Audio. Hentet fra Brandenburg/Stoll (1994).

Det første som skjer er at det i parallell blir foretatt to ting med det digitale lydsignalet, en Fast Fourier Transform (FFT) som den psykoakustiske modellen mottar resultatet av, og en tids/frekvens mapping. Tids/frekvens mappingen deler signalet opp i flere sub-bånd i frekvensplanet (og i tidsplanet).

Det neste som skjer er at den psykoakustiske modellen benytter informasjon fra frekvensspekteret (fra FFT) til å kvantisere og kode de ulike subbåndene. De kodede signalene blir deretter pakket inn i rammer og sendt som en bitstrøm.

Ved utpakking skjer den motsatte prosessen, men denne er langt enklere da den bl.a. ikke trenger noen psykoakustisk modell. Bitstrømmen blir pakket ut og man rekonstruerer de ulike sub-båndkomponentene. Tilslutt blir en reversert frekvens/tids mapping utført slik at vi får et nytt lydsignal. Dette lydsignalet er ikke det samme som det vi begynte med, dette er en ikke-tapsfri prosess fordi vi bl.a. har brukt kvantisering. Det vil imidlertid likevel kunne være av meget god perseptuell kvalitet.

Lagdeling

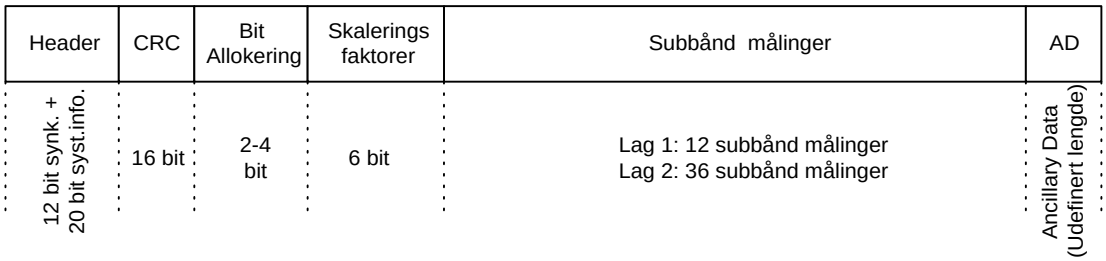
Det er definert tre forskjellige lag (layers) i MPEG-Audio (både for MPEG-1 og MPEG-2). Disse lagene vil hver tilby noe forskjellige metoder for å gjøre hovedoperasjonene i en kodingsprosess. Det er opp til programmereren av applikasjonen å velge et lag som passer applikasjonen. De ulike lagene har forskjellig grad av kompleksitet, lag 1 er enklest mens lag 3 er mest komplisert. Lag 3 krever mest av koderen, men tilbyr en mer effektiv utnyttelse av den psykoakustiske modellen. Dette laget gir best komprimering og blir derfor brukt når vi må klare oss med en lav bitrate. Ved bruk av lag 1 vil koderen bli vesentlig mindre komplisert og dette laget er derfor populært i situasjoner der det er fordelaktig med rask og enkel koding. Det gjelder å finne ut hva som er viktig i den aktuelle applikasjonen når man skal bestemme seg for hvilket lag man vil bruke; kompleksiten til koderen eller kompresjonsgraden.

	Tid/Frekvens mapping	Psykoakustisk modell	Kvantisering og Koding
Lag 1	Multifase filtrering i 32 like store sub-bånd.	Bruker 512 punkts FFT for å beregne maskeringseffekt.	Kvantisering. 6 bit skaleringsfaktor.
Lag 2	Multifase filtrering i 32 like store sub-bånd.	Bruker 1024 punkts FFT for å beregne maskeringseffekt.	Mer presis kvantisering. 3 ganger så lang ramme -> bitraten for skaleringsfaktorene blir bedre komprimert.
Lag 3	Finere oppdeling i sub-bånd samt DCT filtrering (Discrete Cosine Transform).	1024 punkts FFT samt en bedre modell (polynomial prediction model).	Ikke-uniform kvantisering. Adaptiv oppdeling. Mengdekoding (Entropy coding).

Tabellen over viser noen av de viktigste elementene som blir brukt i de ulike lagene for ulike faser i komprimeringen. Informasjonen i denne er hentet fra *Brandenburg / Stoll (1994)*.

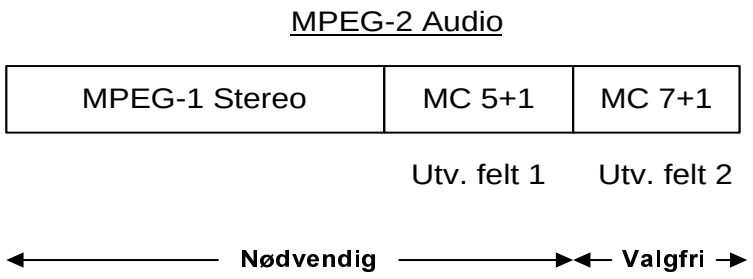
Bitstrømmen

MPEG-2 Audio vil som basis bruke den samme bitstrømmen som MPEG-1 Audio bruker. Et generelt format for denne bitstrømmen er vist i Figur 15. Dette formatet vil variere noe med hensyn til bitlengde på forskjellige felt. Dette er blant annet avhengig av hvilket lag man bruker, målingsrate, og hvilke subbåndsfrekvenser som blir kodet.



Figur 15 Eksempel på bitformat for MPEG-Audio. Hentet fra Brandenburg/Stoll (1994).

Som tidligere nevnt vil MPEG-2 kunne brukes til å representere hele 5 kanaler samt 1 forbedringskanal (5 + 1). Det er også mulig for MPEG-2 å representere 7 + 1 kanaler. Bitstrømmen for MPEG-2 vil derfor i tillegg til en stereo MPEG-1 Audio bitstrøm inneholde ett eller to utvidningsfelt som inneholder informasjon om de ekstra kanalene. Se Figur 16 for en oversikt over bitstrømmen for MPEG-2 Audio. Denne bitstrømmen er kompatibel på en slik måte at den uten problemer kan spilles av på utstyr som ikke støtter de ekstra kanalene.



Figur 16 MPEG-2 Audio bitstrøm. Hentet fra MPEG2 - the compatible multichannel sound system (Philips).

Systemlaget

Systemlaget definerer den delen av MPEG-2 som har ansvaret for å omforme de nedpakkede lyd og bildedata til en slik form at de kan lagres på et medium / sendes over et nettverk.

Systemlaget innbefatter pakking av de ulike elementærstrømmene (PES-pakker), samt multipleksing av disse pakkene til en enkel strøm som egner seg for lagring / oversendelse over nettverk. MPEG-2 definerer to ulike måter å utføre multipleksingen på. Disse gir to ulike strømmer, programstrømmen og transportstrømmen. Programstrømmen egner seg godt for lagring på feilfritt medium, mens transportstrømmen egner seg for oversendelse over nettverk.

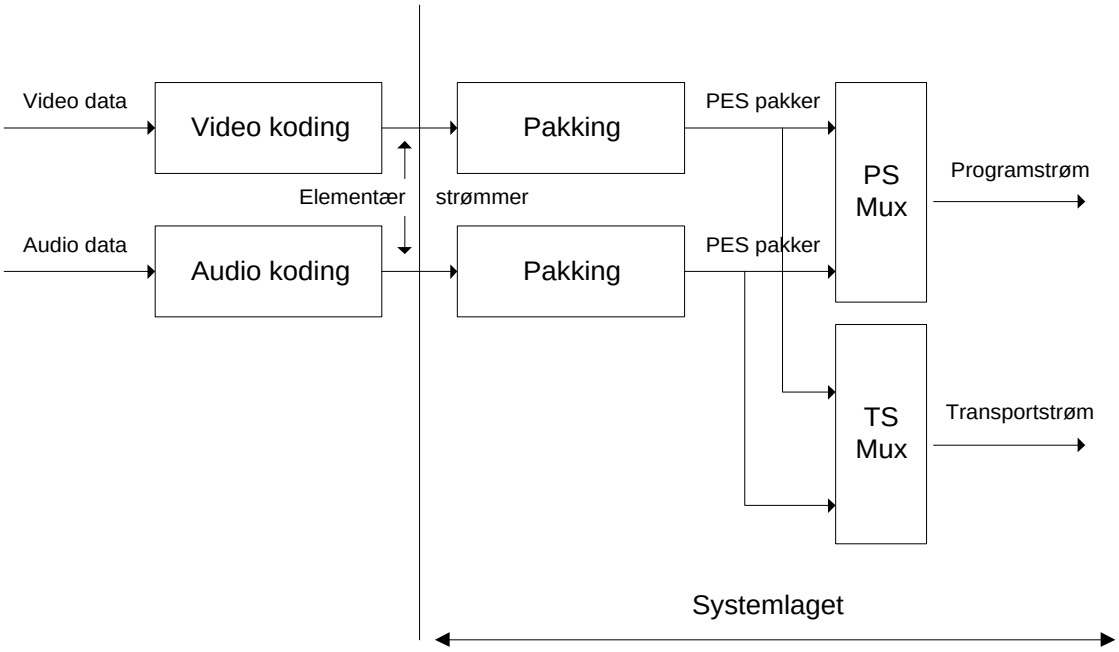
Vi vil i dette kapittelet gi en oversikt over hele denne prosessen. Det vil bli foretatt en kort sammenligning av programstrømmen og transportstrømmen, og vi vil deretter gå litt mer i dybden på transportstrømmen. Vi vil legge vekt på å beskrive hvordan tidsinformasjon og synkronisering blir behandlet/foretatt.

Informasjon om systemlaget er hentet fra ISO/IEC 13818-1 (Systems) samt C.Tryfonas (1996).

Oversikt

Vi vil starte med å definere et *program*. Et program er en videosekvens som kan inneholde flere bestanddeler. Hver av disse er kodet ut i fra samme referanseklokke. Hovedbestanddelene er video og lyd, men et program kan også bestå av annen informasjon, som f.eks data/tekst. Disse delene vil til sammen skape et inntrykk for seeren.

Hver av disse delene kan betraktes som en rekke data som sendes til en mottaker, f.eks over et nettverk. Disse rekkene av data kalles strømmer. Ved hjelp av MPEG-2 kompresjon blir disse strømmene omformet til det som kalles *elementærstrømmer* (Elementary streams), se Figur 17. Det er ikke nødvendig å bruke MPEG-2 kompresjon for å lage en elementærstrøm, for eksempel kan små datamengder direkte danne en elementærstrøm dersom man ønsker det. Systemlaget til MPEG-2 definerer hvordan disse ulike elementær-strømmene blir pakket, satt sammen (multiplekset), og sendt f.eks. over et nettverk.



Figur 17 Oversikt over systemlaget. Hentet fra ISO/IEC 13818-1 (Systems)

Programstrøm og Transportstrøm

MPEG-2 standarden definerer to forskjellige hovedmetoder for multipleksingen. Den ene danner en *programstrøm*, den andre en *transportstrøm*. Som vist i Figur 17 må man velge en metode basert på hva slags dataoverføringsmiljø det er snakk om.

Programstrømmen tilsvarer systemlaget til MPEG-1, en strøm tilsvarer ett og bare ett program. Strømmene består av lange pakker, noe som betyr at det kan få store konsekvenser dersom en pakke blir borte. Pakkene kan også ha variabel pakkelengde, noe som gjør at det blir vanskelig å synkronisere strømmene dersom en pakke forsvinner. Alt dette gjør at bruk av programstrøm egner seg best i tilnærmet feilfrie miljøer, som f.eks lagring på CD-plater / DVD-plater.

Transportstrømmen kan kombinere flere programmer, og sette disse sammen til en enkel strøm. Pakkene i denne strømmen har en fast lengde og det er derfor relativt lett å gjenoppta synkroniseringen dersom en pakke skulle forsvinne. Transportstrømmen egner seg derfor best når det er snakk om overføring over nettverk. Den er imidlertid mer kompleks enn programstrømmen, den er vanskeligere å kode og dekode og bruker dermed mer ressurser.

Det er mulig å konvertere en transportstrøm til en eller flere programstrømmer. Dette kan være nyttig dersom man for eksempel vil lagre programmer på et feilfritt medium. På samme måte kan man konvertere fra en programstrøm til en transportstrøm dersom man vil overføre programmet over et ikke-feilfritt transportmedium. Vi skal ikke gå nærmere inn på disse aspektene her.

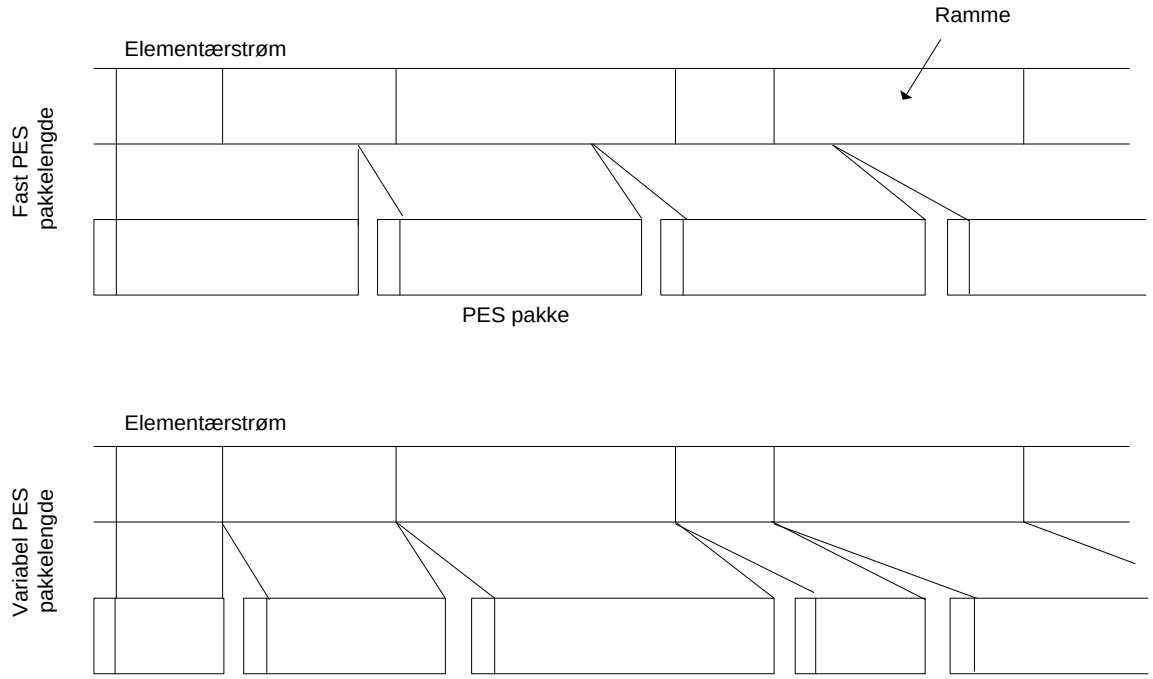
Videotjenere vil som regel benytte nettverk til oversendelse av videoen. Det vil derfor i det etterfølgende bli fokusert på transportstrømmen.

PES-pakkene

Vi har slått fast at etter komprimeringen har vi elementærstrømmer for hver datakilde som inngår i programmet, hovedsakelig video og lyd. Ved konstruering av både en transportstrøm og en programstrøm er det neste som skjer at strømmen blir delt inn i pakker. Disse pakkene kalles *PES-pakker* (Packetized elementary stream). En PES-pakke består av en header og nyttelasten (innholdet). Headeren kan bl.a bestå av tidsmerker (timestamps)

som brukes for synkronisering under dekodingen. Nyttelasten er ganske enkelt en del av elementærstrømmen som er kopiert inn i PES-pakken.

Det er ikke noe krav om at en enhet i elementærstrømmen (f.eks I, B eller P-rammer) i sin helhet også skal tilsvare innholdet i en PES-pakke, men det er fullt mulig å gjøre det slik. Dette leder oss til konseptene faste og variable PES-pakkelengder. Ved bruk av en fast pakkelengde vil enhetene fra elementærstrømmen vilkårlig bli fordelt utover PES-pakkene (husk at disse enhetene har en svært variabel lengde). Om det derimot brukes variable pakkelengder kan vi sørge for at hver enhet blir plassert i en enkelt PES-pakke. Fordelen med å bruke variabel pakkelengde er at det blir lettere for dekoderen å finne begynnelsen og slutten på f.eks en ramme. Ulempen er at det blir en mer komplisert prosess å pakke PES-pakkene, ved bruk av faste pakkelengder vil dette kunne gjøres lettere og raskere.



Figur 18 Konstruksjon av PES-pakker, fast og variabel lengde. Hentet fra C. Tryfonas (1996).

Beskrivelse av PES-pakkenes felt

En PES-pakke består av mange felt, og noen av disse feltene vil kunne ha forskjellig mening avhengig av dataene som PES-pakken består av. Vi skal her nevne noen av de *viktigste* feltene som vil brukes. For en utførlig beskrivelse av alle feltene i en PES-pakke, se ISO/IEC 13818-1 Systems.

packet_start_code_prefix: Dette er et felt på 24 bit som angir starten på en PES-pakke. Bytene har følgende verdi: ‘00000000 00000000 00000001’. Dette gjør det enkelt å identifisere en PES-pakke.

stream_id: Dette feltet består av 8 bit og angir hvilken type og hvor mange elementærstrømmer som denne PES-pakken inneholder.

PES_packet_length: Dette feltet på 16 bit angir antall byte som følger etter dette feltet i PES-pakken.

PTS_DTS_flags: Dette er to bit som angir om det finnes en PTS-verdi og/eller en DTS-verdi i denne PES-pakken. Verdien ‘00’ angir ingen verdier, ‘10’ angir PTS-verdi, men ikke DTS-verdi, og ‘11’ angir at begge finnes. Verdien ‘01’ er ikke tillatt.

PTS(presentation_time_stamp): Dette er et 33 bit felt som angir et tidspunkt for *presentasjon* av rammen (bildet) eller lyden som denne PES-pakken består av. Verdien blir brukt av dekoderen ved presentasjonen. Se avsnittet «Synkronisering», side 43, for en nærmere beskrivelse av bruken. PTS-verdien blir i ISO/IEC 13818-1 beregnet ved hjelp av følgende formel:

$$PTS(k) = ((system\_clock\_frequency \times tp_n(k)) DIV 300) \%2^{33}$$

hvor

system_clock_frequency er 27Mhz for MPEG -2

$tp_n(k)$ er presentasjonstiden for presentajonsenheten $P_n(k)$. En presentasjonsenhet er et dekodet bilde eller audio ramme.

$P_n(k)$ er presjentajonsenheten som tilsvarer den første aksessenheten som følger i eller etter pakken som PTS-verdien er en del av. En aksessenhet er kodet representasjon av en presentasjonsenhet - enten et bilde eller en audio enhet.

DTS(decoding_time_stamp): Dette er et 33 bit felt som angir et tidspunkt for *dekoding* av rammen (bildet) eller lyden som denne PES-pakken består av. Verdien bilr brukt av dekoderen ved dekodingen. Se avsnittet Synkronisering», side 43, for en nærmere beskrivelse av bruken. DTS-verdien blir i ISO/IEC 13818-1 beregnet ved hjelp av følgende formel:

$$DTS(j) = ((system\_clock\_frequency \times td_n(j)) DIV 300) \%2^{33}$$

hvor

system_clock_frequency er 27Mhz for MPEG -2

$td_n(j)$ er tidspunktet for dekoding av aksessenheten $A_n(j)$

$A_n(j)$ er den første aksessenheten som følger i eller etter pakken som PTS-verdien er en del av.

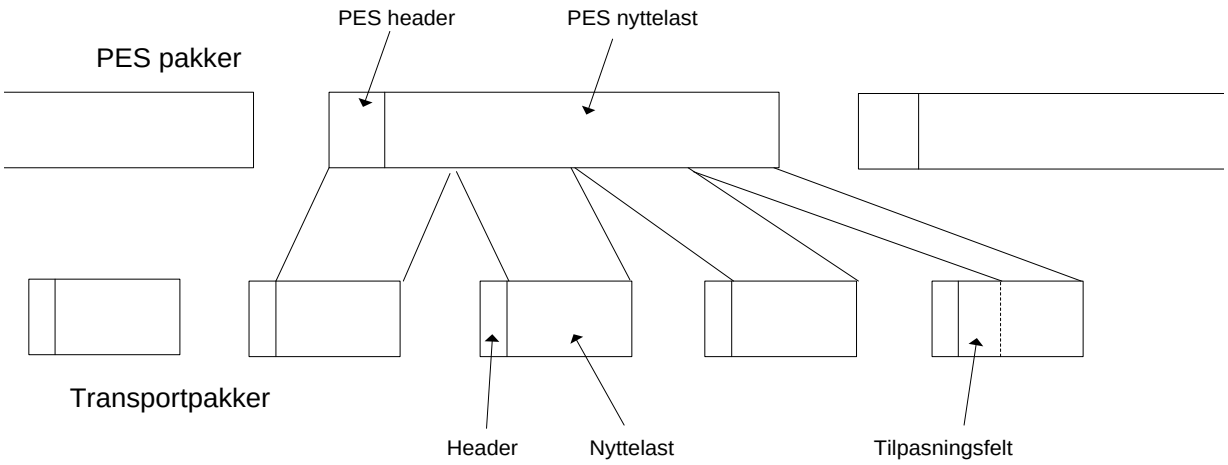
trick_mode_control: Dette er 3 bit som kan brukes til å angi forskjellige typer avspillingsmodus. Verdiene kan angi «fast forward», «slow motion», «freeze frame», «fast reverse», og «slow reverse». Merk at ved bruk av noen av disse modusene vil enkelte felt i elementærstrømmen være ugyldig.

Generering av transportstrømmen

Etter at vi har generert PES-pakker for hver datakilde må vi nå sette sammen (multiplekse) disse til en transportstrøm, altså en strøm av såkalte transportpakker. Ikke alle transportpakker blir brukt til lagring av PES-pakkene, noen transportpakker må brukes til informasjon om programmet/ene som blir sendt. Disse *kalles PSI transportpakker* (Program Spesific Information). Disse pakkene beskrives senere.

Den viktigste egenskapen til transportpakkene er at de har en fast lengde på 188 byte. En pakke består av en header på 4 byte og et felt for selve nyttelasten. I tillegg kan pakken bestå av et *tilpasningsfelt* (Adaptation field), dersom det skulle være behov for det. Dette behandles nærmere senere.

PES-pakkene blir fordelt utover transportpakkene. Innholdet i PES-pakkene (både headeren og nyttelasten) blir kopiert over til transportpakkene. Lengden på en PES-pakke vil normalt være mye større enn lengden på en transportpakke, slik at hver PES-pakke blir fordelt utover flere transportpakker. Det er imidlertid en regel man må følge ved denne fordelingen: den første byten i en PES-pakke skal også være den første byten i en transportpakke. Dette betyr at når den siste biten fra en PES-pakke er plassert i en transportpakke, kan man ikke begynne å plassere neste PES-pakke i denne transportpakken. Dette får igjen den konsekvens at enkelte transportpakker ikke blir fylt helt opp. I disse tilfellene vil det overflødige tomrommet bli fylt av det tidligere nevnte tilpasningsfeltet.



Figur 19 Konstruksjon av transportpakker. Hentet fra C. Tryfonas (1996).

Beskrivelse av transportpakkens header

Headeren til en transportpakke består av 4 byte, altså 32 bit. Vi skal her kort nevne funksjonaliteten til de ulike feltene vi finner i headeren. Se ISO/IEC 13818-1 Systems for en utførlig beskrivelse av disse feltene.

sync-byte: Dette er en byte som blir brukt for synkronsiering og har alltid verdien 47 (hex). Det er den første byten i headeren og dermed også den første i en transportpakke, noe som gjør den egnet til å lett identifisere begynnelsen på hver pakke.

transport_error_indicator: Dette er en bit som angir om det finnes minst en uopprettelig bitfeil i pakken.

payload_unit_start_indicator: Dette er en bit som angir om denne pakken inneholder starten på en PES-pakke (eller PSI-pakke).

transport_priority: Dette er en bit som kan brukes til å gi denne pakken høyere prioritet enn andre pakker innenfor samme elementærstrøm.

PID: Dette er identifikatoren som brukes for å vise hvilken elementærstrøm denne pakken tilhører. Identifikatoren består av 13 bit.

transport_scrambling_control: Dette er to bit som angir om scrambling blir brukt, eventuelt hvilken modus som blir brukt.

adaptation_field_control: Dette er to bit som angir om pakken innholder et tilpasningsfelt og/eller nyttelast.

continuity_counter: Dette er en teller på 4 bit som inkrementeres for hver pakke innen samme elementærstrøm. Dette muliggjør oppdagelse av forsvunnede pakker.

Innhold i tilpasningsfeltet (Adaptation field)

Enkelte transportpakker vil som nevnt ikke fylles helt av en PES-pakke, og resten av denne pakken vil dermed utgjøre tilpasningsfeltet. Dette feltet kan inneholde mange felter, men vi skal bare kort nevne den viktigste funksjonaliteten.

Den første byten kalles **adaptation_field_length** og angir hvor mange etterfølgende byte dette tilpasningfeltet består av. Denne verdien kan dermed være mellom 0 og 183. Det er 188 byte i en pakke og 5 byte er allerede brukt til headeren og denne byten.

Deretter kan det finnes flere 'flagg' som angir forskjellige ting, vi nevner bare PCR-flagget som angir om det finnes et *PCR-felt* (Program Clock Reference) i pakken. PCR-feltet er et felt som inneholder tidsinformasjon som brukes ved dekoding og synkronsisering. Verdien angir når dataene skal komme fram til dekoderen.
PCR-feltet vil kodes i to deler : en basedel og en tilleggsdel som øker presisjonen.

PCR kodes slik:

$$PCR\_base(i) = ((system\_clock\_frequency \times t(i))\, DIV\, 300)\%2^{33}$$

$$PCR\_ext(i) = ((system\_clock\_frequency \times t(i))\, DIV\, 1)\%300$$

$$PCR(i) = PCR\_base(i) \times 300 + PCR\_ext(i)$$

System_clock_frequency er 27Mhz for MPEG -2.

Se avsnittet «Synkronisering», for en nærmere beskrivelse av bruken.

Dersom tilpasningsfeltet ikke blir helt fullt, vil det bli fylt av «stuffing bytes». Dette er byte som inneholder '1111 1111' og som dekoderen vil ignorere.

PSI-transportpakker

Dette er pakker som skiller seg ut fra «vanlige» pakker. Dette er pakker som inneholder informasjon om transportstrømmen. Hver elementærstrøm blir tildelt en identifikator, PID, som er unik innenfor den tilhørende transportstrømmen. PSI-pakkene inneholder informasjon som lar dekoderen finne ut hvor mange programmer som finnes i transportstrømmen og hvor mange elementærstrømmer som finnes i hvert program. Dette blir gjort ved tabeller som beskriver «mappingen» mellom programmer og elementærstrømmene.

PSI-pakkene kan også inneholde en tabell som det er valgfritt å legge ved. Denne tabellen inneholder informasjon om nettverket; frekvenser o.l.

Tilslutt kan vi nevne at dersom det blir brukt «scrambling» av en strøm så kan det sendes en tabell som inneholder informasjon om dette.

Synkronisering

Vi vet at under MPEG-2 prosessen blir flere strømmer satt sammen til en strøm og senere pakket ut igjen til flere strømmer som så skal spilles av. Det er klart at dersom disse strømmene skal spilles av ved riktig tidspunkt må vi ha en synkroniseringsmekanisme. Dette blir gjennomført ved hjelp av "tidsmerker" (timestamps). Ved kodingen blir hver strøm merket på en slik måte at man ved dekodingen kan finne ut av når den enkelte strøm skal avspilles.

Det er hovedsakelig tre typer tidsmerker som blir brukt ved MPEG-2 koding: PCR (Program Clock Reference), PTS (Presentation Timestamp) og DTS (Decoding Timestamp). PTS definerer tidspunktet for presentasjon av en ramme, dvs at rammen innen dette tidspunktet skal være fjernet fra bufferet, dekodet og presentert. DTS definerer tidspunktet for dekodingen, innen dette tidspunktet skal rammen være fjernet fra bufferet og dekodet. Disse to stemplene blir konstruert for hver elementærstrøm. Siden hvert program kan inneholde mange elementærstrømmer er det viktig med synkronisering også på programnivå. Til dette brukes PCR, dette tidsstemplet blir konstruert for hvert program. Dette feltet finnes i tilpasningsfeltet, og definerer når den siste byten i feltet skal komme fram til dekoderen. Vi ser at det naturlige vil være at PCR-verdien er minst, DTS-verdien noe større, og PTS-verdien størst.

Dekoderen vil nå bruke PCR som basis for bruk av PTS og DTS i elementærstrømmer i samme program. PCR blir brukt til å synkronisere klokka hos dekoderen slik at denne klokka kan brukes til å sammenligne mot PTS og DTS. På denne måten blir de innbyrdes tidsforskjellene mellom ulike rammer i elementærstrømmene ivaretatt og dermed synkronisert.

Disse tidsstemplene blir definert i forhold til en systemtid (System Time Clock) som MPEG-2 standarden definerer. Denne systemtiden blir målt i enheter på 27 MHz. Ved kodingen blir tidsmerkene gitt verdier basert på

nåværende verdi av systemtiden. Dette blir gjort ved å beregne en tidsforskjell (offset) som deretter blir lagt til systemtiden for de respektive tidsmerkene. Denne tidsforskjellen vil dermed representere en maksimalverdi for tiden det kan ta å overføre denne rammen, dekode den og (for PTS) presentere den.

DTS og PTS blir plassert i headeren i PES-pakkene. En PTS kan forekomme alene, mens en PES-pakke med en DTS alltid også må inneholde en PTS. Det er ikke nødvendig å overføre tidsmerker for alle rammene, dekoderen klarer fint å synkronisere rammer innenfor et kort tidsrom der transportraten er kjent. Det er satt som et krav at et PTS-felt må forekomme minst hvert 0.7 sekund. Et PCR-felt må forekomme minst hvert 0.1 sekund.

I tillegg til disse tre viktigste stemplene kan det også brukes et par andre varianter som er valgfrie. Disse er ESCR (Elementary Stream Clock Reference) og OPCR (Original Program Stream Reference). ESCR fungerer på samme måte som PCR i den forstand at den sier når pakken skal komme fram til dekoderen. Forskjellen er at den konstrueres for en elementærstrøm, ikke et program, og at den ligger i PES-pakkene. OPCR kan brukes til å ta vare på den originale PCR-verdien under rekonstruksjon av en transportstrøm med ett program fra en annen strøm.

DSM-CC

For å kunne tilby bredbåndsapplikasjoner hjem til forbrukeren, er det viktig å ha åpne protokoller for slike applikasjonstjenester. Om ikke dette eksisterte ville det være nødvendig å definere et proprietært grensesnitt for hver enkel tjeneste. Med en åpen protokoll vil det være mulig for forbrukeren å bruke utstyr av forskjellige art, det være seg personlige datamaskiner eller annet, for å kople seg opp mot forskjellige tjeneste-leverandører.

Digital Storage Media – Command and Control, *DSM-CC*, er et slikt sett med protokoller. Det tilbyr i utgangspunktet kontrollfunksjoner for MPEG-1 og MPEG-2 bitstrømmer. Protokollene er likevel egnet til mer generell bruk.

DAVIC[9] er en internasjonal komité bestående av over 200 forskjellige organisasjoner fra over 20 land. Den arbeider med å definere åpne standarder for levering av digitale audio-visuelle tjenester over forskjellige nettverk. I komiteens DAVIC 1.0 spesifikasjon blir DSM-CC benyttet for å kontrollere interaktive multimedia-sesjoner, samt ressurser benyttet av disse. DAVIC arbeider også med å bruke DSM-CC i tilknytning til Internett aksess.

DSM-CC er uavhengig av transportlaget[10]. Dette innebærer at applikasjoner basert på DSM-CC vil kunne brukes over nettverk basert på varierende teknologier.

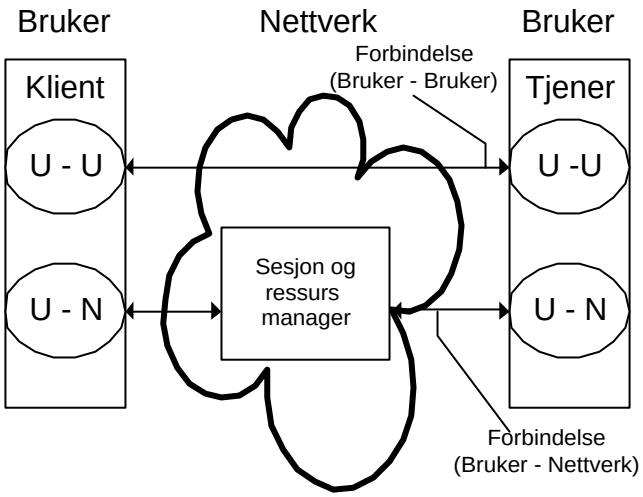
De viktigste protokollsettene i DSM-CC spesifiserer konfigurasjon og nedlasting av data til klient, kontroll av videostrømmer og generelle applikasjons- og kringkastingstjenester. Disse områdene blir beskrevet under, men først gis det en oversikt over den funksjonelle referansemodellen til DSM-CC og tilhørende begreper.

Hvor ikke annet nevnes er det V. Balabanian m.fl(1996) og ISO/IEC 13818-6 som er brukt som referansestoff.

En funksjonell referansemodell for DSM-CC

DSM-CC tar utgangspunkt i en enkel funksjonell modell. Denne beskriver entiteter som klienter og tjenere som kommuniserer over og gjennom et nettverk.

[9] The Digital Audio-Visual Council.
[10] Ref. ISO-modellen



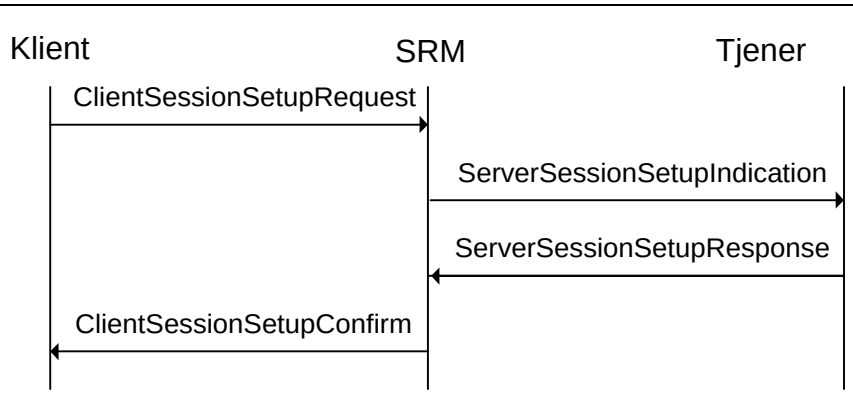
Figur 20 Funksjonell referansemodell for DSM-CC. Hentet fra ISO/IEC 13818-6

Klientene defineres i denne sammenheng som utstyr beregnet for å gjengi digitale media. *Tjenerne* er entiteter utstyrt for å levere og lagre multimedia. *Nettverket* er en samling med kommuniserende elementer som tilbyr en forbindelse mellom klient og tjener. Merk at definisjonene av klient, tjener og nettverk er gjort meget generelle. En tjener kan for eksempel være et kluster av arbeidsstasjoner med spesialiserte arbeidsoppgaver.

Figur 20, hentet fra ISO/IEC 13818-6, viser referansemodellen. Klient og tjener samles under begrepet bruker. I nettverket er det definert en «Sesjons and Ressurs Manager»-entitet, *SRM*. Oppgaven til denne entiteten er blant annet å sette opp forbindelser mellom klient og tjener, tildele ressurser, autentisere klienter og tilby informasjon om konfigurasjon til brukere.

SRM terminerer såkalte *U-N forbindelser* (User-Network). U-N forbindelser brukes for å utveksle *U-N meldinger*. Hensikten med disse meldingene er å kontrollere sesjoner og nettverksressurser. *U-U forbindelser* (User-User) eksisterer mellom klient og tjener. Det kan eksistere flere slike forbindelser på samme tidspunkt mellom samme klient og tjener. Informasjonsflyten i en U-U forbindelse spesifiseres ikke eksplisitt i DSM-CC, men det oppgis et sett med generelle tjenester som en tjener kan tilby til en klient.

En *sesjon* er definert som en assosiasjon mellom to brukere som gir adgang til de ressurser som er nødvendig for å opprette en instans av en tjeneste. For eksempel vil en klient som ønsker å bestille en video fra en video-on-demand (VOD) leverandør måtte sette opp en sesjon med en tjener. Sesjonen avsluttes når filmen er slutt, og klienten ikke har behov for flere tjenester. Figur 21 viser hvilke U-N meldinger som utveksles ved et vellykket oppsett av en sesjon.



Figur 21 U-N meldinger. Her vises meldingene som utveksles under oppsett av en sesjon

U-U forbindelser vil være de deler av en sesjon som opptar mest ressurser. Det er her informasjonsflyten går. Som nevnt vil en sesjon oftest bestå av mer enn en U-U forbindelse. I V.Balabanian m.fl (1996) trekkes det frem et eksempel hvor en forbindelse brukes til å vise video i ett vindu, mens en annen forbindelse brukes til å vise frem bakgrunnen til vinduet.

En sesjon vil alltid bli initiert av en klient. Det er mulig for en tjener å be en klient om å sette i gang en slik initiering ved hjelp av «PassThru» meldinger.

Konfigurasjon av tjener og nedlasting til denne.

I DSM-CC vil en klient vanligvis bruke ett sett med U-N meldinger til å konfigurere seg selv. En slik U-N konfigurasjon kan bli initiert av bruker, av nettverket eller som en konsekvens av at en bruker lytter på en kjent forbindelse.

Nedlastingsprotokollen i DSM-CC er ment å være lettvekts og gi rask nedlasting av data fra tjener til klient eller fra nettverk til klient. Den er konstruert slik at utstyret på klientsiden kan holdes så enkelt som mulig. Den gir for eksempel mulighet til å laste ned et komplett operativsystem med tilhørende programvare til en klient hver gang den starter opp. Dette eliminerer kostnadene knyttet til oppdatering og versjonskontroll.

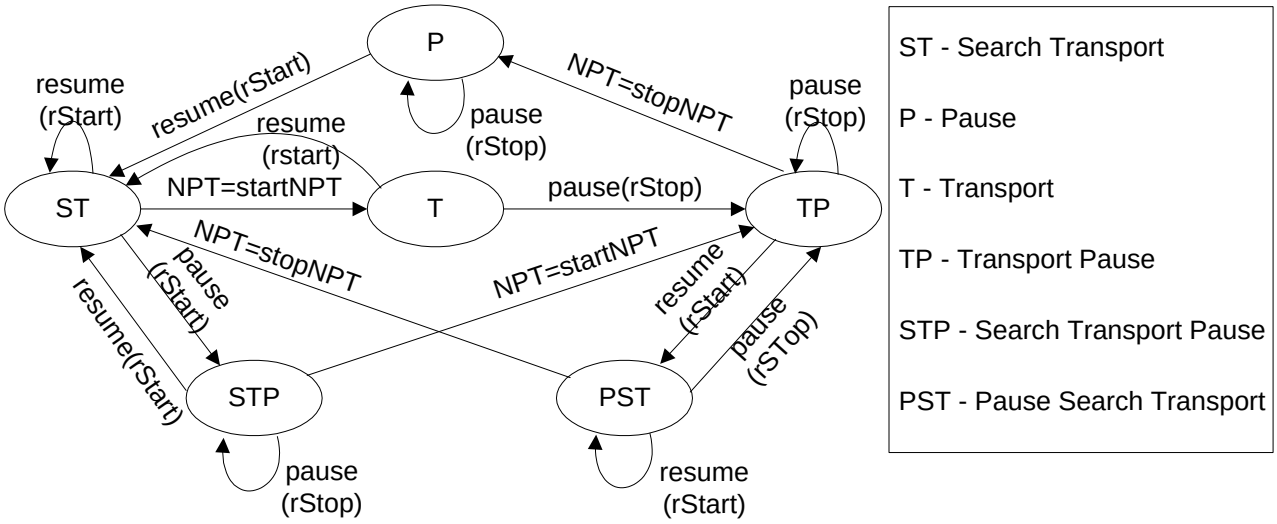
Det er antatt at nedlastingsprotokollen vil finnes i ROM eller flash-RAM hos de fleste klienter. En komplett nedlasting vil overføre et «*Image*» til klienten. Dette er oppdelt i *moduler*, som igjen er oppdelt i *blokker*. Alle blokker, unntatt den siste, vil i et image ha den samme størrelsen. Denne størrelsen blir avtalt i forkant av nedlastingen, og er avhengig av krav til elementer som effektivitet og feildeteksjon.

Protokollen støtter både tradisjonell flytkontrollert nedlasting og ikke-flytkontrollert nedlasting. Det siste alternativet benyttes for å laste ned data fra en tjener til flere klienter samtidig.

Kontroll av videostrømmer

MPEG-2 spesifikasjonen for koding og dekoding av videostrømmer inneholder ikke en beskrivelse av hvordan man skal håndtere tidsstyring. MPEG-2 strømmer inneholder en viss tidsinformasjon, men dette benyttes til å synkronisere audio og video.

For å kunne håndtere tilfeldig posisjonering, pause og spoling av strømmer innfører DSM-CC konseptet «Normal Play Time», *NPT*. Dette er en verdi som indikerer en strøms absolutte posisjon i forhold til starten av strømmen. Når en tjener leverer en strøm, vil den assosiere en NPT-verdi med denne og oppdatere verdien kontinuerlig. NPT kan oppgis i to forskjellige formater; *transport NPT* og *applikasjons NPT*. Førstnevnte format er identisk med PTS[11] formatet. Det er innført for å oppfylle et ønske om å kunne ha samme format på tidsstempler for audio, video og DSM-CC. Applikasjons NPT er oppgitt i sekunder og mikrosekunder for å kunne vært direkte



Figur 22 Tilstandsdiagram for styring av videostrømmer. Hentet fra ISO/IEC 13818-6.

lesbart for mennesker.

Tjeneren som sender strømmen er modellert som en tilstandsmaskin. Transisjon av tilstand skjer ved at mottak av primitiver og ved at visse NPT verdier nås.

Tabellen under viser hvilke metoder som er definert for styring av tilstandsmaskinen. I tillegg til parametrene for NPT, vil det bli overført en skaleringsparameter. Denne angir rate for avspilling og retning. En absoluttverdi lik 1.0 angir normal avspillingshastighet, og negativt fortegn forteller at avspillingsretning skal reverseres.

Kommando	Forklaring
Pause(rStop)	Slutt å send strøm når NPT rStop nås.
Resume(rStart)	Start å send strøm fra posisjon rStart
Status	Hent status om en strøm
Reset	Nullstill tilstandsmaskinen for en strøm
Jump(rStart,rStop)	Når strøm når NPT rStop, forsett å send fra NPT rStart
Play(rStart,rStop)	Send strøm fra NPT rStart til NPT rStop

Metoder for styring av tilstandsmaskin

I Figur 22, hentet fra ISO/IEC 13818-6, er det vist en enkel tilstandsmaskin hvor bare reset(), resume(rStart) og pause(rStop) metodene er tatt i bruk. Play(rStart,rStop) er ikke inkludert fordi funksjonen er identisk med en resume(rStart) fulgt av en pause(rStop). Jump(rStop,rStart) er ikke inkludert fordi funksjonen er identisk med sekvensen pause(rStop) og resume(rStart).

Generelle applikasjon- og kringkastingstjenester

For å støtte multimedia applikasjoner som VOD, spill og teleshopping defineres det i DSM-CC et sett med grensesnitt som bygger på U-U meldinger. Disse grensesnittene er spesifisert i et *IDL* (Interface Definition Language). IDL'en som er brukt stammer fra CORBA 2.0 spesifikasjonen. Språket muliggjør definisjon av grensesnittet til objekter uavhengig av objektenes implementasjon.

[11] Presentation TimeStamp. Se avsnittet om MPEG-2 Transportstrømmer.

Andre løsninger

I dette kapittelet presenteres noen andre videotjenere som implementerer VCR-funksjonalitet. Vi skal kort beskrive arkitekturen til disse videotjenerene og se på hvordan de implementerer VCR-funksjonaliteten.

Microsoft Tiger Video Fileserver

Microsoft Research har utvilket en filtjener de har kalt Tiger, denne er beskrevet i Bolosky m. fl. (1996). Den er en distribuert tjener som kan levere datastrømmer med konstant bitrate til et stort antall klienter på samme tid. Den ble konstruert hovedsakelig for å levere video, men den kan også brukes til å levere andre typer data som krever en konstant, garantert bitrate. Det følger at alle typer video kan leveres av Tiger. I tillegg kan filtjeneren også utføre operasjoner som er vanlige i et filsystem. Arkitekturen til Tiger er som følger: en samling datamaskiner er knyttet sammen over et høyhastighets nettverk (for eksempel ATM). Til hver maskin er det tilkoblet et visst antall disker. Microsoft har valgt å implementere Tiger på vanlige PC'er som kjører Windows NT.

Et viktig mål for Tiger har vært å oppnå en jevn belastning på de tilgjengelige ressursene. Dette oppnås ved å stripe dataene over diskene og maskinene, dvs dele opp dataene og spre de utover flere disker/maskiner. Dette gjør at belastningen blir bedre fordelt på disk-, I/O- og nettressursene, og ytelsen blir bedre. Et annet mål har vært å gjøre tjeneren feiltolerant, dette blir gjort ved å speile (kopiere) data. Dataene blir kopiert til en annen disk på en annen maskin, slik at systemet kan tåle at en hel maskin feiler.

Det mest interessante i denne sammenheng er å se hvordan Tiger oppnår spoling. Dette blir gjort på en relativ enkel måte. Klienten vil under spoling kontinuerlig sende beskjeder til Tiger om å spille av deler av videoen til et gitt tidspunkt. Dermed vil klienten motta stykkevis deler fra den originale videostrømmen. Resultatet er at Tiger ikke trenger å implementere annet enn normal avspilling og tilfeldig aksess inn i strømmen til gitte tidspunkt. Klienten blir på den andre siden tillagt større oppgaver, og blir dermed mer kompleks. Spolekvaliteten vil ikke bli spesielt god siden spolingen består av normal avspilling avbrutt av hopp fram i filmen; vi får en «rykk og napp» effekt.

Sun MediaCenter Server

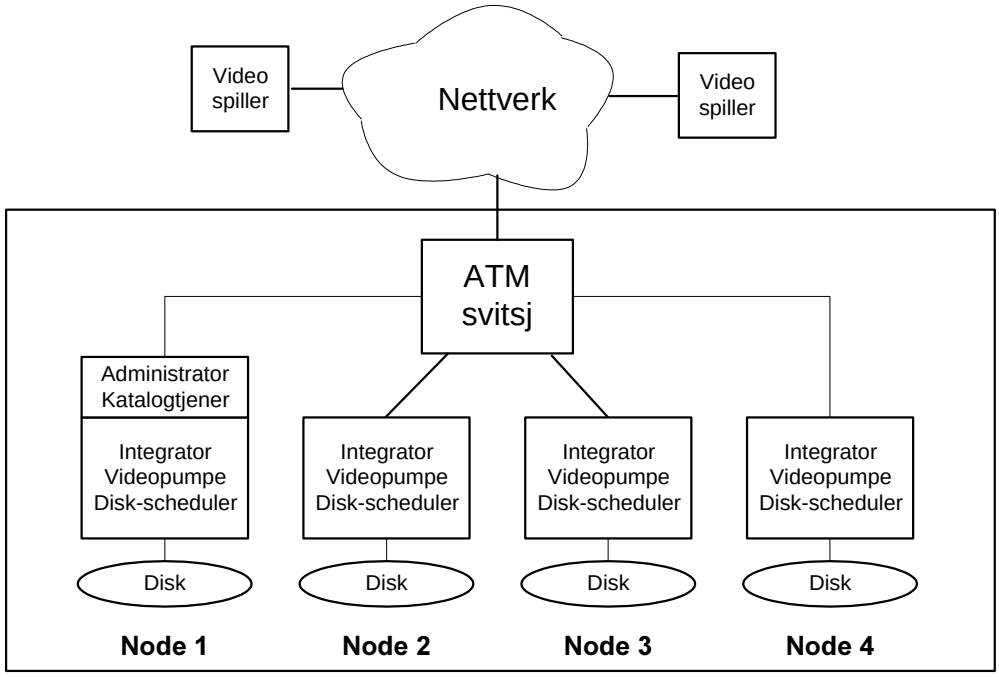
Følgende informasjon er hentet fra web-siden «Sun MediaCenter Servers» (1997). Sun's videotjener, Sun MediaCenter Server (SMC), fungerer (som Tiger) som en filtjener med filsystemoperasjoner. Den består av programvare som er laget spesielt for behandling og oversendelse av videostrømmer. Denne programvaren består av en modifisert versjon av Solaris' operativsystem, nettverksdrivere, et filsystem (Media File System), og en styringsenhet (Media Stream Manager) som styrer brukeraksessene.

SMC bruker et array av disker (Media Storage Array) for lagring av videoen. Diskene er delt opp i grupper der hver gruppe har en paritetsdisk. Systemet håndterer altså feil på disknivå. Filsystemet bruker striping innad i hver gruppe for å fordele belastningen og øke ytelsen. Det finnes tre versjoner av SMC med ulik organisering av denne diskstrukturen. Ulike konfigurasjoner kan også fås for nettverket, både ATM og Fast Ethernet kan brukes.

SMC støtter MPEG-1 og MPEG-2 video. Spoling blir utført ved hjelp av spolefiler. En spolefil skal representere en spolehastighet og ved overgang til spoling må avspillingen byttes fra originalfilen til spolefilen. Krav til spolefilen er at den skal være av samme format som originalfilen, og bitraten skal ikke overstige originalfilens. For å oppnå korrekt posisjonisering i spolefilene må det brukes indeksfiler der det spesifiseres hvor man kan aksessere filen. Dersom slike indeksfiler ikke lages, hoppes det vilkårlig inn i strømmen på et sted som er beregnet utifra bitrate og størrelse på filen. Spolefilene kodes på forhånd utifra det samme bildematerialet som originalfilen lages.

Elvira

Elvira er en eksperimentell videotjener utviklet ved databasegruppa ved IDI. Elvira ble påbegynt høsten 1994 i en diplomoppgave (Langørgen , 1994). Den første utgaven støttet kun MJPEG video, men senere er tjeneren blitt videreutviklet og støtter i dag også MPEG-1. Sandstå m.fl (1997) beskriver dagens utgave av Elvira. Figur 23 viser en skjematisk oversikt over Elvira's arkitektur.



Figur 23 Elvira's arkitektur. Hentet og modifisert fra Sandstå m.fl. (1997)

Tjeneren består av flere noder knyttet sammen med en ATM-svitsj. En node er en arbeidsstasjon med disk. Hver node kjører ulike prosesser for styring av avspilling fra disk. Disse prosessene er en integrator, en videopumpe, og en disk-scheduler. Integratoren har ansvaret for å levere videoen fra noden, samt å svare på forespørsler fra klienten. Videopumpen skal hente videoen fra disk og levere denne til integratoren. Disk-scheduleren kontrollerer diskaksessene etter å ha mottatt forespørsler fra videopumpen. Elvira støtter bruk av striping utover diskene, men kan også benytte ustripet lagring av video.

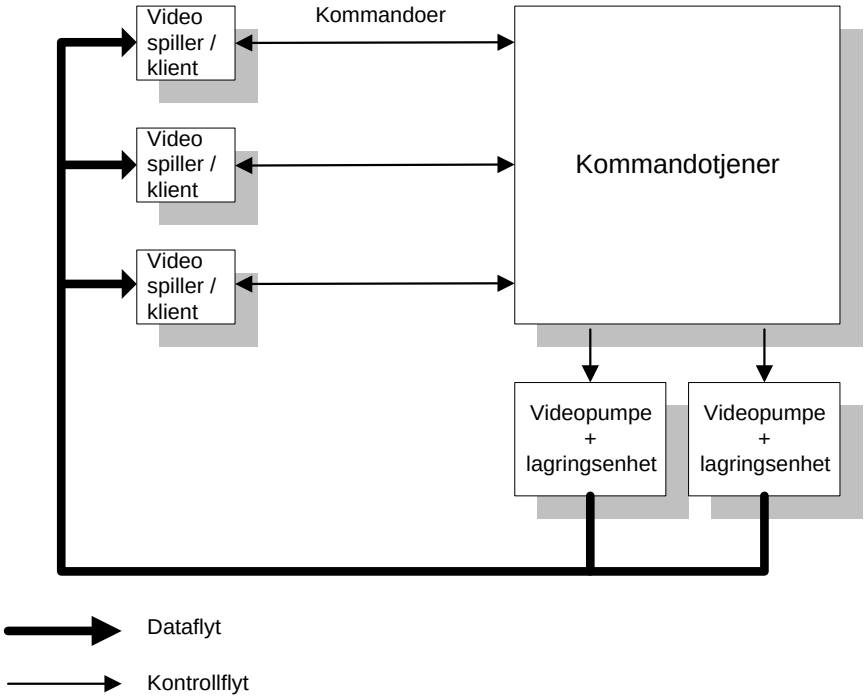
En av nodene vil i tillegg til disse prosessene også ha ansvaret for å kontrollere hele tjeneren. På Figur 23 er det Node 1 som har denne oppgaven. Elvira implementerer to kontrollmekanismer: en administratorprosess (Call control), og en katalogtjener (Catalog manager). Administratorprosessen tar imot forespørsler fra klienter og styrer fordelingen av arbeid over nodene. Katalogtjeneren håndterer informasjon om hvor videoen ligger lagret i tjeneren og hvilket format som er brukt til å kode videoen.

Elvira støtter bruk av MJPEG og MPEG1 og tilbyr både forlengs og baklengs spoling for begge formatene. For MJPEG blir dette foretatt ved å sende over bare en viss andel av bildene avhengig av spolehastighet. For MPEG-1 er to metoder implementert. Den ene bygger på samme prinsipp; man sender over en viss andel bilder, her blir det laget indeksfiler som forteller hvor rammene man kan sende over (I- og P-rammer) ligger. Disse blir så brukt til å aksessere enten bare I-rammene, eller både I- og P-rammene, avhenging av hastigheten. Den andre metoden er konstruksjon av spolefiler. Her blir det laget en ny fil av bilder som plukkes ut fra originalfilen. Bildene plukkes ut slik at en gitt spolehastighet oppnås. Ved spoling vil tjeneren så istedet spille av en av disse filene. Disse to metodene er detaljert beskrevet av Koteng (1996).

Elvira II

Elvira II er en parallell videotjener som utvikles gjennom et samarbeidsprosjekt mellom NTNU og Telenor FoU. I dette kapittelet gis en overordnet beskrivelse av denne, med spesiell vektlegging på filsystemet. Elvira II er fremdeles i utvikling, men den overordnede konstruksjonen som presenteres her, er ferdig. Informasjonen er hentet fra Eurescom (1997), og fra «Overview of the Elvira 2 command server», Sandstå (1997).

Hovedkomponentene i Elvira II er en kommandotjener og flere videopumper. Både kommandotjeneren og hver av videopumpene kjøres på en UNIX arbeidsstasjon og er knyttet sammen gjennom en ATM-svitsj. En grov oversikt over arkitekturen er gitt i Figur 24.



Figur 24 Systemarkitektur for Elvira II

Når en klient ønsker å få levert en video, sender den en anmodning om dette til kommandotjeneren. Denne har tilgang til en katalogtjeneste og en ressursallokator, og finner ut om videoen kan leveres. Om dette er tilfelle vil følgende forbindelser bli benyttet:

- ✓ En TCP/IP forbindelse mellom kommandotjeneren og klient. Denne ble opprettet av klienten.
- ✓ TCP/IP forbindelser mellom kommandotjeneren og videopumper. Disse forbindelsene er permanente.
- ✓ En (flere) UDP/IP[12] forbindelser mellom videopumpe(r) og klient.

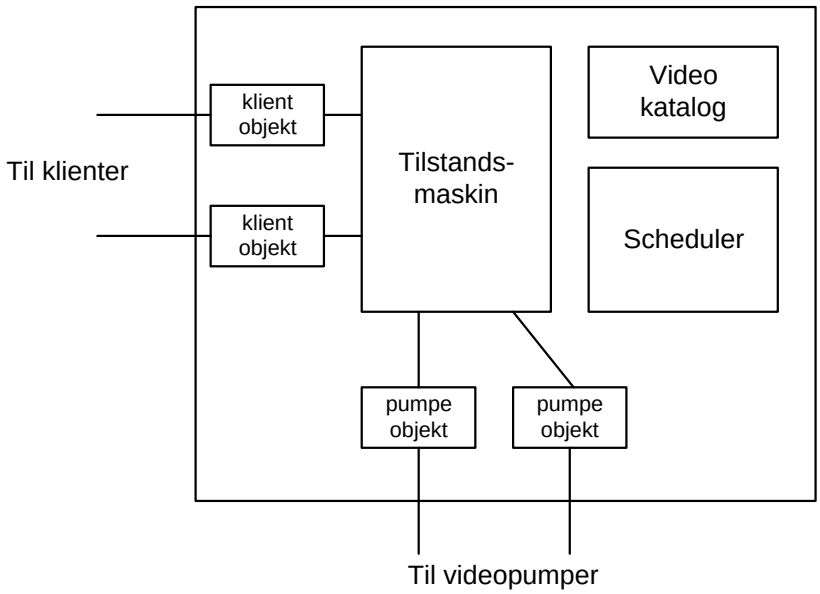
Kommandotjener

Kommandotjeneren kommuniserer med klienten ved hjelp av en protokoll som er basert på MPEG2-kommisjonens DSM-CC protokoll. Denne protokollen inneholder meldinger for initiering/lukking av forbindelse, hurtig- og sakte spoling, pause og oppstart, samt hopping i videostrøm.

[12] Eventuelt ATM AAL5 forbindelse.

I tillegg til å kommunisere med klientene er en av hovedoppgavene til kommandotjeneren å administrere ressursene i systemet. Den skal sørge for at videopumpene blir jevnt belastet og vil avslå forespørsler fra klienter dersom det ikke finnes ressurser til oppgaven.

Kommandotjeneren er bygd opp som vist i Figur 25.



Figur 25 Kommanotjeneren (hentet og modifisert fra Sandstå (1997)

Sandstå (1997) beskriver de viktigste bestanddelene i kommandotjeneren, disse er:

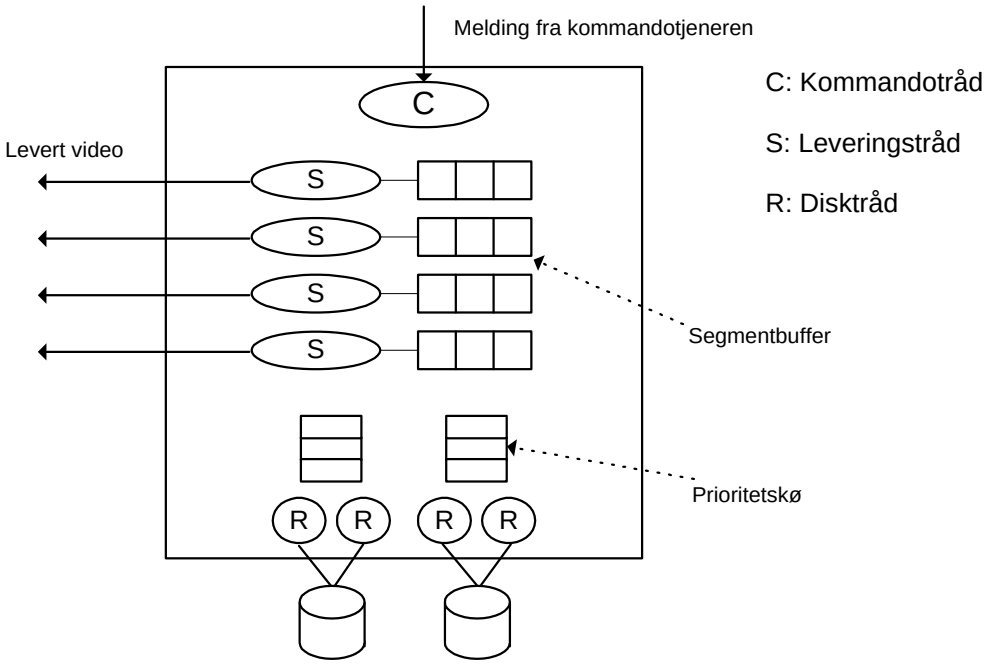
Tilstandsmaskin:
Tjeneren er implementert som en tilstandsmaskin. Endringer i tilstanden vil skje ved at den mottar meldinger fra klienter eller pumper, eller ved at et tidspunkt for utførelse av en kommando er nådd.
Videokatalog:
Denne inneholder informasjon om hvor hver film er lagret og hvilket format denne er kodet på. Denne informasjonen brukes til å finne fram til filmen ved avspilling, men også til å gi klientene informasjon om hvilke filmer som finnes i systemet.
Scheduler:
Denne sikrer at hver levering får tildelt nok ressurser til å gjøre diskaksessering og utsending av containere på nettverket. Dette gjør den ved å ikke tillate flere avspillinger samtidig enn hva tjeneren kan håndtere, styre hvilken videopumpe som skal spille av en film samt styre videopumpenes håndtering av klientkommandoer som påvirker avspillingen.
Klientobjekt:
Dette objektet er ansvarlig for all kommunikasjonen med klienten. En instans av objektet vil være ansvarlig for en klient.
Pumpeobjekt:
Dette objektet vil sende kommandoer til videopumpen, og ta imot svar. Det eksisterer et slikt objekt for hver videopumpe i systemet.

Videopumper

En videopumpe består av en fler-trådet prosess. Følgende tråder er definert for hver pumpe:

- ✓ Kommandotråd: Denne tråden håndterer meldinger fra kommandotjeneren. Typiske meldinger er *allokering*, *frigiving*, *start av*, *stopp*, *modifiser posisjon*, og *angi posisjon* i en videostrøm.
- ✓ Leveringstråd: denne leverer video til klienter. Den inneholder et segmentbuffer som er organisert som et ringbuffer. Det består av tre segmenter. Første segment vil inneholde de neste containerene som skal sendes til klient, mens siste element vil være det segmentet som sist ble lest fra disk. Et slikt segmentbuffer kan ha tilstandene *spiller*, *ledig* og *stoppet*.
- ✓ Disktråd: For hver diskenhet som inneholder lagret videodata blir det opprettet to tråder. Disse henter segmenter fra disk som angitt i en prioriteringskø. Leveringsstråden vil sette inn forespørsler etter segmenter i denne køen.

Figur 26 viser hvordan disse trådene fungerer sammen i en videopumpe.



Figur 26 Videopumpas oppbygning

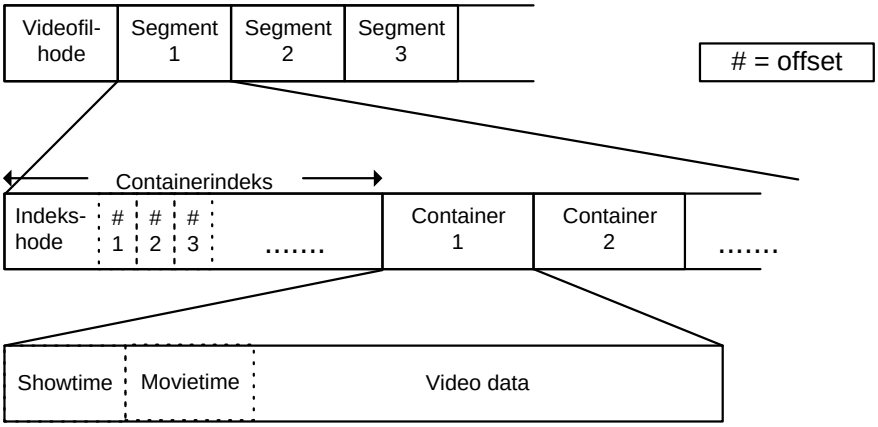
Filsystemet

Elvira II vil bruke et eget filsystem for å lagre videofilene. Filsystemet er enkelt og flatt, og vil bruke og opprettholde enkle tekstfiler som holder oversikt over diskene og ledig diskplass på hver enkelt disk. Elvirafilene kan så legges inn der det måtte være plass til filen.

Filformatet

En elvirafil vil bestå av en filstruktur som vist i Figur 27. En videofil deles opp i et videofilhode og segmenter. Videofilhodet inneholder generell informasjon om videofilen. Et segment er overføringsenheten mellom disk og minne, og har en stor, fast størrelse (f.eks 128 kB). Se vedlegg 3 for en nærmere beskrivelse av filstrukturen.

Hvert segment består av en containerindeks og en del containere. En container er den minste enheten en videopumpe vil håndtere, disse vil i sin helhet bli sendt over nettverket. Containerindeksen inneholder informasjon om antall containere i segmentet samt posisjoneringen og størrelsen av disse.



Figur 27 Filformat under Elvira II

Containere vil inneholde tidsstempler som Elvira II vil bruke til å koordinere utsendelseshastigheten. Tidsstempelet «Movietime» sier hvilket tidspunkt dataene i containeren representerer i den vanlige filmen (Normal Play Time). «Showtime» vil angi når containerdataene skal være framme hos klienten. Disse to stemplene vil være like for normal avspilling, men dersom vi har en form for spoling eller hopping i filmen vil «showtime» måtte endres slik at containerene blir sendt ut tidsnok og i rett rekkefølge.En container kan ha variabel lengde, og er typisk mye mindre enn et segment. Av ytelsemessige hensyn er det valgt å ha en maksimumsstørrelse på containerne som er mindre enn maksimal størrelse på UDP pakker (vanligvis 8 kB for sending over ATM).

Indeksfiler

Kommandotjeneren må vite hvor i filsystemet de ulike filmene ligger. Den må også vite et tidspunkt som knyttes til segmentet slik at den kan hente inn segmentene til rett tid og rett rekkefølge. Dette blir gjort ved å generere *indeksfiler* for hver film. En slik indeks vil inneholde generell informasjon om filmen, og for hvert segment skal det være et innslag som inneholder tidspunkt og lokasjon for segmentet. Lokasjonsbeskrivelsen vil dreie seg om hvilken disk det ligger på, og på hvilken blokk segmentet starter på. Tidspunktet vil representere hvor i filmen dette segmentet tidsmessig hører hjemme og vil angis i millisekunder.

En film kan ha flere instanser av originalfila knyttet til seg der hver instans vil gi en avspilling med ulik hastighet fra originalfila. Indekser for disse instansene skal også ligge i den samme indeksfila. En indeksfil vil dermed kunne bestå av mange instanser der hver instans er en elvirafil som gir en spesifikk avspillingshastighet. For hver instans vil det lagres informasjon om hvert segment som beskrevet over.

Se vedlegg 4 for en detaljert spesifikasjon av en slik indeksfil.

Løsningsalternativer

Dette kapittelet tar sikte på å gi en oversikt over mulige metoder for å oppnå tilfeldig aksess og spoling i en videotjener som Elvira II. Noen metoder vil raskt kunne utelukkes som et sannsynlig alternativ for Elvira II, men tas likevel med her idet de kan være av interesse for videotjenere med andre egenskaper og ytelseskrav, og for kompletthets skyld.

Først vil det bli fortalt litt om hva VCR-funksjonalitet i en digital video egentlig innebærer. Deretter vil det bli beskrevet metoder som kan brukes for implementering av tilfeldig aksess. Til slutt vil flere metoder for å oppnå spoling bli presentert.

For MPEG-2 finnes det to ulike systemstrømmer for lagring av lyd og bilde: programstrømmen og transportstrømmen. Se kapittelet «MPEG-2» for en nærmere beskrivelse av disse to strømmene. Dette kapittelet tar sikte på å beskrive generelle metoder som kan benyttes på begge systemstrømmene. Noen metoder vil likevel rette seg mot en spesiell strøm. I disse tilfellene vil dette bli bemerket.

I dette kapittelet vil det bli lagt vekt på en teknisk beskrivelse av *hvordan* problemene kan løses, en *vurdering* av disse metodene vil bli foretatt i kapittelet «Løsningsvurdering og valg».

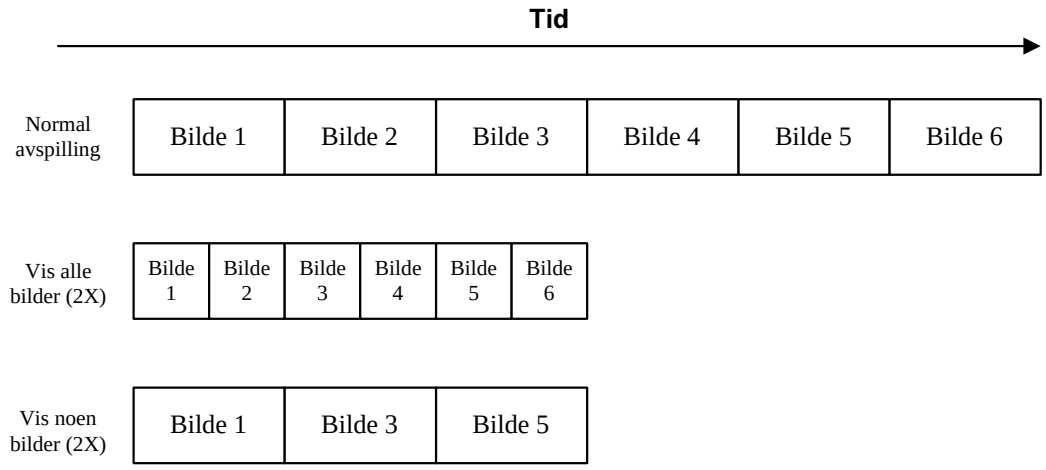
VCR-funksjonalitet

De fleste har erfaring med spoling fra en vanlig videospiller. Det er naturlig å anta at brukere av en digital videotjener vil forvente å kunne utføre spoling på samme måte som de er vant til fra videospilleren. Det er derfor et mål for en videotjener at funksjonaliteten og spolekvaliteten er minst like bra og helst bedre enn for en videospiller.

En videospiller kan som oftest spole både med og uten bilde. Når vi snakker om digital spoling så mener vi som oftest spoling *med* bilde. Spoling *uten* bilde vil i en videotjener erstattes med «Tilfeldig aksess», dvs. man aksesserer et tilfeldig punkt i filmen i løpet av relativt kort tid. Dette er en av de største fordelene ved et digitalt system; man slipper lang ventetid for posisjonering.

Men hva er egentlig spoling? Måtene spolingen blir utført på i disse to tilfellene vil være veldig forskjellige. Spoling i en videospiller blir utført ved at båndet ganske enkelt blir dratt fortere gjennom leseren. Det vil si at hvert bilde blir vist i et kortere tidsrom enn for vanlig avspilling. I et digitalt system vil noe tilsvarende kunne gjøres; man viser hvert bilde i et kortere tidsrom, se Figur 28. Dette medfører imidlertid et problem: datamengden som må overføres fra videotjeneren til klienten vil vokse proporsjonalt med spolehastigheten. Dette er ikke noe problem for en videospiller, men det vil bli svært ytelseskrevende for et datasystem. Dersom dataene for eksempel skal overføres over et nettverk vil nettverksbelastningen skyte i været.

En annen mulighet som finnes er å ikke vise alle bildene. Man kan for eksempel vise kun annethvert bilde. Dersom man da viser hvert bilde like lenge som for normal avspilling har man oppnådd en dobling av hastigheten, se Figur 28. På denne måten vil ikke datamengden som må overføres øke på samme måte. Ulempen ved en slik metode er selvfølgelig kvaliteten; ved høye spolehastigheter vil man «miste» mange bilder og spolingen kan komme til å framstå som «hakkete».



Figur 28 Spolingsprinsipper

For MPEG-standardene vil en implementering av disse enkle spolingsprinsippene kompliseres, særlig gjelder dette for visning av utvalgte bilder. Dette kommer av bildeavhengighetene som finnes, man kan ikke uten videre hente ut et enkeltbilde og vise dette. Man kan heller ikke garantere at datamengden som må overføres blir redusert i «forventet» grad. Dette kommer av at MPEG-2 representerer bildene med ulike rammetyper, og rammestørrelsen vil derfor kunne variere sterkt fra bilde til bilde. Dersom man skal plukke ut enkeltbilder vil det faktisk være sannsynlig at disse bildene vil måtte bruke en datamengde langt over gjennomsnittet. Dette fordi disse må inneholde all informasjon om bildet, i motsetning til andre rammer som bare trenger å inneholde endringsinformasjon (Se kapittelet om MPEG-2 for beskrivelse av I-, P- og B-rammer).

Vi ser at det oppstår problemer med spoling i digital video. Det blir et spørsmål om å velge mellom spolekvalitet og belastning på systemet. De to prinsippene beskrevet over viser dette. Det gjelder å finne metoder som gir en fornuftig avveining. Metodene bør kunne oppfylle de krav som stilles av videotjeneren når det gjelder systembelastning samtidig som de tilfredsstiller kvalitetskrav som brukerne vil stille.

Tilfeldig aksess

Tilfeldig aksess er ganske enkelt en funksjon som tillater brukeren å begynne å se på en video fra et vilkårlig tidspunkt i videoen. Dette kan fra brukerens synspunkt arte seg for eksempel slik: man drar en «slider» over en linje som representerer spilletiden, og slipper denne der man vil at avspillingen av videoen skal begynne. Man kan også gjøre dette på andre måter: innskrivning av tidspunkt, innskrivning av et prosenttall som angir hvor langt ut i videoen man skal starte osv.

Videotjeneren trenger ikke å bry seg med grensesnittet, den skal bare motta et ønsket tidspunkt for avspilling og begynne avspillingen fra et punkt *så nært* dette tidspunktet som mulig. Videotjeneren må altså finne ut hvor i videoen den skal begynne avspillingen. Bruk av MPEG-2 gjør at dette ikke er rett fram, prosessen kan gjøres på forskjellige måter. Følgende to alternativer er funnet: (NB! Merk at alternativ 1 kan kun brukes for Transportstrøm, ikke Programstrøm.)

Identifiser aksesspunkt

For MPEG-2 bør man ikke bare automatisk starte avspillingen på den rammen som brukeren peker ut. Dette kommer av måten strømmen er bygd opp på, med I-, P- og B-rammer. (Se forøvrig avsnittet «MPEG-2» under kapittelet «Bakgrunn» for en mer utførlig beskrivelse av oppbygningen av videostrømmen.)

P og B rammer kan aldri vises uten at det først er dekodet en I-ramme. En I-ramme kan derimot dekodes for seg selv, gitt følgende to betingelser:

1. **Informasjon angående dekodingen er allerede mottatt ved å ha lest et *sekvenshode*[13].**
2. **Denne informasjonen er ikke utdatert, dvs. informasjonen gjelder fremdeles for den aktuelle posisjonen i videostrømmen.**

Betingelse 2 kommer av det faktum at det er mulig å legge inn nye sekvenshoder i strømmen i den hensikt å gi dekoderen beskjed om at den nå må dekode rammene på en annen måte; for eksempel ved å benytte en annen kvantiseringsmatrise.

Dette får den effekt at for å være helt sikker på at man kan starte å dekode en tilfeldig I-ramme må man først lese et sekvenshode som gjelder for den aktuelle I-rammen.

Random_access_indicator

Hvordan finner man så fram til det nærmeste sekvenshodet? MPEG-2 standarden tilbyr bruk av en bit som kalles random_access_indicator (RAI). Denne bit'en finnes i *Transportstrømmen* og har følgende egenskaper:

Når RAI er satt til 1 er følgende punkt sanne:

1. **Den neste PES-pakke tilhørende samme strøm vil inneholde et *aksesspunkt*. Et aksesspunkt er definert på to forskjellige måter for en videostrøm og en audiostrøm:**
   - **Video: Aksesspunktet er den første byten av et video sekvenshode.**
   - **Audio: Aksesspunktet er den første byten av en audioramme.**

2. **Dersom denne pakken tilhører en strøm som blir brukt til lagring av PCR-felt (tidsmerker) så skal den inneholde et PCR-felt.**

Når RAI er satt til 0 er det ikke definert om det kommer et aksesspunkt eller ikke.

Av punkt 1 ser vi at denne bit'en kan brukes til å finne starten på et sekvenshode. Man bør imidlertid kontrollere at den gjelder en videostrøm og ikke en audiostrøm. Når man vet at en transportpakke inneholder en RAI, og har kontrollert at dette gjelder en videostrøm kan man dermed starte avspillingen fra og med denne pakken.

Vi ser at RAI ikke nødvendigvis må brukes, selv om strømmen inneholder aksesspunkt. Dette får den konsekvens at dersom en video er generert uten å bruke RAI, vil det ikke være mulig å oppnå tilfeldig aksess ved hjelp av denne metoden i denne videoen. Det er likevel antatt at produsenter av MPEG-2 videostrømmer i framtiden vanligvis vil komme til å bruke denne bit'en i transportstrømmene.

Bruk av RAI

For at videotjeneren skal kunne finne fram til sekvenshodet må den ha tilgang til informasjon som sier hvor disse sekvenshodene finnes i strømmen. Dette kan gjøres ved å på forhånd generere en tabell som inneholder posisjonen til alle sekvenshodene. Denne må genereres ved hjelp av RAI og kan brukes fortløpende av videotjeneren under kjøring. Tabellen vil inneholde informasjon om hvert aksesspunkt, dvs. hvor i strømmen dette finnes (hvilken pakke, container og segment) og tidspunktet dette aksesspunktet representerer. Når videotjeneren mottar et ønsket tidspunkt fra klienten vil den nå kunne slå opp i denne tabellen og finne nærmeste aksesspunkt.

Filmer uten RAI?

Bruk av RAI krever som nevnt at sekvenshoder er brukt i strømmen, og at RAI er brukt for å identifisere disse. Siden dette ikke er noe krav vil det kunne finnes gyldige MPEG-2 strømmer uten RAI. For å likevel kunne bruke denne metoden på disse filmene kan det være mulig å på forhånd gå igjennom strømmen og sette inn disse

[13] Dette inneholder nødvendige dekodingsparametre.

sekvenshodene før hver GOP, dvs. før den første I-rammen i en gruppe-med-bilder. Siden det alltid må finnes et sekvenshode i starten av en film, kan dette brukes som kopigrunnlag.

Dette vil være en ganske tung prosess, og vil involvere demultipleksing av strømmen, og multipleksing igjen etter at sekvenshodene er satt inn. Dette er imidlertid ikke tidskritisk, prosessen gjøres i sin helhet på forhånd.

Dersom sekvenshoder blir brukt for hver I-ramme, og problemet er at RAI likevel ikke blir brukt, trenger vi ikke gjøre alt dette, da er det nok å analysere strømmen, identifisere sekvenshodene og sette RAI-biten.

Identifiser I-rammer direkte

Istedet for å sette som krav at vi må lese et sekvenshode hver eneste gang vi skal aksessere et tilfeldig punkt, kan vi satse på at det går bra å lese en hvilken som helst I-ramme. Denne metoden forutsetter dermed at måten bildene skal dekodes på, ikke forandrer seg i løpet av filmen. (Dersom den gjør det kan vi få uforutsette resultater.)

Et sekvenshode må imidlertid alltid være lest før dekoderen kan dekode noe som helst. Vi kan derfor ikke starte en avspilling av en film for første gang på et tilfeldig sted ved hjelp av denne metoden. For å få lest et sekvenshode kan vi for eksempel stille som et krav at det alltid skal leses fra begynnelsen av filmen først, her må det alltid ligge et sekvenshode.

Fordelen med denne metoden er at man slipper å være avhengig av bruk av en enkelt bit som i alternativ 1. Bakdelen er at man ikke kan tillate endrede sekvenshoder i stømmene.

Å identifisere I-rammer vil kreve noe demultipleksing og traversering av systemstrømmen. Det finnes to hovedmåter å gjøre dette på. En mulighet er å på forhånd generere tabeller som sier hvor i strømmen man finner starten på I-rammer og hvilke tidspunkt disse representerer. Dette vil være en ikke-kritisk oppgave som gjør at tjeneren lett under kjøring kan finne fram til ønsket posisjon. En annen mulighet er å lete framover/bakover i filmen under kjøring. Dette vil ta tid og ressurser hos videotjeneren og denne varianten vil derfor sannsynligvis ikke være så godt egnet som den første.

Spoling

For at et digitalt videosystem skal tilfredsstille brukerene fullt ut vil det være nødvendig å tilby spoling med bilde. Man skal altså være i stand til å bevege seg igjennom filmen i ulike hastigheter, framover og bakover, og fremdeles se levende bilder. Fra brukerens side vil det være gunstig om dette fortonte seg mest mulig på samme måte som på en vanlig videospiller. Det mest naturlige vil være å la brukeren benytte trykknapper for spolingen. Hastigheten kan angis som forskjellige trykknapper, der hver trykknapp representerer en hastighet i en retning. Det er også mulig å bare bruke to trykknapper for retningsangivelse, og la brukeren angi hastigheten på en annen måte, for eksempel i et datafelt.

Videotjeneren vil motta en ønsket hastighet og en retning, og må foreta seg noe for å oppfylle spoleønsket. Det finnes flere alternativer for å oppnå spoling, og disse presenteres i dette delkapittelet. Metodene vil ha ulike egenskaper de kan vurderes etter, disse vil bli beskrevet i kapittelet «Løsningsvurdering og valg». Det kan likevel her nevnes en viktig egenskap som skiller metodene: preprosessering eller dynamisk prosessering.

- **Preprosessering**:
  Videotjeneren vil ha tilgang til filer/informasjon som allerede eksisterer. Disse vil være generert på forhånd og videotjeneren vil bruke disse for å oppnå spoling (på en eller annen måte). Dette vil sannsynligvis være minst ytelseskrevende for videotjeneren; en del arbeid vil allerede være gjort på forhånd.
- **Dynamisk prosessering:**
  Videotjeneren vil ved utsendelse av videostrømmen dynamisk manipulere originalstrømmen i den hensikt å oppnå en spoleeffekt hos klienten. Dette vil være en prosess som foregår like før/samtidig som leveringen av strømmen foregår. Tjeneren sender altså ikke det samme som ved normal avspilling, men en modifisert versjon.

Preprosesseringen's fordel er at videotjeneren ikke vil bli belastet i samme grad som vil være tilfelle for dynamisk prosessering. Fordelen med en dynamisk metode er at man slipper å bruke særlig mer lagringsplass enn hva originalstrømmen trenger. For en videotjener bør det avklares hvilket argument som skal tillegges mest vekt for å avgjøre hvilken av disse to egenskapene som passer best for tjeneren. Det bør også sies at disse argumentene kan tillegges svært forskjellig vekt avhenging av metoden, for eksempel kan det finnes dynamiske metoder som belaster tjeneren i liten grad osv.

Her følger delkapitler som beskriver forskjellige metoder. Som nevnt vil en vurdering og et valg komme i neste kapittel. Disse metodene beskriver generelle teknikker, det er verd å merke seg at flere av disse utmerket godt kan kombineres slik at de best mulig oppfyller det aktuelle systemets behov.

Endre på tidstempel

Oversikt

En MPEG-2 systemstrøm vil synkronisere lyd og bilde ved hjelp av tidsstempler. Det finnes tre hovedtyper: PCR/SCR, PTS, og DTS. Kortfattet er betydningen av disse som følger:

- **PCR (Program Clock Reference) :** Dette tidsstempelet finnes i transportstrømmen og angir tidspunktet for når den aktuelle pakken skal hentes inn i dekoderen. Det vil også brukes til synkronisering av klokka på dekodermaskinen.

- **SCR (System Clock Reference) :** Dette tidsstempelet finnes i programstrømmen og har tilsvarende funksjon som PCR har i transportstrømmen.

- **PTS (Presentation Time Stamp) :** Dette tidstempelet angir tidspunktet for når det aktuelle bildet eller lyden skal presenteres for brukeren.

- **DTS (Decoding Time Stamp) :** Dette tidsstempelet angir tidspunktet for når det aktuelle bildet eller lyden skal dekodes.

Se kapittel «MPEG-2» for en mer utførlig beskrivelse av betydningen og bruken av disse tidsstemplene.

Manipuleringen

Vi ser at tidsstemplene brukes til å koordinere lyd og bilde ved hjelp av å kontrollere når de skal dekodes og presenteres. Dette kan utnyttes til å framtvinge en spoleeffekt ved å forandre på tidsstemplene. Man kan i teorien kunne få en hvilken som helst spolehastighet på denne måten, i praksis vil det selvfølgelig begrenses av hastigheten på overføringsmedier, maskinytelse osv.

For å oppnå en spolehastighet *høyere* enn normal hastighet må tidsstemplene ganges med en faktor *lavere* enn 1 (mellom 0 og 1). For en spolehastighet *lavere* enn normal hastighet (slow motion) må tidsstemplene ganges med en faktor *høyere* enn 1. Alle tre hovedtyper av tidsstempler må forandres, dvs. ganges med den samme faktoren.

Manipuleringen kan gjøres på to forskjellige måter: ved hjelp av preprosessering eller ved hjelp av dynamisk prosessering av strømmen ved avspilling.

- **Preprosessering:**
  Man har den originale strømmen som utgangspunkt og lager en kopi av denne med kun en forskjell: tidsstemplene er forandret. Dette vil føre til at for hver spolehastighet vi ønsker å ha må vi lage en ny fil som er nøyaktig like stor. Når videotjeneren får beskjed om at den skal spole kan den nå starte å spille av den aktuelle kopien fra tilsvarende posisjon. Den må også sørge for å justere utsendelseshastigheten slik at den samsvarer med spolehastigheten.

- **Dynamisk prosessering:**
  Ved hjelp av denne metoden slipper vi å bruke lagringsplass til ekstra filer. Videotjeneren kan forandre tidsstemplene som finnes i strømmen direkte like før den sender strømmen til klienten.

Dette vil legge en ekstra byrde på tjeneren, men det vil være en relativ enkel oppgave. Også her må utsendelseshastigheten fra tjeneren justeres.

Det er viktig å merke seg at denne metoden alene ikke kan brukes til baklengs spoling. Metoden kan imidlertid kombineres med andre metoder; dersom en annen metode først lager en fil for baklengs avspilling kan denne metoden senere brukes til å få flere spolehastigheter baklengs.

Send utvalgte bilder til klienten

Dette er en metode som går ut på å kun sende utvalgte bilder over til klienten. Dette vil innebære å analysere strømmen, finne fram til de aktuelle bildene for så å sende kun disse. Denne metoden vil kun øke hastigheten på videostrømmen, økningen vil avhenge av antall bilder som plukkes ut. For sakte avspilling bør det brukes andre teknikker.

Denne metoden vil i utgangspunktet være dynamisk; videotjeneren vil måtte gjøre uthentingen i sann tid. Det er imidlertid mulig å gå igjennom strømmen på forhånd for å generere filer som kan hjelpe tjeneren med å finne fram til rammene. Disse filene kan for eksempel inneholde informasjon om hvor de ulike rammene befinner seg i strømmen. En slik framgangsmåte vil dermed være en slags hybrid mellom en dynamisk metode og en preprosesseringsmetode.

For MPEG-2 vil ikke dette være noen triviell oppgave for en videotjener. Elementærstrømmen for video er pakket inn i et systemlag, og siden bildene representeres av ulike typer rammer (I-, P-, og B-rammer) der det finnes bildeavhengigheter, vil oppgaven kompliseres ytterligere.

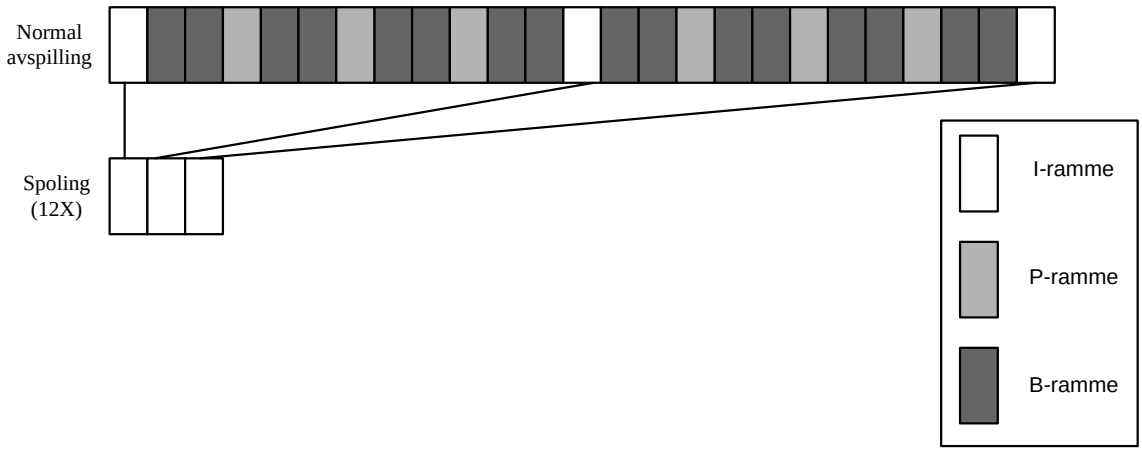
Videotjeneren vil måtte pakke ut de utvalgte bildene fra systemlaget. Ved denne utpakkingen vil videotjeneren derfor måtte fungere som en sanntids demultiplekser. Dersom det er et krav at strømmen som sendes til klienten også skal være en systemstrøm må tjeneren også kunne sette enkeltbildene sammen igjen til en ny systemstrøm (som en enkel sanntids multiplekser). I denne prosessen må tidsstemplene settes/endres slik at den ønskede spolehastigheten oppnås. Dette fordi dersom de tilsvarende gamle verdiene for enkeltrammene blir brukt, vil det ikke bli noen spoling, snarere en normal avspilling med en del bilder utelatt. Denne metoden vil derfor bruke prinsippet skissert i «Endring av tidsstempler» som en innebygget del av hele teknikken. Sakte avspilling kan oppnås ved å justere tidsstemplene, men det er jo meningsløst å bruke denne metoden til dette. Ved sakte avspilling bør man bruke alle bildene, og da er man tilbake til «Endring av tidsstempler».

Man kan dele uthentingsprosedyren opp i to hovedkategorier: Uthenting av kun I-rammer, og uthenting av I- og P-rammer. Hyppigheten av de ulike rammetypene kan varieres. Uthenting av B-rammer anses som uaktuelt siden spole-effekten da vil bli svært liten. Bildeavhengighetene gjør at vi ikke kan hente ut tilfeldige P-rammer uten å hente ut alle I-rammene, og tilsvarende gjelder for B-rammer i forhold til I- og P-rammer (Se avsnitt «MPEG-2» under kapittelet «Bakgrunn»).

Uthenting av kun I-rammer

Dette vil være den enkleste metoden. I-rammer kan dekodes for seg selv og derfor kan hvilke I-rammer som helst hentes ut. Et problem er at det vanligvis vil være en stor rammeavstand mellom I-rammer for MPEG-2. Derfor bør nok alle I-rammene benyttes. Dersom man velger å ikke bruke alle vil nok spolingen framstå som hakkete. Det skulle da heller ikke være nødvendig å utelate noen I-rammer siden den store avstanden gjør at vi får en høy spolehastighet også ved bruk av alle I-rammer.

Figur 29 viser en sekvens med en I-ramme avstand på 12 og en P-ramme avstand på 3, der I-rammene plukkes ut. Dette vil dermed alene gi en spolehastighet på 12X. Det bør noteres at datamengden som må overføres ikke vil bli 1/12 av den originale mengden, den vil bli mye mer. Dette kommer selvsagt av at I-rammer er mye større enn B- og P-rammer.



Figur 29 Spoling med I-rammer

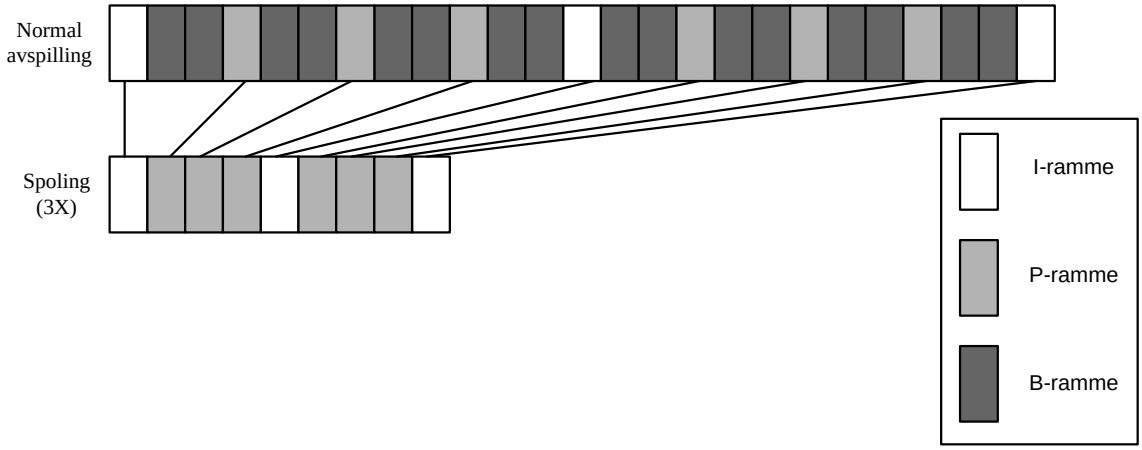
For å redusere spolehastigheten (og dermed nettverksbelastningen) kan det være aktuelt å endre tidsstemplene i tillegg slik at hver I-ramme vises i lengre tid hos klienten. Eventuelt kan det være mulig å sende over «tomme» P-rammer som sier at ingenting skal forandres, dvs. at I-rammen skal vises en gang til.

For å identifisere I-rammer kan tjeneren benytte seg av random_access_indicator (se avsnittet «Tilfeldig aksess»). Denne bit'en identifiserer et sekvenshode, og etter et sekvenshode vil det komme en I-ramme. (Et sekvenshode angir at tilfeldig aksess er mulig, og det får vi ved at en I-ramme er første ramme.) Bruk av RAI vil altså lette arbeidet for tjeneren. Dersom sekvenshoder eller RAI ikke blir brukt i en strøm må tjeneren kontinuerlig de-multiplekse hele strømmen for å identifisere I-rammene.

Baklengs spoling er det også mulig å få til ved hjelp av denne metoden. Dette gjøres ganske enkelt ved å sende over I-rammene i motsatt rekkefølge, dvs. hente den siste først osv.

Uthenting av I- og P-rammer

Ved også å hente noen eller alle P-rammer vil vi kunne oppnå en bedre spolekvalitet. Ved denne metoden er det ikke nok å bruke RAI, her er vi nødt til å de-multiplekse hele strømmen for å kunne identifisere P-rammene. Figur 30 viser uthenting av alle I- og P-rammene i en strøm med I-ramme avstand på 12 og en P-ramme avstand på 3.



Figur 30 Spoling med I- og P-rammer

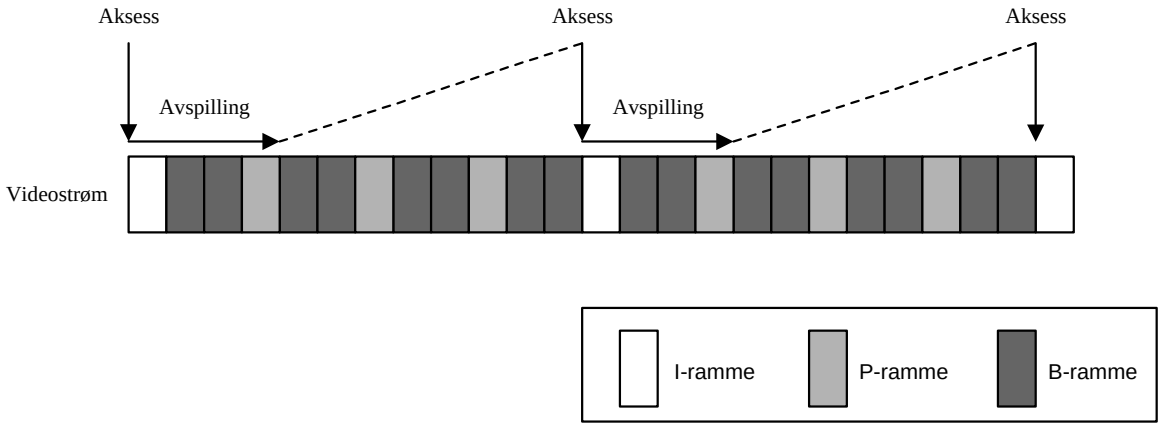
Vi ser at i dette eksempelet får vi en spolehastighet på 3 ganger normal hastighet, noe som er en fornuftig hastighet å tilby i et videosystem.

Ved å redusere antall P-rammer vi tar med, kan vi få en økning av denne hastigheten. Vi kan ikke uten videre redusere antall I-rammer; skal vi fjerne noen av disse må vi også fjerne alle P-rammer som er avhengige av den fjernede I-rammen.

Det er også mulig å få til baklengs spoling ved denne metoden. Det blir riktignok noe mer komplisert enn for uthenting av bare I-rammer. Dette kommer av at P-rammene vil være avhengige av I-rammen som for baklengs avspilling må vises senere enn P-rammene. Dekoderen må altså dekode I-rammen først, og P-rammene etterpå. Det kreves dermed et buffer hos dekoderen slik at den kan omnummerere bildene etter dekodingen; P-rammene vises i motsatt rekkefølge først og I-rammen vises til slutt. Klienten kan dermed sende bildene innen hver sekvens/GOP i riktig rekkefølge, mens hver sekvens/GOP sendes i baklengs rekkefølge.

La klienten ta seg av spoling

Den enkleste metoden for å oppnå spoling (sett fra videotjenerens synspunkt) er å la klienten gjøre jobben. Dette kan for eksempel gjøres ved at klienten kontinuerlig foretar aksesser på en slik måte at en spolingseffekt oppnås. Klienten kan aksessere et punkt i filmen, spille av et eller noen få bilder, hoppe fram (eller tilbake) noen bilder, spille av noen nye bilder osv. Se Figur 31 for en illustrasjon av dette.



Figur 31 Spoling ved aksesser

Ved en slik fremgangsmåte trenger ikke videotjeneren tilby annet enn normal avspilling og tilfeldig aksess. Spolekvaliteten vil imidlertid ikke bli særlig høy, også her vil vi få en hakkete framstilling. Dersom klienten henter et bilde av gangen, vil denne metoden ha noenlunde samme effekt som spoling med I-rammer (se over). Dersom klienten henter / spiller av flere bilder av gangen er det fornuftig å spille av et antall bilder som utnytter B-rammer mest mulig (som vist på figuren).

Å hoppe inn på hver eneste I-ramme (som på figuren) vil nok være vanskelig i praksis pga. forsinkelser o.l. i forbindelse med aksessene. En sjeldnere tilfeldig aksess vil nok være mer gjennomførbar, med flere viste bilder om gangen. Dette vil selvfølgelig føre til en «rykk og napp» spoling.

For en slik metode må klienten fortelle tjeneren hvor dataene den skal ha ligger. Derfor må klienten ha tilgang til en tabell som gir denne informasjonen gitt en bestemt spolehastighet. Denne tabellen bør genereres på forhånd, dermed blir denne metoden en blanding mellom preprosessering og dynamisk avspilling.

Baklengs spoling vil kunne gå an med en slik metode, klienten kan bare bevege seg bakover i aksessene. Dette vil (i teorien) fungere best når klienten bare henter et bilde av gangen. Dersom den spiller av noen bilder for hver aksess vil vi få en blanding av baklengs spoling og forlengs avspilling som sikkert kan virke merkelig for seeren.

Microsoft har i sin videotjener TIGER benyttet en metode som baserer seg på at klienten har ansvaret for spoling. TIGER er beskrevet i Bolosky m.fl (1996).

La klienten få en kopi av originalstrømmen

Det største problemet med spoling er ofte at nettverket ikke klarer å håndtere datamengdene. Dersom klienten får en kopi av hele/deler av originalstrømmen kan klienten spole i denne uten å belaste nettverket. Spolingen vil kunne foregå ved hjelp av en av de dynamiske metodene nevnt over, med den endringen at det er klienten og ikke tjeneren som utfører spolingen.

Dette kan realiseres på ulike måter; det mest naturlige vil nok være å kopiere over filmen til klientens harddisk mens klienten spiller av filmen. Dette vil medføre at spolemulighetene vil bli begrenset: Klienten kan ikke spole i film som han ikke tidligere har spilt av normalt. Foroverspoling vil dermed lide mest under en slik framgangsmåte. En annen metode vil kunne gå ut på å kopiere over hele filmen på forhånd. Lang ventetid og unødvendig nettrafikk når klienten ikke skal se hele filmen gjør at denne varianten ikke er særlig elegant.

Denne metoden vil nok ikke være særlig gunstig for de fleste videotjenere. Det kan imidtertid tenkes at den kan være aktuell i system der det forventes spesielle bruksmønstre. For eksempel kan dette lønne seg dersom det er ventet at hver bruker vil bruke hver film i lange perioder og vil bruke spolefunksjonene flittig.

Spolefiler generert fra systemstrøm

Oversikt

En metode som virker lovende er bruk av *spolefiler*. En spolefil er her en fil som genereres ut ifra den originale strømmen på forhånd. Denne metoden er derfor en preprosesseringsmetode. Spolefilen vil representere den samme filmen, men vil være kodet på en annen måte. Videotjeneren vil få tilgang til disse spolefilene og kan bruke de isteden for den normale avspillingfilen når klienten setter igang en spolesekvens. Poenget med å bruke spolefiler er at de lages slik at de bruker mindre datamengder og dermed ikke belaster nettet så mye ved oversendelse. Dette resulterer i dårligere kvalitet på filmen, men dette kan normalt tolereres ved spoling. Det ideelle vil være å holde nettverksbelastningen konstant uavhengig av spolehastighet. For å få til dette vil kvaliteten selvfølgelig måtte bli dårligere jo større spolehastigheten er.

En spolefil vil representere en spolehastighet i en bestemt retning. Det bør derfor genereres flere spolefiler som hver representerer ulike hastigheter og retninger. Når videotjeneren mottar beskjed om at klienten ønsker å spole skal den stoppe den normale avspillingen, finne spolefilen som samsvarer best med ønsket spolehastighet/retning, finne fram til riktig tidspunkt i spolefilen, og starte avspillingen derfra. Ved å spille av den valgte spolefilen i normal hastighet vil det nå kunne oppnåes en spoleeffekt.

Spolefilene vil måtte genereres på en slik måte at en spolingseffekt oppnås. Dette kan gjøres ved å manipulere på parametre for koding og multipleksing, samt ved å plukke ut enkeltbilder (Se senere i dette avsnittet).

Spolefilkonstruksjon

Å konstruere en spolefil krever mange steg. Grunnen til dette er blant annet at videoen er pakket inn og multiplekset i systemlaget (se kapittel «MPEG-2»). Som nevnt så er en original systemstrøm innparameter i denne prosessen. Siden det ikke trengs lyd for spolefiler trenger vi bare å tenke på video-elementærstrømmen. Den første oppgaven vil derfor være å trekke ut denne elementærstrømmen fra systemstrømmen. Deretter må denne dekodes slik at vi får ut enkeltbilder. Disse enkeltbildene kan nå kodes til en ny elementærstrøm. Denne kodingen kan foregå på flere måter og beskrives nedenfor. Tilslutt må denne pakkes inn i en ny systemstrøm, slik at spolefilene er av samme format som den originale avspillingsfilen.

Spoleteknikker

Det finnes flere teknikker som kan brukes ved kodingen av spolefiler. Det er to ting som skal oppnås: å få en spoleeffekt samt å redusere datamengden (ved å redusere kvaliteten).

Følgende metoder kan brukes til å degradere kvaliteten på videoen:

- Det er mulig å øke komprimeringsgraden for enkeltbildene. Dette kan gjøres ved å benytte andre kvantiseringsmatriser som sørger for grovere kvantisering av hvert enkeltbilde.
- Ved å endre parametre for bevegelsesvektorer kan vi øke komprimeringen.
- Man kan også styre bruken av I-, P- og B-rammer. Her finnes det flere kombinasjonsmuligheter:
  - Å bare bruke I-rammer.
  - Å bruke I- og P-rammer.
  - Å bruke I-, P, og B-rammer.

  For de to siste punktene er det også mulig å variere avstanden mellom de ulike typene, for eksempel kan man la andelen av B-rammer være stor. Bruk av mange P- og B- rammer vil redusere datamengden, men vil redusere aksessoppløsningen. (Se avsnittet om tids-synkronisering).

En intelligent koder vil kunne degradere kvaliteten automatisk ved at brukeren setter en maksimal bitrate som er lovlig for filmen.

For å endre avspillingshastigheten kan følgende teknikk benyttes:

- Man kan endre rammeraten (framerate) for videoen. Denne verdien sier hvor mange bilder som skal vises i sekundet.

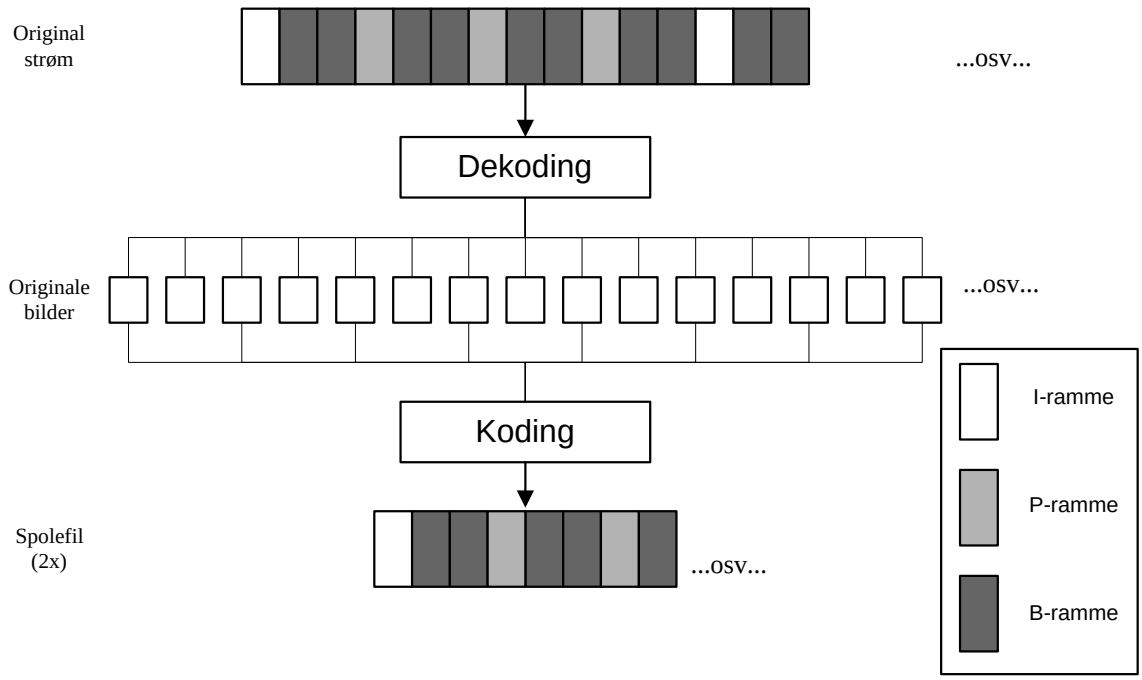
I tillegg til disse metodene vil det være selvsagt være mulig (og nødvendig) å bare benytte en viss andel av bildene, dvs. kutte ut en del bilder. For eksempel kan annethvert bilde brukes for å oppnå dobbel hastighet. Dette vil medføre både en kvalitetssenkning og en hastighetsendring.

En del eksperimentering bør gjøres her for å finne ut hvordan disse metodene kan kombineres på en best mulig måte. Målet er en fornuftig avveining mellom nettverksbelastningen (datamengde) og spolekvalitet.

Her følger noen eksempler på hvordan spolefiler kan lages:

Forlengs spoling, 2x, I, P, og B-rammer

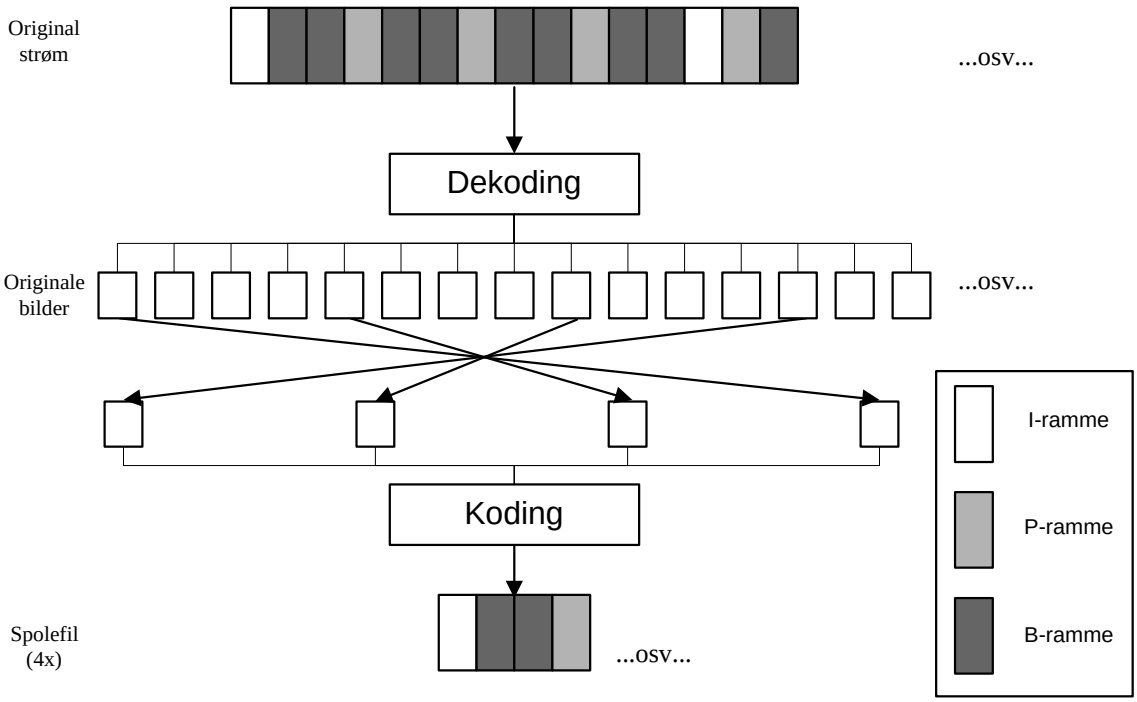
Her velges annethvert bilde for å oppnå en spoleeffekt på 2 ganger normal hastighet. Normale I- og P-ramme avstander velges (I-rammer : 12, P-rammer: 3). Rammeraten kodes slik at den er lik originalfilens rammerate.



Figur 32 Forlengs spoling, konstruksjon av spolefil

Baklengs spoling, 4x, I, P, og B-rammer

For generering av reverserte spolefiler (baklengs spoling), kan rekkefølgen på enkeltbildene ganske enkelt byttes om før koding. Her vises et eksempel for 4 ganger normal avspilling, baklengs spoling. Rammeraten beholdes også her.



Figur 33 Baklengs spoling, konstruksjon av spolefil

Tids-synkronisering

En oppgave som må løses ved bruk av spolefiler er: Hvordan skal tjeneren vite hvor i spolefila den skal starte avspillingen fra når spolingen starter, og hvor i den normale fila skal den fortsette fra når spolingen er ferdig? For å få til dette er det nødvendig å lage noen mappingtabeller som definerer hvilke tidspunkt i normal-fila som tilsvarer hvilke tidspunkt i spolefilene. Denne/disse tabellene må altså kunne brukes både ved begynnelse og avslutning av spolingen.

Aksessoppløsningen, dvs. hvor ofte i spolefila vi kan starte avspillingen, bestemmes av avstanden mellom I-rammer i spolefilen. En kort avstand vil gjøre at vi med stor sannsynlighet kan få en jevn overgang mellom normal avspilling og spoling, men vil gjøre spolefilen større.

I de tilfeller hvor transportstrømmen brukes vil det være fornuftig å bruke RAI (random_access_indicator) for identifisering av aksesspunkt i spolefilene og i originalfilen. Dersom RAI allerede er brukt for tilfeldig aksess i originalfilen (noe som anbefales dersom spolefiler også skal bruke RAI) så trengs det bare å sørge for at spolefilene også kan bruke RAI. Dette kan gjøres ved at det foran hver I-ramme legges inn et sekvenshode ved generering av spolefilene.

Produser filer for spoling direkte

Ved produksjon av en MPEG-2 film trengs det videoutstyr (eller ferdige bilder) og en MPEG-2 koder. Ved koding blir det satt en mengde parametre som bestemmer hvordan filmen vil se ut. I tillegg til å lage hovedfilen kan det være mulig under samme prosess (fra samme bildemateriale) å kode ekstra spolefiler ved å variere disse parametrene.

Dette vil være en pre-prosesseringsmetode som egentlig fungerer på samme måte som ved spolefiler generert fra systemstrøm. Forskjellen er at her slipper vi å demultiplekse og dekode bildene først. Dermed sparer vi tid og arbeid. Bruk av spesiell maskinvare for koding (istedet for software-koding) vil naturligvis være en stor fordel.

Parametre som styrer spolehastigheten vil være rammerate (framerate) og bitrate. Parametre som styrer kvalitet/spolefilstørrelse vil bl.a. være avstand mellom I-, P- og B-rammer og kvantiseringmatriser. I tillegg kan selvfølgelig bare enkelte bilder brukes. Se ellers avsnittet «Spolefiler generert fra systemstrøm» for eksempler på hvordan dette kan gjøres.

SUN's videotjener «SUN MediaCenter Server» (1997), benytter en metode som kan realiseres ved hjelp av denne metoden. Tjeneren krever for spoling et sett av strømmer som alle vil representere ulike spolehastigheter og retninger.

Løsningsvurdering og valg

I forrige kapittel «Løsningsalternativer» ble det beskrevet metoder for å oppnå tilfeldig aksess og spoling med MPEG-2 i en videotjener. Metodene ble generelt beskrevet, det vil si at de i utgangspunktet kan brukes på en hvilken som helst videotjener. For ulike videotjenere vil selvsagt metodene kunne vurderes annerledes avhengig av tjenerens arkitektur osv. Dette kapittelet vil ta for seg disse metodene og vurdere de med hensyn til visse kriterier. Vurderingen vil gjøres med tanke på Elvira II, videotjeneren som blir utviklet ved gruppe for databaseteknikk, IDI.

Først vil vi foreta et valg med tanke på hvilken systemstrøm som bør brukes; programstrømmen eller tranportstrømmen. Deretter vil det bli gitt en oversikt over hvilke kriterier som skal legges til grunn for vurderingen. Hvordan de ulike kriteriene bør vektlegges i forhold til Elvira II vil også bli kommentert. Så vil vi foreta en vurdering og et valg for henholdsvis tilfeldig aksess og spoling. Til slutt oppsummeres valgene og vi ser på hvilke konsekvenser disse vil få.

Programstrøm / Transportstrøm

MPEG-2 standarden tilbyr bruk av to systemstrømmer for lagring og overføring av lyd og bilde. Programstrømmen er hovedsakelig konstruert for lagring på et feilfritt medium. Transportstrømmen vil være delt opp i små transportpakker på 188 bytes. Dersom en pakke forsvinner skal dekoderen takle dette greit, svært lite data vil gå tapt. Samtidig skal den kunne synkronisere seg inn igjen ved hjelp av en synkroniseringsbyte først i pakkene. Den er av disse grunnene egnet for overføring over nettverk og andre medium der det kan oppstå feil.

Elvira II skal sende videoen over nettverk, og derfor virker transportstrømmen som et bra valg. Men Elvira II vil pakke dataene inn i containere som beskrevet i kapittelet «Elvira II». Dette gjør at dersom vi mister en container vil vi miste en hel mengde pakker. Derfor mister dette argumentet litt av sin verdi. Likevel vil transportstrømmen ha en klar fordel framfor programstrømmen: dekoderen vil kunne ta seg raskt inn igjen etter datatapet pga. synkroniseringsbyten som ligger først i alle pakkene. Dette ville ikke ha vært tilfelle ved bruk av programstrømmen.

Et annet argument som taler for bruk av transportstrømmen er at det er mye lettere å behandle strømmen. Dette kommer av den faste lengden på transportpakkene. Det blir enkelt å pakke de inn i containere og det er enkelt å trekke ut nødvendig informasjon fra hver enkelt pakke.

Av disse grunnene velger vi derfor å bruke transportstrømmen som format i Elvira II.

Vurderingskriterier

Her følger en rekke kriterier som en videotjener kan vurderes etter. Man kan identifisere tre hovedkategorier for disse: Tjenestekvalitet, Ressursbruk og Arbeidsmengde (for utvikler og personell). Tjenestekvaliteten vil til syvende og sist være viktigst, men man må huske på at tjenestekvalitet og ressursbruk henger sammen; dårlig ressursbruk vil føre til dårlig tjenestekvalitet.

Tjenestekvalitet

Dette sier noe om hvordan brukeren oppfatter systemet. Tilbys nok funksjonalitet? Er systemet behagelig å jobbe med? Hvordan ser avspillingen ut? Kort sagt: hva vil brukeren synes om systemet? Det finnes mange kriterier å vurdere kvaliteten etter, men vi skal konsentrere oss om de som vedrører tilfeldig aksess og spoling.

Spolekvalitet

Med dette begrepet menes det hvordan spolingen ser ut for brukeren. Dette blir altså subjektivt og derfor vanskelig å måle. Men man kan foreta visse antagelser som tar utgangspunkt i ting som andel bilder vist, tid forløpt mellom hvert bilde og kvalitet på enkeltbilder. Dermed kan vi likevel si noe om hvordan spolekvaliteten kan komme til å bli ved ulike metoder. Generelt kan vi si at kvaliteten på spolingen som regel kan reduseres betraktelig i forhold til vanlig avspilling, og dette gjelder proposjonalt: økende spolehastighet tillater dårligere kvalitet. Likevel er det selvsagt et mål at kvaliteten blir best mulig.

Spolemuligheter

Hvor mange spolehastigheter i hver retning tilbyr systemet? Her tenker vi altså på forlengs og baklengs spoling, og hastigheter både mindre og høyere enn normal avspillingshastighet. Her må det tas hensyn til hvilken bruk videosystemet vil få, nødvendigheten av mange hastigheter vil variere fra bruksområde til bruksområde.

Responstid

Responstiden er tiden det tar å utføre et skifte av avspillingsmodus. Et skifte av modus inkluderer overgangen mellom alle mulige avspillingsmodus, for eksempel skifte fra normal avspilling til forlengs spoling, eller fra baklengs spoling til normal avspilling. Kriteriet responstid inkluderer også tilfeldig aksess; hvor lang tid tar det å hoppe til et tilfeldig punkt i filmen?

Nøyaktighet

Ved skifte av avspillingsmodus bør videotjeneren gjøre dette slik at overgangen blir så jevn som mulig, dvs at den nye avspillingen fortsetter så nært som mulig der den forrige slapp. Det er ikke en selvfølge for alle metodene at den fortsetter på samme sted. Dette gjelder også for tilfeldig aksess: man bør få startet avspillingen så nært som mulig det tidspunktet eller den rammen man ber om.

Elvira II og tjenestekvalitet

Siden Elvira II er en eksperimentell videotjener vil det være viktig å oppnå gode resultater for alle de tre første ovennevnte kriteriene. Poenget med tjeneren er jo å undersøke hvordan en tjener kan konstrueres slik at den blir best mulig å bruke. God responstid er essensielt for at brukeren skal føle seg komfortabel, og det er ønskelig at spolekvaliteten holder bra nivå. Det er også viktig for Elvira II at hver «hovedkategori» av spolemuligheter blir dekket slik at det vises at de er mulige å få til (dvs. baklengs, hurtig og sakte og forlengs, hurtig og sakte). Det er ikke så kritisk at tjeneren er veldig nøyaktig, selv med dårlig nøyaktighet vil systemet kunne brukes på en tilfredsstillende måte.

Ressursbruk

For å oppnå en god tjenestekvalitet er det nødvendig å bruke ressursene på en fornuftig måte. Med ressurser menes her de fysiske komponentene som et videosystem vil bestå av. De ulike metodene som skal vurderes vil alle bruke disse ressursene på forskjellige måter, og noen vil være mer belastende enn andre på enkelte ressurser. Generelt er det viktig at bruken av ressursene blir fordelt slik at det ikke oppstår flaskehalser i systemet. Her følger beskrivelser av ressurser som er viktige i et videotjener-system.

Nettverket

I en videosystem der vi har klienter som bruker en tjener vil vi måtte ha et nettverk. Dette nettverket brukes til både kommunikasjon mellom klient og tjener og oversendelse av video. Nettverket har bare en gitt kapasitet og når flere brukere skal bruke det samme nettverket samtidig er det viktig at hver bruker belaster nettverket minimalt. Særlig ved spoling er det viktig å passe på å ikke overbelaste nettverket.

Diskbelastning

En harddisk vil ha en maksimal overføringskapasitet som vil være en begrensende faktor. Harddiskene blir på samme måte som nettverket belastet hardt dersom store datamengder må overføres. Denne belastningen vil derfor i stor grad være knyttet til nettverksbelastningen. Hver harddisk som brukes vil også ha en viss *aksesstid* og dersom en teknikk går ut på å foreta mange aksesser i løpet av kort tid kan også dette bli et problem. Vi ser at bruken av harddisker vil kunne ha en stor betydning for ytelsen.

Diskforbruk

Videoen trenger (vanligvis) å lagres på disk før den sendes ut til klienten. Ved enkelte spoleteknikker kreves det også at tilleggsinformasjon lagres på harddisk. Et punkt vil derfor være: hvor mye diskkapasitet hos tjeneren kreves for hver film som videotjeneren skal tilby?

Det kan også være aktuelt å bruke harddisken hos klienten, i slike tilfeller må også dette diskforbruket taes hensyn til.

CPU/Minneforbruk

Dette punktet gjenspeiler hvor mye arbeid som videotjeneren må utføre. Dersom spoleteknikker omhandler tunge oppgaver kan tjeneren få problemer med å takle dette dersom flere brukere kommer med forespørsler samtidig. Selv om CPU'ene er kraftige og arbeidsminne er billig er kapasiteten uansett begrenset.

Elvira II og ressursbruk

Det viktigste og vanskeligste å kontrollere for en videotjener vil nok ofte være nettverksbelastningen. Dette gjelder også for Elvira II. Det er viktig at nettverket ikke blir overbelastet idet dette vil kunne føre til store problemer, både med avspilling og annen kommunikasjon. I forbindelse med spoling for Elvira II er det ønskelig at nettverksbelastningen som en bruker påfører systemet ikke øker når vedkommende benytter en spolefunksjon. Dette vil føre til at man kan sette en maksimalgrense for nettverksbelastning for en bruker og dermed en maksimalgrense for antall brukere i systemet.

Diskforbruk per film er også ganske viktig siden en spillefilm i MPEG-2 formatet allerede vil ta store mengder med harddiskplass. En stor økning av diskforbruket vil kunne bli dyrt. Diskbelastningen vil sannsynlivis være enda mer kritisk. I situasjoner med mange brukere vil båndbredden på diskene begrense leveringskapasiteten. Det er derfor også her et mål at nødvendig overføringshastighet fra disk ikke øker ved overgang til spoling.

Det antas at de fleste teknikker ikke vil føre til at CPU/Minneforbruk vil bli noe problem. Pris og ytelse på disse ressursene forbedres stadig. Dersom det *skulle* bli et problem må det selvsagt tas hensyn til, da dette likefullt vil føre til forsinkelser og problemer.

Arbeidsmengde

Dette siste punktet går ut på hvor mye arbeid som kreves for å gjennomføre teknikkene. Vi kan dele dette opp i to kategorier: arbeid som kreves for å utvikle systemet, og arbeid som kreves for drift av systemet.

Utviklingsarbeid

Hvor mye arbeid kreves det for å utvikle et system ved de ulike teknikkene? Dette punktet kan inkludere design, implementasjon og testing.

Driftsarbeid

Det kan tenkes at ulike teknikker kan kreve ulike mengder manuelt arbeid under kjøring av systemet. Dette kan være manuell kjøring av flere programmer, flytting av data, installere ny video i systemet, montering av hardware osv.

Elvira II og arbeidsmengde

Når det gjelder utviklingsarbeid så må nok dette taes hensyn til. For denne oppgaven er det en tidsfrist som skal overholdes og det bør være gjennomførbart å få implementert og testet hovedprinsippet for en teknikk.

Driftsarbeid vil ikke være noe tungtveiende kriterie for Elvira II, det vil uansett ikke bli mye arbeid. Dessuten er dette et forskningsprosjekt der ytelse og funksjonalitet er viktigere. Vi velger derfor i det følgende å ikke legge noe vekt på dette i denne vurderingen for Elvira II.

Tilfeldig aksess

I dette kapittelet vurderer vi de to alternativene som ble funnet for tilfeldig aksess inn i en videostrøm. De vil bli vurdert etter de ovennevnte kriterier og det tas hensyn til at Elvira II vil bruke transportstrømmen som systemformat.

Identifisering av aksesspunkt med RAI

Denne metoden går som nevnt ut på å bruke en bit RAI (Random_access_indicator) som finnes i transportstrømmen til å identifisere aksesspunkt i strømmen. Posisjonene til disse aksesspunktene vil innhentes på forhånd og lagres i en tabellstruktur.

Tjenestekvalitet

For tilfeldig aksess vil det hovedsaklig være to ting som avgjør hvor bra tjenesten er: responstiden og nøyaktigheten. Nøyaktigheten vil være avhengig av hyppigheten av aksesspunkt, det vil si sekvenshoder. Derfor kan den oppnådde nøyaktigheten variere etter hvordan videostrømmen er laget; hvor ofte I-rammer blir brukt og om sekvenshoder blir brukt for hver I-ramme. Det vanlige vil være å ha sekvenshoder for hver I-ramme og med en vanlig I-ramme hyppighet vil vi få anslagsvis 2 aksesspunkt i sekundet for en video av TV-kvalitet. Vi vurderer dette til å være en bra nok nøyaktighet for Elvira II.

For vurdering av responstiden må vi se på hva videotjeneren blir nødt til å gjøre når den får beskjed om å hoppe til et bestemt tidspunkt. Først vil den måtte slå opp i tabellen for å finne den fysiske posisjonen til dataene for det nærmeste aksesspunktet. Dette vil medføre søking i en enkel, sortert tabell. Dersom tabellen kan ligge i arbeidsminne under kjøring vil dette ta svært liten tid. For en 2 timers film og med 2 aksesspunkt i sekundet får vi 2*3600*2 = 14400 aksesspunkt. Det burde ikke være noe problem å ha hele tabellen i minnet. I arbeidsminne blir søketiden i 14400 sorterte elementer ubetydelig. Hvert innslag må inneholde et tidspunkt og posisjon og dette vil sannsynligvis ikke ta mer enn 50 bytes. Dermed vil tabellen anslagsvis kunne ta ca. 700 kb. Dersom dette er en problematisk størrelse å ha i minne vil det kunne gå an å dele inn denne i flere deler slik at bare en del ligger i minnet samtidig. For tilfeldig aksess «langt unna» nåværende posisjon må da tjeneren lese en ny del fra disk. Etter at posisjonen er funnet vil tjeneren kunne fortsette avspillingen direkte derfra. Vi ser at det ikke vil være noe i denne metoden som vil skape noen forsinkelse å snakke om; tvert imot, det er vanskelig å tenke seg en metode som vil være hurtigere.Ved denne metoden vil det være andre aspekter ved Elvira II som eventuelt vil skape en forsinkelse.

Konklusjonen når det gjelder tjenestekvaliteten blir derfor at denne metoden vil egne seg godt. Både nøyaktigheten og responstiden vil være tilfredstillende.

Ressursbruk

Nettverksbelastningen er ikke et problem her, den vil bare bestå av en initiell kommunikasjon. Diskforbruket vil heller ikke være noe problem; tabellen vil være meget liten i forhold til den totale størrelsen på filmen. Dersom tabellen legges i minne vil diskbelastningen (forårsaket av tilfeldig aksess) begrense seg til å lese tabellen en gang ved oppstart av filmen. CPU'en vil heller ikke bli belastet særlig mye, søkingen vil være en liten oppgave siden tabellen allerede er sortert og ikke vil være så veldig stor. Som nevnt vil noe av minnet brukes til lagring av tabellen, men det skulle gå bra, tjeneren vil ha høy nok minnekapasitet. I det hele tatt ser vi at ressursbruk ikke er

noe problem for denne metoden, noe som i høy grad skyldes at dette er en preprosesseringsmetode; mesteparten av arbeidet er gjort på forhånd.

Arbeidsmengde

Arbeidsmengden som trengs for å utvikle programvare som lager disse tabellene vil ikke være så veldig stor. Random_access_indicator er lett identifiserbar i strømmen, det som i tillegg må gjøres er å knytte hvert aksesspunkt til det aktuelle tidspunktet i filmen og til en posisjon i lagringsstrukturen i Elvira II der dette aksesspunktet vil finnes. Det trengs heller ikke å gjøres så mye arbeid med videotjeneren for å bruke denne metoden, oppgaven er ganske triviell.

Identifisering av I-rammer

Dette er en metode som går ut på å identifisere «på egen hånd» alle I-rammene i en strøm. Også her kan det lages en tabell som sier hvor disse I-rammene finnes og hvilke tidspunkt de representerer. Det er også mulig å gjøre dette dynamisk under kjøring.

Tjenestekvalitet

Tjenestekvaliteten vil for denne metoden som oftest være svært lik den første metoden. For bruk av en ferdiglaget tabell vil de samme argumentene som beskrevet i forrige avsnitt gjelde for responstiden. Den vil kanskje til og med bli noe bedre (men marginalt) siden vi ikke leser sekvenshodet ved denne metoden. Å gjøre dette dynamisk vil ta altfor lang tid, vi regner derfor tabellvarianten til å være den eneste aktuelle her, og vurderer bare denne i resten av dette avsnittet.

Nøyaktigheten vil være optimal med denne metoden, vi har mulighet til å starte på hver eneste I-ramme (Noe som også *som oftest* vil være tilfelle med den første metoden).

Problemet med denne metoden er at dersom filmen endrer kodeteknikk underveis, så vil ikke dekoderen få beskjed om dette med en gang hoppet er gjort (siden informasjon om dette ligger i sekvenshodet). Dette kan føre til merkelige bilder fram til neste sekvenshode, eller i verste fall at dekoderen ikke takler situasjonen og stopper.

Ressursbruk

Under kjøring vil vi her ha nøyaktig den samme situasjonen som for «Identifisering av aksesspunkt med RAI». Se derfor avsnittet over for en beskrivelse av ressursbruken. Man vil imidlertid bruke flere ressurser under konstruksjonen av tabellen, dette fordi identifisering av I-rammer er en relativt tung jobb. Dette er likevel ikke så farlig, siden dette er en oppgave som gjøres bare en gang for hver film.

Arbeidsmengde

Her ligger den største forskjellen mellom de to metodene. Programvaren må også her konstruere tidspunkt og posisjonsinformasjon, men før den kommer så langt må den altså finne I-rammene. Dette innebærer demultipleksing og traversering av transportstrømmen. Det vil være en betydelig oppgave å utvikle denne programvaren fra bunnen av. Et alternativ kunne ha vært å benyttet bibliotek/ferdig programvare som gjorde deler av denne oppgaven.

Valg

Vi har sett at disse to metodene som her er blitt vurdert har svært like egenskaper når det gjelder ytelse. Det er imidlertid to ting som taler imot «Identifisering av I-rammer»: dekoderen kan få problemer pga. manglende lesing av sekvenshoder, og arbeidsmengden for utvikling av programvare vil være mye høyere enn for den første metoden.

Ulempen med «Identifisering av aksesspunkt med RAI» er at det ikke er noe krav at sekvenshoder blir brukt for hver I-ramme. For Elvira II vil det nok være uaktuelt å kopiere og sette inn sekvenshoder som beskrevet i «Løsningsalternativer». Dette vil nok foreløpig bli en altfor tungvint måte å gjøre dette på, utviklingsarbeidet vil

bli for stort. Resultatet er at det stilles som et krav fra Elvira's side at filmer må bruke sekvenshoder og RAI for at tilfeldig aksess skal tilbys.

Sett på bakgrunn av at RAI etter all sannsynlighet vil bli brukt i de fleste filmene, vurderer vi likevel denne metoden, «Identifisering av aksesspunkt med RAI», til å være den beste metoden. Vi anbefaler derfor å bruke denne i Elvira II.

Spoling

I kapittelet «Løsningsalternativer» beskrev vi 6 hovedteknikker som hver kunne brukes for å oppnå spoling i et klient/tjener system. Her vil vi vurdere hver enkelt av disse etter kriteriene nevnt tidligere i kapittelet. Vurderingen vil gjøres med tanke på at transportstrømmen vil bli brukt.

Endre på tidsstempel

Tidsstemplene angir tidspunktene for når lyd og bilderammer skal dekodes og presenteres. Kort sagt går denne metoden ut på å manipulere tidstemplene på en slik måte at en spoleeffekt oppnås. Denne manipuleringen kan gjøres på forhånd (preprosessering) eller dynamisk.

Tjenestekvalitet

Denne metoden alene kan ikke brukes til å oppnå baklengs spoling, bare forlengs spoling vil kunne tilbys. Siden baklengs spoling er relativt viktig å ha med vil det nok være nødvendig å kombinere denne metoden med andre dersom den skal brukes. En fordel med denne metoden i forhold til andre er at brukeren her kan velge en hvilken som helst spolehastighet framover, dette kommer av at det bare er en faktor som skal ganges med tidsstemplene. (dette gjelder kun dersom prosesseringen gjøres dynamisk, ved preprosessering må selvsagt antall modifiserte kopier begrenses.)

Dersom systemet har ressurser nok, vil kvaliteten på framoverspolingen bli meget god (for ikke å si optimal). Her blir jo hvert bilde overført i sin helhet. Når det gjelder responstiden vil denne også være bra; alt som trengs å gjøre er å fortelle videotjeneren hvilken faktor som ønskes brukt, så vil den enten finne fram til ferdiglagde kopien eller forandre tidsstemplene dynamisk i strømmen. Ingen av disse oppgavene vil være tunge, så de vil ikke skape noen forsinkelse å snakke om.

Nøyaktigheten vil også kunne bli optimal; man kan fortsette spolingen nøyaktig der man slapp. Dette fordi man her ikke er avhengig av å starte på en I-ramme; selv om tidstemplene er forandret så vil ikke dette ha noe å si for dekodingen og bildeavhengighetene.

Ressursbruk

Her ligger denne metodens største svakheter: nettverks- og diskbelastningen. Mengden data som må overføres fra disk og over nettverket vil vokse proporsjonalt med hastigheten. For spoling hurtigere enn normal avspilling vil belastningen bli uakseptabel. For Elvira II bør ikke belastningen øke nevneverdig ved spoling i forhold til normal avspilling, derfor vil denne metoden være en dårlig løsning for Elvira II når det gjelder hurtig avspilling. Det betyr at spolemulighetene for denne metoden for Elvira II er redusert til sakte avspilling framover (slow motion). Dette vil kunne gjennomføres uten problemer, både nettverks- og diskbelastningen vil jo her senkes.

CPU/minne kapasiteten ved en dynamisk prosessering hos tjeneren vil ikke være noe problem. Tidsstemplene er lett identifiserbare i transportpakkene og alt som kreves er en forandring på noen få bytes. Man må også huske på at tidsstemplene forekommer relativt sjelden i pakkene.

Diskforbruk vil kun være en faktor ved bruk av preprosessering. Da vil vi få en betydelig økning i forbruket: for hver ønsket spolehastighet må man lagre en ny kopi av filmen. Siden ytelse og annet ressursbruk for den dynamiske varianten ellers vil være svært likt, vurderer vi derfor den dynamiske metoden til å være den beste.

Arbeidsmengde

Begge variantene av denne metoden vil kreve lite arbeid fra utviklerens side. Oppgaven er klart definert og relativt enkel. Spesielt gjelder dette for preprosessering, å gjøre det dynamisk vil antagelig kreve noe mer arbeid fordi utsendelseshastigheten da også må forandres dynamisk.

Send utvalgte bilder til klienten

Denne teknikken går ut på å velge ut enkelte bilder og kun sende disse over til klienten i den hensikt å oppnå hurtig spoling. Man kan hente ut bare I-rammer eller både I og P-rammer, og disse enkeltrammene bør settes sammen igjen til en systemstrøm. En indeksfil som sier hvor de utvalgte rammene ligger bør konstrueres på forhånd.

Tjenestekvalitet

Denne metoden vil være nyttig kun for hurtig spoling. Dette kommer selvfølgelig av at antallet bilder blir redusert. Man kan ved multipleksingen endre tidsstemplene slik at slowmotion oppstår, men da er det mye bedre å bruke metoden «Endring av tidstempler». Når det gjelder den visuelle kvaliteten på spolingen så antas den å bli god. Langørgen (1995) og Koteng (1996) har implementert tilsvarende metoder for henholdsvis MJPEG og MPEG1, med gode visuelle resultater. Kvalitetet vil naturligvis ikke kunne måle seg med kvaliteten for «Endring av tidsstempler». Nå fjerner vi jo en god del av bildene, så spolingen vil kunne framstå som noe hakkete ved høye hastigheter.

Nøyaktigheten vil ikke være noe problem, spolingen vil kunne startes med minimum en GOP-avstands nøyaktighet, og bedre dersom P-rammer også blir brukt. Responstiden vil derimot kunne bli noe treg, dette kommer av at tjeneren vil ha mye arbeid å gjøre (se nedenfor).

Ressursbruk

Ved å vise kun en del av bildene vil altså avspillingshastigheten øke (dersom den nye systemstrømmen er konstruert slik at hvert bilde vises like lenge som før). Fordelen i forhold til «Endring av tidsstempler» er at nå slipper man å overføre så mye data, disk- og nettverksbelastningen vil derfor ikke bli så høy. Men det vil fremdeles være et problem å ikke overstige datamengden for normal avspilling. Dette kommer av at de rammene vi fjerner er de som tar minst plass, nemlig B-rammene (og evt. P-rammene). I-rammene, som vi må ha med, tar mye større plass enn de andre og datamengden vil derfor ikke reduseres i samme takt som vi fjerner bilder. Vi ser derfor at også denne metoden blir et problem for Elvira II. Den kan likevel være aktuell for tjenere som kan tillate økt belastning ved spoling.

Det som likevel kanskje er det største problemet med denne metoden er kravene som stilles til CPU/minne. Bildene må demultiplekses fra strømmen, og settes sammen igjen til en gyldig systemstrøm (Dekoderen bør ta imot samme format ved spoling. Dersom man kunne sende en elementærstrøm vil oppgaven reduseres.) Dette er ingen oppgave som med enkelhet kan gjøres i sanntid. Med dagens maskinvare er det tvilsomt om dette kan realiseres.

Arbeidsmengde

Å lage en sanntids demultiplekser og multiplekser vil ikke være noe lite prosjekt. Selv om bibliotek og annen software kunne benyttes hadde det nok vært urealistisk å gjøre dette i denne oppgaven. Dersom man bare bruker I-rammer kan demultipleksingen gjøres noe lettere (ved bruk av RAI), men det vil allikevel bli mye arbeid.

La klienten ta seg av spoling

Prinsippet her går ut på at det er klienten som utfører spolingen ved å be om å få spilt av deler av filmen. Klienten vil ha tilgang til tabeller som sier hvilke data den skal be om gitt en spolehastighet. Tjeneren tilbyr bare tilfeldig aksess.

Tjenestekvalitet

Hurtig spoling både forlengs og baklengs vil være mulig å realisere ved denne teknikken. Ulempen er imidlertid at spolekvaliteten blir heller dårlig. Først og fremst blir spolingen ujevn, den vil bestå av både avspilling og hopp. Dessuten blir den baklengse spolingen heller dårlig, den vil bestå av forlengs avspilling med hopp baklengs i filmen, noe som vil gi en «hakkete» effekt.

For å oppnå en høy spolehastighet må man enten hoppe lange steg eller spille av lite om gangen. Begge deler har sine ulemper: lange hopp vil medføre store sprang i bildeinformasjonen og svært ujevn spoling, mens lite avspilling om gangen vil gi mye kommunikasjon og forsinkelser.

Responstiden for igangsettelse av spoling vil være nesten identisk med responstiden for vanlig tilfeldig aksess. Se vurderingen av dette i avsnittet «Identifisering av aksesspunkt med RAI». Det eneste som kommer i tillegg vil være oppslag i en lokal indeksfil for spoling hos klienten, men dette vil ikke ta noe tid å snakke om dersom den ligger i minne (noe den bør gjøre, hvis den må leses fra disk skapes en liten, men unødvendig forsinkelse).

Nøyaktigheten kan bli noe dårligere enn for de tidligere nevnte metodene. Dersom spolingen innebærer lange hopp vil jo den lokale indeksfilen ikke inneholde så mange steder å hoppe til. Noe stort problem bør det likevel ikke bli da hoppene uansett ikke bør være så veldig lange.

Ressursbruk

Nettverkbelastningen for et videosystem vil nesten bare bestå av overføring av videoen; annen kommunikasjon vil bidra lite til datamengden. Med denne teknikken vil videomengden som blir overført per tidsenhet ikke bli større enn for normal avspilling; den vil bli omtrent identisk. Kommunikasjonen vil øke betraktelig, men dette vil allikevel bli en såpass liten del av den totale nettverkbelastningen at vi anser dette til å ikke være noe problem. Dette gjelder kun når tjeneren har direkte adgang til strømmen og kan sende ut hvilken del av strømmen den ønsker. Elvira II vil bruke *containere* på 8kb som lagrings og oversendelsesenhet og dette vanskeliggjør denne metoden i stor grad. Når klienten ber om spesifikke rammer så vil det sannsynligvis bli oversendt en god del overflødig informasjon i mange containere, og nettverksbelastningen vil øke. Tilsvarende vil gjelde for diskbelastningen.

Tjeneren tilbyr bare tilfeldig aksess så CPU/minneforbruket vil være ubetydelig. (Se avsnittet om tilfeldig aksess.)

Diskforbruket vil ikke øke i betydelig grad, det eneste som må lagres er klientens indekser. Diskbelastningen skal heller ikke bli noe problem, selv med svært korte avspillinger vil ikke tiden mellom aksessene være kortere enn aksesstiden for en harddisk.

Arbeidsmengde

Å konstruere indeksfiler for tjeneren vil ikke være vanskelig, de vil bare være et subsett av filen tjeneren har for tilfeldige aksesser. Tjeneren må også ha litt logikk som styrer kommunikasjon og oversendelse av tilfeldige aksesser, men heller ikke dette bør være noen stor implementasjonsoppgave.

La klienten få en kopi av originalstrømmen

Dette innebærer at klienten får en lokal kopi av strømmen, slik at han kan spole i denne uten å belaste tjeneren eller nettverket. Oversendelsen kan skje enten på forhånd eller under avspilling (kopiering til harddisk).

Tjenestekvalitet

Ved en variant der klienten får oversendt hele eller deler av filmen før avspillingen starter vil dette føre til en lang ventetid før klienten i det hele tatt får startet filmen. Derfor er det nok mer naturlig å ta imot filmen samtidig som den spilles av. Ved hjelp av en *dynamisk* metode kan det nå spoles i film som allerede er sett av brukeren. Vi ser at spolemulighetene vil bli en god del begrenset. Spoling vil ikke fungere i usett materiale, noe som rammer foroverspolingen mest, den blir jo ofte brukt til å lete framover etter interessante ting i filmen.

Nåt det gjelder ting som nøyaktighet, responstid og spolekvalitet så vil dette avhenge av den lokale dynamiske spolemetoden som blir brukt. Likevel kan det sies at det her er muligheter for gode resultater siden vi ikke har et begrensende nettverk å tenke på.

Ressursbruk

Ved denne metoden får vi ingen nettverksbelastning eller diskbelastning (hos tjeneren) under spoling. Men for den første varianten får vi problemer: for at det skal være noe poeng i å overføre hele eller deler av filmen på forhånd bør overføringshastigheten være høyere enn for normal avspilling. Dette vil igjen føre til store problemer for nettverket, vi kan få høye belastningstopper. Vi vurderer derfor denne varianten til kun å være aktuell i systemer der brukerene kan «bestille» en film og få denne oversendt for eksempel innen et visst tidspunkt. Når det gjelder den andre varianten, oversendelse ved avspilling, vil ikke dette være noe problem. For denne varianten blir derfor disk- og nettverksbelastningen totalt mindre enn for de fleste andre metoder.

Tjeneren slipper ved denne metoden å lagre noen ekstra filer i forbindelse med spolingen. Metoden setter imidlertid krav om at hver klient må ha en relativ stor harddiskkapasitet. Den totale harddiskkostnaden for et slikt system vil derfor kunne bli høy.

CPU- og minneproblemer hos tjeneren vil ikke denne metoden forårsake. Siden hver klientmaskin tar seg av spolingen vil tjeneren bli avlastet. Klientmaskinen kan få det verre, i tillegg til å håndtere brukerapplikasjonen skal den skal styre en dynamisk spoling. En kraftig maskin kan bli nødvendig.

Arbeidsmengde

Tjeneren vil med denne metoden naturlig nok bli enklere og lite arbeid trengs å gjøres her. Arbeidsmengden for utvikling av programvare på klientsiden vil kunne bli stor, men vil avhenge av spoleteknikk. Kontrollmekanismer for skifte mellom avspilling og spoling må også implementeres.

Spolefiler generert fra systemstrøm

Dette går ut på å dekode en video for etterpå å kode denne på nytt. Enkeltbilder kan i mellomtiden fjernes og en spolehastighet oppnås. Kodekvaliteten kan reduseres for å minske datamengden. Baklengs spoling kan oppnås ved å snu bilderekkefølgen før koding.

Tjenestekvalitet

En stor fordel med denne metoden er at vi med denne metoden alene kan generere spolefiler for alle slags kategorier av spoling: forlengs og baklengs, hurtig og sakte (selv om andre metoder kan være bedre å bruke for sakte avspilling). Vi får også en større valgfrihet når det gjelder spolehastigheter: Mens vi i andre metoder måtte være selektive når det gjaldt utplukking av enkeltbilder kan vi her velge hvilke bilder vi vil plukke ut. Dette fordi alle bildene dekodes og ikke lenger vil ha noen avhengigheter seg imellom.

Spolekvaliteten kan i utgangspunktet bli like god som for «Send utvalgte bilder til klienten». Vi vil imidlertid kunne redusere noe av denne kvaliteten i form av dårligere kvalitet på hvert enkeltbilde. Dette gjøres i bytte med noe som er viktigere: mindre datamengder. Denne metoden er den eneste som lar oss redusere kvaliteten på hvert enkeltbilde.

Responstiden vil også være akseptabel. Tjeneren må ved spoling begynne å spille av fra en annen fil og dette vil selvsagt føre til en noe dårligere responstid enn for metoder der samme fil kan benyttes. Tjeneren må også slå opp i en overgangstabell som forteller hvor i spolefilen den skal begynne avspillingen. Denne tabellen kan ligge i minne og dette vil derfor ikke ta vesentlig tid. Størrelsen på en slik overgangstabell vil nok ikke være for stor for minne dersom den kun inneholder få spolehastigheter, men man bør huske på at størrelsen vil øke proporsjonalt med antall spolehastigheter som tilbys.

Nøyaktigheten for overgangen til spolefiler vil her begrenses av antall aksesspunkt i spolefilen (dersom det er snakk om hurtig spoling). Dette kommer av at avstanden mellom to aksesspunkt i spolefilen vil «dekke» et større tidsområde i spolefilen siden det er hurtig spoling. Dermed vil denne metoden kunne øke unøyaktigheten noe i

forhold til de andre metodene. Likevel antas det at det blir akseptable resultater, dersom man har to aksesspunkt i sekundet kan man likevel havne maksimalt et halvt sekunds spoling unna den «riktige» posisjonen (gitt en korrekt overgangstabell).

Ressursbruk

Dette er denne metodens sterkeste side. Ved å redusere kvaliteten på enkeltbilder samtidig som vi bruker kun enkeltbilder kan vi sørge for at både disk- og nettverksbelastningen aldri overstiger en gitt verdi uansett spolehastighet. For Elvira II som ikke vil at spolebelastningen skal overskride avspillingsbelastningen er dette en verdifull egenskap.

Diskforbruket vil naturlig nok være en faktor. Dersom en spolefil skal påføre samme nettverkbelastning som den vanlige fila, vil størrelsen på fila måtte være tilsvarende mindre. For eksempel vil en spolefil på 2 ganger normal hastighet måtte være omtrent halvparten så stor som normalfila. Størrelsen på spolefila i forhold til normalfila vil kunne estimeres etter følgende formel: **1/S** (S = Spolehastighet). En videotjener som vil tilby følgende spolehastigheter i hver retning: 2x og 5x, vil trenge omtrent følgende diskkapasitet for spolefilene: 2*(0.5+0.2) = 1.4 ganger normalfila. (Formelen 1/S vil også gjelde for sakte avspilling dersom dette blir gjort ved å *duplisere* bildene før koding. Dette vil selvsagt føre til høyt diskforbruk, andre metoder bør derfor benyttes for sakte avspilling.)

CPU/Minneforbruket vil også her være akseptabelt, mesteparten av arbeidet er gjort på forhånd da spolefilene ble generert. Som nevnt så bør tjeneren ha nok minnekapasistet til å lagre hele overgangstabellen for de ulike spolefilene, slik at best mulig responstid kan oppnås. (Det må foretaes noe søking i tabellen for hvert oppslag.)

Arbeidsmengde

Ferdig software kan og må brukes for uthenting av enkeltbilder (demultipleksing og dekoding), samt oppbygging av ny systemstrøm (koding og multipleksing). Noe programvare bør lages: den bør sy hele prosessen sammen samt utføre manipuleringen av bildene slik at den ønskede spoleeffekten oppnås. Konstruksjon av overgangstabellene må også implementeres. Dette burde ikke være noen uoverkommelig oppgave.

Produser filer for spoling direkte

Dersom man er i den situasjonen at man produserer MPEG-2 filmer, kan man konstruere spolefiler direkte fra det samme bildematerialet. Med det rette utstyret vil dette være en relativ enkel oppgave.

Denne metoden vil resultatmessig tilsvare den forrige, «Spolefiler generert fra systemstrøm». Derfor vil tjenestekvaliteten og ressursbruk vurderes på samme måte som i avsnittet over. Forskjellen ligger i måten å lage spolefilene på, her dekoder vi ikke strømmen først.

Problemet med denne metoden er ganske enkelt at man må produsere filmmaterialet selv (evt. ha tilgang til mange enkeltbilder). For de videotjenere som allikevel skal gjøre dette vil denne metoden naturligvis være best. Elvira II vil bruke ferdige genererte strømmer som utgangspunkt, følgelig vil metoden «Spolefiler generert fra systemstrøm» måtte brukes istedenfor denne.

Oppsummering og valg

Vi har nå vurdert alle de nevnte metodene. Tabellen under oppsummerer hvor bra metodene er i forhold til hvert enkelt kriterium. Figuren bruker plusstegn og minustegn for å angi om metodene er positive eller negative i forhold til kriteriene. Skalaen som er brukt går fra 2 plusstegn (++) som er best, til 2 minustegn (--)som er dårligst. En 0 vil si at denne metoden er noenlunde nøytral for kriteriet.

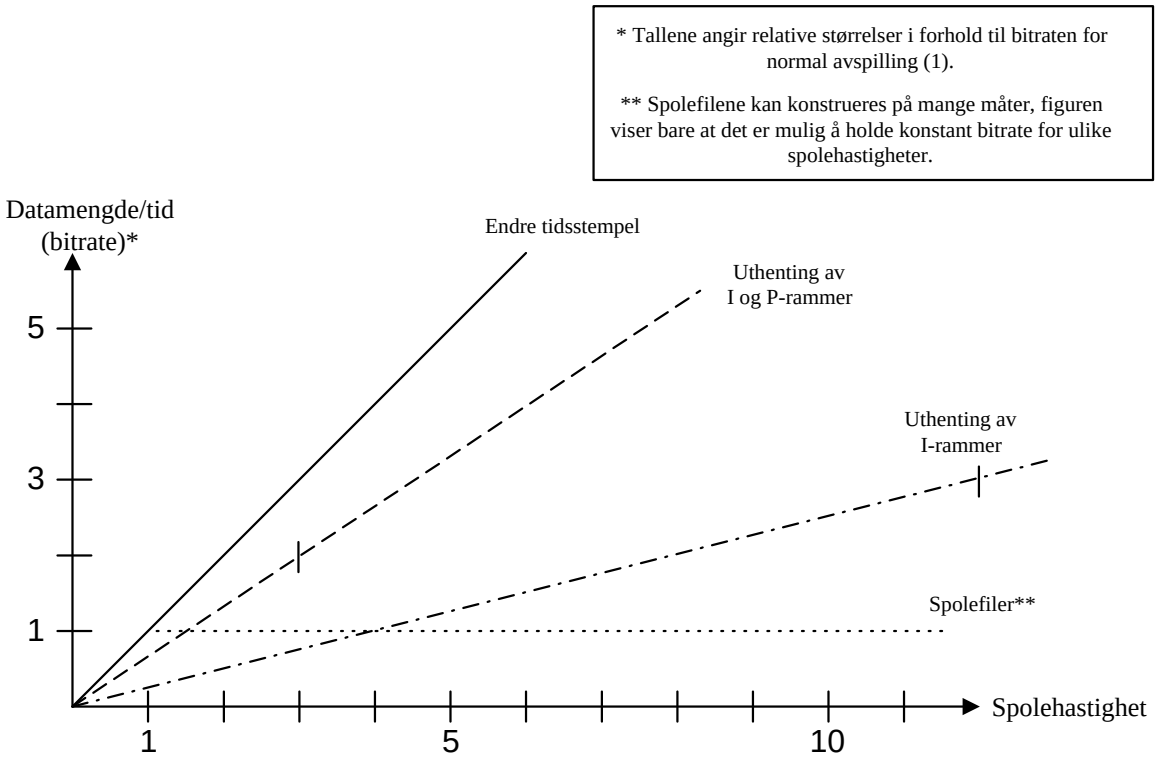
	TIDSSTEMPEL (DYNAMISK)	UTHENTING (I / I OG P)	KLIENTEN SPOLER	KOPI TIL KLIENTEN	SPOLEFILER
Spolekvalitet	++	0	-	+	0
Spolemuligheter	-	+	+	--	++
Responstid	+	-	+	+	0
Nøyaktighet	++	+	0	+	0

Nett-belastning	--	-	+	-	++
Diskbelastning	--	-	+	-	++
Diskforbruk	++	+	+	--	0
CPU/Minne	+	--	+	0	+
Utviklingsarbeid	+	--	0	-	+

Oppsummering av egenskaper

Vi har sett at datamengden som må overføres er noe av det viktigste metodene vurderes etter. Dette fordi det gjenspeiles i både disk- og nettverksbelastning, og disse er begge kritiske ressurser i Elvira II. I Figur 34 ser vi derfor litt nærmere på denne egenskapen; her oppsummerer vi «Overført datamengde» for noen av de aktuelle metodene.

Grafen er basert på enkle estimater basert på følgende data: en strøm med I-ramme avstand på 12, og en P-ramme avstand på 3. Størrelsesberegninger er gjort med utgangspunkt i en relativ størrelsesforskjell mellom I-, P-, og B-rammer på henholdsvis 6:3:1.



Figur 34 Bitrate-estimater

Estimatene er foretatt slik:

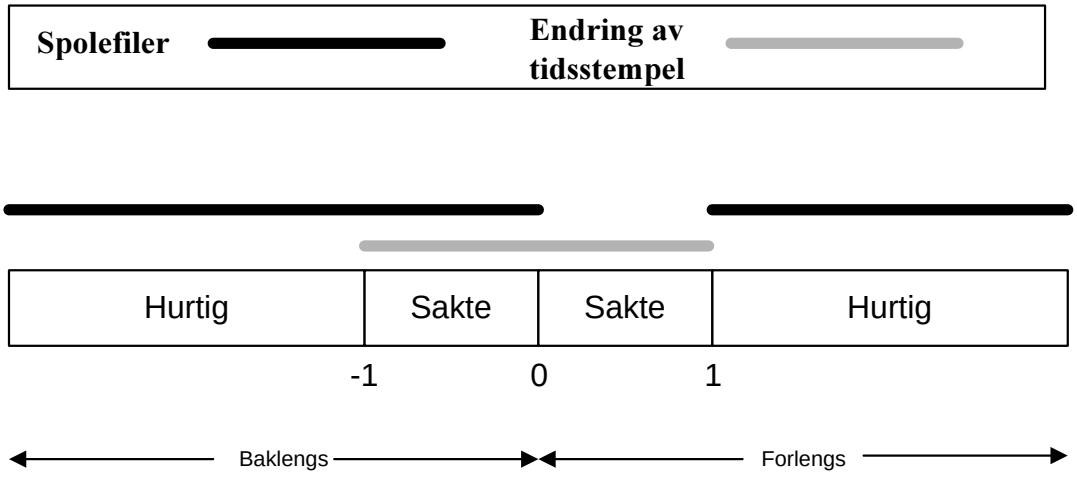
Endre tidstempel: Her øker datamengden proporsjonalt med hastigheten.
I-rammer: Uthenting av hver I-ramme gir en spolehastighet på 12X. (Merket på figuren). I en GOP har vi 1 I-ramme, 3 P-rammer og 8 B-rammer. For hver GOP bruker vi ca. følgende andel datamengde (gitt størrelsesproporsjonene 6:3:1) : (6*1 / 6*1+3*3+1*8) ≈ ¼. Økning i bitrate for 12X blir da (12 * ¼) = 3.
I-og P-rammer: Uthenting av alle I- og P-rammene gir en spolehastighet på 3X. (Merket på figuren). Tilsvarende beregning for andel datamengde blir: (6*1+3*3 / 6*1+3*3+1*8) ≈ 2/3. Økning i bitrate for 3X blir da (3*2/3) = 2.

For å oppnå de andre hastighetene for uthenting av I-rammer / I- og P-rammer (som vist på figuren), må dette kombineres med å endre tidsstempler. De lineære grafene for disse to variantene viser altså denne situasjonen.

Vi ser at «Spolefiler generert fra systemstrøm» er den eneste metoden som klarer å holde datamengden på et konstant nivå ved gradvis hurtigere spoling. I tillegg ser vi fra tabellen «Oppsummering av egenskaper» at denne metoden også klarer seg bra for de andre kriteriene. På bakgrunn av dette velger vi derfor denne metoden til å være den beste til å utføre hurtig spoling i begge retninger for Elvira II. Når det gjelder sakte avspilling framover så velger vi å bruke metoden «Endre tidsstempel», den dynamiske versjonen. Denne metoden har gode egenskaper, og siden det er sakte avspilling slipper vi å operere med store datamengder. For sakte avspilling baklengs anbefaler vi å bruke en kombinasjon av disse to metodene; først genereres det en baklengs fil (for normal hastighet) ved hjelp av spolefilkonstruksjonen, deretter endres tidsstemplene dynamisk for å oppnå sakte avspilling i denne filen.

Disse valgene vil føre til at vi aldri trenger å overføre vesentlig mer data enn for normal avspilling. Det antas også at implementasjonsarbeidet ikke vil bli altfor stort, samtidig som gode resultater vil kunne oppnås.

Figur 35 oppsummerer valget som her er gjort, den viser hvilke metoder som dekker de forskjellige spolemulighetene.



Figur 35 Valg av spolemetoder

Konstruksjon

Dette kapittelet tar sikte på å gi en detaljert beskrivelse av hvordan de valgte løsningene kan realiseres i Elvira II. Som beskrevet i kapittelet «Elvira II», bruker Elvira II et spesielt format for lagring av videofiler. Det første som må avklares er måten en transportstrøm skal legges inn i dette formatet på. Spesielt viktig er det å overføre tidsbegrepet i en MPEG-2 film over til Elvira II formatet slik at videotjeneren kan foreta korrekt tidssynkronisering ved avspilling av filmen.

Deretter gis det en beskrivelse av hvordan spolefiler kan konstrueres. Hvert steg i hele prosessen blir beskrevet relativt detaljert. Det vil bli fokusert på hva som kreves av programmet/prosedyren på hvert steg for å komme fram til et tilfredsstillende mellomresultat. Hva som er et tilfredsstillende mellomresultat vil selvsagt også beskrives.

Vi vil deretter se på hvordan tilfeldig aksess kan implementeres, både i vanlige filer og i spolefiler. Det er viktig at alle filer tillater dette slik at vi har mulighet til å oppnå gode overganger mellom avspillinger av forskjellige videofiler. For at Elvira II skal finne rett sted i filene å hoppe til er det også viktig at *tidsinformasjon* knyttes til hvert aksesspunkt. Vi vil ved hjelp av et eksempel forklare hvordan Elvira II benytter denne informasjonen til å oppnå korrekt synkronisering i forbindelse med skifte mellom ulike avspillingshastigheter og tilfeldige aksesser.

Til slutt ser vi på hvordan sakte avspilling kan innføres i Elvira II.

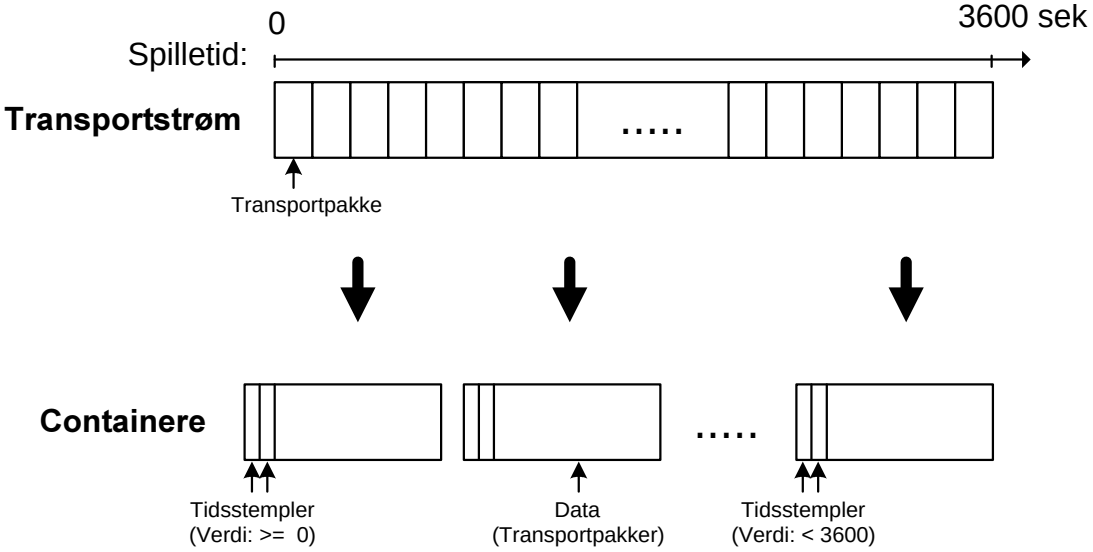
Fra Transportstrøm til Elvira II

I dette avsnittet ser vi på hvordan en transportstrøm kan overføres til Elvira II formatet. Vi vil i første omgang konsentrere oss om hvordan en enkelt avspillingsfil vil behandles. Senere i dette kapittelet vil vi utvide prosessen ved å innføre spolefiler.

Elvira II vil bruke containere som oversendelsesformat. Oppgaven blir derfor først å pakke transportstrømmen inn i disse containerene. Som beskrevet i kapittelet Elvira II, skal det i headeren av containerene ligge to tidsstempler som sier når denne containeren skal sendes ut fra tjeneren. Verdien «Movietime» angir hvilket tidspunkt i filmen dataene i containeren representerer. «Showtime» sier når dekoderen skal ta imot dataene i containeren. For en vanlig avspillingsfil vil disse verdiene være de samme. Formatet på verdiene vil være i millisekunder.

Resten av containeren vil være data fra transportstrømmen. Transportpakkene vil derfor måtte kopieres inn i containerene. Hver container bør inneholde et helt antall transportpakker. Dette fordi containere kan forsvinne under transporten og det er viktig at dekoderen mottar et helt antall transportpakker. Oppførselen kan være uforutsigbar dersom den får bare deler av en pakke. Størrelsen på en container kan være variabel, men vil ha en maksimal størrelse på 8kB.

Hvordan skal så tidsinformasjonen hentes ut fra filmen og settes inn i alle containerene som skal lages? Figur 36 viser utgangspunktet for dette problemet.



Figur 36 Konstruksjon av containere

Eksempelet over viser en transportstrøm med en spilletid på en time. Tidsstemplene i containerene må gis verdier som på best mulig måte gjenspeiler dette. Det betyr at verdiene må variere fra tilnærmet 0 for den første containeren, til nesten 3600 sekunder for den siste containeren. Merk at den siste verdien bør være litt mindre enn den totale tiden siden verdien også representerer de første transportpakkene i containeren.

Tidsinformasjon i strømmen

For å bestemme hvordan vi skal sette tidsverdiene må vi først vite hva slags tidsinformasjon som finnes innebygd i strømmen. Som nærmere beskrevet i kapittelet «MPEG-2», så finnes det tre tidsstempel som må være med i transportstrømmen: PCR, DTS, og PTS.

Her følger en oppsummering av betydningen av disse:

- **PCR** (Program Clock Reference): Definerer tidspunktet for når den siste byten i dette tidsfeltet skal komme fram til dekoderen. Består av 33 bit (base) + 9 bit (extension). Et program (som kan inneholde flere elementærstrømmer) har en løpende PCR-verdi.
- **DTS** (Decoding Time Stamp): Definerer tidspunktet for når påfølgende aksessenhet (ramme) skal dekodes i dekoderen. Består av 33 bit. Hver elementærstrøm har en løpende DTS.
- **PTS** (Presentation Time Stamp): Definerer tidspunktet for når påfølgende aksessenhet (ramme) skal presenteres av dekoderen. Består av 33 bit. Hver elementærstrøm har en løpende PTS.

MPEG-2 standarden krever ikke at noen av disse verdiene i en strøm starter med 0 - den kan starte med en hvilken som helst verdi. Elvira II har derfor to valg ved bruk av disse, den kan kreve at alle strømmer skal starte fra 0 (noe som oftest vil være tilfelle), eller den kan støtte en situasjon der tidsstemplene starter på et tilfeldig tall. Å støtte dette vil være relativt enkelt. I det følgende vil vi likevel gå ut ifra at verdiene starter med 0.

I tillegg finnes det i elementærstrømmen en tidskode som kalles **time_code**. Denne måler tiden i timer, minutter og sekunder og blir gjentatt for hver GOP (Group of Pictures). Denne tidskoden baserer seg på et IEC definert format for tidskoder for videospillere, og kan brukes av dekodingsapplikasjonen dersom den ønsker det (men den spiller ingen rolle i dekodingsprosessen). Vi velger å ikke bruke denne koden, først og fremst fordi den ikke er nøyaktig nok (minste enhet er ett sekund), men også fordi den ligger vanskeligere tilgjengelig (vi må traversere ned til elementærstrømmen) og fordi den blir gjentatt så sjelden.

Da står valget mellom de tre tidsstemplene i transportstrømmen. Som nevnt så skal tidsstemplene i containerene inneholde en verdi som skal brukes av tjeneren som referanse for når den skal sende ut containeren. Vi ser at

PCR-verdien i stor grad har samme funksjon i den opprinnelige strømmen. Derfor virker bruk av PCR lovende. Dessuten så vil transportstrømmene som skal brukes i Elvira II bare inneholde ett program. Derfor vil PCR-verdien alene si hvor i filmen vi tidsmessig befinner oss. Ved bruk av DTS eller PTS måtte vi ha brukt verdier for bare en av elementærstrømmene (fortrinnsvis video) for å få sammenhengende verdier.

Et annet problem med bruk av PTS er at denne sier når en ramme skal presenteres. Siden B-rammene ikke dekodes i samme rekkefølge som de skal vises (i forhold til I- og P-rammer), så vil dette føre til at PTS-verdiene ikke er garantert å stige uniformt. Dette avhenger av om PTS-verdiene blir satt for både P- og B-rammer.

Et siste punkt som taler for PCR er at den forekommer oftere enn de to andre; den skal forekomme minst hvert 0.1 sekund. PTS skal forekomme minst hvert 0.7 sekund, mens DTS ikke trenger å forekomme i det hele tatt (dersom PTS og DTS er lik for en ramme, noe de ofte kan være, så vil ikke DTS settes inn i strømmen for denne rammen.)

Vi velger av disse årsakene å bruke verdien som finnes i PCR-feltet som utgangspunkt for verdien som skal inn i tidsfeltene i containerene.

Uthenting av PCR

PCR-verdien ligger i tilpasningsfeltet (adaptation field). Dette feltet vil, når det brukes, alltid ligge i begynnelsen av en transportpakke. Et flagg i begynnelsen av feltet sier om det eksisterer en PCR-verdi i dette feltet. Det vil derfor ikke vil by på noen vanskeligheter å lokalisere og å hente ut en PCR-verdi.

PCR-verdien er som nevnt kodet i to deler; et base felt (33 bit) og et tilleggsfelt (extension field, 9 bit). Som beskrevet i kapittelet «MPEG-2», sørger dette tilleggsfeltet for å gi verdien en større presisjon. Når tilleggsfeltet har telt opp til 300 vil basefeltet økes med 1, vi får altså en økning i presisjon på 300 ganger. Denne økningen er ikke nødvendig til vårt formål, som vist nedenfor har vi allerede en god nok presisjon med basefeltet. Det velges derfor å ignorere tilleggsfeltet, vi henter bare ut basefeltet.

Verdien som ligger i basefeltet er kodet på en spesiell måte, ligningen for dette er gitt i kapittelet «MPEG-2». Siden tidsstemplene i containerene skal være på formatet millisekunder, må vi foreta en omregning. Med utgangspunkt i ligningen for koding av PCR blir denne omregningen som følger (ved å sette units_per_second til 1000 får vi millisekunder som enhet):

$$t = \frac{PCR\_base \times 300 \times units\_per\_second}{System\_clock\_frequency} = \frac{PCR\_base \times 300 \times 1000}{27000000} = PCR\_base \times \frac{1}{90}$$

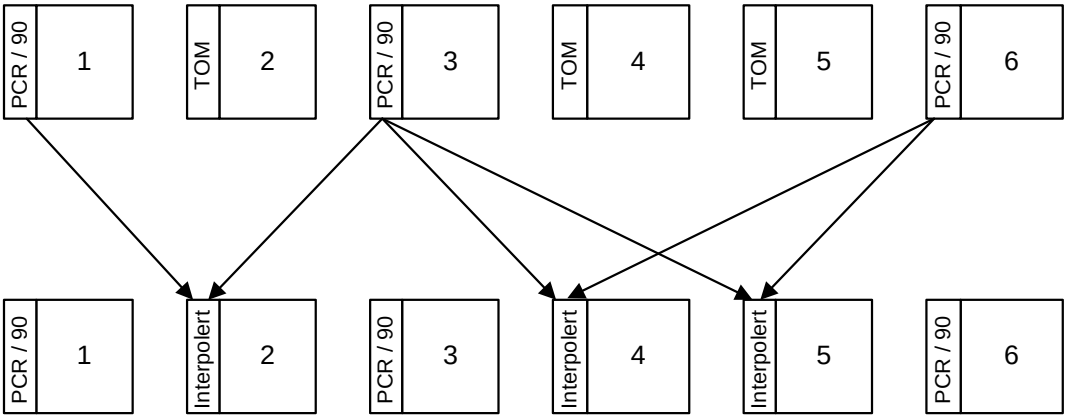
Vi ser at ved å dele PCR_base-verdien på 90 vil vi få verdien i millisekunder. Vi kan nå kontrollere at basefeltet gir oss god nok presisjon: Presisjonen blir 1/90 = 0.011 millisekunder. Dette er i høyeste grad bra nok til å styre utsendelseshastigheten for Elvira II.

Det er naturlig at tidsstempelet for en container hentes fra den første PCR-verdien som finnes i dataene i containeren. For en optimal nøyaktighet burde det tas hensyn til hvor langt ut i containeren denne finnes for dermed å justere verdien på tidsstempelet etter dette. Dette vurderes likevel som unødvendig idet Elvira II ikke krever den grad av nøyaktighet. Containerene er uansett såpass små at dette ikke gjør noe, feilmarginene blir relativt små.

Interpolering

Som nevnt vil en PCR-verdi garantert befinne seg i strømmen hvert 0.1 sekund. Samtidig skal en container i Elvira II ha en maksimal størrelse på 8kB. Dette gjør at vi ikke kan være sikre på at en container alltid vil inneholde en PCR-verdi. Dette er avhengig av bitraten til strømmen, ved lav bitrate vil datamengden per 0.1 sekund kunne være mindre enn en container, men vi har altså ingen garanti for dette ved høyere bitrater. Dette fører til at flere av containerene vil kunne bli stående uten verdier.

For å bøte på dette må de manglende verdiene *interpoleres* ut ifra de eksisterende verdiene. Dette vises i et eksempel i Figur 37. (Containerene skal ha to identiske tidsstempel, figuren viser for enkelhets skyld bare ett).



Figur 37 Eksempel på interpolering

Interpoleringen består som vi ser av å beregne verdien utifra verdiene til de to nærmeste containerene (foran og bak). Dersom man må interpolere for den første eller siste containeren vil man måtte beregne utifra to foranliggende / to etterliggende containere.

Interpoleringen for container nr. N gjøres lineært etter følgende formel:

$$Verdi(N) = \frac{PCR(N+i) - PCR(N-j)}{i+j} \times N$$

Her er i avstanden fram til foranliggende tidsstempel (målt i antall containere) og j avstanden til etterliggende container med tidsstempel.

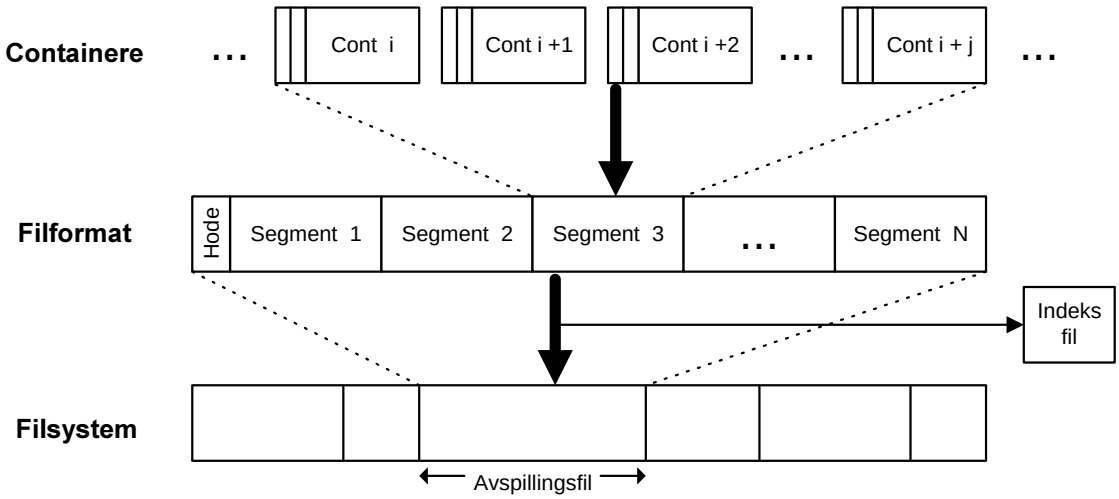
For å utføre denne interpoleringen må vi kjenne til verdien for containere som kommer senere, vi bør derfor gjøre denne operasjonen i to steg; først settes de kjente tidsverdiene, deretter interpoleres de ukjente verdiene.

Elvira II format, filsystem og indeksfiler

Etter at containerene er generert må disse legges inn i Elvira II formatet. Dette formatet er nærmere beskrevet i kapittel «Elvira II», og i vedlegg 3. Containerene pakkes inn i *segmenter* som har en fast størrelse. I begynnelsen av hvert segment skal det ligge en containerindeks som forteller hvor mange containere som finnes i segmentet, hvor i segmentet de begynner og hvor store hver enkelt container er. I begynnelsen av filen skal det ligge et videohode som inneholder informasjon om det totale antall segmenter og containere som er blitt generert, samt annen informasjon som filmnavn osv.

Elvira II definerer et eget flatt filsystem der filmene blir lagret, se kapittel «Elvira II». Når en film blir generert på Elvira II formatet skal den samtidig legges inn i dette filsystemet. For å senere, under avspilling, finne fram til de ulike segmentene som utgjør en film, må videotjeneren vite hvor disse segmentene ligger. Dette gjøres ved at det for hver film lages en *indeksfil.* Denne filen forteller hvor de ulike segmentene for en film ligger i filsystemet, samt en *tilhørende tidsverdi* for dette segmentet. Denne tidsverdien gjør at tjeneren kan holde oversikt over rekkefølge og tidspunkt for når segmentene skal leses inn. Verdien for segmentet skal hentes fra tidsstempelet til den *første* containeren i segmentet.

Figur 38 viser hele prosessen fra containere til en ElviraII-fil som legges inn i filsystemet.



Figur 38 Fra containere til Elvira II filsystem

Indeksfilen må lages under genereringen av elviraformatet, derfor må man kontinuerlig under prosesseringen av segmenter holde oversikt over hvor i filsystemet man legger segmentene. Informasjonen som skal ligge i indeksfilen er :

- *Hastighet*
- *Kodeformat*
- *Rammerate*
- *Segmentstørrelse*
- Antall segmenter
- Antall containere
- For hver segment
    - Tidsstempel
    - Diskidentifikasjon
    - Blokknummer

Dataene skrevet i *kursiv* vil kunne settes umiddelbart, de andre må oppdateres underveis i prosessen.

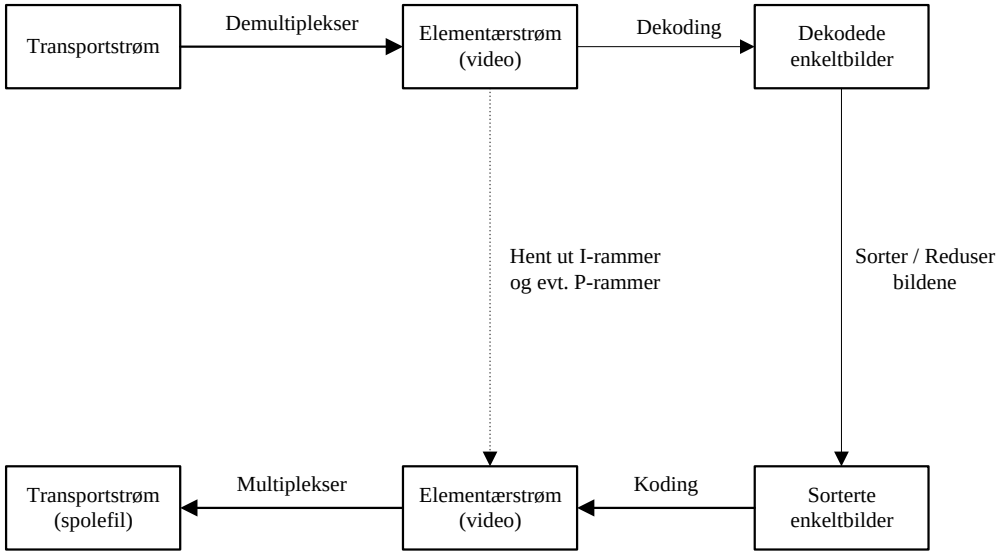
I tillegg til å inneholde segmentinformasjon om normalfila (vanlig avspilling) skal denne indeksfila også inneholde segmentinformasjon for alle instanser av spolefiler for denne filmen. Vi kommer tilbake til hvordan disse delene av indeksfila kan genereres og settes sammen senere i konstruksjonen. Se forøvrig vedlegg 4 for en detaljert beskrivelse av hvordan indeksfilen skal se ut.

Spolefilkonstruksjon

Vi har til nå sett på hvordan en enkelt transportstrøm kan legges inn i Elvira II. Som beskrevet i kapittelet «Løsningsvurdering og valg», har vi valgt å konstruere spolefiler for å oppnå spoling. Før vi går videre med å beskrive hvordan disse kan integreres i Elvira II, vil vi i dette avsnittet beskrive nærmere hvordan de skal konstrueres.

For å konstruere en spolefil med en transportstrøm som utgangspunkt og sluttprodukt må vi gjennom mange steg. Disse stegene er illustrert i Figur 39 nedenfor. Poenget med hele prosessen er at vi får tilgang til enkeltbildene

som et mellomformat. Disse enkeltbildene kan vi behandle før koding, dvs. vi kan plukke ut en viss andel av disse bildene og eventuelt snu rekkefølgen på disse. Spolingseffekten vil avhenge av denne behandlingen. Som vist på figuren er det mulig å unngå dekodingen / kodingen ved å foreta en «snarvei». Dette består av å hente ut I-rammer og eventuelt P-rammer direkte fra elementærstrømmen. I det følgende beskrives disse ulike stegene nærmere.

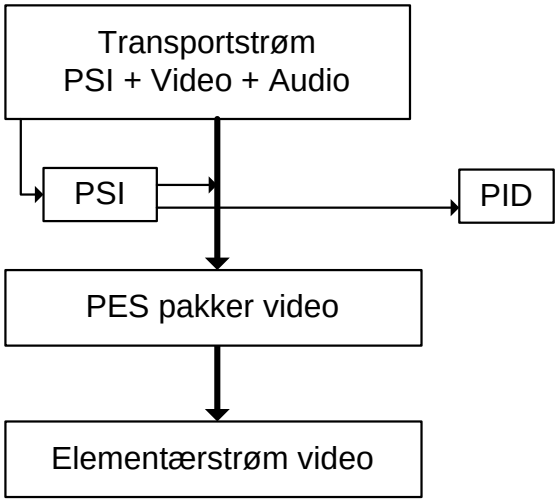


Figur 39 Spolefilkonstruksjon (overordnet beskrivelse)

Beskrivelse av stegene i spolefilkonstruksjonen:

Demultipleksing
En transportstrøm kan inneholde mange programmer som hver igjen kan inneholde mange elementærstrømmer. En strøm i Elvira II vil imidlertid bare inneholde ett program med to elementærstrømmer; video og audio. I forbindelse med spolefilgenereringen er vi som nevnt ikke interessert i lyd, slik at vi vil bare bruke elementærstrømmen for *video.* Elementærstrømmene er som nevnt pakket inn i PES-pakker, og disse er igjen pakket inn i transportpakker. Vi må altså først pakke ut PES-pakkene for video, deretter må elementærstrømmen pakkes ut av disse. Et program som utfører denne oppgaven kalles en demultiplekser.

For å kunne pakke ut en bestemt elementærstrøm vil demultiplekseren bruke informasjon som den finner i PSI pakker (Program specific information). Disse pakkene består av tabeller som forteller hvor mange elementærstrømmer som finnes og hvilken PID (Packet Identifier) hver strøm har fått tildelt. PID vil brukes i hodet til hver transportpakke som inneholder PES-pakker av denne strømmen. Etter å ha lest PSI-pakker kan dermed demultiplekseren enkelt kjenne igjen PES-pakkene som inneholder den ønskede elementærstrømmen. Dette er illustrert i Figur 40.



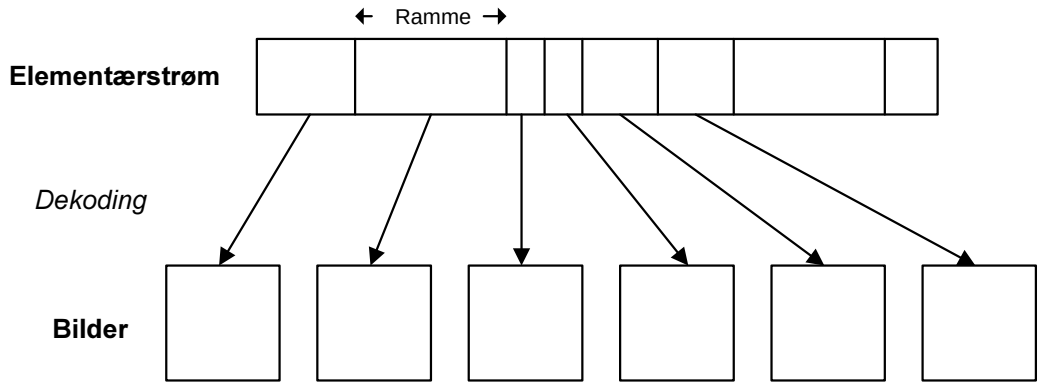
Figur 40 Demultipleksing

Senere skal denne elementærstrømmen utgjøre utgangspunktet for en ny transportstrøm. I tillegg til at dekoderen må ha samme format på spolefilen som for normalfilen, bør video-elementærstrømmen ha samme PID-verdi som den har i normalfilen. Dette fordi en MPEG-2 dekoder som spiller av en transportstrøm vil være innstilt på å ta imot elementærstrømmer med gitte PID-verdier. Disse PID-verdiene bør være de samme før, under, og etter spoling slik at dekoderen ikke får problemer. Dette betyr at PID-verdien må tas vare på ved demultipleksing, og senere brukes som innparameter for konstruksjon av den nye transportstrømmen. Dersom denne verdien ikke hentes ut nå, må dette gjøres senere ved konstruksjon av den nye transportstrømmen. Figur 40 viser at PID-verdien blir hentet ut fra PSI-pakkene (Men den kan også hentes fra transportpakker med video).

En komplett demultiplekser som tar hensyn til alle aspekter ved MPEG-2 standarden vil være rimelig kompleks, så det vil være ønskelig å bruke ferdig programvare som utfører denne oppgaven.

Dekoding
Dekoding av enkeltrammer gjøres for å få tilgang til enkeltbildene i filmen. En dekoder vil ta utgangspunkt i elementærstrømmen og generere enkeltbildene som denne representerer. Den vil ha samme funksjonalitet som en vanlig avspiller, med den forskjell at den skriver bildene til fil istedetfor å vise de på et display. For informasjon om hvordan dekodingsprosessen fungerer henvises det til kapittelet «MPEG-2», samt ISO/IEC 13818-2 Video. De dekodede bildene vil naturlig nok bli vesentlig større enn de tilsvarende rammene i strømmen, noe som bør tas hensyn til ved dekoding av store mengder data.

Dekodingsprosessen er illustrert i Figur 41 (NB, Enkeltbildene presenteres ikke i samme rekkefølge som vist på figuren).



Figur 41 Dekoding

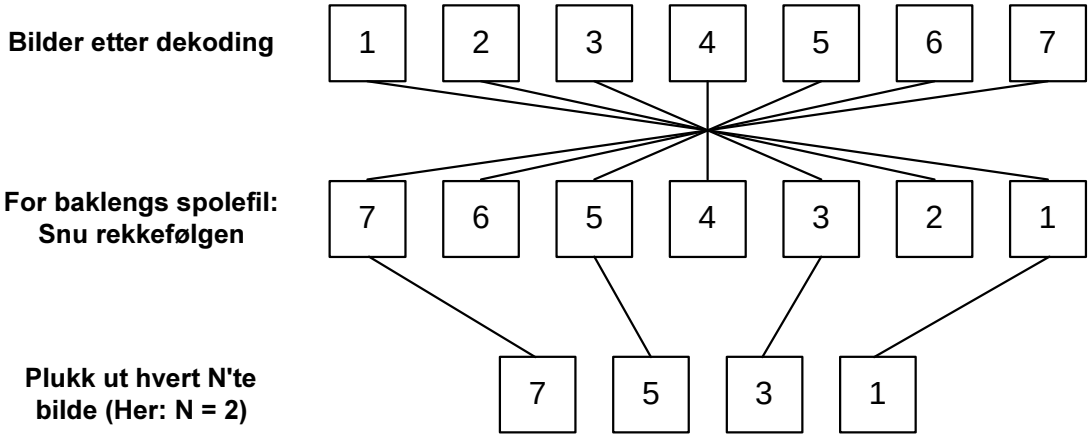
De fleste dekodere har mulighet til å generere bildene i flere ulike formater. Noen av de vanligste formatene er JPEG, GIF, PPM, YUV og TGA. Merk at enkelte format vil lagre et bilde som ulike komponenter i flere filer (YUV er et eksempel på dette). Dette er i og for seg ikke noe problem, men siden vi i neste trinn skal behandle disse filene, anses det som enklest dersom det velges et format som lagrer et bilde i en fil.

For *interlacet* (mellomlinjert) video (se kapittel «Bakgrunn») har dekoderen to muligheter: Den kan enten lagre hver enkelt interlacet ramme som to halvdeler (to felt), eller den kan kombinere dataene fra to rammer og sette disse sammen til de to hele bildene. Av samme grunn som nevnt over anbefaler vi å velge det siste alternativet for dette systemet. Å ha hvert bilde samlet i en fil vil forenkle den etterfølgende prosessen.

Å dekode en MPEG-2 video er en meget kompleks prosess og ferdig programvare som utfører en slik dekoding vil være nødvendig å bruke. Programvaren må støtte bruk av parametre som muliggjør styring av de nevnte dekodingsparametrene.

Sortering av bilder
Denne prosessen vil være en relativt enkel oppgave. Den består i å tilrettelegge de dekodede bildene slik at når de kodes på nytt så vil en spoleeffekt oppnås. Dersom vi går ut ifra at rammeraten ikke endres for spolefilen, vil vi enkelt kunne konstruere en spoleeffekt på N ganger, ved å plukke ut hvert N'te bilde. Dersom spolefilen skal representere baklengs avspilling, må rekkefølgen på bildene snus i tillegg. For baklengs avspilling i normal hastighet beholder vi alle bildene, alt vi gjør er å snu rekkefølgen.

I Figur 42 vises en skisse av hvordan snuprosessen og utvelgelsen foregår for å lage en spolefil for bakoverspoling med dobbel hastighet..



Figur 42 Sortering av bilder

Bildene fra dekoderen vil være lagret i filer, og dekoderen vil måtte ha et system som definerer rekkefølgen på bildene slik de var kodet. Sorteringsprosessen må sørge for en ny rekkefølge på bildene før koding. Dette kan gjøres enten ved å omdefinere dette systemet, eller ved å kopiere innholdet i bildene. Å omdefinere systemet vil sannsynligvis være det mest effektive. Det mest naturlige for dekoderen vil være å la *filnavnene* bestemme rekkefølgen på bildene, i dette tilfellet vil en omdefinering av systemet innebære å gi nye navn til filene.

Programmet som skal utføre denne sorteringen bør ta følgende innparametre: et flagg som sier om rekkefølgen skal snus eller ikke, og en faktor som sier hvilken spolehastighet som skal oppnås.

Etter at bildene er sortert (og kodet), kan bildene på nytt omorganiseres for koding av en ny spolefil. Her har programmet to muligheter; den kan ved ny kjøring benytte de nye bildene som utgangspunkt for reduseringen, dette vil føre til en ytterligere økning i spolehastighet. Den andre muligheten er å kopiere tilbake de originale bildene før kjøring. Dersom det førstnevnte alternativet velges bør man være oppmerksom på at det sannsynligvis ikke vil være mulig å redusere hastigheten igjen siden enkelte bilder allerede er fjernet. Ved en slik framgangsmåte vil man måtte starte med de sakteste spolehastighetene, for så å øke ved de etterfølgende kjøringene. Å snu rekkefølgen vil selvsagt kunne gjøres flere ganger uten slike konsekvenser.

Koding
Etter at bildene er omorganisert vil disse kodes i henhold til ISO/IEC 13818-2 Video standarden. Koderen bør bruke samme system for identifisering av rekkefølgen på bildene som dekoderen brukte. I et dekoder / koder system fra samme produsent vil dette etter all sannsynlighet være tilfelle. (Om ikke dette er tilfellet må programmet som sorterer bildene også tilpasse systemet slik at koderen forstår det.)

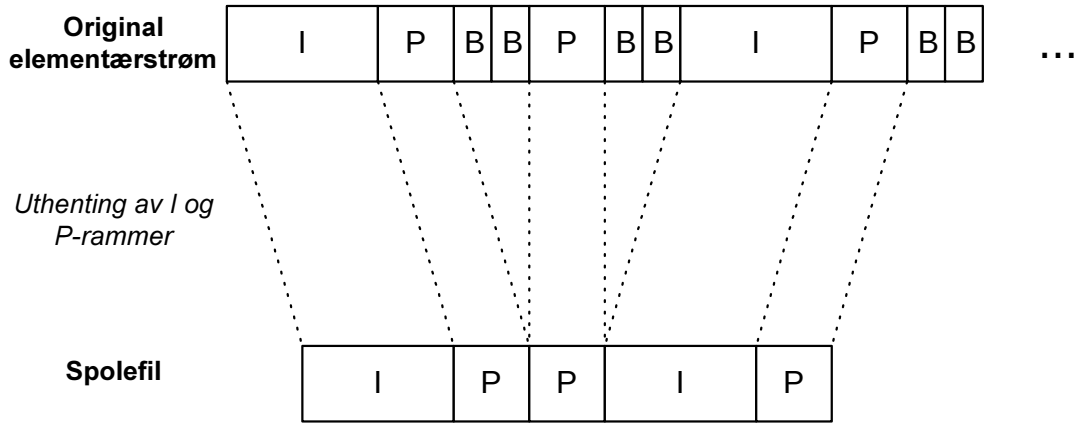
Det finnes mange måter å kode en strøm på. Derfor vil man for de fleste kodere kunne sette en hel mengde parametre. Disse vil styre kompresjonsgrad, avstand mellom ulike typer rammer, størrelse på bildene osv. Det vil være viktig at enkelte av disse parametrene settes til de samme verdiene som de hadde ved originalstrømmen. Et eksempel på dette er størrelsen på bildene. Dessuten må selvfølgelig parametre som styrer bildeformat stemme med bildene som skal kodes.

Dersom det er ønskelig kan bildekvaliteten senkes ved kodingen. Dette vil føre til en lavere nettverksbelastning ved spoling samtidig som at spolefilen vil kreve mindre lagringskapasitet. Parametre for dette kan være kvantiseringsinformasjon (satt i kvantiseringsmatriser), endring av lengden på bevegelsesvektorene, setting av maksimal bitrate, øke avstanden mellom I-rammer osv.

For å endre hastigheten kan vi også endre rammeraten. Det vil imidlertid være enklest å la denne være lik for både originalfil og spolefiler. Dermed vil det kun være antall bilder som plukkes ut som bestemmer spolehastigheten. Dessuten stiller MPEG-2 standarden begrensninger for hvor mange rammerater det er mulig å bruke.

Siden MPEG-2 støtter bruk av interlacing, kan vi også velge om spolefilen skal være interlacet eller progressiv (vanlig). En metode kan være å alltid bruke samme form for koding i spolefilen som i originalfilen. En annen framgangsmåte er å alltid bruke samme metode. Dersom dette gjøres anbefales det å bruke progressiv koding for spolefilene. Dette fordi det er en fordel at spolefilene er så enkle som mulig. En overgang mellom interlacet/ikke interlacet vil sannsynligvis ikke merkes av seeren. Noe eksperimentering bør likevel utføres på dette området.

Direkte uthenting av I- og evt. P-rammer.
Denne prosessen er et *alternativ* til å dekode/kode elementærstrømmen på nytt. Her kan man plukke ut alle eller noen I-rammer fra en elementærstrøm for så å la dette bli en ny elementærstrøm. Antallet I-rammer man vil plukke ut vil være avhengig av spolehastigheten, spolekvalitet og maksimal ønsket nettverksbelastning. Vanligvis bør alle I-rammer benyttes fordi I-rammer forekommer relativt sjeldent i MPEG-2. Det er også mulig å hente ut P-rammer i tillegg til I-rammene, dette vil gjøre at vi kan bruke flere bilder.



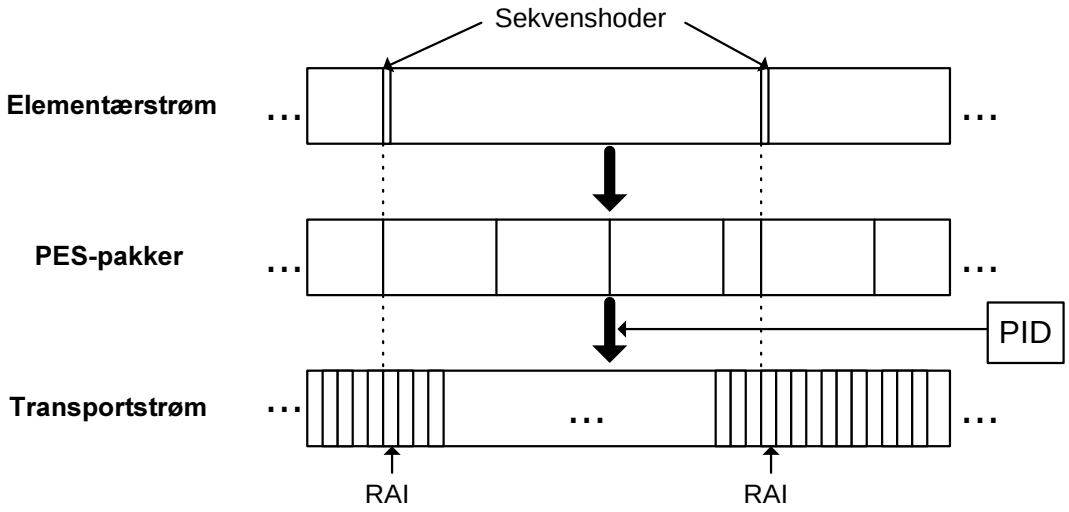
Figur 43 Direkte uthenting av I og P-rammer

Fordelen med en slik framgangsmåte er at vi slipper den tidkrevende og tungvinte prosessen med dekoding, sortering, og koding. Bakdelen er at vi mister friheten til å oppnå hvilken spolehastighet vi vil. Som vi ser av Figur 43 vil spolefilen også måtte inneholde en større datamengde enn hva som hadde vært tilfelle for ny koding. Dette fordi vi går glipp av muligheten til å ha B-rammer i strømmen.

Dette alternativet vil tilsvare den dynamiske metoden «Send bare utvalgte bilder» som beskrevet i kapittel «Løsningsalternativer». Dette er imidlertid en pre-prosesseringsmetode, I/P-rammene blir lagret på en fil på forhånd og dermed spares videotjeneren for mye arbeid i forhold til den dynamiske metoden.

Transportstrøm-multiplekser
Siden Elvira II vil bruke transportstrøm-formatet ved vanlig avspilling, er det viktig at også spolefilene er av dette formatet. Ved skifte av avspillingsmodus vil en dekoder kunne få problemer dersom den ikke mottar det samme formatet etter skiftet. Elementærstrømmen som blir konstruert bør derfor pakkes inn som en transportstrøm. En multiplekser vil kunne gjøre denne oppgaven, da med bare en elementærstrøm som input.

Som nevnt i avsnittet om demultipleksing, bør denne transportstrømmen identifisere elementærstrømmen ved den samme PID-verdien som den originale fila gjorde. Derfor bør nå den uthentede PID-verdien brukes som parameter til multiplekseren. Dette sikrer at dekoderen kontinuerlig kan bruke den samme PID-verdien til identifikasjon av video ved skifte av avspillingshastighet. Figur 44 viser multipleksingsprosessen der den gamle PID-verdien blir brukt.



Figur 44 Multipleksing

Et viktig punkt her er at multiplekseren bør konstruere strømmen slik at tilfeldig aksess er mulig. Dette gjøres ved å benytte RAI (random access indicator) til å identifisere *sekvenshodene* som bør finnes i strømmen. Det vil si at sekvenshodene må bli identifisert i elementærstrømmen og RAI-biten satt i de nødvendige pakkene. Som vi ser av Figur 44 bør PES-pakkene konstrueres slik at en PES-pakke begynner på et aksesspunkt. Dette vil føre til at noen PES-pakker blir kortere enn andre. Dette er ikke noe problem, pakkestørrelsen kan være variabel. Tilfeldig aksess diskuterer vi videre senere i konstruksjonen.

MPEG-2 standarden tillater diskontinuitet i PCR-verdiene. Det vil si at verdien kan gjøre et sprang midt i filmen. Dette kan ikke tillates for en film som skal legges inn i Elvira II. Den vil kreve at PCR-verdiene stiger jevnt og kontinuerlig gjennom hele filmen, slik at de gir et bilde på spilletiden. Multiplekseren må derfor sørge for dette.

Å lage en gyldig transportstrøm med alt det innebærer av korrekte tidsstempler og andre verdier er en stor oppgave. Det er derfor en fordel å kunne bruke ferdig utviklet programvare for denne oppgaven.

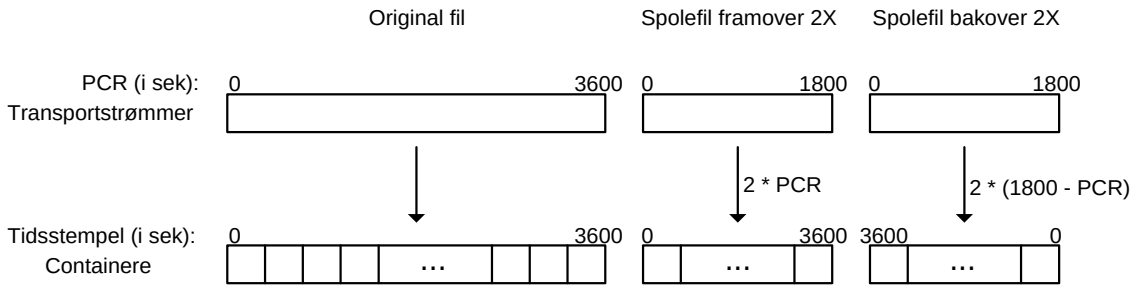
Spolefiler i Elvira II

Tidligere i konstruksjonen beskrev vi hvordan en transportstrøm kan legges inn i et Elvira II filsystem. Det ble da fokusert på å legge inn en strøm med normal hastighet. Nå skal vi se på hvordan de konstruerte spolefilene med ulike hastigheter kan legges inn. Strømmene vil ha samme oppbygning, slik at prosessen beskrevet tidligere også vil gjelde for disse filene. Det vil imidlertid bli en viktig forskjell: setting av tidsinformasjon i containere. Som tidligere nevnt må tidsstemplene i disse representere *normaltida* i en film, slik at tjeneren kan skifte på rett tidspunkt mellom filene. Ved koding av en transportstrøm vil de nye PCR-verdiene ikke reflektere dette, de vil representere tida det tar å spille av spolefilen.

Dette gjør at vi må modifisere på tidsstemplene før vi legger de inn i containerene. Denne modifiseringen vil bestå i å multiplisere tidsverdien med N, der N er spolehastigheten. Dermed vil containerene få tidsstempler som kan brukes til å identifisere hvilket tidspunkt i normalfila denne delen av spolefilen representerer.

For en baklengs spolefil må man i tillegg *snu* rekkefølgen på tidsstemplene. Det vil si at tidsstemplene skal begynne med den høyeste verdien for så å synke til 0. Formelen for dette blir ganske enkel: Nytid = Maks.tid - original tid. Vi må altså kjenne den maksimale tiden før vi beregner tiden for hver enkelt container. Siden vi for interpoleringen allikevel bør dele denne operasjonen i to, er ikke dette noe problem. Vi løper først igjennom og finner ut av de kjente verdiene, da vil vi også kjenne til den høyeste verdien. Etterpå kan disse interpoleres, multipliseres med N, og snus rekkefølgen på.

I Figur 45 vises det hvordan tidsstemplene må manipuleres for spolefilene sammenlignet med originalfilen. Eksempelet viser en originalfil på 1 time, samt to spolefiler for dobbel hastighet, en forlengs og en baklengs.



Figur 45 Konstrukjson av tidsstempel

Som nevnt skal hver container ha to tidsstempel. Det ene tidsstempelet, «Showtime», skal angi når containeren skal komme fram til dekoderen. Tjeneren vil selv modifisere på dette tidsstempelet ved kjøring, fordi den vet spolehastigheten og retningen. Derfor kan også dette tidsstempelet for spolefilene foreløpig settes lik «Movietime», slik det ble gjort for normalfila.

Etter at containerene er generert for en spolefil settes disse inn i Elvira II formatet og filsystemet på samme måte som beskrevet tidligere i konstruksjonen. Indeksinformasjon som sier hvor disse spolefilene blir lagret lages også på samme måte, de nye tidsstemplene vil da danne grunnlaget for tidsinformasjonen for hvert segment.

En film skal ha en indeksfil som i tillegg til normalfilens indeks også inneholder alle instanser av spolefiler for denne filmen. Derfor må indeksinformasjonen for hver spolefil *legges til* indeksfilen for filmen. Se forøvrig vedlegg 4 for en beskrivelse av indeksfilformatet. Indeksfilen må altså genereres i flere trinn, der vi under hver innlegging av en spolefil også må legge til en ny indeks-instans av en avspillingsfil i filen. Normalfila må legges inn som første instans. I begynnelsen av indeksfila skal det også være et hode med navn og lengde på filmen, disse verdiene settes da ved innleggelse av normalfila.

For å oppsummere får vi følgende framgangsmåte for generering av en indeksfil for en film:

- **Legg normalfila inn i Elvira II formatet.**
    - **Generer hodet på indeksfilen.**
    - **Generer indeksene for normalfilen**
- **Legg spolefilene for filmen inn i Elvira II formatet - for hver spolefilm:**
    - **Generer indekser, legg disse til nederst i indeksfila.**

Tilfeldig aksess

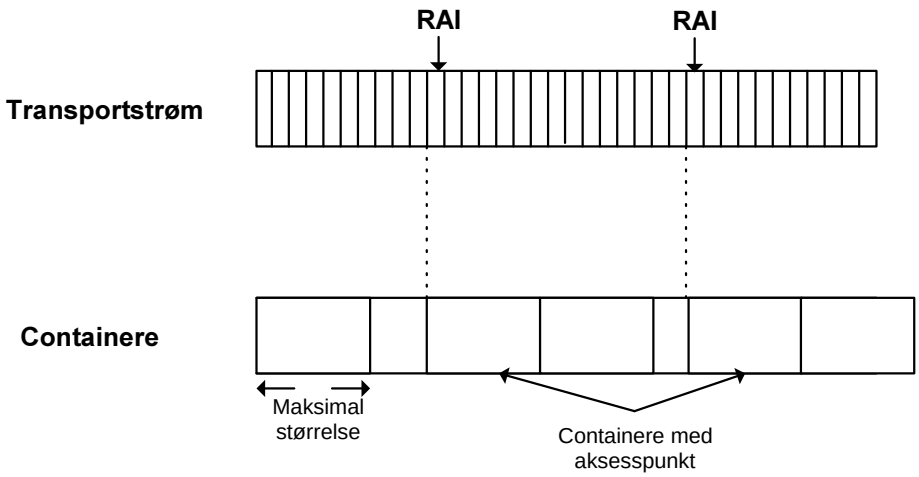
Tilfeldig aksess er nødvendig både for å kunne hoppe i normalfila, og for å kunne hoppe mellom spolefiler (og normalfila). Det betyr at alle filene som er konstruert må tilby tilfeldig aksess dersom gode overganger skal implementeres. Som tidligere nevnt vil tilfeldig aksess kunne støttes ved å identifisere sekvenshoder. Et sekvenshode vil alltid finnes i begynnelsen av en strøm, men må ikke nødvendigvis repeteres. For å implementere tilfeldig aksess på en sikker måte må vi sørge for at strømmen inneholder repeterte sekvenshoder og at disse er identifisert med «random access indicator» (RAI). Se avsnittet om generering av spolefiler for å se hvordan man konstruerer spolefiler med disse egenskapene.

Sekvenshodene vil være naturlige å gjenta for hver GOP (Group of pictures). Det vil si foran hver I-ramme. Avstanden mellom I-rammene vil derfor være bestemmende for hvor ofte vi får et aksesspunkt, dvs. aksessoppløsningen. Det må gjøres oppmerksom på at ved generering av spolefiler vil antall bilder reduseres, og vi vil få et større tidsrom spilt av for en GOP (forutsatt at vi koder spolefilene med samme I-ramme avstand, noe

det er naturlig å gjøre). Dette gjør at aksessoppløsningen per tidsrom vil bli dårligere jo større spolehastighet vi har. Dersom vi i en vanlig fil har to aksesspunkt i sekundet vil vi for en 4X-spolefil ha et aksesspunkt kun annethvert sekund. Dette vil føre til at vi ved hopp inn i spolefiler med høye hastigheter vil kunne få en unøyaktighet i plasseringen. Dette er uunngåelig, men vil likevel ikke være et stort problem; når brukeren skal spole hurtig vil ikke nøyaktighet i overgangen være viktig.

Siden containere er den minste enheten Elvira II vil jobbe med, bør et aksesspunkt starte i begynnelsen av en container. Dette gjør at vi slipper å «ta vekk» begynnelsen av en container på dekodersiden, dekoderen kan begynne å lese fra containeren direkte. Vi vil derfor nå utvide prosessen som konstruerer containere med følgende egenskap: hver gang den finner et aksesspunkt i en transportstrøm (ved å lokalisere en transportpakke med RAI) skal den avslutte arbeidet med nåværende container. Den skal så sette denne pakken og alle etterfølgende inn i en ny container. Dette gjør at den foregående containeren vil bli mindre enn maksimumsstørrelsen, men dette går bra; containerstørrelsen er variabel.

Denne prosedyren er illustrert i Figur 46. (NB! Størrelsesforholdet mellom transportpakkene og containerene er ikke nødvendigvis korrekt, figuren viser et prinsipp)



Figur 46 Konstruksjon av containere med aksesspunkt

Videotjeneren må under kjøring kunne finne fram til containere med aksesspunkt. Dette kan gjøres ved å generere en egen *«aksess-indeksfil»*. Filen vil inneholde en utlisting over tidspunktene for containere som inneholder aksesspunkt. Denne aksessindeksen bør på samme måte som den tidligere nevnte indeksfila slås sammen slik at aksesspunktene for en film og alle dens spolefiler ligger samlet i en fil. Ved hjelp av disse to filene, indeksfila og aksess-indeksfila, kan så tjeneren finne fram til ønsket container. Dette blir beskrevet nærmere i neste avsnitt.

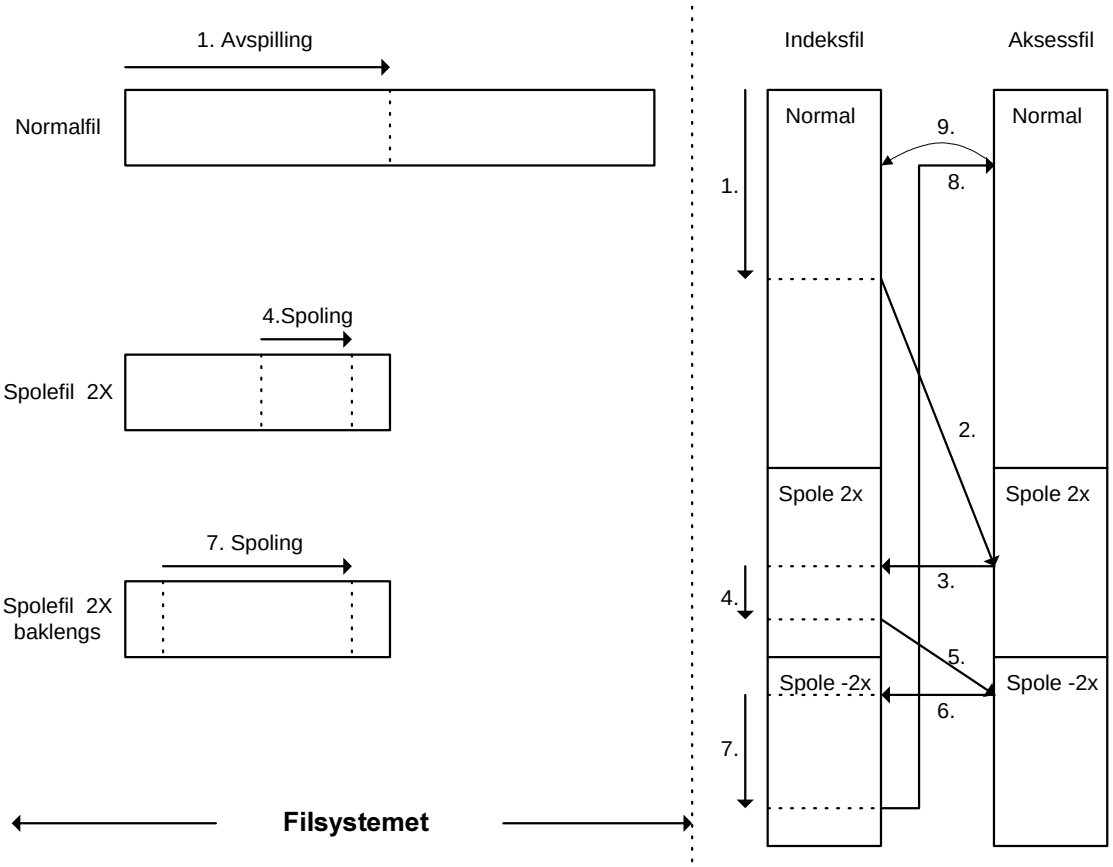
Bruk av indeksfiler og aksess-indeksfiler

Vi skal nå illustrere hvordan tjeneren vil bruke indeksfiler og aksess-indeksfiler sammen. Til dette bruker vi følgende situasjon som eksempel: En film har knyttet til seg to spolefiler, en i hver retning. Brukeren har dermed muligheten til å spole forover og bakover med dobbel hastighet.

Vi tenker oss nå at brukeren foretar følgende operasjoner:

- Starter normal avspilling fra begynnelsen av filmen.
- Ved halvspilt film, start spoling forover.
- Når filmen er spolt nesten helt fram, start spoling bakover.
- Når filmen er spolt nesten helt tilbake, start normal avspilling igjen.

Figur 47 viser hvordan disse operasjonene vil bli utført av tjeneren. Til venstre for den stiplete linjen vises filmen med spolefiler som ligger i filsystemet. Til høyre for linjen ligger indeksfilene som brukes. Avspillinger er merket med piler over videofilene. «Indeksfil» vil brukes kontinuerlig ved avspilling av en fil, derfor vil vi finne igjen de samme pilene med samme nummer ved siden av indeksfilen. De andre pilene mellom indeksfilene representerer aksesser og søking i filen.



Figur 47 Bruk av indeksfiler

De ulike operasjonene tjeneren utfører i Figur 47 er som følger: (numrene tilsvarer numrene på figuren)

1. Avspilling. Bruker indeksfilen til å finner lokasjonen for segmentene som skal leses inn og til å justere hastigheten.
2. Tjeneren har mottatt signal om å spole framover. Den bruker det nåværende tidspunktet i filmen til å søke i aksessfila for framoverspoling etter det nærmeste tidspunktet. Dette tidspunktet vil være et gyldig tidspunkt for igangsetting av spoling.
3. Tidspunktet funnet i aksessfila skal samsvare med tidspunktet for en container i spolefila. Tjeneren søker i indeksfila for framoverspoling etter segmentet som inneholder containeren og leser inn dette segmentet.
4. Tjeneren finner containeren og starter avspilling herfra. Den bruker nå indeksfila for framoverspoling til å finne segmentene og å sende disse til rett tidspunkt.
5. Tjeneren har mottatt signal om å spole bakover. Den bruker det nåværende tidspunktet i filmen til å søke i aksessfila for bakoverspoling etter det nærmeste tidspunktet. Dette tidspunktet vil være et gyldig tidspunkt for igangsetting av spoling. Merk at siden vi er sent i filmen vil dette tidspunktet representere et sted tidlig i bakoverfila.
6. Tidspunktet funnet i aksessfila brukes til å søke i indeksfila for bakoverspoling, den finner segmentet som inneholder containeren med dette aksesspunktet, og leser inn dette.
7. Tjeneren starter avspilling fra funnet container, den bruker indeksfila for bakoverspoling.

8. Tjeneren mottar signal om å spille normal avspilling. Den bruker det nåværende tidspunktet til å søke etter det nærmeste aksesspunktet i aksessfila for normalfila.
9. Tjeneren er klar til å spille av med normal hastighet.

Sakte avspilling

Til slutt vil vi beskrive hvordan sakte avspilling (slow motion) kan implementeres. Elvira II har muligheten til å sende containerene til klienten i langsommere hastighet. Siden dekoderen da umulig kan spille av filmen i normal hastighet, vil en form for hastighetssenking forekomme. Hvordan denne avspillingen vil bli er derimot usikkert, kvaliteten kan bli relativt dårlig. Dekoderbufferet vil gå tomt, og vi vil kunne få en hakkete avspilling.

For å bøte på dette har vi valgt å benytte en dynamisk metode; manipulering av tidsstempel i strømmen. Dette betyr at disse mekanismene må inngå i tjeneren, og må kjøres under utsendelsesprosessen. Det finnes to måter å gjøre dette på: Den første går ut på at tjeneren for hver container den skal sende ut går igjennom alle transportpakkene som finnes i denne containeren, analyserer disse, og manipulerer eventuelle tidsstempel den skulle finne. Det bør imidlertid undersøkes hvor prosessorintensiv denne operasjonen er, det er mange transportpakker å kontrollere. Den andre metoden vil hjelpe på dette, her vil vi utvide containerhodet med en tabell som sier hvilke transportpakker i containeren som inneholder tidsstempler. Dermed slipper vi å kontrollere så mange pakker.

I tillegg til å modifisere på stemplene må selvsagt tjeneren justere utsendelseshastigheten av containerene til klienten.

For sakte avspilling framover kan vi bruke originalfila som utgangspunkt. Vi kan bruke de samme indeksfilene som er generert for fila, men tjeneren må justere lesehastigheten for segmentene avhengig av hastigheten på avspillingen. For sakte avspilling bakover må det være generert en spolefil for normal avspilling bakover. Også indeksfilene for denne spolefilen kan brukes av tjeneren.

Fordelen med en slik metode er at alle spolehastigheter mellom 0 og 1 kan oppnås. Tidsstemplene må ganges med en faktor som er større enn 1. For å få til avspilling med halv hastighet (0,5) må tidsstemplene ganges med 2. Formelen for faktoren blir altså 1/X, der X er en avspillingshastighet mellom 0 og 1.

Tidsstemplene som må modifiseres er:

- PCR (Program Clock Reference): Finnes i tilpasningsfeltet (adaptation field). Kan lett identifiseres og modifiseres. Tidsstempelet sier når disse dataene skal komme fram til dekoderen.
- PTS (Presentation Time Stamp): Finnes i hodet til en PES-pakke. Kan relativt lett identifiseres og modifiseres. Tidsstempelet sier når neste ramme i strømmen skal presenteres.
- DTS (Decoding Time Stamp): Finnes i hodet til en PES-pakke. Kan reltaiv lett identifiseres og modifiseres. Tidsstempelet sier når neste ramme i strømmen skal dekodes.

Det er viktig at alle disse stemplene modifiseres og at alle multipliseres med samme verdi. Dersom dette ikke blir gjort vil oppførselen til dekoderen være uforutsigbar.

For å modifisere på tidsstemplene kreves det en del behandling av enkeltbit. Dette fordi stemplene er kodet i uregelmessige mønster i bytene, og gjerne med markeringsbit innimellom. Spesielt gjelder dette for PTS og DTS, PCR er noe enklere kodet. Ellers henvises det til ISO/IEC 13818-1 Systems for en beskrivelse av hvordan stemplene er kodet. (Se også vedlegg 6).

Implementasjonsbeskrivelse

Det er i løpet av denne oppgaven implementert et system som består av både egenutviklede programmer og ferdige programmer/verktøy. Det er tatt sikte på å implementere så mye av konstruksjonen som mulig, med spesiell vekt på å få implementert og testet bruk av spolefiler i videotjeneren.

Dette kapittelet beskriver programmene som er implementert og andre programmer som er brukt i dette systemet. Vi skal også si litt om prosessen og arbeidsomgivelsene for implementasjonen.

Overordnet beskrivelse

Først vil vi gi en overordnet beskrivelse av hvilke moduler det implementerte systemet består av. Her legges det vekt på å gi en forståelse av helheten, en mer detaljert beskrivelse av hvert enkelt program vil gis til slutt i kapittelet. Alle programmene kjører under et UNIX operativsystem.

Hva er implementert?

Det er implementert en programpakke som konstruerer spolefiler. Denne pakken består av følgende programmer:

- **ts_demux:** Dette er en demultiplekser som henter ut en elementærstrøm fra en transportstrøm. Det er laget av Christos Tryfonas, University of California.
- **mpeg2decode:** Dette programmet dekoder enkeltbildene i en MPEG-2 video-elementærstrøm. Det er laget av MPEG Software Simulation Group.
- **wsort:** Dette er et program som sorterer de dekodede enkeltbildene for koding av spolefil. Dette programmet er egenutviklet.
- **mpeg2encode:** Dette er en koder som genererer en MPEG-2 video-elementærstrøm fra enkeltbilder. Det er laget av MPEG Software Simulation Group.
- **mux:** Dette er en enkel transportstrøm-generator. Fra en video-elementærstrøm lager den en transportstrøm. Den nye strømmen inneholder ikke gyldige tidsstempler. Dette programmet er egenutviklet.

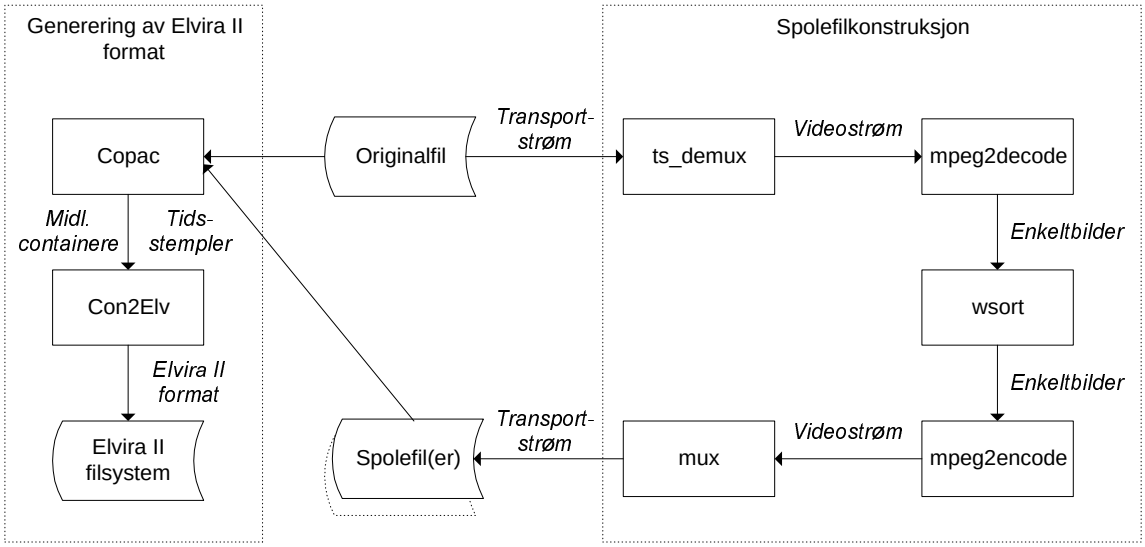
Konstruksjonen av spolefiler består som vi ser av flere program som er utviklet utenfra, dette har vært nødvendig da flere av disse trinnene er svært kompliserte. Spesielt gjelder det for dekoding og koding av video. Også demultipleksing og multipleksing av en transportstrøm er en relativ komplisert oppgave. For demultipleksingen har vi benyttet et ferdig program, men for multipleksertrinnet har det ikke lykkes oss å finne et slikt program tilgjengelig på Internet. Det ble derfor nødvendig å lage et egenutviklet program som lager en svært enkel transportstrøm fra en elementærstrøm. Dette medførte noen endringer i forhold til konstruksjonen. Dette kommer vi nærmere tilbake til senere i kapittelet i avsnittet for programbeskrivelser.

Det er også implementert en programpakke som overfører både originale filer og dens spolefiler inn i Elvira II formatet. Denne programpakken er basert på og videreutviklet av programmer laget i et prosjekt våren 1997 (Remmem/Østerhagen). Pakken består av to deler:

- **Copac:** Dette programmet behandler en transportstrøm og konstruerer midlertidige containere, og genererer samtidig en fil som består av tidsstemplene i transportstrømmen.
- **Con2Elv:** Dette programmet bruker de genererte filene laget av copac. Containere blir lagt inn i Elvira II formatet, og tidsstemplene i disse blir interpolert utifra fila med tidsstemplene fra copac. Nødvendige indeksfiler blir også generert.

Som nevnt i konstruksjonen var det nødvendig med to gjennomløpinger av tidsstemplene i containerene slik at en interpolering kunne gjennomføres. For å gjøre disse prosessene så modulære og uavhengige av hverandre som

mulig, ble det besluttet å lage to adskilte programmer for dette. Filer på et mellomformat danner dataoverføringen mellom programmene. Også disse programmene og de nevnte filene vil beskrives senere i kapittelet.



Figur 48 Oversikt over systemet

Figur 48 viser en oversikt over hele systemet, den viser alle programmer og sammenhengen mellom disse.

I tillegg til dette systemet er det implementert noen klasser/programmer som er brukt til testing og eksperimentering og som senere kan benyttes ved videreutvikling. Disse er:

- **Tpacket:** Dette er en klasse som er brukt til å manipulere data i en transportpakke. For eksperimentering er det inkludert mange funksjoner, men spesielt er funksjonene for uthenting og modifisering av tidsstempler interessant.
- **Eltos:** Dette er et enkelt program som trekker ut transportstrømmen fra en Elvirafil. På denne måten er det testet at elvirafilen er gyldig. Programmet er videreutviklet fra et program laget av Remmem/Østerhagen, våren 1997.

Klassen tpacket ble brukt til å eksperimentere med strømmen. For å studere hvordan endringer av tidsstempler vil virke inn på avspillingen ble denne klassen brukt til å lage manipulerte strømmer. Særlig ble det undersøkt hvordan sakte avspilling kan utføres. Denne klassen kan senere brukes/videreutvikles ved innføring av sakte avspilling for MPEG-2 i Elvira II.

Hva er ikke implementert?

Dette systemet er ikke fullstendig implementert i henhold til konstruksjonen, noen ting gjenstår. Disse er:

- **Korrekt multiplekser**

Programmet mux gjør en rekke forenklinger som var nødvendige å gjøre pga. den begrensede tiden for denne oppgaven. Spesielt gjelder dette for genereringen av tidsstempler, se programbeskrivelsen senere i kapittelet. Det bør derfor på dette trinnet senere innføres en fullt implementert multiplekser.

- **Innføring av sakte avspilling i Elvira II**

Konstruksjonen beskrev en metode som benyttet modifiserte tidsstempel kombinert med lavere overføringshastighet mellom Elvira II og tjeneren. Det er som nevnt utført tester med modifiserte tidsstempler i en generert strøm, men dette prinsippet er ikke integrert i Elvira II. Spolingen har vært prioritert, og det har heller ikke vært mulig å teste dette i Elvira II fordi vi ikke har en dekoder som leser direkte fra et nettverk.

Brukte verktøy

Programmeringspråket C++ ble brukt til å implementere dette systemet. Dette fordi språket er objektorientert, og fordi det allerede blir brukt for Elvira II. Det var også dette språket jeg kjente best til på forhånd. Til kompilering ble GNU's kompilator g++ brukt.

For å spille av både elementærstrømmer og transportstrømmer ble databasegruppas MPEG-2 kort brukt. Dette kortet kommer fra Optibase og kalles VideoPlex. Kortet fungerer bra, men har den svakhet at den ikke kan lese direkte fra nettverket. Dette gjorde at det ikke var mulig å koble dette direkte opp mot Elvira II. For å spille av strømmer ble det derfor nødvendig å alltid ha disse på fil.

I tillegg ble det også brukt en programvare-spiller som kunne dekode MPEG-2 video-elementærstrømmer. Denne kalles mpeg2play, og er utviklet av Stefan Eckart.

De tre programmene som utgjør deler av systemet, vil beskrives nærmere senere i kapittelet.

De fleste programmene som er brukt i denne oppgaven kan lokaliseres ved hjelp av WWW-siden «MPEG - Pointers and Resources», som er vedlikeholdt av Tristan Savatier. Denne siden anbefales dersom man er interessert i programvare eller informasjon relatert til MPEG-standardene.

Erfaringer/Testing

En god del tid ble brukt til eksperimentering i denne oppgaven. Først måtte de nødvendige programmene lokaliseres, deretter var det nødvendig å sette seg inn i bruken av de. Mesteparten av eksperimenteringen besto nok i å kode videostrømmer med ulike parametre. Disse strømmene ble underveis testet enten på MPEG-2 kortet eller på programvare-spilleren.

Det største problemet jeg støtte på var at det ikke var mulig å finne en fungerende multiplekser på Internet. Jeg fant et bibliotek som skulle kunne brukes til å lage multiplekserapplikasjoner, men dette fungerte ikke som det skulle. Kildekoden var gammel, og det virket som om prosjektet ikke ble fulgt opp. Det ble derfor nødvendig å selv lage et program som konstruerte en transportstrøm fra en elementærstrøm, mux. Grunnet dårlig tid var det nødvendig å foreta en god del forenklinger. Likevel spiller MPEG-2 kortet fint av strømmene programmet genererer.

Noen av programmene ble videreutviklet fra prosjektet «MPEG-2 & Elvira II» (Remmem/Østerhagen, 1997). Disse programmene ble implementert under Windows NT med Visual C++. Siden resten av Elvira prosjektet er laget for kjøring under UNIX, valgte jeg for denne oppgaven å implementere i UNIX-omgivelser. Dette gjør at vi slipper problemer med kompilering, filbehandling o.l. ved integrering av programmene. Selv om jeg ikke hadde mye erfaring med programmering i UNIX-omgivelser, gikk denne overgangen svært smertefritt.

Programbeskrivelser

Vi deler opp programbeskrivelsen i to deler, først beskriver vi de ferdige progammene som er brukt, deretter beskriver vi de egenutviklete. Vi skal se på funksjonalitet og bruk av programmene. For de egenutviklede ser vi også på programoppbygningen.

Andre programmer

ts_demux

Dette programmet er skrevet av Christos Tryfonas. Den henter ut en elementærstrøm fra transportstrømmen. Inn-parametre er PID-verdien på elementærstrømmen og navnet på transportstrømmen.

Bruken er som følger:

```
ts_demux <transportstrøm> <PID-verdi>
```

ts_demux er en del av en større pakke. To andre programmer i pakken kan brukes til å finne ut hvilke PID-verdier elementærstrømmen(e) i transportstrømmen har. Disse er ts_filter og ts_viewer. ts_filter produserer en binærfil med informasjon om strømmen, ts_viewer kan brukes til å lese denne fila. Ellers henvises det til dokumentasjonen som følger med denne pakken.

Som nevnt i konstruksjonen skal PID-verdien nå tas vare på for å bruke denne senere. Som vi skal se senere, gjør vi i dette systemet multipleksingen på en litt spesiell måte. Vi trenger derfor ikke å ta vare på denne verdien for denne utgaven av systemet.

mpeg2decode

Dette programmet er skrevet av MPEG Software Simulation Group. Den brukes til å dekode enkeltbildene i en videostrøm og legge disse ut på fil. Det er mulig å lagre bildene i følgende formater: YUV (tre komponenter), SIF, TGA, PPM, og X11 (visning i et X-vindu). Det er også mulig å lagre interlacede bilder i enten rammeformat (hele bildet i en fil) eller i fieldformat (halve rammer danner en fil). Begge disse opsjonene angis ved parametre på kommandolinja. Bruken er som følger:

```
mpeg2decode [opsjoner] <utfil%d> -b <videostrøm>
```

Parameteren «utfil%d» angir at det skal genereres bilder med navnet «utfil» + bildenummer. Se forøvrig dokumentasjon som følger med programmet.

For systemet vårt velger vi å lagre bildene i rammeformat, slik at hvert bilde blir lagret i en fil. Dette angir vi ved opsjonen -f. Koderen vi bruker i neste trinn støtter bare bruk av PPM og YUV formater, så vi må derfor velge en av disse. Vi velger PPM fordi det blir unødvendig komplisert med YUV, her vil filene lagres i tre komponenter. Vi velger PPM ved å sette opsjonen -o3.

mpeg2encode

Dette programmet er skrevet av MPEG Software Simulation Group. Ved å ta en mengde bilder lagret i et format vil den kode en MPEG video elementærstrøm. Den kan kode både MPEG-1 og MPEG-2. Siden dette programmet følger med i samme pakke som mpeg2decode, vil den direkte kunne benytte filnavnene på de genererte bildene som grunnlag for rekkefølgen den skal kode bildene i. «Basenavnet» på filnavnene angis i en parameterfil som også inneholder en mengde andre parametre. Som nevnt godtar denne koderen bare to bildeformater: YUV og PPM. Bruken av programmet er:

```
mpeg2encode <parameterfil> <utfil>
```

Parameterfila inneholder mange innslag og det er viktig at hvert innslag ligger på riktig linjenummer i tekstfila. Se vedlegg 2 for en beskrivelse av fila.

Vi kan spesielt nevne bruken av parameteren *bitrate*. Denne sier hvilken bitrate strømmen vil få, altså hvor mye data som skal representere strømmen per tidsenhet. Ved å redusere denne verdien, sette lavere bitrate, vil vi automatisk få en redusering i kvaliteten. Dermed slipper vi å endre på andre parametre som kvantiseringsinformasjon o.l.

Som beskrevet i konstruksjonen bør koderen i vårt system repetere sekvenshoder for hver GOP (Group of pictures) slik at tilfeldig aksess kan implementeres. Vi analyserte strømmen og fant at denne koderen ikke gjorde dette. Det ble derfor nødvendig å repetere sekvenshodene selv, denne operasjonen blir gjort i mux, se senere i kapittelet. En mer profesjonell maskinvare-koder vil sannsynligvis tilby repetering av sekvenshoder, dersom en slik senere blir integrert i systemet kan denne egen-implementerte repeteringen sløyfes.

Egne programmer

wsort

Dette programmet utfører en sortering av enkeltbilder som beskrevet i konstruksjonen. Programmet er rettet mot filformatet som mpeg2decode og mpeg2encode bruker. Det sorterer filene som er generert ved å gi nye navn til filene. Ved å oppgi en faktor som parameter kan filene konstrueres slik at når de kodes på nytt vil videoen få en spoleeffekt som er lik faktoren. Man kan også oppgi om rekkefølgen skal snus slik at det blir en baklengs avspillingseffekt i forhold til originalfilen. Bruken av wsort er:

```
wsort <filbase> <faktor> <baklengs>
```

«Filbase» vil være navnet på filen, men unntatt tallet som angir nummeret og etternavnet.

Faktoren kan være et hvilket som helst tall større enn 1, også kommatall. Det anbefales likevel å velge hele tall slik at avstanden mellom de utplukkede bildene blir konstant. Denne versjonen setter en øvre grense på 30, men denne er slett ikke nødvendig å sette (men det vil være meningsløst å kode en større spolehastighet).

Den siste parameteren <baklengs> er ikke nødvendig å sette, standard retning er forover (ingen snuing av rekkefølge). Ved å skrive «b» for denne parameteren blir rekkefølgen snudd.

Programmet vil uansett faktor alltid inkludere det første og det siste bildet. Dette sikrer at begynnelse og sluttrammen alltid vil være lik.

Programmet vil slik det er kompilert nå alltid gå utifra at det første bildet er nummer 0 (filbase0.ppm). Dette kan meget enkelt endres ved å endre på variabelen *firstfile* i programmet. Et alternativ kan være å senere la denne verdien kunne angis som parameter.

mux

Dette programmet lager en forenklet transportstrøm utifra en video-elementærstrøm. Den er ingen multiplekser i den forstand at den pakker inn bare en strøm, ikke flere. Bruken er som følger:

```
mux <elementærstrøm> <pesfil> <transportstrøm>
```

<pesfil> er bare et midlertidig filformat, dette lagrer PES-pakkene som konstrueres.

Dette programmet vil, dersom repeterte sekvenshoder ikke allerede finnes, legge inn sekvenshoder foran hver GOP. Dette ble som sagt nødvendig å gjøre for strømmene generert av mpeg2encode. Dette er egentlig ikke en multipleksers oppgave, dersom en ordentlig multiplekser blir innført her, bør denne prosessen trekkes ut og gjøres på forhånd (dersom mpeg2encode fremdeles brukes).

Det ble gjort noen forenklinger, den viktigste er denne: PTS og DTS-tidsstempler blir ikke lagt inn i strømmen, og PCR-verdiene blir ikke satt i henhold til hvor lang filmen vil bli, men etter hvor lang originalfilmen var. Lengden på originalfilmen er hardkodet, og det blir beregnet en inkrementasjon for PCR-verdien fra PES-pakke til PES-pakke. PCR-verdien blir sammen med en eventuell RAI-bit satt i en pakke kun bestående av et tilpasningsfelt. Denne pakken blir sendt før hver PES-pakke. Manglende tidsstempler gjør at dette ikke er en 100% gyldig transportstrøm. Likevel klarer MPEG-2 kortet å spille av strømmen fint. Vi regner med at den bruker rammerateinformasjonen istedetfor tidsstemplene til å styre hastigheten. Oppførselen er imidlertid ikke garantert for andre dekodere.

PCR-verdiene blir beregnet utifra lengden til originalstrømmen fordi dette forenkler situasjonen her, dekoderen kan allikevel ikke bruke gyldige tidsstempel. Nå kan vi bruke denne lengden for alle spolefilene. Samtidig forenkler dette innsettingen i containere i neste trinn, innlegging i elviraformatet, nå slipper vi å multiplisere med spolehastighet før innlegging. Dette er altså en endring i forhold til konstruksjonen. Ved innføring av en skikkelig multiplekser med gyldige PCR-verdier må det gjøres som i konstruksjonen.

Vi kan nevne en ytterligere forenkling som ble gjort for å gjøre dette så raskt som mulig (tidspresset var stort): En transportstrøm skal inneholde PSI-pakker, disse pakkene forteller hvor mange elementærstrømmer og programmer som finnes i strømmen, samt deres PID-verdier. Istedet for å lage disse pakkene, ble de først kopiert fra originalfila, lagt på en fil kalt «psifile», og senere brukt av mux til å legge disse inn i strømmen. De ble lagt inn i begynnelsen av hvert aksesspunkt slik at dekoderen med en gang kan dekode strømmen. VIKTIG: fila «psifile» må ligge i katalogen ved kjøring av mux.

Denne framgangsmåten gjør at vi slipper å ta vare på PID-verdien til elementærstrømmen ved demultipleksing, nå vil denne kopieres automatisk.

Denne metoden kunne implementeres for vår testfil fordi originalfila, som spolefila, også inneholdt bare en elementærstrøm. Dermed vil PSI-informasjonen bli den samme. Ved originalfiler med flere elementærstrømmer vil denne metoden ikke fungere. Man kan eventuelt løse dette ved å bruke uthentede PSI-pakker fra en annen fil enn originalstrømmen, men da må man passe på at PID-verdiene blir de samme for originalfila og spolefilen.

Klassebeskrivelser

ElStream:
Denne klassen inneholder hovedfunksjonaliteten for mux. Den vil analysere elementærstrømmen og inneholder den overordnede logikken som konstruerer PES-pakker og TP-pakker. Den vil foreta det meste av filbehandlingen og vil avgjøre om aksesspunkt skal legges inn for PES-pakker/TP-pakker. Først vil den konstruere PES-pakker fra elementærstrømmen, deretter vil den lage transportpakker fra disse PES-pakkene.
PES:
Denne klassen brukes til å vedlikeholde et hode til en pes-pakke. Verdier kan settes og hodet kan skrives til fil. Klassen vil brukes av ElStream under generering av PES-pakker.
TP:
Denne klassen vil administrere dataene i en transportpakke. Klassen ElStream vil bruke denne under den siste fasen i genereringen, fasen som lager transportpakker. Funksjonalitet for å lage pakker med tilpasningsfelt ligger i denne klassen.

Copac

Dette programmet utgjør første del i prosessen med å overføre en tranportstrøm til elviraformatet. Både originalfiler og spolefiler vil måtte gå igjennom denne prosessen. Programmet konstruerer midlertidige containere, og henter ut tidsinformasjon fra strømmen (PCR-verdier). Det blir produsert en containerfil og en tidsstempelfil. Bruken er som følger:

```
copac <transportstrøm> <containerfil> <interpoleringsfil>
```

Tidsstempelfila skal senere brukes til interpolering i neste trinn, denne kalles derfor <interpoleringsfil>. Containerfila vil i tillegg til å inneholde transportpakker fra strømmen, inneholde et hode. Dette hodet er ikke identisk med hodet i den ferdige containeren, dette er som sagt et midlertidig format som skal brukes av det siste progammet.

Containerene vil konstrueres slik at dersom det finnes et aksesspunkt i strømmen vil dette alltid finnes i begynnelsen av en container (se konstruksjonen). Dette medfører at noen containere (de like før et aksesspunkt) ikke vil være like full som de andre. For enkelhets skyld vil de midlertidige containere som skrives til disk alltid ha en fast lengde, slutten av containeren vil da være uvesentlig data. Dette medfører veldig lite ekstra lagring, totalt sett.

Mellomformater

Følgende tabell beskriver containerformatet. Merk at siden PCR-verdien (basefeltet) består av 33 bit må vi ha 5 byte for å lagre denne verdien.

Innhold	Forklaring
Hodelengde (short)	Lengden på hodet (i antall byte).
Datalengde (short)	Lengden på dataene etter hodet (i antall byte).

Containeridentifikasjon (unsigned int)	Identifiserer en container. De nummereres fortløpende etterhvert som de genereres.
Random access (short)	Angir om det finnes et aksesspunkt i containeren. Dersom den ikke har det vil denne ha verdien -1.
PCR-byte1 (char)	Denne byten vil kun inneholde den øverste bit'en (bit 32) av PCR-verdien (som laveste bit i denne byten). En verdi på 255 angir at det ikke finnes noe PCR-verdi for denne containeren og at de andre PCR-bytene derfor skal ignoreres.
PCR-byte2 (char)	Denne byten inneholder bit'ene 24-31 av PTS-verdien.
PCR-byte3 (char)	Denne byten inneholder bit'ene 16-23 av PTS-verdien.
PCR-byte4 (char)	Denne byten inneholder bit'ene 8-15 av PTS-verdien.
PCR-byte5 (char)	Denne byten inneholder bit'ene 0-7 av PTS-verdien.
Data (transportpakkene)	Dette er dataene i containeren. Den vil inneholde et helt antall transportpakker.

Følgende tabell angir innholdet i interpoleringsfila for *en* container som inneholdt en PCR-verdi. Dette innholdet vil repeteres for hver container som inneholdt en PCR-verdi.

Innhold	**Forklaring**
containeridentifikasjon (unsigned int)	Identifiserer nummeret til den containeren som inneholder denne PCR-verdien.
PCR-verdi (unsigned int)	Dette er en tallverdi som er regnet om utifra byte 2-5 i PCR-feltet. Dette gjør at denne er egnet for interpolering. NB! Den øverste bit'en (bit 32) vil ikke komme med i beregningen, denne må hentes i tillegg fra containerfila dersom man vil bruke den.

Vi ser at vi har utelatt den øverste bit'en i PCR-feltet. Dette har vi gjort fordi PCR-feltet består av 33 bit, mens datatypen int (på denne plattformen) bruker 32 bit.

Klassebeskrivelser

Stream2Container:
Denne klassen leser inn og behandler en transportstrøm. Den vil bruke klassen IContainer til å generere midlertidige containere og skrive disse til fil. Funksjoner finnes for å hente ut tidstempler og informasjon om aksesspunkt fra strømmen. Det finnes logikk for å konstruere containerene slik at eventuelle aksesspunkt alltid vil finnes i den første pakken i containeren.
IContainer:
Denne klassen administrerer dataene i en container på mellomformatet. Verdier i hodet kan settes og transportpakkene kan kopieres inn. Klassen inneholder funksjoner både for å skrive containeren til fil samt lese den fra fil.

Con2Elv

Dette programmet fullfører prosessen med å overføre en transportstrøm til elviraformatet. Det vil lese inn de to filene generert av copac og vil bruke disse til å konstruere en elvirafil som legges inn i et elvira filsystem. (Se vedlegg 3 for en beskrivelse av filsystemet.) Bruken er som følger:

```
con2elv  <containerfil> <interpoleringsfil> <indexfil> (<retning>)
```

Containerfila og interpoleringsfila skal være de samme som ble generert av copac.

Con2elv genererer en indeksfil som sier hvor konstruerte segmenter ligger i filsystemet. Navnet på denne fila gies som en parameter. I tillegg vil det som beskrevet i konstruksjonen genereres en aksess-indeksfil, denne vil få samme navn som indeksfila, men med et tillegg «.acc». Se vedlegg 4 for en beskrivelse av disse filene. Programmet skriver nå ut en indeksfil for hver av disse for hver innlegging, uansett om det er en spolefil eller en

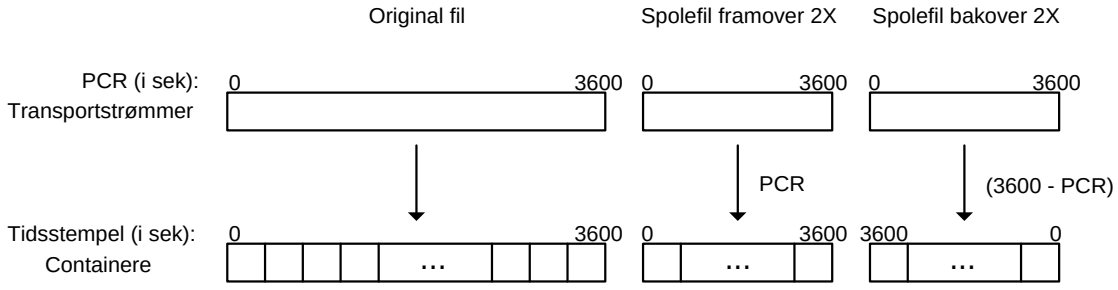
originalfil. Indeksfilene for en film skal inneholde både indeksene for originalfila og alle tilhørende spolefiler, dette er foreløpig en mangel ved denne implementasjonen (indeksene blir foreløpig satt sammen manuelt etterpå).

For å legge inn containerene i elviraformatet og generere indeksfila for disse blir klasser som er implementert utenfor dette prosjektet brukt. Disse klassene er generelle klasser for innleggelse av containere (uavhengig av filmformat) til en elvirafil, og er en integrert del av Elvira II. Aksess-indeksfila blir foreløpig generert helt på egen hånd av con2elv.

Tidsinformasjonen som legges i indeksfilene og i hodet på de ferdige containerene interpoleres for de containerene som ikke allerede har tidsstempler.

Som nevnt i konstruksjonen vil tidsstemplene måtte snues ved innleggelse av baklengs spolefiler. Ved innleggelse av en baklengs spolefil må dette angis ved å sette en parameter <retning>, her skriver man da «b». Standard retning er framover.

På grunn av den forenklede tidsstempelkonstruksjonen for spolefilene (i programmet mux) slipper vi nå å utføre noen multiplisering av tidsstemplene for spolefilene som beskrevet i konstruksjonen. Dette må derimot innføres dersom spolefiler med ordentlige PCR-verdier blir innført. Figur 49 viser hvordan dette gjøres nå, se Figur 45, side 90 for å sammenligne med konstruksjonen.



Figur 49 Konstruksjon av tidstempler (implementert versjon)

En del av informasjonen om filmen som legges i indeksfila blir foreløpig hardkodet, ved en senere utvidelse bør disse verdiene settes dynamisk med utgangspunkt i inndata. (Denne informasjonen er filmnavn, avspillingshastighet, rammerate og diskid.)

Klassebeskrivelse:

ElviraFile:
Denne klassen inneholder all logikk for innleggelsen av både spolefiler og originalfiler. Den inneholder funksjoner for innlesing av filer, interpolering, og indeksfil-generering. For innlesing av de midlertidige containerene vil også denne klassen benytte klassen IContainer som er beskrevet under copac. Den vil også benytte andre klasser i Elvira II som er laget utenfor dette prosjektet.

Målinger og resultater

I dette kapittelet vil det presenteres hvordan det implementerte systemet fungerte i praksis. Mye arbeid ble lagt ned underveis for å kontrollere at resultatene fra hvert delprogram var korrekte og kunne brukes videre. Nå vil vi kontrollere at hele systemet fungerer som det skal og gir brukbare resultater. Ytelsen til enkeltprogrammene har ikke vært prioritert under utviklingen, men vi vil foreta målinger og vurderinger både for ferdige programmer og egenutviklede slik at vi får et begrep om ting som tidsforbruk og diskforbruk.

Resultatene ble også integrert og testet i Elvira II slik at vi kunne se at metodene som er implementert er gyldige og fungerer som de skal. Vi beskriver senere i kapitlet hvordan dette ble gjort og foretar noen målinger som viser belastningen på disk og nettverk under kjøring.

Sakte avspilling er ikke implementert i dette prosjektet, spoling har vært prioritert. Vi har likevel foretatt en liten test av prinsippet, denne beskrives til slutt i kapittelet.

Utgangspunkt for testingen

For å kontrollere at dette systemet fungerer som det skal, har vi under hele denne prosessen brukt en transportstrøm kalt «np.m2t». Denne er hentet fra Internet og stammer fra Philips Research Laboratories Eindhoven. Strømmen er kodet med 8.0 Mbit/s, og inneholder en videostrøm som er kodet med 6.0 Mbit/s. Den inneholder ikke lyd. Filmen er på ca. 37 sekunder og størrelsen på fila er ca 37 MB.

Denne strømmen ble valgt fordi den var den eneste vi per dags dato kunne finne som hadde en høy bitrate og dermed høy bildekvalitet, samtidig som den var en relativ lang teststrøm. MPEG-2 transportstrømmer finnes det fremdeles ikke mange av rundt omkring på nettet. Det at den ikke hadde lyd gjorde ingenting, idet vi ikke skulle bruke lyd i spolefilene.

Til å spille av denne og spolefilene vi konstruerte har vi brukt databasegruppas MPEG-2 dekoder VideoPlex fra Optibase. Dette kortet kan spille av både elementærstrømmer og systemstrømmer for MPEG-1 og MPEG-2. Kortet har imidlertid ikke støtte for å lese en strøm fra et nettverk, den leser kun fra fil. Alle strømmer vi skulle teste måtte derfor være ferdiggenererte filer som lå på disk.

For å konstruere spolefiler av teststrømmen vår kjørte vi den igjennom mange steg. Disse inkluderer bruk av egenutviklet programvare samt ferdig programvare hentet fra nettet. Alle programmene kjører under et UNIX operativsystem. I denne testingen har vi brukt Axil320-maskiner med SunOS 5.5.1. Maskinene har en 125MHz HyperSparc prosessor og 128 MB minne.

Stegene er:

- Demultipleksing. **ts_demux** av Christos Tryfonas, University of California.
- Dekoding. **mpeg2decode** fra MPEG Software Simulation Group.
- Sortering. **wsort** (egenutviklet)
- Koding. **mpeg2encode** fra MPEG Software Simulation Group.
- Enkel transportstrømgenerering. **mux** (egenutviklet)
- Generering av Elvira II format:
    - **copac** (egenutviklet)
    - **con2elv** (egenutviklet)

I neste avsnitt skal vi se på ytelsen til disse programmene.

Ressursbruk

Det er hovedsakelig to aspekter man må se på når det gjelder ressursbruk for dette systemet. Det ene er: hvor lang *tid* tar hvert steg? Dette vil for det meste avhenge av CPU-belastning og diskaksessering. Det andre er: hvordan er *diskforbruket* for mellomresultatene for hvert steg?

For å vurdere disse aspektene har vi foretatt målinger for hvert steg i prosessen. For å måle tiden har vi brukt programmet *time* som måler tiden det har tatt og samtidig skriver ut hvor stor del av tiden som er brukt til CPU-operasjoner. Formatet på resultater er som følger:

```
<bruker> <kjerne> <total tid> <% CPU>
```

Første argument sier hvor lang tid (antall sekunder) som er brukt av CPU'en i brukermodus, dvs. kjøring av brukerdefinerte operasjoner i programmet. Det andre argumentet sier hvor lang tid den har brukt i kjernemodus, dette er utførelse av systemkall i operativsystemet. Disse to gir altså en indikasjon på hva slags type program som kjøres.

For å vurdere diskforbruk noterer vi oss størrelsen på filene generert fra hvert steg. Vi vil i det følgende likevel bare kommentere diskforbruket dersom det kan bli et problem. Dersom resultatfilene fra en program tar omtrent like stor plass som innparameter-filene, vil vi ikke vurdere det til å være et problem.

Som nevnt bruker vi en strøm på ca 37 sekunder for å teste systemet. I en fullstendig implementert videotjener vil det være naturlig å ha hele spillefilmer i systemet. Det vil derfor her være naturlig å ekstrapolere tallene vi får for testfilen slik at vi kan få en pekepinn på hvilke ressurser dette systemet vil kreve ved innleggelse av en film med lang lengde. Dette kan gjøres fordi tidsforbruket vil øke lineært med prosessert datamengde. Testfilen er en strøm kodet med 8 Mbit/s, det vil si den bruker 1MByte per sekund avspilt video. En slik bitrate vil ikke være usannsynlig for en vanlig spillefilm. Dersom vi går utifra samme bitrate vil en to timers film være i størrelsesorden 3600*2/37 = 194 ganger så stor. Denne faktoren kan vi runde av til 200, deretter kan vi bruke denne til å foreta en vurdering av de ulike verdiene vi kommer fram til. (Vi skal bare ha riktig størrelsesorden på tallene, vi trenger ikke å være nøyaktige her.)

Testresultatene ble som følger.

ts_demux

Tidsforbruk
Vi demultiplekset filen np.m2t på ca. 37 MB, *time* ga følgende resultat:

```
455.38u  6321.97s  2:03:00.43  91,8 %
```

Vi ser at dette programmet brukte over 2 timer på prosessen! Det er klart at dette er urimelig lang tid i forhold til oppgaven som utføres. Dersom dette programmet skulle ha demultiplekset en 2 timers spillefilm ville vi fått et tidsforbruk på flere hundre timer, noe som selvfølgelig er helt uakseptabelt. Hvorfor dette programmet bruker så lang tid på noe som en dekoder skal gjøre i sanntid er en gåte. Til vårt formål var likevel ikke dette noe problem, vi skulle jo bare pakke ut en elementærstrøm en gang. Denne programbiten bør imidlertid byttes ut med et mer effektivt program dersom systemet settes i større bruk.

mpeg2decode

Tidsforbruk
Vi dekodet hele video-elementærstrømmen og fikk følgende tidsforbruk:

```
1848.94u  118.25s  41:45.54  78,5 %
```

Vi ser at det tok i overkant av 40 minutter å dekode 37 sekunder med video, og å lagre bildene på PPM format. Dersom programmet skulle ha vært brukt på en to timers film ville dekodingen ha tatt ca. (40 * 200)/60 = 133 timer = 5,5 dager. Vi ser at det vil være en svært tidkrevende prosess, men vil utføres kun en gang for hver film.

Diskforbruk
Videoen inneholdt 938 enkeltbilder. Hvert enkeltbilde i denne videoen ble lagret på PPM format og tok ca 1.2 MB. Dermed tok alle bildene tilsammen over 1GB. Å lagre alle bildene for en 2 timers film vil dermed kunne komme til å ta 200 GB, noe som er en relativt stor mengde data. Problemet kan reduseres noe ved å bruke et annet bildeformat enn ppm. Koderen vi senere brukte kunne kun ta PPM eller YUV-formatet, og siden YUV lagrer bildene i tre komponenter, valgte vi å bruke PPM for enkelhets skyld. YUV-formatet vil bruke ca halvparten så stor diskplass, så det bør i framtiden vurderes å utvide systemet slik at YUV også kan brukes.

wsort

Tidsforbruk
Vi har testet dette programmet ved å utføre følgende oppgave: snu rekkefølgen på alle bildene og plukk ut hvert fjerde bilde. (Lag en baklengs spolefil på fire ganger normal hastighet). Antall bilder før reduksjon var 938 og tidsforbruket ble:

```
0.18u  12.10s  1:52.43  10,9 %
```

For å gjennomføre samme operasjon uten å snu bildene (forlengs spolefil) fikk vi følgende resultat:

```
0.11u  4.00s  0:52.11  7,8 %
```

Vi ser at dette tar i underkant av ett eller to minutter, avhengig av retning på spolefilen. Sammenlignet med de andre operasjonene ser vi at denne operasjonen tar svært liten tid. Vi ser forøvrig at bare rundt 10 % av tiden ble brukt til CPU-operasjoner, mye tid ble brukt til diskaksesser noe som er naturlig siden vi skifter navn på en mengde filer.

Diskforbruk
Ved denne metoden reduserer vi antall bilder i katalogen, og gjør at vi bruker *mindre* diskplass. Men på grunn av akkurat dette kan det være en fordel å ha ekstra kopier av filene slik at vi slipper å dekode filmen på nytt om vi skulle ønske å lage nye spolefiler med lavere hastighet. Ved å ta slike kopier vil den totale nødvendige lagringskapasiteten under og etter denne prosessen bli opptil dobbelt så stor. En bedre metode er derfor å være sikker på spolehastighetene man vil ha, og produsere disse i stigende rekkefølge. En mer kompleks utgave av wsort vil kunne gjøre prosessen reversibel uten bruk av kopiering av filer, og bør derfor vurderes å lages i framtiden.

mpeg2encode

Tidsforbruk
Tiden det tar å kode en spolefil vil være avhenging av spolehastigheten, eller mer direkte: antall bilder som skal kodes. Vi skulle lage en spolefil på 4 ganger normal hastighet, wsort ga oss dermed 236 bilder å kode. Kvaliteten på spolefilen vil også innvirke på hastigheten, vi reduserte kvaliteten noe ved å sette bitraten til 5Mbit/s for denne filmen. *Time* ga oss følgende måling:

```
2353.90u  24.63s  40:26.96  98 %
```

Dette tok ca like lang tid som det tok å dekode 4 ganger så mange bilder, vi ser at koding er tyngre enn dekoding. Siden det i tillegg er naturlig å lage mange spolefiler slik at vi for en film kan tilby mange spolehastigheter, vil dette totalt være en langt større oppgave enn hva dekodingen var. Vi kan regne ut hvor lang tid det ville tatt å lage en baklengs fil i normal hastighet: 40 * 4 = 160 minutter. For en to timers film får vi da 160*200/60 = 533 timer = 22 dager! I tillegg bør vi altså lage flere spolehastigheter. Vi ser at dette er en uakseptabel lang tid.

mux

Tidsforbruk

Denne enkle transportstrømgeneratoren er ganske effektiv. Etter å ha pakket inn spolefilen vi kodet i forrige trinn ga time følgende resultat:

```
2.24u  1.46s  0:04.19  88,3 %
```

Det tok altså i overkant av fire sekunder å generere en transportstrøm ut av en elementærstrøm på ca 6MB. Denne tiden blir ubetydelig sammenlignet med dekoding og koding.

copac:

Tidsforbruk
Til dette programmet brukte vi originalfila som testfil. (37MB). Tidsforbruket ved å produsere midlertidige containere og en interpoleringsfil ble som følger:

```
4.48u  9.24s  0:27.59  49,7 %
```

Dette tok altså ca. 28 sekunder, dette går relativt fort.

con2elv

Tidsforbruk
Programmet tok containerene og interpoleringsfila laget av copac (fra originalfila) og la disse inn i en Elvirafil (på samme harddisk der innfilene lå slik at vi slapp å overføre dataene over nettet) *Time* ga følgende resultat:

```
17.90u  12.73s  1:23.87  36,5 %
```

Dette tok nesten ett og et halvt minutt noe som også er ganske raskt.

Konklusjon

Vi ser at for konstruksjon av spolefiler, som krever de fem øverste trinnene i tillegg til de to siste, er det dekoding og spesielt koding som tar mesteparten av den totale tiden. Selv om man vil kunne kjøre programmene på kraftigere prosessorer enn hva som er gjort her, vil disse to trinnene kunne ta mange dager. Konklusjonen er at software-dekodere og kodere ikke er egnet til bruk av produksjon av spolefiler for lange filmer med høy bitrate. Dersom et slikt system skal brukes til å legge inn slike filmer, må disse trinnene erstattes med hardware-produkter.

Det kan være interessant å se hvordan dette systemet ville ha klart oppgaven dersom vi ser bort ifra dekoding og kodingsprosessen. (dvs. antar at vi har hardware for disse). Vi ser også bort ifra det ekstremt ueffektive demultiplekserprogrammet. Vi kan summere tidene for noen operasjoner og multipliserer med 200 slik at vi får en total tid for en to timers lang film:

Å legge inn en originalfil: (copac + con2elv) = (28 + 83)* 200/60 = 370 minutter = **6 timer.**

Å lage en baklengs spolefil med 4 ganger hastighetsøkning:
(wsort + mux + copac + con2elv) = (112 + 4 + 28/4 + 83/4) * 200/60 = 479 minutter = **8 timer.**

Å lage en forlengs spolefil med 4 ganger hastighetsøkning:
(wsort + mux + copac + con2elv) = (52 + 4 + 28/4 + 83/4) * 200/60 = 279 minutter = **4,6 timer.**

Vi ser at en forlengs spolefil kan produseres en god del raskere, disse slipper man å snu rekkefølgen på. Her må det nevnes at wsort i dag er relativt ueffektiv, ved å effektivisere denne kan vi oppnå store forbedringer. For spolefiler må det selvsagt legges til tid for demultipleksing og hardware-dekoding/koding, dersom disse stegene gjøres i sann tid vil de ta tilsammen (2*3) = 6 timer. Likevel så viser disse beregningene at de egenutviklete programmene ikke forhindrer et tidforbruk som er til å leve med.

Når det gjelder det store diskforbruket for de dekodede enkeltbildene så er det klart at noe må gjøres dersom man vil dekode lange filmer. En ting som kan vurderes er å innføre bruk av tapestreamere som direkte lagringsmedium, dvs man skriver direkte til tape ved dekoding, og leser ved koding.

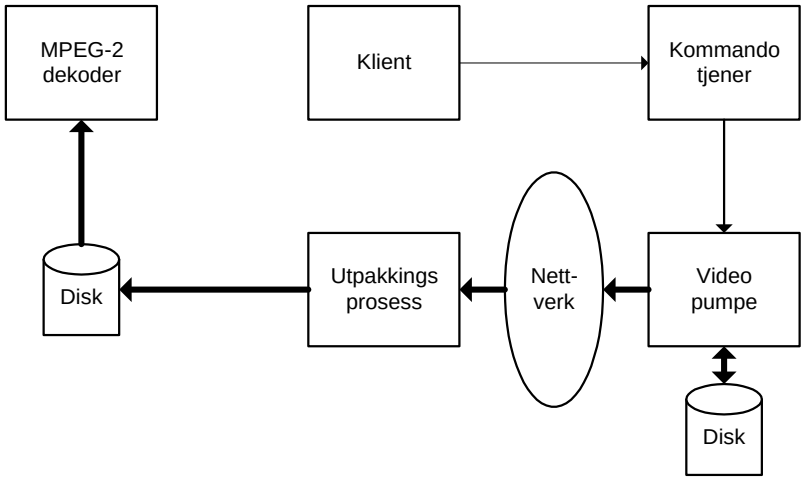
Testkjøring under Elvira II

Vi produserte to spolefiler for testing, en for baklengs spoling og en for forlengs. Begge hadde en hastighet på 4 ganger normalhastigheten, og var kodet med en bitrate på 5 Mbit/s (den originale var på 6 Mbit/s). Dette ble gjort for å teste hvordan en slik reduksjon i bitraten ville gi utslag for bildekvalitet og ressursbelastningene. Selv om originalfilen var interlacet, kodet vi spolefilene med vanlig progressiv koding.

Det første som ble gjort etter at spolefilene var konstruert var naturligvis å teste de hver for seg på MPEG-2 kortet for å kontrollere at de virkelig ga en korrekt spoleeffekt og at filmene så bra ut. Dette var vellykket, og neste trinn var nå å teste fleksibel avspilling i Elvira II. De ble derfor deretter lagt inn i en elvirafil sammen med originalfila. Indekser ble produsert for hver av filene. Programmet con2elv lager en indeks for hver gang den legger inn en avspillingsfil, men setter ikke disse sammen etterhvert til en fil for hver film, slik som Elvira II skal ha indeksen. Vi måtte derfor manuelt sette sammen de tre indeksene til en indeksfil.

Elvira II støtter ennå ikke bruk av aksess-indeksfila, de aktuelle verdiene i disse måtte derfor leses manuelt og legges inn i klienten. Klienten ville da vite hvilken tidspunkt den skulle spørre etter ved innlesing av ny avspillingsfil.

Som nevnt kunne ikke MPEG-2 kortet lese fra et nettverk, og vi kunne derfor ikke teste leveransen fra tjeneren direkte. Videoen ble derfor istedet sendt til en prosess som tok imot containerene, pakket ut dataene og la disse på fil. Deretter kunne vi spille av denne fila på MPEG-2 kortet. Denne prosessen er vist i Figur 50.



Figur 50 Testoppsett

Visuell kvalitet

Hvordan ser så den leverte videoen ut? Det er tre ting som bør vurderes: avspilling, overgang til et aksesspunkt, og tidssynkroniseringen ved overgangen.

Avspilling:

Blir videoen (normal og spoling) spilt slik den skal? Dette så meget bra ut. Det hendte at vi fikk noen forstyrrelser i bildet fordi en og annen container forsvant i forbindelse med oversendelsen, men dette er tjenerens feil. Spolekvaliteten var bra, selv om vi har fjernet ¾ deler av alle bildene ser det meget bra ut. Vi kan se at bildekvaliteten er noe dårligere, vi kan skimte grenser mellom makroblokkene. Likevel vurderer vi kvaliteten til å være meget bra. Overgangen mellom interlacet og progressiv koding går også helt fint, varierende formater skapte ingen problemer.

Overgang til aksesspunkt
Hvordan ser overgangen ut når vi skifter fra en avspillingshastighet til en annen? Containerene er laget slik at for de som inneholder et aksesspunkt så skal dette bestandig forekomme i den første pakken i containeren. Derfor skal det etter overgangen begynnes rett på et aksesspunkt, og vi bør få en bra overgang uten merkelige effekter. Dette fungerte meget bra, vi fikk fine overganger, og det virker som om denne metoden har vært en fornuftig framgangsmåte.

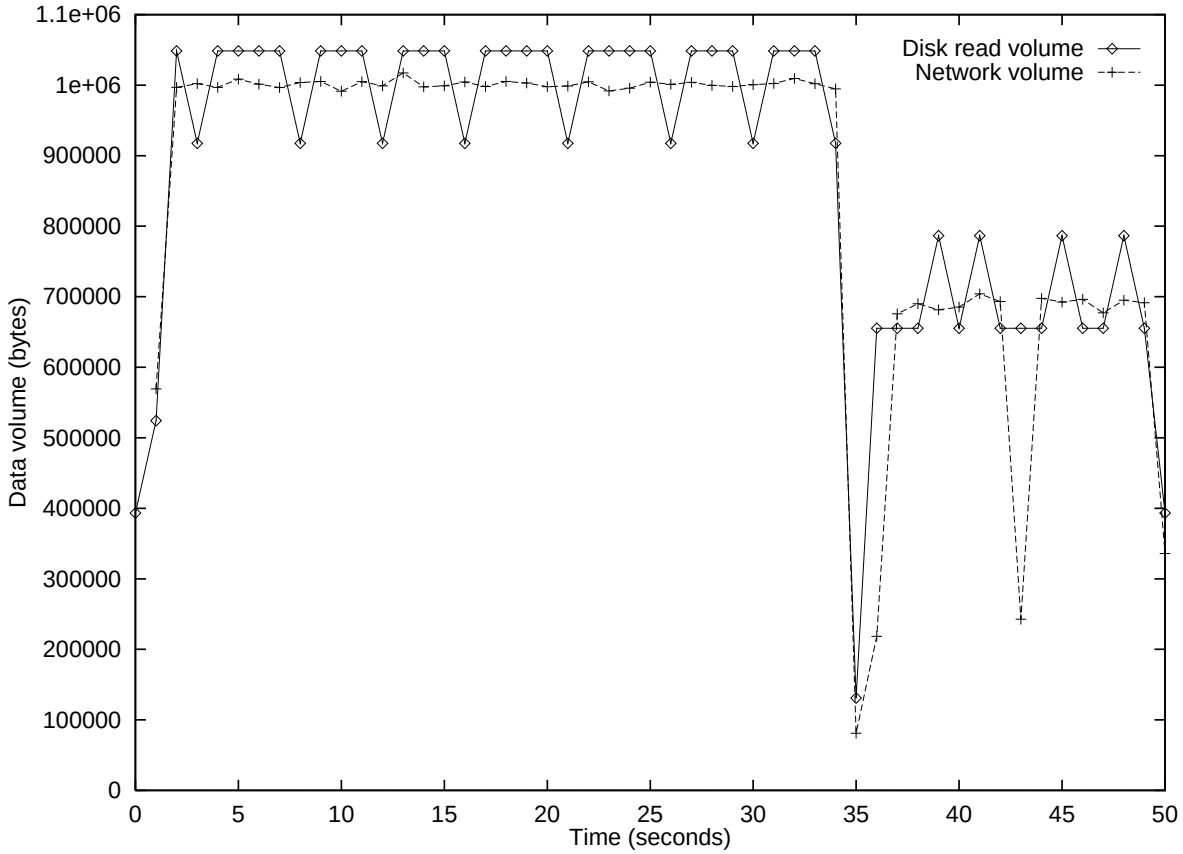
Tidssynkronisering ved overgang
Havner vi på tilnærmet samme sted i filmen etter et skifte i avspillingshastighet? Tjeneren skal finne fram til den riktige containeren ved å finne det nærmeste tidspunktet i aksess-indeksfila, dette ble som sagt gjort manuelt. En viss unøyaktighet vil finne sted siden vi ikke har så mange aksesspunkt for spolefilene som for normalfila. Vi erfarte at denne tidssynkroniseringen fungerte ganske bra. Noen ganger hadde vi litt mer flaks enn andre, men det ble aldri dårlige resultat. Den relativt høye spolehastigheten gjorde at det var vanskelig å se de tidsforskyvningene som forekom.

Ytelsesmålinger

Det ble under levering av videoen også foretatt noen ytelsesmålinger av diskbelastningen og nettverksbelastningen under kjøringen av videoleveringen. Det som er interessant å se er hvordan den varierer ved forskjellige avspillingshastigheter, klarer den å spille av spolefilene uten å øke belastningen nevneverdig?

Det ble nå sendt kommandoer til tjeneren om å spille av filmen med normal hastighet i 34 sekunder, deretter ble det sendt kommando om å spole tilbake i 7 sekunder, for deretter å spole framover i 7 sekunder igjen.

Som tidligere vist på Figur 50 sender nå videopumpa den etterspurte videoen over nettverket. Vi målte nå nettverksbelastningen og diskbelastningen som ble påført ressursene. Disse målingene er vist i Figur 51, her er begge deler vist i samme figur som en funksjon av tiden (Den kraftige reduseringen av ressursbruk i et kort tidsrom ved skifte av avspillingshastighet kommer av at Elvira II foreløpig legger inn en pause ved slike skifter).



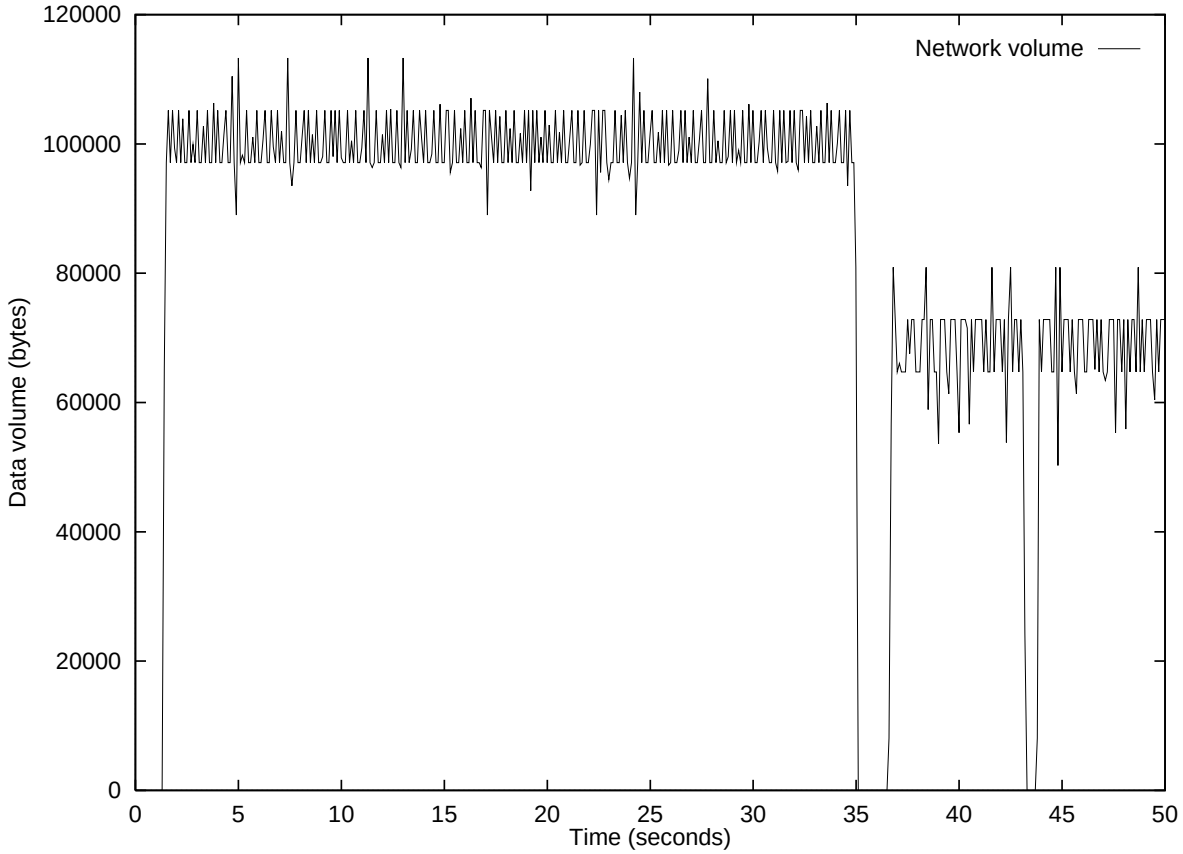
Figur 51 Disk og nettverksbelastning per sekund

Diagrammet viser datamengden som er hentet fra disk og sendt til nettverk per sekund. Vi ser at de to linjene følger hverandre relativt bra, noe som er forventet siden dataene sendes direkte ut på nettet fra videopumpa.

Vi ser at for spolingen blir belastningen på både disk og nettverk faktisk redusert. Også dette er forventet siden vi i tillegg til å ha redusert antall bilder i spolefilene også reduserer kvaliteten på bildene. I tillegg inneholder den originale strømmen en del tomme pakker som gjør at denne er større en den hva som hadde vært nødvendig for å inneholde videostrømmen. (Strømmen er på 8 Mbit/s, mens videoen bare er på 6 Mbit/s.) Transportstrømmen for spolefilene vil ikke inneholde like mye overflødig informasjon.

Vi ser at originalfila ligger på omtrent 1.000.000 bytes i sekundet, dette gir som ventet en bitrate på 8Mbit/s. Spolefilene ligger på ca. 700.000 bytes i sekundet, noe som blir 5,6 Mbit/s. Dette viser at belastningen ville ha blitt redusert selv om den originale strømmen ikke hadde vært unødvendig stor, bare den originale videostrømmen er jo på 6.0 Mbit/s.

Av disse målingene kan vi konkludere med at dersom transportstrømmen genereres på samme måte for originalfila og spolefilene samtidig som bitraten for videoen er lik, så vil den totale bitraten bli tilnærmet lik. Siden spolefilene aldri vil inneholde unødvendig eller ekstra informasjon og vi kan redusere bitraten for videoen som vi vil, vil vi alltid kunne produsere spolefiler som gir en lavere total bitrate enn originalfila. Konklusjonen blir at bruk av spolefiler er en metode som alltid kan sikre fornuftig ressursbruk.



Figur 52 Nettverksbelastning

Figur 52 viser en mer detaljert framstilling av nettverksbelastningen. Dette diagrammet viser belastningen per tidels sekund. Her ser vi tydeligere at videopumpa tar pause i et kort tidsrom ved skifte av avspillingshastighet.

Ellers viser dette diagrammet det samme som Figur 51, men her ser vi tydeligere at belastningen fordeler seg noenlunde jevnt og at det ikke blir noen høye topper for belastningen.

Sakte avspilling

Elvira II kan sende ut containere i en hastighet som tilsvarer sakte avspilling. Dette gjør at klienten/dekoderen får dataene i den hastigheten den bør spille av med. Siden vi ikke har et dekoderkort som kan lese fra nettverk har vi ikke fått testet hvordan dette vil fungere i praksis for MPEG-2. Kortet kan naturligvis ikke spille av videoen fortere enn hva den mottar den, men avspillingen vil sannsynligvis bli dårlig visuelt sett. Dette bør derfor testes ut når det blir muligheter for det.

Som beskrevet i konstruksjonen er det mulig å senke hastigheten ved å endre på tidsstemplene i strømmen. Dersom den ovennevnte metoden ikke gir gode resultater vil dette kunne bøte på det visuelle. Når tidsstemplene blir modifisert vil dekoderen spille av rammene etter disse, og en jevn avspillingshastighet vil kunne oppnås.

Vi har for å teste denne metoden utført dette ved hjelp av preprosessering; vi har modifisert på en transportstrøm på forhånd for deretter å spille av denne på MPEG-2 kortet.

Tidsstemplene som ble modifisert var PCR, PTS, og DTS. Disse ble multiplisert med en faktor for å oppnå en hastighetssenking proporsjonal med faktoren. Vi lyktes i å få denne hastighetssenkingen. Kvaliteten på avspillingen var grei, men vi fikk ikke noen perfekt avspilling, bildene beveget seg noe «rykkete». Dette kommer høyst sannsynlig av at denne strømmen var interlacet, for vanlig progressiv video ville vi sannsynligvis fått en helt fin avspilling.

Mer eksperimentering bør gjøres på dette området både for interlacet og progressiv video for å finne ut om dette er en god nok metode.

Konklusjon

I dette kapittelet gis det en konklusjon på arbeidet som er utført i denne hovedoppgaven. Først vil vi gi en totalvurdering av hele systemet som er utviklet. Her ser vi på hvordan systemet fungerer som helhet, og om målsettingen som vi ga i problemformuleringen er nådd.

Systemet slik det er implementert i dag er ikke fullstendig, det finnes rom for forbedringer og utvidelser på flere punkter. Vi vil gi en beskrivelse av disse mulighetene her, slik at det skal bli enklere å videreutvikle systemet i framtiden.

Til slutt gis det en generell oppsummering av hele prosjektet med tanke på oppgavetypen, arbeidsprosessen og erfaringer i forbindelse med arbeidet.

Vurdering av resultatet

Systemet som er utviklet består hovedsakelig av to hoveddeler. Den første delen består av flere programmer som konstruerer spolefiler. Den andre delen består av to programmer som legger filer inn i Elvira II formatet.

Funksjonaliteten til systemet som helhet er:

- Det kan konstrueres spolefiler for alle hastigheter større enn normal avspillingshastighet.
- Det kan konstrueres spolefiler for både forlengs og baklengs avspilling.
- Spolefilene vil støtte tilfeldig aksess, dvs. de inneholder aksesspunkt som kan identifiseres.
- Originalfila og spolefilene kan legges inn i et Elvira II filsystem.
- Det genereres indeksfiler som kan brukes til å lokalisere filmen i Elvira II filsystemet.
- Det genereres en aksess-indeksfil slik at tilfeldig aksess kan gjøres i både originalfila og i spolefiler.

Denne funksjonaliteten er testet i Elvira II med vellykket resultat. Tjeneren leste både originalfila og spolefilene fra filsystemet under kjøring og containerene ble sendt over et nettverk. Siden vi ikke hadde tilgang til en dekoder som kunne lese fra nettverk, skrev vi innholdet til fil og spilte av denne etterpå. Ved avspilling av denne fikk vi et meget bra resultat. Vi kan derfor konkludere med at metodene for å oppnå spoling har fungert bra.

Det er to problemer med spolefilkonstruksjonen slik den foregår i dette systemet. Det ene er at det tar for lang tid. Prosessen innebærer dekoding og koding av store mengder data, og dette blir gjort ved hjelp av programvare. Det bør derfor brukes maskinvare for disse delene av prosessen i framtiden. Det andre problemet er at enkeltbildene som produsereres ved dekoding av en film vil ta meget stor plass. Det vil derfor være nødvendig å ha tilgang til spesiell stor lagringskapasitet.

Det er ikke implementert en ferdig løsning for sakte avspilling i Elvira II. Det er likevel foreslått en metode for dette (modifisering av tidsstempel) og implementert funksjoner som senere kan brukes i Elvira II for å oppnå dette. Prinsippet er testet ved å spille av filer der alle tidsstemplene er modifisert og det virker som om dette kan være en lovende metode.

Videre arbeid

Det finnes mange måter å forbedre dette systemet på. Her skal vi sette opp noen punkter og komme med anbefalinger til hvordan dette best kan gjøres. De nevnes her i noenlunde prioritert rekkefølge når det gjelder viktigheten av utvidelsene.

- **Multipleksing**

Det bør innføres en profesjonell multiplekser på trinnet som genererer en transportstrøm fra en elementærstrøm. Nå brukes det et egenutviklet program som gjør en rekke forenklinger. Den viktigste mangelen er at tidsstempler ikke blir konstruert på en skikkelig måte. PCR-verdier blir ikke satt på en skikkelig måte, disse blir kunstig generert utifra forhåndinformasjon om hvor lang originalfilmen er. PTS/DTS-stempler blir ikke satt i det hele tatt. MPEG-2 kortet vårt klarer likevel å spille filmen fint, vi regner med at den bruker rammeraten til å styre hastigheten. Det er derimot ikke gitt at alle dekodere vil godta dette. Merk at innføring av skikkelige PCR-verdier vil føre til at behandlingen av disse må justeres ved innlegging i elviraformatet (i programmet con2elv), slik at det gjøres som beskrevet i konstruksjonen. Slik det gjøres nå slipper vi å multiplisere med en faktor siden de kunstige tidsstemplene i strømmen allerede gjenspeiler originalfila. Det er også verdt å nevne at vår egenutviklede generering av transportstrøm på underveis kopierer sekvenshodet og plasserer dette foran hver Group-of-pictures (GOP) slik at tilfeldig aksess er mulig. Dersom denne fjernes bør denne delen av programmet trekkes ut og likevel utføres før multipleksingen. Eventuelt kan koderen byttes ut med en koder som legger inn sekvenshoder for hver GOP.

- **Sakte avspilling**

Eksperimentering med bruk av sakte avspilling i en reell situasjon bør utføres. Dette kan ikke gjøres uten å ha tilgang til en dekoder som kan lese direkte fra et nettverk. Det er mulig at dette kan implementeres ganske enkelt ved å kun sende containerene med lavere hastighet til dekoderen. Den kan jo likevel ikke spille av filmen fortere enn den mottar dataene. Hvordan avspillingen ved en slik løsning vil bli er uklart, sannsynligvis vil det være nødvendig å benytte metoden med å endre på tidsstemplene for å få en pen avspilling. Innføring av «trick mode» for sakte avspilling bør også vurderes, siden vi i disse feltene kan sette en verdi for hvor mange ganger en ramme (field for interlaced video) skal gjentas (field_rep_cntr). Problemet med dette er at disse feltene ikke finnes i en vanlig strøm, og man kan ikke sette inn ekstra felt i en strøm, dette vil ødelegge strømmen. For bruk av «trick mode» må det derfor konstrueres nye filer.

- **Sammenslåing av indeksgenerering**

Programmet con2elv genererer idag en ny indeksfil for hver gjennomkjøring. Idet indeksene for en film og alle dens spolefiler skal ligge i samme fil må denne idag konstrueres ved å manuelt sette disse sammen. Programmet bør utvides slik at disse indeksfilene automatisk blir laget i sin helhet. Dette gjøres ved å legge indeksen til den originale indeksfila istedet for å lage en ny for hver innleggelse av en spolefil. Dette bør være en meget enkel oppgave.

- **Bedre tidssynkronisering**

For å utføre hopp mellom originalfiler og spolefiler blir det generert aksess-indeksfiler som inneholder tidspunktene for gyldige aksesspunkt i filene. For spolefilene blir disse tidspunktene satt ved å ganske enkelt hente ut det kunstige tidstempelet i spolefilen i forbindelse med aksesspunktet. Dette fungerer bra for en så liten fil vi har eksperimentert med i denne oppgaven, men etterhvert som filmen blir lengre vil det kunne oppstå større og større unøyaktigheter på grunn av de kunstige tidsstemplene. Dette kan tenkes løst ved å innføre noen «sjekkpunkter» i strømmen med en viss avstand slik at tidsstemplene i containerene kan synkronisere seg inn igjen. Disse sjekkpunktene kan brukes slik at en ramme i originalstrømmen vil korrespondere med en ramme i spolefila, og man vil sette samme tidspunkt i containerene som inneholder disse rammene.

- **Skifte av bildeformat for dekoding**

Et problem ved det nåværende systemet er at de enkeltkodede bildene tar veldig stor plass. Formatet som nå blir brukt (PPM) komprimerer ikke bildene spesielt godt. Eksperimenter viser at formatet YUV tar ca. halvparten så stor plass, og det bør derfor vurderes å skifte til dette formatet. Bakdelen med dette formatet er at bildene blir lagret i tre komponenter. Sortering for bruk i spolefil kompliseres derfor litt, men ikke mer enn at dette burde la seg gjennomføre uten for mye arbeid.

- **Utvidelse av wsort**

Wsort er programmet som reduserer / sorterer bildene før koding av spolefil. Bakdelen med denne er at når den reduserer bilder ikke kan gjenopprette de fjernede bildene. Prosessen er derfor ikke reversibel, enkeltbildene må enten kopieres tilbake eller dekodes på nytt dersom det skal lages en ny spolefil som skal inneholde flere bilder enn den forrige. Ved å utvide wsort kan navneskiftingen gjøres på en måte som gjør at vi etter koding kan gjenopprette den opprinnelige rekkefølgen på filene.

- **Effektivisering av programmene**

Ingen av programmene i dette systemet er laget med tanke på at de skal være spesielt effektive. På grunn av tidspress og det faktum at vi jobber med små filer har det vært langt viktigere å tenke på korrektheten. Ved bruk

av større filer i et fullskala-system bør man tenke på å effektivisere progammene. Spesielt kan det være aktuelt å lage større bufre ved lesing og skriving fra/til disk.

Oppsummering

Jeg synes dette har vært en spennende og utfordrende oppgave å jobbe med. Oppgaven har krevd mye arbeid, men ikke mer enn at det var gjennomførbart i denne tidsrammen. Arbeidet har i tillegg til å skrive denne rapporten bestått i studering av kilder (ikke minst MPEG-2 spesifikasjonen), programmering, og leting/testing av verktøy som kunne brukes. MPEG-2 kortet til databasegruppa har også vært flittig brukt underveis for å teste konstruerte filer. Jeg synes dette har vært en fin og variert blanding av arbeidsoppgaver, og føler at jeg har lært mye på mange områder under dette prosjektet.

Samarbeidet med databasegruppa synes jeg har fungert meget bra. Jeg har hatt ukentlige møter med faglærer Roger Midtstraum og veileder Olav Sandstå og har kontinuerlig fått råd og veiledning. De har også vært behjelpelige med å kommentere foreløpige utkast til arbeid. I tillegg har Olav vært svært behjelpelig i arbeidet med å integrere og teste filmene i videotjeneren Elvira II. De fortjener en stor takk for innsatsen.

Referanser

Litteratur:

ISO/IEC 13818 - Information technology - Generic coding of moving pictures and associated audio information:

Består av:
ISO/IEC 13818-1 Systems
ISO/IEC 13818-2 Video
ISO/IEC 13818-3 Audio
ISO/IEC 13818-6 Extensions for DSM-CC

Stein Langørgen:
Ekperimentell videotjener for ATM.
Institutt for datateknikk, NTNU, 1994

Roger Koteng:
Videotjener med VCR-funksjonalitet for MPEG-1.
Institutt for datateknikk, NTNU, 1996.

Olav Sandstå, Stein Langørgen og Roger Midtstraum:
Video Server on an ATM Connected Cluster of Workstations
Institutt for datateknikk og informasjonsvitenskap, NTNU, 1997.

Daniel Remmem / Dag Helge Østerhagen
MPEG2 & Elvira II
Institutt for datateknikk og informasjonsvitenskap, NTNU, 1997

Eurescom, 1997
Artikkel hentet fra Eurescom P617-D3, s 33-40, 1997.

Bolosky, Barrera, Draves, Fitzgerald, Gibson, Jones, Levi, Myhrvold, Rashid.
The Tiger Video Fileserver
Microsoft Research, 1996

A. Murat Tekalp:
Digital Video Processing.
Prentice Hall, 1995.

F. Fluckiger:
Understanding networked multimedia.
Prentice Hall, 1995.

Foley, van Dam, Feiner, Hughes:
Computer Graphics. Principles and Practice.
2nd Edition, Addison Wesley 1993.

D. E. McDysan and D.L. Spohn:
ATM - Theory and Application.
McGraw-Hill, 1995.

W. Richard Stevens:
TCP/IP Illustrated, Volume 1 - The Protocols.
Addison Wesley, 1994.

Christos Tryfonas:
MPEG-2 Transport over ATM Networks.
University of California, September 1996.

Karlheinz Brandenburg og Gerhard Stoll:
ISO-MPEG-1 Audio: A generic standard for coding of high quality digital audio
Journal of the Audio Engineering Society, Oktober 1994.

V.Balabanian m.fl:
An introduction to Digital Storage Media - Command and Control.
IEEE Comm Mag, November 1996.

World Wide Web (WWW):

Overview of the Elvira 2 command server - Olav Sandstå
http://www.idi.ntnu.no/olavsa/elvira2/cs.ps

Sun MediaCenter Servers - Sun, 1997
http://www.sun.com/products-n-solutions/hw/servers/smc_external.html

MPEG Pointers and Resources - Tristan Savatier.
http://www.mpeg.org

MPEG-2 Archive - Phade software.
http://www.mpeg2.de

A beginners guide for MPEG-2 standard - Victor Lo.
http://www.ee.cityu.edu.hk/~edap064/mpeg/BeginnersGuideForMPEG2Standard.html

Indeo video - Intel corporation.
http://www.intel.com/pc-supp/multimed/indeo/

Distributed multimedia - Jon Crowcroft.
http://www-dept.cs.ucl.ac.uk/staff/jon/dummy/

MPEG Moving Pictures Expert Group Information - Luigi Filippini.
http://www.vol.it/MPEG/

MPEG Audio compression technology - Philips.
http://www.philips.com/sv/newtech/mpeg/

MPEG 2 - the compatible multichannel sound system - Philips.
http://www.km.philips.com/bumd/mpeg/mpeg2sur.htm

Sub-Band Coding - Otolith.
http://www.otolith.com/pub/u/howitt/sbc.tutorial.html

The QuickTime FAQ - Charles Wiltgen.
http://www.QuickTimeFAQ.org

Vedlegg 1: Kildekode

I dette vedlegget presenteres kildekoden for programmene utviklet. Først presenteres de to programmene som overfører MPEG-2 filer til Elvira II formatet, copac og con2elv. Deretter presenteres programmene wsort og mux, disse blir brukt i forbindelse med spolefilkonstruksjonen. Til slutt kommer en klasse TPacket, denne er ikke en integrert del av systemet, men tas med idet den kan være nyttig ved fremtidige utvidelser av systemet.

Copac

def.h

```
// Used by both copac and con2elv

#ifndef _DEFM2_H
#define _DEFM2_H

const short  TP_SIZE        = 188;             // Transport packet size
const short  TP_NUMBER      = 20;              // Number of transportpackets in buffer
const short  TP_BUFFER_SIZE = TP_SIZE * TP_NUMBER;  // Total buffer size.

const double PCR_TO_MSEC    = 90;         // Relation PCR value / 90 -> Milliseconds.
const int    SEGMENT_SIZE   = 128 * 1024; // The size of segments in Elvira II.
const int    CONT_DATASIZE  = 8*1024;     // max size of containers.

#define VIDEONAME "Pooltime"

#define MP2ELVIRADISK "/home/baldrick/d/danielr/filsys/disk.1" // temporary...

#endif _DEFM2_H
```

copac.c

```
// Main program for running the copac application

#include <iostream.h>
#include "str2cont.h"

void main(int argc, char *argv[])
{
   if (argc != 4) {
      printf("Transforms MPEG-2 stream to intermediary Elvira 2 format.\n");
           printf("Use con2elv on output.\n\n");
           printf("Syntax: copac <in_filename> <out_filename> <out_interfilename>\n");
      exit(-1);
   }

   Stream2Container copac(argv[1],argv[2],argv[3]);

   printf("\nStarting container-packer program (copac).\n");

   if (copac.Transform())
     printf ("\nCopac has finished, everything went well!\n");

   else
     printf ("\nError in transformation process!!!!\n");

}
```

str2cont.h

```
// This class contains functions responsible for the making of
// the containerformat described by the class Container.

#ifndef _STREAM2CONT_H
#define _STREAM2CONT_H

#include "container.h"
#include "../def.h"
#include <fstream.h>


// Transport packet type values: None, Adaptation Field, Unit Start, Adaptation Field and Unit
Start
enum TP_TYPE {NONE, AF, US, AF_US};
// Stream type values for PES layer: Other, video/audio;
enum STREAM_TYPE {OTHER, AUDIO,VIDEO};


class Stream2Container {
private:

   // Variables

   long  m_total_TP;               // Total number of Transport packets.
   long  m_read_TP;                // Indicates current tp's read from buffer.
   long  m_count_TP;               // Current tp's processed.
   short m_containercounter;       // Current position in current container.
   int   m_numberPCR;              // Total number of PCR's in file.
   bool  m_random;                 // Have we found a RAI?
   PCR_TYPE m_nextpcr;             // Store pcr-value for next container

   IContainer *m_current_container;  // Container currently being processed.
   char m_TP_buffer[TP_BUFFER_SIZE]; // The buffer which contains the packets

   FILE *m_infile;
   FILE *m_outfile;
   FILE *m_interfile;

   // Methods

   // Returns number of packets in buffer
   long  FillTPBuffer(void);
   // TP_pointer points to transportpacket number in TP_buffer
   // E.G : Value of 3 equals third transport packet.
   TP_TYPE GetTPType(long TP_pointer);

   // Updates container head with values from AF
   bool  UpdateContainerAF(long TP_pointer);
   // Updates container head with values from PES header
   bool  UpdateContainerPES(long TP_pointer, short PES_start);
   // Used to find start of PES packet
   short GetAFLength(long TP_pointer);
   STREAM_TYPE GetSTREAMType(long TP_pointer, short PES_start);
   int GetPID(int offset);

public:

   // Initializes filenames
   Stream2Container(char *in_filename,char *out_filename,char *out_internname);
   ~Stream2Container();

   bool Transform();   // Transforms a MPEG2 file/stream into EII containers

};
#endif _STREAM2CONT_H
```

str2cont.c

```
#include <string.h>
#include <iostream.h>
#include <sys/stat.h>
#include <stdio.h>
```

```
#include <fstream.h>
#include <fcntl.h>
#include "../def.h"
#include "str2cont.h"
#include "container.h"

#ifdef WIN32
 #include "../common/bool.h"
 #include <io.h>
#endif

//-------------------------------------------------
// Constructor for the Stream2Container class.
//-------------------------------------------------
Stream2Container::Stream2Container(char *in_filename,char *out_filename,char *out_internname) {

   m_infile = fopen(in_filename,"r");
   m_outfile = fopen(out_filename,"w+");
   m_interfile = fopen(out_internname,"w+");

   // Zeroes TP_buffer
   for(int i = 0; i < (TP_BUFFER_SIZE) ; i++) m_TP_buffer[i]=0;

   // Calculate number of Transport packets

   struct stat info;
   stat(in_filename,&info);
   printf("File size: %i\n",info.st_size);
   m_total_TP = info.st_size / TP_SIZE;
   m_read_TP  = 0;
   m_containercounter = 0;
   m_count_TP = 0;
   m_numberPCR = 0;
   m_random = false;
   m_nextpcr.byte1 = 255;

}

//-------------------------------------------------
// Destructor for the Steam2Container class.
//-------------------------------------------------
Stream2Container::~Stream2Container() {

   printf ("\nTotal number of PCR's: %i\n",m_numberPCR);
   fclose(m_infile);
   fclose(m_outfile);
   fclose(m_interfile);
}
//-------------------------------------------------
// This is the function that does the main work. It calls the other
// functions and contains the main logic.
//-------------------------------------------------
bool Stream2Container::Transform() {

   short PES_start;
   short numberofcontainers     = 0;
   long  TP_pointer             = 0;
   long  container_size         = CONT_DATASIZE;
   long  tp_buffer_size         = 0;

   TP_TYPE tptype;
   STREAM_TYPE stype;

   bool cont = true;

   if (m_outfile == NULL || m_interfile == NULL){
     printf("Could not make outfiles, aborting!");
     return false;
   }

   printf ("\nContainer packing process started...\n");
   printf ("Total TP: %i\n",m_total_TP);

   TP_pointer=0;
   m_read_TP = FillTPBuffer();

   if (m_read_TP == 0) {
      printf("Could not read from infile...aborting.\n");
      return false;
   }

   tp_buffer_size = m_read_TP;
```

```
   m_current_container = new IContainer();
   m_current_container->SetRandomAccess(1); // assume a file can be played from the start...


   printf ("Start loop...readTP: %i\n",m_read_TP);

   // Loop while there are transport packets left to read from file;
   while (cont) {
      if (m_read_TP >= m_total_TP) cont = false;

      // Loop while there are transport packets left in buffer and space left
      // for at least one TP in current container and no RAI found.
      while ((TP_pointer < tp_buffer_size) && (m_containercounter <= (container_size -
TP_SIZE)) && !m_random) {
          PES_start=0;
           tptype = GetTPType(TP_pointer);

          switch (tptype) {

            case NONE:
                      break;
            case AF:
                      UpdateContainerAF(TP_pointer);
                      break;
            case US:
                      PES_start=4;
                      stype = GetSTREAMType(TP_pointer, PES_start);
                      if ((stype == VIDEO) || (stype == AUDIO)){
                              if (stype == VIDEO) {
                                      UpdateContainerPES(TP_pointer, PES_start);
                                      printf("V");
                              }
                              else if(stype == AUDIO)
                                      printf("A");

                      }
                      break;
            case AF_US:
                      UpdateContainerAF(TP_pointer);
                      // should maybe check if PES_start > TP_size here!
                      PES_start= 4 + GetAFLength(TP_pointer);
                      stype = GetSTREAMType(TP_pointer, PES_start);
                      if ((stype == VIDEO) || (stype == AUDIO)){
                              if (stype == VIDEO) {
                                      UpdateContainerPES(TP_pointer, PES_start);
                                      printf("V");
                              }
                              else if(stype == AUDIO)
                                      printf("A");
                              }
                      break;
               }

         // if we have found a random access, skip this part and
         // insert packet in next container instead.
         if(!m_random){
           memcpy (&(m_current_container->data[m_containercounter]),
&m_TP_buffer[TP_pointer*TP_SIZE],TP_SIZE);
           m_containercounter += TP_SIZE;  // Count in bytes
           TP_pointer++;
           m_count_TP++;
           printf(".");
         }
        }

        // Container full. Write to disk and generate new.
        if ((m_containercounter >= (container_size - TP_SIZE)) || m_random == true) {
           if (m_containercounter > 0) {
           m_current_container->SetDataLength(m_containercounter);
           m_current_container->SetID(++numberofcontainers);
           m_current_container->WriteToDisk(m_outfile, m_interfile);
           printf ("\nContainer generated\n");
           m_current_container->PrintContainerInfo();
         }
         delete m_current_container;
         m_current_container = new IContainer();
         m_containercounter = 0;

         // if this new container starts with a random access, write the last packet.
         if(m_random) {
           printf("Starting construction of a container with random access.\n");
           m_current_container->SetRandomAccess(1);
```

```
           if(m_nextpcr.byte1 != 255) m_current_container->SetPCR(m_nextpcr);
           memcpy (&(m_current_container->data[m_containercounter]),
&m_TP_buffer[TP_pointer*TP_SIZE],TP_SIZE);
                        m_containercounter += TP_SIZE;  // Count in bytes
                        TP_pointer++;
                        m_count_TP++;
           printf(".");
           m_nextpcr.byte1 = 255;
           m_random = false;
         }
      }

      // Transport packet buffer empty
      if (TP_pointer >= tp_buffer_size) {
      tp_buffer_size = FillTPBuffer();
      m_read_TP += tp_buffer_size;
      TP_pointer = 0;
      }

      // Makes sure the last TP's are put in the last container.
      if (m_count_TP < m_total_TP) cont = true;

      }

    // Write last container to file (if not empty).
    if (m_containercounter > 0) {
      m_current_container->SetDataLength(m_containercounter);
      m_current_container->SetID(++numberofcontainers);
      m_current_container->WriteToDisk(m_outfile, m_interfile);
      m_current_container->PrintContainerInfo();
    }

    delete m_current_container;

    return true;
}

//-------------------------------------------------
// This function fills the transportpacket buffer with new
// data from the transportstream defined by m_infile.
//-------------------------------------------------
long Stream2Container::FillTPBuffer(){

  long nrpackets = 0;

  // Read first packet.
  fseek(m_infile, m_read_TP*TP_SIZE,SEEK_SET);
  nrpackets = fread(&m_TP_buffer[0], 1, TP_BUFFER_SIZE, m_infile);

  nrpackets = nrpackets / TP_SIZE;

  return nrpackets;
}

//-------------------------------------------------
// This function checks which type of transportpacket this is.
// Does it contain: an adaptation field, a unit start (pes start),
// none, or both?
//-------------------------------------------------
TP_TYPE Stream2Container::GetTPType(long TP_pointer){

  long offset = (TP_pointer*TP_SIZE) + 1;

  // offset points to payload unit start indicator byte, offset + 2 to adaptation field
control.

  // Returns the correct transportpacket type.

  if ((m_TP_buffer[offset+2]&48) == 48 && (m_TP_buffer[offset]&64) == 64) return AF_US;
  if ((m_TP_buffer[offset+2]&48) == 32 || (m_TP_buffer[offset+2]&48) == 48) return AF;

  // Checks payload unit start indicator bit.
  if ((m_TP_buffer[offset]&64) == 64) return US;

  return NONE;

}
//-------------------------------------------------
// This function updates the current container with info
// taken from the adaptationfield we are currently studying.
// This info can be: a RAI and/or a PCR-field.
//-------------------------------------------------
```

```
bool Stream2Container::UpdateContainerAF(long TP_pointer){

  TP_TYPE type = GetTPType(TP_pointer);
  STREAM_TYPE stype;

  // No adaptation field present in this packet.
  if (type == US || type == NONE) return false;

  // --- First we extract the random access indicator flag: ---

  // Calculates offset to the byte which contains the random access indicator flag.
  long offset = (TP_pointer*TP_SIZE) + 4 + 1; // transport packet header (4) + 1.

  // Is the random access indicator set? If it is, check if it is a
  // video packet. If the PES-packet doesn't start in this packet, we
  // can't check this, then we assume it is video.

  if ((m_TP_buffer[offset]&64) == 64) {
    if(type == AF_US){
       int PES_start = 4 + GetAFLength(TP_pointer);
                stype = GetSTREAMType(TP_pointer, PES_start);
           if (stype == VIDEO){
          m_random = true;
       }
    }
    else {
      m_random = true;
    }
  }

  // --- Then we extract the PCR field: ---

  PCR_TYPE pcr;

  // If the PCR value is already set for this container, return true.
  if (m_current_container->GetPCR().byte1 != 255) return true;

  // Is the PCR flag set?
  if ((m_TP_buffer[offset]&16) == 16) {

       offset ++; // next byte contains the start of the PCR field.

       // See transport packet structure in MPEG-2 spesification to understand this.
       // PCR bit values are extracted and inserted into the pcr structure.

       pcr.byte5 = ((m_TP_buffer[offset+4]&128)/128) + ((m_TP_buffer[offset+3]&127)*2);
       pcr.byte4 = ((m_TP_buffer[offset+3]&128)/128) + ((m_TP_buffer[offset+2]&127)*2);
       pcr.byte3 = ((m_TP_buffer[offset+2]&128)/128) + ((m_TP_buffer[offset+1]&127)*2);
       pcr.byte2 = ((m_TP_buffer[offset+1]&128)/128) + ((m_TP_buffer[offset+0]&127)*2);
       pcr.byte1 = ((m_TP_buffer[offset+0]&128)/128);

       printf("PCR");
       m_numberPCR++;
  }
  else pcr.byte1 = 255;

  if (m_random) m_nextpcr = pcr; // set this pcr for next container
  else m_current_container->SetPCR(pcr);

  return true;

}
//-------------------------------------------------
// This function updates the container with info from
// The PES-header we are currently studying.
// Can be used for getting PTS/DTS values and more.
// Nothing is currently used from this PES-header.
//-------------------------------------------------
bool Stream2Container::UpdateContainerPES(long TP_pointer, short PES_start){


/* Removed...we will not use this code

  TP_TYPE type = Get_TP_type(TP_pointer);


  // No PES field present in this packet.
  if (type == AF || type == NONE) return false;

  PTS_TYPE pts;
  // If the PTS value is already set for this container, return true.
  if (m_current_container->GetPTS().byte1 != 255) return true;
```

```
  // Calculates offset to the byte which contains the PTS valueflag
  long offset = (TP_pointer*TP_SIZE) + PES_start + 7;

  // Makes sure the PTS value exists in the current buffer.
  if (offset >= TP_BUFFER_SIZE - 7) return false;


  // Update the header in current container
  if ((m_TP_buffer[offset]&128) == 128) { // if PTS flag set
       offset += 2;

    // See PES packet structure in MPEG-2 spesification to understand this.
    // PTS bit values are extracted and inserted into the pts structure.

    pts.byte5 = ((m_TP_buffer[offset+4]&254)/2) + ((m_TP_buffer[offset+3]&1)*128);
    pts.byte4 = ((m_TP_buffer[offset+3]&254)/2) + ((m_TP_buffer[offset+2]&2)*64);
    pts.byte3 = ((m_TP_buffer[offset+2]&252)/4) + ((m_TP_buffer[offset+1]&1)*64) +
((m_TP_buffer[offset+1]&2)*128);
    pts.byte2 = ((m_TP_buffer[offset+1]&252)/4) + ((m_TP_buffer[offset+0]&2)*32) +
((m_TP_buffer[offset+0]&4)*32);
    pts.byte1 = ((m_TP_buffer[offset+0]&8)/8);

    printf("PTS");
  }
  // PTS did not exist in this packet.
  // else pts.byte1 = 255;

  //m_current_container->SetPTS(pts);

  */
  return true;

}
//--------------------------------------------------
// This function examines the adaptationfield in the
// transportpacket if it exists and returns the length.
//--------------------------------------------------
short Stream2Container::GetAFLength(long TP_pointer){

  TP_TYPE type = GetTPType(TP_pointer);

  // no adapt.field present in this packet.
  if (type == US || type == NONE) return 0;

  // calculates offset to the byte which contains the adapt.field length.
  long offset = (TP_pointer*TP_SIZE) + 4;

  // returns the length of the adaptation field.
  return ((short) m_TP_buffer[offset] & 255) + 1; // +1 :adapt.length + THIS BYTE

}


//--------------------------------------------------
// This function examines the pes-packet in this transp.packet
// and returns the type of the elementary stream it contains.
//--------------------------------------------------
STREAM_TYPE Stream2Container::GetSTREAMType(long TP_pointer, short PES_start){

  // calculates offset to the byte which contains the stream type.
  long offset = (TP_pointer*TP_SIZE) + PES_start + 3;

   // If stream_id equals an audio or video stream return type VIDEO or AUDIO
  if ((m_TP_buffer[offset] & 224) == 192)
       return AUDIO;
  if ((m_TP_buffer[offset] & 240) == 224)
       return VIDEO;
  else
       return OTHER;
}
//--------------------------------------------------
// This function calculates the PID value in a transp. packet.
//--------------------------------------------------
int Stream2Container::GetPID(int offset) {

   unsigned short tmp;
   unsigned short pid = m_TP_buffer[offset+1];
   pid &= 31;
   pid <<= 8;
   tmp = m_TP_buffer[offset+2];
   tmp &= 255;
```

```
   pid += tmp;
   return pid;
}
//-----------------------------------------------------
```

container.h

```
// This class defines the intermediary format used by copac and con2elv.
// It offers methods for reading and writing the containers to disk and for
// setting the values in the class.

#ifndef _ICONTAINER_H
#define _ICONTAINER_H

#include "../def.h"
#include <fstream.h>
#include <stdio.h>

// Total bytes used on disk for each container. CHANGE this if structure on disk
changes!!!!!!!!
const int CONT_DISK_LENGTH = 15 + CONT_DATASIZE;

// This struct defines 5 bytes which are used to store the PCR-field.
typedef struct MY_PCR_TYPE {

   unsigned char byte1;      // Most Significant, Value of 255 indicates invalid PCR.
   unsigned char byte2;
   unsigned char byte3;
   unsigned char byte4;
   unsigned char byte5;      // Least Significant

} PCR_TYPE;

class IContainer {
private:
   short          m_headerlength;
   short          m_datalength;     // Bytes of data in data[] field
   unsigned int   m_containerID;
   PCR_TYPE       m_original_PCR;
   unsigned int   m_original_PCRValue;

   short m_random_access_offset;  // Value of -1 indicates no random access in this container.

public:
    unsigned char data[CONT_DATASIZE];  // The data in the container (TP packets)

    IContainer();
   ~IContainer();
   void     PrintContainerInfo(void);    // Prints information about container
   void     PrintPCRValue(void);         // Prints the value of the PCR.
   unsigned int CalculatePCRValue(void); // Calculates the value from the 5 bytes.

   void     SetRandomAccess(short value);// This container starts with a RAI!
   void     SetDataLength(short value);  // Effective datalength in container.
   void     SetPCR(PCR_TYPE pcr);        // Sets the PCR bytes.
   void     SetID(unsigned short id);    // Number (id) of container.

   // Returns the respective values:
   short          GetRandomAccess();
   short          GetDataLength();
   PCR_TYPE       GetPCR();
   unsigned short GetID();

   // Diskoperations:
   int  ReadFromDisk(FILE *fhandle);
   void WriteToDisk(FILE *fhandle, FILE *finterpolate);

};
#endif _ICONTAINER_H
```

container.c

```
// Implementationfile for the class IContainer.

#include "container.h"


IContainer::IContainer()
{
   m_headerlength          = sizeof (IContainer) - CONT_DATASIZE;
   m_original_PCR.byte1    = 255;           // No valid original PCR (yet)
   m_random_access_offset  = -1;            // No valid random access TP
   m_datalength            = 0;             // No data in data[] array.
   m_original_PCRValue     = 0;

}

IContainer::~IContainer()
{

}
//-----------------------------------------
void IContainer::SetRandomAccess(short value)
{
       m_random_access_offset = value;
}
//-----------------------------------------
void IContainer::SetDataLength(short value)
{
       m_datalength = value;
}
//-----------------------------------------
void IContainer::SetPCR(PCR_TYPE pcr)
{
       m_original_PCR.byte1 = pcr.byte1;
       m_original_PCR.byte2 = pcr.byte2;
       m_original_PCR.byte3 = pcr.byte3;
       m_original_PCR.byte4 = pcr.byte4;
       m_original_PCR.byte5 = pcr.byte5;

       // Calculates the PCR value and puts the value in m_original_PCRValue.
       if (m_original_PCR.byte1 != 255) CalculatePCRValue();

}
//-----------------------------------------
void IContainer::SetID(unsigned short id)
{
       m_containerID = id;
}

//-----------------------------------------
short IContainer::GetRandomAccess()
{
       return m_random_access_offset;
}
//-----------------------------------------
short IContainer::GetDataLength()
{
       return m_datalength;
}
//-----------------------------------------
PCR_TYPE IContainer::GetPCR()
{
       return this->m_original_PCR;
}
//-----------------------------------------
unsigned short IContainer::GetID()
{
       return m_containerID;
}

//-----------------------------------------
void IContainer::WriteToDisk(FILE *fhandle, FILE *finterpolate)
{

  printf("\nWriting container to file...\n");
```

```
  fwrite(&m_headerlength, 1, sizeof(m_headerlength), fhandle);
  fwrite(&m_datalength, 1, sizeof(m_datalength), fhandle);
  fwrite(&m_containerID, 1, sizeof(m_containerID), fhandle);
  fwrite(&m_random_access_offset, 1, sizeof(m_random_access_offset), fhandle);
  fwrite(&m_original_PCR.byte1, 1, sizeof(m_original_PCR.byte1), fhandle);
  fwrite(&m_original_PCR.byte2, 1, sizeof(m_original_PCR.byte2), fhandle);
  fwrite(&m_original_PCR.byte3, 1, sizeof(m_original_PCR.byte3), fhandle);
  fwrite(&m_original_PCR.byte4, 1, sizeof(m_original_PCR.byte4), fhandle);
  fwrite(&m_original_PCR.byte5, 1, sizeof(m_original_PCR.byte5), fhandle);
  fwrite(data, 1, CONT_DATASIZE, fhandle);

  // writes a help file used for later interpolating of PCR values.
  if (m_original_PCRValue != 0) {
          fwrite(&m_containerID, 1, sizeof(m_containerID), finterpolate);
          fwrite(&m_original_PCRValue, 1, sizeof(m_original_PCRValue), finterpolate);
  }

}
//-----------------------------------------
int IContainer::ReadFromDisk(FILE *fhandle)
{

  int nobyt = 0;
  nobyt += fread(&m_headerlength,1,sizeof(m_headerlength),fhandle);
  nobyt += fread(&m_datalength,1,sizeof(m_datalength),fhandle);
  nobyt += fread(&m_containerID,1,sizeof(m_containerID),fhandle);
  nobyt += fread(&m_random_access_offset,1,sizeof(m_random_access_offset),fhandle);
  nobyt += fread(&m_original_PCR.byte1,1,sizeof(m_original_PCR.byte1),fhandle);
  nobyt += fread(&m_original_PCR.byte2,1,sizeof(m_original_PCR.byte2),fhandle);
  nobyt += fread(&m_original_PCR.byte3,1,sizeof(m_original_PCR.byte3),fhandle);
  nobyt += fread(&m_original_PCR.byte4,1,sizeof(m_original_PCR.byte4),fhandle);
  nobyt += fread(&m_original_PCR.byte5,1,sizeof(m_original_PCR.byte5),fhandle);
  nobyt += fread(data,1,CONT_DATASIZE,fhandle);

  // Calculates the PCR value and puts the value in m_original_PCRValue.
  if (m_original_PCR.byte1 != 255) CalculatePCRValue();

  return nobyt;
}
//-----------------------------------------
void IContainer::PrintContainerInfo(void)
{
    printf("\nContainerID: %i\n", (int) m_containerID);
    printf("Headerlength: %i\n", (int) m_headerlength);
    printf("Numbers of TP's in container: %i\n", m_datalength / TP_SIZE);
    printf("Efficient data in container: %i\n", m_datalength);

    if ((int)m_original_PCR.byte1 != 255) {
        printf("PCRvalue is (int) %u\n", m_original_PCRValue);
        // stupid extrabit...most significant
    if ((m_original_PCR.byte1 & 1) == 1) printf("+ extrabit!!!");
    }
    if ((int)m_random_access_offset != -1) {
       printf("Random Access exists in TP %i.\n",(int) (m_random_access_offset / TP_SIZE));
    }

         printf("\n\n");
}

//-----------------------------------------
unsigned int IContainer::CalculatePCRValue(void)
{

   // Missing stupid upper bit in field here but....

   unsigned int value = 0;

   value = value + m_original_PCR.byte2;
   value <<=8;
   value = value + m_original_PCR.byte3;
   value <<=8;
   value = value + m_original_PCR.byte4;
   value <<=8;
   value = value + m_original_PCR.byte5;

   m_original_PCRValue = value;
   return value;
}
```

Con2Elv

con2elv.c

```
// Main program for running the con2elv application

#include <iostream.h>
#include <stdio.h>
#include "elvira.h"


void main(int argc, char *argv[])
{
   if (argc != 4 && argc != 5) {
      printf("Transforms Elvira II intermediary files (generated by Copac)\n");
      printf("to Elvira II format\n\n");
      printf("Syntax: con2elv <in_file> <in_interfile> <indexfile> (<direction>)\n");
      printf("Default direction is forward. (For backward: \"b\")\n\n");
      exit(-1);
   }

   ElviraFile con2elv(argv[1],argv[2],argv[3]);

   printf("Starting program.\n");

   // Check if this elvirafile is for backward winding.
   if (argc == 5){
     if (strcmp("b",argv[4]) == 0) con2elv.SetBackward();
   }

   if (con2elv.Generate()) printf ("\nElvira II File format generated.\n");

   else printf ("\nError in Elvira II File format generation!!.\n");

}
```

elvira.h

```
// This class is responsible for inserting the containers into the Elvirafile.
// It must interpolate timestamps, and make indexfiles for locating segments and
// for locating random access points.

#ifndef _ELVIRAFILE_H
#define _ELVIRAFILE_H

#include <fstream.h>
#include "../def.h"
#include "../../pump/moviefile.h"
#include "../copac/container.h"
#include "../../pump/videodir.h"

// A struct which defines an entry in the interpolate file.(one container)
struct interpolate {
       int containerID;
       int PCRValue;
};

class ElviraFile {
private:

   // Variables
   FILE        *m_infile;
   FILE        *m_interfile;
   char         m_internname[100];   // name of interpolatefile
   char         m_ixname[100];      // name of indexfile
   IContainer  *m_container;        // pointer to a container object

   ofstream      m_accessindexfile;  // file for locating random access points
   int           m_headersize;       // size of moviefileheader
   unsigned int *m_interarray;       // array storing values used for interpolation.
   int           m_no_interentries;  // number of entries in the array.

   NPTime        m_time;             // a value of nptime format (millisec)
   MFheader      m_MFheader;         // an object which defines the moviefileheader.

   VideoInfo     m_videoinfo;                // videoinfo object.
   InstanceInfo* m_instances[MAXINSTANCES];  // array of instances.
   int           m_noinstances;              // no of instances in m_instances
   int           m_totalmsecs;       // difference highest - first PCR-value.
   int           m_norandomindicators;   // number of random access indicators
   bool          m_backward;             // is this a backward winding file?
   unsigned int  m_lastentrypcr;         // last entry in interfile

public:

   // Constructor, initializes variables
   ElviraFile(char *in_filename, char *in_internname, char *ixname);

  ~ElviraFile(void);

   // Generate a file of EII File Format from containers.
   // Returns false if a problem occurs
   bool Generate(void);

   void SetBackward(void); // For backward winding files, switches dir. of timestamps.

private:

   // Reads the interpolate-file which contains data needed
   // for interpolation of PCR values.
   bool ReadInterFile(void);

   // Returns Interpolated PCR value for current container
   // Responsible for converting MPEG-2 PCR to ElviraII NPTime scale.
   NPTime GetInterpolatedPCR(unsigned int id);

   // Makes access file for random access into a file.
   void  WriteAccessFile(const InstanceInfo *info);

   // Checks if a random access is possible for this container.
   short ExtractRandomAccess(int offset);

   void  SetVideoInfo (void);    // sets dummy values for now....

};
```

```
#endif _ELVIRAFILE_H
```

con2elv.c

```
// Implementationfile for the class ElviraFile

#include <string.h>
#include <stdlib.h>
#include <iostream.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <math.h>
#include "elvira.h"
#include "../../pump/moviefile.h"
#include "../../common/bool.h"
#include "../../filsys/diskfreelist.h"
#include "../../filsys/filsys.h"
#include "../../filsys/moviefileop.h"
#include "../copac/container.h"
#include "../def.h"
//-------------------------------------------------
// Constructor for the class ElviraFile
//-------------------------------------------------
ElviraFile::ElviraFile(char *in_filename,char *in_internname,char *ix_name) {

   m_infile      = fopen(in_filename ,"r");
   m_interfile   = fopen(in_internname,"r");
   strcpy(m_internname,in_internname);
   strcpy(m_ixname, ix_name);

   // effective headersize, value not used in program...
   m_headersize  = MF_NAMELEN + 6*sizeof(int);
   m_backward = false;
   m_noinstances = 0;
   m_norandomindicators = 0;
}
//-------------------------------------------------
// Destructor for the class ElviraFile
//-------------------------------------------------
ElviraFile::~ElviraFile() {

   delete [] m_interarray;
   fclose(m_infile);
   fclose(m_interfile);

}
//-------------------------------------------------
// This is the function that contains the main logic and
// is responsible for controlling the transfer of the containers
// into the Elvira II format.
//-------------------------------------------------
bool ElviraFile::Generate() {

   if (m_infile == NULL || m_interfile == NULL) {
       printf ("Error, could not read infiles.");
       return false;
   }

   SetVideoInfo(); // dummy values...

   char tmp[100];
   strcpy(tmp,m_ixname);
   m_accessindexfile.open(strcat(tmp,".acc"), ios::out);

   // Create an Index object for storing the instance indexes
   VideoIx* index = new VideoIx(m_videoinfo.id, m_videoinfo.videoname);

   MovieFileOp moviefile;
   const InstanceInfo *instinfo;
   DSM_Scale Speed;

   if (m_backward) Speed = DSM_Scale(-1,1);
   else Speed = DSM_Scale(1,1);


   // Create an Elvira Moviefile for storing the video containers
   if(moviefile.New(MP2ELVIRADISK, m_videoinfo.videoname, VE_MPEG2, 25,
                    Speed,
                    SEGMENT_SIZE, true) != 0){
```

```
  printf("Could not make a moviefile.\n");
  return false;
}

int firstpcr = -1;
unsigned int highestpcr = 0;
NPTime temp;

// Read the interpolate-file made by copac.
if (ReadInterFile() == false) {
    printf ("Error when reading PCRfile");
    return false;
}

// get the last entry from interfile, use it for switching the timestamps
// for backward videofiles.
m_lastentrypcr = (unsigned int) (m_interarray[2*m_no_interentries-1] / PCR_TO_MSEC);

bool cont = true;
unsigned int efflength = 0;

m_container = new IContainer();

// reads the first container from infile.
int contsize = m_container->ReadFromDisk(m_infile);

if (contsize <= 0) {
  printf ("Could not read a container from infile.\n");
  return false;
}

int id = m_container->GetID();
int t=1;
while (cont == true) {
    // Calculate values for the header entry for current container
    m_time = GetInterpolatedPCR(id);

    // for calculating tot. no. seconds (ca):
    if (firstpcr == -1) firstpcr = m_time.GetTime();
    if (m_time.GetTime() > highestpcr)
        highestpcr = m_time.GetTime();

    efflength = m_container->GetDataLength();

    // prints timeinformation (useful for debugging)
    printf ("ContNr %i : ", id);
    m_time.print();
    printf (" Eff length: %i",efflength);
    printf ("\n");

    temp = m_time; // use a temp nptime...
    // If backward winding, switch direction of timestamps in containers
    if (m_backward){
       temp = NPTime(m_lastentrypcr) - m_time;
       if (temp < 0) temp = 0; // make sure its not negative
    }
    if (moviefile.WriteContainer(temp,temp,efflength,
          &(m_container->data[0])) != 0) return false;

    //m_container->PrintContainerInfo();

    if (m_container->GetRandomAccess()!=-1){
       instinfo = moviefile.GetInstance();
       WriteAccessFile(instinfo);
    }

    // Update moviefile header
    m_MFheader.ncontainers++;

    // reads a new container.
    contsize = m_container->ReadFromDisk(m_infile);
    id = m_container->GetID();
    if (contsize <= 0)cont = false;
}


// Calculate total number of msecs. (Assumes 1 program!!)
m_totalmsecs = highestpcr - firstpcr;
printf ("\nTotal number of millisecs: %i\n", m_totalmsecs);
m_videoinfo.duration.SetTime(m_totalmsecs);

instinfo = moviefile.GetInstance();
```

```
   index->insert(*instinfo,m_videoinfo.duration);

   // writes the indexfile.
   index->write(m_ixname);

   return true;

}

//-------------------------------------------------
// This function reads the interpolatefile made by copac.
// Each entry consists of a containerid (number) and a pcr-value.
//-------------------------------------------------
bool ElviraFile::ReadInterFile() {
  // Read the whole interpolate-file and inserts the int-values in a buffer.

  struct stat info;
  stat(m_internname,&info);
  //printf("Interfile size: %i\n",info.st_size);
  int length = info.st_size;

  if(length < 16) {
   printf ("Too few entries in interfile\n");
   return false;
  }

  m_no_interentries = (length / sizeof(unsigned int))/2;
  m_interarray = new unsigned int[m_no_interentries*2];

  if (fread(m_interarray,1,length,m_interfile) == 0)
    return false;

  else return true;

}
//-------------------------------------------------
// This function writes one entry in the access-indexfile.
//-------------------------------------------------
void ElviraFile::WriteAccessFile(const InstanceInfo *info)
{
   // info->write(stdout);

   //Get the random access offset (if set), puts info into accessindexfile.
   short randomaccess = m_container->GetRandomAccess();
   const SegInfo *seg;
   seg = info->lastSegInfo(); //get current seg...
   NPTime temp = m_time;
   if (m_backward){
     temp = NPTime(m_lastentrypcr) - m_time;
     if (temp < 0) temp = 0; // make sure its not negative
   }

   m_accessindexfile << " PCR(msec):  " << temp.GetTime();
   m_accessindexfile << " T ";
   //m_accessindexfile << " SegID: " << seg->id.unit << " " << seg->id.blockno;
   //m_accessindexfile << " ContID: " << m_container->GetID();

   m_accessindexfile << endl;

}
//-------------------------------------------------
// This function sets some values for the video. (dummy values)
//-------------------------------------------------
void ElviraFile::SetVideoInfo (void)
{
   // dummy values for now....set values into a videoinfo class.
   m_videoinfo.duration = 7000;
   m_videoinfo.id       = 311;
   strcpy(m_videoinfo.videoname, VIDEONAME);

   // ...values into the class MFheader.
   m_MFheader.magic          = MF_MPEG2;       // MPEG-2 file
   m_MFheader.segsize        = SEGMENT_SIZE;   // Segment size
   m_MFheader.firstsegoffset = MF_HDRSIZE;     // Offset to first segment
   m_MFheader.firstdiroffset = 0;               // ????
   memcpy(m_MFheader.name, m_videoinfo.videoname, MF_NAMELEN);

}
//-------------------------------------------------
// This function interpolates the value for a container with
// a given id. It uses the array of entries read from the interpolatefile.
//-------------------------------------------------
```

```
NPTime ElviraFile::GetInterpolatedPCR(unsigned int containerID) {

   // The constant PCR_TO_MSEC in this function will define the relation between
   // the PCR values and the respective time the container will have to be
   // sent (in millisecs).

        double interpolate;
        int i=0, i2=0;

        NPTime time;

        // Locate first entry which is equal or greater than containerID
        while((containerID > m_interarray[2*i]) && (i < m_no_interentries-1)) i++;

        i2 = 2*i;

        // Special case - first entry in array
        if(i == 0) {
                // If containerID equals first entry, just fetch PCR value
                if(containerID == m_interarray[i2]) {
                        time = (unsigned int) (m_interarray[i2+1] / PCR_TO_MSEC);
                        printf("EQUAL %i ",i); // debug
                        printf("%i  ",m_interarray[i2]);
                        return time;
                }
                // Else calculate an interpolated value
                else {
                        interpolate = (m_interarray[i2+3] - m_interarray[i2+1]);
                        interpolate = interpolate / (m_interarray[i2+2] - m_interarray[i2]);

                        time = (unsigned int) ((m_interarray[i2+1] - (m_interarray[i2] -
containerID) * interpolate) / PCR_TO_MSEC);
                        printf("INTER %i ",i); // debug
                        printf("%i  ",m_interarray[i2]);
                        return time;
                }
        }
        // The normal case - entry found somewhere in array
        else {
                // If containerID equals last entry, just fetch PCR value
                if(containerID == m_interarray[i2]) {
                        time = (unsigned int) (m_interarray[i2+1] / PCR_TO_MSEC);
                        printf("EQUAL %i ",i); // debug
                        printf("%i  ",m_interarray[i2]);
                        return time;
                }
                // Else calculate an interpolated value
                else {
                        interpolate = abs(m_interarray[i2+1] - m_interarray[i2-1]);
                        interpolate = interpolate / (m_interarray[i2] - m_interarray[i2-2]);
                        time = (unsigned int) ((m_interarray[i2-1] + (containerID -
m_interarray[i2-2]) * interpolate) / PCR_TO_MSEC);
                        printf("INTER %i ",i); // debug
                        printf("%i  ",m_interarray[i2]);
                        return time;
                }
        }

        return 0;             // Will never be executed
}
//-------------------------------------------------------
// When this function is called, it will prepare the class for
// contruction of backward winding files.
//-------------------------------------------------------
void ElviraFile::SetBackward()
{
  printf("Setting backward winding.\n");
  m_backward = true;
}
//-------------------------------------------------------
```

wsort

wsort.c

```
// This program sorts and renames a number of files which are organized as follows:
// <filebase><number><fileending>.
// The numbering tells the order of files. Some files are selected and the filenames
// are renumbered such that these files makes the new order.
// A factor decides how many and which files are to be selected.

#include <stdlib.h>
#include <stdio.h>
#include <fstream.h>
#include <string.h>
#include <sys/stat.h>

const char imtype[10] = ".ppm";     // type of image file
const char PARfile[20] = "PAL.par"; // org.parameter file for coding
int firstfile = 0;                  // number of first file to be coded

char *dir;
char *in;
char *out;
float factor;
int nofiles = 0;
char end[2]="\0";

void GetFilename(char *filename,char *name, int number);
int  CountFiles(char *name);
void Switch(char *name);
int  Reduce(char *name);
void UpdatePAR(int linenr, char *text);

main (int argc, char* argv[])
{

  if ((argc != 4) && (argc != 3))
    {
      printf("Syntax is wsort <filebase> <factor> <direction> \n");
      printf("Example: wsort in 3 B\n");
      exit(-1);
    }

  // Initializing
  if (argc == 4) dir = argv[3];
  else dir = "F";
  in = argv[1];

  sscanf(argv[2],"%f", &factor);    // reads float values.
  printf("Factor is %f\n", factor);


  if (factor < 1 || factor > 30) { // One may increase the number 30 if necessary...
     printf("Factor out of range (must be 1-30)\n");
     exit(-1);
  }

  printf("Starting windsort...\n");

  nofiles = CountFiles(in);
  printf("Number of files found: %i\n",nofiles);

  if (nofiles == 0){
    printf("No files of given inputformat found, aborting...\n");
    exit(-1);
  }

  if ((strcmp (dir,"b")==0) || (strcmp(dir,"B")==0)){
     Switch(in);                     // Switch direction before renaming
  }

  int noframes = Reduce(in);
```

```
  printf("New number of frames: %i\n", noframes);

  // makes changes to the parameterfile, add more changes here if wanted.
  UpdatePAR(2,strcat(in,"%d    /* name of source files (UPDATED BY WSORT)"));
  UpdatePAR(8,strcat(lltostr(noframes,end),
      "        /* no of frames (UPDATED BY WSORT)*/"));

}

//---------------------------------------
// This function switches the direction of the files
//---------------------------------------
void Switch(char *name)
{
  char tmpfile[100];
  char orgfile[100];

  printf("Switching of files in progress.\n");

  // temporary renaming
  for (int i = 0 ; i <  nofiles ; i++){
     GetFilename(&tmpfile[0],name,i+firstfile);
     strcpy(orgfile,tmpfile);
     strcat(tmpfile,".tmp");
     rename(orgfile,tmpfile);
  }
  // change direction
  for (int j = 0 ; j < nofiles ; j++){
     GetFilename(&tmpfile[0],name,j+firstfile);
     strcat(tmpfile,".tmp");
     GetFilename(&orgfile[0],name,nofiles-j-1+firstfile);
     rename(tmpfile,orgfile);
  }

  printf("Switching completed.\n");
}
//---------------------------------------
// This function picks some files and rename them
// such that these files make the new order.
//---------------------------------------
int Reduce(char *name)
{
  int frame = firstfile + 1;   // the number of the next frame (always keep first)
  double next = firstfile + factor;   // next frame-limit
  char orgfile[100];
  char newfile[100];

  printf("Reducing in progress.\n");

  // Pick some files, and rename these.
  for(int i = firstfile + 1 ; i < firstfile + nofiles ; i++){
    if (i >= next || i == firstfile + nofiles - 1) {  // always pick last frame.
      GetFilename(&orgfile[0],name,i);
      GetFilename(&newfile[0],name,frame);
      rename(orgfile,newfile);
      frame++;
      next += factor;
    }
  }

  // remove the extra files which are left.
  struct stat statinfo;
  for(int j = frame ; j < firstfile + nofiles ; j++){
    GetFilename(&orgfile[0],name,j);
    if (stat(orgfile,&statinfo)==0) remove(orgfile);
  }

  printf("Files reduced according to the factor %f\n", factor);
  return frame-firstfile;
}
//---------------------------------------
// This function returns the number of files present with
// the given syntax.
//---------------------------------------
int CountFiles(char *name)
{
   int c = 0;
   char filename[100];      // total filename
   struct stat statinfo;
   bool cont = true;
   for (c=firstfile ; cont == true ; c++){
      GetFilename(&filename[0],name,c);
```

```
      // printf("Checking:%s\n",filename);
      if (stat(filename,&statinfo)!=0) cont = false;
   }
   return c-1-firstfile;
}
//---------------------------------------
// This function constructs a filename from a base and a number.
//---------------------------------------
void GetFilename(char *filename,char *name, int nr)
{
   char number[10];
   char end[2]="\0";
   strcpy(filename,name);
   strcat(filename,lltostr(nr,end));
   strcat(filename,imtype);
}
//---------------------------------------
// This function updates the parameterfile with changes needed
// for the coding.
//---------------------------------------
void UpdatePAR(int linenr, char *text)
{

  ifstream param;
  int currline = 1;
  char PARtmp[20] = "par.tmp";
  param.open(PARfile, ios::in);
  if(!param){
    printf("Warning: Could not find the PAR-file!\n");
    return;
  }
  ofstream newparam;
  newparam.open(PARtmp, ios::out);
  if (!newparam){
    printf("Warning: Could not update parameter file.\n");
    return;
  }

  char linetxt[200];

  param.getline(linetxt,200);
  while(!param.eof()){
    if (currline == linenr) newparam << text << endl;
    else newparam << linetxt << endl;
    currline++;
    param.getline(linetxt,200);
  }
  rename(PARtmp,PARfile); // copy back changes
}
//-----------------------------------------
```

mux

mux.c

```
// Main program for the application mux.

#include <iostream.h>
#include "elstream.h"

void main(int argc, char *argv[])
{

   if (argc != 4) {
     printf("Syntax: mux <elem. stream> <pesfile> <tpfile>\n");
     exit(-1);
   }

   ElStream stream(argv[1],argv[2],argv[3]);

   printf ("Starting transportstream generator (mux)...\n");

   if (stream.Generate())
       printf ("\nStream constructed\n");

   else
       printf ("\nError in construction process!!!!\n");

}
```

elstream.h

```
// This class defines functions for making a simple transportstream
// from an elementarystream. PES-packets are made as a middle-step

#ifndef ELSTREAM_H
#define ELSTREAM_H
#include <fstream.h>
#include <stdio.h>

#define PSINAME "psifile"

const int ESRATE = 5000000; // Elementary bitstream rate.
const int VIDBUF = 24000; // A buffer for storing the el.stream.
const int PESBUF = 13000; // must be larger than maxpeslength...
const int MAXPESLENGTH = 12000;
const int MAXSEQLEN = 150;  // 140(header) + 10(extension)
const int MAXRANDOM = 1000; // max number of random access points.
const int FIRSTPCR = 100;    // starts almost from 0.
const int LASTPCR = 3330000; // 37 sec , this makes the pcr-values "normalplay-time".

class ElStream {
private:
   // Variables
   FILE *m_infile;
   FILE *m_pesfile;
   FILE *m_tpfile;
   FILE *m_seqinfile;
   char m_pesname[100];

   char m_psifilename[100];  // for stealing psipackets.
   char m_psibuf[372];       // ...put them here...

   unsigned int m_firstpcr;  // for interpolating
   unsigned int m_lastpcr;   //        "
   double  m_pcrinc;    // increment from pes-packet to pes-packet

   int  m_totalbytesread;
   int  m_nobuffer;
   int  m_nopes;     // number pes packets produced.
   int  m_noseq;     // number sequence headers inserted.
   int  m_notp;      // number TS packets produced.
   char *m_inbuffer; // inbuffer for reading the el.stream.

   int  m_seqlength;              // length of sequence header.
   char m_seqheader[MAXSEQLEN];   // sequence header is stored here.
   int  m_existrandom[MAXRANDOM]; // this array stores info about accesspoints in PES.
   int  m_norandom;               // how many accesspoints exists?

   // Methods

   void MakePES(int size);   // Main function for making PES-packets
   int  MakeTP(int size);    // Main function for making TP-packets.
   void WritePES(int start, int end, bool seqheader); // Write a PES-packet to file.
   bool ReadSequenceHeader(void);  // Read a sequence header from file.
   void WriteSequenceHeader(void); // Write          "

   void AnalyzeBuffer(int size);  // This function writes info about the el.stream.
   int  FillBuffer(int size,FILE *file);  // return number of bytes read.
   unsigned int CalculateValue(char *buffer, int offset); // Value of a int.
   void ReadPSIInfo(void);   // Read PSI packets from a orig. tpstream.
   void WritePSIInfo(void);  // Write PSI packets to a file.
   bool ExistRandom(int pesnr); // Is a pes-packet is identified as a accesspoint?

public:

    ElStream(char *in_filename,char *pes_filename,char *tp_filename);

   ~ElStream();

    bool Generate();   // Main function for this class.

};

#endif ELSTREAM_H
```

elstream.c

```
// Implementationfile for the class ElStream

#include <string.h>
#include <iostream.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
#include "elstream.h"
#include "pes.h"
#include "tp.h"

//-------------------------------------------------
// Constructor for the Elstream class.
//-------------------------------------------------
ElStream::ElStream(char *in_filename, char *pes_filename,char *tp_filename) {

   m_infile = fopen(in_filename ,"r");
   strcpy(m_pesname,pes_filename);
   m_tpfile = fopen(tp_filename, "w+");
   strcpy(m_psifilename,PSINAME);   // hardcoded so far.

   m_totalbytesread = 0;
   m_nobuffer = 1;
   m_nopes = 0;
   m_noseq = 0;
   m_notp = 0;
   m_norandom = 0;
   m_firstpcr = FIRSTPCR;
   m_lastpcr =  LASTPCR;
   m_pcrinc = 0;

}
//-------------------------------------------------
// Destructor for the Elstream class.
//-------------------------------------------------
ElStream::~ElStream() {

   fclose(m_infile);
   fclose(m_pesfile);
   fclose(m_tpfile);
   delete m_inbuffer;

}
//-------------------------------------------------
// This function contains the main logic for producing the
// transportstream.
//-------------------------------------------------
bool ElStream::Generate() {

   m_pesfile = fopen(m_pesname,"w+");

   if (m_infile == NULL || m_pesfile == NULL) {
       printf ("Error, could not open files.");
       return false;
   }

   if (!ReadSequenceHeader()){
     return false;
   }

   //testing...
   //WriteSequenceHeader();
   //exit(-1);

   m_inbuffer = new char[VIDBUF];

   printf("\nMaking PES packets\n\n");

   int bytesread = FillBuffer(VIDBUF,m_infile);
   if (bytesread < 0) {
         printf ("Error reading first buffer from infile!\n");
         return false;
   }

   // This loop makes PES-packets.
   while(bytesread > 0){
     //AnalyzeBuffer(bytesread);
```

```
      MakePES(bytesread);
      m_totalbytesread += bytesread;
      bytesread = FillBuffer(VIDBUF,m_infile);
    }

    fclose(m_pesfile); // pes file is ready, close file
    m_pesfile = fopen(m_pesname,"r"); // opens for reading

    printf("Total bytes read from infile: %i\n",m_totalbytesread);
    printf("Total number of pes-packets: %i\n",m_nopes);
    printf("Total number of extra seq headers written: %i\n",m_noseq);


    // now we make tp_packets...
    printf("\nMaking Transportpackets\n\n");

    // Calculate pcr increment
    m_pcrinc = (m_lastpcr - m_firstpcr)/m_nopes;
    printf("PCR incrementation: %f\n",m_pcrinc);
    //printf("first: %u\n",m_firstpcr);

    ReadPSIInfo(); // steal the psi packets from orig file...

    //WritePSIInfo(); // for making a psi file. Uncomment these two lines if you
    //exit(-1);       // want to make a psifile. The original stream must be present
                      // under the name "psifile". Rename afterwards.

    //printf("first: %u\n",m_firstpcr);
    m_nopes = 0;
    m_totalbytesread = 0;

    bytesread = FillBuffer(PESBUF,m_pesfile);
    if (bytesread < 0) {
      printf ("Error reading first buffer from pesfile!\n");
      return false;
    }
    //printf("first: %u\n",m_firstpcr);

    // This loop makes the transport packets from the PES-packets
    while(bytesread > 0){
      int pespos = MakeTP(bytesread);
      m_totalbytesread += pespos;
      fseek(m_pesfile,m_totalbytesread,SEEK_SET);
      bytesread = FillBuffer(PESBUF,m_pesfile);
      m_nopes++;
    }
    printf("PES packets processed: %i\n", m_nopes);
    printf("Total number of packets: %i\n", m_notp);

    return true;
}
//--------------------------------------------------
// This function contains the main logic for constructing
// the PES-packets. If sequence headers are not found in
// front of GOP's, this function will copy the orig. seq.
// header into the PES-packets, thus allowing random access.
//--------------------------------------------------
void ElStream::MakePES(int size){

  int cnt = 0;
  int peslen = 0;
  int noseqheaders = 0;
  int oldpesstart = 0;
  bool writeseqheader = false; // must always be a seqheader first...
  bool foundseqheader = true;  // is there a seqheader at start of gop?
  unsigned int value;

  if (size == 0) return;

  for (cnt = 0 ; cnt < size ; cnt++){
    if(m_inbuffer[cnt] == 0){
      if(m_inbuffer[cnt+1] == 0){
        if(m_inbuffer[cnt+2] == 1){
          value = (unsigned int) m_inbuffer[cnt+3]&255;
          // a NEW sequence header / GOP (not first)!!
          if(value == 179){
            if(cnt>0 && m_nopes != 0){
              printf("Found seq_header, writing PES-packet.\n");
              WritePES(oldpesstart,cnt-1,writeseqheader); // prev pes
                              }
            oldpesstart = cnt;
            foundseqheader = true;
```

```
            writeseqheader = false;
            peslen = 0;
          }
          if(value == 184 && foundseqheader == false){
            //printf("Found GOP start, writing pes-packet.\n");
            WritePES(oldpesstart,cnt-1,writeseqheader); // prev. pes-packet
            oldpesstart = cnt;
            writeseqheader = true;
            peslen = 0;
          }
        }
     }
  }
    if(peslen >= MAXPESLENGTH){
      //printf("Max limit, writing pes-packet from %i to %i\n", oldpesstart,cnt-1);
      WritePES(oldpesstart,cnt-1,writeseqheader);
      oldpesstart = cnt;
      foundseqheader = false;
      writeseqheader = false;
      peslen = 0;
    }
    peslen++;
  }
  // last PES packet...MAY be small...
  if(oldpesstart<size) {
    //printf("Last PES-packet...\n");
    WritePES(oldpesstart,size-1,writeseqheader);
  }
}
//-------------------------------------------------
// This function contains the main logic for constructing
// the transport packets. An packet containing only an adaptation field
// (for RAI and PCR) and PSI-packets are inserted before each PES-packet.
//-------------------------------------------------
int ElStream::MakeTP(int bufsize)
{

  unsigned int usedpes = 0;
  unsigned int pessize;
  unsigned int tpsize;
  unsigned int pcr;
  int notp = 0;

  // Checks if the start of pes-packet contains a pes-start code. (0-0-1).
  if (m_inbuffer[0] == 0 && m_inbuffer[1] == 0 && m_inbuffer[2] == 1){
    pessize = CalculateValue(m_inbuffer,4) + 6; // +6 include the start...
    printf("\nNew PES: Pessize is %i\n",pessize);
  }
  else {
    printf("WARNING: No pes start found at start of pes-packet\n");
    pessize = bufsize; // ignore pes starts
  }

  TPacket tp;
  tp.Init();

  m_firstpcr = FIRSTPCR; // why do I need to do this??

  printf("first: %u\n",m_firstpcr);

  pcr = (unsigned int) (m_firstpcr + (int)m_nopes*m_pcrinc);

  // Writes an adaptation packet, checks random access also
  tp.MakeAdapPacket(ExistRandom(m_nopes),pcr,m_tpfile);
  m_notp++;

  WritePSIInfo(); // write psi info before PES-start.

  tp.Init();
  tp.SetUnitStart();

  while (usedpes < pessize){
    if (usedpes > (pessize - 184)) {
      tpsize = pessize - usedpes;
      printf("Last TP: effective tpsize is %i\n",tpsize);
    }
    else tpsize = 184; // 188 - header(4)

    tp.Write(m_inbuffer,m_tpfile,usedpes,tpsize);
    usedpes += tpsize;
    tp.Init();
    notp++;
```

```
  }
  printf("Made %i tp's from pes-packet\n",notp);
  m_notp += notp; // total number

  return pessize;
}
//-------------------------------------------------
// This function reads the sequence header from an el.stream.
// It reads the maximum number of bytes a header may have.
// Then it calculates the real length.
//-------------------------------------------------
bool ElStream::ReadSequenceHeader(void){

  if(fread(m_seqheader,1,MAXSEQLEN,m_infile) <= 0){
    printf("Could not read sequence header\n");
    return false;
  }

  // now we have to calculate the real length...

  m_seqlength = 22; // absolute minimum total length

  // the length can be added 2*64 bytes if qmatrices are present:

  if((m_seqheader[11]&2)==2){
    m_seqlength += 64;
    if((m_seqheader[75]&1)==1) m_seqlength += 64;
  }
  else if ((m_seqheader[11]&1)==1) m_seqlength += 64;

  rewind(m_infile); // reset filepointer.
  return true;
}
//-------------------------------------------------
// This function writes a pes-packet to file after it has
// been constructed. Sets certain values as bitrate and random
// access first.
//-------------------------------------------------
void ElStream::WritePES(int start, int end, bool sequence)
{
   // Some of the following logic demands that the el_stream
   // only has a sequence header at the start.

   Pes pes;

   unsigned int len = end-start+1;   // datalength
   if (sequence || m_nopes == 0) {
     if(m_nopes != 0)len += m_seqlength; // add seqheader length
     pes.SetESRate(ESRATE);
     m_existrandom[m_norandom++] = m_nopes; // random access at this pes
   }
   pes.SetLength(len);
   pes.WriteHeader(m_pesfile);

   if (sequence) WriteSequenceHeader();

   int no = fwrite(&m_inbuffer[start],1,end-start+1,m_pesfile);
   //printf("Writing PES-data, %i bytes.\n",no);
   m_nopes++;
}
//-------------------------------------------------
// This function writes the sequence header to the pesfile when needed.
//-------------------------------------------------
void ElStream::WriteSequenceHeader(void){

  printf("Writing the sequence header, %i bytes.\n",m_seqlength);
  fwrite(m_seqheader,1,m_seqlength,m_pesfile);
  m_noseq++;
}
//-------------------------------------------------
// This function can be used for printing info about a el.stream.
//-------------------------------------------------
void ElStream::AnalyzeBuffer(int size)
{

  int cnt = 0;
  unsigned int value;
  printf("Analyzing buffer.\n");
  for (cnt = 0 ; cnt < size-3 ; cnt++){
    if(m_inbuffer[cnt] == 0){
      if(m_inbuffer[cnt+1] == 0){
        if(m_inbuffer[cnt+2] == 1){
```

```
          value = (unsigned int) m_inbuffer[cnt+3]&255;
          if(value == 179 || value == 181 || value == 184){
             printf("Startcode found in byte %i  ", m_totalbytesread+cnt);
             printf("Value is: %i\n",value);
          }
        }
      }
    }
  }

}
//-------------------------------------------------
// This function fills the el.stream buffer.
//-------------------------------------------------
int ElStream::FillBuffer(int size, FILE *file)
{

  // fseek(m_infile, , SEEK_SET);
  // printf("Reading buffer %i.\n",m_nobuffer);
  m_nobuffer++;
  int no = fread(m_inbuffer,1,size,file);
  // printf("Read %i bytes into buffer\n",no);
  return no;
}
//-------------------------------------------------
// This function calculates an int value from 2 (4) bytes.
//-------------------------------------------------
unsigned int ElStream::CalculateValue(char *buffer, int offset) {

       unsigned int value = 0;
       // switch these bytes for unix/NT
       value = value + (unsigned int) (buffer[offset] & 255);
       value <<= 8;
       value = value + (unsigned int) (buffer[offset+1] & 255);
       //value <<= 8;
       //value = value + (unsigned int) (buffer[offset+2] & 255);
       //value <<= 8;
       //value = value + (unsigned int) (buffer[offset+3] & 255);
       //
       return value;
}
//-------------------------------------------------
// This function reads psi-info from a given file.
// Assumes two first packets are psipackets. (as for np.m2t)
//-------------------------------------------------
void ElStream::ReadPSIInfo(void)
{
   FILE *psi;
   psi = fopen(m_psifilename,"r");

   if (psi == NULL) {
    printf("Could not read psi info\n");
    exit(-1);
   }

   fread(m_psibuf,1,2*188,psi);
}
//-------------------------------------------------
// This function writes the psiinfo as two packets to the
// stream under generation.
//-------------------------------------------------
void ElStream::WritePSIInfo(void)
{
   printf("Writing PSI packets\n");
       if (fwrite(m_psibuf,1,2*188,m_tpfile) <= 0){
     printf("Could not write PSI packets\n");
     exit(-1);
   }
   m_notp += 2;
}
//-------------------------------------------------
// This function examines if there exists a access points
// for this pes-packet.
//-------------------------------------------------
bool ElStream::ExistRandom(int pesnr)
{
  for (int i = 0; i < m_norandom ; i++){
    if (m_existrandom[i] == pesnr) return true;
  }
  return false;
}
//-------------------------------------------------
```

pes.h

```
// This class contain functionality for performing
// operation on pes-packet header. Is used by class Elstream.

#ifndef PES_H
#define PES_H

const int MAXPES = 100; // Maximum length of pes-header...

class Pes {

  private:
    char pesbuf[MAXPES];     // buffer containing the header.
    int bufsize;
    unsigned int optheadlen; // length of optional fields.

    int datalength;

    void WriteByte(char byte);  // for debugging
    void UpdateLength(int length); // update length of pes-packet

  public:

    Pes(void);
    void WriteHeader(FILE *file);      // write header to file.
    void SetESRate(unsigned int rate); // set the bitrate of elstream.
    void SetLength(unsigned int len);  // set length of pes-packet.

};

#endif PES_H
```

pes.c

```
// Implementationfile for the class Pes.

#include <stdio.h>
#include <stdlib.h>
#include "pes.h"

//--------------------------------------
// Constructor. Sets a lot of default values.
// See MPEG-2 Systems to understand the settings.
//--------------------------------------
Pes::Pes(void)
{
  pesbuf[0] = 0;
  pesbuf[1] = 0;
  pesbuf[2] = 1;     // start code prefix
  pesbuf[3] = 224;   // 1110 0000 - streamid, video
  pesbuf[4] = 0;     // packet length, set this later
  pesbuf[5] = 0;     //              "
  pesbuf[6] = 128;   // 10 00 0 0 0 0 - res,scr,pri,alg_ind,cright,org
  pesbuf[7] = 0;     // 00 0 0 0 0 0 0 - pts/dts,escr,rate,trick,copy,crc,ext,len
  pesbuf[8] = 0;     // length of opt. headers (indicated by prev. byte).

  pesbuf[9] = 0;
  pesbuf[10] = 0;
  pesbuf[11] = 0;
  pesbuf[12] = 0;
  pesbuf[13] = 0;
  pesbuf[14] = 0;


  bufsize = 9;
  optheadlen = 0;

}
//--------------------------------------
// Writes current header to the file.
//--------------------------------------
void Pes::WriteHeader(FILE *file)
{

   int no = fwrite(pesbuf,1,bufsize,file);
   // printf("Writing PES header: %i bytes\n", no);

}
//--------------------------------------
// Sets the bitrate for the el.stream.
//--------------------------------------
void Pes::SetESRate(unsigned int rate)
{

   printf("Setting bitrate to %i bits/sec\n",rate);

   rate = (int) rate / 400;   // in units of 50 bytes/sec (400 bits/s)
   pesbuf[bufsize+2] = 1 + (rate&127)*2;
   int temp = rate&128;
   rate >>=8;
   pesbuf[bufsize+1] = temp/128 + (rate&127)*2;
   temp = rate&128;
   rate >>=8;
   pesbuf[bufsize] = temp/128 + (rate&63)*2 + 128;

   pesbuf[7] = (pesbuf[7]&239) + 16; // set es-rate bit.
   optheadlen += 3; // add 3 to the length of opt fields
   //WriteByte(pesbuf[8]);
   //WriteByte(pesbuf[bufsize]);
   //WriteByte(pesbuf[bufsize+1]);
   //WriteByte(pesbuf[bufsize+2]);
   bufsize += 3;
}
//--------------------------------------
// Sets length of pes-packet.
// Must call this just before writing pes-packet...
//--------------------------------------
void Pes::SetLength(unsigned int len)
{
   // length AFTER 6 first bytes.
   unsigned short totallength = bufsize + len - 6;
```

```
   // totallength = 0; // testing just 0...
   pesbuf[5] = totallength&255;
   totallength >>= 8;
   pesbuf[4] = totallength&255;

   pesbuf[8] = optheadlen&255;

   //WriteByte(pesbuf[4]);
   //WriteByte(pesbuf[5]);
}
//-------------------------------------
// Useful for testing the bytes set.
//-------------------------------------
void Pes::WriteByte(char byte)
{
   // just for testing
   int val=128;
   for (int j=0;j<8;j++) {
     if((byte & val)>0) printf("1");
     else printf("0");
       val=val/2;
     }
   printf("\n");
}
//-------------------------------------
```

tp.h

```
// This class contains functions for making a TS packet.
// Many values are set as "default".

#ifndef TP_H
#define TP_H

const int PIDVALUE = 289; // = 121 hex

// Timestamp for 33 bits, PTS, DTS and base of PCR can be used.
typedef struct MY_TIMESTAMP {

   unsigned char byte1;      // Most Significant
   unsigned char byte2;
   unsigned char byte3;
   unsigned char byte4;
   unsigned char byte5;      // Least Significant

} TIMESTAMP;

class TPacket {

  private:
    char data[188]; // The packetdata.
    int pid;
    void SetPID();
    void SetPCR(unsigned int pcr);
    TIMESTAMP CalculateStampBytes(unsigned int value);

  public:

    TPacket(void);
    void Init();
    void Write(char *buf,FILE *file,int start,int size); // Write packet to file.

    void SetUnitStart();  // A packet with a unitstart (pesstart)
    void MakeAdap(int adaptlen, bool random, unsigned int pcrval);
    void MakeAdapPacket(bool random, unsigned int pcr, FILE *file);

};

#endif TP_H
```

tp.c

```
// Implementationfile for the class TPacket.

#include <stdio.h>
#include <stdlib.h>
#include "tp.h"

//-------------------------------------------------
// Constructor for TPacket.
//-------------------------------------------------
TPacket::TPacket(void)
{
  pid = PIDVALUE;
}
//-------------------------------------------------
// Initialization of some values.
//-------------------------------------
void TPacket::Init(void){

  data[0] = 71;    // sync byte
  data[1] = 0;
  data[2] = 0;
  data[3] = 0;
  SetPID();

}
//-------------------------------------------------
// This function makes an adaptationfield packet
//-------------------------------------------------
void TPacket::MakeAdap(int adaptlen, bool random,unsigned int pcr)
{
  if (adaptlen < 1 || adaptlen > 184){
    printf("Error, adaplen out of bounds: %i\n",adaptlen);
    exit(-1);
  }
  // set adaptation control bytes
  if(adaptlen == 184)data[3] = (data[3]&207) + 32;
  else data[3] = (data[3]&207) + 48;

  data[4] = adaptlen-1; // -1 : not counting this byte
  if (adaptlen > 1) data[5] = 0;

  if (random) data[5] = (data[5]&191) + 64; // set RAI!!

  int stuffstart = 6;
  if(pcr>0){
    SetPCR(pcr);
    stuffstart += 6;
  }

  for (int i = stuffstart ; i < 4 + adaptlen ; i++){
    data[i] = 255; // stuffing
  }

  printf("Wrote adaptation field: %i bytes\n",adaptlen);
}
//-------------------------------------------------
// This function writes the packet to file.
//-------------------------------------------------
void TPacket::Write(char *buf,FILE *file,int start,int size)
{
   if (size > 184) {
     printf("Error, tpdata bigger than 184!!");
     exit(-1);
   }

   int adaptlen = 0;

   // Writes an adaptationfield if necessary.
   if (size<184){
     adaptlen = 184-size;
     MakeAdap(adaptlen,false,0); // no pcr...
   }
   else{
     data[3] = (data[3]&239) + 16; //set AF control-payload only
   }

   memcpy(&data[4+adaptlen],&buf[start],size);
```

```
   fwrite(data,1,188,file);
// printf("Writing TP: %i eff. bytes\n", size);

}
//----------------------------------------
// This function is Only for creating a packet with
// only adapt. field + PCR. (no data).
//----------------------------------------
void TPacket::MakeAdapPacket(bool random,unsigned int pcr,FILE *file)
{
   printf("Writing adaptation field only packet with PCR %i\n",pcr);
   if(random) printf("Random access!!!!\n");
   MakeAdap(184,random,pcr);
   fwrite(data,1,188,file);
}
//------------------------------------------------
// This function sets the unit start bit for the packet.
//------------------------------------------------
void TPacket::SetUnitStart(void)
{
   data[1] = (data[1]&191) + 64;
}
//----------------------------------------
// This function sets the PID value.
//----------------------------------------
void TPacket::SetPID(void)
{
  unsigned int value = pid;

  data[2] = value&255;
  value>>8;
  data[1] = (data[1]&224) + value;

}
//-------------------------------------
// This function sets the PCR value.
//-------------------------------------
void TPacket::SetPCR(unsigned int pcrval)
{
   TIMESTAMP pcr;
   // set pcr flag
   data[5] = (data[5]&239) + 16;

   pcr = CalculateStampBytes(pcrval);

   // See MPEG-2 Systems specification.
   data[6] = (pcr.byte2 & 254)/2;
   data[7] = ((pcr.byte2 & 1) * 128) + ((pcr.byte3 & 254)/2);
   data[8] = ((pcr.byte3 & 1) * 128) + ((pcr.byte4 & 254)/2);
   data[9] = ((pcr.byte4 & 1) * 128) + ((pcr.byte5 & 254)/2);
   data[10] = ((pcr.byte5 & 1) * 128) + 126; // + marker bits + 0(ext)
   data[11] = 0; // rest of extension field, set this to 0.

}
//------------------------------------------------
// This function calculates the timestamp bytes from a
// value.
//------------------------------------------------
TIMESTAMP TPacket::CalculateStampBytes(unsigned int value)
{

   TIMESTAMP time;

   time.byte5 = value&255;
   value >>= 8;
   time.byte4 = value&255;
   value >>= 8;
   time.byte3 = value&255;
   value >>= 8;
   time.byte2 = value&255;
   time.byte1 = 0; // so far...

   return time;
}
//-------------------------------------
// A useful function for testing the bytes set
//-------------------------------------
/*void TPacket::WriteByte(char byte)
{
   // just for testing
   int val=128;
```

```
   for (int j=0;j<8;j++) {
      if((byte & val)>0) printf("1");
      else printf("0");
        val=val/2;
      }
   printf("\n");
}*/
```

Klassen Packet

tpacket.h

```
// This class performs operations on a tranport packet. It can load a packet,
// modify timestamps, set certain bits, and return information about the packet.

#ifndef _PACKET_H
#define _PACKET_H

#include "../def.h"
#include <fstream.h>
#include <stdio.h>
#include <string.h>


// Transport packet type values: None (not loaded yet),
// Normal, Adaptation Field, Unit Start, Adaptation Field and Unit Start
typedef enum {NONE, NORMAL, AF, US, AF_US} TP_TYPE;

// Stream type id's for PES layer: video/audio + others (see page 30 in 13818-1);
typedef enum {NOID, PSMAP, PRIVATE1, PADDING, PRIVATE2, AUDIO, VIDEO, ECM,
              EMM, DSMCC, ISOIEC, RESERVED_ID, PSDIR} STREAMID;

// Trick mode values: None, Slowmotion forward/backward, Fastforward, Fastbackward.
typedef enum {NOTRICK, SLOWF, SLOWR, FASTF, FASTR, FREEZE, RESERVED_TRICK} TRICKMODE;


// Timestamp for 33 bits, PTS, DTS and base of PCR can be used.
typedef struct MY_TIMESTAMP {

       unsigned char byte1;  // Most Significant
       unsigned char byte2;
       unsigned char byte3;
       unsigned char byte4;
       unsigned char byte5;  // Least Significant

} TIMESTAMP;

class Packet {
private:

    char           m_data[TP_SIZE];  // The content of the transport packet
    TP_TYPE        m_type;
    bool           m_loaded;         // Datapacket is loaded.
    int            m_PTSoffset; // Offset to start of PTS-field. (if PTS exists)
    int            m_DTSoffset; // Offset to start of DTS-field. (if DTS exists)

    // methods (private);
    bool         CalculateType (void);
    unsigned int CalculateStampValue(TIMESTAMP);
    TIMESTAMP    CalculateStampBytes(unsigned int value);
    int          GetAFLength(void);
    STREAMID     CalculateStreamID(void);
    // Checks if a PES-packet exists and returns offset to the start.
    bool         CalculatePESStart(int &start);
    // Checks if the trickbyte exists and returns offset to it.
    bool         CalculateTrickByteOffset(int &trickoffset);

public:

    // methods (public);

    Packet();
   ~Packet();

       void LoadData(char *ptr);
       void Modify(void);
       void ReturnData(char *ptr);


       // functions for checking if fields/values exist in packet.
       bool RandomAccess(void);
       bool Discontinuity(void);
```

```
        bool SplicingPoint(void);
        bool PCR(void);
        bool PTS(void);
        bool DTS(void);

        int  GetPID(void);                  // Get PID value.
        bool GetTrickMode(TRICKMODE &trick); // Returns false if trickmode is not used.


        // Get calculated values
       bool GetPCRValue(unsigned int &value);
       bool GetPTSValue(unsigned int &value);
       bool GetDTSValue(unsigned int &value);

        // Get timestamps as 5 bytes.
       bool GetPCRStamp(TIMESTAMP &pcr);
       bool GetPTSStamp(TIMESTAMP &pts);
       bool GetDTSStamp(TIMESTAMP &dts);

       // Change values
        bool ChangePCR(unsigned int value);
        bool ChangePTS(unsigned int value);
        bool ChangeDTS(unsigned int value);
        bool SetDiscontinuity();


};

#endif _PACKET_H
```

tpacket.c

```
#include "tpacket.h"

Packet::Packet()
{

   m_loaded = false;    // Data not loaded yet.
   m_type   = NONE;
   m_PTSoffset = 0;
   m_DTSoffset = 0;

}
//-----------------------------------------

Packet::~Packet()
{

}
//-----------------------------------------
void Packet::LoadData(char *ptr)
{
    memcpy (m_data, ptr, TP_SIZE);
    if (m_data[0] == 71) m_loaded = true; // Check that the first byte is the sync byte.
    CalculateType();   // Calculates the type of packet.

}
//-----------------------------------------
void Packet::Modify()
{
    return;
}

//-----------------------------------------
void Packet::ReturnData(char *ptr)
{
    memcpy (ptr, m_data, TP_SIZE);
}
//-----------------------------------------
bool Packet::RandomAccess()
{
    // Check that this packet contains an adaptation field.
    if ((m_type != AF) && (m_type != AF_US)) return false;

    // Is the random access indicator set?
    if ((m_data[5]&64) == 64) return true;
    else return false;
}
//-----------------------------------------
bool Packet::Discontinuity()
{
    // Check that this packet contains an adaptation field.
    if ((m_type != AF) && (m_type != AF_US)) return false;

    // Is the discontinuity indicator set?
    if ((m_data[5]&128) == 128) return true;
    else return false;

}
//-----------------------------------------
bool Packet::SplicingPoint()
{
    // Check that this packet contains an adaptation field.
    if ((m_type != AF) && (m_type != AF_US)) return false;

    // Is the splicing point flag set?
    if ((m_data[5]&4) == 4) return true;
    else return false;

}
//-----------------------------------------
bool Packet::PCR()
{

    // Check that this packet contains an adaptation field.
    if ((m_type != AF) && (m_type != AF_US)) return false;
```

```
    // Is the PCR flag set?
    if ((m_data[5]&16) == 16) return true;
    else return false;

}
//-----------------------------------------
bool Packet::PTS()
{
    int PES_start = 0;

    if (!CalculatePESStart(PES_start)) return false;

    // Gets streamID, if the stream is of one of some given id's, return false (see 13818-1
p27).
    STREAMID id = CalculateStreamID();
    if (id == PSMAP || id == PADDING || id == PRIVATE2 || id == ECM ||
              id == EMM || id == PSDIR) return false;

    // Calculates offset to the byte which contains the PTS valueflag.
    int offset = PES_start + 7;

    // Makes sure the PTS value exists in this packet.
    if (offset > 181) return false;

    // Is PTS flag set?
    if ((m_data[offset]&128) == 128) { m_PTSoffset = offset + 2; return true;}
       else return false;

}
//-----------------------------------------
bool Packet::DTS()
{
    int PES_start = 0;

    if (!CalculatePESStart(PES_start)) return false;

    // Gets streamID, if the stream is of one of some given id's, return false (see 13818-1
p27).
    STREAMID id = CalculateStreamID();
    if (id == PSMAP || id == PADDING || id == PRIVATE2 || id == ECM ||
              id == EMM || id == PSDIR) return false;

    // Calculates offset to the byte which contains the DTS valueflag.
    int offset = PES_start + 7;

    // Makes sure the DTS value exists in this packet.
    if (offset > 176) return false;

    // Is DTS flag set?
    if ((m_data[offset]&64) == 64) { m_DTSoffset = offset + 7; return true;}
       else return false;

}
//-----------------------------------------
int Packet::GetPID (void)
{

   if (m_loaded == false) return -1;

   int pid = 0;
   pid = (m_data[2]&255) + (m_data[1]&31)*256;

   return pid;

}
//-----------------------------------------
bool Packet::GetTrickMode (TRICKMODE &trick)
{

   trick = NOTRICK;
   if (m_loaded == false) return false;

   int trickoffset = 0;
   // Checks if trickbyte exists and gets offset to it if it does.
   if (!CalculateTrickByteOffset(trickoffset)) return false;

   if (trickoffset > 187) return false;  // Just to make sure.

   if      ((m_data[trickoffset] & 224) == 0)   trick = FASTF;
   else if ((m_data[trickoffset] & 224) == 32)  trick = SLOWF;
   else if ((m_data[trickoffset] & 224) == 64)  trick = FREEZE;
   else if ((m_data[trickoffset] & 224) == 96)  trick = FASTR;
```

```
   else if ((m_data[trickoffset] & 224) == 128) trick = SLOWR;
   else trick = RESERVED_TRICK;

   return true;
}
//-----------------------------------------
bool Packet::GetPCRValue (unsigned int &value)
{
    if (!PCR() || m_loaded == false) return false;

        TIMESTAMP pcr;
        GetPCRStamp(pcr);
        value = CalculateStampValue(pcr);
        return true;

}
//-----------------------------------------
bool Packet::GetPTSValue (unsigned int &value)
{
    if (!PTS() || m_loaded == false) return false;

        TIMESTAMP pts;
        GetPTSStamp(pts);
        value = CalculateStampValue(pts);
        return true;

}
//-----------------------------------------
bool Packet::GetDTSValue (unsigned int &value)
{
    if (!DTS() || m_loaded == false) return false;

        TIMESTAMP dts;
        GetDTSStamp(dts);
        value = CalculateStampValue(dts);
        return true;

}

//-----------------------------------------
bool Packet::GetPCRStamp(TIMESTAMP &pcr)
{

        if (!PCR() || m_loaded == false) return false;

        int offset = 6; // this byte contains the start of the PCR field.

        // See transport packet structure in MPEG-2 spesification to understand this.
        // PCR bit values are extracted and inserted into the pcr structure.

        pcr.byte5 = ((m_data[offset+4]&128)/128) + ((m_data[offset+3]&127)*2);
        pcr.byte4 = ((m_data[offset+3]&128)/128) + ((m_data[offset+2]&127)*2);
        pcr.byte3 = ((m_data[offset+2]&128)/128) + ((m_data[offset+1]&127)*2);
        pcr.byte2 = ((m_data[offset+1]&128)/128) + ((m_data[offset+0]&127)*2);
        pcr.byte1 = ((m_data[offset+0]&128)/128);

        return true;

}
//-----------------------------------------
bool Packet::GetPTSStamp(TIMESTAMP &pts)
{

        if (!PTS() || (m_loaded == false) || (m_PTSoffset == 0)) return false;

        // See transport packet structure in MPEG-2 spesification to understand this.
        // PTS bit values are extracted and inserted into the pts structure.

        pts.byte5 = ((m_data[m_PTSoffset+4]&254)/2) + ((m_data[m_PTSoffset+3]&1)*128);
        pts.byte4 = ((m_data[m_PTSoffset+3]&254)/2) + ((m_data[m_PTSoffset+2]&2)*64);
        pts.byte3 = ((m_data[m_PTSoffset+2]&252)/4) + ((m_data[m_PTSoffset+1]&1)*64) +
((m_data[m_PTSoffset+1]&2)*128);
        pts.byte2 = ((m_data[m_PTSoffset+1]&252)/4) + ((m_data[m_PTSoffset+0]&2)*32) +
((m_data[m_PTSoffset+0]&4)*32);
        pts.byte1 = ((m_data[m_PTSoffset+0]&8)/8);

        return true;

}
//-----------------------------------------
bool Packet::GetDTSStamp(TIMESTAMP &dts)
{
```

```
        if (!DTS() || (m_loaded == false) || (m_DTSoffset == 0)) return false;

        // See transport packet structure in MPEG-2 spesification to understand this.
        // DTS bit values are extracted and inserted into the dts structure.

        dts.byte5 = ((m_data[m_DTSoffset+4]&254)/2) + ((m_data[m_DTSoffset+3]&1)*128);
        dts.byte4 = ((m_data[m_DTSoffset+3]&254)/2) + ((m_data[m_DTSoffset+2]&2)*64);
        dts.byte3 = ((m_data[m_DTSoffset+2]&252)/4) + ((m_data[m_DTSoffset+1]&1)*64) +
((m_data[m_DTSoffset+1]&2)*128);
        dts.byte2 = ((m_data[m_DTSoffset+1]&252)/4) + ((m_data[m_DTSoffset+0]&2)*32) +
((m_data[m_DTSoffset+0]&4)*32);
        dts.byte1 = ((m_data[m_DTSoffset+0]&8)/8);

        return true;

}
//-------------------------------------------------
bool Packet::ChangePCR(unsigned int value)
{
   TIMESTAMP pcr;

   if (!PCR() || m_loaded == false) return false;

   int offset = 6; // this byte contains the start of the PCR field.

   pcr = CalculateStampBytes(value);

   m_data[offset+0] = (pcr.byte2 & 254)/2;
   m_data[offset+1] = ((pcr.byte2 & 1) * 128) + ((pcr.byte3 & 254)/2);
   m_data[offset+2] = ((pcr.byte3 & 1) * 128) + ((pcr.byte4 & 254)/2);
   m_data[offset+3] = ((pcr.byte4 & 1) * 128) + ((pcr.byte5 & 254)/2);
   m_data[offset+4] = ((pcr.byte5 & 1) * 128) + (m_data[offset+4]&127); // + existing bits...

   return true;
}
//-------------------------------------------------
bool Packet::ChangePTS(unsigned int value)
{
    TIMESTAMP pts;

    if (!PTS() || m_loaded == false) return false;

    pts = CalculateStampBytes(value);

    m_data[m_PTSoffset+0] = (m_data[m_PTSoffset+0]&241) + ((pts.byte2 & 192)/32); // existing
bits
    m_data[m_PTSoffset+1] = ((pts.byte2 & 63)*4) + ((pts.byte3 & 192)/64);
    m_data[m_PTSoffset+2] = ((pts.byte3 & 63)*4) + ((pts.byte4 & 128)/64) + 1; // markerbit ->
+1!!
    m_data[m_PTSoffset+3] = ((pts.byte4 & 127)*2) + ((pts.byte5 & 128)/128);
    m_data[m_PTSoffset+4] = ((pts.byte5 & 127)*2) + 1;                          // markerbit ->
+1!!

    return true;
}
//-------------------------------------------------
bool Packet::ChangeDTS(unsigned int value)
{
    TIMESTAMP dts;

    if (!DTS() || m_loaded == false) return false;

    dts = CalculateStampBytes(value);

    m_data[m_DTSoffset+0] = (m_data[m_DTSoffset+0]&241) + ((dts.byte2 & 192)/32); // existing
bits
    m_data[m_DTSoffset+1] = ((dts.byte2 & 63)*4) + ((dts.byte3 & 192)/64);
    m_data[m_DTSoffset+2] = ((dts.byte3 & 63)*4) + ((dts.byte4 & 128)/64) + 1; // markerbit ->
+1!!
    m_data[m_DTSoffset+3] = ((dts.byte4 & 127)*2) + ((dts.byte5 & 128)/128);
    m_data[m_DTSoffset+4] = ((dts.byte5 & 127)*2) + 1;                          // markerbit ->
+1!!

    return true;
}
//-------------------------------------------------
bool Packet::SetDiscontinuity(void)
{

   // Check that this packet contains an adaptation field.
   if ((m_type != AF) && (m_type != AF_US)) return false;
```

```
   // Set the discontinuity-indicator.
   m_data[5] = ((m_data[5]&127) + 128);

   return true;

}
//-------------------------------------------------
int Packet::GetAFLength(void)
{

   // No adaptation field present in this packet.
   if (m_type != AF_US && m_type != AF) return 0;

   // returns the length of the adaptation field.
   return ((int) m_data[4] & 255) + 1; // + 1 :adaptation length + THIS BYTE

}
//-----------------------------------------
unsigned int Packet::CalculateStampValue(TIMESTAMP time)
{
    // Missing stupid bit here but....

    unsigned int value = 0;

    value = value + time.byte2;
    value <<=8;
    value = value + time.byte3;
    value <<=8;
    value = value + time.byte4;
    value <<=8;
    value = value + time.byte5;

    return value;
}
//-----------------------------------------
TIMESTAMP Packet::CalculateStampBytes(unsigned int value)
{

   TIMESTAMP time;

   time.byte5 = value&255;
   value >>= 8;
   time.byte4 = value&255;
   value >>= 8;
   time.byte3 = value&255;
   value >>= 8;
   time.byte2 = value&255;

   time.byte1 = 0; // so far...

   return time;
}
//-----------------------------------------
bool Packet::CalculateType()
{

  if (m_loaded == false) return false;

  int offset = 1;

  // offset points to payload unit start indicator byte, offset + 2 to adaptation field
control.

  // Returns the correct transportpacket type.

  if ((m_data[offset+2]&48) == 48 && (m_data[offset]&64) == 64) m_type = AF_US;
  else if ((m_data[offset+2]&48) == 32 || (m_data[offset+2]&48) == 48) m_type = AF;

  // Checks payload unit start indicator bit.
  else if ((m_data[offset]&64) == 64) m_type = US;

  else m_type = NORMAL;

  return true;

}
//-------------------------------------------------
STREAMID Packet::CalculateStreamID(void)
{

  int PES_start = 0;
```

```
  // Checks that this packet contains a PES-start and returns the offset.
  if (!CalculatePESStart(PES_start)) return NOID;

  // Calculates offset to the byte which contains the stream type.
  int offset = PES_start + 3;

  if (offset > 187) return NOID;

  // Returns the correct type of ID, examining the stream_id field.
  if      ((m_data[offset] & 224) == 192) return AUDIO;
  else if ((m_data[offset] & 240) == 224) return VIDEO;
  else if ((m_data[offset] & 255) == 188) return PSMAP;
  else if ((m_data[offset] & 255) == 189) return PRIVATE1;
  else if ((m_data[offset] & 255) == 190) return PADDING;
  else if ((m_data[offset] & 255) == 191) return PRIVATE2;
  else if ((m_data[offset] & 255) == 240) return ECM;
  else if ((m_data[offset] & 255) == 241) return EMM;
  else if ((m_data[offset] & 255) == 242) return DSMCC;
  else if ((m_data[offset] & 255) == 243) return ISOIEC;
  else if ((m_data[offset] & 255) == 255) return PSDIR;
  else return RESERVED_ID;

}
//-----------------------------------------------------
bool Packet::CalculatePESStart(int &start)
{

   // Check that this packet contains a unit start (PES-packet/PSI packet).
   if (m_type != AF_US && m_type != US) return false;

   // The header of the transportpacket is 4 bytes.
   start = 4;

   // If adaptation field exists, move the PES-start further.
   if (m_type == AF_US) start += GetAFLength();

   // Check that this really is a PES-packet and not a PSI-packet. (packet start code)
   if ( ((int)m_data[start] != 0) || ((int)m_data[start+1] != 0) ||
      ((int)m_data[start+2] != 1) ) return false;

   return true;

}
//-----------------------------------------------------
bool Packet::CalculateTrickByteOffset(int &trickoffset)
{

   int PESstart = 0;
   // Check that the packet contains the start of a PES-packet and gets offset.
   if (!CalculatePESStart(PESstart)) return false;

   // Gets streamID, if the stream is of one of some given id's, return false (see 13818-1
p27).
   STREAMID id = CalculateStreamID();
   if (id == PSMAP || id == PADDING || id == PRIVATE2 || id == ECM ||
       id == EMM || id == PSDIR) return false;

   // Calculates offset to the byte which contains the DTS valueflag.
   int offset = PESstart + 7;

   // Checks trickmode flag.
   if ((m_data[offset] & 8) == 0) return false;

   // Calculates new offset to trickmode byte:
   trickoffset = offset + 1;   // skip headerlength-byte...
   if ((m_data[offset] & 128) == 128) trickoffset += 5;  // PTS is set.
   if ((m_data[offset] & 64) == 64) trickoffset += 5;    // DTS is set.
   if ((m_data[offset] & 32) == 32) trickoffset += 6;    // ESCR is set.
   if ((m_data[offset] & 16) == 16) trickoffset += 3;    // ESRATE is set.
   // trickoffset should now point to the trickbyte.

   return true;

}
```

Vedlegg 2: Parameterfil for mpeg2encode

Her følger en beskrivelse av hvert innslag i parameterfila som mpeg2encode benytter. Hver linje i tabellen tilsvarer en linje i tekstfila. Se dokumentasjon for mpeg2encode for en nærmere beskrivelse av hvert innslag.

Eksempel	Forklaring
MPEG-2 Test Sequence	Første linje ignoreres, her kan det legges en tekst
utfil%d	Navn på kildefilene
-	Navn på rammer om disse mellomlagres. (- : lagrer ikke)
-	Navn på intra-kvantiseringsmatrise. (- : bruk standard)
-	Navn på inter-kvantiseringsmatrise. (- : bruk standard)
stat.out	Navn på statistikkfil. (- : stdout)
2	Format på inputfiler. (2 = PPM)
100	Antall rammer som skal kodes.
0	Det første rammenummeret. (0: fil0.ppm)
00:00:00:00	Tidskoden til første ramme. Legges inn i strømmen.
12	Antall rammer i GOP
3	I/P-ramme avstand.
0	Type MPEG-format. (0: MPEG2 1: MPEG1.)
0	Ramme/field som inputbilder. (0: Ramme)
704	Horisontal størrelse på bilder.
576	Vertikal størrelse på bilder.
2	Aspect Ratio (2 = 4:3)
3	Rammerate (3 = 25)
5000000	Bitrate (i bit/s)
112	Størrelse på nødvendig input-buffer i dekoder.
0	1 = Low-delay mode.
0	Constrained parameter. (Alltid 0 for MPEG-2)
4	Profil (4 = Main)
8	Nivå (Level) (8 = Main)
0	Progressiv sekvens. (0 = Interlaced)
1	Makroblokkformat (1 = 4:2:0)
1	Videoformat (1 = PAL)
5	Color-primaries.
5	Transfer-characteristics.
5	Matrise-koeffisienter.
704	Horisontal størrelse på display.
576	Vertikal størrelse på display.
0	Intra DC-presisjon. (0: 8 bit.)
1	Kod Top-field først. (Kun for interlacing)
0 0 0	Bevegelseskompensasjon (I P B). 1 1 1 : Kun rammeprediksjon.
0 0 0	«Concealment»-bevegelsesvektorer (I P B)
1 1 1	Kvantisering-skalatype (1: ikke-lineær).
1 0 0	Intra-vlc-format. (0: tabell 0 (MPEG-1) 1: tabell 1.)
0 0 0	Alternativt skanningsmønster. (0: standard)
0	Gjenta første felt.
0	Progressiv ramme. (0: interlacet ramme)
0	Intra-slice refresh picture period.
0	Rate-kontroll: reaction parameter

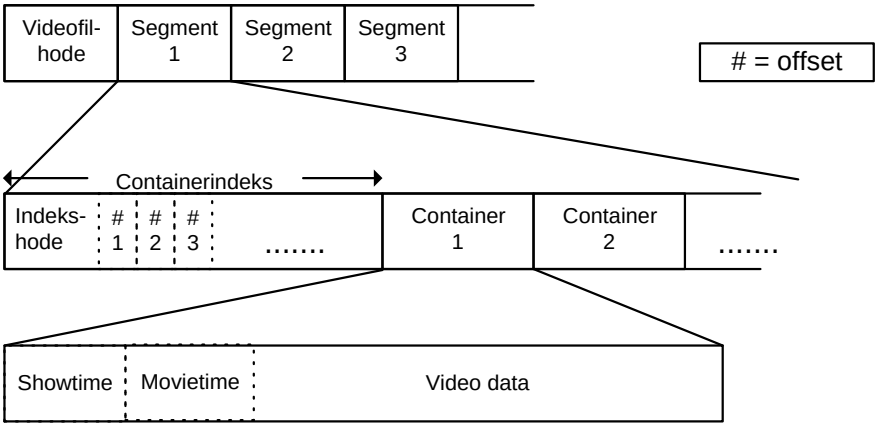
0	Rate-kontroll: intitial average activity.
0	Rate-kontroll: Xi
0	Rate-kontroll: Xp
0	Rate-kontroll: Xb
0	Rate-kontroll: d0i
0	Rate-kontroll: d0p
0	Rate-kontroll: d0b
2 2 11 11	Bevegelsesvektorer, maksimal lengde (forover) - P-ramme
1 1 3 3	Bevegelsesvektorer, maksimal lengde forover - B1-ramme
1 1 7 7	Bevegelsesvektorer, maksimal lengde bakover- B1-ramme
1 1 7 7	Bevegelsesvektorer, maksimal lengde forover- B2-ramme
1 1 3 3	Bevegelsesvektorer, maksimal lengde bakover- B2-ramme

Vedlegg 3: Elvira II filformat

Dette dokumentet beskriver filformatet for videofiler i et Elvira II filsystem.

Overordnet struktur

En Elvira II fil består av et videofilhode, og en rekke segmenter. Både videofilhodet og segmentene vil ha en fast størrelse. Et segment definerer enheten som vil leses fra disk og vil typisk være i området 64kB - 512kB. Hvert segment vil inneholde containere, se Figur 53 for en illustrasjon av oppbygningen.



Figur 53 Oppbygging av en Elvira II fil

Videohode

Videohodet til filmen vil inneholde informasjon om filmen som er lagret. Størrelsen er fast og defineres av konstanten MF_HDRSIZE i fila `pump/moviefile.h`. Følgende tabell definerer innholdet:

Feltnavn	Datatype	Størrelse	Forklaring
magic	int	4	Angir kodeformat for videoen.
name	char	MF_NAMELEN	Navnet på filmen.
segsize	int	4	Størrelsen til segmentene.
nsegs	int	4	Antall segmenter i fila.
ncontainers	int	4	Antall containere i fila.
firstsegoffset	int	4	Offset til første segment i fila.
firstdiroffset	int	4	?

Disse dataene vil ta mindre plass enn størrelsen til hodet, resten av hodet vil være tomt. Flere innslag kan derfor settes inn senere dersom det skulle være ønskelig.

Containerindeks

En containerindeks sier hvor containere ligger i dette segmentet. Den består av følgende:

- **Container Index Header:** (indekshode) Dette er ett felt som angir antall innslag i containereindeksen (antall containere i segmentet). Denne vil være av datatypen int.

- **Container Index Entries:** For hver container som ligger i segmentet etter containerindeksen vil det finnes et slikt innslag. Et innslag er beskrevet i følgende tabell:

Feltnavn	Datatype	Størrelse	Forklaring
movietime	NPTime	sizeof (NPTime) = 4	Angir tidspunktet i filmen. Format: millisekunder.
offset	int	4	offset (avstand) fra starten av segmentet til begynnelsen av containeren.
size	int	4	Lengden til en container.

Datatypen NPTime er definert i filen `common/nptime.h`

Containere

En container definerer enheten for oversendelse over nettverk. Hodet til containere er bygd opp som vist i følgende tabell:

Feltnavn	Datatype	Størrelse	Forklaring
showtime	NPTime	sizeof(NPTime) = 4	Tidspunkt for visning av dataene i containeren.
movietime	NPTime	sizeof(NPTime) = 4	Tidspunktet i filmen disse dataene representerer.

Resten av containeren vil være videodata, som vist i Figur 53.

Vedlegg 4: Indeksfiler

Dette dokumentet beskriver indeksfilene som skal benyttes av Elvira II. Det finnes to typer; en som kalles indeksfil (movieindex) og en som kalles aksess-indeksfil.

Indeksfil (Movieindex)

Elvira II vil benytte følgende indeksfil til å identifisere segmenter som tilhører en avspillingsfil. Tidsinformasjon knyttes til hvert segment og brukes ved innhenting av segmenter. Indeksfilen inneholder indekser for alle avspillingsfilene (instanser) som tilhører en film.

Format:

```
name "title"
length seconds
instance instanceno speed format framerate segsize nsegments ncontainers
time segmentid offset  (for each segment in the instance)
```

Her er segmentid = maskinid:diskid:blokknummer.
Setningen som inneholder intansinformasjon vil gjentas for hver avspillingsfil.

Eksempel:

```
name "Pooltime"
length 5435.42
instance 1 1/1 MPEG2 25 131072 291 4652
0.000 0:0:8 0
0.133 0:0:264 0
0.260 0:0:520 0
...
instance 2 4/1 MPEG2 25 131072 50 795
0.001 0:0:218688 0
0.736 0:0:218944 0
1.472 0:0:219200 0
...
```

Aksess-indeksfil

Denne filen brukes til å identifisere aksesspunkt i en avspillingsfil. Tidspunkt og lokasjon knyttes til hvert aksesspunkt. Indekser for flere avspillingsfiler tilhørende en film skal ligge i samme fil, dette er ikke inkludert i denne beskrivelsen (formatet er ikke bestemt på det nåværende tidspunkt).

Format:

```
time disk(T/F) segmentid containerid
```

segmentid er maskinid:diskid:blokknummer

Eksempel:

```
0.001 T 0:0:119336 1
1.567 T 0:0:120104 54
3.410 T 0:0:121128 118
```

. . .

Vedlegg 5: MPEG-2 Videostrøm

Dette dokumentet beskriver en video-elementærstrøm som pseudokode. Informasjonen er hentet fra ISO/IEC 13818-2. Faktiske bit i strømmen er uthevet.

Som vi ser av strømmen vil `sequence_header` være felles for MPEG-1 og MPEG-2. For MPEG-2, vil sekvenshodet følges av `sequence_extension`, et tillegg til sekvenshodet.

Vi beskriver her bare den overordnete strukturen, samt innholdet i sekvenshodet og sekvenstillegget. For en beskrivelse av de andre bestanddelene henvises det til ISO/IEC 13818-2.

Syntaks : Videosekvens	**Ant. bit**
video_sequence() {	
next_start_code()	
sequence_header()	
if(nextbits() == extension_start_code) {	
sequence_extension()	
do {	
extension_and_user_data(0)	
do {	
if(nextbits() == group_start_code) {	
group_of_pictures_header()	
extension_and_user_data(1)	
}	
picture_header()	
picture_coding_extension()	
extension_and_user_data(2)	
picture_data()	
} while ((nextbits() == picture_start_code ||	
(nextbits() == group_start_code))	
if(nextbits() != sequence_end_code) {	
sequence_header()	
sequence_extension()	
}	
} while (nextbits() != sequence_end_code)	
} else {	
/* ISO/IEC 11172-2 (MPEG-1)*/	
}	
sequence_end_code	32
}	

Syntaks : Sekvenshode	**Ant. bit**
sequence_header() {	
sequence_header_code	32
horizontal_size_value	12
vertical_size_value	12
aspect_ration_information	4
frame_rate_code	4
bit_rate_value	18
marker_bit	1
vbv_buffer_size_value	10
constrained_parameters_matrix	1
load_intra_quantiser_matrix	1
if (load_intra_quantiser_matrix)	
intra_quantiser_matrix[64]	8*64
load_non_intra_quantiser_matrix	1
if (load_non_intra_quantiser_matrix)	
non_intra_quantiser_matrix[64]	8*64
next_start_code()	
}	

Syntaks : Sekvenstillegg	**Ant. bit**
sequence_extension() {	
extension_start_code	32
extension_start_code_identifier	4
profile_and_level_indication	8
progressive_sequence	1
chroma_format	2
horizontal_size_extension	2
vertical_size_extension	2
bit_rate_extension	12
marker_bit	1
vbv_buffer_size_extension	8
low_delay	1
frame_rate_extension_n	2
frame_rate_extension_d	5
next_start_code()	
}	

Vedlegg 6: Transportstrømmen

Informasjonen i dette dokumentet er hentet fra ISO 13818-1, og beskriver transportstrømmen ved bruk av pseudokode. Faktiske bit i bitstrømmen er uthevet.

Syntaks : MPEG-2 tranportstrøm
MPEG_transport_stream() {
do {
transport_packet()
} while (nextbits() == sync_byte)
}

Syntaks : Transportpakke	**Ant. bit**
transport_packet() {	
sync_byte	8
transport_error_indicator	1
payload_unit_start_indicator	1
transport_priority	1
PID	13
transport_scrambling_control	2
adaptation_field_control	2
continuity_counter	4
if (adaptation_field_counter == ‘10’ || adaptation_field_counter == ‘11’) {	
adaptation_field()	
}	
if (adaptation_field_counter == ‘01’ || adaptation_field_counter == ‘11’) {	
for (i=0;i<N;i++) {	
data_byte	8
}	
}	
}	

Syntaks : Adaptation field	**Ant. bit**
adaptation_field() {	
adaptation_field_length	8
if (adaptation_field_length>0) {	
discontinuity_indicator	1
random_access_indicator	1
elementary_stream_priority_indicator	1
PCR_flag	1
OPCR_flag	1
splicing_point_flag	1
transport_private__data_flag	1
adaptation_field_extension_flag	1
if (PCR_flag == ‘1’) {	
program_clock_reference_base	33
reserved	6
program_clock_reference_extension	9
}	
if (OPCR_flag == ‘1’) {	
original_ program_clock_reference_base	33
reserved	6
original_ program_clock_reference_extension	9
}	
if (splicing_point_flag == ‘1’) {	
splice_countdown	8
}	
if(transport_private_data_flag == ‘1’) {	
transport_private_data_length	8
for (i=0;i< transport_private_data_length;i++) {	
private_data_byte	8
}	
}	
if(adaptation_field_extension_flag == ‘1’) {	
adaptation_field_extension_length	8
ltw_flag	1
piecewise_rate_flag	1
seamless_splice_flag	1
reserved	5
if (ltw_flag == ‘1’) {	

```
                        ltw_valid_flag                                  1
                        ltw_offset                                     15
                }
                if (piecewise_rate_flag == '1') {
                        reserved                                        2
                        piecewise_rate                                 22
                }
                if (seamless_splice_flag == '1') {
                        splice_type                                     4
                        DTS_next_au[32..30]                             3
                        marker_bit                                      1
                        DTS_next_au[29..15]                            15
                        marker_bit                                      1
                        DTS_NEXT_au[14..0]                             15
                        marker_bit                                      1
                }
                for (i=0;i<N;i++) {
                        reserved                                        8
                }
        for (i=0;i<N;i++) {
                stuffing_byte                                           8
        }
}
```

Syntaks : PES pakke	Ant. bit
PES_packet () {	
packet_start_code_prefix	24
stream_id	8
PES-packet_length	16
if(stream_id != program_:stream_map	
&& stream_id != padding_stream	
&& stream_id != private_stream	
&& stream_id != private_stream_2	
&& stream_id != ECM	
&& stream_id != EMM	
&& stream_id != program_stream_directory) {	
'10'	
PES_scrambling_control	2
PES_priority	2
data_alignement_indicator	1
copyright	1
original_or_copy	1
PTS_DTS_flags	1
ESCR_flag	2
ES_rate_flag	1
DSM_trick_mode_flag	1
additional_copy_info_flag	1
PES_CRC_flag	1
PES_extension_flag	1
PES_header_data_length	8
if (PTS_DTS_flags == '10') {	
'0010'	4
PTS[32..30]	3
marker_bit	1
PTS[29..15]	15
marker_bit	1
PTS[14..0]	15
marker_bit	1
}	
if (PTS_DTS_flags == '11') {	
'0011'	4
PTS[32..30]	3
marker_bit	1
PTS[29..15]	15
marker_bit	1
PTS[14..0]	15
marker_bit	1
'0011'	4
DTS[32..30]	3
marker_bit	1
DTS[29..15]	15
marker_bit	1
DTS[14..0]	15
marker_bit	1
}	
if (ESCR_flag == '1') {	
reserved	2
ESCR_base[32..30]	3
marker_bit	1
ESCR_base[29..15]	15
marker_bit	1
ESCR_base[14..0]	15
marker_bit	1
ESCR_extension	9
marker_bit	1

```
                }
                if (ES_rate_flag == ‘1’) {
                        marker_bit                                              1
                        ES_rate                                                22
                        marker_bit                                              1
                }
                if (DSM_trick_mode_flag == ‘1’) {
                        trick_mode_control                                      3
                        if (trick_mode_control == ‘’000’) {
                                field_id                                        2
                                intra_slice_refresh                             1
                                frequency_truncation                            2
                        }
                        else if (trick_mode_control == ‘001’) {
                                field_rep_cntrl                                 5
                        }
                        else if (trick_mode_control == ‘010’) {
                                field_id                                        2
                                reserved                                        3
                        }
                        else if (trick_mode_control == ‘011’) {
                                field_id                                        2
                                intra_slice_refresh                             1
                                frequency_truncation                            2
                        }
                        else if (trick_mode_control == ‘100’) {
                                field_rep_cntrl                                 5
                        }
                        else {
                                reserved                                        5
                        }
                }
                if (additional_copy_info_flag == ‘1’) {
                        marker_bit                                              1
                        additional_copy_info                                    7
                }
                if (PES_CRC_flag == ‘1’) {
                        previous_PES_packet_CRC                                16
                }
                if (PES_extension_flag == ‘1’) {
                        PES_private_data_flag                                   1
                        pack_header_field_flag                                  1
                        program_packet_sequence_counter_flag                    1
                        P-STD_buffer_flag                                       1
                        reserved                                                3
                        PES_extension_flag_2                                    1
                        if (PES_private_data_flag == ‘1’) {
                                PES_private_data                              128
                        }
                        if (pack_header_field_flag == ‘1’) {
                                pack_field_length                               8
                                pack_header()
                        }
                        if (program_packet_sequence_counter_flag == ‘1’)
{
                                marker_bit                                      1
                                program_packet_sequence_counter                 7
                                marker_bit                                      1
                                MPEG1_MPEG2_identifier                          1
                                original_stuff_length                           6
                        }
                        if (P-STD_buffer_flag == ‘1’) {
                                ‘01’                                            2
                                P-STD_buffer_scale                              1
                                P-STD_buffer_size                              13
                        }
                        if (PES_extension_flag_2 == ‘1’) {
                                marker_bit                                      1
                                PES_extension_field_length                      7
                                for(i=0;i<PES_extension_field_length;i++)
{
                                        reserved                                8
                                }
                        }
                }
                for (i=0;i<N;i++) {
                        stuffing_byte                                           8
                }
                for (i=0;i<N2;i++) {
                        PES_packet_data_byte                                    8
                }
        }
        else if(stream_id == program_:stream_map
        || stream_id == private_stream_2
        || stream_id == ECM
        || stream_id == EMM
        || stream_id == program_stream_directory) {
```

```
                for (i=0;i<PES_packet_length;i++) {
                        PES_packet_data_byte                                    8
                }
        }
        else if (stream_id == padding_stream) {
                for (i=0; i<PES_packet_length;i++) {
                        padding_byte                                            8
                }
        }
}
```