UNIVERSITETET I OSLO
Institutt for informatikk

Fordelen ved å implementere applikasjons spesifikk shaping og køing på et IXP2400

Masteroppgave

Øystein Yri Sunde

30. januar 2006

Forord

Jeg vil gjerne takke mine veiledere Carsten Griwodz og Pål Halvorsen for en utrolig bra støtte og veiledning igjennom hele masterstudie. Jeg vil også takke mine mine studiekameater Andreas Petlund og Tom Anders Dalseng for hjelp ,gode diskusjoner og gjennomlesnng. En takk også til Christian Søhol med litt korrektur lesning. Siste takk til alle gode venner og familie som har hjulpet med støtte.

Sammendrag

Det har blitt mye mer vanlig at vi sender store strømmer med data over Internet. Dette kan være fordi slutt-brukerene har fått raskere linje.. Store strømmer med data til mange forskjellige klienter fører til nye utfordringer. Dette kan være å sørge for at disse strømmene kommer frem med en bra kvalitet. Vi kan løse dette ved å bruke dyr hardware, eller vi kan bruke forskjellige tekknikker for å løse dette. Vi har sett på måter å løse dette ved bruk av nettverksprossessorer. Dette har vi tenkt kombinere med applikasjons spesifikk shaping og prioritetskø. Vi har sett på forskjellige metoder får å løse dette senere i oppgaven.

Innhold

Preface **i**

1 Innledning **1**

1.1 Bakgrunn Og Motivasjon 1

1.2 Problemstilling 5

1.3 Oppgavestruktur 6

2 Bakgrunnsinformasjon **7**

2.1 Video on demand 7

2.2 Lagdelt Video 8

2.3 Prioriteter 10

2.3.1 Prioritetskøing Med Kun En Kø 12

2.3.2 Prioritetskøing Med Flere Køer 12

2.3.3 Evaluering Av Muligheter 13

2.4 Shaper 14

2.5 Multimedia Protokoller 15
2.5.1 SDP 16
2.5.2 RTSP 17
2.5.3 RTP 19
2.6 Sammendrag 21

3 Arkitektur **23**
3.1 Nettverksprosessorer 23
3.1.1 Radisys ENP-2611 24
3.1.2 IXP2400 Brikkesett 26
3.2 Programvare 30
3.2.1 Radisys ENP SDK 3.5 30
3.2.2 Intel IXA SDK 3.51 30
3.3 Pakkebuffersystemet 32
3.4 Evualering 33
3.5 Sammendrag 37

4 Design **38**
4.1 Utforming på grunnlag av Intel SDK 38
4.2 Komponenter Fra SDK 41
4.2.1 Rx 41

4.2.2 Tx 41
4.3 Nye Komponenter 42
4.3.1 Shaper 42
4.3.2 Design Av adgangskontroll 45
4.3.3 Parsing Av Mediakontrollpakker 45
4.3.4 Prioritets Sortering 46
4.3.5 Prioritetsvelger 47
4.3.6 Tidtager 48
4.4 Sammendrag 48

5 Implementasjon **49**
5.1 Shaper 49
5.1.1 Prioritering 53
5.1.2 Prioritetssortering 54
5.1.3 Prioritetsvelger 55
5.2 Sammendrag 55

6 Evaluering **56**
6.1 Testoppsett 56
6.2 Priorities Testing 57
6.3 Shaper Testing 65

6.4 Tidsmåling 68

6.5 Sammendrag 69

7 konklusjon **71**

7.1 Resultat 71

7.2 fremtidig arbeid 72

A Kode for de forskjellige mikromaskinene **77**

A.1 shaper.c 77

A.2 prioritetsortering.c 83

A.3 Prioritetsvelger.c 87

A.4 timer.c 89

Figurer

1.1 tjener topologi 2

1.2 Pakke sortering 6

2.1 temporal pakke lagdeling 9

2.2 Spatial lagdeling 10

2.3 Prioritet med fler Køer 13

2.4 Rtp header 20

3.1 Skjema for ENP-2611[6] 25

3.2 oppsett av IXP2400 brikkesett[2] 26

3.3 Skjema mikromaskin 27

3.4 Bufferdeskriptor 33

3.5 Testsapplikasjonen 34

4.1 Flytskjema for mikromaskinene 40

4.2 token bucket 43

4.3 flytskjema for shaper 44

5.1 shaper Struct 50

5.2 Minne områder for Shaper 53

6.1 Test oppsett 57

6.2 tidsforskjellen mellom 1. prioritets pakker med ixp 60

6.3 tidsforskjellen mellom 1. prioritets pakker med Linux uten last 62

6.4 tidsforskjellen mellom 1. prioritets pakker med Linux uten last 63

6.5 Gjennomsnittlige verdier med tar 65

6.6 Gjennomsnittlige verdier med standard avvik 66

6.7 1100 og 800 kbps 67

6.8 1100 og 700 kbps 68

6.9 1100 og 600 kbps 69

Tabeller

2.1 Eksempel for en DESCRIBE forespørsel 18

2.2 Eksempel for en DESCRIBE svar 18

3.1 Måling av gjennomstrømning for testapplikasjonen 35

3.2 Måling av ping tider 36

6.1 Latens på 1.prioritets pakker ved en 100 Mpbs strøm 59

6.2 Latens på 2.prioritets pakker ved en 100 Mpbs strøm 61

6.3 Latens på 1.prioritets pakker ved en 200 Mpbs strøm 61

6.4 Latens på 2.prioritets pakker ved en 200 Mpbs strøm 61

6.5 Latens på 1.prioritets pakker ved en 400 Mpbs strøm 64

6.6 Tidsmåling med en tar prosess 64

6.7 Måling med 200 uten prioriteter på ixp 64

Kapittel 1

Innledning

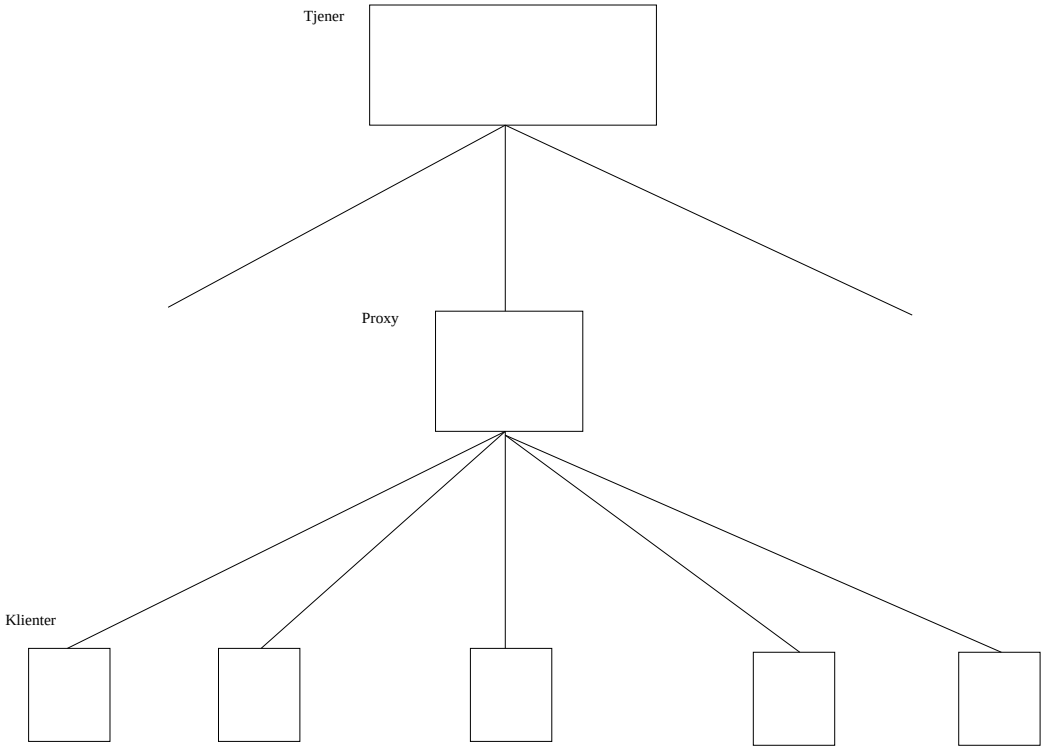
Etter hvert som vi nå får raskere og raskere Internet tilkoblinger, og backbone nettverket er bra utstyrt med fiber, vil man nå kunne bruke nettet til å komme med bedre tilbud for å sende filmer over Internet og mobile nett. Man kan da tilby ulike løsninger for å tilby en del populære filmer som man kan se som ligger på en tjener som kan være enten i nettet til en Internet service provider (ISP) tilbyder, eller koblet til backbone-nettet som kommuniserer bra med ISP-nettet forbrukeren sitter på. Etter hvert vil selskaper kanskje kunne konkurrere med kabel tv og vanlig tv siden de da vil kunne tilby fullstendig bruker interaktivitet, og forbrukeren kan velge hva man vil se når man vil. Man kan også sende vanlige tv programmer [8]og serier, å gi et komplett tv tilbud. Siden vi per dags dato har veldig varierende båndbredde til sluttbrukeren, må vi finne metoder som kan kompensere for dette slik at flest mulig vil kunne benytte seg av tilbud med multimedia over nett.

1.1 Bakgrunn Og Motivasjon

En av metodene vi da kan benytte oss av er å bruke en proxytjener som vi ser fra figur 1.1 som skal stå imellom en tjener og klienten. Vi regner med at vi alltid vil ha god forbindelser mellom tjeneren og proxyen og noe mer varierende forbindelse mellom proxyen og klienten, så det kan være en bra plass å legge inn metoder i proxyen som skal sikre best mulig kvalitet

for strømmene. En proxy vil også kunne avlaste tjeneren siden den vil ta en del av arbeidsoppgavene fra tjeneren. Vi vil også få mindre press på nettnodene til tjeneren siden oppkoblingene vil gå via proxyen.

Figur 1.1 tjener topologi



En av metodene vi kan benytte er prioritisering av pakkene som sendes, siden det vil være veldig fordelaktig å skille de pakkene som er viktige i en strøm fra de som er mindre viktige. Vi vil med denne metoden gradere informasjonen vi skal sende i strømmen, hvor den viktigste informasjonen vil bli lagt i de høyeste prioriterte pakkene. Dette vil hjelpe når klienten eller nettverket ikke kan ta hele båndbredden som strømmen krever, slik at vi kan droppe de laveste prioritetspakkene. Med denne metoden vil klienten få den best mulige kvalitet som er mulig utfra den båndbredden klienten har på sin Internet forbindelse. Siden det da gjerne vil være en del brukere som benytter seg av proxyen vil vi gjerne også begrense trafikken til hver strøm. Det kan da være lurt å benytte seg av en shaper som skal sørge for at ikke en strøm kommer og ødelegger for de andre strømmene.

Det er også bra at for tjeneren at vi bruker en proxy til å avlaste tjeneren

siden det da vil bli mindre å gjøre for tjeneren og den kan ha flere klienter som den kan håndtere.

Vi kan for eksempel bruke prioriteringsmekanismer i online-spill som anarchy online[1], hvor det er masse informasjon som skal flyttes mellom tjeneren og klienten. Online-spill har som regel mange brukere, og vi trenger informasjon om alle med liten forsinkelse for at det skal fungere optimalt, det vil alltid være ting som er mindre viktig informasjon enn andre[12]. Vi kan si at et tre eller himmelen er mindre viktig enn en mulig motstander som skal drepe deg. Det kan derfor være en ide om at vi man prioriterer pakkene slik at vi alltid vil få de pakkene som inneholder informasjon om motstandere og personer fremfor landskapsgrafikk. Det vil også stilles krav til forsinkelse når man skal sende informasjon fra tjeneren til klienten, det kan da også være hensiktsmessig å bruke flere proxyer som snakker med tjeneren og at proxyene også snakker med hverandre på ting som er veldig viktige slik at vi greier å få informasjonen som er viktig for spillere rasket ut til hver enkelt spiller. Med prioritering vil da slik pakkeflyt være mulig ved at vi skiller med prioriteter hvor pakkene skal.

Et annet eksempel på hvor prioritering av pakker kan lønne seg er i video konferanser[9] hvor det stilles høye krav til forsinkelser og synkronisering. Det er da veldig viktig å få de riktige pakkene igjennom slik at deltagerne vil kunne kommunisere på en effektiv måte. Dette kan gjøres ved at vi lagdeler videoen slik at vi har et baselag som vi er nødt til at deltagerne kan kommunisere, og hvis det er vi har mer plass på linjen vil vi kunne legge til flere forbedringslag. Det er på denne måten vi kan prioritere pakkene slik at vi kan få en bedre kvalitet enn hvis vi bare skulle ha sendt etter beste evne. Det stilles også krav til synkronisering mellom tale og bilde slik at deltagerne kan kommunisere på en bra måte at det ikke blir latens hvor alle ender med å snakke i kor på hverandre. Prioritiseringen vil også hjelpe her med å holde de kravene som stilles for at det skal bli på en tilfredstillene måte og at vi møter vår tidsfrister.

Video on demand(VOD) er et felt hvor det også kan lønne seg med prioritering. Siden vi da i mye større grad kan skalere videoen. Vi kan sørge for at vi alltid vil få de viktigste pakkene igjennom. Dette vil da være de pakken vi må ha for å kunne se videoen. De pakkene vi da kategoriserer som mindre viktige vil da være ting som gjør videoen bedre. Dette vil være fordelaktig fordi vi da vil kunne gi et bredere tilbud for de som vil ha VOD. Klienten trenger nødvendigvis ikke ha den beste linjen får å bruke VOD. Vi vil også sørge for kvalitetssikring hvis det viser seg at nettet må

kaste pakker.

Det dyreste og enkleste måten er å ha masse hardware på tjenersiden og at tjeneren har en linje med høy hastighet, og dermed kan man sende mange unicast strømmer med pakker som er perfekt for den brukeren som skal se. Dette ville ikke blitt noen bra løsning siden dette kan føre til mye og dyr hardware. Vi har forskjellige metoder for at det skal bli billigere. Hvis man skal lage smarte tjenere kan man koble forskjellige teknikker inn i tjeneren, for det man vil er at flest mulig skal kunne bruke tjeneren og motta video fra den med god kvalitet. Det er mange bra teknikker som er tilgjengelig, som kan gjøre at tjeneren greier mye mer trafikk og sender bra kvalitet til klientene. Som for eksempel lagdelt video[26], prioritet kasting av pakker [28] og shapeing av videostrømmer.

Fra denne kombinasjonen venter vi at proxyen vi skal lage vil kunne utnytte de forskjellige teknikkene på en slik måte at vi alltid vil ha den best mulig kvalitet på videostrømmen som klienten kan få. Siden vi alltid sender igjennom så mange pakker som det er mulig og hvis vi mister noen pakker vil de pakkene bli av de lavere prioriterte pakkene. Vi venter også at hvis vi putter disse metodene sammen i en proxy at vi skal kunne få et robust system som kan ha mange brukere på denne måten avlaste en tjener som skal sende ut strømmer med data. Dette vil føre til at når det blir press på linjen vil de viktigste pakkene komme frem og føre til at det blir mer skalerbart. Som igjen kommer til å få den beste mulige kvalitet for klienten.

Det kan også være lurt å bruke multicast slik at man samler opp den del brukere og sender en strøm som blir kopiert ut til alle som skulle se på den sammen mediastrømmen. Sjansen for at alle vil se den på akkurat samme tid er ikke så veldig stor så da må man ha noen strategier slik at man bruker minst mulig ressurser til flest mulig brukere. Man kan for eksempel bruke HMSM (Hierarchical multicast stream merge) [13] hvor man kombinerer noen gode metoder som piggy backing[15], dynamic skyscraper[14] og patching[20] Ideen er at man starter strømmer ved jevne mellom rom og når man det kommer en ny bruker så man han cashe fra multicast strømmen. Slik at det tilslutt vil bli en multicast strøm. Man kan da benytte seg av proxy tjenere som kan ha prefix cashing slik at man har første delen av filmen på en proxytjener og alle som begynner å se filmen vil først motta den delen fra proxyseveren først og når de har sett den delen vil de motta resten fra en hovedtjener. Vi kan benytte denne metoden for å samle opp brukere slik at vi får flest mulig i samme multicastgruppe,

for da sparer tjeneren seg for mange strømmer og kan ha flere brukere. Vi kan også bruke metodene hver for seg men det vil da ikke gi like stor optimalisering.

Vi kunne benyttet oss på metoder som skal miske prossesor tid og kontekst bytter ved å senke minne kopieringer intert[16] . Vi har også tenkt å benytte oss av nettverksprossesorerenhet (NPE)som skal avlaste proxytjeneren. Nettverksprossesorer kan vi programere og legge inn funksjoner for å avlaste prossesoren på proxytjeneren[29]. Dette kan være en måte å få flere strømmer igjennom proxyen.

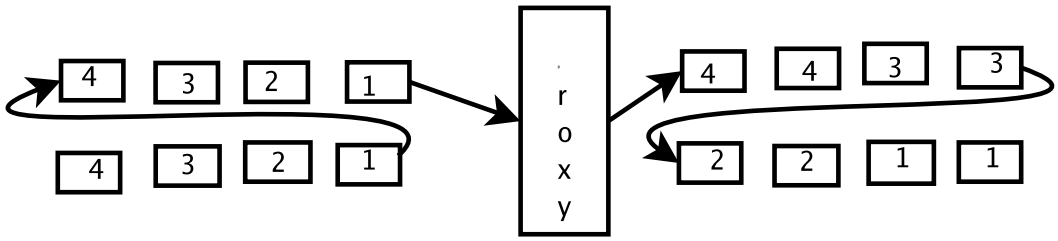
1.2 Problemstilling

Når man skal ha en tjener som får mye å gjøre kan det ofte være lurt å finne måter man kan avlaste tjeneren på eller dele arbeidsoppgaver til flere maskiner. Det kan da lønne seg å lage en proxy som kan stå mellom tjeneren og klienten for å hjelpe å avlaste tjeneren. Vi har som regel god linje mellom proxyen og tjeneren. Det kan være mer varierendes linjekvalitet mellom proxyen og klienten. Det vil derfor lurt å ha kvalitetssikrings metoder som prioritering av pakkene liggende på proxyen. Vi kan også ha andre kvalitetssikkrings metoder som sørger for at ikke en strøm tar all båndbredden som er tilgjenglig for proxyen. Dette har vi fordi vi ikke vil at andre strømmer skal blir berørt i en så stor grad når en strøm plutselig får en burt med pakker.

Vi skal senere se på en proxytjener hvor vi får inn en lagdelt video hvor hvert lag har sin egen prioritet . Vi gjerne ha de høyeste prioriterte pakken innen for et bestemt tidspunkt slik at vi får den beste kvaliteten på videoen til klienten. Vi ser på figur 1.2 hvordan pakkene var før og etter proxyen. Vi skal også lage en shaper på strømmen slik at en strøm ikke skal ødelegge for de andre. Dette har vi tenkt å bygge på et NPE som skal få ned jitter og latens på selve strømmen slik at vi får bedre kvalitet ut til klienten.

Dette kan være interessant får nett som går ut til sluttbrukeren som er av lavere kvalitet som for eksempel trådløsenett eller mobilenett. Da vil

Figur 1.2 Pakke sortering



1.3 Oppgavestruktur

Vi skal se hvordan vi skal strukturere denne oppgaven videre.

I kapittel 2 skal vi se litt på lagdeltvideo, og hvilke metoder vi har får for å dele dem i lag. Vi skal se på forskjellige prioriterings metoder som gjør at vi vil alltid vil få igjennom de viktigste pakkene.

I kapittel 3 skal vi se på arkitekturen til IXP2400 kortet. Vi skal se hvilke komponenter som finnes og hva vi kan bruke fra Software developer kit(SDK) som følger med kortet. Vi skal også se litt på måten vi kan sende igjennom pakker på en effektiv måte.

I kapittel 4 skal vi se hvordan vi har designet systemet vårt. Vi har her prøvd å utnytte alle styrkene vi har ved kortet slik at vi et fleksibelt design. Vi skal se mer på hva slags komponenter vi skal gjenbruke fra SDKen til Radisys.

I kapittel 5 skal vi se videre på hvordan vi har implementert de løsningene vi så på i kapittel 4.

I kapitel 6 skal vi teste systemet vårt mot det som finnes fra før i Linux kjernen vi skal se på testresultater og sammenligne disse.

I kapittel 7 som er konklusjonen skal vi konkludere oppgaven samt se på ting som hadde vært morsomt å sett videre på.

Vi har lagt med koden som appendix

Kapittel 2

Bakgrunnsinformasjon

Vi skal i dette kapittelet se på forskjellige prioriteringsmekanismer og måter vi streamer video på. Først skal vi avklare noen viktige konsepter.

2.1 Video on demand

VOD brukes om når vi skal hente en film som ligger på en tjener, og vi skal streame den ned til den maskinen vi sitter på. Med denne metoden vil vi måtte ha en viss båndbredde tilgjengelig, fordi filmene er vanligvis er beregnet for visse båndbredder. Har vi ikke nok båndbredde er vi nødt til å benytte oss av forskjellige skaleringsmetoder.

Det finnes mange forskjellige måter å sende dataene som skal ut til klienten. Det som er viktig er at man finner en som passer til den bruken man skal ha. Når vi streamer video så vil alle pakkene ha en videoramme som de tilhører. Hvis pakken kommer tidligere en avspillingstidspunktet for pakken, må klienten lagre pakken i et buffer. Klienten kan hente pakken fra bufferet Etter hvert som den spiller av videoen. Hvis den kommer for seint vil hele pakken være verdiløs. Hvis man driver en videotjener vil man gjerne at flest mulig skal kunne bruke den uansett hva slags spesifikasjoner de har på sin pc eller Internet linje. Da vil vi gjerne sende i en hastighet slik at vi har et lite buffer på klientsiden. Når vi har et buffer kan

vi håndtere pakker som er litt forsinket. Vi trenger da mekanismer for å finne ut hvilken hastighet som er best for klienten.

Vi trenger også en felles protokoll for at tjeneren og klienten skal kunne greie å utveksle informasjon. Denne informasjonen kan være hvilken port de skal bruke, eller hvilken hastighet den skal sende på. Til dette bruker man gjerne Real Time Streaming Protocol (RTSP)[18], som setter opp en forbindelse mellom klient og tjener slik at de kan utveksle denne informasjonen.

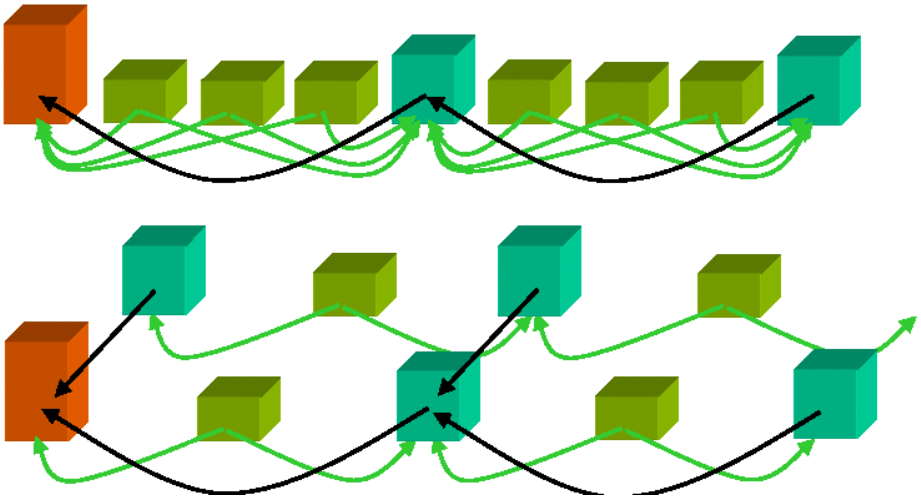
Hvis vi finner ut at klienten ikke har rask nok linje til å avspille videoen på den bitraten som videoen er ment å ha, kan vi ved hjelp av forskjellige metoder prøve å få best mulig kvalitet. En av disse metodene kan være å dele opp filmen i lag, hvor hvert lag gir en forbedring til filmen. Vi kan da droppe noen av de høyestelagene slik at vi fortsatt kan se filmen, men ved en dårligere kvalitet.

2.2 Lagdelt Video

Når vi skal sende video over Internet vil vi ha større behov for skalerbarhet. Dette kan vi gjøre ved at vi deler filemen i flere lag. I lagdelt video deler man inn filmen inn i flere lag. Man har vanligvis et baselag som vi må ha for å se filmen. Dette er det absolutt minste kravet for å kunne se videoen. Poenget med lagdeling er at man lettere kan skalere filmen til forskjellige båndbredder. Over baselaget vil man gjerne bygge på med kvalitetslag. Dette kan gjøres på forskjellige måter avhengig av hvilken lagdeling vi tenker å benytte oss av. Hvis vi har alle lagene vil vi få veldig bra kvalitet, men vi skal likevel kunne se videoen hvis vi ikke alltid får alle lagene, men da med redusert kvalitet. Lagdelt video passer veldig bra sammen med prioritering av video fordi det er lettere å identifisere de viktige pakkene i en lagdelt video. De lavere lagene er alltid mer prioritert enn de høyere lagene, fordi vi ikke kan bruke et av de høyere lag uten å ha alle de lagene under. Det er fordi de høyere lagene bygger på de lavere lagene. Vi har en del forskjellige måter å lagdele video på. De benytter seg alle av prinsippet om forskjellige lag, men teknikken om hvordan vi deler dem er forskjellig:

- Temporal lagdeling benytter seg av fordeling av de individuelle bildene over flere lag, som vil si at vi legger de bildene som vi absolutt trenger i baselaget og legger de andre bildene oppover. Jo flere lag vi da mottar, jo høyere blir bilderaten av videoen. Fra figur 2.1 ser vi hvordan rammene blir organisert. De største Boksene er I-rammer, de litt mindre er P-rammer og de minste er B-rammer. B-rammene har avhengigheter som vi ser som piler til nærmeste P eller I på begge sider, mens P-rammene har avhengig av den forrige P eller I-rammene. Dette kan variere avhengig av hvilken kodec vi bruker. Hvis vi bruker en "moving picture expert group " (mpeg) kodec benytter den seg av I, P og B-rammer. Disse rammene vil vi gjerne organisere på en slik måte at I-rammer er i bunn, kanskje noen P-rammer. Vi vil deretter legge b-rammene oppover. Lagene blir laget med tanke på at bitraten skal bli høyere for hvert lag vi har, slik at det 1. laget har en bitrate på for eksempel 10 Kbps mens de 2 første lagene har 15 Kbps. Temporal lagdeling er den enkleste metoden å implementere, men det blir ikke alltid den beste kvaliteten [22][25].

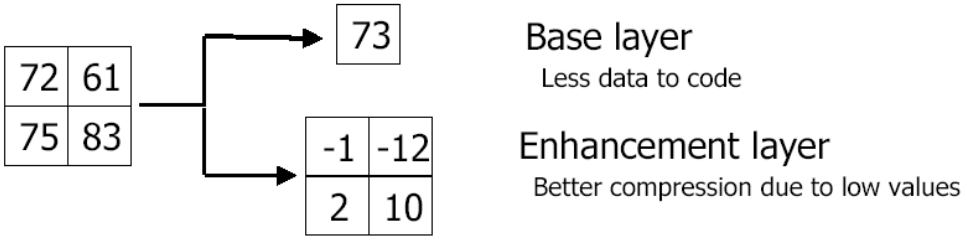
Figur 2.1 temporal pakke lagdeling



- Spatial lagdeling deler opp videoen slik at man har et baselag og flere forbedringslag. Dette gjøres vanligvis ved at vi nedskalerer base laget, dvs at du slår sammen 4 pixler og koder dem som en. Vi ser fra figur 2.2 metoden vi koder spatial lagdeling. Vi har 4 pixler med verdiene 72, 61, 75 og 83. Vi velger oss en verdi 73 som blir baselaget. Forbedrings lagene vil være de verdien som gjør at vi får de origi-

nale verdiene dvs -1, -12, 2 og 10. Baselaget sendes på vanlig måte, hvis klienten mottar forbedringslag vil man først da dekode begge lagene og slå dem sammen hvis man ikke rekker å få alle forbedringslagene vil man bare dekode de lagene vi har fått, men dette fører til å kvaliteten ikke blir så bra[22].

Figur 2.2 Spatial lagdeling



- I "Signal-to-noise-ratio" (SNR) koder man baselaget i"discrete cosinus transform" (DCT), og når man har et eller flere forbedringslag vil man kode dette ved en invers DCT og trekke fra originalen. Til slutt DCT-koder man resultatet. Når man mottar lagene vil man ta de lagene man har og dekode dem ved å kjøre invers DCT. Ved denne metoden komprimerer vi dataene ganske godt, noe som er viktig når man skal ha en skalerbar bitstrøm [26].

2.3 Prioriteter

Det er noen ganger som noe informasjon er viktigere en annen informasjon. Det kan for eksempel være en lagdelt video hvor baselaget er viktigere enn forbedrings lagene, fordi resten av dataene bygger på disse. Da kan det ofte være lurt å ha en skeduling som tar hensyn til akkurat dette. Når vi har en lagdelt video må vi kunne skille på dem og da kan det være lurt å bruke prioritetskøer som alltid prøver å få igjennom de viktigste pakkene, når det blir press på køen. Dette krever at vi på forhånd vet hvilke pakker som er viktige og at de er identifisert enten i headere eller pakkeinnhold. Vi kan ordne prioritetskøene på litt forskjellige måter. Noen av de vanligste metodene er:

- Assured Forwarding (AF): AF er en del av Differentiated services (DIFFSERV) som er en arkitektur som skal gi støtte for ulike tilbud til nettverks-strømmer. En av karakteristikkene til DIFFSERV er at den samler strømmer i grupper og behandler alle strømmene i gruppen som en strøm. AF støtter leveranse av Internet protocol (IP) pakker i fire forskjellige uavhengige AF klasser for videresending. Innenfor hver AF klasse kan en IP pakke bli tilordnet en av tre forskjellig grader av kastepresedens. Det AF skal gjøre er å sikre forskjellige grader av sikkerhet for at vi får videresendt en pakke. AF har opptil 4 forskjellige klasser vi kan klassifisere pakkene i, hvor hver klasse har forskjellige ressurser som båndbredde og bufferplass. Dette vil gjøre at vi har større sjanse for at vi får igjennom en pakke hvis vi setter pakken i en klasse som har mer båndbredde eller bufferplass, enn i en som har mindre. Det er også forskjellige regler for når vi skal kaste en pakke, hvis vi har congestion så vil pakkens AF klasse som pakke være avgjørende for om vi skal kaste pakken eller ikke[27].

- Expidited Forwarding (EF): EF er også en del av DIFFSERV og kan bli brukt for å bygge et ende til ende system som har lite tap, lite latens og lite jitter. Pakketap, latens og jitter henger alle sammen med køer som kommer som følge av sending av pakker i et nettverk. Køer kommer som regel fra at innraten til en node er større enn utraten. Det vi prøver med EF, er å minimere køene. Dette skjer ved at vi prøver å finne en veldefinert minimum utsendingsrate, hvor vi med veldefinert mener at den er uavhengig av den dynamiske tilstanden til noden. Det andre EF krever er at senderaten er mindre enn minimum utraten. Dette skjer ved hjelp av forskjellige metoder som for eksempel shaping eller policing. Når vi minimerer køene kan vi lett si hvor lang tid det vil ta å sende pakker igjennom nettverket. Ulempen er at det ikke er så lett å skalere med EF[21].

- Weigthed Fair Quing (WFQ): Rettferdig køer en mekanisme som skal gi rettferdig køer. Denne teknikken prøver vi å gi alle strømmer som skal igjennom noden like mye tid. Det vil si at hvis har to mediastrømmer som skal igjennom en proxy og videre, ville proxyen gitt strømmene like mye båndbredde. Det krever jo at begge strømmene har noe å sende. Med WFQ vil vi kunne gi strømmen forsjellig vekt. Dette vil føre til at vi kan definere at den ene strømmen skal kjøres 90 % av tiden dermed vil den siste strømmen ha 10% igjen å kjøre på.

- Class Based Quing (CBQ): Er en metode som deler nettverklinkens ressurser inn i flere klasser slik at hver klasse har en forhånds-innstillt

båndbredde. Vi kan da spesifisere hvem som hører til hvilken klasse etter IPadresse, port eller protokoll. Hver klasse kan også ha underklasser slik at vi kan få et helt hierarki av klasser med forskjellige krav. Hver klasse har da en kø, og hvis denne køen blir full vil vi kaste en eventuelt ny pakke som vil bli lagt inn i den køen. Med denne metoden kan vi sette opp et ganske komplekst system hvor vi kan prioritere det vi vil. Med denne metoden vil noen av køene få høyere båndbredde eller prioritet, men i hver kø er det første man inn er første mann ut (FIFO) som gjelder.

2.3.1 Prioritetskøing Med Kun En Kø

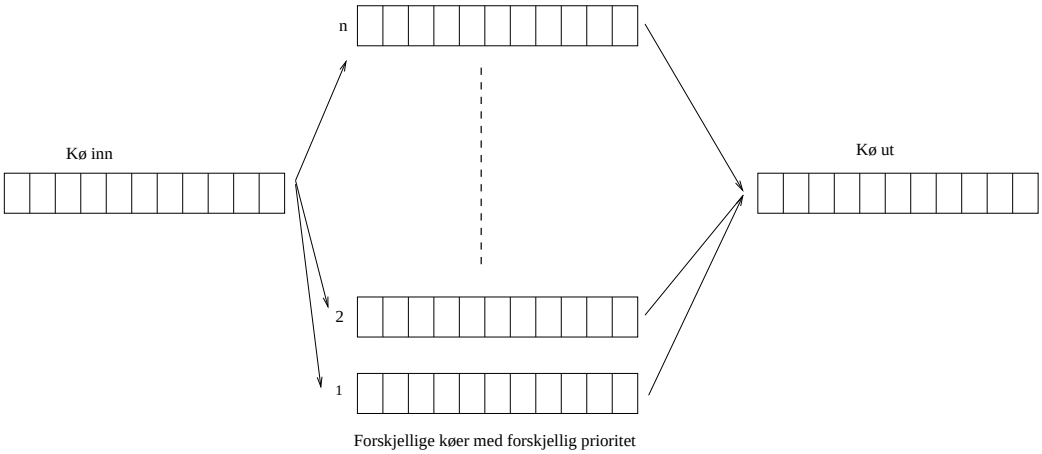
Når vi kun har en kø som vi skal sortere pakkene på må vi gå igjennom hele køen slik at man alltid legger pakker med høyest prioritet først og de lavere prioriterte pakkene Etter hvert. Dette betyr at vi må gå igjennom køen for å finne ut hvor vi skal plassere pakken. Dette kan ta litt tid hvis køen blir lang. Vi vil også få et annet problem, hvis det ligger et jevnt antall pakker på køen og vi får inn like mye som vi sender ut, vil pakkene med lavest prioritet på køen aldri bli sendt, siden det hele tiden kommer inn nye pakker. En måte å løse dette på er at vi sorterer pakkene innenfor et visst tidsrom. Det vil si at alle de pakkene som tilhører et gitt tidsrom som ikke blir sendt innen dette tidsrom vil bli kastet. Da vil vi få sortert de pakkene som vi har innen for visse tidsrom.

2.3.2 Prioritetskøing Med Flere Køer

Når vi sorterer pakkene på flere køer slipper vi å gå igjennom hele køen for å finne ut hvor vi skal legge pakken, men vi må finne ut hvilken kø vi skal legge den i. Dette er avhengig av hvor mange køer man har til rådighet, men hvis vi har like mange køer som det er prioriteter blir det bare å sjekke hvilken prioritet pakken har, og deretter legge den på den riktige køen. Vi ser fra figur 2.3 hvordan vi kan organisere køene hvor vi har en inn-kø og en ut-kø, mellom disse køene kan vi ha en del prioritetskøer. Når vi skal ta ut pakker fra køen kan vi gjøre det på litt forskjellige måter avhengig av hva vi skal bruke den til. En måte å gjøre det på er å alltid ta fra den høyeste prioritets køen så lenge det er pakker i køen og deretter ta av de

etterfølgende prioritets køene når den første blir tom. Vi kan også ta et visst antall fra den første køen, deretter et mindre antall fra den neste køen til vi har gått igjennom alle køene. Vi kaster pakker når køen vi skal legge pakkene på er full, men istedenfor å kaste den pakken vi holder på med, kaster vi den heller første siden det er større sjanse for at den er forsinket. Det er mer effektivt å gjøre det på denne måten siden vi da slipper å gå igjennom hele køen for å plassere pakkene. Det vil mest sannsynlig bli de lavere prioritets køene som blir fulle, siden vi alltid vil ta mest pakker fra de høyeste prioriterte køene, og det er når køene blir fulle at vi kaster pakker.

Figur 2.3 Prioritet med fler Køer



2.3.3 Evaluering Av Muligheter

Vi har her sett på en del av mulighetene vi har ved valg av prioriteringskø. Vi skal bruke lagdelt video og sende den igjennom en proxy ut til en klient. AF vil ikke hjelpe oss så veldig siden den ser mest på IP pakker, og det som er interessant for oss ligger inne i pakkeinnholdet. Vi kan derfor ikke bruke denne metoden.

EF er en annen metode som virker som et bra alternativ for VOD, men problemet er at vi aldri vet helt sikkert hva båndbredden til klientene er. Det vil derfor være vanskelig å sette opp dynamiske forbindelser som vil fungere i praksis. Hvertfall hvis vi over hele Internet skal finne ruter som ikke skal kunne ha køer i nodene.

Hvis vi har en lagdelt film med ville det ikke fungert å bruke WFQ som prioritets algoritme siden hver strøm vil bli prioritert ikke pakkene i en strøm. Vi kunne ha gjort det slik at en strøm hadde vært viktigere enn en annen. Det vil si at en strøm med høy bitrate hadde fått mer tid en en med lavere. Men det ville fortsatt ikke funger siden det vi vil er å ha de pakkene i en strøm som er viktige igjennom.

Hvis vi ser på svakhetene til CBQ er det at klassene med sine begrensninger er satt opp på forhånd. Det kan da være veldig vanskelig å forutsi hva som er viktig hele tiden. Det er derfor denne ikke vil fungere for videostreaming.

Vi trenger noe som fungerer for videostreaming. Det vil si noe som er dynamisk nok til at vi kan endre det. Det som da fungerer for oss er en vanlig prioritetskø med flere køer. Siden vi skal bruke den til video med flere strømmer er det greit at vi ikke hele tiden henter fra første prioritets køen så lenge det er pakker der men finner en metode som henter litt ut fra alle køene.

2.4 Shaper

Internet bruken vi har i dag har ført at vi har andre former for problemer i dag enn for noen år siden. Det har blitt en god del mer strømmer som skal sendes. Dette kan være alt fra video til online spilling. Dette fører til at vi har større variasjoner av bitrater igjennom nettet. Det kan for eksempel være i et online spill hvor plutselig skjer noe da vil det bli sendt masse pakker. Det kan også være fordi vi har varierendes båndbredde for video. Vi vil da ha mekanismer som hindrer at andre strømmer som ikke genererte burst blir påvirket av burstene slik at de også må kaste pakker. Den strømmen som fikk en for stor burst vil miste noen pakker da. Dette kan igjen føre til større sjanser for at vi får burst eller at vi må kaste pakker.

Hvis vi bruker en shaper som vil være en slags "token bucket" som skal sørge for at en strøm ikke vil overstige en viss bitrate. Da ville den problematikken være løst. Den vil gi en strøm ett visst antall billetter slik at de ikke skal kunne ødelegge andre strømmer hvis den plutselig får burst. Shaperen må bli satt opp på en båndbredde slik at vi vet hvor mange billetter den skal få. Hvis det det ikke er ledig billett legges vi pakken i en

vente kø slik at de kan vi sendt etter på ett sendere tidspunkt hvis det lar seg gjøre. På den måten sikker vi at strømmen ikke tar for stor del av den ledige båndbredden slik at dette ødelegger for andre strømmer.

Vi kan ha forskjellige lengder på ventekøen. Hvis vi har en lang kø vi det mer sannsynlig at vi får gjennom pakkene, det kan bare ta litt tid. Noen ganger trenger vi pakken på en bestemt tid ellers er det ikke noe vits å ha dem i det hele tatt. Det kan derfor være ressurs sløsing å ta vare på en pakke for å så bare kaste den på ett senere tidspunkt. Vi trenger derfor ikke alltid å ha lang kø noen ganger kan det være like greit med en kort kø.

Det er også et spørsmål om vi skal ha et tak på hvor mange billetter vi kan ha. Hvis vi ikke har noen tak kan vi spare opp masse billetter. Hvis vi får altfor mange billetter kan vi få burst likevel. Har vi derimot ett tak vil vi ikke komme i den situasjonen.

2.5 Multimedia Protokoller

Når vi streamer video vil vi gjerne ha litt kontroll på videoen slik at vi kan f. eks starte, stoppe, pause eller slutte videoen. Vi vil ikke alltid bare starte en film og motta pakker til filmen er ferdig. Siden vi deler opp filmen i pakker vil vi også ha kontroll over hvilken filmdel som er i hvilken pakke. Det kan også hende at vi streamer separat audio og video, vi trenger derfor en protokoll som håndterer denne problemstillingen. Vi vil da skille mellom mediapakker og kontrollpakker til videostrømmen. Dette bruker vi følgene protokoller til:

- Session Description Protocol (SDP): Denne protokollen er ment for å beskrive multimediasesjoner med det formål å annonsere multimediasesjonen eller å sette opp en multimediasesjon. [17]
- Real Time Streaming Protocol (RTSP): Dette er en applikasjonsprotokoll som kontrollere levering av strømmer fra en tjeneren til en klient. Innebygget i RTSP kommer med et stort rammeverk for å ha kontroll med strømmene. [18]

- Real Time Transport Protocol (RTP): Dette er en protokoll som sørger for ende til ende netverkstransport. RTP passer for applikasjoner som sender sanntidsdata som audio, video og andre løpende data. [19]

Når en klient skal koble seg opp mot en videotjener vil den opprette en RTSP forbindelse med tjeneren. Denne forbindelsen vil sette opp de parameterne som trengs for å streame video. Dette vil være parametere som hvilken hastighet videoen skal sendes på, hvilken port tjeneren skal sende RTP-strømmen på. Parametrene om hvordan filmen skal avspilles ligger på en SDP fil denne sendes over i RTSP forbindelsen. Når alt er satt opp brukes RTP til å streame videoen. Da trenger vi egentlig ikke RTSP-forbindelsen mer. Vi skal bruke denne protokollen i vår proxy. Vi bruker SDP til å identifisere når vi får nye strømmer som skal igjennom tjeneren slik at vi allokerer plass til informasjonen om strømmen i minne, og når vi får en RTSP teardown frigjør vi plassen igjen.

2.5.1 SDP

SDP[17] brukes ofte for å gi informasjon om en multimediasesjon. Det kan være at det er er en videostrøm som skal settes opp, en konferanse eller en IP-telefoni samtale. Vi bruker SDP til å gi informasjon om multimediastrømmen til de brukerne som skal motta multimediastrømmen. Dette er kun for å prøve å beskrive sesjonen som skal sendes, ikke for å forhandle om sendekriteriene. SDP inneholder:

- Sesjonsnavn og formål
- Tiden sesjonen er aktiv
- Informasjon hvordan man mottar media (adresser, porter og format)
- Informasjon om båndbredde
- Kontaktinformasjon om de ansvarlige for sesjonen

2.5.2 RTSP

RTSP[18] setter opp en eller flere strømmer med enten video, audio eller andre media. Siden RTSP ikke er en transportprotokoll i seg selv, er den nødt til å bruke enten en TCP, UDP eller en annen transportprotokoll. Videotjeneren ser på RTSP sesjonen som er satt opp. Hvis vi har flere RTSP sesjoner på videotjeneren vil man skille mellom disse ved å se på hvilken sesjon forespørselen gjelder. Transportmetoden vil ikke ha noe å si for forbindelsen.

RTSP kontrollerer en strøm som kan bli sendt med en separat protokoll. Den trenger ikke nødvendigvis være sendt med den samme protokollen som RTSP dataene. Vi kan for eksempel sende RTSP pakkene med TCP og dataene med UDP. Mediastrømmen sender dermed uavhengig av om vi har satt opp forbindelsen med RTSP. Mediatjeneren trenger derfor å ha en sesjons status for å holde orden på om mediastrømmen sender eller ikke sender. Vi har følgende RTSP metoder som endrer statusen til RTSP sesjonen:

- RTSP SETUP: Tjeneren starter en RTSP sesjon når den mottar en SETUP pakke, og den retjener ressurser slik at den har nok til å sende en mediastrøm.
- RTSP PLAY eller RECORD: Tjeneren starter sendning eller mottak av strømmen som ble satt opp via SETUP.
- RTSP PAUSE: Tjeneren pauser sending av mediastrømmen men frigjør ingen ressurser, slik at vi kan starte mediastrømmen på et senere tidspunkt.
- RTSP TEARDOWN: Tjeneren stopper sending av mediastrømmen og frigjør ressursene slik at den kan ta imot andre forespørsler.

Det finnes flere RTSP forespørsler som kan være viktige. Vi har RTSP DESCRIBE, som klienten sender for å få en beskrivelese av en mediastrømmen. Beskrivelsen kan bestå av en SDP fil som ligger på tjeneren med beskrivelse av hvordan mediet er tenkt sendt, hvilken båndbredde det har og så videre. Tjeneren gir da et RTSP svar tilbake til klienten som består av SDP filen. et eksempel på en describe forespørsel er 2.1 hvor tjeneren kanskje vil gi et svar som 2.2

```
DESCRIBE rtsp://192.168.10.3:9070/layerdummy128k.desc RTSP/1.0
Cseq: 1
Accept: application/sdp
```

Tabell 2.1: Eksempel for en DESCRIBE forespørsel

```
RTSP/1.0 200 OK
CSeq: 1
Content-Type: application/sdp
Content-Length: 170
Content-Base: rtsp://192.168.10.3/layerdummy128k.desc/

v=0
o=mzink
s=Layer Dummy 2
c=IN IP4 192.168.10.3
b=AS:700
a=framerate:25.000000
t=0:05:0:2
m=video 9030 RTP/AVP 95
a=fmtp:lcrtp
a=x-priorities:1234
a=x-blocksize:1024
```

Tabell 2.2: Eksempel for en DESCRIBE svar

2.5.3 RTP

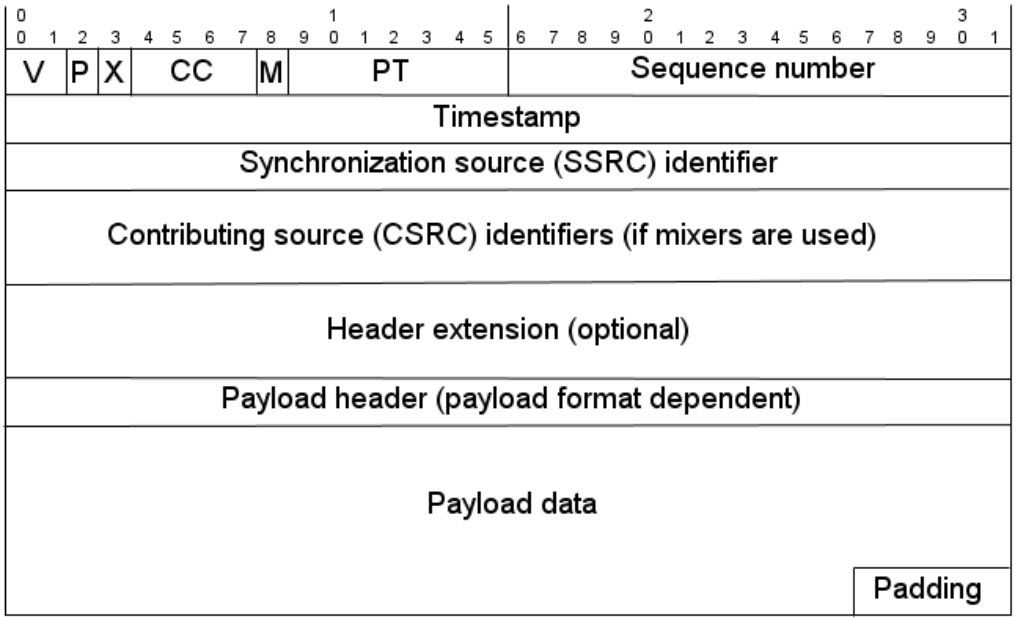
RTP[19] sørger for ende til ende transportfunksjoner som passer for sanntids media. RTP-headeren inneholder et identifikasjonsfelt for å identifisere mediastrømmen som pakken tilhører. Det er også et tidsstempel og sekvensnummerering slik at vi vet om vi har mistet noen pakker og rekkefølgene på pakkene.

RTP består egentlig av to deler. Selve transportdelen RTP og en rapporterings del "RTP control protocol" (RTCP). RTP headeren kan vi se i figur 2.4 som har følgene viktige felter:

- Versjons nummeret (V): De 2 første bitene i RTP-headeren forteller oss hvilken versjon av RTP vi benytter oss av. Den mest vanlige er versjon 2. Versjon 1 er utdatert.
- Padding (P): Dette feltet er på 1-bit. Dette bitet brukes i noen krypterings algoritmer.
- tillegg (x): Dette feltet er på 1-bit. Det forteller oss om det er noen tilleggsheadere etter RTP-headeren
- Contributing source count (cc): Dette feltet er på 4-bit, og forteller oss hvor mange csrc identifikasjoner som følger etter RTP-headeren.
- marker (m): Dette feltet er 1-bit og brukes for å angi spesielle hendelser i strømmen og bruken er avhengig av de forskjellige payload typene.
- payloadtype (pt): Dette feltet er på 7-bit, det sier oss hvilket format pakkeinnholdet er på.
- Sekvensnummer: Sekvensnummeret er på 16-bit, og øker med 1 for hver RTP pakke som sendes. Mottaker bruker dette til å finne pakketap på mediastrømmen, og for levere pakkene i riktig rekkefølge. Nummeret trenger ikke begynne på 0, det kan bli satt med et tilfeldig tall.
- Tidsstempel: Tidsstempelet er 16-bit, og sier når RTP pakka skal spilles av klienten. Tidsstempelet brukes for at klienten kan ha et buffer med pakker og synkronisere pakkene.

- Synchronization source (SSRC): Dette feltet er 32 bit, og brukes av til å unikt identifisere mediastrømmen slik at klienten kan ha flere mediastrømmer på engang.
- Contributiong Source (CSRC): Dette er hvis det er flere kilder til mediastrømmen og en mixer vil da bruke dette felte til å slå sammen disse.

Figur 2.4 Rtp header



RTCP

RTCP sender tilbake rapporter med informasjon om progresjonen til mediastrømmen med noen sekunders mellomrom. Dette er for å gi tilbakemelding om pakketap. Siden det ikke er noen vits i å sende pakker om igjen vil dette være en bra måte å gjøre det på. Hvis mediatjeneren finner ut at det er mye pakketap kan den da redusere bitraten på sendingen. De forskjellige rapportene som RTCP sender er:

- Sender-rapport er en rapport som mediatjeneren sender ut til sine

mottagere og kan inneholde antall sendte pakker, tidsstempel og synkronisering.

- Mottagerrapport sender statistikk for mottatte pakker tilbake til mediatjeneren. Pakkene inneholder pakketapsinformasjon og informasjon om tidsvariasjoner.
- SDES Inneholder kilde beskrivelse og SSRC informasjon
- BYE pakke når deltageren slutter av en sesjon eller når SSRC forandres
- APP brukes av applikasjons utvidelser

2.6 Sammendrag

I dette kapittelet har vi sett på en del muligheter som vi har med lagdelt video og forskjellige prioriterings mekanismer. Vi har funnet ut at det ville bli best med en prioritetskø som sorterer på forskjellig prioritet. Dette var fordi at vi da kan få de viktigste pakkene i strømmen igjennom. Dette både vil gjøre at vi alltid vil få de viktigste pakkene igjennom, dermed også den beste kvaliteten på videoen. Det vil også bli mest skalerbart siden vi kan sende samme filmen til to forskjellige brukere, selv om den ene har dårligere linje. Da vil den brukeren med færre ressurser få lavere kvalitet. Vi har også sett på formater for kontroll av mediastrømmer mellom klient og tjener ved hjelp av RTSP og RTP.

Vi har også sett hvordan vi kan sikre at strømmene ikke ødelegger hverandre, fordi en av dem plutselig får mange pakker på en gang. Dette gjøres ved at vi har en shaper som deler ut billetter til strømmene, slik at når en pakke skal gjennom må den ha billett. Hvis pakken ikke har billett legges den i en ventekø og må vente til det er ledig billetter. Vi sikrer ikke at alle pakkene kommer igjennom ved bruk av shaper men vi beskytter de andre strømmene og hindrer at det ikke kommer for mange pakker slik at ikke en klient bruker alle ressursene.

Vi har sett på forbedringer som kan gjøres i software. Men vi kan muligens effektivisere enda mer, ved å benytte oss av et nettverksprosseseringsenhet (NPE). NPE er skal kunne avlaste vertsmaskinen og gi bedre ytelser

enn en vanlig maskin. Vi skal se mer på NPE i neste kapittel og hvordan vi skal implentere de metodene vi har sett på i dette kapittelet på NPE i det neste etter det igjen.

Kapittel 3

Arkitektur

Vi har i forrige kapittel sett på de mulighetene vi har med å levere en mediastrøm på best mulig måte ved hjelp av programvare. Et skritt videre vil være å se på muligheten for om vi kan få bedre kvalitet uten alt for mye mer hardware. Det er jo nødt til å bli en avveiing mellom kostnader og ytelse. Vi har sett på muligheter i programvare for å få en best mulig mediastrøm fra en tjener til en klient. En annen ting vi kan prøve er å benytte oss av en nettverksprosseseringsenhet (NPE). En NPE skal gi oss mer fleksibilitet enn en vanlig maskin, og ytelse som er opp mot de ferdig støpte produktene. Samtidig som vi benytter oss av en NPE kan vi likevel benytte oss av vertsmaskinen, vi kan til og med fylle verten med så mange kort som det er plass til. Fordelen skal være at vi får bedre ytelse, det vil si at vi minsker jitter og latens. Vi skal også videre se på Intel sin plattform for programvare for NPE.

3.1 Nettverksprosessorer

Selve definisjonen på en NPE er i følge Comer [11]:

En nettverkprosessor er en hardwarekomponent som kombinerer lav kost og fleksibilitet som er programmerbart. Nettverksprossesorer er byggeblokker brukt til å konstruere nettverkssystemer.

Mye av dagens begrunnelser for å velge NPE kommer fra økonomiske grunner. For selskaper som skal kjøpe eller bygge nettverkssystemer, kan NPE drastisk senke kostnader og tid som trengs brukes til produsere nytt utstyr. Ved å benytte NPE istedenfor ferdiglaget ASIC vil det bli billigere og mer fleksibelt. NPE vil også senke kostnaden på langsikt siden vi kan modifisere eller oppgradere kortet uten anskaffelse av ny hardware.[11] [24]

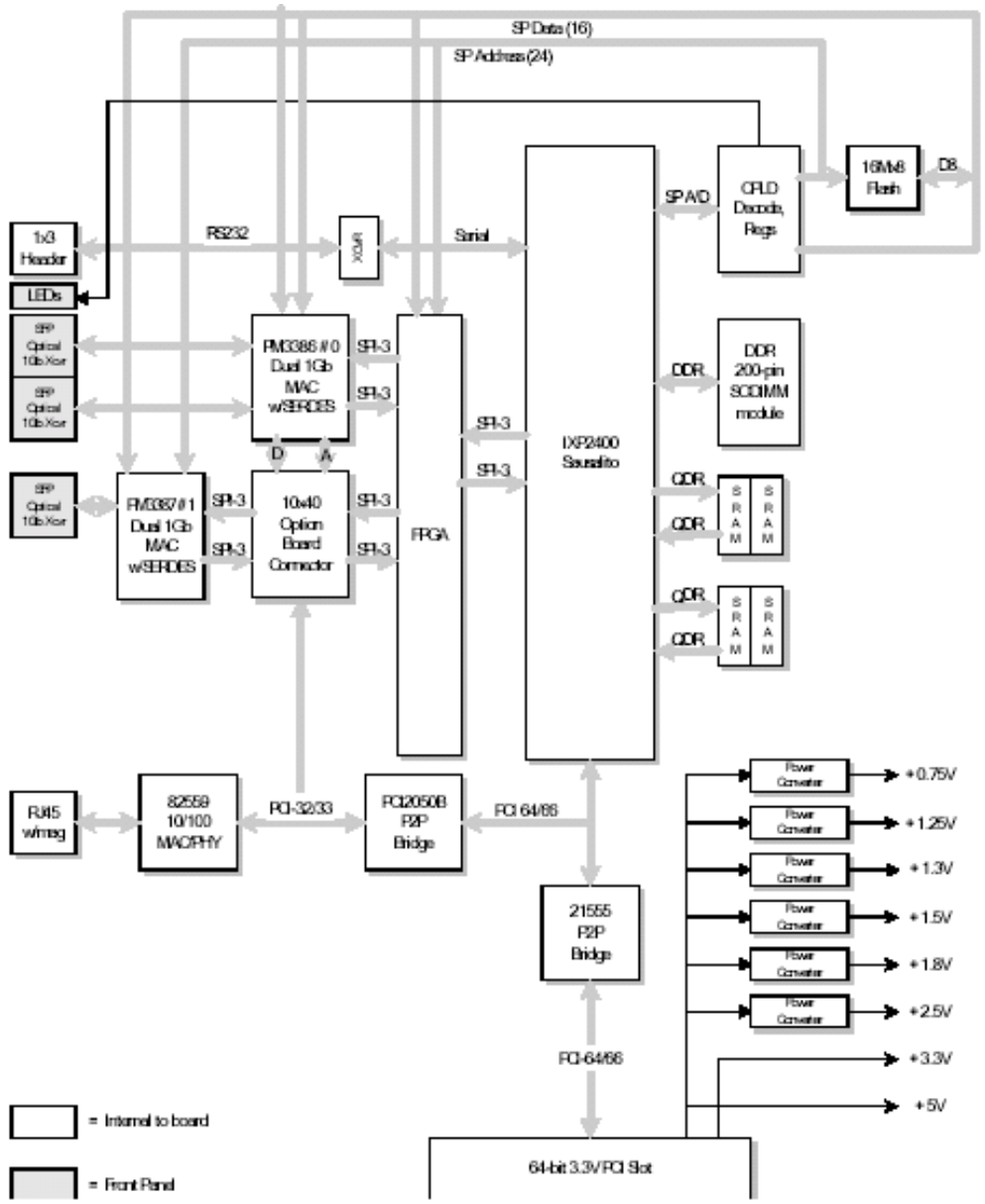
Vi skal se på en spesiell NPE med brikkesett fra Intel kalt IXP2400 [3]. Med IXP2400 NPE skal man kunne legge inn software som har fleksibilitet og er rask nok til å kunne konkurrere med ASIC. IXP2400 NPE kan håndtere og manipulere raske strømmer på en effektiv måte. Vi skal senere i kapitlet se på hvordan IXP2400 er bygd opp og hvordan vi forventer at vi skal få bedre ytelse. Vi skal også se litt på hvilke verktøy vi har tenkt å benytte oss av.

3.1.1 Radisys ENP-2611

Figur 3.1 viser koblingsskjema for komponentene som er med på ENP-2611[5] NPE.

- IXP2400 brikkesettet er kjernen i hele kortet som inneholder både mikromaskin og XScale kjernen. Brikkesettet har også en del kontrollenhet for å kommunisere med de andre delene av ENP-2611 komponentene.
- 3 Gigabit optiske grensesnitt: Vi har 3 optiske gigabit grensesnitt som er tilkoblet PM3386 og PM3387. PM338x er kontrollere som styrer de optiske grensesnittene. PM3386 er koblet til 2 optiske grensesnitt mens PM3387 er tilkoblet 1 optisk grensesnitt.
- SPI-3 briged FPGA: Denne enheten brukes til å snakke med de fysiske grensesnittene på kortet. Denne linken går fra PM338x og IXP2400 brikkesettet.
- 256 MB double data rate (DDR) DRAM: DRAM kan aksiseres fra både XScale og mikromaskin. Her vil selve pakkene være er typisk eksempel på hva som lages i DRAM.

Figur 3.1 Skjema for ENP-2611[6]



- 8 MB SRAM: SRAM kan aksessers fra både XScale og mikromaskin. Her vil vi lagre metadataene og delte variablene
- 16 MB Flashminne: Flashminne brukes til lagre driver og bootloader for kortet. Vi har også lagt Linux kjernen inn på flashminnet, og her

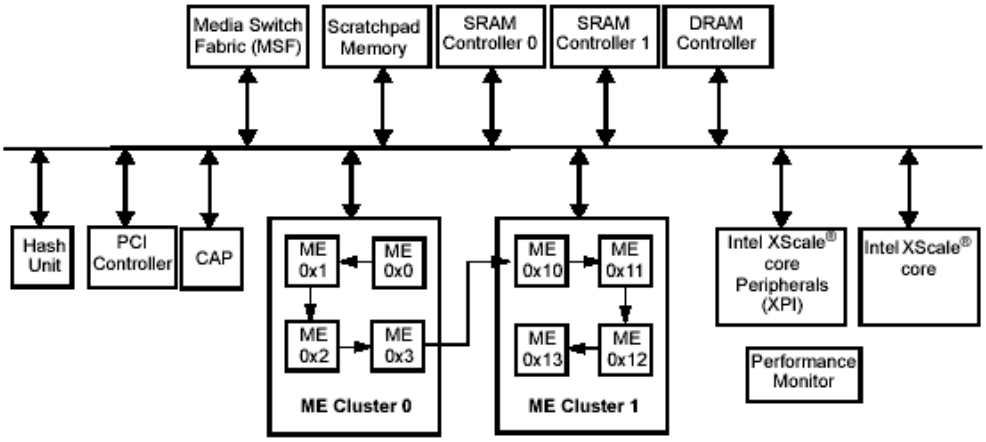
legges det vi trenger til å starte kortet.

- Klokkegenerering: Vi har en klokke som styrer systemklokken til IXP2400. Den skal også styre klokkene til grensesnittene IXP2400 MSF/FPGA og FPGA/PM338x.
- 2 PCI-PCI bro: Den første PCI2050B broen er en synlig PCI bro som skal koble sammen den indre 64-bits bussen til den ytre 32-bits PCI bussen. Dette er veien ut igjennom debug 10/100 grensesnitt. Den andre broen Intel 21555 skal koble sammen den indre 64-bits bussen til 64 eller 32-bits bussen til verten. Dette skal blant annet kunne gi direct memory accsess (DMA) støtte og dele prosessene med verten.
- Ethernetkontroller: Vi har en ethernetkontroller Intel 82559 som brukes for å styre 10/100 debugporten

3.1.2 IXP2400 Brikkesett

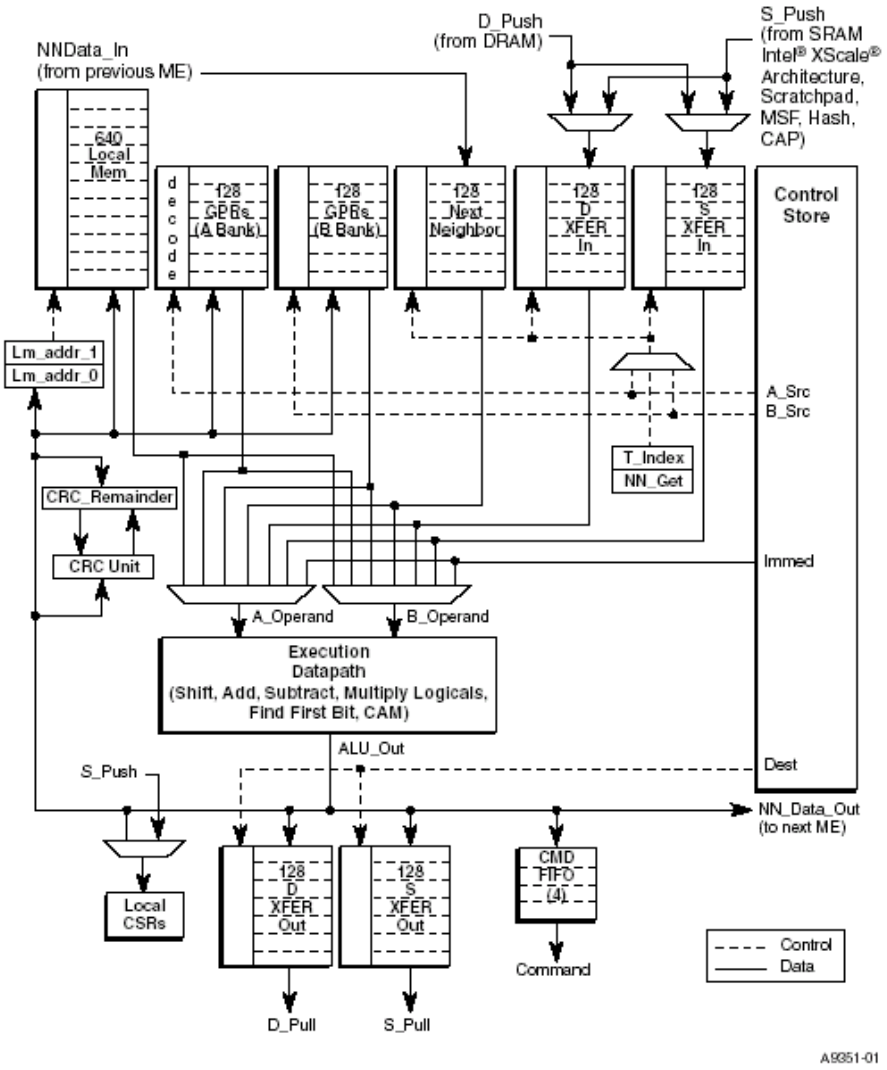
Brikkesettet er laget av Intel og er bygd på et Radisys enp2611[5] kort. På brikkesettet er det en del komponenter som vi ser på figur 3.2 de er bedre forklart under:

Figur 3.2 oppsett av IXP2400 brikkesett[2]



- En 32-bits XScale RISC prosessor: Denne XScale prosessoren er kompatibel med ARM versjon 5. XScale skal vanligvis initiere, og utføre,

Figur 3.3 Skjema mikromaskin



høyere nettverksoppgaver. Den har også 3 typer minne den kan aksessere. Det å sende noe opp til XScale er tregt, og det er ment for unntak eller når noe uvanlig skjer.

- Åtte 32-bits programerbare mikromaskiner: mikromaskinnene kan kjøre opptil 8 tråder parallelt. Fra figur 3.3 ser vi skjema for mikromaskinene, hvor vi har flere forskjellige registertyper som SRAM overflyttnings registre, DRAM overflyttnings registre, generell bruks(GB)

og nabo register. Disse kan brukes når vi programmerer på mikromaskinene. Det må benyttes forskjellige register når vi leser og skriver til minne. Naboregister er ment for å brukes til å sende data til den neste mikromaskinen, naboforholdet ser vi fra figur 3.2 hvor det går en pil fra den ene mikromaskinen til den neste. Vi har også noen GB registre. Disse kan vi bruke til hva vi finner for godt. Vi ser tilslutt at vi har lokalt minne i hver mikromaskin, dette er på 640 KB. Utenom lokaltminne kan vi også aksessere SRAM og DRAM. For å gjøre det trenger vi sxfer og dxfer som er registre som minne leser fra og skriver til.

- En kanals DDR SDRAM kontroller: SDRAM minnekontolleren som sitter i IXP2400 kortet støtter en enkel 64-bits kanal med dobbel datarate. Vi har et adresserom på 2GB. Kortet har også støtte for error-correcting code (ECC). SDRAM kan akseserers fra mikromaskinene, XScale og PCI bussen. Det er her vi legger pakkedataene mens pakken går igjennom mikromaskin-pipelinen.

- To uavhengige SRAM kontrollere: Det er 2 uavhengige SRAM kontroller som begge støtter pipelined QDR synkront SRAM. De kan også støtte en coprosessor som henger sammen med QDR signalisering. SRAM kan aksessers via mikromaskinene, XScale eller PCI.Det er her vi legger metadataene til pakkene og delte variabler som tellere og synkroniserings variabler.

- 64-bits 2.2 PCI kontroller. Denne kontrolleren styrer dataflyten som skal igjennom PCI bussen ut til verten.

- MSF (Media and Switch Fabric interface) Denne brikken er brukt som en link mellom det fysiske laget og IXP2400 chipen eller/og til en switch fabric. MSF inneholder en separat mottager- og sendegrensesnitt. De kan begge settes separat opp til å håndtere enten Utopia (level 1, 2 og 3), Pos-Phy (level 2 og 3) eller CSIX protokoller. Mottager- og sendeportene kan gå begge veier, og er fullstendig uavhengig av hverandre. Hver port har 32 datasignaler, 2 klokker, et sett med signaler og et sett med paritysignaler. MSF bussen opererer på 25 til 125 MHz, og alle signalene blir mottatt på økingskanten på klokka. MSF brikken henter pakken opp fra det fysiske lag ved at den deler opp i små m-pakker. Dette er gjort for at vi kan gjøre kortet mye mer kompatibelt med de forskjellige fysiske standarder. Vi kan stille inn kortet til å ha m-pakke størrelser på 64,128 eller 256 bytes.

- Hash enhet: Vi har støtte for å kalkulere hash verdier i hardware. Dette kan hjelpe oss når vi skal slå opp en hash verdi, fordi dette vil da gå mye raskere.

- 16 KB Scracthpadminne: Schratchpadminnet er organisert som 4K 32-bit words som kan aksessers via mikromaskinene og XScale, og støtter følgene operasjoner. Vi kan lese eller skrive 1 til 16 32-bits word i en singel instruksjon, men vi må skrive minimum 4 bytes om gangen når vi skriver til det. Det er støtte for atomisk lesing og skrive. Scracthpadminne er støttet som en tredje minneressurs i tillegg til SRAM og SDRAM, som er delt mellom Mikromaskinene og XScale. Mikromaskinene og XScalen kan distribuere minne aksess mellom disse 3 typer minne ressurser slik at man kan få flere minne aksesser som skjer parallelt. Til å flytte data mellom mikromaskinene brukes scratchringer. Det finnes 16 hardwarestøttede ringer som fungerer på samme måte som en round robin scheduler. De begynner på bestemte områder i minne og kan være fra 0 til 1024 byte store. De 12 første ringene har også funksjonalitetet som gjør at programmerer kan sjekke om de er fulle eller tomme. Hvis man trenger mer enn 16 scratchringer er vi nødt kan vi bruke software baserte ringer, men disse vil da ikke bli like raske.

- Chip wide control og status register (CAP). Denne brikken brukes til å kommunisere mellom forskjellige prosesser. Den kan også brukes til å kommunisere på kryss av mikromaksinene.

- Intel XScale kjerne tilbehør: Dette består av Interruptkontroller, 4 klokker, en serial Universal Asynchronous Receiver/Transmitter (UART) og 8 general-purpose IO (GPIO). Det er også en ytelses overvåker som inneholder registre for å måle ytelsen.

Vi har dermed sett at vi har en masse ressurser som vi kan benytte oss av. Det er 8 mikromaskiner som alle kan jobbe parallelt, alle disse er programerbare samt at det er en XScale vi kan benytte oss av. Dette vil tilsi at vi kan lage effektive og fleksible programmer.

3.2 Programvare

Det følger med software developer kit (SDK) [5] fra Radisys som lager kortet. Det følger også med en SDK fra Intel som lager IXP2400 brikkesettet. De har lagt med et par hjelpemidler som skal hjelpe utviklere med å sette opp drivere og med debugging. Det følger også med et rammeverk fra Intel hvor de har lagt med eksempler på hvordan de har ment man skal bruke kortet.

3.2.1 Radisys ENP SDK 3.5

Det følger med en SDK fra Radisys[5]. Denne inneholder driver til Pm3386, Pm3387 og Spi3 bruen. Den inneholder også en testapplikasjon som er skrevet i mikrokode. Det følger også med en linux distrubisjon Montavista[4] som kan kjøres på XScalen. Alle disse verktøyene er med for å hjelpe til med å bruke kortet på en effektiv måte.

Driverne som følger med er helt nødvendige for å motta og sende pakker med ENP2611. Det finnes mange innstillings muligheter for flytkontroll. Vi kan også ha stille driverne inn så vi kan beregne checksum på ethernetlaget på hardware.

Testapplikasjonen gir eksempler hvordan man kommer i gang. Det eneste testapplikasjonen gjør er å ta imot pakker fra en port, sende dem ut på en annen port å telle pakkene. Hele applikasjonen er skrevet i mikrokode. Den setter opp alle delene som er nødvendig for å sende pakker gjennom kortet. En viktig del er mottak (Rx) og sende (Tx) blokken som følger med SDKen.

3.2.2 Intel IXA SDK 3.51

Intel har også lagt med en SDK[3] som følger med kortet her følger det med et byggesteinsbibliotek som blant annet inneholder en workbench, for utvikling av programvare med kompilatorer, bibliotek. Dette er for å

hjelpe utviklere som skal bruke kortet med å komme i gang, ved å ha et rammeverk de kan bruke.

Utvikerworkbenchen er et windowsprogram som hjelper deg med debugging og simulering av applikasjoner som lages for mikromaskinene. Med workbenchen kan man se på hva som ligger i de forskjellige delene av minne, som scratch, sram, dram og registrene. Vi kan sette stoppepunkter som gjør at vi kan se alle verdiene på bestemte steder i programkjøringen, dette er til stor hjelp når vi debugger siden det ikke er så lett å se variablene i mikromaskinen, fordi vi har ikke noen utskrift til skjerm. Det betyr at eneste mulighet vi har uten om workbenchen er å sende verdiene opp til XScale for å så skrive dem ut der. I workbenchen er det også muligheter for å simulere i software før vi kjører det i hardware. Her vil det være enda større muligheter for debugging og kontroll. Vi kan simulerere strømmer igjennom kortet. Vi kan til og med stoppe eksekveringen av programmet når en verdi er større enn x. Denne muligheten er kun for mikromaskinen.

Det følger også med kompilatorer for å kompilere mikroC og assembler for mikrokode. Disse gjør at vi kan få koden over til det instruksjoner som kortet støtter. Grunnen til at det heter mikroC og mikrokode er at det er ikke vanlig C og assembler. MikroC ligner veldig på C men det har noe mindre funksjonalitet.. Det følger også med en linker for å linke objekt filene sammen til en kjørbar fil.

Biblioteket som følger med er satt sammen av en del standardfunkjoner både i microC og mikrokode som skal hjelpe utvikler med å spare tid når det utvikles applikasjoner for kortet. Disse bibliotekene har støtte både for mikromaskinene og XScalen.

Det følger med et rammeverk for hvordan Intel har tenkt at pakkeflyten skal foregå. Dette er ment å gi et raskt rammeverk som utviklere skal kunne bygge på. Det består av et pakkebuffersystem som skal gi bra ytelse, det er også forskjellige moduler som kan settes sammen slik at man kan bygge videre på dette.

3.3 Pakkebuffersystemet

En viktig ting i rammeverket til Intel SDK er pakkebuffersystemet, dette er et system som skal få pakken igjennom IXP 2400 på en rask og effektiv måte. Pakkebuffersystemet er bygd for at vi skal gjøre ting på forskjellige mikromaskiner samtidig, dette gjør at ting kan gjøres parallelt. Når ting skal skje parallelt kommer andre problemer, som for eksempel synkronisering. Dette slipper vi tenke på når vi benytter oss av pakkebuffersystemet. Fordi hver pakke har en egen pakkedeskriptor som flyttes mellom mikromaskinene, slik at bare en mikromaskin kan arbeide på pakken.

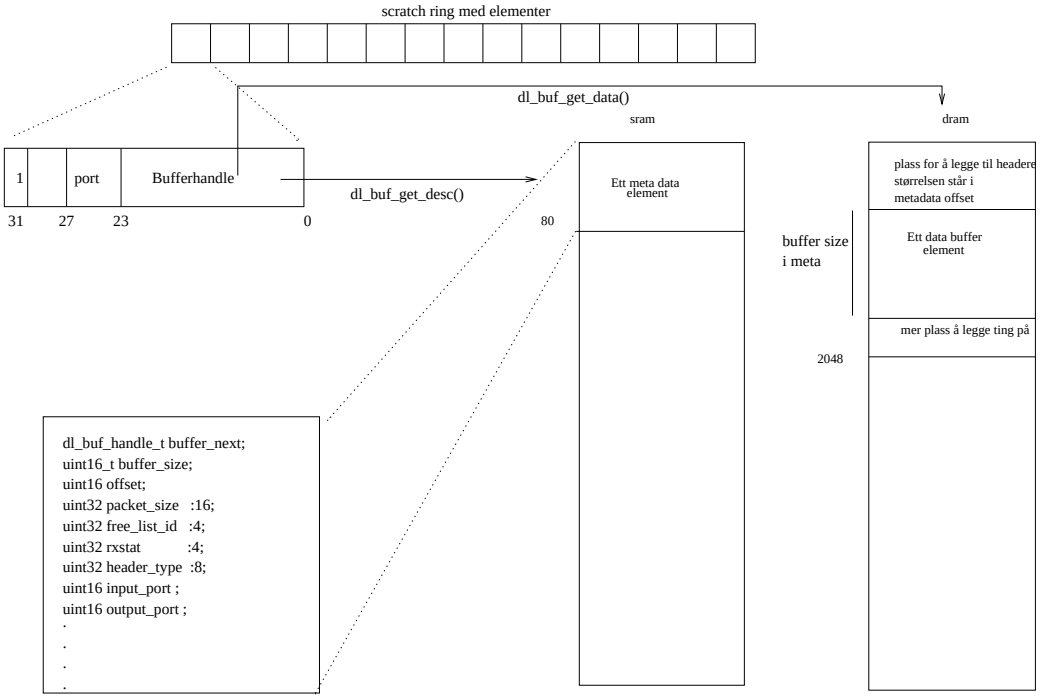
Når vi får inn en pakke legger vi dataene i SDRAM. Her ligger pakken hele tiden fra den kommer inn til den går ut. Før og etter er det ledig plass slik at vi kan legge på flere headere og forandre på størrelsen. Når pakken er ute av systemet frigis plassen. Når vi legger pakken i SDRAM lager vi en pakkedeskriptor som vi da sender videre. Når en mikromaskin har denne pakkedeskriptoren vet den hvor pakken ligger og kan arbeide på denne pakken. Pakkedeskriptoren er bare 32-bit stor slik at det vil ikke ta lang tid å sende den rundt mellom de funksjonelle enhetene. Når vi sender pakkedeskriptoren fra en mikromaskinen til den neste sender vi den på scratchringen. Figur viser 3.4 en pakkedeskriptor som ligger på en scratchring. De 24 minst signifikante bitene er en bufferdeskriptor hvor vi finner ut hvor pakken ligger ved hjelp av *dl_get_data*. *dl_get_data* er en metode som finner hvor bufferdeskriptoren peker i SDRAM.

Når det kommer inn en pakke lagres også data om pakken(meta data). Dette legges i SRAM. Disse dataene blir også liggende på samme sted så lenge vi behandler pakken på kortet. Disse kan også hentes ut av pakkedeskriptoren på samme måte som vi henter ut dataene. Vi bruker bare en annen metode som heter dl_get_desc. dl_get_desc er en metode hvor vi henter ut hvor bufferdeskriptoren peker i SRAM. Vi ser dermed at ved hjelp av pakkedeskriptoren kan vi hente ut både dataene om pakken som ligger i SDRAM og metadataene som ligger i SRAM. Dette gjør at vi ved hjelp av en liten kalkulasjon får tilgang til alt vi trenger om pakken.

Når vi gjør det på denne måten vil vi slippe å tenke på synkronisering og vi vil ha en slags pipeline konsept hvor vi kan gjøre litt med pakken så sende den videre og gjøre mer. Det betyr at ting kan skje parallelt. Dette er fordi det er kun en mikromaskin som behandler pakken om gangen. Når

en mikromaskin har tilgang på pakkedeskriptor vil den kunne behandle pakken så sender den videre til neste mikromaskin.

Figur 3.4 Bufferdeskriptor

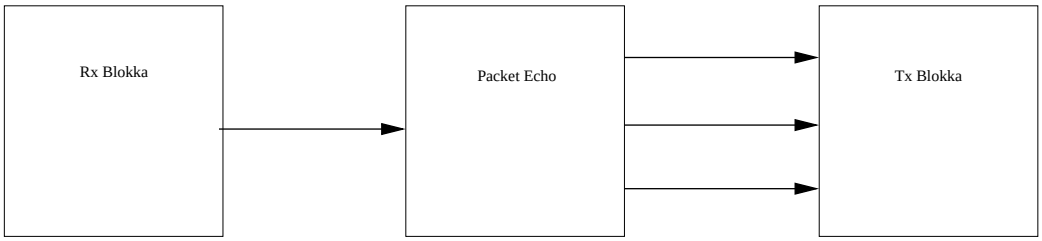


3.4 Evualering

Vi skal se litt på effektiviteten til testapplikasjonen. Ettersom vi kan kjøre flere ting i parallell og vi har dedikert hardware for å ta imot pakker og sende dem, så vil vi forvente at det vil være ganske effektivt.

Vi tester da med en testapplikasjon som bruker 3 mikromaskiner som vist i figur 3.5. Den følger med fra Radisys SDK, og er utviklet på mikrokode. Alle mikromaskinene i testapplikasjonen kjører med 8 tråder hver. Dette skal gjøre at vi inne i mikromaskinen også kan kjøre ting partallet. Det betyr at vi kan eksekvere kode i en tråd mens en annen tråd venter på system ressurser. Dette gjør at vi kan gjøre flere ting på en gang. De forskjellige mikromaskinene gjør:

Figur 3.5 Testsapplikasjonen



- Rx blokken er den første mikromaskinen i testapplikasjonen. Den skal ta imot pakkene og lagre metadata og en pakke deskriptor. Rx blokken skal også legge pakkene i SDRAM og metadataene i SRAM. Så sender den videre pakke deskriptoren til neste mikromaskin. Rx blokken hadde bare trengt å sende pakkedeskriptoren, men sender også litt av meta dataene blant annet inn port og bufferstørrelse. Her er også trådene organisert slik at vi har 2 tråder som sjekker om det har kommet inn noe på hver port. Vi vil da ha 2 tråder til overs. Disse sjekker port 4 som på dette kortet ikke eksisterer.

- Packet echo er den mikromaskinen som ligger i midten. Denne får pakkedscriptoren og noe av metadataene. Vi kunne selvsagt fått alle metadataene ut av pakkedeskriptoren. Men dette er et eksempel fra Radisys hvor de gjør det på en litt mer intuitiv måte. Det packet echo skal gjøre er bare å se hvilken port vi får inn en pakke på og setter, deretter utporten til den etterfølgende port (port 2 vil da bli 0). Porten så satt i pakkedeskriptoren fra det 24 til 27 bitet. Pakken blir så sendt videre til siste mikromaskin. Det er 3 scratchringer fra Packet echo til Tx, en til hver utport. Vi legger derfor pakken i scratchringen som tilhører den porten vi vil sende ut på. Denne mikromaskinen kjører også 8 tråder som alle har samme kode.

- Tx blokken er den som skal sende pakkene ut. Den er organisert slik at 2 tråder har ansvar for hver sin port. Disse vil hele tiden sjekke om det har kommet noen pakker inn på den scratchringen de har ansvar for å sende pakken ut på porten sin. Den får pakke deskriptoren fra forrige mikromaskin og frigir også bufferne som pakkene lå i.

Når vi tester gjennomstrømningen av testapplikasjonen gjort av Radisys ser vi fra tabell 3.1. Vi ser hvordan forskjellig pakkestørrelse påvirker systemet, og hvor mange rammer som kan gå igjennom pr sekund. Vi ser at

Pakke Størrelse (m/CRC)	Teoretisk Maks Gjennomstrømming (rammer/sek)	Målt Gjennomstrømning (rammer/sek)	Lineær Prosent
64	1.488.095	1.488.087	100%
128	844.595	844.590	100%
129	833.333	827.842	99.34%
256	452.899	452.897	100%
257	449.640	433.692	96.45%
512	234.962	234.961	100%
513	234.082	234.082	100%
1024	119.732	199.732	100%
1025	119.503	119.502	100%
1518	81.274	81.275	100%
Variasjon	157.370	157.363	99.99%

Tabell 3.1: Måling av gjennomstrømning for testapplikasjonen

de fleste pakkestørrelser vil fungere ganske bra. Den som fungerer dårligst er 257 KB hvor vi får 96.45% av rammene igjennom. Dette kan være fordi størrelsen på pakken gjør at vi ikke får plass i ett minne segment. Dette betyr at vi har en god implementasjon i testapplikasjonen. Det passer veldig bra siden vi skal bruke Rx og Tx blokken i vår egenapplikasjon.

Vi har videre utvidet testappliksjonen slik at den ikke bare skal sende på neste port, men vi skal i tillegg også endre på headere i pakken. Vi skal snu IP-adressene og ethernettadressene, og vi trenger derfor bare å gjøre forandringer i packetechoblokken siden de andre bare sender og mottar pakker. Vi har valgt å skrive om packet echo til mikroC siden det da er mye lettere å lese og skrive kode. Selve oppsettet vil da ligne testapplikasjonen bortsett fra at vi bytter ut packet echo. Når vi tester dette sender vi pakker fra en maskin til kortet som snur headeren og sender tilbake til avsender. Det vil da se ut som vi kommuniserer med en annen maskin, men vi snakker bare med oss selv.

Utvidelsen er et samarbeidsprosjekt mellom meg og Andreas Petlund [23]. Selve testingen er det Andreas som har gjort for sin masteroppgave. Han har også testet mot implementasjoner med bytting av headere på XScale, Linux i userspace og Linux på kernelspace.

Testene er gjort ved at vi sender en ping pakke til applikasjonen som kjører

Pakke størrelse	Maks	Min	Gj.snitts	Median	Std avvik	antall
Linux-vert						
userspace						
98 bytes	19331μs	11μs	111μs	105μs	310μs	99380
1497 bytes	16216μs	16μs	172μs	174μs	102μs	99960
Linux-vert						
kernelspace						
98 bytes	19723μs	11μs	111μs	101μs	378μs	98902
1497 bytes	19092μs	16μs	168μs	178μs	182μs	99806
IXP						
XScale						
98 bytes	1145μs	54μs	123μs	109μs	44μs	100000
1497 bytes	1146μs	110μs	171μs	173μs	45μs	100000
IXP						
mikromaskin						
98 bytes	1169μs	28μs	98μs	99μs	38μs	100000
1497 bytes	1150μs	84μs	151μs	162μs	50μs	100000

Tabell 3.2: Måling av ping tider

på IXP2400 kortet. Kortet snur headerene, og sender pakken tilbake til den maskinen som pinger. Så sender maskinen et pingsvar som snus i IXP2400 tilbake til seg selv. Vi måler da hvor lang tid som er brukt fra når vi pinger til vi får svar. Vi sender 100 000 pingpakker og måler tiden fra vi sendte pakken til vi får ping svar tilbake.

Vi ser fra resultatene i tabellen 3.2 at vi vil få en forskjell i ytelsen ved å kjøre på mikromaskinene i forhold til å kjøre på en vanlig Linux-vert. Vi ser at vi får alle pakkene igjennom når vi bruker IXP NPE[7]. Når vi kjører på en Linux-vert mister vi pakker både når vi har tilsvarende applikasjon i userspace og kernelspace. Vi ser også at det er raskere å gjøre ting på mikromaskinene mot å gjøre det på XScale. Det å gjøre ting på XScalen er faktisk enda tregere enn å gjøre ting på Linux-verten ifølge gjennomsnittlige målinger. Men dette er uten last på Linux-verten hadde vi derimot hatt operasjoner som hadde kjørt i bakgrunnen ville XScalen blitt raskere. Ser vi derimot på standard avvik ser vi at både mikromaskinene og XScale er mindre mye mindre enn Linux-verten, som betyr at det blir mindre spredning på resultatene.

3.5 Sammendrag

Vi har i dette kapittelet sett litt på hva nettverksprosesseringskort er godt for. Vi har spesielt sett på IXP2400 NPE, og hvordan dette er bygd opp. Vi har sett på de SDKene som følger med kortet og hvilken funksjonalitet som det er i dem. Videre har vi forklart pakkebuffersystemet som brukes på kortet, hvor vi legger pakken i SDRAM og metadataene i SRAM. Mellom mikromaskinene sender vi bare en liten pakkedeskriptor.

Vi har også vist at vi har god gjennomstrømning på kortet. Vi sjekket forskjellig type pakkestørrelse og vist at i snitt hadde vi en effektivitet på 99.99%, dette er jo veldig bra. Vi utvidet også testapplikasjonen til å snu på headerene i pakken og testet dette mot andre gjeldene implementasjoner på Linux-verten. Utfra dette så vi at IXP NPE hadde bedre gjennomsnittstider for mikromaskinene.

Vi vil ut ifra de testene vi har gjort så langt tro at det vil bli en forbedring å bruke IXP NPE fremfor å bare bruke en Linux maskin. Vi skal derfor i det neste kapittelet legge mer funksjonalitet på IXP NPE for å se om det også vil lønne seg for et større system.

Kapittel 4

Design

Vi skal lage en proxy som skal gi kvalitetssikring for strømmer vi sender igjennom proxyen. En av egenskapene den skal ha er at den vil prøve å få igjennom de viktigste pakkene først. Vi velger derfor å implementere en prioritetskø. Vi vil også kvalitetssikre systemet vårt mot burst, og at noen strømmer stjeler hele båndbredden. Vi vil derfor også implementere en shaper også. Dette skal implementeres på et IXP 2400 NPE kort [3]. Vi håper da dette vil gi oss mindre latens og jitter enn en vanlig Linux maskin. Vi håper også å avlaste verten ved å bruke IXP 2400 NPE.

4.1 Utforming på grunnlag av Intel SDK

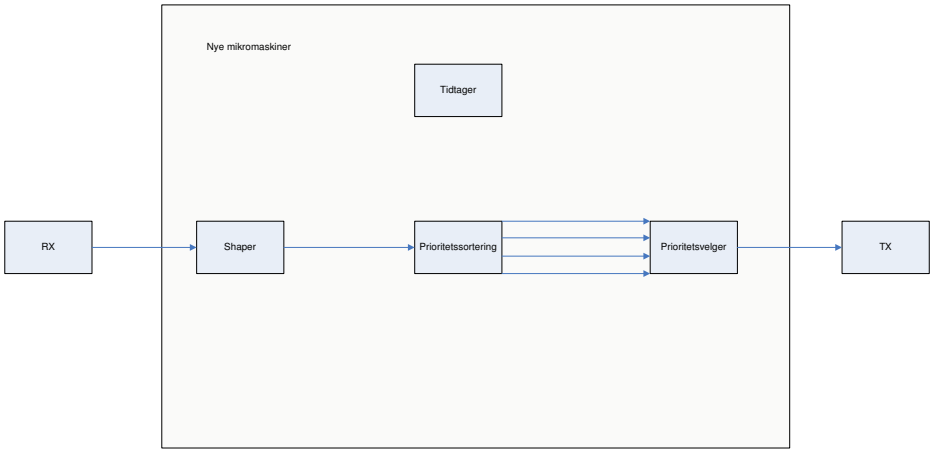
Når vi skal lage applikasjoner for IXP 2400 kortet vil vi gjerne utnytte oss av styrkene til kortet. Siden vi vet at kortet har en XScale og 8 mikromaskiner som kjører uavhengig av hverandre vil vi gjerne prøve å spre koden på en slik måte at det meste av koden kjøres parallelt. Vi kan ved bruk av scratchringene kommunisere gjennom delt minne mellom de ulike mikromaskinene. Vi kan dermed ved hjelp av scratchringer koble sammen flere mikromaskiner, slik at de kan jobbe sammen. Vi kan velge at alle kjører parallelt eller vi kan kjøre et pipeline hvor alle mikromaskinene gjør en del av den totale jobben. Inne i mikromaskinene har vi også 8 forskjellige tråder som kan alle kjøre forskjellig kode, dette gjør at vi har veldig fleksibelt system med masse innstillings muligheter.

Vi har valgt å ta i bruk et pipelinekonsept hvor hver mikromaskin gjør litt, og sender pakkene videre. Det passer veldig bra med bufferdeskriptoren i figur 3.4 som vi så på i forrige kapittel. Vi kan med denne metoden legge data og metadataene på et sted, og vi har en bufferdeskriptor som kan henvise oss til data i begge typer minne. Buffedeskriptoren er 24 bit stor og får plass i en int, det vil ikke lang tid å sende den videre til neste mikromaskin. I kapittel 3 forklarte vi at vi har 16 hardware støttede scratchringer som vi kan lese og skrive til fra alle mikromaskinene. Vi kan skrive og lese 1 til 16 32-bits word fra dem i en atomisk operasjon. Dette gjør at vi kan veldig raskt overføre data mellom mikromaskinene. Vi skriver 1 32-bits word dette gjør at vi sparer buffer plass på scratchringen. Vi har veldig stor fleksibilitet men hvordan vi organiserer pakkeflyten vi kan kjøre alle mikromaskinene parallelt, hvor vi kopierte pakkedeskritoren slik at alle hadde tilgang til den samme pakken. Dette hadde ført til at vi måttet hatt låser slik at vi ikke overskrev hverandre. Siden vi ikke vil at applikasjonen skal slite med låser fordi vi prøver å aksessere en pakke fra to forskjellige mikromaskiner, har vi valgt å bruke ett pipelinekonsept. Det vil fungere bra med pakkedeskriptoroppsettet, fordi den mikromaskinen som har pakkedeskriptoren har pakken, gjør de endringene som den skal gjøre, og sender den videre til neste mikromaskin. Det vil da kun være en mikromaskin av gangen som har tilgang til pakken.

Vi har satt opp 6 mikromaskiner som vi ser fra figur 4.1 hvor pilene er forskjellige scratchringer og boksene er mikromaskiner.

- Rx-blokken henter pakkene fra det fysiske lag og legger dem inn i bufferdeskriptorsystemet.
- Shaperen skal sørge for at en strøm med pakker ikke stjeler all båndbredden og ødelegger for andre mediastrømmer som går igjennom kortet.
- Prioritetssorteringen skal sortere mediastrømmens pakker etter hvilken prioritet de har på 4 forskjellige scratchringer.
- Prioritetsvelgeren skal ta ut pakkene fra scratchringene på en slik måte at vi alltid tar de høyeste prioritets pakkene først.
- Tx-blokken får en pakkedeskriptor og sender pakken ned på det fysiske lag .
- Tidtagerblokken skal måle tid. Den brukes hovedsaklig til testing.

Figur 4.1 Flytskjema for mikromaskinene



4.2 Komponenter Fra SDK

Når vi skal bygge applikasjonen vår ser vi hva som eksisterer fra før i Intel SDK. Vi har allerede sett fra forrige kapittel at Rx og Tx blokkene som testappliksjonen til Radisys bruker er effektive. Vi vil derfor gjenbruke disse. Disse blokkene skal som skal stå på enden, og de sørger for å imot pakkene og sender dem ut.

4.2.1 Rx

Rx blokken henter pakken fra msf til mikromaskinen. Måten den gjør dette på er at den får små m-pakker og setter sammen igjen i Rx blokken. Når den har fått pakken legger den data pakken inn i SDRAM og informasjon om pakken inn i SRAM. Informasjonen om pakken kaller vi for metadata. Når pakken og metadataene er lagret lager vi en bufferdeskriptor slik at vi kan hente begge deler av pakken opp igjen på et senere tidspunkt. Rx blokken har også som ansvar å initiere alle komponenter som trengs. Når den har fått laget pakkedeskriptoren sendes den videre på scratchringen slik at neste mikromaskin kan gjøre det som den skal gjøre. Denne blokken er med i Radisys sin SDK og målinger som vi viste i forrige kapittel fra tabell 3.1 viser at den er effektiv.

4.2.2 Tx

Tx blokken kan få pakker på 3 forskjellige scratch ringer. Hvilken scratchring pakken kommer inn på bestemmer hvilken port den skal ut på. Blokken er bygd opp slik at det er 2 kontekster for hver fysisk port, det vil si at det er 2 tråder som egentlig ikke gjør noen ting, bare går i løkke og sjekker om det har kommet noe inn på en scratchring som ikke brukes. Hvis vi ser på blokken vil vi kunne dele opp hele trådenes løp i 3 faser. I første fase undersøker om det er kommet noe inn på scratchringen. Andre fase deler pakken opp i m-pakker, og tredje fase sender pakken ut på MSF. Som sagt er det 2 tråder for hver fysisk port, og disse samarbeider for å få så lite overhead som mulig med hensyn på minneaksesser.

4.3 Nye Komponenter

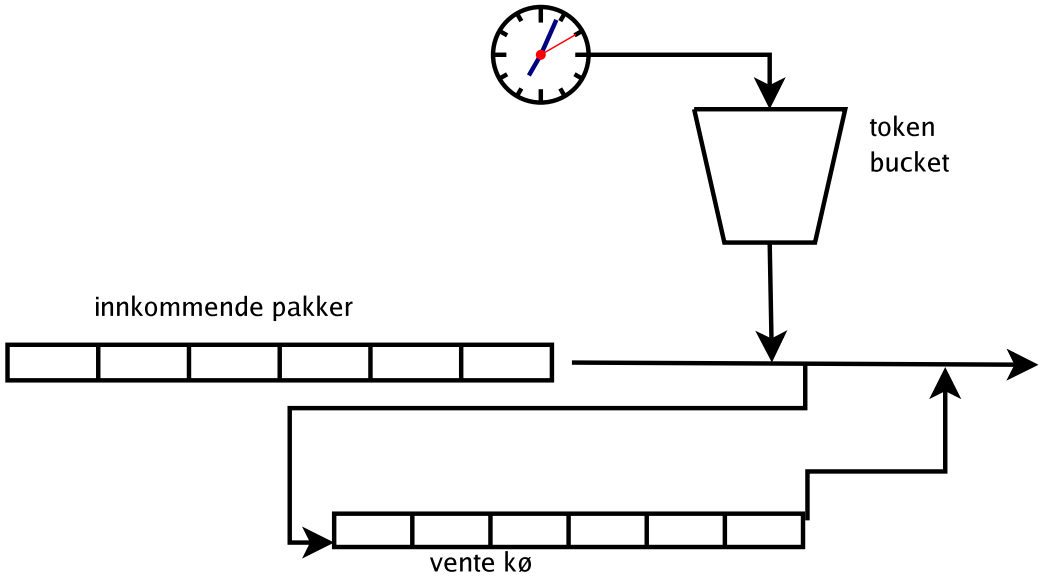
Siden vi ikke har alle komponentene vi trenger i SDKen trenger vi å lage våre egne mikroblokker. Den første vi vil sørge for er at vi ikke får strømmer som kan ødelegge for andre ved at de tar store deler av båndbredden. For at vi skal prøve å sikre dette best mulig vil vi sette opp en shaper. Vi vil også ha en prioritetskø som skal sørge for at vi vil få de mest viktigste delene av strømmen igjennom hvis det blir press på tjeneren.

4.3.1 Shaper

Den første mikromaskinen som ligger i pipelinen kjører shaperblokken. Den skal sørge for at det ikke plutselig kommer en mediastrøm som tar større en båndbredde enn hva som var avtalt i forhåndsforhandlingene. Hvis dette skjer må vi legge de pakkene som overskrider den satte båndbredden i en ventekø. Dette gjør vi fordi hvis vi tillater at det kommer en slik topp med pakker kan dette ødelegge for prioriteringen til andre strømmer og til og med sulte dem ut. For å løse dette bruker vi en "token bucket" som vi ser i figur 4.2.

Prinsippet til en token bucket er at hver mediastrøm får et visst antall billetter per tidsenhet, og så lenge det er tilgjenglige billetter for mediastrømme vil vi slippe igjennom pakker ved at hver pakke konsumerer en billett. Hvis det ikke er ledige billetter vil vi legge pakkene på en ventekø. Vi kan bestemme vi vil ha et endelig antall billetter som vi kan ha, eller om vi vil sette et maks tak. Fordelen ved å sette et maksimum billetter vi kan få er at vi ikke kan spare opp masse billetter. Dette gjør at vi kan få en liten topp over det som vi skal shape på. Fordelen med den er at vi får sendt litt mer pakker og toppene vil normalt sett ikke bli alt for store. Hvor mange billetter som mediastrømmen får, blir beregnet fra hvilken båndbredde og pakkestørrelse som er satt opp i starten av mediastrømmen. Vi har valgt å ikke ha noen tak på billettene fordi vi har liten ventekø vi vil derfor ikke få veldig masse pakker som kommer samtidig. Ventekøen som vi legger pakker i når det ikke er ledige billetter vil vi ikke ha alt for stort. Det kan medføre at vi mister tidsfristen for pakken og den blir forsinket. Når køen blir full må vi begynne å kaste pakker. Vi kaster de pakkene som har ligget lengst på køen siden de kan ha misset fristen allerede.

Figur 4.2 token bucket

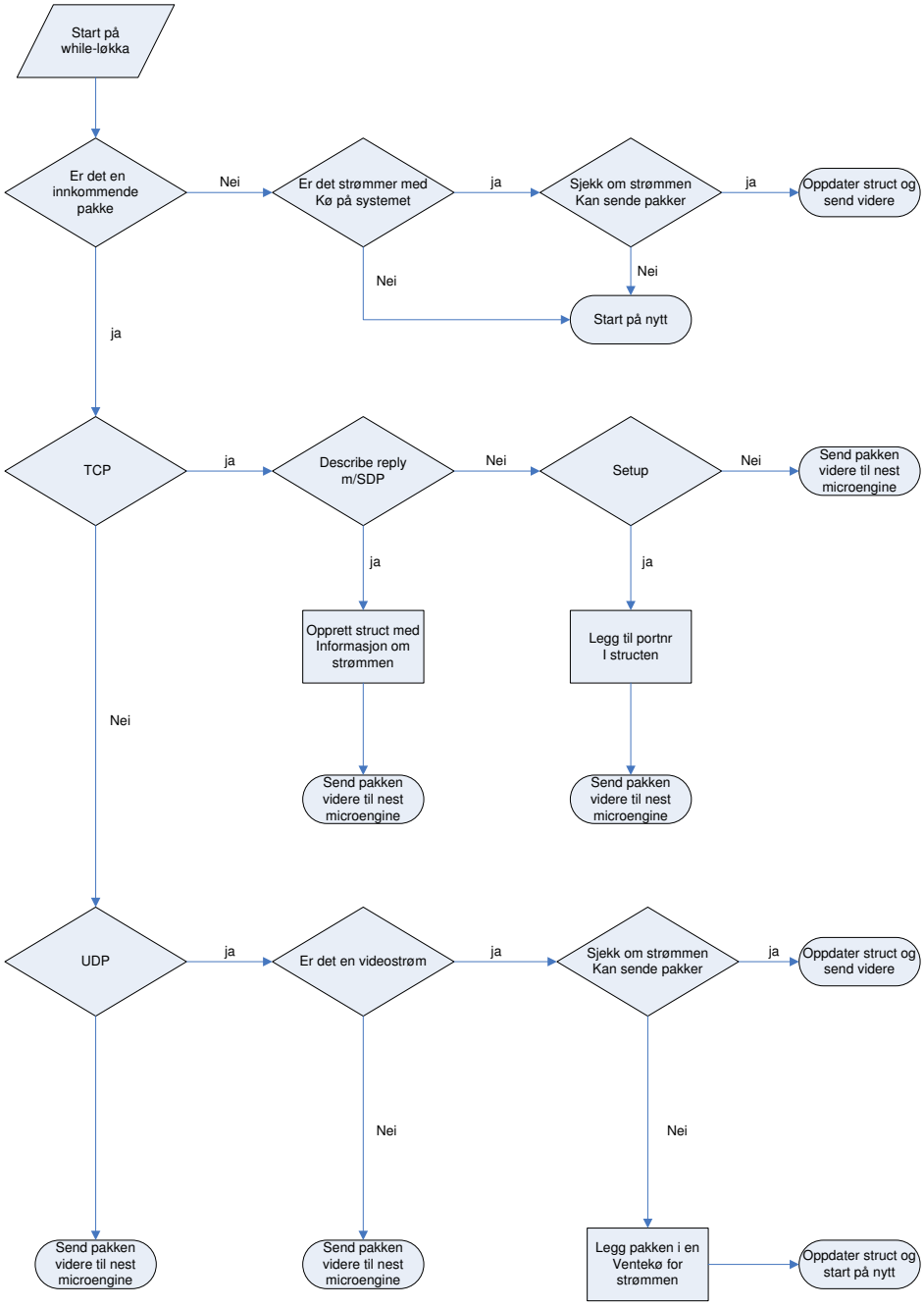


De pakkene som ikke er del av en mediastrøm vil ikke bli prioritert, vi trenger derfor ikke å begrense disse pakkene. Pakkene vil bli lagt i kø som "best effort" pakker, derfor vil dette i liten grad påvirke prioriteringen. Siden vi bare shaper mediastrømmer vil "work conservation" bare oppnås med nok "best effort" trafikk. Det vil si at hvis vi kan kaste pakker selv om det er ledig båndbredde. Vi ser på figur 4.3 hvordan gangen i shaperblokken er.

Shaperen er en "token bucket" som er som skal begrense pakkene som kommer igjennom. Dette gjøres ved at vi bare slipper pakker når det er billetter ledige. Vi har designet shaperen slik at vi skal beregne tiden det tar fra første pakken i strømmen til pakken som kommer inn. Vi sjekker deretter hvor mange pakker som skulle været sendt på denne tids differansen. Vi sjekker deretter hvor mange pakker vi har sendt, hvis vi har sendt mindre pakker enn det som skulle vært sendt gir vi en billett. Vi kan på denne måten på en rask måte sjekke om vi har ledig billetter. Det krever bare at vi måler tidsdifferansen fra første pakke i strømmen og kalkulere hvor mange pakker som skulle været sendt.

Siden mikromaskinene hele tiden vil gå i en løkke hvor det første vi gjør er å sjekke om det har kommet inn noen pakker på fra forrige mikromaskinen, vil det si at vi egentlig ikke har noe å gjøre hvis det ikke kommer inn

Figur 4.3 flytskjema for shaper



noen pakker. Vi har vi prøvd å utnytte denne tiden ved å sjekke om noen av strømmene som er allokert på proxyen, har noen pakker i en ventekø. Hvis det er strømmer som har pakker på en ventekø vil vi sjekke om vi kan sende noen av dem. Dette gjøres ved å sjekke om det har blitt ledig noen billetter for den strømmen. Dersom det er ledige billetter vil vi sende en pakke til neste mikromaskin. Dette er gjort for å prøve å utnytte mikromaskinene maksimalt. Hvis det derimot ikke er noen ledige billetter vil vi ikke gjøre noen ting.

4.3.2 Design Av adgangskontroll

Det kan også tenkes at vi vil kunne bestemme om vi vil tillate en ny strøm igjennom proxyen. Det ville vært bra om vi hadde slik kontroll, siden da kan vi avise strømmer når all ledig båndbredde var brukt opp. Når vi oppretter en ny strøm bestemmer vi hvor mye båndbredde den skal få. Vi vet derfor hvor mye båndbredde de allokerte strømmene har til sammen. Når vi setter opp en ny strøm vet vi om vi har plass til denne nye strømmen, eller om vi må avise den. Dette vet vi fordi vi vet hvor båndbredde vi har brukt av den totale båndbredden. Vi vil ikke ta hensyn til "best effort " trafikken det er kun strømmene som må tillates eller avises. Dette ville spart oss for masse ressurser siden vi da ikke kan allokere for mange strømmer slik at vi er nødt til å levere lavere kvalitet på alle strømmene. Vi må vel med en slikt design sende pakker tilbake ,som gjør at vi stopper TCP-strømmen til klienten. Dette har vi ikke rukket å implementere på vårt system.

4.3.3 Parsing Av Mediakontrollpakker

En viktig del av shaperen er at vi parser mediakotrollpakkene for å opprette nye strømmer. Disse pakkene er RTSP-DESCRIBE svar med SDP og RTSP-SETUP. Med disse pakkene skal vi få nok informasjon til å kunne shape strømmen på en bra måte. Vi får i RTSP-DESCRIPE svar med SDP pakkestørrelse og båndbredden filmen er ment å spille på. Denne pakken bruker vi også til å identifisere at vi skal opprette en ny strøm gjennom proxyen. Fra RTSP-SETUP får vi hvilken port vi skal sende RTP-strømmen på. Denne porten veldig viktig for å kunne identifisere hvilken strøm vi

håndterer, hvis det er flere strømmer til samme IP-adresse. For å finne ut når vi får inn en RTSP-DESCRIBE svar med SDP sjekker vi om "Content-Type" i RTSP-protokollen er lik "application/sdp".

4.3.4 Prioritets Sortering

Vi har mange forkjellig måter vi kan prioritere pakkene på. De mest kjente metodene er de som vi så på i kapittel 2 som var EF, AF, CBQ og WFQ. Siden vi har flere forskjellige mikomaskiner velger vi på benytte oss av 2 forskjellige mikromaskiner til å prioritere pakkene.

Prioriteringssorteringen skal plukke ut mediapakkene og sortere dem etter prioritet. Alle de andre pakkene vil da håndteres som "best effort" trafikk.

Vi lar RTSP og RTCP pakkene være 4.prioritet. Disse vil gå igjennom så lenge det er kapasitet. RTSP pakker består gjerne av oppkoblingspakker til tjeneren eller avsluttningspakker. Når vi begynner å kaste pakker fordi køene blir fulle er det allerede ganske mye last på proxyen. Da er det heller ikke noen vits å tillate flere oppkoblinger igjennom proxyen. Tjeneren vil da kanskje ikke motta disse pakkene, og vil dermed ikke starte en ny strøm. Siden disse pakkene sendes med TCP vil de bli sendt på nytt det kan derfor hende at de kommer frem tilslutt. Vi kunne også ha hatt oppkoblingspakkene som 1. prioritet, men dette ville ført til at vi ville fått flere mediastrømmer igjennom, og da ville vi mistet flere pakker når det ble press på proxyen. Det hadde vært bedre når en klient ville avslutte en mediastrøm siden da ville tjeneren slutte å sende pakker på et tidligere tidspunkt. På den måten ville det blitt en mer effektiv tjener, men siden vi heller vil at de strømmene som blir satt opp skal ha bedre kvalitet legger vi oppkoblings- og avslutnings RTSP-pakken ned til 4. prioritet. Et av problemene til å la RTSP pakken gå igjennom som 4. prioritet var at vi hvis vi avsluttet strømmen på ett tidlig tidspunkt ville det gått litt tid før pakkene ville ha kommet igjennom. Det er hvis pakken hadde kommet gjennom i det hele tatt. En mulig løsning på dette hadde vært at vi kunne gått igjennom alle RTSP pakkene, og hvis det var en teardown kunne vi ha satt denne til 1. prioritet. Dette er ikke gjort på grunn av programmeringskompleksitet.

Grunnen til at vi har valgt å ha 4 scratchringer fra denne til neste mikromaskin er at vi må sortere pakkene etter prioritet, og vi må uansett legge dem over i en scratchring til neste mikromaskin. Vi kan enten sortere dem ved hjelp av en liste hvor vi må traversere hele listen for å finne ut hvor vi skal legge dem, eller vi kan benytte oss av at vi bare kan slå opp et felt og legge dem rett over i scratchringen til neste mikromaskin etter hvilken prioritet pakken skal ha. Det vil ta mye kortere tid å legge dem rett over i scratchringen derfor gjør vi dette. Hvis det viser seg at scratchringen er full kaster vi pakken. Køene blir fulle fordi vi ikke rekker å ta dem ut i på neste mikromaskin. Siden vi har designet vårt system slik at vi skal ta ut flest pakker fra de høyeste prioritetene, vil det først bli kø i de laveste prioritetene.

4.3.5 Prioritetsvelger

Siden vi har delt opp prioriteringen hvor vi sorterer i en mikromaskin og tar ut i den neste har vi valgt en enkel løsning for å ta ut av køene. Prioritetsvelgerblokken skal ta ut pakker fra de 4 scratchringene og sende dem videre til Tx blokken. Det vi vil er at vi alltid tar mest ut fra 1. prioritetskøen siden de viktigste pakkene er der, men vi vil heller ikke sulte ut de andre køene fordi det kommer for mange 1. prioritets pakker. Vi kan derfor ikke bare ta ut av 1. prioritet så lenge det er pakker der, vi må ha et system som tar litt fra alle køene. Måten vi løser dette på er at vi har en for-løkke som tar ut 4 pakker fra 1. prioritetskøen deretter tar vi ut 3 pakker fra 2. prioritetskøen. Fra den 3. tar vi ut 2 pakker og den siste 1 pakke. Dette vil gjøre at vi alltid vil få 1.prioriteten litt før den egentlig skal komme frem. Vi kan også stille om slik at vi sjekker om det har kommet inn noen av de høyre prioritets køene. Det vil da bli en ren prioritetskø som ikke er rettferdig.

Vi kan også bruke andre strategier for å ta ut av køene. Vi trenger ikke ta et statisk antall pakker fra hver kø. Vi har valgt å gjøre det på den måten fordi det var enkelt. Vi kan selvsagt også få en mer komplekst utvelgelse av hvilken kø vi skal ta pakken på. Vi kunne for eksempel sjekke hvor mange pakker det ligger på den køen vi skal legge pakkene videre på. Da ville vi Utfra hvor full den var nektet laver prioritets pakker.

4.3.6 Tidtager

Tidtagerblokken er kun for vi skal få målt tiden det tar mellom mikromaskinene, siden vi trenger en felles tid for alle. Når vi skal måle en tid sender vi et signal til denne blokken, og legger med en peker til hvor vi kan legge tiden vi skal bruke. Når tidtagerblokken får et signal leser den av tiden og legger tiden til hvor pekeren peker. Pekeren peker til et bestemt sted for den pakken vi skal måle. Dette gjøres for at vi kan lese av tiden. Den er hovedsaklig ment for testing så tidene vi får må vi lese av når vi mottar pakken til klienten. Vi må gjøre det på denne måten siden vi skal ha en tid som er felles for alle mikromaskinene. Det er egentlig en egen tidtager i hver eneste mikromaskin. De skulle ha fungert slik at vi kunne synkronisert dem, slik fungerte det ikke. Derfor måtte vi lage denne blokken. Vi har en egentid i shaperen for der trenger vi ikke en felles tid for alle mikromaskinene. Klokken blir generert ved at vi øker telleren med 1 for hvert 16 slag som prosessoren på XScalen genererer.

4.4 Sammendrag

Vi har i dette kapittelet sett på forskjellige måter vi kan designe systemet. Vi har sett at vi har fordelt oppgavene utover de forskjellige mikromsakinene. Vi har sett på hvilke oppgaver som vi har lagt på hver del. Vi har valgt å shape strømmene separat fra hverandre, slik at hver strøm kan kun ha den båndbredde de er satt opp med. Det vil si at vi kan kaste pakker selv om vi har ledig båndbredde. Vi har lagt shaperen helt separat fra prioritetkøen slik at vi kan risikere å vi kaster 1. prioritets pakker i shaper og slippe igjennom en 4.prioritets pakke. Prioritetskøen vår vil alltid prøve å få igjennom de viktigste pakkene. og kaste de som den ikke greier å få sendt.

Kapittel 5

Implementasjon

I det følgene kapittelet ser vi på komponentene. Vi skal gå gjennom de implementasjonspesifike avgjørelsene som vi gjorde i implementasjonen av systemet. De bygger på forrige kapittelet hvor vi gikk gjennom designet til systemet.

5.1 Shaper

Formålet med en shaper er at den skal hindre et stort utbrudd av pakker på en gang. Det vil si at hvis en videostrøm plutselig sender et utbrudd som er på 800 kbps og videostrømmen er satt opp til å sende på 700 kbps vil da shaperen ikke slippe igjennom mer enn 700 kbps. De resterende pakkene vil da bli lagt i en ventekø. Så lenge det er pakker i ventekøen vil nye pakker bli lagt til ventekøen isteden for at de blir sendt videre i systemet. Dette skjer ved hjelp av et billettsystem hvor strømmen får billetter etter hvor mye båndbredde videostrømmen er satt opp med. Når en videostrøm får en ny billett vil vi ta ut den første pakken i ventekøen og sende den videre til priortetssorrtingsblokken. Dette skal hindre at en videostrøm stjeler båndbredde fra andre strømmer. Shaperen oppretter en ny struct for hver strøm som opprettes. Denne skal holde orden på hvor mange billetter videostrømmen har til en hver tid. Den skal også holde orden på hvor mange pakker som er i en ventekø, hvor mange pakker som er sendt. Vi kan se denne structen i figur 5.1 som ligger i DRAM.

Figur 5.1 shaper Struct

```
typedef struct
{
   union
   {
      struct
      {
         unsigned int ipdest;
         unsigned int token;

         unsigned int head;
         unsigned int tail;

         unsigned int port;
         unsigned long long bandwith;

         unsigned long long packet_size;
         unsigned long long time;

         unsigned long long diff;
         unsigned int qpacket;

         unsigned int packet_sent;
         unsigned int packets_waiting[20];

         streamptr next;
         streamptr prev;
      };
      unsigned int value[33];
   };
} stream_holder;
```

Når vi får inn en ny pakke brukes *ipdest* og *port* til å identifisere hvilken strøm pakken tilhører. Vi kunne kanskje ha greid oss med kun IP-adressen, men da kan ikke samme mottager ha 2 forskjellige strømmer. Når klienten kobler seg opp mot videotjeneren sender den en RTSP DESCRIBE. Tjeneren sender tilbake et svar på forespørselen med SDP beskrivelse, der det står beskrevet båndbredden og pakkestørrelse, som videostrømmen skal sendes med. Vi legger pakkestørrelsen og båndbredden inn i structen *packet_size* og *bandwith*. Denne informasjonen bruker shaperen til å bestemme hvor mange billetter som videostrømmen skal få ved hjelp av formelen 5.1. Vi har ikke noen adgangskontroll som sier at proxyen ikke har noen ledig plass, men siden vi har lagt RTSP pakkene som 4. prioritet vil disse pakkene bruke mye mer tid hvis det er mye trafikk. Det kan till og med hende at de ikke kommer frem og da vil ikke klienten greie å koble seg opp mot tjeneren. Dette vil lage litt ekstra trafikk men dette er lite i forhold til resten av trafikken.

$$token = \frac{bandwith * diff}{packetsize} \tag{5.1}$$

Når den første RTP pakken til videostrømmen kommer igjennom proxyen vil *time* bli satt. Det blir referansen som alle pakkene i videostrømmen vil bli målt mot for å finne *timediff* som er differansen til den første RTP pakken. Feltet *token* sier hvor mange billetter som videostrømmen har fra den første pakken i videostrømmen, men vi må trekke fra hvor mange pakker som er sendt for å finne hvor mange billetter vi egentlig har tilgjenglig. Vi vil alltid anta at vi har fast pakkestørrelse siden vi kan spesifisere dette ved start av en strøm.

Hvis det ikke er noen ledige billetter vil vi legge pakken i en ventekø, og i structen har vi en variabel *qpacket* som sier hvor mange pakker som ligger i ventekø. Hvis det er noen pakker i ventekø vil vi ikke slippe igjennom nye pakker som kommer igjennom proxyen, de vil også bli lagt til i ventekøen. Når vi får en ny billett vil vi ta ut den første pakken fra ventekøen og sende den videre til neste mikromaskin. Ventekøen er et array som ligger i structen og heter *packetswaiting*. Den har et hode og en hale peker, som vi bruker til å legge inn bufferhandle til pakken til vi igjen har nok billetter til å sende pakken videre. Ventekøen er 20 elementer stort siden vi ikke vil ha altfor lang liste med ventede pakker. Dette er fordi vi vil bare ta unna et lite utbrudd med pakker som sendes altfor fort. Hvis ventekøen blir full

kaster vi den eldste pakken og setter inn den nye pakken. Vi gjør det på denne måten for å gjøre det mindre sannsynlig at det vil ligge en pakke på ventekø som i utgangspunktet er forsinket allerede.

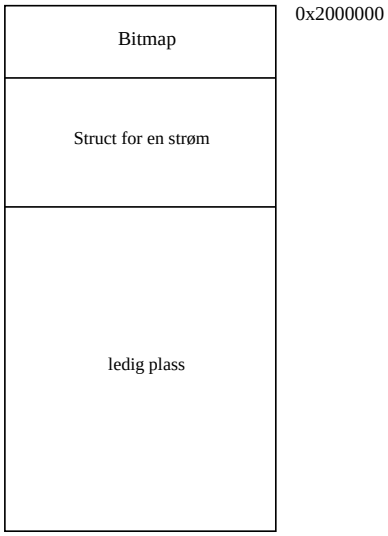
Vi har lagt alle strømstructene inn i en lenket liste som vi må gå igjennom for å finne den bestemte structen som tilhører videostrømmen. Dette skjer ved at vi finner IP-adressen og portnummeret til videostrømmen. Portnummeret til videostrømmen finner vi fra RTSP SETUP. Der står det klientens RTP portnummer og RTCP portnummer. IP-adressen er den IP-adressen som sender RTSP SETUP. Siden vi har en lenket liste til å holde orden på alle de forskjellige videostrømmene betyr det at hvis vi har mange videostrømmer, vil det ta lenger tid enn hvis vi har få videostrømmer. Det ville da kanskje vært lurt å benytte oss av en hash funksjon som det er hardwarestøtte for på kortet. Da ville det antageligvis gått raskere med flere strømmer. Dette har ikke blitt implementer på grunn av tidsmangel.

Siden det ikke er noen metoder for å allokere minneområder på mikromaskinenene må man sørge for å skrive allokerings koden selv. Vi ser fra figur 5.2 hvordan vi har organisert allokeringen. Vi har satt av et minne område fra 0x2000000 og utover i DRAM hvor den første delen er en bitmap tabell av områdene hvor vi kan legge stuctene av videostømmen. etter bitmape tabellen ser vi ett område som er i bruk. Alle minne områdene vil være like store. Når et minne områdene er i bruk vil vi skrive et 1 tall i tabellen slik at ikke en annen videostrøm vil skrive over verdiene. Plassen i tabellen sier oss hvilket området i minne vi snakker om fordi alle områdene er like store, vil plasseringen i tabellen bestemme offset til tildelte minne område. Vi allokerer det første ledige minnområde når vi får inn et svar fra RTSP DESCRIBE med SDP. Da nuller vi ut område som vi har fått tildelt og setter inn de verdiene som tilhører den nye videostrømmen. Vi frigjør minneområdet når vi får inn en pakke med TCP-FIN da setter vi bare inn 0 i tabellen over ledige områder, og en annen videostrøm kan da bruke den delen av minnet.

Shaperen ligger like etter Rx blokken, siden vi vil sørge for at strømmene med store utbrudd ikke skal ødelegge for prioriteringen på det senere tidspunkt. Shaperblokken leser scratchringen til Rx blokken, ser hva slags pakke det er, og skiller på 3 forsjellige pakker: svar fra en RTSP-DESCRIBE med SPD, RTSP SETUP, og RTP. Resten av pakkene sendes bare videre til neste mikromaskin. Et svar på en RTSP DECRIBE med SDP blir som nevnt over brukt til å sette videostrømmens båndbredde og pakkestørrelse. Denne pakken sier oss også at det kommer en ny strøm vi må derfor allokere

et minne område til strømmen som vi forklarte i forrige kapittel. Når det kommer inn en RTSP SETUP vil vi sette opp hvilket portnummer som videostrømmen bruker. Når en RTP-pakke mottas vil shaperblokken sjekke om det er noen ledige billetter og sende den videre hvis det er ledige billetter, ellers vil den bli lagt på en ventekø. Shaper blokken vil alltid gå i en løkke og sjekke om det kommer pakker inn på scratchringen. Den vil behandle pakker etter mønsteret forklart over, men hvis det ikke kommer inn noen pakker vil shaper blokken gå igjennom alle de eksisterende videostrømmene og sjekke om det er noen som har pakker i en ventekø. Hvis det er en videostrøm med ventekø vil shaperblokken sjekke om det er ledige billetter for den videostrømmen og sende ut like mange pakker som det er ledige billetter.

Figur 5.2 Minne områder for Shaper



5.1.1 Prioritering

Formålet med prioritetssorteringen er at vi skal sørge for at de viktigste pakkene kommer igjennom først. Dette blir viktig for oss siden vi har lagdelt video hvor hvert lag har hver sin prioritet. Vi vil gjerne ha alle 1. prioritets pakkene gjennom, siden de er viktigst for filmen. De andre lagene vil gi en kvalitetsforbedring. Prioriteringen er delt opp i 2 mikromaskiner hvor den første delen identifiserer hvilken prioritet pakken har, og den andre delen tar ut pakkene og sender dem videre til Tx blokken. Det er 4

scratchringer mellom disse 2 mikromaskinene, hvor hver scratchring har sin prioritet. Vi vil alltid ta mest pakker fra de høyeste prioritetskøene, siden de er de viktigste pakkene. Hvis ikke Prioritetsvelgeren greier å ta unna pakkene vil det bli kø på scratchringen. Dette kan skje når vi har mer trafikk enn det utlinjen greier å håndtere. Når scratchringene er full kan vi ikke legge på flere pakker vi må derfor kaste pakkene isteden for. Vi vil jo alltid ta flest pakker ut av de høyeste prioritetene vi vil derfor miste flest pakker fra de laveste prioritetene.

5.1.2 Prioritetssortering

Den første mikromaskinen skal bare hente ut hvilken prioritet pakken skal ha. Vi skiller mellom RTP, RTSP, RTCP og andre pakker. Hvis vi får inn en RTP-pakke sjekker vi om den har et prioritetsfelt. Hvis den har det legger vi pakken inn på den scratchringen den tilhører.

Hvis det er en RTSP pakke legger, vi den i 4. prioritet. Grunnen til at vi legger den på 4. prioritet er fordi hvis det er liten last på proxyen vil ikke dette ha noen effekt. Vi regner med andre ord ikke med å miste noen pakker. Hvis det derimot er mye last på proxyen og vi må kaste en del av 4. prioritets pakkene, vil vi kanskje ikke ha flere videostrømmer igjennom uansett. Vi kunne også ha lagt den som første prioritet. Da ville vi sørget for at de kom igjennom men siden de eneste RTSP-pakkene vi får igjennom vil være enten start av ny videostrøm eller avslutning av en videostrøm, ville det gått på bekostning av andre videostrømmer som gjerne vil ha sine pakker igjennom. Vi har derfor valgt å legge RTSP pakkene til 4. prioritet. RTCP legger vi også som 4. prioritet dette er rapporter til klienten.

Den siste kategorien av pakker er “andre pakker” dette pakker som ikke kommer under en av kategoriene over, det vi si at det ikke tilhører noen videostrøm. Vi vil derfor ikke prioritere disse. Vi vil heller ikke bare kaste dem men la dem gå igjennom hvis det er plass på 4. prioritetskøen.

Gangen i prioritetssorteringen vi da være følgende: Vi henter ut en pakke fra scratchringen fra shaperblokken sjekker hva slags pakke vi har, og etter kriteriene over, legger pakken på den scratchringen den hører til. Når vi legger en pakke på scratchringen oppdaterer vi en variabel som kan sees ved den neste mikromaskinen.

5.1.3 Prioritetsvelger

Den andre mikromaskinen i prioriteringen tar ut pakkene fra de 4 scratchringene og legger dem ut på den porten de skal ut på. Siden det er viktig å alltid få ut de høyest prioriterte pakkene prøver vi å ta flest pakker fra de høyeste prioritetsringene.

Dette gjør mikromaskinen ved at de den sjekker om det er noen pakker på den første scratchringen. Deretter prøver den å ta ut 4 pakker. Hvis det ikke er nok pakker på scratchringen hopper den til neste prioritet. Der sjekker den først om det har kommet inn noen høyere prioritets pakker. Dersom det ikke har det, prøver den å ta ut 3 pakker når den har tatt ut alle, eller at det ikke er fler pakker igjen prøver den å ta ut 2 pakker fra 3. prioritets køen. Deretter prøver den å ta ut 1 fra den køen med lavest prioritet. Vi sjekker alltid før vi tar ut fra en kø om det har kommet inn en viktigere pakke, slik at vi alltid vi får de viktigste pakkene først men uten å sulte ut de andre køene ved at vi bare tar fra de to med høyest prioritet. Vi har også implementer dette slik at vi ved å legge til en linje gjøre at det blir

5.2 Sammendrag

Vi har i dette kapittelet sett hvordan vi har bygd opp systemet vårt. Vi har sett hvordan vi har organisert mikromaskinene på en mest mulig effektiv måte. Vi har kombinert programvareteknikker fra kapittel 2 med NPE fra kapittel 3. Vi håper ved å kombinere disse tingene at vi vil få et system som skal få en ytelsesforbedring. Vi skal i neste kapittel sjekke ut om vi faktisk greier å få avlastet vertsmaskina ved bruk av NPE.

Kapittel 6

Evaluering

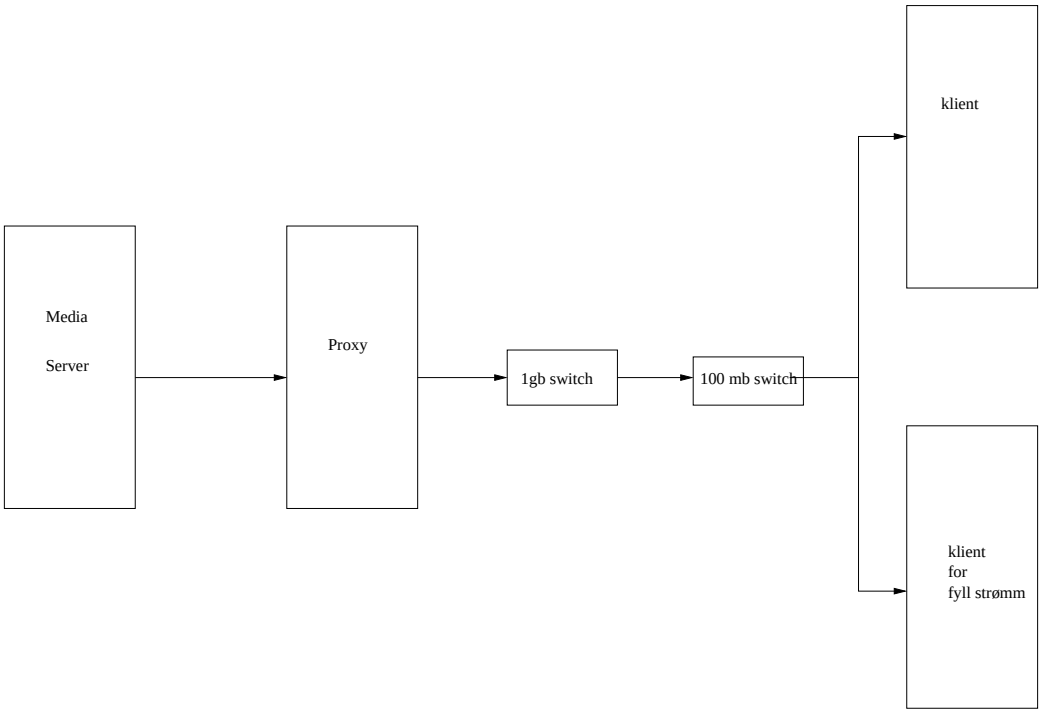
For å sjekke om IXP nettverksprosseseringsenhet (IXP NPE) hjelper til med å avlaste en Linux-vert har vi tenkt å bruke prioritetskøer og shaping av strømmen. Vi vil kjøre tester for å undersøke om IXP NPE vil greie å behandle alle data, og om den vil gi mindre jitter og latens. Vi vil også undersøke om vi vil kunne kjøre flere samtidige strømmer enn en Linux maskin. Vi vil sjekke om det vil bli noen ytelse forskjeller på Linux-verten når vi har et IXP NPE. Vi vil også sjekke om vi kan ha flere samtidige strømmer som gjør at vi ikke trenger så mye hardware eller kan bruke linux verten til andre ting.

6.1 Testoppsett

Når vi har testet prioritetssystemet har vi brukt oppsett fra figur 6.1. Maskinen vi har videotjeneren på er en Pentium 4 som kjører Linux Suse 9.2 den har 1GB ram. Proxy maskinen er når vi kjører Linux-verten en Pentium 4 som kjørte Linux Suse 9.3 med 1GB ram. Når vi hadde IXP NPE brukte vi en Pentium 4 med Linux Suse 9.0 med 512MB ram. Siden vi bruker IXP NPE bruker vi ikke noen komponenter fra vertsmakinen bortsett fra strømmen. Vi kan der for ha 2 forskjellige maskiner som proxymaskin uten at det vil påvirke resultatet. Den maskinen som tar imot er en Pentium 4 som kjører Linux Suse 9.3 den har 512MB ram. Vi ser at vi etter proxyen har først en 1GB switch deretter en 100MB switch, dette er fordi vi vil ha

en linje inn på proxyen på 1000 mbps mens ut fra proxyen vil vi bare ha 100 mbps. Dette får vi med måten vi har satt opp switchene på. Vi må ha begge 2 siden kortet bare greier å kommunisere med enheter som har 1GB grensesnitt. Når vi så har en innstrøm på over 100 mbps vil prioritets køene inne i applikasjonene våres fylles opp og vi må kaste pakker. Til å måle tiden med har vi brukt tcpdump[10]. Siste maskinen i figuren er får å ta unna noe av strømmene som vi bruker som bakgrunnslast.

Figur 6.1 Test oppsett



6.2 Priorities Testing

Når vi skal teste systemet vårt vil vi sjekke om vi vil få bedre ytelse med IXP NPE på selve strømmen og vi vil sjekke om vi vil kunne avlaste linux verten. For å teste dette vil vi sende 2 strømmer gjennom kortet, hvor den første strømmen kun skal sende pakker som bakgrunnslast for å gi litt press på proxyen mens den andre strømmen er en lagdelt mediastrøm som skal prioriteres. Vi måler på mottagersiden tidsdifferansen mellom

hver pakke med samme prioritet. Vi tester her IXP NPE mot det som allerede finnes i dag fra før i Linux kjernen. For å få til dette bruker vi "tc qdisc". Det vi ønsker å sjekke, er om det faktisk blir mindre latens når vi sender strømmen igjennom et IXP NPE, i forhold til å sende en strøm igjennom en vanlig Linux proxy. Tester vil vise om det faktisk vil ha noe å si for ytelsen på mottagersiden når vi har last på proxyen. Når vi testet prioritets implementasjonen brukte vi 4 maskiner, som var satt opp slik figur 6.1 viser. Alle maskinene er satt opp med Linux, og til proxy byttet vi på å bruke IXP NPE og en Linux-vert. Linux-verten var satt opp med qdisc. Qdisc er en kjernemodul som kan sette opp forskjellige skedulig, shapeing og poicing. Den settes opp ved at vi velger hvilken grensesnitt vi vil benytte, deretter kan vi sette opp et antall køer. Vi har satt opp våre køer med prioritets køer. Vi har satt opp qdisc med 4 forskjellig køer hvor hver kø har forskjellig prioritet. Køene vil ta alle pakkene som er på den første køen før den tar noen pakker fra den neste, det blir dermed ikke en rettferdigkø. Proxyen blir også i tillegg belastet ved at vi kjører applikasjoner i bakgrunnen. Dette gjøres for å belaste CPU på proxyen. Etter proxyen kjører vi strømmen igjennom en 1 GB switch deretter en 100 MB switch. Dette er fordi kortet vårt ikke kan snakke med 100 MB grensesnitt. Vi vil også legge press på selve prioritetskøene inn i proxyen. Hvis vi bare har 100 mbps ut av proxyen og vi sender over denne hastigheten inn, vil det danne seg køer på systemet. Vi har derfor alltid med en bakgrunnslast på 100 mbps som vi kjører igjennom systemet vårt på 4. prioritet. Før vi leser av målingene deler vi disse strømmen slik at vi ikke får så mye press på den maskinen som skal lese av målingene. Dette er får å få mer nøyeaktige tider.

Vi har sendt videostrømmen med pakkestørrelse på 1024 KB og med hastigheter 100, 200 og 400 Mbps mens bakgrunnslast strømmen som blir behandlet som 4 prioritet blir sendt på 100 Mbps og har pakke størrelse på 1024 KB. Teoretisk betyr det at vi ikke bør begynne å miste 1. prioritets pakker før vi kommer opp på 400 Mbps. Det kan hende vi mister pakker før siden det er ikke sikkert at 100 MB switchen og nettverkskortet på klienten greier å ta imot 100 Mbps. Vi regner med å miste 4. prioritets pakker med 100 Mbps siden vi da har 25 Mbps 1. 2. og 3. prioritet, men 125 Mbps som skal igjennom for 4. prioritet. Det betyr at vi teoretisk må miste 80 % av pakkene som har 4. prioritet fordi de vil bli fordelt likt på bakgrunnslast strømmen og mediastrømmen. Vi kan miste mer fordi ikke switchen eller kortet greier 100 mbps. Når vi kommer opp på en hastighet på 200 Mbps, vil vi begynne å miste 3. prioritets pakker, siden vi på 1. og 2. prioritet vil ha 100 Mbps som er det mottager maskinen greier å ta i mot.

	IXP	Linux-vert uten last	Linux-vert m/lese last	Linuxvert m/zip last	Linux-vert m/ zip og lese last
maximum	0.856ms	43.511ms	200.008ms	364.494ms	123.908ms
minimum	0.000ms	0.000ms	0.000ms	0.004ms	0.007ms
gjennomsnitt	0.320ms	0.321ms	0.322ms	0.328ms	0.321ms
median	0.351ms	0.349ms	0.348ms	0.348ms	0.348ms
stdev	0.085ms	0.189ms	0.859ms	1.145ms	0.652ms

Tabell 6.1: Latens på 1.prioritets pakker ved en 100 Mpbs strøm

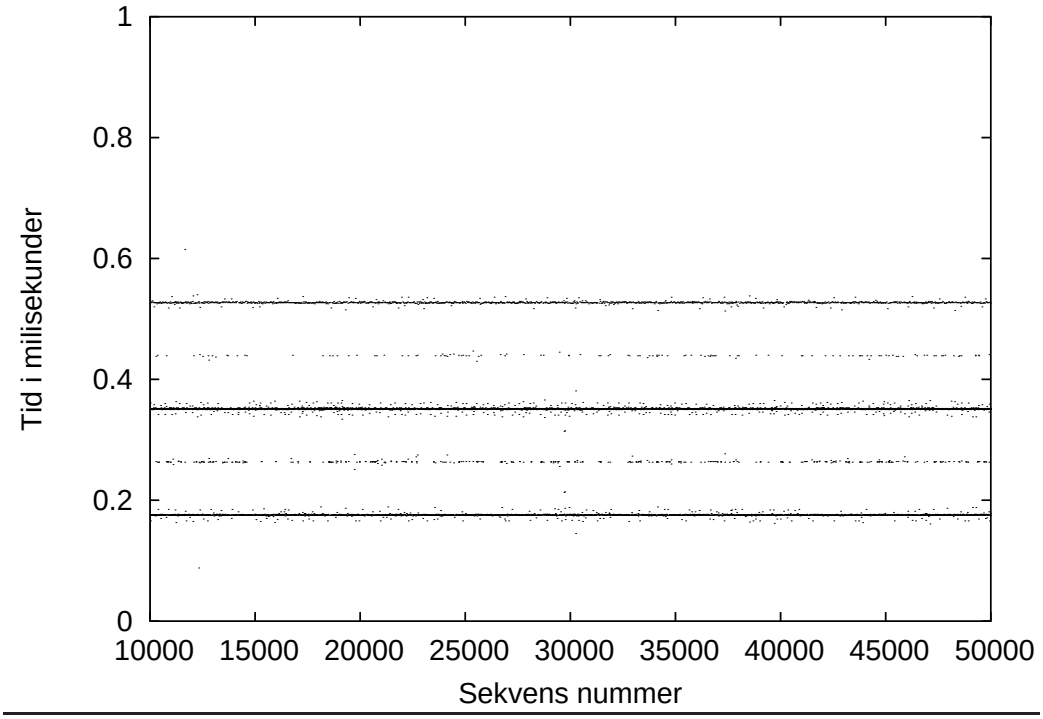
På 400 Mbps vil vi bare ha 1. prioritet igjennom siden vi sender 100 Mbps med 1. prioritets pakker.

Vi begynner med resultatene fra mediastrøm på 100 Mbps hvor vi har 1.prioritetets pakker. Vi ser fra resultatene fra tabell 6.1 at det er rimelig jevne gjennomsnitt verdier mellom Linux-verten og IXP NPE. Den som har høyeste gjennomsnittlig verdien er når proxy kjører operasjoner med gzip load, men det er ikke så veldig mye høyere en de andre. Men vi ser at standard avviket er mindre på IXP NPE enn det er på noen av de kjøringene som er på Linux-verten, max-verdien er også mye høyere. Vi kan av figur 6.2 se plottet over hvordan dette ser ut med IXP NPE. vi kan se Linux-vert uten last i figur 6.3 og Linux-vert med last i figur 6.4. Vi ser at tcpdump[10] som er brukt til å ta ut resultatene ikke har nøyaktig nok klokke til å ta ut alle differansene men vi ser en tydelig trend at det blir mye bedre på IXP NPE i forhold til Linux-vert uten last, og når vi i tillegg legger last på Linux-verten vil det bli enda større forskjeller. Dette er fordi IXP NPE er uavhengig av last på Linux-verten.

Ser vi videre på målingene som er gjort med en 100 Mbps mediastrøm med 2. prioritet, ser vi fra tabell 6.2 at det på IXP NPE er mer likt som 1. prioritet, der standardavviket er blitt noe mindre, og maks verdien er litt høyere. På Linux-verten ser vi at gjennomsnittslatens er blitt høyere og at det er litt dårligere kvalitet en det er for 1 prioritets pakkene. Vi ser her at gzip fortsatt er den operasjonen på Linux-verten som tar mest ressurser. Målingene er ganske like fordi vi ikke sender med stor nok bitrate for å kaste 1.priotitets pakker. Siden det ikke blir noe press på disse pakkene vil de bare sendes igjennom proxyen raskest mulig.

Når vi sender med en raskere båndbredde på 200 Mbps ser vi, fra 1.priori-

Figur 6.2 tidsforskjellen mellom 1. prioritets pakker med ixp



tets pakkenes latens i tabell 6.3 at gjennomsnittlatens går ned i forhold til da vi sendte på 100 Mbps. Dette vil jo være fordi pakkene kommer i tettere rekkefølge når vi sender raskere. Vi registrer at på IXP NPE er det nesten bare å dele gjennomsnitt og median på 2, og at det eneste som er forskjellen på IXP NPE og Linux-verten er standardavviket og maksverdiene. Det er heller ikke noe press her siden vi ikke kaster noen pakker.

Ser vi derimot på 2.prioritet på 200 Mbps fra tabellen 6.4, ser vi at det blir større forskjeller. Vi kan faktisk miste noen pakker her fordi vi sender 50 Mbps 1.prioritet og 50 Mbps 2. prioritet og det er ikke alltid at mottager kortet greier å ta imot 100 Mbps. Vi har fått gjennom litt flere pakker med Linux-verten enn IXP NPE, en mulig grunn for dette er at Linux-verten har større buffer på køene. Dette gjør at når tjeneren er ferdig å sende mediastrømmen, vil Linux-verten sende de pakkene som ligger igjen på køene. Disse pakkene er alt for trege og er verdiløse i forhold til videobruk, men de trekker ned gjennomsnittet for Linux-verten. Vi ser at vi har gjennomsnitt på Linux-verten uten last og IXP NPE er ganske likt mens det går betraktelig oppover når vi belaster Linux-verten. Vi ser at her er det fileread og fileread/zip load som er den tyngste oppgaven.

	IXP	Linux-vert uten last	Linux-vert m/lese last	Linuxvert m/zip last	Linux-vert m/ zip og lese last
maximum	0.966ms	43.250ms	200.007ms	364.494ms	123.734ms
minimum	0.011ms	0.022ms	0.004ms	0.007ms	0.000ms
gjennomsnitt	0.320ms	0.321ms	0.329ms	0.334ms	0.326ms
median	0.351ms	0.348ms	0.348ms	0.348ms	0.348ms
stdev	0.069ms	0.182ms	0.864ms	1.152ms	0.652ms

Tabell 6.2: Latens på 2.prioritets pakker ved en 100 Mpbs strøm

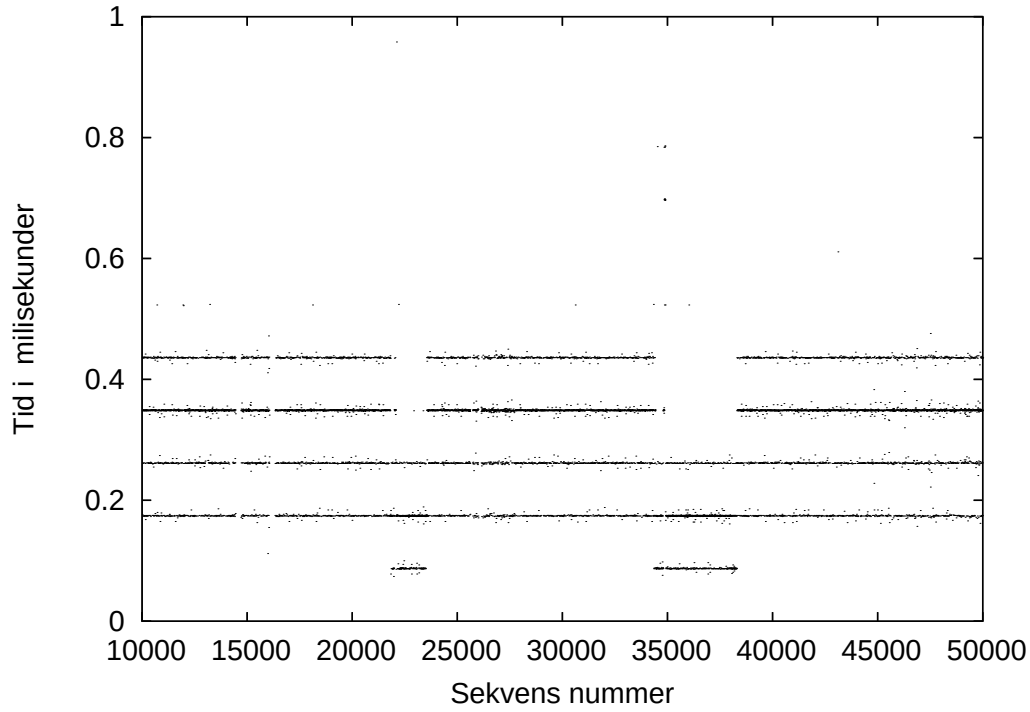
	IXP	Linux-vert uten last	Linux-vert m/lese last	Linuxvert m/zip last	Linux-vert m/ zip og lese last
maximum	16.721ms	42.592ms	67.218ms	54.813ms	50.122ms
minimum	0.000ms	0.003ms	0.003ms	0.003ms	0.000ms
gjennomsnitt	0.160ms	0.160ms	0.160ms	0.160ms	0.160ms
median	0.176ms	0.174ms	0.174ms	0.174ms	0.174ms
stdev	0.053ms	0.155ms	0.467ms	0.456ms	0.465ms

Tabell 6.3: Latens på 1.prioritets pakker ved en 200 Mpbs strøm

	IXP	Linux-vert uten last	Linux-vert m/lese last	Linuxvert m/zip last	Linux-vert m/ zip og lese last
maximum	16.720m	42.243m	67.217m	54.813m	50.122m
minimum	0.007m	0.000m	0.003m	0.003m	0.003m
gjennomsnitt	0.195m	0.193m	0.234m	0.224m	0.234m
median	0.176m	0.175m	0.175m	0.175m	0.175m
stdev	0.071m	0.157m	0.560m	0.539m	0.557m

Tabell 6.4: Latens på 2.prioritets pakker ved en 200 Mpbs strøm

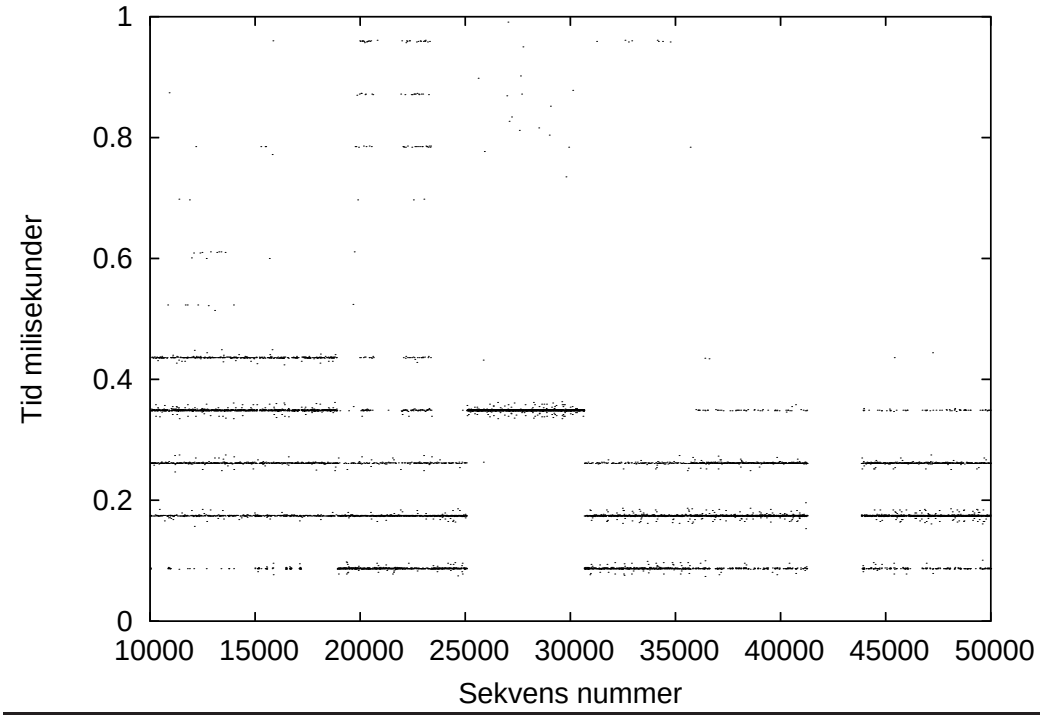
Figur 6.3 tidsforskjellen mellom 1. prioritets pakker med Linux uten last



Når vi sender en strøm på 400 Mbps og det er ikke rettferdig køing, kan vi forvente at det bare kommer igjennom 1. prioritets pakker. Vi tar derfor bare med målingene for 1. prioritetspakkene. Det ble droppet en god del 1. prioritets pakker på Linux-verten og det kom en del 2. prioritets pakker igjennom. Dette kan være fordi Linux-verten ikke greier å prosessere den fulle bitraten av 1. prioritets pakker og må kaste noen av pakkene. Vi ser at også noen pakker fra 2. 3. og 4. prioritet kommer igjennom, grunnen til dette er som nevnt over at Linux-verten har lengre prioritetskøer. Når vi ser på resultatene på figur 6.5 ser vi at IXP NPE har et betydelig bedre gjennomsnitt og det vil sannsynligvis stige med Linux-verten etter hvor mye last den har. Vi ser også at gjennomsittslatens ikke er så forskjellig som vi ville trodd på forhånd. Dette kan komme av at Linux qdisc er en kjernemodul og kjører på kjernen. Vi har lastet proxyen vår med bruker prosesser som ikke er så høyt prioritert. Vi ser fortsatt at det blir utslag på tidene, som viser at IXP NPE er bedre en Linux-verten på alle gjennomsnittslatensene og standard avvik..

Ser vi derimot på tiden det tar å gjøre ting på Linux-verten når vi bruker maskinen til en proxy med prioriteringskø. Vi ser fra tabell 6.6 at det vil ta

Figur 6.4 tidsforskjellen mellom 1. prioritets pakker med Linux uten last



litt lengre tid. Tiden som er brukt i sys og user er litt større men hvis vi ser på tiden som blir brukt i real så er det 11 sekunder i snitt på hver av tar prosessene. Dette er en ganske stor forskjell. Mesteparten av tiden som er brukt er den sammenlagte tiden. System og user tiden er det små forskjeller. Dette kan bety at mye av tiden som brukes går med på kontektbytter og shedrulering. Vi har kjørt den samme mediastrømmen gjennom i denne testen som vi har gjort i testene over, og brukerprosessen vi kjører er et program som kjører en tar prosess med zip 300 ganger. Vi kan se dataene på figur 6.5. Vi ser at gjennomsnittsverdien er over 11 sekunder større, og standardavviket er nesten dobbelt så stort. Dette betyr at vi frigjør masse ressurser ved å benytte oss av et IXP NPE .

I tabell 6.7 ser vi målinger som er gjort uten at vi har prioritert strømmen. Vi ser her at alle lagene er like prioriterte og det er det vil ville ha forventet på forhånd. Dette vil si at det lønner seg å prioritere isteden fordi vi da vil få flere av de pakkene som er viktige i forhold til de som vi rangerer som mindre viktige.

Når vi så slår sammen alle tabellene i en figur 6.6 som bruker gjennom-

	IXP	Linux-vert uten last	Linux-vert m/lese last	Linuxvert m/zip last	Linux-vert m/ zip og lese last
maximum	20.390m	56.459m	829.084m	49.370m	662.014m
minimum	0.000m	0.003m	0.000m	0.003m	0.003m
gjennomsnitt	0.129m	0.139m	0.159m	0.154m	0.154m
median	0.088m	0.089m	0.174m	0.174m	0.174m
stdev	0.071m	0.153m	1.552m	0.427m	1.494m

Tabell 6.5: Latens på 1.prioritets pakker ved en 400 Mpbs strøm

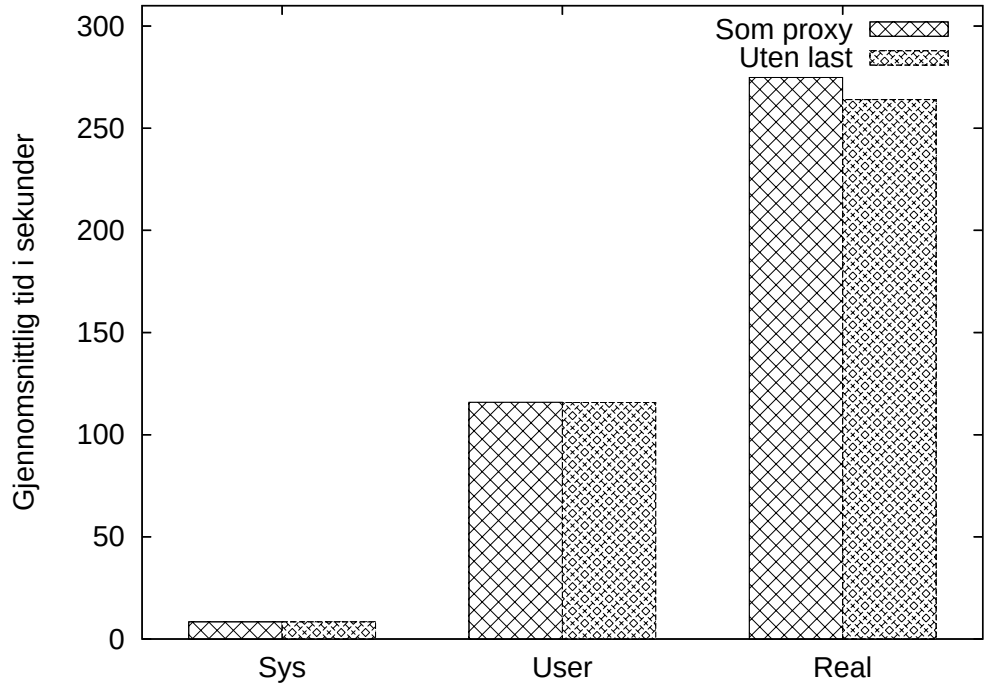
	Uten mediastrøm			Med mediastrøm		
	sys	user	real	sys	user	real
gjennomsnitt	8.361s	115.737s	263.997s	8.393s	115.899s	274.867s
median	8.350s	115.730s	262.180s	8.370s	115.860s	270.590s
stdev	0.216s	0.262s	11.633s	0.225s	0.295s	26.446s

Tabell 6.6: Tidsmåling med en tar prosess

	1.prio	2.prio	3.prio	4.prio
maximum	682.530m	682.010m	682.185m	683.942m
minimum	0.000m	0.004m	0.004m	0.003m
gjennomsnitt	0.569m	0.573m	0.578m	0.580m
median	0.266m	0.276m	0.346m	0.350m
stdev	4.720m	4.735m	4.757m	4.770m

Tabell 6.7: Måling med 200 uten prioriteter på ixp

Figur 6.5 Gjennomsnittlige verdier med tar

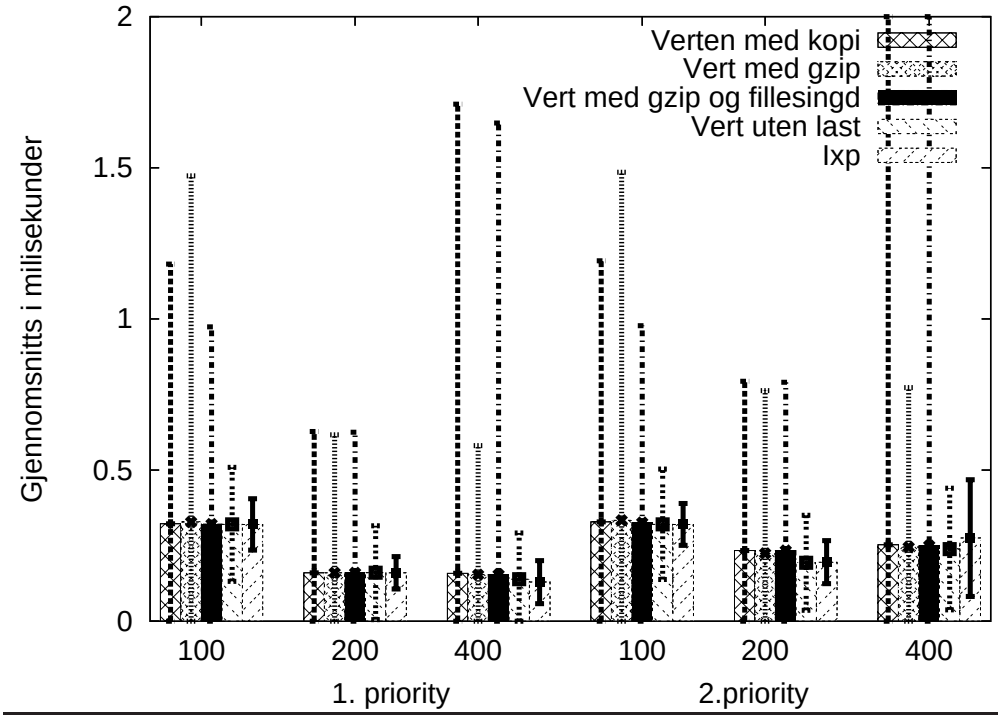


snittsverdien med feillinjer ser vi at IXP NPE kommer best ut i stort sett alle tilfeller. Vi ser at gjennomsnittlig så er forskjellen mellom IXP NPE og Linux verten ganske lik på 1.prioritets pakkene, bortsett fra 400 Mbps der er IXP NPE en del bedre. Ser vi derimot på avvikslinjene ser vi at det er IXP NPE er bedre på alle områder. Vi ser også klart at hvis vi belaster Linux-verten vil dette ha veldig mye å si for standardavviket. Det er bare 2. prioritets strømmen på 400 Mbps som kommer dårligere ut. Som nevnt over er det fordi Linux qdisc har lengre prioritetskøer slik at de bufrer opp en del pakker som kommer etter at strømmen er ferdig. Dette skyldes at Linux-verten har satt lengre køer til sin prioritets kø, og at Linux-verten kaster mer 1. prioritets pakker slik at den får flere 2. prioritets pakker igjennom.

6.3 Shaper Testing

Når vi skal teste shaperen skal vi teste på 2 forskjellige måter. Vi skal teste når vi sender med en variabel båndbredde og kjøre strømmen igjennom

Figur 6.6 Gjennomsnittlige verdier med standard avvik

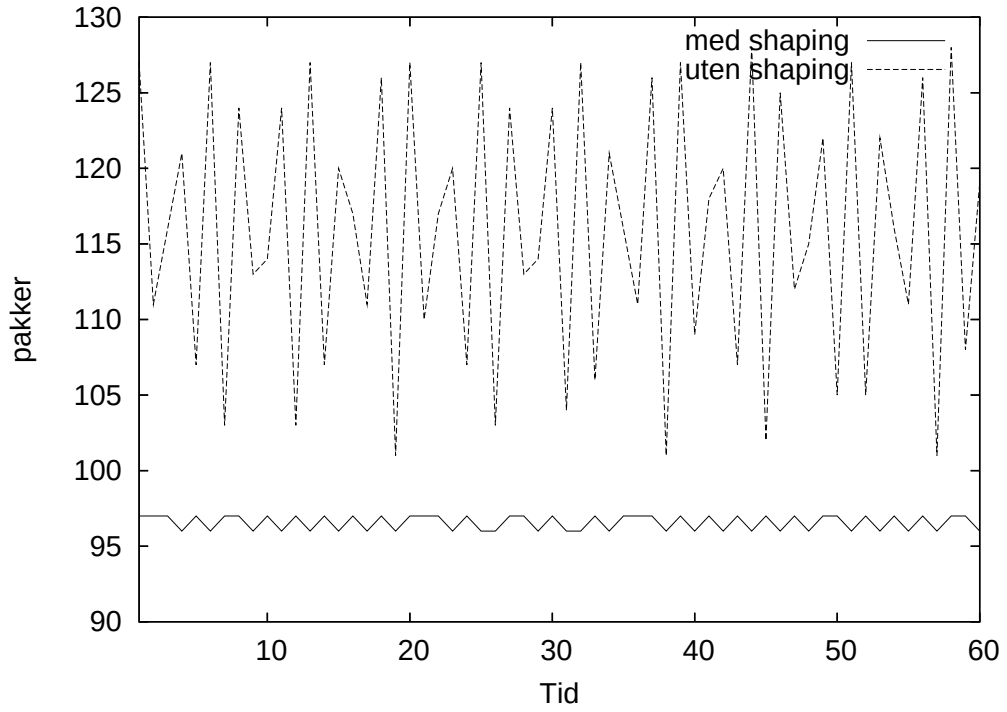


kortet. Da forventer vi at shaperen skal jevne ut strømmen slik at det blir mer jevnt på mottager siden. Vi vil også teste på å sende en høy strøm og shape den på en mindre båndbredde slik at vi vil kutte ned på båndbredden til strømmen. Vi har nå testet med flere typer strømmer. Vi har satt opp slik at vi sender med 100 pakker på 1100 Kbps og deretter 100 pakker på bandbredde 800, 700 og 600 Kbps. Pakkene er alle sammen 1024 KB store og vi shaper strømmen på 800 Kbps.

Vi ser fra figur 6.7 at vi når vi shaper strømmen vil vi ha en rimelig rett linje som varierer mellom 98 og 97 pakker pr sekund. Når vi ikke shaper strømmen vil den variere mellom 103 og 125. Dette viser tydelig at shaperen fungerer. Vi vil være nødt til å miste noen pakker på grunn av shapingen.

Når vi senker sendings variansen fra 800 kbps til 700 kbps ser vi fra figur 6.8 at den shapede strømmen ikke har forandret seg stort, mens den ushapede strømmen har mye større varians. Den vil av og til gå under 800 Kbps, som er hva vi shaper på. Dette er fordi vi bare sender den strømmen som kommer igjennom, og noen ganger er senderaten under 800 kbps.

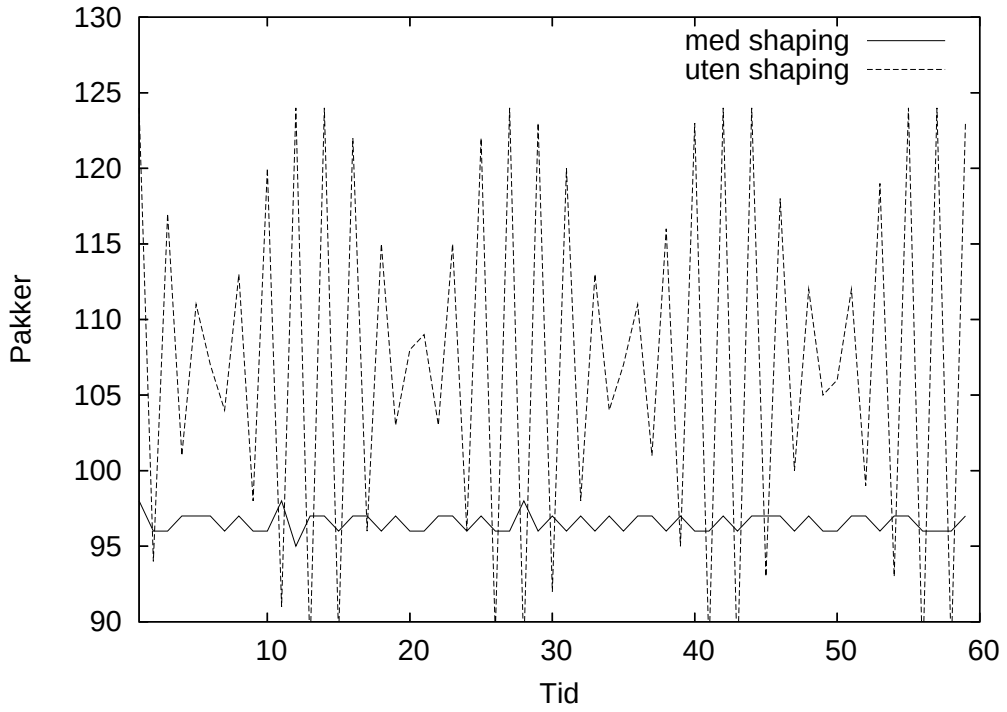
Figur 6.7 1100 og 800 kbps



Grunnen til at den shapede strømmen ikke forandrer seg mye er fordi vi har en buffer som lagrer pakker og som kan sende dem når det blir ledig i forhold til hva shaper er konfigurert til. Vi ser at dette vil fungere fint ved at vi jevner ut strømmen, og vi ikke har det store fallet som den ushapede strømmen har.

Når vi tester med strømmen som varierer fra 1100 Kbps til 600 Kbps, ser vi fra figur 6.9 at den shapede strømmen begynner å variere litt mer. Grunnen til at den går litt over 800 kbps på noen felter her er at vi vil ha spart opp en del billetter når vi har fått de 100 pakkene på 600 Kbps og når vi da begynner å få pakker på 1100 Kbps vil vi da kunne sende litt over før vi vil gå tilbake til 800 Kbps. Hvis vi hadde hatt et større buffer, ville vi hatt en mer jevn strøm på den shapede strømmen enn hva vi ser nå, men vi vil ikke samle på mengder av pakker som er for gamle siden disse vil bli kastet av klienten uansett. Det er derfor mer ressursvennlig å ha et passende buffer både i forhold til at vi sparer minne og linjeresursser som kunne ha blitt brukt på pakker som er relevante for strømmen. Vi ser her også at siden vi ikke har noen tak på antall billetter som blir tildelt at vi også får en

Figur 6.8 1100 og 700 kbps



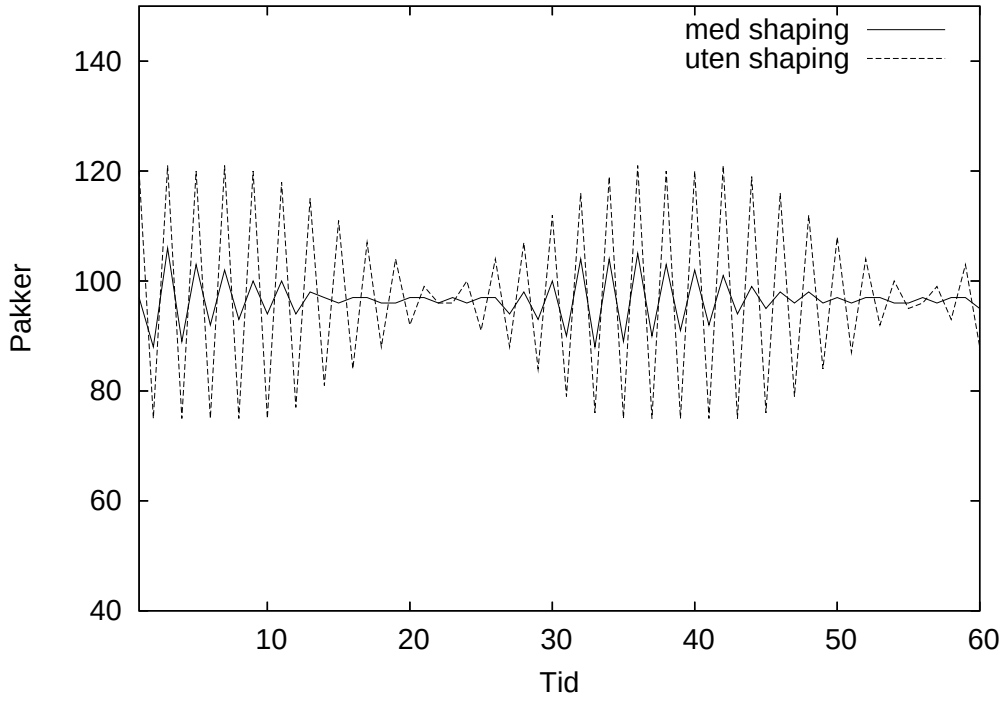
liten på når vi shaper strømmen også. Men den blir ikke alt for stor hadde vi hatt et tak ville den holdt seg rundt den bitraten strømmen egentlig skulle hatt.

6.4 Tidsmåling

Når vi skal måle tiden en pakke går igjennom systemet

Til å teste programmet mitt har vi brukt et program som følger med kortet som kalles for workbench. Det er et program som man loader koden inn i mikromaskinene og har oversikt over diverse minne områder eller register som blir brukt av mikromaskinene. Det er jo forskjellige register for hver av mikromaskinene. Det går også an å stoppe på kjøringen på bestemte områder slik at man kan se hva som ligger i minnet, eller registrert til enhver tid. Siden det ellers er veldig vanskelig å debugge på mikromaskine-

Figur 6.9 1100 og 600 kbps



ne viser dette programmet til å være veldig nyttig, siden den eneste andre måten er å legge verdier i minnet for å så bruke XScale til å lese ut verdier.

Ethereal er også et program som er greit siden man da kan se hva slags verdier man får inn og ut. Kombinert vil dette funke veldig bra siden man da kan sjekke minne området til pakken og se hva som ligger i pakken i bestemte områder og dermed finne hvor man skal lete mer etter feilen.

6.5 Sammendrag

Vi ser fra de testene vi har utført at priotitetskøen ville blitt bedre på et IXP NPE enn det ville ha blitt på en Linux-vert siden testene viser at vi minsker jitter på mottager som vist på figurene og tabellene. Med mindre jitter vil vi ikke bare bruke dette til prioritetskøer og shaping for lagdelt video, men også til andre applikasjoner som trenger lavere jitter. Dette kan være alt fra

spillapplikasjoner og andre jittersensitive applikasjoner. Vi har også vist at IXP NPE vil avlaste maskinen. Når en tar prosess vil ta over 10 sekunder lengre tid på en Linux-vert når den er satt opp som en proxy, kan det bety at det er mye å spare med avlasting ved hjelp at IXP NPE. Så når testene viser at vi minsker jitter og at vi avlaster vertsmaskinen slik at vi kan gjøre andre ting vil det være bra ting å bruke.

Shaperen på måten vi har implementert den her vil også gjøre at vi har mer kontroll over strømmen vi sender, ikke bare alle strømmene sammenlagt, som den ville ha gjort hvis vi ville hatt den som en kjerneprossess i Linux-verten. Dette medfører at vi ville ha en mye større kontroll over hver enkelt strøm og vi har vist i tester at den vil fungere på den måten den er tenkt å fungere. Med shaperen som den er satt opp her vil proxyen selv greie å bestemme om den kan ta imot en ny forespørsel eller om den har så mye å gjøre at det ikke ville være nok båndbredde til å gi tilfredsstillende kvalitet på videoen.

Kapittel 7

konklusjon

Vi skal nå konkludere vårt arbeid. Vi har sett på om vi vil få bedre ytelse ved å bruke et IXP NPE enn ved å bruke Linux maskin. IXP NPE er en IXP2400 nettverksprossessor fra intel.

7.1 Resultat

Vi har i denne oppgaven sett på forskjellige måter å som skal bedre ytelsen og avlaste Linux-verten på . Vi har brukt IXP NPE som en proxy som skal stå mellom en tjener og klient. Vi har spesielt sett på strømmer med forskjellige prioriteter. Vi har sett på hvordan systemet oppfører seg når den må kaste pakker. Vi har da sørget får å få gjennom de høyestprioriterte pakkene. Vi har også sørget for at strømmene ikke ødelegger for hverandre med burst. Vi har lagt denne funksjonaliteten på IXP NPE. Vi har deretter testet med tilsvarende funksjonalitet i Linux kjernen.

Vi har sett på hvordan vi bedre kan kvalitetssikre strømmer ved å benytte oss av metoder som shaping og prioritetskøing. Disse metodene har vi Implementert på et IXP NPE for å se om de vil yte bedre enn på en vanlig Linux maskin. Vi har sett om vi ved bruk av IXP NPE vil avlaste en Linux-vert. Vi har også implementert ressurs-resservering Utfra RTSP med SDP informasjonen. Denne er transparent for tjeneren og klient. Dette hjelper oss med passe på strømmene som går gjennom IXP NPE slik at de ikke

stjeler resurser fra hverandre. Vi vil med denne metoden slippe at en strøm som kan ødelegge for alle de andre strømmene.

Etter vi har kjørt tester har vi sett at IXP NPE vil avlaste Linux-verten. Vi bruker betydelige mer resursers når vi ikke avlaster med IXP NPE. Disse resursene er prosessor tid og kontekst bytter som vi kan bruke til andre ting. Vi har også sett at vi får lavere i gjennomsnitts latens tider når vi kjører igjennom en IXP NPE i forhold til å kjøre på en vanlig Linux maskin. Det betyr at vi kan benytte oss av IXP NPE når vil avlaste Linux-vertsmaskinen.

Vi har også sett at vi vil få mindre jitter når vi bruker IXP NPE. Det betyr at vi vil ha mindre variasjon i behandlings tiden av pakken på IXP NPE enn det er på Linux maskinen. Med mindre jitter vil vi få informasjonen omtrent like raskt. Dette gjør at slipper å jevne ut hastighetsforskjellene på klienten og trenger dermed mindre buffer på mottakersiden.

7.2 fremtidig arbeid

For videre arbeid Utfra hva vi har gjort i oppgaven vil være å utvide til et mer fleksibelt adgangskontroll system. Vi kan da nekte nye strømmer tilgang gjennom proxy. Dette vil sikre de strømmene som allerede er satt opp på kortet. Vi kan også forandre litt på måten vi har organiserer structene til strømmene i shaper blokken. Slik vi har organisert det nå har vi en lenketliste. En smartere måte å gjøre det på er å bruke en hash tabell. Dette vil gjøre at vi ikke bruker så lang tid hvis det er mange strømmer gjennom proxyen. En siste utvidelse vil være å ha en kommunikasjon mellom shaperen og prioritetskøen slik at vi kan få flere token hvis det ikke er noe press på linjen. Vi kan også se litt mer på hvilke pakker som shaper kaster og sørge for at vi ikke ville kaste de viktigste pakkene. Vi kan også sette et tak for hvor mange billetter vi vil ha tilgjenglig slik at vi ikke kan spare masse billetter. Vi kan også implementere forskjellige shaping og skedulings algoritmer slik at vi på en lett måte kan bytte mellom dem. Dette vil føre til at vi kan få større fleksibilitet.

De komponentene som vi har utviklet er ganske generelle så vi kan ganske lett utvidet dem med mer funksjonalitet. Vi kan uten store problemer forandre algoritmene som styrer prioriteten slik at vi kan dekke andre

bruksområder. Det kunne videre vært spennendes å utvide systemet slik at vi kan få en nettverksemulator. Problemet med å ha nettverksemulator på Linux er at kjernen vil alltid miste litt pakker hvis nettverkskortet blir litt presset. Vi vil også ha problemer med at jitter at det blir unøyaktige resultater fordi pakkene blir ujevnt behandlet. Det problemet har vi ikke på IXP NPE. Det som trengs for å utvide systemet til en nettverksemulator er et system som kaster en bestemt mengde pakker. Det trenges også en mekaniske for å forsinke pakkene. For å forsinke pakkene trenger vi å buffere ett antall pakker. Det kan vi ha gjøre ved at vi køer dem på en scratchring og tar dem ut etter hvor lenge de skulle vært forsinket.

Bibliografi

[1] Anarchy online. http://www.anarchyonline.com.

[2] Intel ixp2400 hardware reference manual.

[3] Intel ixp2400 hardware reference manual.

[4] The monta vista website. http://www.mvista.com.

[5] Radisys enp sdk 3.5 programmes guide.

[6] Radisys enp2611 hardware reference manual, November 2005. http://www.radisys.com/files/supportdownload/007-01419-0003-ENP-2611.pdf.

[7] The intel network processors family website, january 2006. http://www.intel.com/design/network/product/npfamily/index.htm.

[8] Norsk riks kingkasting, januar 2006. http://www.nrk.no.

[9] Tandberg website, january 2006. http://www.tanberg.net.

[10] The tcpdump/libcap project homepage, january 2006. http://www.tcpdump.org/.

[11] Douglas E. Comer. *Network System Design*. Person Prentice Hall, 2006.

[12] S Rooney D.Bauer and P.SCotton. Network infrastructure for massivly distributed games. In *In Proceedings of the Workshop an System Support for Games*, pages 36–43, Germeny Apr 2002.

[13] Derek Eager, Mary Vernon, and John Zahorjan. Optimal and efficient merging schedules for video-on-demand servers. In *MULTIMEDIA '99: Proceedings of the seventh ACM international conference on Multimedia (Part 1)*, pages 199–202, New York, NY, USA, 1999. ACM Press.

[14] Derek L. Eager and Mary K. Vernon. Dynamic skyscraper broadcasts for video-on-demand. In *MIS '98: Proceedings of the 4th International Workshop on Advances in Multimedia Information Systems*, pages 18–32, London, UK, 1998. Springer-Verlag.

[15] Leana Golubchik, John C. S. Lui, and Richard Muntz. Reducing i/o demand in video-on-demand storage servers. In *SIGMETRICS '95/PERFORMANCE '95: Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 25–36, New York, NY, USA, 1995. ACM Press.

[16] Pål Halvorsen. *Improving I/O performance of multimedia Servers*. PhD thesis, Thesis for the Dr. Scient at University of Oslo, Oslo Norway 2001.

[17] M. Handley and V. Jacobson. Sdp: Session description protocol, rfc 2327, 1998.

[18] A. Rao H.Sculzerinne and R. Lanphier. Real time streaming protocol(rtsp),rfc 2326, 1998.

[19] V. jacobson H.Sculzerinne, S. Casner and R. Frederickkey. Rtp: A transport protocol for real-time applications, rfc 3550, 2003.

[20] Kien A. Hua, Ying Cai, and Simon Sheu. Patching: a multicast technique for true video-on-demand services. In *MULTIMEDIA '98: Proceedings of the sixth ACM international conference on Multimedia*, pages 191–200, New York, NY, USA, 1998. ACM Press.

[21] V. jacobson K. Nichols and K. Poduri. An expedited forwarding phb, rfc 2598, 1999.

[22] A. Luczak M. Domanski and S. Mackowiak. Spatial- temporal scalability fo mpeg video coding. *IEEE Trans. Circuts Syst vol 10*, oct 2000.

[23] Andreas Petlund. Investegating the distribution of functionality for building server hypercube with 2400 cards. Master's thesis, Universitetet i Oslo, 2005.

[24] Carsten Griwodz Pål Halvorsen. Inf5060 multimediakommunikasjon med nettverks prossesorer ifi uio.

[25] Carsten Griwodz Pål Halvorsen. Inf5070 media storage and distibution systems ifi uio.

[26] Li Weiping. Overview of fine granularity scalebility in mpeg-4 video standard. *IEEE Transactions on circuts and system for video technology*, mars 2001.

[27] J. Heinanen F. Baker W. Weiss and J. Wroclawski. Assured forward phb group ,rfc 2597, 1999.

[28] Hui Zhang and Domenico Ferrari. Rate-controlled static-priority queueing. *Computer Science Division*, 1993.

[29] P Halvorsen Ø.Hvamstad, C Griwodz. Offloading multimedia proxies using network processors. In *In Proceedings of the International Network Conference(INC)*, pages 113–120, Samos Island, Greece July 2005.

Tillegg A

Kode for de forskjellige mikromaskinene

A.1 shaper.c

```
#include <ixp.h>
#include <dl_system2.h>
#include <dl_buf.c>
#include <dl_meta.c>
#include <hardware.h>
#include <sig_functions.c>
#include <ixp_lib.h>
#include <ixp_crc.c>
#include <rtl.c>
#include <ixp_cycle.c>
#include <libc.c>
#include <string.h>


typedef __declspec(dram) struct stream_holder *streamptr;
typedef __declspec(dram) struct
{
union
{
struct
{
unsigned int ipdest;
unsigned int token;

unsigned int head;
unsigned int tail;

unsigned int port;
unsigned long long bandwith;

unsigned long long packet_size;
unsigned long long time;

unsigned long long diff;
unsigned int qpacket;

unsigned int packet_sent;
unsigned int packets_waiting[20];

streamptr next;
```

```
streamptr prev;
};
unsigned int value[33];
};
} stream_holder;

__declspec(dram) stream_holder *head;
__declspec(dram) stream_holder *tail;
__declspec(dram) stream_holder *sq;
__declspec(sram_read_reg) unsigned int  rx_msg[6];
__declspec(sram_write_reg) unsigned int txReq;
__declspec(sdram) unsigned char *p_pkt_hdr;

SIGNAL sig_scr_put;
SIGNAL system_init_done;
SIGNAL sig_scr_get_rx;

unsigned int  cseq, sdp;
unsigned int  high, highint;
__declspec(gp_reg) unsigned long long
 timestamp;
int stream_start;
U32 prio;
U8 type;
U8 rtcp;

/*noen faste variabler for uttak av pakken */
#define PRIORITY     74
#define TYPE         23
#define SSID     50
#define SDP_LOC                 117
#define SETUP_LOC    66
#define PORT_LOC   175
#define UDP   0x11
#define TCP   0x06
#define RTCP   0xc8
#define SDP             0x2f736470
#define SET                                0x53455455
#define TEST             0x74657374

extern dl_buf_handle_t  dlBufHandle; // The current buffer handle.


////////////////////////////////////////////////////////////////////////////////
//
// get_time()
//
//   Description:
//
// Demonstrate the timestamp capability. Gets the time and from the system.
//
//   Outputs:
//
// Modifies a global variable 'timestamp' which is a unsigned long long (64bits).
//
// Inputs
//
// None.
//
// CONSTANTS
//
//
// Size:
// N/A
////////////////////////////////////////////////////////////////////////////////
unsigned long long get_time(void){

  unsigned int low, high;

  low = (unsigned long long)(local_csr_read (local_csr_timestamp_low)) ;
  high = (unsigned long long)(local_csr_read (local_csr_timestamp_high));

  timestamp =  (unsigned long long)(low) | (unsigned long long)(high) << 32;

  sq->packet_size = timestamp;
  // returns milisec
  return (timestamp ) ; //& 0xffffffff);

}

INLINE void token_init(){

  int i;
```

```
  for(i=0;i<37 ;i++)
    sq->value[i] = 0;
}


INLINE void put_ring(int ring,SIGNAL *sig){
  scratch_put_ring(&txReq,
   (void*)(ring << 2),
   1,
   sig_done,
   sig);

  wait_for_all(sig);
}
INLINE void find_adress(){

  __declspec(dram) char * indexptr;
  char tabel_entry;
  int i =  0;

  indexptr = (__declspec(dram) char *) STREAM_TABLE_BASE;

  while(1){

    tabel_entry = ua_get_u8(indexptr,i);
    if(tabel_entry == 0) {//(0x8 & (int) indexptr) >> 3 == 0)
      sq = (__declspec(dram)stream_holder *) (STREAM_TABLE_BASE
      + STREAM_TABLE_INDEX_SIZE
      + (i * STREAM_TABLE_ENTRY_SIZE));
      token_init();
      ua_set_8_dram(indexptr,i,1);
      break;
    }
    i++;
    indexptr = indexptr;
  }
}

INLINE void free_stream(int ipdest){
  __declspec(dram) char * indexptr;
  stream_holder *tmp;
  int addr, i;

  indexptr = (__declspec(dram) char *) STREAM_TABLE_BASE;



  tmp = head;

  while(tmp != NULL){
    if ( tmp->ipdest == ipdest){
      addr = (int)tmp;
      i = ((addr - STREAM_TABLE_BASE - STREAM_TABLE_INDEX_SIZE) / STREAM_TABLE_ENTRY_SIZE);
      ua_set_8_dram(indexptr,i,0);
      break;
    }
    tmp = (__declspec(dram) stream_holder *)tmp->next;
  }
}


void find(unsigned int ipaddr, unsigned int port){

  sq = head;

  while(sq != NULL){
    if ( sq->ipdest == ipaddr & sq->port == port){

      break;
    }
    sq = (__declspec(dram) stream_holder *)sq->next;
  }
}

INLINE void update_token(){

  if(sq->time == 0){
    sq->bandwith = (U64)((U64)(sq->bandwith) / (U64)(sq->packet_size));
    sq->time = get_time();
    sq->token++;
    stream_start = 1;
```

```
  }
  else{
    //find out if the time is right to give more tokens
    sq->diff = get_time()- (U64)(sq->time);

    /* we take how many packet's that are availble on this time - packet sent to find how
       many free token we have */
    sq->token = (unsigned int)((U64)(sq->diff) * (U64)(sq->bandwith) * 0x72 >> 32);
  }
}

INLINE void send_prev_packet(){
  dlBufHandle.value = sq->packets_waiting[sq->head];
  txReq = sq->packets_waiting[sq->head];
  prio = ((sq->packets_waiting[sq->head] & 0x70000000) >> 28) ;

  if( sq->head == 19)
    sq->head = 0;
  else
    sq->head++;

  sq->qpacket--;

}

INLINE void wait_for_tokens(){
  if( sq->qpacket < 20){
    sq->qpacket++;

  }
  else{
    dlBufHandle.value = sq->packets_waiting[sq->head];
    Dl_BufDrop(dlBufHandle);
    if( sq->head == 19)
      sq->head = 0;
    else
      sq->head++;
  }

  /*fra 27 -> 31 skal jeg putte på prioriteten  slik at jeg har den når jeg tar ut fra vente kø*/
  sq->packets_waiting[sq->tail] = (rx_msg[0] & 0x00FFFFFF)
    |(dlMeta.inputPort << 24)
    | (prio << 28)
    |(1 << 31);

  if(sq->tail == 19)
    sq->tail = 0;
  else
    sq->tail++;

}

INLINE int token_bucket(){
  if( NULL != sq){

    update_token();

    if ( /*100000000*/sq->token < sq->packet_sent){
      wait_for_tokens();
      return 0;
    }
    else{
      sq->packet_sent++;
      return 1;
    }
  }
  return 2;
}

INLINE int check_qpackets(int new){



  stream_holder *tmp;

  sq = head;
  while(sq != NULL){
    if ( sq->qpacket > 0){

      update_token();
      if (sq->token > sq->packet_sent){
sq->packet_sent++;
send_prev_packet();
```

```
return 1;
      }
    }
    sq = (__declspec(dram) stream_holder *)sq->next;

  }
  return 0;

}
main()
{
SIGNAL_PAIR dram_sig;
__declspec(sram) dl_meta_t *pMeta;
__declspec(dram_write_reg) unsigned int stream_null[STREAM_TABLE_INDEX_SIZE/4];
int i, send_prev, test, nexttest;
U32 ipdest, port, setup;
U8 fin;

__declspec(dram) char* start;
__declspec(dram) char* end;
__declspec(dram) char* end1;


if (ctx() == 0) {

  for(i = 0;i<STREAM_TABLE_INDEX_SIZE/4;i++)
    stream_null[i] = 0;


  dram_write(stream_null,(volatile void __declspec(dram) *)STREAM_TABLE_BASE, (STREAM_TABLE_INDEX_SIZE/8), sig_done,&dram_sig);
  wait_for_all(&dram_sig);


  head = NULL;
  tail = NULL;
  sq = NULL;
  stream_start = 0;

  __assign_relative_register(&system_init_done, ME_INIT_SIGNAL);



#ifdef WAIT_FOR_COMMON_RESOURCE_INITIALIZATION
  wait_for_all(&system_init_done);

#endif

  while(1){
start:
    scratch_get_ring(rx_msg,
     (void*)(POS_RX_RING_OUT << 2),
     5,
     sig_done,
     &sig_scr_get_rx);

    wait_for_all(&sig_scr_get_rx);

    if (rx_msg[0]){


      dlBufHandle.value = rx_msg[0];

      pMeta = (__declspec(sram) dl_meta_t *)Dl_BufGetDesc(dlBufHandle);


      p_pkt_hdr = (__declspec(sdram) unsigned char *) (Dl_BufGetData(dlBufHandle) +
       pMeta->offset);


      /* Om det er Tcp eller Udp pakke vi jobber med*/
      type = ua_get_u8(p_pkt_hdr ,TYPE);

      /*finner ipdestinasjon*/
      ipdest = ua_get_u32(p_pkt_hdr, 14 + 16);
      port = ua_get_u16(p_pkt_hdr, 36);

      sdp = ua_get_u32(p_pkt_hdr,SDP_LOC);
      setup = ua_get_u32(p_pkt_hdr,SETUP_LOC);
      if (setup == SET){
start = p_pkt_hdr + 175;
sq->port = strtol_sdram(start,0x2d,0);
      }
      if (sdp == SDP){
```

```
find_adress();


/* put the new stream in the linked list*/
if(NULL == head){

  head = sq;
  tail = sq;
}
else{
  (__declspec(dram) stream_holder *)sq->prev = tail;
  (__declspec(dram) stream_holder *)tail->next = sq;

  tail = sq;
}
sq->ipdest = ipdest;

start = p_pkt_hdr + 257;
end = (__declspec(sdram) char *)0xa0;
sq->bandwith =  (strtol_sdram(start, *end,0) * 1001)/8;
start = p_pkt_hdr + 365;
sq->packet_size = strtol_sdram(start, *end,0);
      }


      txReq = (rx_msg[0] & 0x00FFFFFF) | (pMeta->inputPort << 24) |(1 << 31);
      if(type == UDP & sdp != SDP){

nexttest = ua_get_u32(p_pkt_hdr,42);
if (nexttest == TEST){
  free_stream(ipdest);
  token_init();
  head = NULL;
  tail = NULL;
  sq = NULL;
  stream_start = 0;
}
find(ipdest, port);
i = token_bucket();
if( i == 1){
  if (sq->qpacket > 0){
    wait_for_tokens();
    send_prev_packet();
  }
  ua_set_32_dram((__declspec(sdram) unsigned char *)p_pkt_hdr,pMeta->bufferSize,type);


  pMeta->bufferSize = pMeta->bufferSize + 8;

  put_ring(IPV4_TO_QM_SCR_RING_BASE,&sig_scr_put);

}
else if ( i == 2){
  ua_set_32_dram((__declspec(sdram) unsigned char *)p_pkt_hdr,pMeta->bufferSize,dlBufHandle.value);

  pMeta->bufferSize = pMeta->bufferSize + 8;
  put_ring(IPV4_TO_QM_SCR_RING_BASE,&sig_scr_put);

}


      }
      else{

ua_set_32_dram((__declspec(sdram) unsigned char *)p_pkt_hdr,pMeta->bufferSize,0x2);
ua_set_32_dram((__declspec(sdram) unsigned char *)TIMESTAMP_BASE, 8 ,dlBufHandle.value);
pMeta->bufferSize = pMeta->bufferSize + 8;

put_ring(IPV4_TO_QM_SCR_RING_BASE,&sig_scr_put);

      }


    }
    else{
//o packet in ,check if there is a queue to take packet

      if(stream_start == 1){
if (check_qpackets(0) == 1){

  ua_set_32_dram((__declspec(sdram) unsigned char *)TIMESTAMP_BASE, 8 ,pMeta->bufferSize + p_pkt_hdr);
  signal_me_ctx1(0x11,ECHO_SIG);
```

```
  pMeta->bufferSize = pMeta->bufferSize + 8;

  put_ring(IPV4_TO_QM_SCR_RING_BASE,&sig_scr_put);
}
     }
    }
  }//while
}//if
} /* main*/

//exit(unsigned int arg){}
```

A.2 prioritetsortering.c

```
#include <ixp.h>
#include <dl_system2.h>
#include <dl_buf.c>
#include <dl_meta.c>
#include <hardware.h>
#include <sig_functions.c>
#include <ixp_lib.h>
#include <ixp_crc.c>
#include <rtl.c>
#include <ixp_cycle.c>
#include <libc.c>
#include <string.h>



/*sjekker om scratch ringene er fulle*/
#define RING_TX_F inp_state_scr_ring9_full
#define RING_1_F inp_state_scr_ring10_full
#define RING_2_F inp_state_scr_ring7_full
#define RING_3_F inp_state_scr_ring8_full
#define RING_4_F inp_state_scr_ring11_full


/*keeps track of packets that have been placed on each ring*/
__declspec(export dram) unsigned int inprio1 = 0;
__declspec(export dram) unsigned int inprio2 = 0;
__declspec(export dram) unsigned int inprio3 = 0;
__declspec(export dram) unsigned int inprio4 = 0;

/*keeps track of over all prio packet that have entered the microengine*/
__declspec(export dram) int droped1 = 0;
__declspec(export dram) int droped2 = 0;
__declspec(export dram) int droped3 = 0;
__declspec(export dram) int droped4 = 0;
__declspec(export dram) int prio1 = 0, prio2 = 0, prio3 = 0, prio4 = 0;

/*noen faste variabler for uttak av pakken */
#define PRIORITY     42//74
#define TYPE         23
#define SSID     50
#define SDP_LOC         117
#define UDP   0x11
#define TCP   0x06
#define RTCP   0xc8
#define SDP     0x2f736470
#define TEST     0x74657374
extern dl_buf_handle_t  dlBufHandle; // The current buffer handle
unsigned int timestamp;
U32 prio;
U8 type;
U8 rtcp;
__declspec(gp_reg) unsigned int ipdest;
__declspec(sram_read_reg) unsigned int  rx_msg;
__declspec(sram_write_reg) unsigned int txReq;
__declspec(sdram) unsigned char *p_pkt_hdr;
__declspec(sram) dl_meta_t *pMeta;



//unsigned int sramAddress;
```

```
/*Signals that we use*/
SIGNAL sig_scr_get_rx;
SIGNAL sig_scr_put_tx;
SIGNAL sig_scr_put_1;
SIGNAL sig_scr_put_2;
SIGNAL sig_scr_put_3;
SIGNAL sig_scr_put_4;
SIGNAL sig_never;
SIGNAL system_init_done;



INLINE void put_ring(int ring,SIGNAL *sig)
{
scratch_put_ring(&txReq,
(void*)(ring << 2),
1,
sig_done,
sig);

wait_for_all(sig);
}
////////////////////////////////////////////////////////////////////////////////
//
// get_time()
//
//   Description:
//
// Demonstrate the timestamp capability. Gets the time and from the system.
//
//   Outputs:
//
// Modifies a global variable 'timestamp' which is a unsigned long long (64bits).
//
// Inputs
//
// None.
//
// CONSTANTS
//
//
// Size:
// N/A
////////////////////////////////////////////////////////////////////////////////
int get_time(void){

  unsigned int low, high;

  low = (unsigned long long)(local_csr_read (local_csr_timestamp_low)) ;
  high = (unsigned long long)(local_csr_read (local_csr_timestamp_high));

  timestamp = (unsigned long long)(low);

  // returns milisec
  return(timestamp/37500); //& 0xffffffff);

}

void switch_header(){

  SIGNAL_PAIR dram_sig;


  __declspec(gp_reg) unsigned int ipsrc, ethsrc1, ethdest1;
  __declspec(gp_reg) unsigned short ethsrc2, ethdest2;
  U16 iptype, port;
  int time, nexttest;


  dlBufHandle.value = rx_msg;

  pMeta = (__declspec(sram) dl_meta_t *)Dl_BufGetDesc(dlBufHandle);

  p_pkt_hdr = (__declspec(sdram) unsigned char *) (Dl_BufGetData(dlBufHandle) +
   pMeta->offset);

  ua_set_32_dram((__declspec(sdram) unsigned char *)TIMESTAMP_BASE ,16, pMeta->bufferSize + p_pkt_hdr);
  signal_me_ctx2(0x11,ECHO_SIG);
  pMeta->bufferSize = pMeta->bufferSize - 4;

  if(pMeta->inputPort == 1){
    /*  Hente ut src og dest adr peker fra ethernet header.
```

```
adressen er på 6 bytes, så gjør det i to operasjoner */
    ethdest1 = ua_get_u32(p_pkt_hdr, 0);
    ethdest2 = ua_get_u16(p_pkt_hdr, 4);
    ethsrc1 = ua_get_u32(p_pkt_hdr, 6);
    ethsrc2 = ua_get_u16(p_pkt_hdr, 10);
    /* Hente ut src og dest adr peker fra IP header*/
    ipsrc = ua_get_u32(p_pkt_hdr, 14 + 12);
    ipdest = ua_get_u32(p_pkt_hdr, 14 + 16);


    /* snur hedarene*/
    ua_set_32_dram(p_pkt_hdr,0,ethsrc1);
    ua_set_16_dram(p_pkt_hdr,4,ethsrc2);
    ua_set_32_dram(p_pkt_hdr,6,ethdest1);
    ua_set_16_dram(p_pkt_hdr,10,ethdest2);
    ua_set_32_dram(p_pkt_hdr,14 + 12, ipdest);
    ua_set_32_dram(p_pkt_hdr,14 + 16, ipsrc);
  }

  /* Om det er Tcp eller Udp pakke vi jobber med*/
  type = ua_get_u8(p_pkt_hdr ,TYPE);
  /*sjekker om det er rtcp pakker*/
  rtcp = ua_get_u8(p_pkt_hdr,43);

  iptype = ua_get_u16(p_pkt_hdr,12);

  if (iptype == 0x8808){
    pMeta->inputPort = 1;
    //Dl_BufDrop(dlBufHandle);
    prio = 1;
  }

  txReq = (rx_msg & 0x00FFFFFF) | ((pMeta->inputPort)<< 24) |(1 << 31);

  if(type == UDP && rtcp != RTCP){

    port = ua_get_u16(p_pkt_hdr,36);
    prio = ua_get_u32_dram(p_pkt_hdr,PRIORITY);
    if(prio == 1)
      prio1++;
    else if(prio == 2)
      prio2++;
    else if(prio == 3)
      prio3++;
    else if(port == 0x1b42)
      prio4++;

  }

  nexttest = ua_get_u32(p_pkt_hdr,42);
  if (nexttest == TEST){
    prio = 1;
    pMeta->bufferSize =  pMeta->bufferSize + 16;
    ua_set_32_dram((__declspec(sdram) unsigned char *)p_pkt_hdr + pMeta->bufferSize,-16,prio1);
    ua_set_32_dram((__declspec(sdram) unsigned char *)p_pkt_hdr + pMeta->bufferSize,-12,prio2);
    ua_set_32_dram((__declspec(sdram) unsigned char *)p_pkt_hdr + pMeta->bufferSize,-8,prio3);
    ua_set_32_dram((__declspec(sdram) unsigned char *)p_pkt_hdr + pMeta->bufferSize,-4,prio4-1);
    prio1 =0;
    prio2 =0;
    prio3 =0;
    prio4 =0;
  }
}
void place_on_ring(int prio){

  if (prio == 1|| type == TCP ) {
    /*sjekker at ringen ikke er full, hvis den er full kaster den pakka*/
    if(!inp_state_test(RING_1_F)){
      put_ring(PACKET_PRIO_RING_1,&sig_scr_put_1);
      inprio1++;
    }
    else{
      droped1++;
      Dl_BufDrop(dlBufHandle);
    }
  }
  else if (prio == 2){
    //sjekker at ringen ikke er full, hvis den er full kaster den pakka
    if(!inp_state_test(RING_2_F)){
      put_ring(PACKET_PRIO_RING_2,&sig_scr_put_2);
      inprio2++;
    }
    else{
```

```
      droped2++;
      Dl_BufDrop(dlBufHandle);
    }
  }
  else if (prio == 3){
    //sjekker at ringen ikke er full, hvis den er full kaster den pakka
    if(!inp_state_test(RING_3_F)){
      put_ring(PACKET_PRIO_RING_3, &sig_scr_put_3);
      inprio3++;
    }
    else{
      droped3++;
      Dl_BufDrop(dlBufHandle);
    }
  }
  else {


    //sjekker at ringen ikke er full, hvis den er full kaster den pakka
    if(!inp_state_test(RING_4_F)){
      put_ring(PACKET_PRIO_RING_4,&sig_scr_put_4);
inprio4++;
    }
    else{
      droped4++;
      Dl_BufDrop(dlBufHandle);
    }
  }
}


main() {

  if (ctx() == 0) {

    __assign_relative_register(&system_init_done, ME_INIT_SIGNAL);


#ifdef WAIT_FOR_COMMON_RESOURCE_INITIALIZATION
      wait_for_all(&system_init_done)

#endif

while(1){
start:
  scratch_get_ring(&rx_msg,
   (void*)(IPV4_TO_QM_SCR_RING_BASE << 2),
   1,
   sig_done,
   &sig_scr_get_rx);

  wait_for_all(&sig_scr_get_rx);


  if (rx_msg){
    /* the ring is not empty */


    switch_header();


    place_on_ring(prio);
  }


} /*while(1)*/

  }
  else { /* if(ctx() != 0) */
    __asm{ctx_arb[kill]};
  }
} /* main*/

//exit(unsigned int arg){}
```

A.3 Prioritetsvelger.c

```
#include "select_utils.c"


/*sjekker om scratch ringene er fulle*/
#define RING_TX_0 inp_state_scr_ring6_full
#define RING_TX_2 inp_state_scr_ring9_full

__declspec(export dram) unsigned int testing = 0;
__declspec(export dram) unsigned int dropped = 0;
__declspec(import dram) unsigned int inprio1;
__declspec(import dram) unsigned int inprio2;
__declspec(import dram) unsigned int inprio3;
__declspec(import dram) unsigned int inprio4;
__declspec(import dram_read_reg)int current_timestamp;

__declspec(export dram) int prio1out = 0;
__declspec(export dram) int prio2out = 0;
__declspec(export dram) int prio3out = 0;
__declspec(export dram) int prio4out = 0;

main() {
  SIGNAL sig_scr_get_rx;
  SIGNAL sig_scr_put_tx0;
  SIGNAL sig_scr_put_tx1;
  SIGNAL sig_scr_put_tx2;
  SIGNAL sig_scr_1;
  SIGNAL sig_scr_2;
  SIGNAL sig_scr_3;
  SIGNAL sig_scr_4;
  SIGNAL sig_never;
  SIGNAL slow_down_sig;
  SIGNAL system_init_done;

  int i;


  if (ctx() == 0){

    /*assign signal from other micro engine to a signal nummer in this one*/

    __assign_relative_register(&system_init_done, ME_INIT_SIGNAL);

#ifdef WAIT_FOR_COMMON_RESOURCE_INITIALIZATION
wait_for_all(&system_init_done);
#endif


while(1){
runde:
  for(i = 0; i < 4; i++){
    if(inprio1 > prio1out){

      if (get_ring_data(PACKET_PRIO_RING_1,&sig_scr_1)){
Dl_BufDrop(dlBufHandle);
goto runde;
      }
      prio1out++;
      if (port == 0){
while(inp_state_test(RING_TX_2)){}
put_ring_data(PACKET_TX_SCR_RING_2,&sig_scr_put_tx2);

      }
      else if(port == 1){
while(inp_state_test(RING_TX_2)){}
put_ring_data(PACKET_TX_SCR_RING_1,&sig_scr_put_tx1);

      }
      else if(port == 2){
while(inp_state_test(RING_TX_0)){}
put_ring_data(PACKET_TX_SCR_RING_0,&sig_scr_put_tx0);

      }
      dropped++;
    }
    else
      break;
  }

  for(i = 0; i < 3; i++){
```

```
    if(inprio1 > prio1out )
      goto runde;
    if(inprio2 > prio2out){
      if (get_ring_data(PACKET_PRIO_RING_2,&sig_scr_2)){
Dl_BufDrop(dlBufHandle);
goto runde;
      }
      prio2out++;
      if (port == 0){
while(inp_state_test(RING_TX_2)){}
put_ring_data(PACKET_TX_SCR_RING_2,&sig_scr_put_tx2);

      }
      else if(port == 1){
while(inp_state_test(RING_TX_2)){}
put_ring_data(PACKET_TX_SCR_RING_1,&sig_scr_put_tx1);

      }
      else if(port == 2){
while(inp_state_test(RING_TX_0)){}
put_ring_data(PACKET_TX_SCR_RING_0,&sig_scr_put_tx0);
      }
      dropped++;


    }
    else
      break;

  }

  for(i = 0; i < 2; i++){
    if(inprio1 > prio1out || inprio2 > prio2out)
      goto runde;
    if(inprio3 > prio3out){

      if (get_ring_data(PACKET_PRIO_RING_3,&sig_scr_3)){
Dl_BufDrop(dlBufHandle)
  goto runde;
      }
      prio3out++;
      if (port == 0){
while(inp_state_test(RING_TX_2)){}
put_ring_data(PACKET_TX_SCR_RING_2,&sig_scr_put_tx2);

      }
      else if(port == 1){
while(inp_state_test(RING_TX_2)){}
put_ring_data(PACKET_TX_SCR_RING_1,&sig_scr_put_tx1);

      }
      else if(port == 2){
while(inp_state_test(RING_TX_0)){}
put_ring_data(PACKET_TX_SCR_RING_0,&sig_scr_put_tx0);

      }
      dropped++;

    }
    else
      break;

  }

  for(i = 0; i < 1; i++){
    if((inprio1 > prio1out) || inprio2 > prio2out || inprio3 >prio3out)
      goto runde;
    if(inprio4 > prio4out ){

      if (get_ring_data(PACKET_PRIO_RING_4,&sig_scr_4)){
Dl_BufDrop(dlBufHandle);
goto runde;
      }
      prio4out++;
      if (port == 0){
while(inp_state_test(RING_TX_2)){}
put_ring_data(PACKET_TX_SCR_RING_2,&sig_scr_put_tx2);

      }
      else if(port == 1){
while(inp_state_test(RING_TX_2)){}
put_ring_data(PACKET_TX_SCR_RING_1,&sig_scr_put_tx1);
      }
```

```
      else if(port == 2){
while(inp_state_test(RING_TX_0)){}
put_ring_data(PACKET_TX_SCR_RING_0,&sig_scr_put_tx0);

      }
      dropped++;
    }
    else
      break;
  }

}/*end while*/
  }
  else{
    __asm{ctx_arb[kill]};
  }
} /* main*/

//exit(unsigned int arg){}
```

A.4 timer.c

```
#include <ixp.h>
#include <dl_system2.h>
#include <hardware.h>
#include <sig_functions.c>
#include <rtl.c>
#include <libc.c>
#include <string.h>

////////////////////////////////////////////////////////////////////////////////
//
// get_time()
//
//        Description:
//
// Demonstrate the timestamp capability. Gets the time and from the system.
//
//   Outputs:
//
// Modifies a global variable 'timestamp' which is a unsigned long long (64bits).
//
// Inputs
//
// None.
//
// CONSTANTS
//
//
// Size:
// N/A
////////////////////////////////////////////////////////////////////////////////
int get_time(void){
  unsigned int timestamp;
  unsigned int low, high;

  low = (unsigned long long)(local_csr_read (local_csr_timestamp_low)) ;


  timestamp = (unsigned long long)(low);

  return(timestamp);

}



main(){

  SIGNAL Txtimestamp;
  SIGNAL Rxtimestamp;
  SIGNAL echo_timestamp;
  SIGNAL select_timestamp;
  SIGNAL shaper_timestamp;
```

```
  if (ctx() ==  0){
    U32 time,loc;

    __assign_relative_register(&Rxtimestamp, RX_SIG);
    while(1){
      wait_for_all(&Rxtimestamp);

      time =  get_time();
      loc = ua_get_u32_dram((__declspec(sdram) unsigned char *)TIMESTAMP_BASE,0);
      ua_set_32_dram((__declspec(sdram) unsigned char *)loc,-4,0x1);
    }
  }
  else if (ctx() ==  1){
    unsigned int time, loc;
    __assign_relative_register(&shaper_timestamp, ECHO_SIG);
    while(1){
      wait_for_all(&shaper_timestamp);

      time =  get_time();
      loc = ua_get_u32_dram((__declspec(sdram) unsigned char *)TIMESTAMP_BASE,8);
      ua_set_32_dram((__declspec(sdram) unsigned char *)loc,0,0x2);
    }
  }
  else if (ctx() ==  2){
    unsigned int time, loc;
    __assign_relative_register(&echo_timestamp, ECHO_SIG);
    while(1){
      wait_for_all(&echo_timestamp);

      time =  get_time();
      loc = ua_get_u32_dram((__declspec(sdram) unsigned char *)TIMESTAMP_BASE,16);
      ua_set_32_dram((__declspec(sdram) unsigned char *)loc,-4,0x3);
    }
  }
  else if (ctx() ==  3){
    unsigned int time, loc;
    __assign_relative_register(&select_timestamp, SELECT_SIG);
    while(1){
      wait_for_all(&select_timestamp);

      time =  get_time();
      loc = ua_get_u32_dram((__declspec(sdram) unsigned char *)TIMESTAMP_BASE,24);
      ua_set_32_dram((__declspec(sdram) unsigned char *)loc,-4,0x4);
    }
  }
  else if (ctx() ==  4){
    unsigned int time, loc;
    __assign_relative_register(&Txtimestamp, TX_SIG);
    while(1){
      wait_for_all(&Txtimestamp);

      time =  get_time();
      loc = ua_get_u32_dram((__declspec(sdram) unsigned char *)TIMESTAMP_BASE,16);
      ua_set_32_dram((__declspec(sdram) unsigned char *)loc,0,0x5);

      signal_me(0x10, 14);
      signal_me_ctx1(0x10, 14);
      signal_me_ctx2(0x10, 14);
      signal_me_ctx3(0x10, 14);

    }
  }

}
```