UNIVERSITETET I OSLO
Institutt for informatikk

Bufferhåndtering i INSTANCE

Espen Jorde

Hovedoppgave til Cand. Scient. graden

1. november 2000

Forord

Takk til min hovedveileder Vera Gobel! Videre rettes en takk til Thomas Plagemann som også har bistått med nyttig veiledning. Ikke minst takk til Pål Halvorsen som gjenom store deler av tiden har vært sentral i INSTANCE-gruppen og kommet med uvurdelig hjelp. Hadde det ikke vært for Påls oppmuntring hadde denne oppgaven kanskje aldri blitt ferdig. Takk også til Ketil Lund for hjelp, støtte og innspill.

Jeg retter også en takk til administrasjonen ved Universitetsstudiene på Kjeller (UniK). De har gitt hjelp og lagt til rette alle de praktiske ting rundt det å gjennomføre et hovedfag. Takk! Driftsgruppen på UniK har vært behjelpelig med det nødvenige utstyret til utvikling, implementering og testing av prinsippene i denne oppgaven. Til sist en stor takk til venner, familie og samboer for at de har levd med meg og mitt vekslende humør under og ikke minst mot slutten av hovedfagsperioden.

Denne oppgaven er satt med 11pt. times og produsert med LaTeX. Bibliografien er produsert med BibTeX.

Kjeller 13. november 2000
Espen Jorde

Sammendrag

Denne oppgaven tar for seg minnehåndteringen i en spesiell tjener, kalt en INSTANCE-node. INSTANCE-noden er ingen tradisjonell, multifunksjonell tjener. Den er spesialisert enhet for hurtig, billig og effektiv distribusjon av video, lyd og multimediadata til mange samtidige klienter.

Bufferhåndteringen er designet for å støtte kontinuerlige datastrømmer med både variabel og konstant bitrate. Vi har laget en zero-kopy in-kernel datapath basert på minneområder som deles mellom disksystemet og nettverkssystemet. Hver datastrøm for tilordnet to fast allokerte buffere og lesing og sending veksler mellom disse. Disse to minneområdene kalles INSTANCE-buffere, og i tilleg har hver datastrøm en INSTANCE-strøm som kontrollerer håndteringen av bufferne.

Vi har implementert konseptet i NETBSD ved å legge inn fire nye systemkall. Målingene viser at koseptet reduserer CPU-bruken ved datadistribusjon kraftig. I en situasjon der bufferkopiering er en begrensende faktor oppnår vi økt båndbredde gjennom maskinen ved å bruke det nye konseptet for bufferhåndtering sammelignet med tradisjonelle read-write kall.

Til sist viser vi at valg av teknologi gjør INSTANCE-noden fullstendig skalerbar. Gjennom økning av båndbredden i nettet og ved å sette flere INSTANCE-noder i paralell kan vi oppnå resultater som er direkte proposjonalt med antallet noder.

Innhold

Figurer

Tabeller

Kapittel 1

Innledning

1.1 Motivasjon og Bakgrunn

Informasjonsteknologien er preget av rask utvikling på alle områder. Lagringskapasitet, nettverkshastighet og prosessorhastighet øker hurtig. Stadig flere tjenester blir mulige. Ikke minst ser vi en utvikling hvor stadig flere og bedre tjenester blir tilgjengelige for vanligere bruker til en stadig lavere pris.

Vi ønsker derfor å se på mulighetene for å tilby *Multimedia on Demand (MoD)* ut til sluttbrukerne. Med MoD mener vi en kombinasjon av video, lyd, tekst og bilder levert via datanett og vist med hjelp av en datamaskin. Dataene skal leveres "on Demand", det vil si at dataene vises *når* brukerne ønsker det. I dag kan man oftest velge mellom god kvalitet men lang nedlastningstid, eller lav kvalitet og kort nedlastningstid. Foreløpig virker høykvalitets MoD og *Video on Demand (VoD)* levert til beboere i byer og tettsteder som en fjern fremtid. Prisene til nå, ikke minst på nettverkstjenester gjør det foreløpig dyrere å overføre en videofilm over et nettverk enn å tilby den samme filmen i en videobutikk. Det å frakte data lagret på fysiske lagringsmedia som en videofilm er faktisk en både rimelig løsning og en løsning med svært høy "overføringshastighet", som Tanenbaum sier:

> *«Never underestimate the bandwith of a station wagon full of tapes hurtling down the highway.»*
>
> *Andrew S. Tanenbaum*[39]

Vi mener det er en reel konkurranse mellom MoD og VoD og distribusjon av data via fysiske media. For at folk skal velge MoD fremfor videobutikken må MoD

enten tilby varer butikken ikke kan skaffe, eller være like bra eller bedre på pris og kvalitet.

Selv om MoD ut til hjemmene ikke er vanlig blir distribuering av *Multimedia (Multi-Media MM)*-data over nettverk likevel vanligere i andre sammenhenger. Man kan høre nyheter fra radiostasjoner og se sekvenser fra tv og filmproduksjoner over nettet. Som et eksempel kan det nevnes at traileren til filmen "Star Wars Episode I - The Phantom Menace" ble lastet ned over en million ganger i løpet av det første døgnet (12. mars 1999)[2]. Videre har distribusjon av musikk over Internett nærmest eksplodert i omfang. Så stort at musikkbransjen ser det som en seriøs trussel og har satt i gang en rekke rettssaker. Alt dette tyder på at distribusjon av forskjellige varianter av MM over nettverk vil øke fremover. Etter vår mening er det dermed bare et spørsmål om tid før MoD og VoD blir tilgjengelig i vanlige hjem.

MoD kan ha mange bruksområder fremover. Et av dem vil trolig være undervisning. Til undervisning kan det være attraktivt for elever å se forelesninger de ikke kunne delta på, eller repetere viktige forelesninger før eksamen. Flere og flere forelesere bruker nå datapresentasjoner som vises via en videoprosjektør i stedet for å skrive på en tavle. Med et enkelt videokamera og en mikrofon kan man ta opp forelesningen. Hvis man kombinerer presentasjonene med et digitalisert opptak av forelesningene har man en ypperlig applikasjon for en enkel MoD-tjener. I denne sammenhengen vil produksjon og distribusjon av videofilmer med stor sannsynlighet være mer tungvint og trolig også mer kostbart enn å distribuere det elektronisk.

Andre tenkte tjenester for en MoD-tjener er video på hoteller, der en sentral datamaskin kan erstatte mennesker som i dag betjener videospillere med forskjellige videokassetter for å kunne tilby filmer på rommene. Også muligheten til å tilby film på fly er en mulig nisje for MoD-tjenere med dagens teknologi.

Selv om vi mener at MoD og VoD først vil komme i full skala senere, har vi nå nevnt flere områder der MoD etter vår mening med fordel kan benyttes allerede i dag. Hvilken teknologi er så ønskelig for å tilby MoD? Vanlige hjemmemaskiner er nå så kraftige at de uten store problem kan ta imot og spille av video og andre MM-data med høy kvalitet. Jobben blir dermed å lage en MoD-tjener som kan betjene disse klientene uten at prisen på denne MoD-tjeneren blir uakseptabelt høy.

Ønsket om å designe en MoD-tjener til en akseptabel pris og helst basert på vanlige PC-komponenter ligger til grunn for oppstarten av prosjektet *Intermediate Storage Node Concept (*INSTANCE*)*. Prosjektet er startet på *Universitetsstudiene på Kjeller (UniK)*. Målet er å bygge en spesialisert og dedikert MoD-tjener basert på standard hardware, kalt en INSTANCE-node.

For å bygge en MoD-tjener basert på standard PC komponenter er det imidlertid en del flaskehalser som kan oppstå i maskinen og som skaper problemer når man

ønsker å bruke den som en MoD-tjener. Både diskhastighet og nettverkshastighet har økt. Prosessorene har blitt stadig kraftigere. Derimot har flytting av data internt i maskinen blitt en flaskehals. I et tradisjonelt operativsystem blir data kopiert fra minneområde til minneområde flere ganger på veien mellom disken og nettverket. Denne kopieringen tar tid og begrenser klart den maksimale gjennomstrømningshastigheten fra disk og ut på nettverk. Kostbare minnekopieringsoperasjoner mellom disk og nettverk er et kjent problem og det er det kommet mange forslag til nye mekanismer for å redusere kopieringen. Ennå har ingen av disse mekanismene etter vår mening pekt seg ut som den beste. Ikke minst er lite gjort når det gjelder å implementere slike mekanismer i operativsystemer med stor kommersiell utbredelse. Denne oppgaven vil derfor se på forskjellige slike mekanismer i lys av de krav som stilles til INSTANCE-noden.

Denne oppgaven er skrevet som en hovedoppgave til graden Cand. Scient. ved Institutt for Informatikk, Universitetet i Oslo. Målet med denne oppgaven har dermed både vært å bidra i utviklingen av INSTANCE-noden. Samtidig er oppgaven skrevet som en eksamensbesvarelse til Cand. Scient. graden. Begge disse målene vil nødvendigvis prege oppgaven.

1.2 Problemstilling

Vi tar i denne oppgaven utgangspunkt i INSTANCE-noden, en spesialisert tjener for distribusjon av MoD. MM-data som lyd, video og andre datatyper stiller store krav til ytelse og en effektiv håndtering av dataene. Ikke bare er datamengdene store, men i en MoD-applikasjon må de også komme frem til klienten til riktig tid dersom de skal være av noen verdi.

Det peker seg ut fire viktige delsystemer, eller komponenter i INSTANCE-noden. Den første er applikasjonen som sørger for at riktige data når riktig mottager. Den andre er det vi kaller disksystemet. Disksystemet er den delen av *operativsystem (OS)* som står for lagringen av data på disk. Herunder hvordan de lagres og hvor de lagres. Videre har vi nettverkssystemet som det tredje komponenten. Nettverkssystemet er ansvarlig for å dele dataene inn i pakker og sende dem ut på nettverket i retning klientene. Den fjerde komponenten er mekanismen for bufferhåndtering. Med bufferhåndtering mener vi i oppgaven å flytte dataene fra disksystemet og til nettverkssystemet På en slik måte som applikasjonen ønsker. Det er denne komponenten vi skal ta for oss i denne oppgaven.

Den tradisjonelle bufferhåndteringen har vært designet for å håndtere mange applikasjoner og mange brukere på en maskin. Et av målene for bufferhåndteringen har vært å effektivisere diskhåndteringen ved å lagre data i *Inn/ut (Input/Output, I/O)*-buffere før de overføres til applikasjonen. Dette gjør at dersom en applikasjon leser samme data flere ganger etter hverandre, kan alle leseoperasjoner etter den

første hente data fra I/O-bufferet i stedet for å foreta en tidskostbar leseoperasjon fra disken. Disse I/O-bufferne brukes også til å samle flere små leseoperasjoner til en stor. De gjør dette ved å lese mer data enn nødvendig i den ofte riktige antagelse at dersom en applikasjon leser litt av en fil vil den trolig lese mer senere. Bufferhåndteringsmekanismen står videre for å kopiere data fra I/O-bufferne og inn i applikasjonens minneområde. Hensikten her er å skape sikkerhet ved at applikasjonen ikke har direkte tilgang til I/O-bufferne som tilhører kjernen av OS-et. Dersom dataene skal sendes via et nettverk vil bufferhåndteringsmekanismen kopierer data tilbake i kjernen av OS-et. Der blir de så delt opp i passe deler og sendt ut via nettverket. Alt dette gir en lang rekke kostbare minnekopieringsoperasjoner.

En INSTANCE-node er ingen generell tjener men en spesialisert tjener for MoD. Dermed er ikke antagelsene bak designet av den tradisjonelle bufferhåndteringen gjeldene. Dersom man sender en videofilm ut via ett nettverk er det snakk om store datamengder, og applikasjonen vil lese store og ikke små mengder data om gangen. Sannsynligheten for at samme data leses to ganger etter hverandre blir liten fordi dataene er lineære av natur. I en INSTANCE-node vil en tradisjonell bufferhåndtering dermed gi få eller ingen av de fordelene den er designet for. En bufferhåndteringsmekanisme i INSTANCE-noden bør i stedet ha som mål å få dataene raskt og effektivt gjennom systemet.

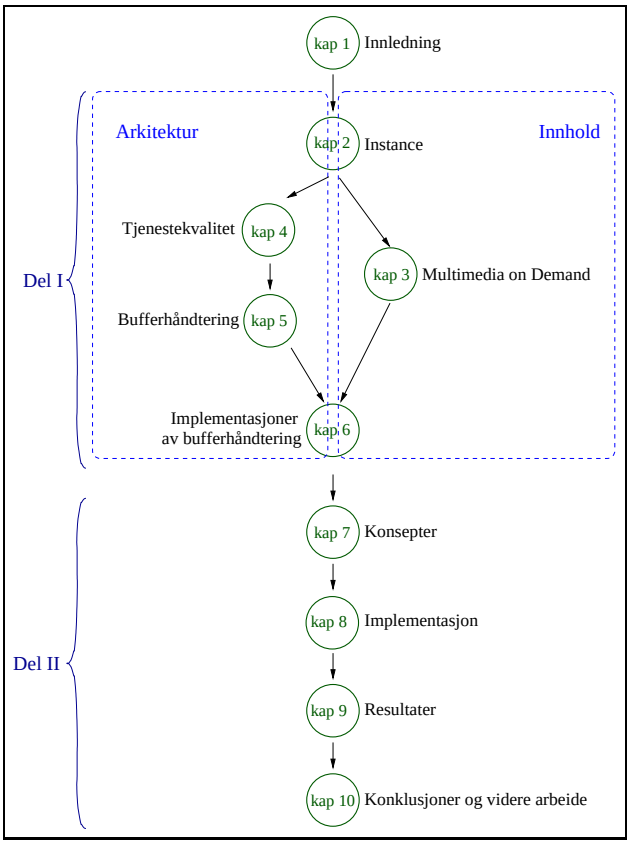
For å kunne optimalisere dataflyten gjennom INSTANCE-noden best mulig er det en fordel at dataflyten er mest mulig forutsigbar. For å få til dette er de ønskelig å komme frem til en effektiv og ressursbesparende måte å distribuere data på. Den bør gi langvarige strømmer av data gjennom noden og tilfredsstille flest mulig klienter med hver strøm. Samtidig må metoden gi en god kvalitet på tjenesten til klientene. Vi mener derfor at det er en fordel å se dataflyten internt i INSTANCE-noden og eksternt mellom INSTANCE-noden og klientene i sammenheng. I arbeidet med denne oppgaven har vi derfor jobbet utfra følgende problemstilling:

> *«Hvordan skal data håndteres internt i en* INSTANCE*-node og generelt i et* INSTANCE*-nettverk for å oppnå størst mulig kapasitet med best mulig kvalitet og minst mulig ressursbruk?»*

1.3 Struktur

Denne oppgaven er delt inn i to hoveddeler. Først del I, som beskriver bakgrunnen som oppgaven bygger på. Etter dette følger del II som beskriver konseptene vi har kommet frem til, realisering av konseptene og resultater. Figur 1.1 illustrerer flyten i oppgaven og den kapitlenes logiske rekkefølge.

I del I ser på bakgrunnen for bufferhåndteringen i INSTANCE-noden. Bakgrunnsdelen er splittet i to underdeler, innhold og arkitektur. Innholdsdelen tar for seg de



Figur 1.1: Flytskjema for oppgaven

tjenestene INSTANCE-noden skal tilby og innholdet i disse tjenestene. Delen om arkitektur tar for deg krav og muligheter til arkitekturen i bufferhåndteringen.

I kapittel 2 tar vi for oss grunnlaget for oppgaven, det vil si INSTANCE-konseptet slik det forelå ved oppgavens begynnelse. Det var dette materialet som lå til grunn for alt det videre arbeidet. Vi ser her tidlig i oppgaven arkitektur og innhold i sammenheng. Vi går så inn på MM og MoD i kapittel 3. Dette fordi det er MM-data INSTANCE-noden som en MoD-tjener skal distribuere. Deretter kommer en introduksjon om temaet *tjenestekvalitet (Quality of Service, QoS)* i kapittel 4. At data skal distribueres med en hvis kvalitet stiller klare krav til bufferhåndteringen. Vi ser så på prinsipper for bufferhåndtering i kapittel 5 og vurder prinsippene i lys av de krav distribusjon av MM-data stiller, og kravet til QoS. Til slutt ser vi på andres arbeide med bufferhåndtering i kapittel 6. Noen av disse konseptene ser designet av bufferhåndteringen og innholdet som skal distribueres i sammenheng.

I del II kommer vi til den del av oppgaven der vi presenterer våre egne konsept for bufferhåndtering. I kapittel 7 presenterer vi først INSTANCE-nodens plassering og funksjon i nettverket. Videre introduserer vi to nye konsepter for minnehåndtering, INSTANCE-bufferet og INSTANCE-strømmen. Vi beskriver hvordan INSTANCE-bufferne og INSTANCE-strømmen brukes til å flytte data gjennom INSTANCE-noden. Kapittel 8 om implementasjonen beskriver hvordan vi har laget en prototype på en INSTANCE-node der vi tester de nye konseptene for minnehåndtering i praksis. Beskrivelse av testene med prototypen og de resultater vi har oppnådd hører hjemme i kapittel 9. Kapittel 10 inneholder konklusjoner og drøftinger av resultatene samt beskriver det som gjenstår av videre arbeide med konsepter og implementasjon.

Del I

Bakgrunn

Kapittel 2

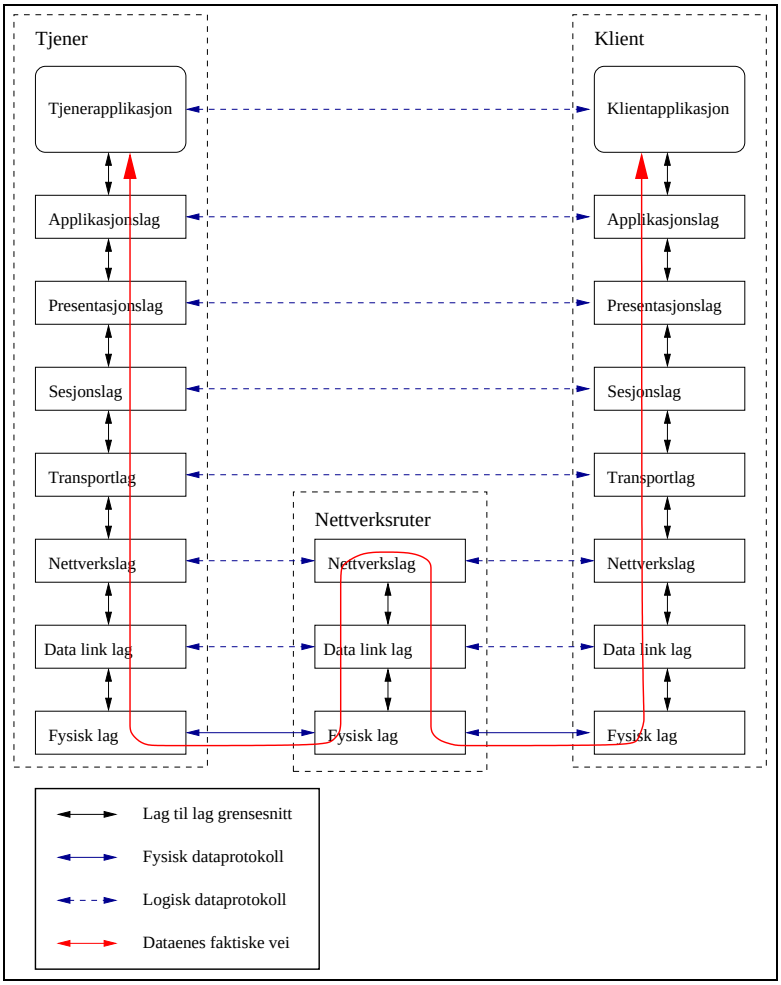
INSTANCE

Dette kapittelet beskriver konseptet *Intermediate Storage Node Concept (*INSTANCE*)* [36]. Kapittelet er ment å gi en rask gjennomgang av INSTANCE-konseptet. Vi har med hensikt plassert dette kapittelet i begynnelsen av oppgaven. Dette oss mulighet til senere i oppgaven å vurdere forskjellige teknikker i lys av hva som kan egne seg best i en fremtidig INSTANCE-node. Ulempen ved å ta dette allerede i kapittel 2 er at vi vi dermed ser oss nødt til å benytte begreper og konsepter i dette kapittelet som ikke blir belyst før senere i oppgaven. Kapittelet kan og bør derfor leses som en innledning og innføring i INSTANCE-konseptet, og ikke en detaljert forklaring. Vi vil senere, i kapittel 7, gå lenger inn på detaljene i INSTANCE-konseptet. Vi ser der på hvordan INSTANCE-noden fungerer i nettverket og hvilke buffermekanismene vi foreslår brukt.

I dette kapittelet ser vi først i avsnitt 2.1 raskt på den lagdelte modellen for datakommunikasjon og prinsipper for distribusjon i datanettverk. Dette avsnittet legger et grunnlag for beskrivelsen av idéene bak INSTANCE-noden i avsnitt 2.2. Videre ser vi i avsnitt 2.3 på hvordan prinsippene i *Integrated Layer Processing (ILP)*[14] påvirker designet av noden. Avsnitt 2.3.3 tar for seg delsystemene i INSTANCE-noden og forslår å slå disse sammen til et system. Vi gir så i avsnitt 2.4 en rask introduksjon til hovedtemaet i denne oppgaven, bufferhåndteringen i INSTANCE-noden. Vi avslutter kapittelet med avsnitt 2.5 der vi raskt oppsummerer beskrivelsen av INSTANCE-konseptet.

2.1 Datakommunikasjon

Vi går nå raskt inn på noen av de grunnleggende prinsippene innen datakommunikasjon. Vi forklarer begreper som er nødvendige for forståelsen av dette kapittelet og andre prinsipper senere i oppgaven. Vi ser først på prinsippet om den



Figur 2.1: Den lagdelte OSI modellen

lagdelte kommunikasjonsprotokollen og tar videre for oss grunnleggende distribusjonsterminologi. For en grundig innføring i datakommunikasjon anbefaler vi Tanenbaum[39].

2.1.1 Den lagdelte kommunikasjonsprotokollen

Beskrivelsene av INSTANCE-konseptet baserer seg på den lagdelte *Open System Interconnection (OSI)*[26] modellen. Selv om den rådende protokollen i dag, TCP/IP, er en firedelt protokoll, brukes OSI-modellen fra *International Standards Organization (ISO)* fremdeles som et grunnlag når man skal beskrive datakommunikasjon og kommunikasjonsprotokoller.

Ideen om en lagdelt kommunikasjonsprotokoll har mange fordeler. Ved å sette sammen forskjellige lag i flere kombinasjoner kan man lett utvide en protokoll til å støtte nye typer hardware eller nye tjenester. Figur 2.1 illustrerer OSI modellen. Vi ser en tjener og en klient, og mellom dem en *ruter*. En ruter er en spesialisert enhet hvis oppgave er å sørge for at data sendes til riktig datanett for å nå en mottager. Figuren viser at kommunikasjonssystemet hos tjener og klient består av syv lag. Mellom disse er det et *lag til lag grensesnitt.* All kommunikasjon mellom lagene skal foregå via disse veldefinerte grensesnittene. Applikasjonen kommuniserer ved å overføre data til det øverste laget. Dette laget legger til nødvendig informasjon for å kunne kommunisere med det tilsvarende laget i det andre endesystemet og sender data videre ned gjennom laget under.

De fire øverste lagene kommuniserer direkte med det andre endesystemet. De tre nederste lagene er *node til node* protokoller. Med det mener vi at de inneholder nødvendig informasjon for å få data videre til neste *node* i nettverket. Både endesystemene og ruterne er noder i systemet.

Det at det er så mange som syv lag i hvert endesystem kan lett føre til at prosesseringen av dataene og overføringen av data mellom lagene tar mye av kapasiteten i endesystemet. Dette er trolig en av grunnene til at OSI modellen nå brukes mest for å beskrive datakommunikasjon. Den brukes i til dels svært liten grad som en implementert kommunikasjonsprotokoll slik meningen var. I stedet er *TCP/IP* protokollen nærmest enerådende. Det faller utenfor denne oppgaven å beskrive TCP/IP modellen og årsakene til at denne og ikke OSI-modellen er utbredt. Vi viser derfor i stedet til Tanenbaum[39] dersom leseren ønsker mer informasjon enn dette.

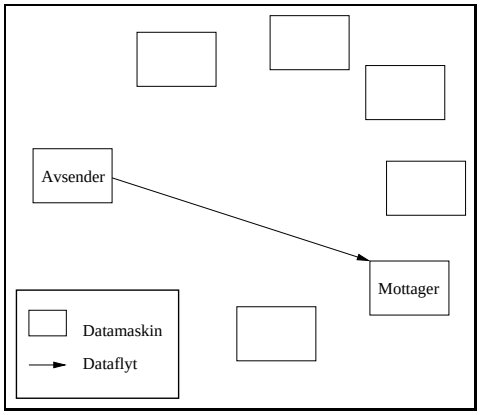
2.1.2 Distribusjonsmetoder

I et datanettverk er det tre grunnleggende måte å distribuere data på. Først har vi *unicast,* også kalt *en til en* kommunikasjon. Unicast betyr at avsender av data sender data til en og bare en datamaskin. Figur 2.2 illustrerer dette. I unicast er det avsender som bestemmer hvor dataene skal sendes.

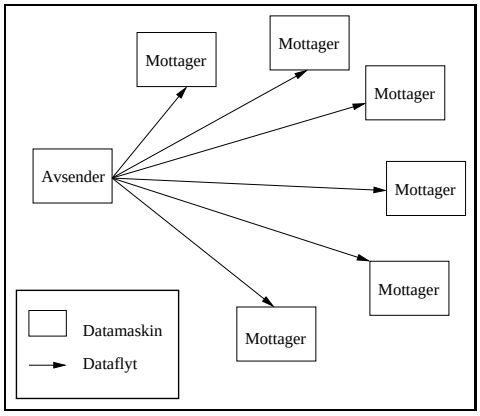
Man kan si at *broadcast,* eller *en til alle* kommunikasjon er det motsatte av unicast. Her sendes data til alle[1] maskinen i nettverket. Figur 2.3 illustrerer broadcast prinsippet. Også i broadcast er det avsender som bestemmer hvor dataene skal sendes.

Det er en klar ulempe med å benytte broadcast. Hvis broadcast benyttes ukritisk vil det raskt sendes mye data til maskiner som ikke er interessert i disse dataene. På bakgrunn av dette er *multicast,* eller *en til mange* kommunikasjon innført. Figur 2.4 illustrerer prinsippet. I multicast er det mottagerne av data som bestemmer om de

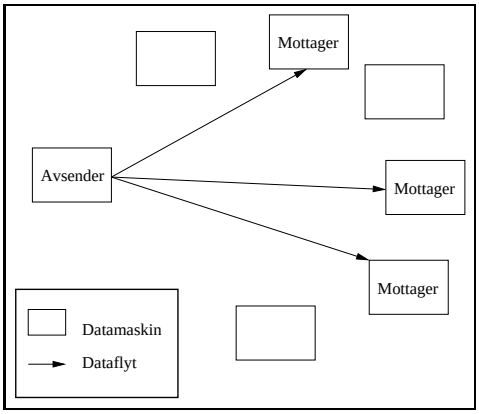
[1] Av kapasitetshensyn gjelder dette normalt alle lokale maskiner. Hvis broadcast meldinger skulle vært videreformidlet til *alle* maskiner i verden ville kapasiteten naturlig nok bli sprengt.



Figur 2.2: Unicast eller “en til en” kommunikasjon



Figur 2.3: Broadcast eller “en til alle” kommunikasjon



Figur 2.4: Multicast eller “en til mange” kommunikasjon

ønsker dataene eller ikke. Avsender sender data til en av flere spesielle adresse beregnet på multicast. Mottageren bestemmer så om de ønsker å motta data på denne adressen. Deering et. al.[18] gir mer informasjon om multicast.

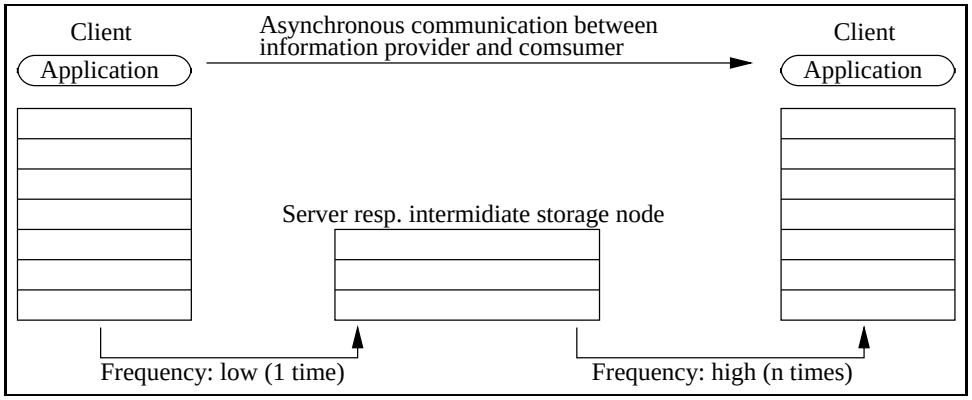
2.2 Idéene bak INSTANCE

INSTANCE står som sagt for «Intermediate Storage Node Concept». Hensikten med INSTANCE er å konstruere en rask og effektiv tjener som er i stand til å støtte *Tjenestekvalitet (Quality of Service, QoS)*[2]. I en vanlig klient-tjener arkitektur sees tjeneren på som et fullstendig endesystem, og må derfor inneholde alle lag i kommunikasjonsprotokollen. Grunnidéen i INSTANCE er å i stedet se på tjeneren, eller INSTANCE-noden, som en node mellom eieren av dataene og klienten:

> «Servers, which are called *intermediate storage nodes* in INSTANCE, are seen as a vehicle to transfer information in asynchronous mode from provider to consumer.»
>
> *Plagemann et. al.[36]*

Den idémessige forskjellen mellom en tradisjonell tjener og en INSTANCE-node er viktig. Det betyr at en INSTANCE-node ikke behøver alle de lag i modellen for kommunikasjonsprotokoller som en tradisjonell tjener har. I figur 2.5 kan man se dette. Vi ser her at en INSTANCE-node kun behøver de tre nederste lagene i protokoll stacken, fra nettverkslaget og nedover. Funksjoner som tilhører lagene over,

[2] Vi beskriver prinsippene i QoS nærmere i kapittel 4.



Figur 2.5: Protokoll håndtering i INSTANCE[36]

som inndeling i nettverkspakker, koder for korrigering av feil i pakkene, konvertering til nettverksformater og andre funksjoner som tilhører de fire øverste lagene gjøres av tilbyderen av data *før* dataene overføres INSTANCE-noden.

En INSTANCE-node er en asynkron server. Asynkron betyr her at det ikke foregår interaktiv kommunikasjon mellom eieren av dataene og klienten. I INSTANCE-noden utnyttes dette til fulle. Når tilbyderen av data "sender" data til klienten, blir alle pakkene lagret på INSTANCE-noden. Pakkene lagres komplette med adresse til mottaker og feilkorrigeringskoder på dataene. Når INSTANCE-noden skal sende data videre, endres bare mottaker-adressen i pakken, og pakken sendes av gårde. Dette er ment å gi god innsparing i den tiden INSTANCE-noden bruker på å behandle pakken i forhold til en tradisjonell server.

2.3 ILP i INSTANCE

Et viktig prinsipp som ligger bak designet av INSTANCE-noden er *Integrated Layer Processing (ILP)*[14]. ILP går kort ut på et brudd med idéen om strengt lagdelte kommunikasjonsprotokoller, slik dette kommer til uttrykk i OSI-modellen med sine syv lag. Clark et. al. skriver:

> *«Most discussions of protocol design refer to the ISO reference model[26], or some other model that uses layering to decompose the protocol into functional modules. There is, however, a peril in using only a layered model to structure protocols into components, which is that layering may not be the most effective modularity for implementation.»*

D. D. Clark et. al.[14]

I ILP ser man på mulige gevinster ved å slå sammen elementer i forskjellige lag som utfører oppgaver på de samme dataene. Et eksempel kan være å slå sammen dekryptering, presentasjonskonvertering og feilkontroll og gjøre dette i en prosess på en hel dataenhet, i stedet for å gjøre dette som tre separate prosesser forskjellige steder i protokollstacken. På denne måten unngår man at dataene sendes fra minnet og til prosessoren i maskinen[3] en gang for hvert lag, for å utføre oppgaver på de samme dataene hver gang. Det å utføre oppgaver tilhørende forskjellige lag i en prosess bryter klart med prinsippene i OSI modellen. I OSI modellen er det n forutsetning at at all kommunikasjon mellom lagene må skje gjennom strengt definert grensesnitt.

2.3.1 Ulemper med ILP

Det kan også være ulemper med prinsippene i ILP. Brustolini et. al.[9] kommer i sitt arbeide med GENIE inn på et problem med ILP. De beskriver problemer i ILP som oppstår idet systemer med ILP skal takle feil i dataene. Ved å slå sammen flere operasjoner, kan man risikere å jobbe med data med feil. Sett f.eks. at en datapakke mottatt med TCP inneholder en bitfeil. Dersom man slår sammen sjekksum undersøkelsen med oppgaver tilhørende applikasjonen vil applikasjonen få dataene før de er sjekket for feil. Derom applikasjonen bruker dataene videre kan vi oppleve uforutsigbare feil. Dette kan gi store problemer, da dette bryter med semantikken for TCP som garanterer at applikasjonen skal motta kun feilfrie data.

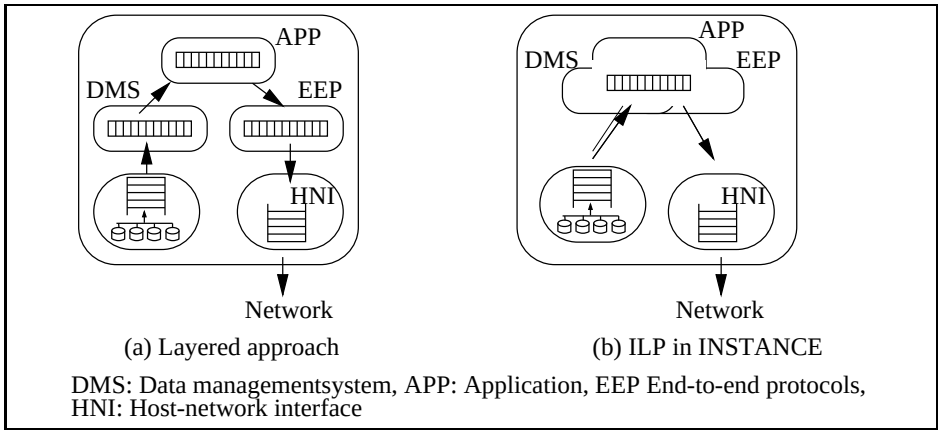
2.3.2 Tanker rundt ILP

Man kan etter vår mening se ILP som en kritikk av OSI-modellen. Tanenbaum[39] sier at i 1990 (da Clark skrev artikkelen om ILP) trodde “alle” at OSI modellen skulle bli den rådende implementasjonen. Vi vet nå at det aldri skjedde, og at OSI modellens kompleksitet trolig var en av grunnene til dette. Sånn sett kan Clarks kritikk i 1990 virke svært forutseende.

2.3.3 ILP og Delsystemene i INSTANCE

I følge Plagemann et. al. er det tre viktige delsystemer i INSTANCE:

[3] Vi beskriver oppbygningen av datamaskinen og delene i den i avsnitt 5.1.



Figur 2.6: Tradisjonell tjenerimplementasjon(a) og ILP i INSTANCE(b) [36]

Datahåndtering systemet (Data Management System). Inneholder databasefunksjonalitet, holder orden på hvilke pakker som tilhører hvilke grupper av pakker osv.

Applikasjonen Håndterer kontakten med brukere. Dette er den applikasjonen den enkelte INSTANCE-node er satt til å håndtere.

Ende til ende protokoll (End-to-end protocol). Sørger for å sende pakkene til mottager. Håndterer evt. resending ved pakketap etc.

I en tradisjonell server ville disse vært adskilt som tre forskjellige prosesser (se figur 2.6(a)). I tråd med prinsippene i ILP, slås datahåndterings systemet, applikasjonen og ende til ende protokollen sammen i en prosess (se figur 2.6(b)). På denne måten spares prosesseringstid ved at man reduserer antallet kopierings operasjoner fra buffer til buffer inne i maskinen. I tillegg sparer man noe av den tiden det koster å bytte fra prosess til prosess, såkalt context switching.

2.4 Bufferhåndteringen

I tradisjonelle OS-er overføres data først fra disk til et diskbuffer i kjernen av operativsystemet. Så kopieres data fra kjernen til minnet til applikasjonen som håndterer dataene. Senere kopieres data fra applikasjonen til nettverksdelen av OS-et. Til slutt nok en kopiering over til nettverkskortet. Minnehåndteringen i INSTANCE er ment å basere seg på *zero-copy buffers*. Zero-copy buffring betyr at data ikke kopieres fra minneområde til minneområde intern i INSTANCE-noden. Data leses fra disk og inn i et buffer. Dataene skal *ikke* kopieres videre derfra før det overføres til nettverkskortet og ut på nettverket.

2.5 Oppsummering

For å oppsummere er en INSTANCE-node en asynkron server, optimalisert for hurtighet og med mulighet for å støtte QoS. Vår oppgave er å komme frem til en minnehåndteringsmekanisme som passer inn i dette konseptet. For å komme dit vil vi videre i oppgaven først gå inn på innholdet INSTANCE-noden skal formidle, multimedia og MoD. Videre ser vi på hva som ligger i begrepet QoS. Til sist i bakgrunnsdelen av oppgaven ser vi på prinsipper for bufferhåndtering og tidligere arbeider med zero-copy minnehåndtering.

Idéene om lagring av nettverkspakker på disk, sammenslåing av feilkorrigering på disk og i nett og nye idéer for nettverksprotokollene er det dessverre hverken tid eller plass til å behandle på en god måte i denne oppgaven. Dette er oppgaver som undersøkes av andre og derfor publiseres av disse. Disse løsningene var heller ikke klare idet denne oppgaven ble gjennomført. Vi nøyer oss derfor med å lage en bufferhåndteringsmekanisme basert på de delene av INSTANCE-konseptet som lå til grunn ved begynnelsen av oppgaven. Selvsagt ønsker vi å lage en buffermekanisme som egner seg til å passes inn med de andre ikke fullførte delkonseptene. Vi ønsker å legge til rette for dette men overlater imidlertid selve dette arbeidet til andre som kommer etter oss.

Kapittel 3

Multimedia on Demand

Multimedia on Demand (MoD) er et relativt nytt begrep. Ordet bygger videre på uttrykket *Video on Demand (VoD)* som beskriver et system Der en datamaskin distribuerer video til klienter på forespørsel. MoD brukes her om en tjener som gir MM-data til klienter på forespørsel.

I dette kapittelet går vi først inn på hva multimedia er. Avsnitt 3.1 inneholder to forskjellige definisjoner på MM og en liten diskusjon omkring hva vi mener med begrepet. I avsnitt 3.2 ser vi på hvilke type data MM består av og ser på egenskapene til disse. Etter å ha sett på hva MM er går vi i avsnitt 3.3 nærmere inn på hva som ligger i begrepet Multimedia-on-Demand. Etter dette følger avsnitt 3.4 hvor vi ser på forskjellige prinsipper for distribusjon av MM-data. Vi avslutter kapittelet med avsnitt 3.5 der vi ser på MoD og INSTANCE i sammenheng. Vi ser på hvilke typer data INSTANCE-noden kan egne seg til og hvilke krav MoD stiller til designet av INSTANCE-noden.

3.1 Hva er multimedia?

Multimedia er et relativt nytt ord. Det er et relativt vidt begrep, og det eksisterer flere forskjellige definisjoner av ordet. Steinmetz et. al.[38] definerer MM i en datamaskinkontekst:

> *«Multimedia means, from the user's perspective, that computer information can be represented through audio and/or video, in addition to text, image, graphics and animation.»*
>
> *Steinmetz et. al.[38]*

Vi forstår denne definisjonen til å gjelde det som typisk selges som MM til en datamaskin som for eksempel MM-leksikon. Et MM-leksikon inneholder gjerne tekst, bilder, video og lyd. Altså en kombinasjon av mange forskjellige media. En ulempe med definisjonen til Steinmetz et. al. er at den beskriver MM *kun* sett i lyset av datahåndterte media. Det passer for såvidt bra siden vi i denne oppgaven beskriver MM i datamaskiner, men det er også interessant og se MM mer i et generelt perspektiv. The New International Webster Comprehensive Dictionary[37] har er mer generell definisjon på MM:

> «**mul·ti·me·di·a** (mul'ti·mē'dē·ə) *adj. Relating to or using two or more media, especially a combination apprehended by different senses, as sight and hearing.»*
>
> *The New International Webster Comprehensive Dictionary[37]*

Begge disse definisjonene er etter vår mening relativt vage. Det går klart frem av begge at MM er en *kombinasjon* av forskjellige media. I henhold til Websters definisjon kan man faktisk definere en videofilm som MM, siden den inneholder *«. . . a combination [of media] apprehended by different senses, as sight and hearing.»*. Steinmetz et. al. krever derimot minst et media i *tillegg* til video *«. . . audio and/or video, in **addition** to text, image, graphics and animation.»*.

Vi velger å legge oss et sted midt i mellom disse definisjonene. Vi ønsker å kunne tilby MM i tråd med definisjonen til Steinmetz et. al. I tillegg ser vi for oss at en INSTANCE-node godt kan brukes til å distribuere bare musikk, eller bare video. Selv om *bare* musikk eller video ikke er MM i streng forstand. Det viktige her er at INSTANCE-noden kan fylle de krav distribusjon av slike data stiller til systemet.

3.2 Datatyper i multimedia

Som vi så i forrige avsnitt må en MoD-tjener være i stand til å distribuere datatyper som bilder, tekst, video og lyd. Vi vil i dette avsnittet gå nærmere inn på disse datatypene og se på egenskaper ved disse. Disse egenskapene legger premisser for det videre arbeidet i oppgaven, da det er en forutsetning at INSTANCE-noden skal være i stand til å distribuere slike data med en tilfredsstillende kvalitet.

3.2.1 Kontinuerlige og diskrete datatyper

Datatypene deles gjerne inn i *kontinuerlige* og *diskrete* datatyper. Kontinuerlige datatyper er data som varer over tid, for eksempel video, lyd og animasjoner. Diskrete

datatyper er datatyper hvor tid ikke er en faktor. Eksempler på diskrete datatyper er tekst og bilder. Også diskrete datatyper kan i MM-sammenheng være tidsavhengige. For eksempel kreves det at underteksten til en videofilm vises til bestemte tidspunkt. Uansett stiller allikevel kontinuerlige og diskrete datatyper forskjellige krav til en MoD-tjener. Kontinuerlige datatyper må sendes ut mer eller mindre kontinuerlig, mens diskrete datatyper gjerne sendes ut enkeltvis med noe tid mellom. Mer om dette i avsnitt 3.2.3.

3.2.2 Komprimering av data

Datatyper som lyd og video krever til dels svært mye lagringsplass. Dette gjør det kostbart å lagre og ikke minst tar det relativt sett svært lang tid å overføre slike datatyper gjennom et nettverk. Det er derfor etterhvert laget flere metoder for å pakke, eller *komprimere* dataene slik at de tar mindre plass.

Valget mellom komprimerte og ukomprimerte formater er et valg mellom lagringskapasitet og båndbredde på den ene siden og prosessorkraft på den andre. Det å komprimere data krever som regel mer kapasitet enn det å dekomprimere dem. Men mens komprimering kan gjøres i forkant og dermed ikke er tidsbegrenset, må dekomprimering ved avspilling skje i sanntid dersom dataene skal dekomprimeres idet de presenteres. Dermed er det dekomprimeringshastigheten som er avgjørende i et system der dataene gjøres klare på forhånd.

For få år siden var selv relativt kraftige arbeidsstasjoner ikke kraftige nok til å vise komprimert video med en akseptabel datarate. Dette fordi dekoding av komprimerte data nødvendigvis tar prosessorkraft, og slik kraft ikke var tilgjengelig for normale brukere. Selv om det den gang var enda dårligere med lagringskapasitet og nettverksbåndbredde var visning av komprimerte data forbeholdt til dels svært dyre maskiner med spesielt raske og dyre nettverk og dyre lagringsmedia. I de siste årene har utviklingen av prosessorene nå kommet så langt at selv en vanlig hjemmemaskin har nok kraft til å vise fullskjerm komprimert video med full datarate på 25 bilder per sekund. Selv sett i lys av dagens store harddisker tar ukomprimerte MM-data tar svært mye plass. Komprimeringen muliggjør lagring av større mengder av video og lyd. I praksis er derfor nå de aller fleste rådende formater for video og lyd i datamaskiner komprimert.

Det finnes to typer komprimering, *lossy* og *loss-less* komprimering. Lossy komprimering betyr at data går tapt under komprimering, loss-less at de originale data kan gjenskapes fullstendig. Høyest grad av komprimering får man gjerne ved å bruke lossy komprimering. Å tape informasjon som i stor grad forringer den subjektive kvaliteten virker lite aktuelt. Det er i stedet utviklet komprimeringsteknikker som tar utgangspunkt i hvordan vi mennesker oppfatter lyd og bilde. Ved å fjerne detaljer som det antas at den vanlige lytter eller seer ikke ville lagt merke til, oppnår man en høy grad av komprimering.

Tabell 3.1 i avsnitt 3.2.4 kommer en oversikt over forskjellige dataformater. Der er det beskrevet både komprimerte og ukomprimerte formater og deres datarater.

3.2.3 Variabel eller konstant bitrate

Datastrømmer bestående av MM-data har gjerne en ujevn bitrate. Med det mener vi at mengden data som sendes ut per tidsperiode, for eksempel datamengden per tiendedels sekund, varierer. En slik datastrøm med varierende datamengder kalles en datastrøm med *variabel bitrate (Variable Bit Rate, VBR)*. En datastrøm der bitraten ikke varierer, kalles en datastrøm med *konstant bitrate (Constant Bit Rate, CBR)*.

De er to vanlige grunner til at en datastrøm bestående av MM-data har VBR egenskaper. Den ene er at komprimeringsteknikkene som brukes for å komprimere MM-data ofte gir VBR strømmer. Videokomprimering med MPEG er et godt eksempel på dette. Det kreves mer data for å vise et bilde fult av bevegelse som et billøp, enn å vise et stillestående bilde av en svart nattehimmel. Forklaringen er at dersom et bilde i en videostrøm er helt likt det forrige, trengs det egentlig ikke mer informasjon enn at dette bildet er likt det forrige. Man trenger ikke tegne opp bildet på nytt. Et bilde i en film om et billøp skiller seg gjerne mye fra det forrige bildet fordi filmen er full av bevegelse. Datastrømmen må dermed inneholde data om hvordan alle bilene ser ut og hvor de er plassert. Den andre grunnen til at en datastrøm med MM-data har VBR egenskaper er dersom den inneholder elementer av diskrete datatyper. Dersom datastrømmen består av bakgrunnsmusikk til stillbilder som vises med jevne mellomrom, vil dataraten gjerne øke hver gang et nytt stillbilde blir overført.

Vi kan dermed si at svært mange datastrømmer med MM-data får VBR egenskaper. Det vil derfor etter vår mening være en forutsetning at buffermekanismen i INSTANCE-noden støtter VBR datastrømmer.

3.2.4 Eksempler på dataformater

I dette avsnittet tar vi for oss eksempler på dataformater for lyd og bilde. Tabell 3.1 gir en grov oversikt over forskjellige datatyper og deres egenskaper. Det understrekes at dataratene, med unntak av dataraten for Audio-CD, er anslag. For de alle datatyper komprimert med lossy komprimering blir dataraten et valg ved komprimering basert på en avveining mellom hvor mye man taper og hvor høy/lav datarate man ønsker.

[1] For datatyper med variabel bitrate blir dataraten bare et anslag.

[2] Anslag basert på tall fra Steinmetz et. al.[38]. Går ut fra et interlaced PAL bilde (annenhver linje vises 50 ganger i sekundet). Med 575 aktive linjer og en horisontal oppløsning på 425, får vi

Format	Type	Datarate[1]	VBR/CBR
Ukomprimert TV	Video + lyd	ca. 100 Mb/s[2]	CBR
MPEG-2	Video + lyd	ca. 2–4Mb/s[3]	VBR
DVD	Video + lyd	ca. 7–9Mb/s[4]	VBR
DivX[5]	Video + lyd	ca. 1Mb/s[6]	VBR
Audio CD	Lyd	ca. 1,4Mb/s[7]	CBR
MP3	Lyd	112–128kb/s[8]	CBR

Tabell 3.1: Oversikt over egenskaper ved kontinuerlige dataformater

3.3 Multimedia Systemer

For at det skal være en hensikt med en MoD-tjener må vi naturligvis ha tilsvarende klienter som kan håndtere MM. Både tjener og klient er etter definisjonen et MM-system. Steinmetz et. al.[38] gir følgende definisjon på et MM-System:

> *«A multimedia system is characterized by computer-controlled, integrated production, manipulation, presentation, storage and communication of independent information which is encoded at least through a continous (time-dependent) and a discrete (time-independent) medium.»*
>
> *Steinmetz et. al.[38]*

Av denne definisjonen går det frem at et MM-system *må* tilby både tidsavhengige data som video og lyd *og* tidsuavhengige data som for eksempel tekst, bilder med

244.375 billedpunkter 25 ganger per sekund. Benytter vi 16 bits/pixel fargeoppløsning og 25 bilder per sekund får vi 97.750.000b/s. I tillegg til dette kommer ukomprimert digitalisert lyd. Vi anslår dermed grovt dette til totalt ca. 100Mb/s.

[3] PAL-TV med lyd. Bitraten varierer ettersom hvor kraftig komprimeringen er.

[4] Bitraten varierer med komprimeringsgraden.

[5] DivX[20] er et format for komprimering av video/lyd med en etter forholdene subjektivt meget bra kvalitet i forhold til bitrate. DivX er basert på MPEG-4 standarden. Standarden er etter vår mening dårlig dokumentert, og brukes mye til piratkopiering av video over Internett. Derav muligens den dårlige dokumentasjonen. Fordi den har så bra forhold komprimering/kvalitet har vi likevel valgt å ta den med her. Kvaliteten oppgis å tilsvare SVHS/DVD.

[6] Anslag fra på data fra DivX Digest[20]. Oppgitt som ca. bitrate ved en oppløsning på 640x480.

[7] 44,1kHz stereo, 16bit/kanal

[8] Bitraten bestemmes ved koding, 112 og 128kb/s er de vanligste dataratene ved 44,1kHz og joint-stereo koding.

mere. En MoD-klient må tilfredsstille disse kravene for å kunne kalles et MM-system. Vi kommer senere i oppgaven inn på at tidsavhengige data krever en annen behandling i tjeneren enn tidsuavhengige data. Av den grunn kan det være fornuftig å bruke forskjellige mekanismer for å håndtere henholdsvis tidsavhengige og tidsuavhengige data. Muligens kan det være en idé å gå så langt som til å la to forskjellige tjenere samtidig tilby henholdsvis tidsavhengige og tidsuavhengige data til en og samme klient.

Dagens OS-er har tradisjonelt vært designet for å håndtere tidsuavhengige data på en god måte. De har i liten grad vært designet for å håndtere tidsavhengige data. Så mens håndtering av tidsuavhengige data i slike OS-er etterhvert er blitt meget bra, er det mange problemer forbundet med distribusjon av video, lyd og andre typer tidsavhengige data. Vi har derfor i vårt arbeide med INSTANCE-noden konsentrert oss om å få noden til å håndtere tidsavhengige data og venter med vurderingen om INSTANCE-noden også skal levere tidsuavhengige, eller om flere tjenere skal samarbeide om å levere data til en MoD klient.

3.4 Distribusjon av MM-data i MoD

Det er etterhvert laget en rekke forskjellige systemer for MoD og VoD. De forskjellige systemene bruker forskjellige teknikker for å distribuere dataene. Disse teknikkene kan deles inn i tre hovedgrupper:

Klient - Tjener MM-data overføres fra tjener til klient. Distribusjonen er oftest basert på unicast. Denne teknologien er sentrert rundt brukeren. Brukeren kontakter Tjeneren og initierer overføring.

Broadcasting/multicasting MM-data broadcastes eller multicastes til de som er interessert. Denne teknikken er sentrert rundt dataene. En eller helst flere brukere mottar dataene slik de blir sendt ut.

Parallell samtidig Broadcasting En datastrøm med MM-data splittes opp i flere delstrømmer som sendes ut i parallell, slik at forskjellige deler av datastrømmen kringkastes samtidig. Brukeren henter først inn begynnelsen av dataene fra en delstrøm, så fortsettelsen fra andre delstrømmer inntil alle data er mottatt. Dataene buffres gjerne hos klienten og vises mens de mottas.

I de neste avsnittene beskriver vi forskjellige idéer for MoD og VoD distribusjon. De forskjellige prosjektene benytter gjerne en eller flere av de ovennevnte teknikkene.

3.4.1 Tradisjonell Video on Demand

Tradisjonell VoD er basert på klient - tjener prinsippet. Tjeneren mottar ønsker fra klienten, og dersom tjeneren har ressurser til det åpnes en datastrøm fra tjeneren til klienten. Data leses fra disk og sendes over nettverk til klienten.

Tradisjonell VoD gir grunnlag for støtte for god brukerkontroll. Siden dataene går til bare en klient er det mulig å lå klienten be tjeneren om fremoverspoling, bakoverspoling og pause av dataene. Ulempen med tradisjonell VoD er at store ressurser brukes på hver klient. Dermed er kostnaden per klient stor.

3.4.2 Near Video on Demand

Near Video on Demand (NVoD) er en tjeneste som gir tilnærmet VoD egenskaper, men med en lavere resursbruk i tilfeller der flere klienter ønsker samme data til tilnærmet samme tid [17]. NVoD er basert på prinsippet om broadcasting/multicasting av MM-data. Ideen er å starte avspilling av video til faste tider, og la alle som ønsker det koble seg til en slik datastrøm på det første mulige tidspunktet etter at de har bedt om tjenesten.

Fordelen med NVoD er at man bruker mindre ressurser enn ved VoD dersom flere klienter mottar samme datastrøm. Ved bruk av multicast teknikk bruker man minst mulig nettressurser. En broadcast eller multicast datastrøm kan i prinsippet betjene et ubegrenset antall klienter. Ulempene men NVoD er at man reduserer QoS. Brukeren kan ikke starte avspilling når han/hun måtte ønske det. I tillegg mister man muligheten for interaktiv bruk som hurtigspoling, hopp frem og tilbake og pause. Pause og tilbakespoling kan tilbys dersom klienten lagrer alle data den mottar, men fremoverspoling er ikke mulig med en ren NVoD løsning. Vi har også et vanskelig valg mellom å sende mange datastrømmer i parallell, eller at brukeren kan risikere å vente lenge på å motta dataene dersom avspillingen starter med lange mellomrom.

3.4.3 Quasi Video on Demand

Quasi Video on Demand (QVoD) er basert på NVoD, så det er således også en broadcasting/multicasting teknikk. Abram-Profeta et. al.[1] forslår en mer fleksibel politikk for når avspillingen av en film skal starte. QVoD ligner NVoD, men i stedet for å starte avspilling til bestemte tider, starter avspilling når et tilstrekkelig antall har meldt seg for å se videoen. QVoD kan konfigureres til å starte hvis første person som ønsker å se filmen har ventet for lenge, eller nå et visst antall har meldt seg, eller en kombinasjon av dette. På denne måten kan man sette opp en mer detaljerte politikk for når avspillingen skal starte. Vi kan si at populære datastrømmer

distribueres med en bedre QoS enn tilfellet gjerne er for NVoD ved at QVoD kan konfigureres til å sende disse ut med en kortere tidsintervall. Ulempen er at klientene som ønsker mindre populære strømmer må vente lenger og slik for en dårligere QoS.

3.4.4 Greedy Diskconserving Broadcasting

I likhet med NVoD og QVoD er *Greedy Diskconserving Broadcasting (GDB)* fra Gao et. al.[22] en metode for multi eller broadcasting av datastrømmer. GDB bygger på *Pyramid Broadcasting*[41] og *Skyscraper Broadcasting*[24].

GDB skiller seg fra QVoD og NVoD ved måten dataene kringkastes på. I NVoD og QVoD kringkastes datastrømmene i sin helhet fra begynnelse til slutt og i samme hastighet som ved avspilling. GDB deler datastrømmene opp i understrømmer ved å for eksempel la de to første sekundene utgjøre en datastrøm, de tre neste sekundene enn annen datastrøm, tredje datastrøm fire og en halv sekunder og så videre. GDB overfører så understrømmene raskere enn fremvisnigshastigheten. For eksempel kan en understrøm på to sekunder overføres med 50% økt hastighet, det betyr at den overføres på $1^1/_3$sekund. Hver understrømmene sendes ut periodisk i en kanal, der en kanal for eksempel kan være en mulitcastadresse. Nedlastet data lagres hos mottaker, gjerne på disk (derav navnet Greedy *Diskconserving* Broadcasting). Siden overføringen går raskere enn avspillingen, vil det mens den første pakken på to sekunder spilles av være tid til å laste ned neste pakke på tre sekunder.

Fordi en datastrøm sendes flere ganger i parallell, med høyere hastighet enn ved avspilling blir lasten høyere enn å sende dataene ut en gang med normal hastighet. GDB er derfor beregnet på strømmer som blir aksessert mye, typisk strømmer med flere titalls samtidige mottagere.

Fordelen med GDB er at ventetiden fra klienten bestemmer seg for å motta en datastrøm, til avspilling begynner er liten. Med en understrøm med to sekunder varighet som sendes ut i løpet av $1^1/_3$sekund vil klienten i verste fall vente opp mot $1^1/_3$sekunder før han har begynner å motta første understrøm og kan begynne avspillingen. Hvis vi skal tilby samme responstid med QVoD eller NVoD ville vi måtte sende en ny datastrøm hvert $2^2/_3$sekund. Som Gao et. al.[22] viser er GDB mer effektiv enn om tilsvarende responstid ved bruk av NVoD og mange klienter.

3.5 Valg av strategi i INSTANCE

I dette avsnittet ser vi MM-data, MoD og INSTANCE-konseptet i sammenheng. Vi vurderer hvilke datatyper INSTANCE-noden kan egne seg best for. Vi ser også

på hvilken eller hvilke strategier for distribuering av MoD som egner seg best for INSTANCE-noden.

3.5.1 Datatyper i INSTANCE

Hvilke datatyper er det så vi vil forvente å håndtere i en INSTANCE-node?. Alle trender går nå i retning av mer og mer utstrakt bruk av komprimerte formater. Komprimert video og lyd, komprimert lyd alene og komprimerte data som del av multimedia presentasjoner. Dette er en naturlig konsekvens av at prosessorkapasiteten i datamaskinene har blitt så stor at real-time dekoding av komprimerte formater er mulig.

Det er et spørsmål om vi ønsker å benytte INSTANCE-noden også til distribusjon av tekst og bilder. Foreløpig konsentrer vi oss om distribusjon av forskjellige former for kontinuerlige datastrømmer. Årsaken er at dette forenkler beregninger av ressursbruk slik at det er lettere å tilby tjenester med god kvalitet. I tillegg eksisterer det allerede mye forskning på optimalisering av distribusjonen av diskrete data som tekst og bilder. Caching av data i *WWW-proxies*[23][28] har vist seg effektivt ved å lagre kopier av populære data nære klienten. Vi velger derfor å konsentrere arbeidet med bufferhåndteringen i INSTANCE-noden til å håndtere kontinuerlige datastrømmer på en god måte. Ettersom mer og mer data får VBR egenskaper vil det være en forutsetning at mekanismen for bufferhåndtering er i stand til å håndtere slike strømmer.

3.5.2 MoD teknologi i INSTANCE

Valg av MoD teknologi blir en avveining mellom kvalitet på tjenesten frem til brukeren og antallet samtidige brukere og kostnad per bruker. Dersom vi har et lite antall brukere per datastrøm gir tradisjonell tradisjonell VoD meget bra kvalitet til brukeren. Dersom vi har et større antall klienter, konkluderer Dan et. al.[17] og Bär et. al.[3] med at det er en stor sannsynlighet for at noen datastrømmer er langt mer populære enn andre. Disse rapportene baserer sine konklusjoner på statistikk over utleie av videofilmer. Etter vår mening ville det trolig derfor være en god idé å bruke en type teknologi for å distribuere de mest populære data og en annen for å distribuere mindre populære data. Hva som er mest effektivt ved flere samtidige brukere er avhengig av antallet brukere. Ved svært mange samtidige brukere vil GDB gi den beste kombinasjonen av rask responstid og lav ressursbruk. Ved et mindre antall vil QVoD muligens egne seg bedre.

Alle typer MoD strømmer med kontinuerlige datatyper gir opphav til kontinuerlige datastrømmer. Vi konsentrer oss derfor om å lage en buffermekansime for kontinuerlige datastrømmer og overlater den endelige beslutningen om valg av distribusjonsstrategi til de som til slutt setter de forskjellig konseptene sammen til den

endelig INSTANCE-noden. Vi mener allikevel at GDB virker som en meget aktuell teknikk med flere samtidige klienter og velger å ta utgangspunkt i denne i det videre arbeidet. Løsninger vi utvikler for bruk med kontinuerlige datastrømmer i GDB vil trolig enkelt kunne tilpasses til andre distribusjonsmetoder.

Kapittel 4

Tjenestekvalitet

Tjenestekvalitet (Quality of Service, QoS) er et viktig begrep ikke minst når man snakker om lyd og bilde i nettverksbaserte tjenester. Ikke fordi QoS ikke er viktig i andre sammenhenger, men fordi det det gjerne først er når QoS blir for dårlig at brukeren interesserer seg for dette. Dette skyldes at det kan være vanskeligere å tilby QoS når dataene skal transporteres gjennom et nettverk. Kontinuerlige datastrømmer av lyd og video sent gjennom et nettverk krever mye ressurser både i endesystemene og i nettverket. Dersom disse ressursene ikke finnes, for eksempel ved at det ikke er kapasitet i nettverket til å overføre alle dataene, kan resultatet bli hakkede bilde og og forvrengt lyd når dataene vises for sluttbrukeren. Dette er et eksempel på dårlig QoS. Det er laget flere definisjoner på QoS. Vogel et. al.[42] bruker følgende definisjon:

> *«Quality of service represents the set of those quantitative and qualitative characteristics of a distributed multimedia system necessary to achieve the required functionality of an application.»*
>
> Vogel et. al.[42]

QoS er også standardisert som en del av OSI modellen. I ISO's standard for OSI modellen står følgende om QoS:

> *«Quality of Service (QoS) is the collective name given to a set of parameters associated with (N)-data transmission among (N)-service-access-points.»*
>
> ISO Standard ISO-7498 Del 1, avsnitt 5.10.1.1[26]

Vi vil nå videre gå mer inn på detaljene i QoS. Avsnitt 4.1 ser vi på grunnleggende prinsipper for QoS. Vi så på QoS i MoD-sammenheng i avsnitt 4.2. Etter dette ser vi i avsnitt 4.3 på QoS i nettverk og på mulighetene for å reservere ressurser i nettverket. Vi avslutter med avsnitt 4.4 der vi ser på hvilke muligheter vi har til å støtte QoS i INSTANCE-noden.

4.1 QoS prinsipper

For å kunne beskrive QoS trengs det først en del begreper som et rammeverk. Dette rammeverket av begreper er nyttig når vi siden skal definere elementene i QoS mer i detalj. Vi går derfor først inn på QoS-prinsipper som spesifikasjon av QoS, forhandling, semantikk og adgangskontroll.

4.1.1 Spesifikasjon

Et problem når det gjelder QoS er å avgjøre hva som er god og hva som er dårlig QoS.

For brukeren er det for såvidt den subjektive QoS-en som er den viktigste. Han eller hun har gjerne en klar formening om at lyden er for dårlig, bildet for kornete eller flyten i en video for dårlig. Ulempen med denne typen spesifikasjon er at slike krav er vanskelig å implementere. Brukerne har også ofte forskjellige ønsker. På grunn av dette er det vanlig å spesifisere QoS i mer målbare størrelser.

4.1.2 Forhandling

En faktor i bestemmelse av QoS kan være forhandling om kvaliteten. Et eksempel på dette møter vi ofte hvis vi laster ned video fra Internett i dag. Ofte blir man møtt med et valg om å se videoen i forskjellige oppløsninger. Man velger så den oppløsningen som passer for båndbredden man sitter på, slik at nedlastingen ikke skal ta fryktelig lang tid. Her foregår det en forhandling ved å tjeneren tilbyr forskjellige QoS-egenskaper og klinten velger det sett egenskaper som passer best for klienten.

4.1.3 Semantikk

Dersom man har kommet frem til et resultat i QoS forhandlingene, kommer spørsmålet, hva skjer så. Kan tjeneren garantere den QoS som er tilbudt? Dette er spørsmål om *Semantikken* i QoS beskrivelsen. Med semantikken mener vi ting som: Hva betyr en QoS spesifikasjonen i praksis? Vil vi virkelig motta den kvaliteten vi er lovet? Vi opererer med forskjellige typer QoS-semantikk[]:

Garantert QoS Det hardeste kravet. Vi *skal* motta den kvaliteten vi er forespeilet.

Best Effort QoS Oversatt til norsk betyr dette "så bra som mulig". Tjeneren gjør sitt beste for å oppfylle kravene, men kan ikke garantere noe.

Tvungen QoS Tjeneren gjør alt den kan for å garantere kvaliteten. Dersom kvaliteten faller under kravet, brytes forbindelsen.

Terskel QoS Nesten som tvungen QoS, men når kvaliteten faller under kravet gis i stedet en melding til klienten om dette. Det er så opp til klienten om den vil bryte forbindelsen eller ikke.

Maksimal QoS Ved maksimal QoS er det klienten som stiller krav om at den ikke ønsker *mer* kvalitet enn et visst mål. Et eksempel kan være at man ikke ønsker høyere datarate enn det en oppringt modemlinje kan håndtere. En annen at man ikke ønsker å betale mer enn for et vist QoS nivå.

4.1.4 Allokering og adgangskontroll

For å kunne garantere QoS i tjenere, nett og mottakere kan det være nødvendig å allokere nødvendige ressurser. Minne, båndbredde i nett, båndbredde til disk, alt dette og mere er ressurser som er nødvendige for å kunne yte en tjeneste. Dersom man skal garantere QoS, eller gjøre et godt forsøk på dette er det aktuelt å beregne ressursforbruket for en forbindelse og så reservere dette. Det finnes teknologier for allokering i endesystemet.

Hånd i hånd med allokering kommer adgangskontroll. Hvis en forbindelse behøver en ressurs som ikke er tilgjengelig, må enten kvaliteten senkes på denne eller andre forbindelser, eller forbindelsen må avvises. Ofte velger man det siste alternativet dersom man operer med garantert QoS. Er det ikke nok ressurser avvises oppkoblingen. Et problem er adgangskontroll med variabel bitrate. Vin et. al.[40] har foreslått statistiske metoder for adgangskontroll i en tjener som distribuerer VBR-data. Izquierdo et. al.[27] har flere forslag til statistisk analyse av VBR datastrømmer.

4.2 Multimedia on Demand og tjenestekvalitet

Til nå har vi beskrevet QoS med teoretiske begreper. Så kommer spørsmålet: Hva betyr egentlig QoS i praksis? I de neste avsnittene skal vi gi eksempler på hvilke QoS kriterier vi opplever i en MoD applikasjon.

4.2.1 Jitter

Jitter beskriver “jevnheten” i en datastrøm. Hvis vi ser på video er det viktig at bildene kommer i jevnt tempo, hvis ikke opplever vi hakking i bildet, og at bildet “henger”. Liten jitter betyr at dataene ankommer i jevn hastighet, høy jitter at dataene ankommer i blokker med ujevne mellomrom. Jitter kan tallfestes som maksimal avstand mellom tiden dataene skulle ankommet, og det tidspunkt de faktisk ankommer.

En vanlig måte å motvirke jitter er å lagre dataene hos klienten en periode før de vises, også kalt “buffring”. Dette betyr at data fra en strøm med høy jitter allikevel kan vises frem i et jevnt tempo. Ulempen er at det tar tid mellom dataene mottas og de vises.

4.2.2 Bildekvalitet

Bildekvalitet er et vanskelig begrep å tallfeste. Bildekvalitet betyr hvor “pent” bildet ser ut. Dårlig bildekvalitet kav være bilder som “hakker”, “blokkete” bilder eller at farvene er “gale”. Det er vanskelig å fastslå objektive kriterier for bildekvalitet, men dette er allikevel gjerne det viktigste kriteriet for sluttbrukeren.

En målbar faktor for bildekvalitet er oppløsning. Antall bildepunkter i bredden ganger antall bildepunkter i høyden. Mange anser dårlig oppløsning som dårlig bildekvalitet. Oppløsning er allikevel ikke nødvendigvis et godt nok mål for bildekvalitet. Moderne kompresjonsteknikker som MPEG komprimer ved å til en hvis grad slå sammen punktene. Spesielt i bilder med mange detaljer kan man se at bildekvalitet kan variere selv om antallet punkter er de samme.

4.2.3 Responstid

Responstid er et QoS kriterie vi ofte legger merke til. Hvor lang tid tar det fra vi trykker på knappen til ønsket reaksjon skjer? En av de viktigste responstidene er oppstartstiden. Tiden det tar fra vi ønsker å sette i gang avspilling til avspilling faktisk skjer.

4.2.4 Brukerinteraksjon

Brukerens mulighet til å påvirke informasjonen han/hun mottar kan sees som en del av QoS-en for MoD. Dette innebærer hvorvidt brukeren gis mulighet til å spole frem og tilbake i videostrømmer, om brukeren kan påvirke synkroniseringen av data.

Brukerpåvirkning gir en rekke problemer i forhold til andre QoS parametere. Dette skyldes at brukerinteraksjon i de fleste tilfeller innebærer endret ressursbruk. I de tilfeller det ressursbruken minskes vil dette ikke være et problem. Økes derimot ressursbruken, vil dette være en utfordring å håndtere.

4.3 Nettverk og tjenestekvalitet

Det å tilby QoS gjennom et pakkesvitsjet nettverk er problematisk. I praksis er støtten for QoS i protokoller som IP liten eller ingen. Det er derfor et behov for andre løsninger for kunne reservere ressurser i nettverket dersom nettverks-QoS skal kunne tilbys.

Zhang et. al.[44] introduserte i 1993 reservasjonsprotokollen Resource ReSerVation Protocol (RSVP). RSVP protokollen er senere akseptert som en serie med *request for comment (rfc)*[5][29][4] av *Internet Engineering Task Force (IETF)*[1]

I motsetning til et forbindelsesorientert nettverk som et telefonnett, gir et pakkesvitsjet nettverk som Internett tradisjonelt ingen garanti for at data kommer frem til mottager. I et telefonnett opprettes det en forbindelse når man ringer et nummer, og båndbredde reserveres i begge retninger hele veien mellom samtalepartnerne. I et nett som Internett opprettes det ingen slik forbindelse, hver pakke rutes i stedet etter beste evne fra avsender til mottaker. Dersom pakken ikke kommer frem er det opp til avsender å sende en ny pakke. RSVP gir brukeren av en tjeneste muligheten til å reservere ressurser i nettet. På den måten kan man få bra QoS også i et pakkesvitsjet nettverk.

RSVP kan brukes til å reservere ressurser i en *en til en* forbindelse der det kun er en avsender og en mottaker. Ressurser reserveres og bekreftelse gis tilbake til den som ba om reserveringen. Reserveringen gjelder bare en vei, så dersom det ønskes reservering begge veier, må begge parter reservere ressurser. RSVP støtter også reservering i *en til mange* forbindelser som ved bruk av multicast.

4.4 Tjenestekvalitet i INSTANCE

Det er et klart mål at INSTANCE-noden skal støtte QoS[36]. Det betyr at bufferhåndteringen på en eller annen måte må kunne tilby støtte til et visst QoS nivå. Vi mener det vil være lettere å tilby god QoS dersom vi lager en effektiv. Å lage en effektiv bufferhåndteringsmekansime blir dermed et klart mål for denne oppgaven.

[1] IETF er en interesseorganisasjon for utvikling av Internett[25]. En rfc er det nærmeste man kommer en standard for Internett og internettprotokollene. Blant standardene fastslått i rfc-er er kjente protokoller som TCP, IP og UDP i tillegg til en lang rekke andre protokoller.

Vi mener at adgangskontroll vil være en god metode for å sikre at datastrømmene får tilgang på tilstrekkelig minne- Vi mener videre at minne i dag er en såpass rimelig ressurs, at vi kan forsvare å fast allokere minne til hver datastrøm og dermed garantere at minne finnes for hver datastrøm. Ved fast allokering slipper vi også kostnaden ved å allokere minne ved behov, og vi tror vi på den måten også får en mer effektiv minnehåndtering.

Kapittel 5

Bufferhåndtering

I dette kapittelet ser vi på forskjellige aspekter ved bufferhåndtering. Innholdet i dette kapittelet er dels basert på informasjon fra Patterson et. al.[35].

Først i dette kapittelet ser vi i avsnitt 5.1 kort på oppbygningen av en datamaskin. Hensikten er å ha et rammeverk slik at vi senere kan forklare flyten av data i en MoD-tjener. I avsnitt 5.2 gir vi så en rask innledning til bufferhåndtering i INSTANCE-noden. I avsnitt 5.3 ser vi på inndelingen av datalagringen i forskjellige nivåer med varierende hastighet, størrelse og pris. Videre i avsnitt5.4 ser vi på begrepet virtuelt minne. Avsnitt 5.5 beskriver inndelingen av minnet i soner for å sikre systemet. Vi går igjennom forskjellige metoder for å flytte minne i en datamaskin i avsnitt 5.6. Videre ser vi på dataflyten i et større bilde i avsnitt 5.7. I avsnitt 5.8 ser på allokering av buffere og på spørsmålet om en ny buffermekansime skal erstatte eller supplere de gamle. i avsnitt 5.10 konseptene i dette kapittelet i forhold til INSTANCE-konseptet og INSTANCE-noden.

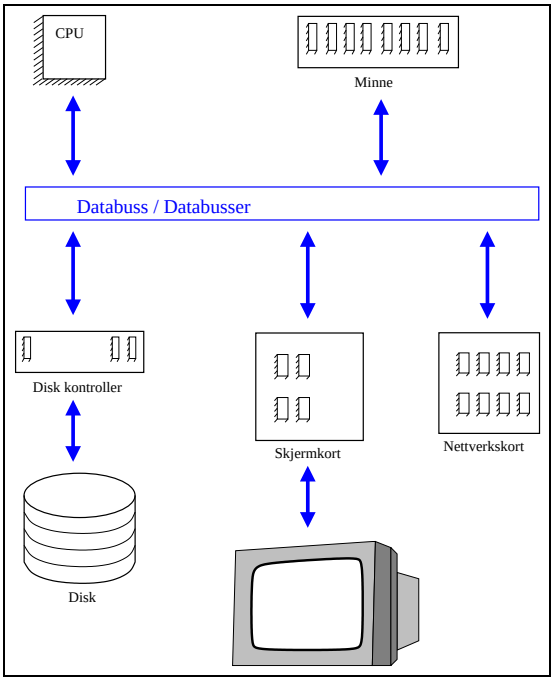
5.1 Kort om oppbygningen av en datamaskin

Før vi går nærmere inn på datahåndteringen i maskinen går vi raskt inn på oppbygningen av datamaskinen. Figur 5.1 viser en skisse over oppbygningen. Komponentene i en datamaskin kan grovt deles inn i fire deler:

Prosessoren *Prosessoren (Central Processing Unit, CPU)* utfører programmene.

Minnet Alle data som prosessoren skal jobbe med må finnes i minnet.

Inn/Ut enheter *Inn/Ut enhetene (Input/Output, I/O)* står for kommunikasjon mellom prosessoren og “resten av verden”. I/O enhetene består for eksempel av skjerm, tastatur, disker og nettverkskort.



Figur 5.1: Forenklet skisse av en datamaskin

Databusser/Databusser Kommunikasjonen mellom CPU, Minne og I/O enheter går via brede databusser.

Figur 5.1 viser disse enhetene. CPU-en står for utføringen av programmene i datamaskinen. Dataene CPU-en jobber med ligger lagret i minnet. Det gjelder også data som leses fra disk. Den mellomlagres i minnet før CPU-en bruker dem. All informasjon mellom CPU, minne og I/O enheter foregår via databusser.

Oppbyggingen av en datamaskin er i virkeligheten ofte langt mer komplisert enn figur 5.1 viser. Vi har gjerne langt flere I/O enheter. Databussen er splittet opp i flere forskjellige busser som kan operere i forskjellige hastigheter. Vi kan også ha flere prosessorer i samme datamaskin.

Det viktige i dette avsnittet er *ikke* detaljene i designet. Prinsippene og oppdelingen er viktig, og ikke minst det faktum at de forskjellige delene i systemet opererer med forskjellige hastigheter og det faktum at de må kommunisere via busser. CPU-en opererer mange ganger raskere enn bussen og minnet. Dersom disse CPU-en skulle jobbe direkte med data i minnet ville CPU-en stått mye stille mens den ventet på data som skulle leses direkte fra minnet via databussen. Det er også heller sjelden at nettverket har samme hastighet som bussen nettverkskortet sitter på. Disken er normalt langt tregere enn databussene, i tillegg kan hastigheten variere og lesingen

kan stoppe helt opp når disken må flytte et lesehode for å kunne lese data et annet sted på disken.

Alle hastighetsforskjellene i maskinen skaper problemer. Hvis alt skulle operere med hastigheten til den tregeste enheten ville ting gå svært sakte. Derfor finnes det *buffere* og andre mekanismer forskjellige steder i maskinen. Data mellomlagres i bufferne slik at hastighetsforskjellene kan utjevnes. Dette gjelder for eksempel forholdet mellom CPU-en og minnet. CPU-en kan operere med hastigheter mange ganger høyere enn minnet. For å løse dette problemet finnes det i mange datamaskiner ett eller to nivåer med *cache* mellom CPU-en og databussen som minnet sitter på. Cache er ett hurtigminne som gjør at prosessoren har hurtig tilgang til data den tidligere har jobbet med. Tilsvarende har nettverkskort et eget buffer for å håndtere forskjellen i hastighet mellom databussen og nettverket. Disken har også gjerne et eget buffer og skjermkortet lagrer alle skjermbildene i et eget buffer før det vises på skjermen.

Mange busser med forskjellige hastigheter og buffring flere steder i maskinen betyr at dataflyt internt i systemet er et komplekst område der det kan finnes mange flaskehalser. En hurtig prosessor hjelper lite dersom databussen ikke kan flytte nok data. Dersom databussen er en flaskehals vil en raskere prosessor gi lite eller ingen gevinst i total ytelse for maskinen. Det samme gjelder hvis maskinen bruker mye tid på å kopiere data fra sted til sted internt i minnet. En mulighet er å kjøpe en maskin med en raskere databuss og minne. En annen mulighet er å designe systemer med henblikk på forutsetningene. Dersom databussen eller minnekopiering er en flaskehals vil det være mye å hente på å optimalisere dataflyten slik at minst mulig data strømmer gjennom bussen og minne kopieres ingen eller færrest mulig ganger. Det er dette vi ønsker å se på videre i oppgaven.

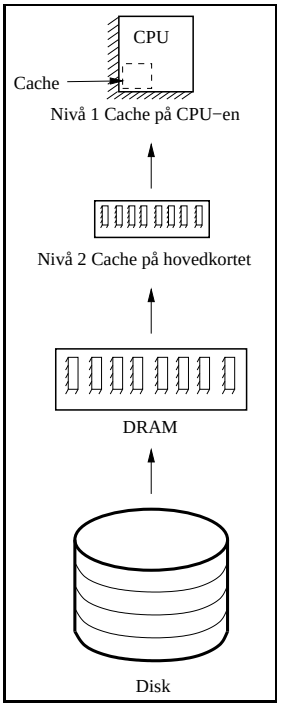
5.2 Innledning om buffere

Håndtering av data skjer på flere nivåer. Vi skisserer her en inndeling i tre nivåer:

Det laveste nivået er den direkte håndteringen av minnet. Hvordan minnet allokeres og deallokeres. Man kan håndtere alt minne i en pool og allokere etter behov. Hvis man har datastrømmer med kjente karakteristikker kan man forhåndsallokere minne etter kjente eller beregnede behov.

Midt på treet har vi hvordan data flyter i systemet. Om data kopieres fra et systems minneområde til et annet minneområde, eller om disse systemene på en eller annen måte deler et eller flere område.

Øverst ligger håndteringen av datastrømmene og resursene tilknyttet disse. Skal hver mottager av data ha sine egne ressurser, eller er det ønskelig at flere mottagere



Figur 5.2: Inndelingen av datalagring i nivåer

deler de samme ressursene. Her er det et valg mellom en til en forbindelser mellom tjener og klient, og forskjellige metoder for multicasting til mange klienter på en gang.

Vi vil videre i oppgaven søke å se arbeidet i disse tre lagene i sammenheng. Håndteringen av datastrømmene i et nettverk kan etter vår mening påvirke designet av dataflyten internt i tjeneren. Flyten av data internt i maskinen påvirker så hvordan vi velger å håndtere allokeringen av data i maskinen.

5.3 Lagringshierarkiet

Vi vil nå se på hvordan minnehåndtering tradisjonelt foregår i et OS. Senere ser vi hvordan dette vil arte seg for en MoD-tjener.

Vi starter med å se på lagringshierarkiet og fordeler og ulemper med dette. Oppbevaring av data er ordnet i lag, der de høyere lagene er raskere, dyrere og inneholder mindre datamengder. Tradisjonelt ligger harddisken nederst. Harddisken har flere fordeler. Den er rimelig. Den kan inneholde store datamengder. Den er også persistent, dvs. at dataene bli liggende der etter at maskinen er slått av.

Det er mulig å bygge en datamaskin som jobber direkte mot disken, men dette ville være ekstremt tregt. Derfor kopierer man dataene inn i minnet før man gjør noe med det. Minnet er gjerne mange faktorer mindre enn disken, men også mange ganger raskere. Her brukes gjerne *Dynamic Random Access Memory (DRAM)*, det er billig, men tregt hvis man sammenligner med hastigheten på CPU-en.

Fordi DRAM er tregt, er det en begrenset faktor hvis CPU-en skulle arbeide direkte med dette. For å bøte på dette, er det vanlig å legge et nivå eller to til med raskere minne mellom DRAM og CPU-en. Dette lagret kalles *cache* og lagrer data CPU-en har jobber med eller data som ligger tett inntil slike data. Bakgrunnen er at man ønsker å utnytte lokalitet i sted, at man gjerne aksesserer data som ligger ved siden av hverandre etter hverandre og lokalitet i tid, at man ofte bruker de samme dataene flere ganger i CPU-en.

Som cache brukes typisk *Static Random Access Memory (SRAM)*, SRAM er vesentlig raskere enn DRAM, koster også vesentlig mer, og tar mer plass. I det siste har flere begynt å ha to eller flere lag med cache mellom minnet og CPU-en. En større SRAM-cache plassert på hovedkortet og en mindre nærmere, CPU-en. Den siste kan gjerne plasseres på samme chipen som CPU-en, slik at avstanden blir minst mulig.

Til slutt har man registrene i CPU-en. Det er her dataene som CPU-en regner på ligger.

Ved prosessering blir så dataene kopiert fra disk til minne, fra minne til cache og til slutt inn i Registrene før de blir behandlet. Dette kan virke tregt, men fordi hastigheten øker så mye jo nærmere de kommer CPU-en er dette lønnsomt.

Det man kan se her er at lokalitet i tid er en vesentlig forutsetning for at dette skal lykkes. Dersom man kun aksesserer dataene en eneste gang blir det mye kopiering til liten nytte. I en MoD-tjener vil det også være aktuelt å ikke behandle dataene i CPU.en en enste gang, men sende dem rett ut i nettet.

Fordi det å kopiere alle data via CPU-en er kostbart, er *Direkte minne aksess (Direct Memory Access, DMA)*. utviklet. Dette er en metode der data kopieres fra en kilde på bussen til en annen, uten at dataene er innom cache eller CPU. Her gir CPU-en en kommando til DMA-en å flytte en hviss mengde data fra en posisjon til en annen. Dette kan brukes til å kopiere data internt i minnet, fra disk til minne eller fra minne til nettverkskort. CPU-en gir en kommando som inneholder startpunkt og lengde på dataene, samt hvor det skal kopieres.

5.4 Virtuelt minne og sideutbytting

Tradisjonelt har prisen på minne utgjort en stor del av prisen på en datamaskin. I tillegg var produksjonen av store minnebrikker vanskelig. Resultatet ble maskiner

med svært lite minne sammenlignet med dagens maskiner. Deler av informasjonen i dette avsnittet er hentet fra Denning[19].

I avsnitt 5.3 har vi sett hvordan man gir inntrykk av et stort raskt minne på prosessoren ved å cache data fra DRAM i cachen. En lignende teknikk blir brukt for å konseptuelt øke størrelsen på DRAM-en. For å få til dette benyttes to teknikker, *virtuelt minne* og *sideutbytting.* Vi skal går raskt inn på disse to begrepene.

5.4.1 Virtuelt minne

Allerede i 1950 årene laget designerne av Atlas maskinen på universitetet i Manchester et system med *virtuelt* minne. Hensikten var å lette arbeidet med å håndtere maskiner med forskjellige minnestørrelser. For å slippe å kompilere programmene på nytt for hver maskin, innførte de konseptet virtuelt minne.

Hensikten med et virtuelt minne er blant annet å gi inntrykk av et større minne enn det størrelsen på det fysiske minnet. Dette får man til ved å opprette et stort *virtuelt adresserom*. Enheter i dette adresserommet, kalt sider[1]. Disse virtuelt adresserte sidene *mappes* så til fysiske sider. Med det menes at nå programmet adresserer en virtuelt adresse, så oversetter minnesystemet dette til en fysisk adresse og henter data fra riktig minneside. Figur 5.3 illustrerer dette. Vi ser at det virtuelle adresserommet fra 1–n er langt større en det fysiske som i denne figuren er på 48KB[2]. Vi ser at ikke alle adresser i det virtuelle adresserommet er mappet til fysisk minne. Det er naturlig da størrelsen på dette adresserommet normalt er satt så stort at det ikke skal gå fullt. Minnesystemet vedlikeholder en tabell som forteller hvilke virtuelle adresser som mapper til hvilke fysiske adresser.

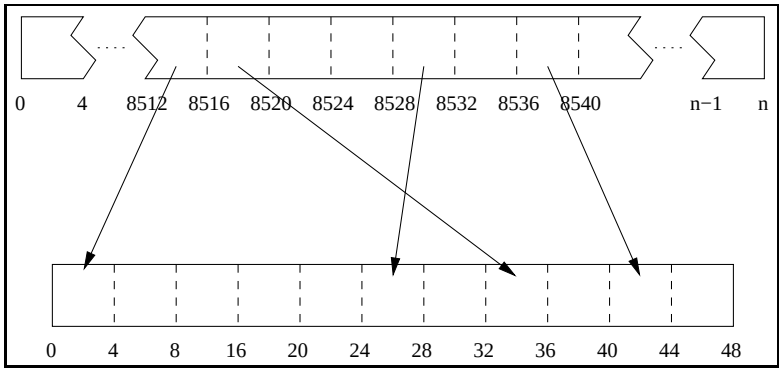
5.4.2 Sideutbytting

Vi har sagt at hensikten med det virtuelle minne var å gi inntrykk av et minne som er større en det fysiske minne. Hvor blir det da av de delene av det virtuelle minnet som ikke får plass i det fysiske minnet? Det er her sideutbytting kommer inn. Når det fysiske minnet er fult er det opp til sideutbyttingssystemet å flytte lite brukte minnesider fra det fysiske minnet over til disk. Dermed frigjøres minne til nye formål. Når det igjen spørres etter som er flyttet til disk, er det sideutbyttingssystemets oppgave å kopiere det inn igjen i minne og evt. flyte andre sider over til disk.

Fordi det å skrive og lese til disk er relativt trege operasjoner, legges mye resurser inn på å minimere antallet slike operasjoner. Dels gjøres dette ved å bergene

[1] Som et eksempel kan vi nevne at i dagens `NetBSD` er en side 4KB

[2] Dette er kun en illustrasjon. I dagens datamaskiner er størrelsen på det fysiske minne selvsagt langt større



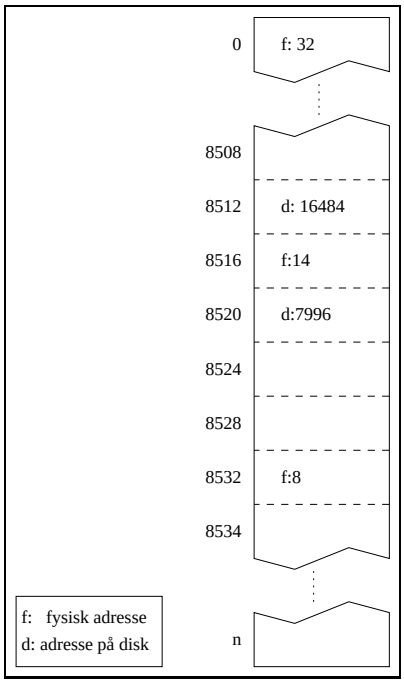
Figur 5.3: Mapping av virtuelt minne til fysisk minne.

sannsynligheten for hvilke minnesider som minst sannsynlig vil bli aksessert i den nærmeste tiden. Dels holder minnesystemet orden på hvilke fysiske minnesider som er endret siden lesing fra disk. Dersom en side som skal flyttes til disk ikke er endret siden den ble lest fra disk, er det ikke nødvendig å skrive den til disken på nytt. Det er nok å endre oppslaget i tabellen fra å peke på den fysiske minnesiden til siden på disk. Figur 5.4 illustrerer en slik tabell.

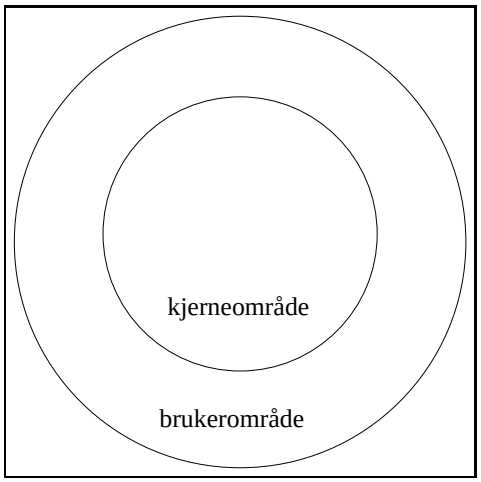
5.5 Sikkerhet og soneinndeling

Datasikkerhet er et viktig element i minnehåndteringen. For å muliggjøre dette har de fleste OS-er beregnet for flere samtidige brukere et system med med rettigheter for minnet. Vanligvis ordnes dette ved at hver minneside har en sett med rettigheter og gjerne en eier. Man deler gjerne inn i minnet i *brukerområdet (userspace)* og *kjerneområdet (Kernelspace).* Disse er i NETBSD skilt fra hverandre ved at de har hvert sitt virtuelle adresserom. Brukerne kan ikke aksessere kjerneområdet direkte. Dette er en enkel og rask metode for å sikre systemet. Bare prosesser som er en del av OS-et kan aksessere kjerneområdet. Dermed vil andre prosesser som enten inneholder feil eller som er ondsinnede ikke lett kunne ødelegge for OS-et ved å endre ting i kjerneområdet.

En side ved å skille kjerneområde fra brukerområde er at data må flyttes frem og tilbake mellom de to områdene dersom systemer i kjerneområdet og brukerområde skal kommunisere med hverandre. Vi ser mer på dette i neste avsnitt.



Figur 5.4: Tabell for sideutbytting.



Figur 5.5: Soneindeling av minnet i et operativsystem

5.6 Dataflyt og semantikk

Vi har sett at det i et multibruker datasystem er innført en rekke begrensninger på minnet for å sikre stabilitet og sikkerhet. Dette gir svært stabile systemer med høy oppetid gir brukerne en hviss sikkerhet. Som vi skal se er disse systemene derimot ikke særlig godt egnet dersom vi skal lage en MoD server.

Vi skal nå se på forskjellige prinsipper for dataflyt i en tjener. Pasquale et. al.[34] bruker begrepet *I/O Pipeline* for å beskrive minneflyt. I/O Pipeline beskriver en gjentatt aktivitet hvor

1. En prosess leser inn et stort dataobjekt.
2. Data blir så sekvensielt overført mellom domene til forskjellige prosesser, som alle kan lese eller forandre deler av dataene.
3. En prosess sender ut de muligens modifiserte dataene.

Druschel et. al.[21] bruker et noe smalere begrep, en *I/O datapath.* De påpeker at i de fleste I/O operasjoner er et kun avsenderapplikasjonen som endrer dataene. De senere applikasjonene

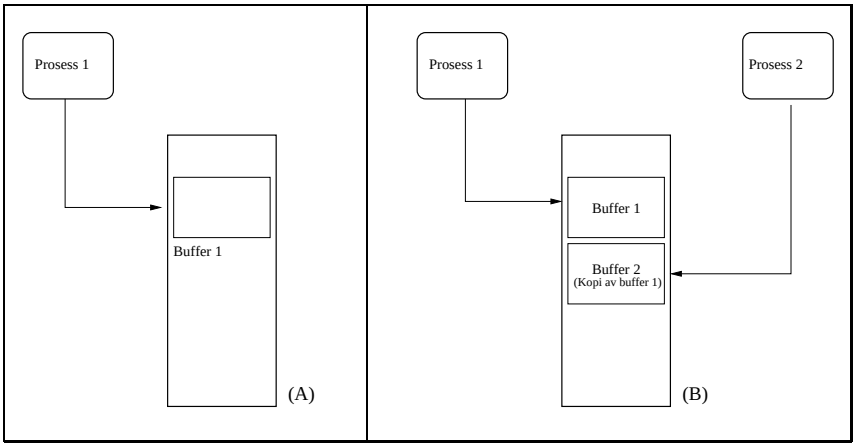
Vi skal bruke denne modellen for å beskrive forskjellige strategier for minneflyt i en tjener. Senere vil dette danne grunlaget for diskusjonen om valg av strategi for minneflyt i INSTANCE-noden. Videre i dette kapittelet ser vi på minneflyten fra kjernen til en prosess i brukerområdet. Det vi beskriver er like relevant hvis data skal transporteres fra en prosess til en annen, eller fra en prosess og tilbake til kjernen.

5.6.1 Kopiere minne

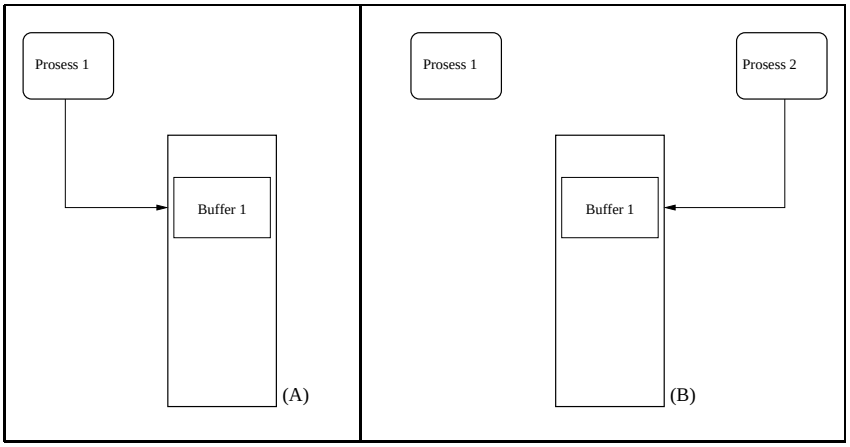
Den vanlige metoden for å gjøre data i kjernen av OS-et tilgjengelig for en prosesse er å kopiere dataene inn i brukerområdet til prosessen. Figur 5.6 viser dette. Vi ser situasjonen før (figur 5.6(A)) og etter kopiering (figur 5.6(B)). Kopiering av minne er enkelt å få til, men koster mye tid. Til tross for ulempene er dette er den måten som er valgt i en rekke systemer.

5.6.2 Flytte minne

Som et alternativ til å kopiere, kan man “flytte” dataene minnet fra kjernen til prosessen. Figur 5.7 illusterer dette før flytting (figur 5.7(A)) og etter (figur 5.7(B)).



Figur 5.6: Kopiering av minne (copy)

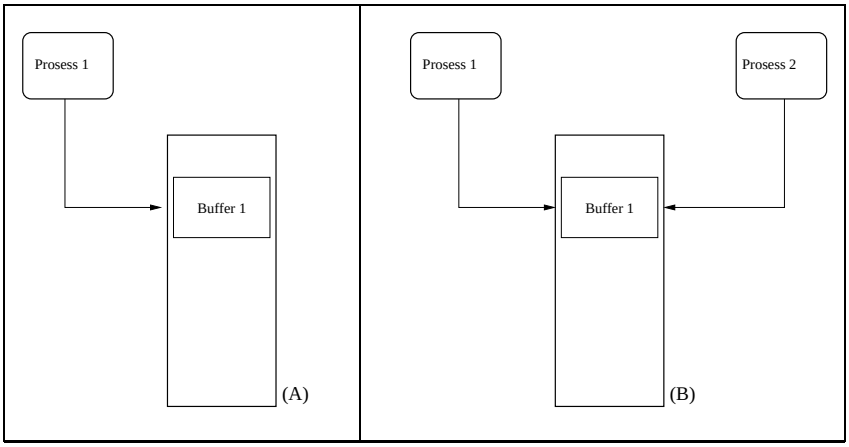


Figur 5.7: Flytting av minne (move)

Flytting av minne betyr at man overdrar rettighetene til den fysiske minnesiden fra kjernen til en prosessen. Dette kan skje ved at referansen i kjernens virtuelle adresserom fjernes og erstattes med en ny referanse i prosessens virtuelle adresserom. Dette er svært mye raskere en å kopiere minne. Ulempen kan være at kjernen mister tilgangen til minnet. Hvis kjernen trenger dette er flytting av minne uegnet.

5.6.3 Dele minne

En tredje metode som illustreres av figur 5.8 er deling av minnet. Figuren viser situasjonen før deling (figur 5.8(A)) og etter deling (figur 5.8(B)). Deling av minne betyr at man gir en annen prosess full tilgang til å lese og skrive til en minneside.



Figur 5.8: Deling av minne (share)

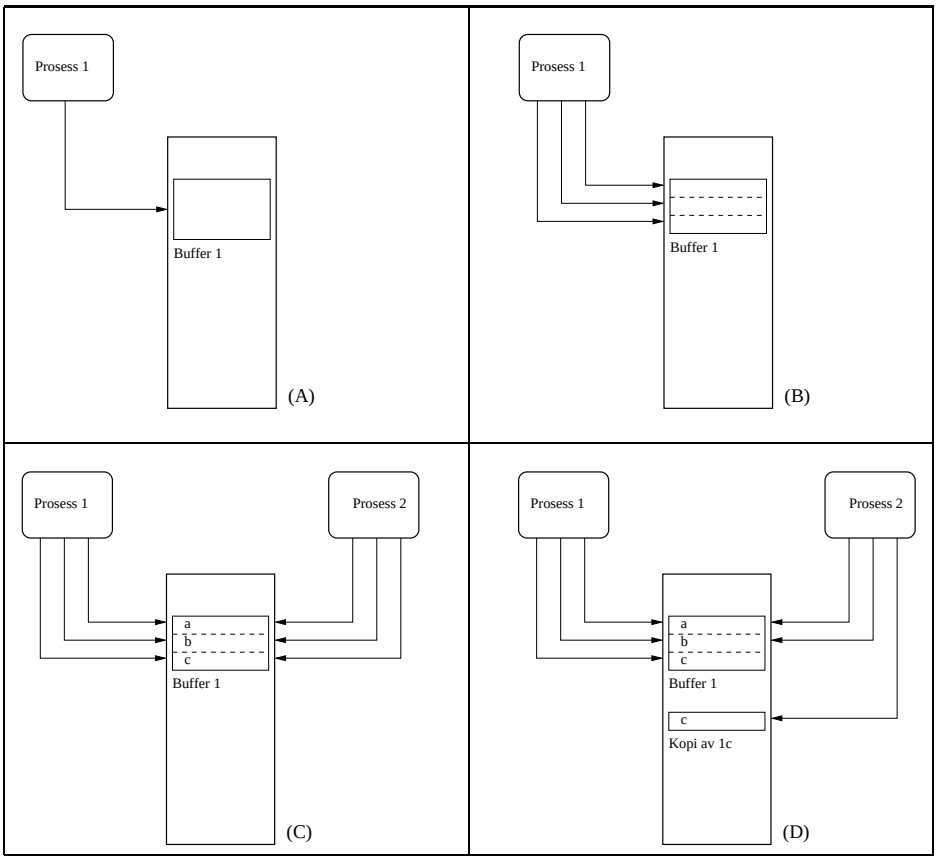
Fordi minnet i kjerneområdet er skilt fra brukerområde er dett en metode som kanskje er mest aktuell for kommunikasjon mellom to deler av kjernen eller mellom to prosesser i brukerområde.

At en prosess deler minne med en annen prosess betyr at den andre prosessen Dette er en meget rask metode dersom to prosesser eller deler av kjernen skal utveksle data. Ulempen er at prosessene står fritt til å skrive til minne når de måtte ønske, noe som kan skape feil hvis ikke prosessene handler “fornuftig”. Dette betyr at prosessene som deler minne bør vite om hverandre eller oppføre seg etter bestemte regler som fører til at de oppfører seg hensynsfult.

5.6.4 Copy on Write

Nok en mulighet er *Copy on Write (CoW)*. CoW ligger et sted midt i mellom deling av minne og kopiering av minne. Kjernen kan bruke CoW til å gi en prosess tilgang til et minneområde. Prosessen får tilgang til minnet ved at kjernen og minnet begge har referanser til samme fysiske minneside som ved deling av minne. Det som skiller CoW fra deling av minne er at det i CoW finnes en mekanisme som overvåker at ingen skriver til det felles området. Dersom en prosessen prøver å skrive til en minneside delt med CoW, gripe minnehåndteringssystemet inn. En ny minneside allokeres og dataene fra den delte siden kopieres over til den nye siden. Dermed for den skrivende prosessen sin egen kopi av siden og påvirker ikke den eller de andre som leser fra den delte siden.

Figur 5.9 illusterer CoW. Figur 5.9(A) viser at prosess 1 eier et minneområde, buffer 1. Figur 5.9(B) illusterer at buffer 1 omfatter tre minnesider. Prosess 1 gir prosess 2 tilgang til dataene ved å bruke CoW (figur 5.9(C)). Prosess 2 behøver



Figur 5.9: Copy-on-write

ikke vite at minne er delt med CoW. Når prosess 2 skriver til minneside c, oppreter minnehåndteringssystemet en egen side for prosess 2, kopierer dataene fra det delte minneområdet og flytter prosess 2s referanse til å peke på den nye siden. Prosess 2 skriver nå til en privat kopi av dataene.

En fordel med CoW er at den kan erstatte det orginale minnekopieringen. Prosessene som bruker CoW merker ingen forskjell i forhold til tradisjonell kopiering. De har sine egne "private" minnesider som bare endres dersom de selv skriver til dem. Den eneste forskjellen er at tidskostnaden ved kopiering flyttes fra det øyeblikket prosessene ber om å "kopiere" dataene, til det øyeblikket de eller den andre prosessen skriver til det CoW-delte minnet. Fordi prosessene ikke alltid skriver til et minneområde de har "kopiert" fra kjernen elller en annen prosess kan CoW redusere tidsforbruket ved minnekopiering.

5.7 Dataflyten i en tjener

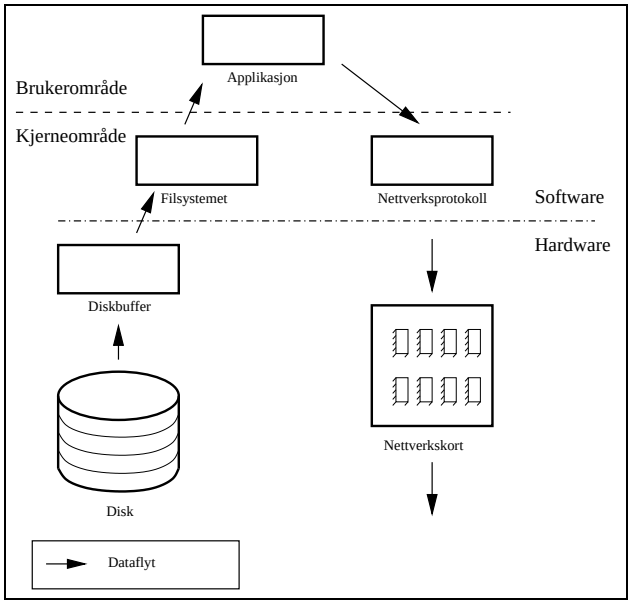
I en MoD tjener er dataflyten gjennom tjeneren gjerne stor. Det betyr at flaskehalser lett kan oppstå i systemet. Slike flaskehalser kan for eksempel være at hastigheten på disken er for dårlig slik at man ikke kan få lest så mye man ønsker. En annen viktig og kjent flaskehals er tiden det tar dersom data skal kopieres fra sted til sted i maskinens minne. Figurene 5.10, 5.11 og 5.12 Viser eksempler på mulige måter å flytte data på fra disk og til nettverk.

5.7.1 Tradisjonell dataflyt

I en tradisjonell tjenerarkitektur som legger applikasjonen i OS-ets brukerområde tar dataene en lang vei fra disk til nett. Figur 5.10 viser et eksempel på dette. Denne figuren viser hvordan data først kopieres fra de fysiske platene på disken. Alle moderne disker har et eget minne på disken slik at lesehastigheten på disken og overføringshastigheten ikke behøver å være den samme. Det samme gjelder for nettverkskortet. I tillegg til denne bufringen lokalt på disken og på nettverkskortet viser figur 5.10 tre kopieringer. Først fra diskbufferet til filsystemets I/O buffer. Videre inn i applikasjonens buffer i brukerområde slik at applikasjonen kan lese og skrive til bufferet. Etterpå kopieres dataene tilbake i kjerneområdet og til kommunikasjonssystemets I/O buffer. Tislutt kopieres dataene over til nettverkskortet og sendes ut på nettet.

5.7.2 Zero-copy dataflyt

Kopiering fra buffer til buffer internt i maskinen er en tidkrevende prosess[32]. Dette blir dermed lett en flaskehals i et system som skal transportere mye data fra



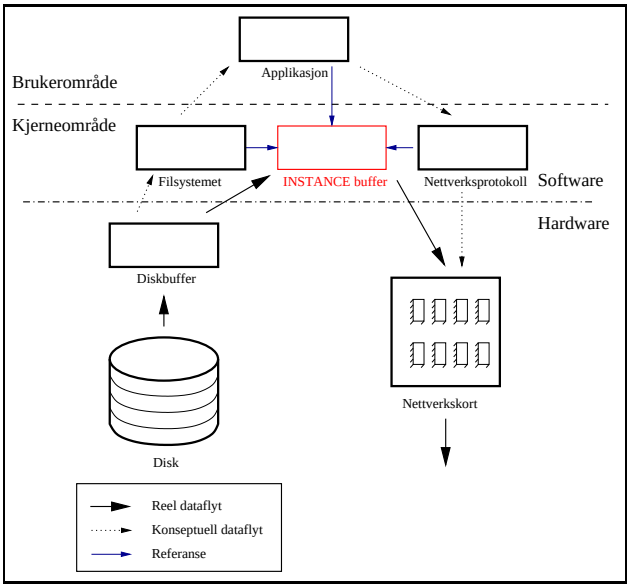
Figur 5.10: Tradisjonell dataflyt i en tjener.

disk til nettverk. Dette har ført til at flere har foreslått og implementert forskjellige metoder for ungå kopiering internt i maskinen, såkalt *zero-copy datapath* bufferhåndteringsmekansismer. Figur 5.11 illustrer en en mulig vei for dataene i en tjener som ikke innebærer kopiering av data.

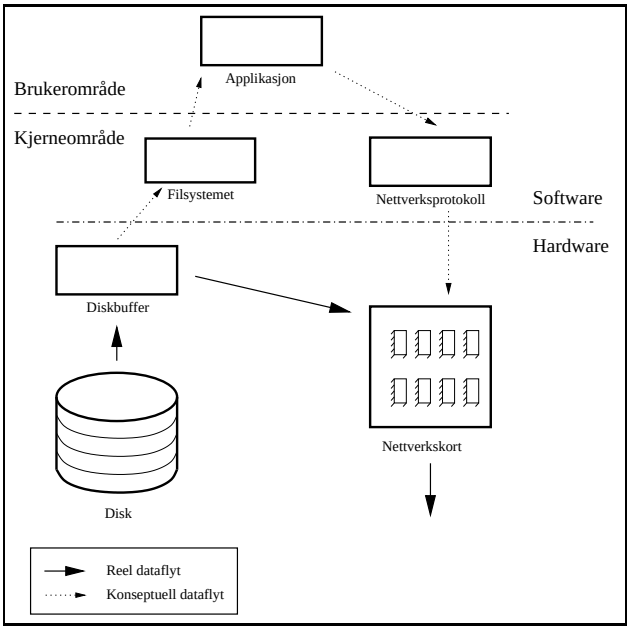
5.7.3 En annen mulig dataflyt?

Figur 5.12 viser en minneflyt der dataene ikke kopieres inn i minnet i det hele tatt. Her blir data kopiert direkte fra diskbufferet til bufferet på nettverkskortet.

Denne modellen *kan* muligens gi en enda raskere dataflyt enn modellen i figur 5.11. Den vil alikevel ikke være særlig aktuell med dagens datamaskinarkitektur. En forutsetning vil være at dataene kommer fra disken i den rekkefølgen de skal sendes ut på nettverket. Dette vil gå på bekostning av disksystemets mulighet til å sortere leseoperasjonene på den måten som gir raskest mulig lesing. Det er mulig at denne modellen kan bli aktuell i fremtiden, men i dagens situasjon vil det være svært vanskelig å få til. Vi velger derfor å se bort fra denne modellen når vi lager en bufferhåndteringsmekanisme for INSTANCE-noden.



Figur 5.11: Zero-copy dataflyt



Figur 5.12: Ideell minneflyt?

5.8 Andre elementer i bufferhåndteringen

Vi ser nå på andre sider i bufferhåndteringen som ikke har passet inn i de andre avsnittene. Dette gjelder elementer som når bufferne allokeres og hvorvidt en ny metode for bufferhåndtering skal erstatte eller supplere de orginale metodene.

5.9 Allokering av buffere

Et kritisk punkt i minnehåndteringssystemet er hvordan buffere blir allokert. Strategien i en rekke systemer er optimalisert for å maksimalisere utnyttelsen minnet. Det er foresått og implementert flere forskjellige teknikker. Felles for alle er at de bruker CPU-tid for å maksimalisere minneutnyttelsen.

5.9.1 Buffertranspares

Et viktig spørsmål ved design av en bufferallokeringsmekanisme, er *transparens*. Med dette mener vi hvorvidt mekanismen for bufferallokering er synlig for programmene. En muilighet er å la en ny mekansime erstatte den gamle uten å endre grensesnittet programmene opererer mot. Dermed kan programmer benytte den nye mekanismen uten at det er nødvendig å skrive om programmene. Alternativet er å innføre en mekanisme som eksplisitt kalles fra programmene. Med denne stategien må alle programmene skrives spessielt for å nytte den nye teknikken.

Siden det er en betydelig hindring å skulle skrive om programmene som akkseserer buffermekanismen, ser dette i utgangspunkt ut som det beste alternativet. Ulempen er at alterantivene for design av en mulig mekansime begrenses kraftig dersom den nye mekansimen skal oppføre seg slik som den gamle.

En annen mulighet er å lage en ny metode som operer parallelt med den gamle. På den måten kan nye programmer skrives spessielt med tanke på å utnytte fordeler med en ny bufferhåndteringsmekanisme, mens de gamle programmene fortsetter å fungere.

5.9.2 Blokkstørrelser i datastrømmer

Et viktig spørsmål når man skal buffre en datastrøm internt i maskinen er størrelsen på bufferne. Chang et. al.[13] introduserer to forskjellige prinsipper for inndeling av VBR datastrømmer i blokker, *konstant datalengde (Constant Data Length, CDL)* og *konstant tidslengde (Constant Time Length, CTL)*.

CDL betyr at blokkenes størrelse er fast. Dermed vil et sekund med data fylle et varierende antall blokker. CTL betyr at blokkene inneholder data for en bestemt tidslengde. Dermed består et sekund med data av et fast antall blokker, men blokkenes størrelse varierer.

5.9.3 Bufferintegritet

Bufferintegritet er viktig ved minnehåndtering. Det handler rett og slett om hvorvidt en programentitet kan gå ut fra at dataene i et buffer ikke blir endret av andre. Det er nettopp dette som er en av de viktigste grunnene til innføringen av minnehirearkiet som vi diskuterte i avsnitt 5.5, segmenteringen av minnet og bakgrunnen for at den vanligste minnehåndteringsmekanismen er kopiering.

5.10 Bufferhåndteringen og INSTANCE

Vi ønsker å lage en zero-copy datapath i INSTANCE-noden. Valget står dermed etter vår mening mellom move, share og CoW teknikker.

CoW er opptatt av å bevare en copy semantikk. CoW krever et system som mapper sider inn og ut av områder i det virtuelle minnet. Slik mapping tar tid. Cranors[15] målinger viser at i hans system tar remapping av 4KB faktisk mere tid enn å kopiere de samme dataene.

Move teknikkene krever også at minnet, eller referanser til det flyttes fra virtuelt område til område. I tillegg kommer kostnadene ved å allokere data i begynnelsen av datapathen, og deallokering når bufferet når slutten.

Valget vår faller derfor på share prinsippet. Vi har en dedikert node og har dermed større muligheter for å sikre oss at de enhetene som aksesserer bufferet ikke korrumperer dataene. En av hensiktene med INSTANCE-noden er at dataene skal ligge klar for sending på disk. Det betyr at applikasjonen ikke behøver å gjøre annet en å styre dataflyten. Applikasjonen har intet behov for å aksesserer dataene direkte. Ved å velge en in-kernel datapath vil en share teknikk etter vår mening egne seg godt.

Sideutbytting er ikke viktig. Siden data skal fraktes fra disk til nett ville det være direkte destruktivt å page data ut på disk. Vi foreslår derfor å bruke “wired down” buffere.

Siden vi velger en share semantikk kan vi også unngå mye av kostnadene ved allokering og deallokering av minne. Vi kan klare oss med å allokere nødvendig minne ved oppstart av en datastrøm, og deallokere minnet ved datastrømmens slutt.

Kapittel 6

Implementasjoner av bufferhåndtering

Det er en rekke eksempler på forskjellige måter å implementere buffere og håndteringen av disse. Vi vil i dett kapittelet se på en del eksempler av eksisterende implementasjoner og forslag til nye mekanismer som helt eller delvis er implementert.

Vi tar først for oss den "tradisjonelle" bufferhåndteringen i NETBSD i avsnitt 6.1. Vi ser på bufferhåndteringen i akkurat NETBSD da dette er OS-et vi har valgt som grunnlag for vår implementasjon (begrunnelsen for dette valget kommer i avsnitt 8.1.2). Vi ser så på andres mekanismer for zero-copy bufferhåndtering. I avsnitt 6.2 ser vi på mmbuf. Container Shipping beskrives i avsnitt 6.3. IO-Lite kommer så i avsnitt 6.4 og Genie i avsnitt 6.5. Til slutt vurderer vi i avsnitt 6.6 hvordan disse konseptene egner seg for bufferhåndteringen i INSTANCE-noden.

6.1 NETBSD, en "vanlig" implementasjon

Vi tar med Minnehåndteringen i NETBSD som et eksempel på tradisjonell minnehåndtering. I tillegg er dette viktig bakgrunnsstoff da INSTANCE-noden er implementer på NETBSD plattformen. NetBSD er en relativt utbredt variant av 4.4BSD UNIX. Informasjonen i dette avsnittet er hovedsakelig hentet fra McKusick et. al.[30] og Wright et. al.[43]. Kildekode er hentet fra NETBSD's offisielle webside[31].

Minnehåndteringen i NETBSD er ikke optimalisert for distribusjon direkte mellom disk og nettverk. NETBSD er et generelt Operativsystem, og løsningene er derfor laget for å fungere bra i så mange sammenhenger som mulig. Vi ser derfor separat på først den delen som håndterer lesing fra disk, deretter sending til nett.

```
struct buf {
  LIST_ENTRY(buf) b_hash;       /* Hash chain. */
  LIST_ENTRY(buf) b_vnbufs;     /* Buffer's associated vnode. */
  TAILQ_ENTRY(buf) b_freelist;  /* Free list position if not active. */
  struct buf *b_actf, **b_actb; /* Device driver queue when active. */
  struct proc *b_proc;          /* Associated proc; NULL if kernel. */
  volatile long b_flags;        /* B_* flags. */
  int b_error;                  /* Errno value. */
  long b_bufsize;               /* Allocated buffer size. */
  long b_bcount;                /* Valid bytes in buffer. */
  long b_resid;                 /* Remaining I/O. */
  dev_t b_dev;                  /* Device associated with buffer. */
  struct {
  caddr_t b_addr;               /* Memory, superblocks, indirect etc. */
  } b_un;
  void *b_saveaddr;             /* Original b_addr for physio. */
  daddr_t b_lblkno;             /* Logical block number. */
  daddr_t b_blkno;              /* Underlying physical block number. */
                                /* Function to call upon completion. */
  void (*b_iodone) __P((struct buf *));
  struct vnode *b_vp;           /* Device vnode. */
  int b_dirtyoff;               /* Offset in buffer of dirty region. */
  int b_dirtyend;               /* Offset of end of dirty region. */
  struct ucred *b_rcred;        /* Read credentials reference. */
  struct ucred *b_wcred;        /* Write credentials reference. */
  int b_validoff;               /* Offset in buffer of valid region. */
  int b_validend;               /* Offset of end of valid region. */
  off_t b_dcookie;              /* Offset cookie if dir block */
};
```

buf.h

Figur 6.1: `struct buf`

6.1.1 Disklesing og `buf` strukturer

I NETBSD, som i mange andre generelle operativsystemer, er disksystemet skrevet får å optimalisere programmer som har mange små lese og skriveoperasjoner. Dette er gjort ved å legge et lag med systembuffere mellom disken og prosessene. Prosessene skriver til og leser fra disse bufferne i stedet til disken. På denne måten kan man gi et inntrykk av at lese og skriveoperasjoner kan foregå byte for byte, mens den i virkeligheten foregår i langt større enheter, kalt blokker.

Disse bufferne består av `buf` strukturer. Hver `buf` struktur buffrer en hvis del av disken. Når programmet er ferdig med å lese eller skrive til denne delen av disken, frigjøres `buf` strukturen, og en ny `buf` allokeres for å buffre neste ønskede blokk. Hver `buf` buffrer typisk mellom 8KB og 64KB. Minste enhet er en diskblokk, som i NETBSD normalt er 8KB.

Figur 6.1 viser kildekoden til `buf` strukturen. Strukturen har en lang rekke peker til

andre deler av systemet. Noe av det viktigste i strukturen er pekerne `b_blkno` og `b_lblkno`. `b_lblkno` peker på den logiske blokken vi ønsker å lese og `b_blkno` på den fysiske. Ellers er det peker til vnoden det leses fra (`b_vp`), bufferstørrelse (`b_bufsize`), dirtypekere for å håndtere writeback mm.

Fordi det ligger buffere mellom prosessene og disken i NETBSD, er det flere muligheter for å optimalisere diskaksessen ytterligere. En av dem er *prefetching*. Prefetching er å lese data fra disk før disse blir etterspurt. Prefetching kan settes i gang etter at prosessen har lest en hviss mengde data i rekkefølge. Disksystemet antar da at prosessen også skal lese videre i filen, og leser mer en en blokk om gangen.

6.1.2 Nettverk og `mbuf` strukturer

Alle data som går ut på nettverket i en NETBSD maskin må innom en `mbuf`. Dette er en del av Net3, en standard TCPIP implementasjonen for 4.4BSD.

En `mbuf` er en buffer struktur spesielt tilpasset nettverkssystemet. De er alltid like store, men innholdet i en `mbuf` kan variere ettersom hva den skal brukes til. Det er egne `mbuf` strukturer for pakke headere. Små datamengder som skal ut på nettverket kan puttes inni en egen`mbuf`. Dersom mer enn 208 byte skal sendes, brukes et ekstern `mbuf`. Denne inneholder en peker til et eksternt buffer. Et slikt eksternt buffer kan inneholde bare data eller både pakkeheader og data.

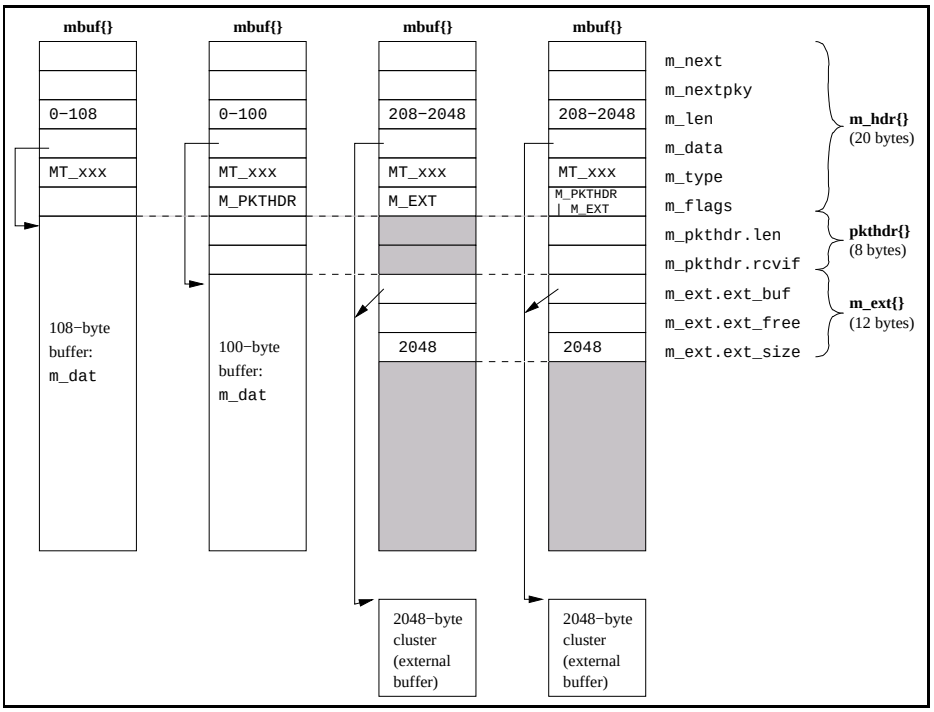
Det er vanlig at `mbuf` strukturer kjedes sammen i kjeder. Først en header mbuf, og så en eller flere `mbuf` strukturer med data eller `mbuf`-er med peker til eksterne data. Disse kjedene kan igjen lenkes sammen i køer før sending.

Allokering av `mbuf`-er skjer fra to minnepooler, og de frigjøres tilbake til poolene etter bruk. Det er en pool for `mbuf`-ene og en for de eksterne clusterne for lagring av større pakker.

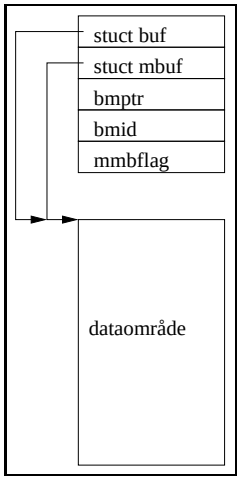
6.1.3 Andre minneløsninger i NETBSD

Store deler av minnehåndteringen har i løpet av de siste årene blitt skrevet om. Et nytt minnehåndteringssystem, *UVM*[15][16] har erstattet det gamle VM minnehåndteringssystemet.

Minnehåndteringen er blitt effektivisert på flere områder. Det som er mest relevant sett i lys av zero-copy minnehåndteringen er innføringen av konseptene *Page Loanout*, *Page Transfer* og *Map Entry Passing*. Page Loanout gitt en prosess mulighet til å "låne" minnesider read-only til kjernen eller en annen prosess. Page Transfer muliggjør å overføre hele minnesider fra kjernen til prosesser i brukerområdet uten å måtte kopiere disse. Map Entry Passing gir mulighet til å flytte



Figur 6.2: Forskjellige typer `mbuf` strukturer[43]



Figur 6.3: mmbuf

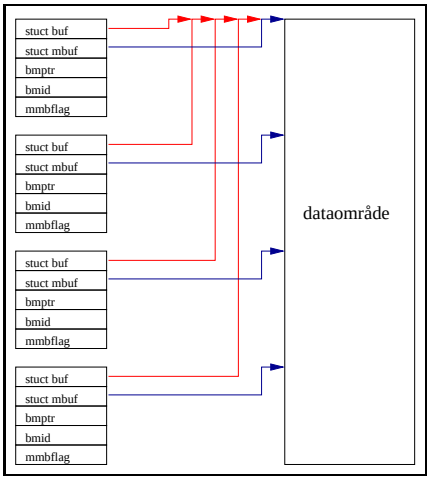
data mellom prosesser, eller la prosesser dele data gjennom å la begge ha virtuelle minnesider som referer de samme fysiske minnesidene.

6.2 mmbuf

Multimediabuffers (mmbuf)[10] er designet for å håndtere MM-data i MoD-tjeneren *Massively-Parallel And Real-time Storage (MARS)*[10][11][12].

Minnehåndteringen i mmbuf er basert på move-semantikken. Figur 6.3 illustrerer mmbuf-strukturen. Hver mmbuf inneholder en `mbuf` struktur og en `buf` struktur. På denne måten kan mmbuf-en brukes som en `buf` i disksystemet og en `mbuf` i nettverkssystemet. Dermed kan samme buffer brukes i både disksystemet og nettverkssystemet. Figur 6.4 viser hvordan flere mmbuf-er kan kobles sammen i et *mmbuf cluster*. Ved å kjede sammen flere mmbuf-er kan man lese større blokker en de man sender.

Dataflyten gjennom MARS starter med at en mmbuf allokeres fra en *mmbuf pool*. En mmbuf, eller den første mmbuf-en i en mmbuf cluster gis til disksystemet og data leses inn i bufferet. mmbuf-en/ene gis så til nettverkssystemet og sendes ut på nett. Når dataene er sendt frigjøres de tilbake til mmbuf poolen. Ved å bruke mmbuf får man dermed en in-kernel zero-copy datapath.



Figur 6.4: mmbuf cluster

6.3 Container Shipping

Container Shipping er designet av Pasquale et. al.[34] og bygger på virtuell flytting av data mellom domenene. Ved overføring av data fra et domene til et annet mappes minnet over til det nye domenet og fjernes fra avsender domene. Data flyttes altså fra et domene til et annet. Dette gir en mekanisme der alle domene i en I/O-Pipeline kan både lese og skrive til bufferet, uten at det er nødvendig med fysisk kopiering. I tillegg til dette skiller Container Shipping eksplisitt evnen til å flytte data og evnen til å aksessere dataene. En applikasjon som sender ut data, har ofte ikke engang behov for å lese dataene, bare å sende data fra filsystemet videre til nettverket. Det å mappe alle dataene fra filsystemet og inn i adresserommet til applikasjonen kan være kostbart og unødvendig hvis dataene bare skal sendes videre. De bruker derfor en variant av *Lat Sidemapping (Lazy Pagemapping.* Lat sidemapping brukes vanligvis om et system der sider mappes inn ved behov med en feilhåndterer som mapper inn sider som blir aksessert men som ikke er mappet inn. Siden en slik feilhåndterer kan gi forsinkelser som ikke kan planlegges og overhead også ved aksessering av sider som er mappet inn, velger Container shipping en metode der man eksplisitt må angi at sider skal mappes inn før de aksesseres. Bare de sidene som skal aksesseres mappes inn, og ikke nødvendigvis hele dataobjektet.

Ulempen med Container-shipping må være hvis at avsenderdomenet mister tilgangen til dataene. Et problem oppstår dersom samme data leses en gang fra disk og skal sendes flere ganger etter hverandre i løpet av et relativt kort tidsrom. Dersom en fil leses flere ganger i rask rekkefølge, må dataene leses fra disk hver gang og plasseres i en ny Container. Pasquale et al. foreslår at bufferne kan returneres til avsender etter bruk. Denne løsningen kan redusere dette problemet, men fjerner

det ikke nødvendigvis helt. Dette problemet reduserer fordelen man oppnår ved å buffre ofte aksesserte filer. Siden diskaksess kan være en flaskehals kan dette gi relativt store tap av ytelse. Når det gjelder videostrømmer vil dette trolig allikevel ikke skje, da samme data sjelden aksesseres med korte tidsintervaller. Ser man derimot på ytelsen for en webtjener kan dette gi vesentlig redusert ytelse i forhold til andre buffermekanismer.

6.4 IO-Lite

IO-Lite er som navnet tilsier er bufferstrategi beregnet på I/O fra Pai et. al.[33]. IO-lite bygger videre på *fbufs* fra Druschel et. al.[21]. IO-Lite er en CoW mekanisme som tillater at data skrevet av filsystemet flyter gjennom OS-et og leses av nettverkssystemet uten at kopiering er nødvendig.

IO-Lite kombinerer re-mapping av sider med delte read-only buffere. De uforanderlige bufferne aksesseres indirekte via dataobjekter kalt buffer aggregater. Bufferaggregatene inneholder pekere til de områdene av bufferne som inngår i dataene bufferaggregatet representerer. Flere bufferaggregat kan peke til det samme bufferet. Når data sendes fra en prosess til en annen gjøres dette ved at bufferaggregatet kopieres over til mottakeren, mens minnesidene bufferaggregatet peker på gjøres lesbar for mottaker prosessen. Siden bufferne ikke kan endres, kan alle prosessene som har mottatt bufferaggregater benytte samme buffer. Dersom en prosess skriver til et buffer, blir et nytt buffer allokert, og endringene legges der. Det nye bufferet lenkes inn i bufferaggregatet, slik at det ikke er nødvendig å kopiere hele bufferet, kun allokere plass til endringene.

Bufferne er samlet i buffer-pooler. Hver pool har en liste over de prosessene som kan lese bufferne i poolen. Dette passer godt inn i prinsippet med I/O Pipeline fra Pasquale et al.[34]. Hvert buffer inneholder en oversikt over hvilke prosesser og hvilke bufferaggregater som peker på bufferet. Når det ikke lenger er pekere til bufferet blir ikke bufferet deallokert. I stedet puttes det tilbake i poolen av ledig buffere. En annen mulighet er at prosessen i begynnelsen av I/O Pipelinen får tildelt skriveaksess til bufferet når den er den eneste som aksesserer dette. Dermed kan bufferet fylles på ny.

IO-Lite oppnår zero-copy ved lesing. Den minker også tiden som brukes til å allokering og deallokering av buffere, da bufferne bare legges tilbake i poolen når de er tomme i stedet for å deallokeres. De sparer minne ved at bruken av identiske kopier av samme data i minnet minimaliseres. IO-Lite krever imidlertid resurser for å kopiere bufferaggregater til nye prosesser hver gang data flyttes til en ny prosess. Dette krever også remapping av sider inn i den nye prosessens adresserom. Begge deler krever en del ressurser.

6.5 Genie

Genie, fra Brustolini et. al.[9][7][8][6] implementerer en mekanisme de kaller *Emulated Copy*. Dette er en transparent buffermekanisme som gir en effektiv bufferhåndtering mens copy-semantikken opprettholdes. Målet med GENIE er altså å lage en mekanisme for minnehåndtering som har samme egenskaper som allerede eksisterende mekanismer når det gjelder dataintegritet og grensesnitt.

Viktig i GENIE er *Midlertidig kopi ved skriving (Transient copy-on-write, TCoW)*. TCoW bygger videre på kopi ved skriving. TCoW benytter seg av COW, men kun i et begrenset tidsrom. Når en applikasjon sender en kopi til output, setter TCoW mekanismen i gang. Dersom applikasjonen skriver til bufferet før output er ferdig, vil systemet lage en kopi av de minnesidene som berøres slik at ikke outputdataene blir ødelagt. GENIE håndterer dette ved å holde rede på antall output og input operasjoner for hvert buffer. TCoW aktiviseres dersom telleren for outputreferanser er større enn null.

GENIE gir normalt langt bedre ytelse enn ved normal bufferkopiering, med samme semantikk og dataintegritet. Ved å bruke kopiering ved små datamengder, og remapping ved større datamengder oppnår de like gode egenskaper som kopiering ved små datamengder og bedre ved større datamengder. GENIE tilbyr flere egenskaper til applikasjonene enn vanlig kopieringsalgoritmer via tilleggs API-er.

GENIE modifiserer i tillegg sideutbyttingsalgorimtene slik at sider som er klar for input ikke byttes ut.

6.6 Sett i sammenheng med INSTANCE

Vi tar iO Pipeline fra fbufs. Vi vet hvor dataene skal ved bufferallokering. Ide med buf og mbuf fra mmbuf. Stor buffere fra Tiger?

6.6.1 Det vanlige minnehåndteringen i NETBSD

Både Page Loanout og Page Transfer gir muligheter til Zero-Copy bufferhåndtering. Begge disse mekanismene forutsetter imidlertid at data flyttes fra kjerneområde til brukerområde, eller internt i brukerområde. Vi har bestemt oss for å lage en in-kernel datapath så dette utelukker dette.

Map Entry Passing kan muligens benyttes til vårt formål. Vi kan tenke oss at vi har tilgang på de samme dataene i den samme fysiske siden fra disksystemet, applikasjonen og nettverkssystemet. Problemet er hvordan applikasjonen skal kunne

kontrollerer dataflyten. Man trenger en metode for å styre disse dataene. Vi trenger mer enn et delt minneområde for å klare å styre dataflyten.

De nye metodene i UVM er et verktøy for å lage nye zero-copy datapath-er og vil trolig kunne forbedre de eksisterende datapathene brukt av vanlig lese og skrive kall en hel del. For en optimalisert og spesialisert løsning som den vi er ute etter blir det for generelt.

6.6.2 mmbuf

Mmbuf er en meget lovende idé. Håndtering av dataflyten ved å kombinere buf og mbuf strukturer som peker på samme minneområde løser problemene med håndteringen av dataene vi etterspurte i avsnitt 6.6.1. mmbuf gir en in-kernel datapath og det passer bra inn i vårt konsept. Ulempen med mmbuf i vår sammenheng er at fordi mmbuf baserer seg på en move modell får vi en kostnad ved å allokere og frigjøre mmbuf-er fra mmbuf-poolen. Vi mener at siden vi ønsker å støtte kontinuerlige datastrømmer vil vi spare på å allokere buffrer fast i stedet for å allokere og frigjøre bufferne under bruk.

6.6.3 Container Shipping

Container Shipping er interessant metode. Prinsippet om at eierskap til data ikke er direkte koblet til muligheten til å aksessere dataene passer godt med vår idé om at applikasjonen ikke har behov for å aksessere dataene. Dette gir en løsning på problemet med move-semantikken vi beskrev i avsnitt 5.10, nemlig at vi må remappe minnet frem og tilbake mellom kjernen og brukerområde. Container Shippings lazy pagemapping gjør at vi slipper denne kostnaden siden applikasjonen ikke har behov for å aksessere dataene. Imidlertid har også Container shipping en kostnad ved å allokere og frigjøre minne for hver gang gjennom I/O Pathway-en da de bruker en move modell.

6.6.4 IO-Lite

IO-Lite er en CoW implementasjon og bruker således ressurser på å kunne håndtere eventuelle skriveoperasjoner inn i det felles bufferet. Siden vi i avsnitt 5.10 avgjorde at vi ikke skulle bruke CoW da dette ikke er nødvendig i vår spesielle applikasjon, mener vi at IO-Lite ikke er aktuell som buffermekanisme i INSTANCE-noden.

6.6.5 Genie

Genies TCoW-mekanisme kan synes bedre en CoW-mekanismen i IO-Lite. Det er mulig å la diskhåndteringssystemet beholde bufferet, og skrive til bufferet på nytt når nettverkssystemet ikke lenger aksesserer dette. Allikevel ønsker vi å prøve å lage en mindre generell bufferhåndteringsmekanisme for INSTANCE-noden.

Del II

Konsepter, implementasjon og resultater

Kapittel 7

Konsepter

I dette kapittelet skal vi beskrive hva vi gjør i INSTANCE-noden for å flytte fra disk og ut til klientene. Vi ønsker å gjøre dette på en enkel og rask måte, samtidig som vi ønsker best mulig kvalitet. Målet er også å lage en skalerbar løsning slik at INSTANCE-noden skal kunne brukes til alt fra enkle lokale løsninger, til løsninger der store mengder data skal kunne spres til et stort antall klienter. Alt dette ønsker vi å kunne gjøre med enkel hyllevare utstyr til en lavest mulig pris.

For å kunne komme frem til en god løsning går vi i avsnitt 7.1 først gjennom forutsetningene som ligger til grunn for konseptet. Disse analyserer vi for krav vi må tilfredsstille og egenskaper vi kan utnytte.

Vi ser så på de store linjene i konseptet i avsnitt 7.2. I tillegg til tradisjonell VoD, med en til en forbidelser, ønsker vi å beskrive et konsept der et tilnærmet ubegrenset antall klienter kan betjenes av én INSTANCE-node. Dette fordi et av målene med INSTANCE-prosjektet er å designe en *effektiv* enhet. Her viser vi derfor hvordan vi kan benytte de samme dataene til å tilfredsstille et stort antall samtidige klienter ved å benytte en multicast teknikk som GDB[22]. Distribusjon av data i nettverket er ikke direkte tema i denne oppgaven. Når vi allikevel går inn på dette er det for å vise at det er mulig å betjene klientene med kontinuerlige datastrømmer. Denne forutsetningen er viktig for det videre designet av bufferhåndteringen, så vi velger derfor å trekke opp de store linjene i distribusjonen før vi går inn på detaljene for selve bufferhåndteringen.

Videre i avsnitt 7.3 viser vi hvordan data håndteres internt i INSTANCE-noden for å oppnå de mål vi allerede har stilt oss. Vi kommer til at ved å tilpasse bufferhåndteringen til nettopp varige, kontinuerlige datastrømmer kan vi oppnå en effektiv bufferhåndtering som egner seg godt for INSTANCE-noden. Dette konseptet for bufferhåndtering i INSTANCE-noden er ikke ment å erstatte all bufferhåndtering i noden. De originale buffermekanismene i NETBSD er utviklet for å håndtere mere

tradisjonelle datahåndtering som håndtering av små spredde repeterende leseoperasjoner (se avsnitt 6.1). De nye prinsippene for bufferhåndtering er designet slik at de skal kunne fungere side ved side med de gamle metodene og eventuelle andre nye metoder.

I avsnitt 7.4 viser vi hvordan distribusjonsmetoden og den interne minnehåndteringen sammen gjør INSTANCE-noden fullstendig skalerbar. Vi avslutter med å sammenligne de konseptene vi kommer frem til her med konseptene til tilsvarende løsninger som vi beskrev i kapittel 6.

7.1 Forutsetninger

Når vi skal lage en modell for bufferhåndteringen i INSTANCE-noden er det en del forutsetninger som ligger til grunn. Disse forutsetningene gir visse egenskaper til datastrømmene vi håndterer som vi kan bruke til å optimalisere og effektivisere bufferhåndteringen. I de neste avsnittene tar vi for oss følgende forutsetninger:

- INSTANCE-noden er en spesialisert enhet.
- INSTANCE-noden håndterer bestemte datatyper.
- INSTANCE-noden er beregnet for populære data.
- Dataenes levetid i INSTANCE-konesptet.
- Forutsigbarhet i datastrømmene.
- Egenskaper ved dagens datamaskiner og nettverk.
- Krav til kvalitet i distribusjonen.

7.1.1 INSTANCE-noden, en spesialisert enhet

Vi baserer oss på at en INSTANCE-node er en spesialisert enhet. Tidligere var ruting av data i nettverket gjerne en av mange tjenester i en tjener. I de aller fleste tilfeller er dette nå en tjeneste som utføres av spesiell hardware designet for å rute data raskt og rimelig. På samme måte bruker vi i INSTANCE-konseptet en dedikert node til distribusjon av data i stedet for å la dette være en av mange tjenester i en vanlig tjener. Dette gir oss muligheter til å optimalisere INSTANCE-noden kun for dette formålet.

7.1.2 Datatyper i INSTANCE

I INSTANCE arbeider vi med forskjellige typer av kontinuerlige strømmer med MM-data. Andre typer data som tekstdokumenter og bilder kan bedre håndteres og distribueres på andre måter. Vi konsentrerer oss derfor her om distribusjon av kontinuerlige datastrømmer.

7.1.3 Dataenes popularitet

I tillegg til å bruke tradisjonell VoD er INSTANCE-noden beregnet til å distribuere populære data med multicast. Med populære data mener vi data der det er mange brukere som i løpet av en periode ønsker å motta dataene. kan antas at noen data er langt mer populære enn andre. Dan et. al. hevder at fordelingen av filmer brukerne av VoD ser, trolig slett ikke er jevn, men at et fåtall filmer utgjør hovedparten av nedlastingen[17]. De begrunner dette med undersøkelser av populariteten til leievideoer. Senere konkluderer også Bär et. al. med det samme basert på ytterlige data[3]. Bär et. al. finner at populariteten er stor når filmene er nye. Populariteten avtar så relativt raskt med tiden. Ingen av disse undersøkelsene bygger på et virkelig VoD eller MoD system, men det er grunn til å anta at oppførselen i det minste vil ligne på det disse undersøkelsene viser. I en situasjon med videofilmer vil antallet filmer være en faktor som begrenser hvor mange som kan se de mest populære filmene. De som ikke får se dem da, vil vente for så å se dem senere. I INSTANCE-konseptet vil det i prinsippet ikke være noen begrensninger på hvor mange brukere som kan se den samme filmen. Det vil derfor ikke være unaturlig å tenke seg at toppene i popularitet muligens kan bli enda større enn tallene for leievideoer.

7.1.4 Dataenes levetid

Strukturen med en INSTANCE-node som distribuere data fra en INSTANCE-tjener stiller et krav til dataene. Dataene må være egnet for mellomlagring utenfor tjeneren. Dette betyr at INSTANCE-konseptet hovedsakelig er beregnet på data med en viss levetid. Med dette mener vi at dataene ikke oppdateres kontinuerlig, men i stedet oppdateres ved behov. For kontinuerlig oppdaterte data, for eksempel kontinuerlige radio og TV sendinger egner multicast direkte fra tjeneren seg langt bedre.

7.1.5 Forutsigbarhet

Forutsetningen i avsnitt 7.1.4 utleder enda en viktig forutsetning. Siden dataene ikke oppdateres, kan vi regne ut en del tall om dataene før vi begynner å distribuere

dem. Data som bitrater og størrelse kan enkelt beregnes. Disse beregningene kan gjøres i INSTANCE-noden, eller de kan gjøres av tjeneren og overføres til noden sammen med dataene. Vi kommer nærmere tilbake til dette i avsnitt 7.2.3.

7.1.6 Egenskaper ved dagens datamaskiner og nettverk

Egenskapene til den hardware vi benytter i INSTANCE-noden er også noe som vi må ta hensyn til ved design av konseptet. Datamaskinene har til nå utviklet seg svært raskt. Ytelse, pris og kapasitet har utviklet seg meget raskt. En moderne hjemmedatamaskin leveres minimum med en harddisk som har plass til to spillefilmer med meget god kvalitet på lyd og bilde. Kapasiteten i nettverket til vanlige forbrukere har ikke vokst i like hurtig tempo, men teknologien har nå muliggjort nett med stor kapasitet. Alt tyder på at denne raske utviklingen vil fortsette.

PC-enes høye kapasitet betyr at vi i liten grad behøver å designe konseptet for å spare resurser i klientene. Siden nettverket ofte er en knapp ressurs ønsker å belaste nettverket så lite som mulig. Selve INSTANCE-noden ønsker vi å designe for å utnytte kapasiteten i noden best mulig, uten at noden behøver å bygge en node av spesielt dyre komponenter. Det ønskede resultatet er å tilby flest mulig samtidige brukere tjenester fra noden.

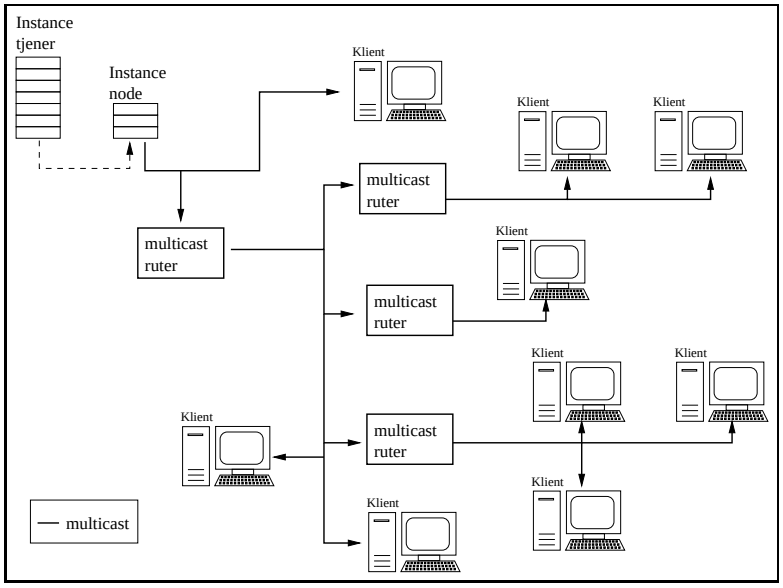
7.1.7 Dataene skal distribueres med et akseptabelt kvalitetsnivå

Det er en forutsetning av dataene skal distribueres med et visst QoS. Dette stiller både krav til den overordnede distribusjonsteknikken, og designet av selve bufferhåndteringen.

7.2 Makroperspektivet

I tillegg til tradisjonell VoD ønsker vi å kunne tilfredstille mange samtidige klienter med en form for multicast distribusjon. Dette kapittelet tar derfor for seg et konsept for distribusjon basert på prinsippene i GDB[22]. Vi skisserer hvordan dataene flyter i INSTANCE-modellen og hvordan disse datastrømmene håndteres. Vi beskriver hvordan data og metadata skilles og distribueres på forskjellig vis.

Som vi beskrev i kapittel 2 skiller INSTANCE-modellen seg fra tradisjonell datahåndtering ved at dataene ikke flyter direkte mellom tjener og klient. Dataene mellomlagres i stedet i INSTANCE-noden på samme måte som en web-cache[28][23] lagrer deler av dataene på Internett slik at ikke alle dataene må transporteres fra tjener til klient hver gang.



Figur 7.1: Bruk av multicast for å distribuere data fra INSTANCE-noden

Ved å bruke multicast som distribusjonsmetode for dataene kan et ubegrenset antall klienter betjenes med en INSTANCE-node. Figur 7.1 illustrerer hvordan dataene i nettet.

En INSTANCE-node sender hele tiden ut de samme strømmene. Det er opp til nettverket hvilke strømmer som skal hvor i nettverket. Dersom en strøm ikke er etterspurt i det hele tatt, vil den allikevel bli sendt ut fra noden. Den vil ikke bli videresendt av den første multicastruteren, da ingen lenger ut i nettverket etterspør den. På denne måten er det nettverket, og ikke INSTANCE-noden som har arbeidet med å styre datastrømmen.

7.2.1 Dataflyten

Til selve distribusjonen av data henter vi mye fra GDB[22]. Vi bruker multicast for distribusjon og skiller data fra metadata for å unngå flaskehalser.

Tankene bak distribusjonsmodellen ble først presentert i Fra GDB henter vi idene om hvordan en datastrøm kan inndeles slik at vi kan bruke et minimum av multicast datastrømmer til å nå et maksimalt antall klienter med raskest mulig delay. GDB beskrives nærmere i avsnitt 3.4.4.

I GDB er det behov for lagringskapasitet hos klienten. Klienten må også kunne ta imot data raskere enn avspillingen. Med størrelsen på disk og minne i de PC-

er som selges i dag burde lagringskapasitet være et lite problem. Båndbredden på nettverket er også raskt økende.

7.2.2 RSVP

For å reservere ressurser i nettverket vil vi benytte RSVP[44]. RSVP er beskrevet i avsnitt 4.3. På denne måten vil vi få en distribusjon med et brukbart QoS nivå.

7.2.3 Metadata

Til nå har vi sett på hvordan data strømmer fra INSTANCE-noden til klientene. GDB sier at dataene deles opp i understrømmer, men ingenting om hvordan klientene vet *hvilke* understrømmer som inneholder hvilke data. Informasjon om hvilke strømmer som inneholder hvilke data samt informasjon om egenskapene til datastrømmene kaller vi *Metadata*. Disse metadataene må beskrive:

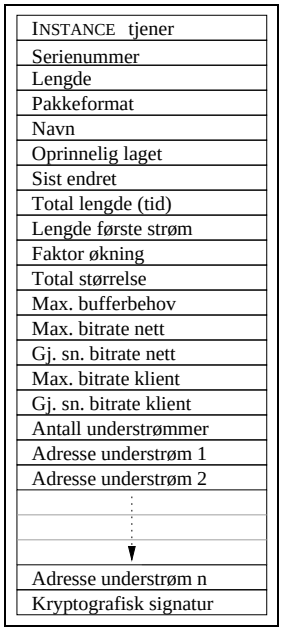
- Hvor dataene er
- Hva slags data det er.
- Hvilke krav som stilles til klienten
- Hvordan dataene skal håndteres

Figur 7.2 gir et inntrykk av hva slags metadata vi tenker oss blir nødvendig i INSTANCE-kopnseptet. Detaljene i nettverkssystemet er imidlertid ikke tema i denne oppgaven, derfor overlater vi dette til andre som arbeider videre med hele INSTANCE-konseptet.

7.2.4 Distribusjon av metadata

Distribusjonen av metadata skaper en del problemer. Belastningen på INSTANCE-tjener / klient og nettverket må vurderes slik at det ikke oppstår flaskehalser i distribusjonen av metadataene.

En mulighet er å bruke multicast også til distribusjonen av dette. Dette vil imidlertid neppe være fornuftig med mindre antallet klienter er meget stort. Dataene ville måtte distribueres svært ofte, slik at ikke klientene må vente uforholdsmessig lenge før de mottar metadataene. Denne løsningen vil derfor trolig belaste nettverket uforholdsmessig mye.

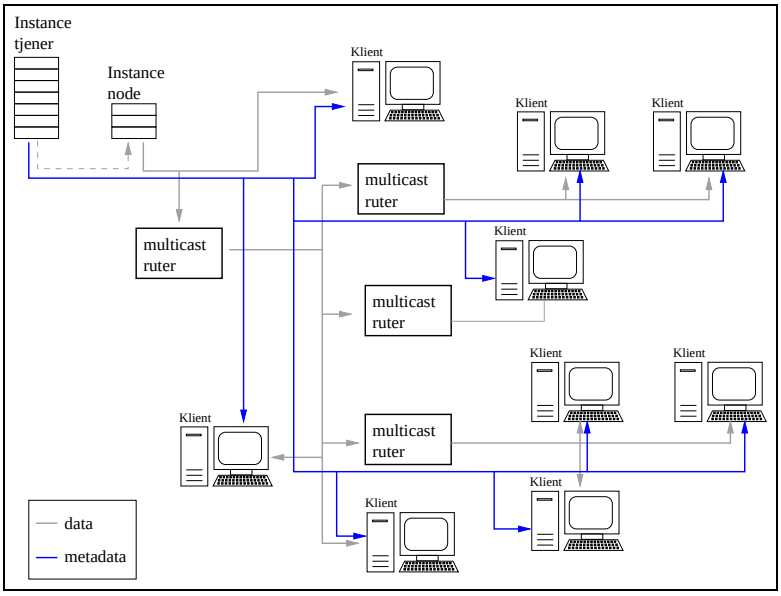


Figur 7.2: Innholdet i en pakke med metadata

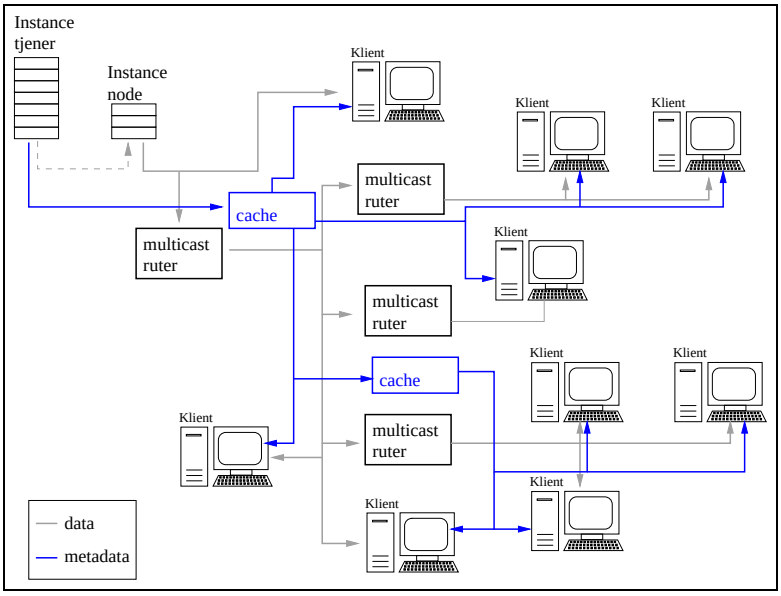
Den andre muligheten er å bruke unicast for metadataene. Det er ikke ønskelig å la denne trafikken gå fra INSTANCE-noden. Dette ville innføre en belastning på noden som ikke kan beregnes på forhånd. Dette kan igjen føre til at noden ikke klarer å yte tilstrekkelig QoS på dens egentlig oppgave, distribusjon av datastrømmer. Vi plasserer derfor denne oppgaven i INSTANCE-tjeneren. Dette vises i figur 7.3.

Det er også mulig å begrense belastningen på INSTANCE-tjeneren. Vi kan utnytte det faktum at datastrømmene endres relativt sjelden. Det gir igjen at metadataene endres sjelden. Dermed åpnes det for å benytte en metode som er kjent fra World Wide Web, distribuert caching av data i nettverket[28][23]. Ved å mellomlagre metadataene forskjellige steder i nettet vil klientene sjelden behøve å kontakte tjeneren direkte. Figur 7.4 viser denne idéen.

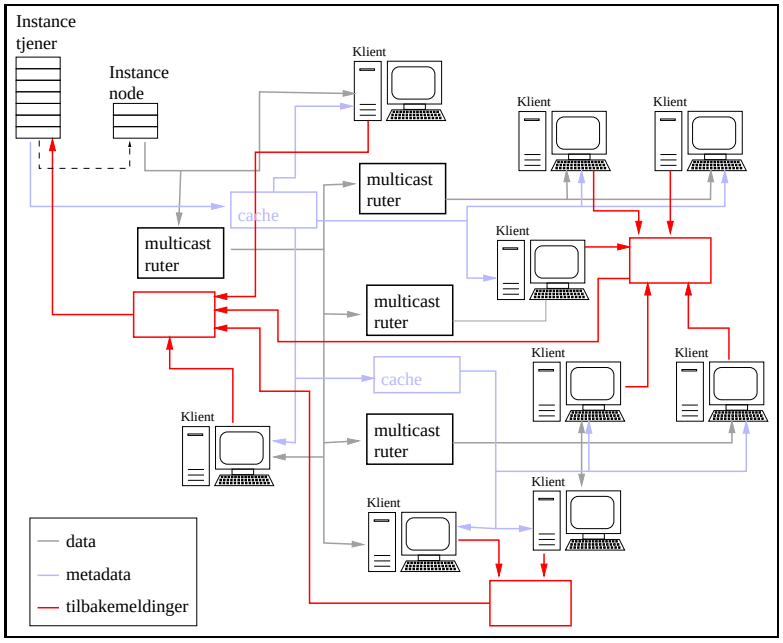
Det er et viktig skille mellom tradisjonell webcaching og bruk av cache for metadata i INSTANCE-konseptet. I en tradisjonell webcache kan man tolerere at dataene i cachen til tider ikke er fullstendig oppdatert. De vil allikevel ofte ha en viss verdi. Dersom dataene i en INSTANCE-cache er for gamle vil dette i de fleste tilfeller føre til at klienten ikke klarer å motta hele datastrømmen. Det er derfor ønskelig med en mekanisme der tjeneren kan spre nye metadata til alle cachene. Et brukbart alternativ er igjen multicast.



Figur 7.3: Distribusjon av metadata.



Figur 7.4: Mellomlagring av metadata i nettet (caching).



Figur 7.5: Håndtering av responser fra klientene

7.2.5 Tilbakeflyt av informasjon

Vi har til nå beskrevet et system der INSTANCE-tjeneren distribuerer data til klientene via noden uten noen tilbakemeldinger fra klientene. Ofte vil det være ønskelig med slike tilbakemeldinger, ikke minst fordi INSTANCE-konseptet baserer seg på at dataene som distribueres er populære.

Dersom disse dataene skulle gå direkte fra klientene kunne dette gi en storm av meldinger til tjeneren. For å unngå dette trengs det et nett av enheter som samler responser fra klientene i oppsummeringer som sendes oppover i treet og samles i tjeneren.

7.3 Mikroperspektivet

Vi har slått fast i avsnitt 7.1 at INSTANCE-noden er en spesialisert enhet. Dette gir oss muligheten til å tilpasse OS-et i INSTANCE-noden til de oppgavene noden skal utføre. Vi tar utgangspunkt i OS-et NetBSD og lager våre egne metoder for bufferhåndtering spesielt tilpasset kontinuerlige datastrømmer i INSTANCE-noden. I dette avsnittet ser vi nærmere på hvordan håndteringen av data internt i noden kan optimaliseres innenfor de gitte forutsetningene.

Minnehåndteringen er tilpasset streaming av data fra disk til nettverk. Vi har tilpasset løsningen til nodens normale arbeid, som nettopp er å flytte data fra disk til nett. Overføring av data fra INSTANCE-tjeneren til INSTANCE-noden, er arbeid som kan gjøres med vanlige mekanismer, og i eventuell ledig tid på noden.

Mekanismen vi foreslår er beregnet på å håndtere de kontinuerlige datastrømmene. Annen datahåndtering i INSTANCE-noden blir trolig best utført med andre eksisterende mekanismer eller andre nye mekanismer.

7.3.1 Forutsetninger for Bufferhåndteringen

Også ved design av den interne bufferhåndteringen er det fornuftig å ta utgangspunkt forutsetningene for INSTANCE-konsepetet fra avsnitt 7.1:

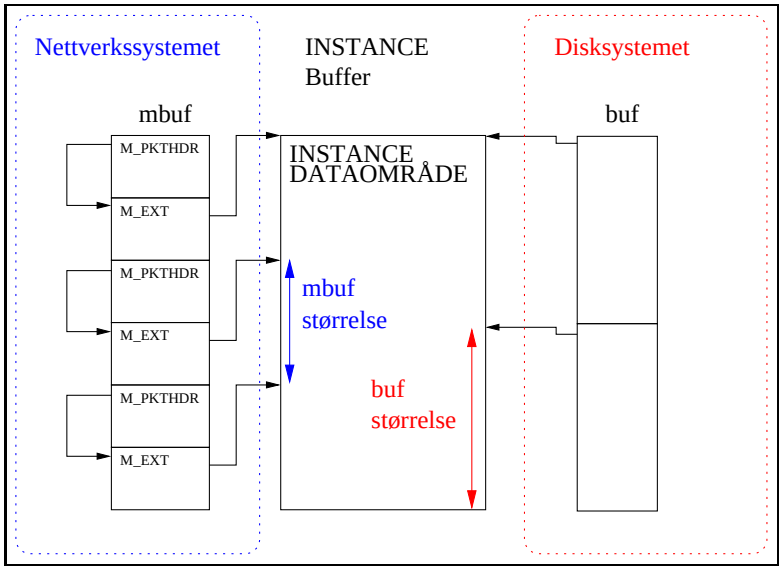
- INSTANCE-noden er en spesialisert enhet.
- INSTANCE-noden håndterer bestemte datatyper.
- INSTANCE-noden er beregnet for populære data.
- Dataenes levetid i INSTANCE-konesptet.
- Forutsigbarhet i datastrømmene.
- Egenskaper ved dagens datamaskiner og nettverk.
- Krav til kvalitet i distribusjonen.

7.3.2 Dataflyten i INSTANCE-noden og INSTANCE-strømmen

Vi går her inn på hvordan data flyter fra disksystemet, gjennom INSTANCE-bufferne og ut gjennom nettverkssystemet. Vi ser her på de generelle prinsippene. Vi kommer tilbake til detaljene i kapittel om implementasjon (kapittel 8), nærmere bestemt i avsnitt 8.6.1.

Dataflyten i INSTANCE-noden har sterke likhetstrekk med *I/O Pipeline* fra Container Shipping[34]. Vi har som en viktig forutsetning at datastrømmene er forutsigbare. Som i Container Shipping vet vi at dataene vil følge en forhåndsdefinert vei gjennom systemet. En datastrøm, slik den er beskrevet i makroperspektivet, vil internt i maskinen alltid flyte fra den samme filen og til den samme nettadressen. Denne interne flyten kaller vi en INSTANCE-*strøm*.

På samme måte som mmbuf-ene i MARS[12] (se avsnitt 6.2) ser vi bufferne i INSTANCE-noden som tilhørende flere undersystemer på en gang. Figur 7.6 viser dette. Vi ser hvordan mbuf strukturer i nettverkssystemet og buf strukturer i



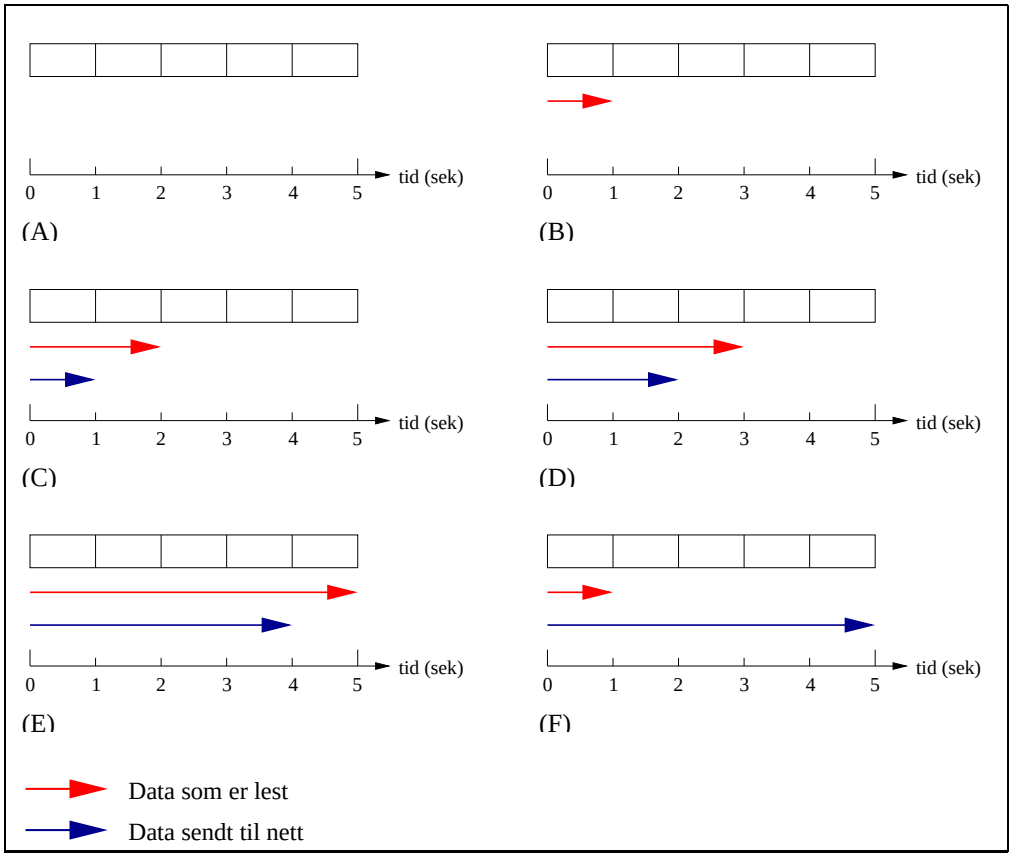
Figur 7.6: Illustrasjon av hvordan INSTANCE-bufferet eksisterer samtidig både i nettverkssystemet og disksystemet

disksystemet samtidig peker på dataene i et INSTANCE-buffer. Dette kan sees på som en avart av ILP[14]. ILP tar i hovedsak for seg prosessering av flere samtidige oppgaver, mens vi gjør som i MARS og lar data befinne seg på flere steder i systemet på en gang ved å la buffere opptre som både disk buffer for disksystemet og nettverksbuffer for nettverkssystemet.

I avsnitt 5.9.2 beskrev vi prinsippene fra Chang et. al.[13] om Konstant Data Lengde (CDL) og Konstant Tids Lengde (CTL). En forutsetning som ikke er vurdert her er om bufferstørrelsen nødvendigvis må være den samme sett fra forskjellig deler av OS-et. I en "vanlig" implementasjon av dataflyt i et OS (se f.eks. avsnitt 6.1) er det slik at størrelsen på bufferne i disksystemet ofte er forskjellig fra størrelsen i nettverkssystemet. Dette skyldes rett og slett at optimal størrelse er forskjellig i de to delene av OS-et. I mmbuf[10] gjør de det mulig med større `buf` strukturer enn `mbuf` strukturer ved at flere mmbufer kobles sammen i et mmbuf cluster (se figur 6.4. Figur 7.6 viser at i INSTANCE-konseptet er størrelsen av dataområdet som pekes på av en mbuf er forskjellig fra området en buf peker på. Dette gjør at bufferstørrelsen disksystemet og bufferstørrelsen i nettverkssystemet kan optimaliseres uavhengig av hverandre.

En annen fordel er at vi lettere kan håndtere bufferstrømmer med variabel bitrate. Vi allokerer buffere slik at vi har nok minne til å tilfredsstille toppene i bufferbehov.

Vurderer vi dette i forhold til CDL og CTL ser vi dette: En INSTANCE-strøm ser fra disksystemet ut til å ha CDL egenskaper, med andre ord at det leses en fast meng-



Figur 7.7: Dataflyt i INSTANCE-noden

de data til varierende tidspunkt. Fra nettverkssystemet ser det i stedet ut som om om bufferet har CTL egenskaper, dvs. at varierende mengde data sendes ut til faste tider. Dette betyr at disksystemet kan optimaliseres til å lese data raskest mulig mens dataene sendes ut i nettverket til mest mulig riktig tid. Dette betyr at vi kan optimalisere disksystemet til å lese store datamengder i hver leseoperasjon. Nettverksdelen er derimot optimalisert for å få en jevnest mulig bitrate ut på nettverket, og får å unngå at nettverkssystemet må fragmentere mbuf-ene de får levert.

Figur 7.7 viser hvordan data flyter gjennom INSTANCE-noden. Figur 7.7 (A) viser situasjonen før vi begynner å sende. Det første sekundet leses et stykke, men ingenting sendes ennå (7.7 (B)). Sekund 2 leses mer, mens det som ble lest sist sendes(7.7 (C)). Dette repeteres så videre (7.7 (D)).

Figur 7.7 (E) viser situasjonen etter 5 sekunder. Når vi når slutten av dataene starter vi å lese fra begynnelsen igjen (7.7 (F)).

7.3.3 Allokering av buffere

Vi har tatt utgangspunkt i to av forutsetningene. Den ene er det at forutsigbarheten i INSTANCE-konseptet gjør at vi kan definere hver datastrøm som en INSTANCE-strøm. Det andre er tilgjengeligheten og prisen på minne i dag. I forhold til prisen på f.eks. raskere disker, er kostnaden ved å øke minne i dagens maskiner liten.

Vi har valgt å allokere bufferne fast. Vi allokerer også faste mbuf og buf strukturer. Som vi så i avsnitt 6.1 blir både mbuf-ene og buf-ene normalt allokert og deallokert før og etter hver disklesing og før og etter hver sending av nettverkspakker. Bufferne er allokert i kjernens minneområde. De er såkalt "wired", hvilket vil se at de ikke kan legges ut på disk av det virtuelle minnesystemet. En slik utkopiering til disk ville være direkte ødeleggende for minnehåndteringen i INSTANCE-noden.

En av ulempene ved å allokere bufferne i kjernen er at de ikke er direkte skrivbare for brukerprosessene. Dette ville selvsagt være uakseptabel dersom vi laget en generell mekanisme for minnehåndtering. I INSTANCE-konseptet er vi derimot ute etter å lage en optimalisert løsning kun for streaming fra disk til nettverk. Vi har derfor ikke implementert noen mekanisme for skriving fra brukerprosesser til INSTANCE-bufferne. Her vil andre allerede eksisterende mekanismer NETBSD egne seg bedre.

7.3.4 Bufferstørrelse

Valget for bufferstørrelse i en datastrøm går tradisjonelt mellom CTL og CDL (se avsnitt 5.9.2). I INSTANCE-konseptet Har vi valgt en hybrid av disse to. I avsnitt 7.3.2 så vi at vi i INSTANCE-konseptet lar et og samme buffer opptre i forskjellige deler av kjernen samtidig. Vi har så valgt en løsning der disse bufferne ikke ser ut til å ha samme størrelse for disksystemet og nettverkssystemet.

Vi allokerer buffere i forhold til maksimalt behov i en bestemt tidsperiode. Vi har valgt å ta utgangspunkt i perioder på ett sekund. Fordelen er at denne løsningen garanterer at det alltid vil være nok plass i bufferet. Vi slipper også det arbeidet det betyr å allokere og deallokere nødvendig bufferkapasitet.

Senest ved overføring av data fra INSTANCE-tjeneren til INSTANCE-noden beregnes maksimal datarate i en periode. Denne informasjonen lagres som metadata sammen med selve dataene og brukes når sending av datastrømmen initieres.

7.3.5 Blokkstørrelse og pakkestørrelse

Som vi beskrev i avsnitt 7.3.2 er dataene i et INSTANCE-buffer samtidig tilgjengelig i både disksystemet og nettverkssystemet gjennom henholdsvis `buf` og `mbuf`

strukturer. Størrelsen på disse er uavhengige av hverandre.

Maksimal størrelse på en `buf`'s dataområde er 64kB. Minste størrelse er lik blokkstørrelsen i filsystemet, siden lesing alltid foregår i blokker og minste antall blokker som kan leses nødvendigvis er én. Det er mulig å bruke `buf`-er med variabel størrelse. Disksystemet kan nemlig lese flere blokker om gangen, såkalt cluster read. Det er imidlertid aldri mulig å garantere lesing av mer en en blokk om gangen. Vi ville dermed hverken ha forutsigbar størrelse på `buf`-ene og dermed heller ikke vite hvor mange vi trenger. Siden vi ønsker maksimal størrelse på hver read og fast allokerte `buf` strukturer er større diskblokker nyttig. Vi kan alltid garantere lesing av en blokk, og med blokker på 64kB vil vi alltid klare å lese maksimalt av det det er plass til i en `buf`. Blokkstørrelsen i filsystemet settes idet filsystemet formateres.

Størrelsen på `mbuf` strukturene er et annet spørsmål. Minste interessante størrelse er maksimal pakkestørrelsen i nettverket minus størrelsene på pakkeheaderne som må legges til.

7.3.6 Allokering av buffere

Vi allokerer bufferne i det distribusjon av data starter og frigjør dem når datastrømmen ikke lenger skal sendes ut. Siden dataene har relativt lang varighet er dette er noe som skjer relativt sjelden.

Bufferne allokeres fra kjernens malloc-område. Størrelsen på dette området må tilpasses behovet til INSTANCE-konseptet ved kompilering av kjernen. De er "wired-down", dvs. at de ikke kan flyttes nover til disk av det virtuelle minnesystemet. En slik utkopiering til disk ville ikke være spesielt gunstig.

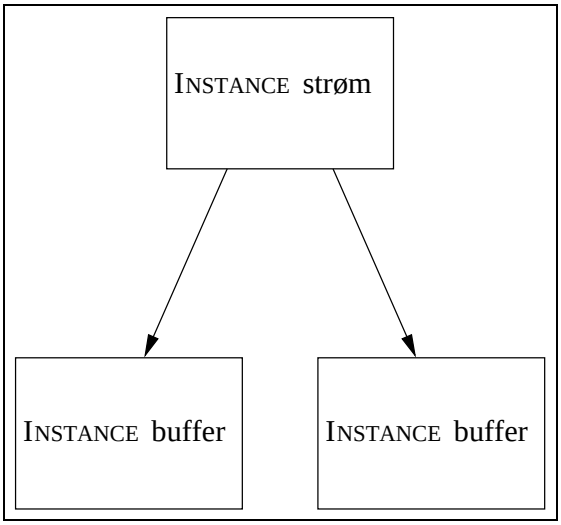
7.3.7 Doble buffere

For å kunne både lese fra disk og sende ut data på samme tid har vi valgt å allokere to buffere til hver datastrøm. Slik kan vi tilnærmet samtidig lese fra disk til det ene bufferet mens vi sender data ut på nettverket fra det andre.

Vi har derfor laget en struktur, en INSTANCE-strøm, for å holde rede på informasjon om datastrømmen. Hver INSTANCE-strøm har så to INSTANCE-buffere som inneholder selve dataene. Figur 7.8 illustrerer dette.

7.3.8 Tidsperioden

For å synkronisere datastrømmene i INSTANCE-noden treng det en scheduleringsmekainsme. Denne er dessverre ikke klar, men vi antar at vi kan gå ut fra at den



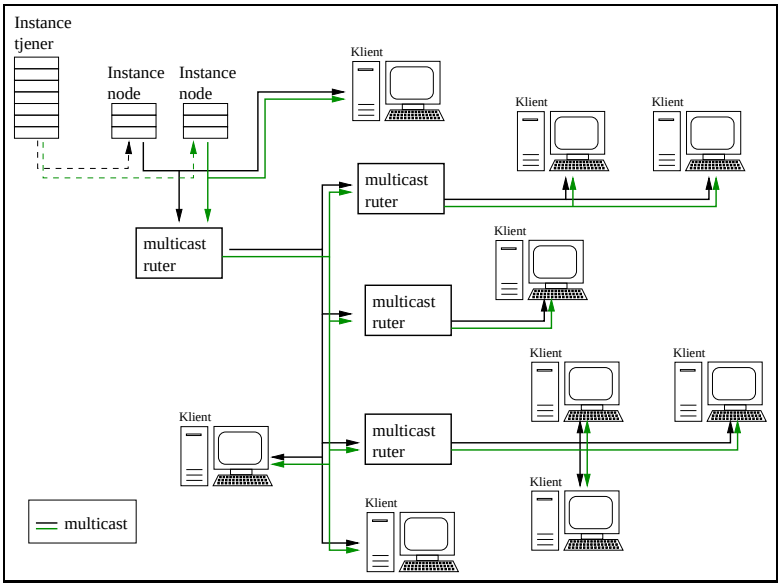
Figur 7.8: Hver datastrøm håndteres av en INSTANCE-strøm og to INSTANCE-buffere

vil operere ut fra et prinsipp om at dataene sendes ut periodevis. Hvis vi buffrer dataene i klienten vil en periode på for eksempel et sekund kunne egne seg bra. En periode på et sekund gjør at leseoperasjonene kan samles opp i relativt store blokker mens vi samtidig for en rimelig jevn datastrøm ut av INSTANCE-noden.

7.3.9 Caching av data

Mange minnehåndteringssystemer legger mye krefter i å cache data. Med caching mener vi her å beholde data lenger i minne enn nødvendig, i tilfelle noen skulle trenge de samme dataene senere. Logikken er den at siden det å lese fra disk er en svært langsom prosess, er det riktig å bruke en del resurser på å søke å unngå unødvendige leseoperasjoner.

I INSTANCE-noden vet vi imidlertid hele tiden svært mye om hvem som kommer til å aksessere hvilke data. Nok en fordel med at noden er en dedikert node. Vi vet at sannsynligheten for at en datafil skal bli lest av mer en en prosess er svært liten. I normal operasjon skal dette ikke skje. Derfor kan vi se bort fra fordelene ved caching, og heller spare noen resurser ved å ikke engang vurdere det.



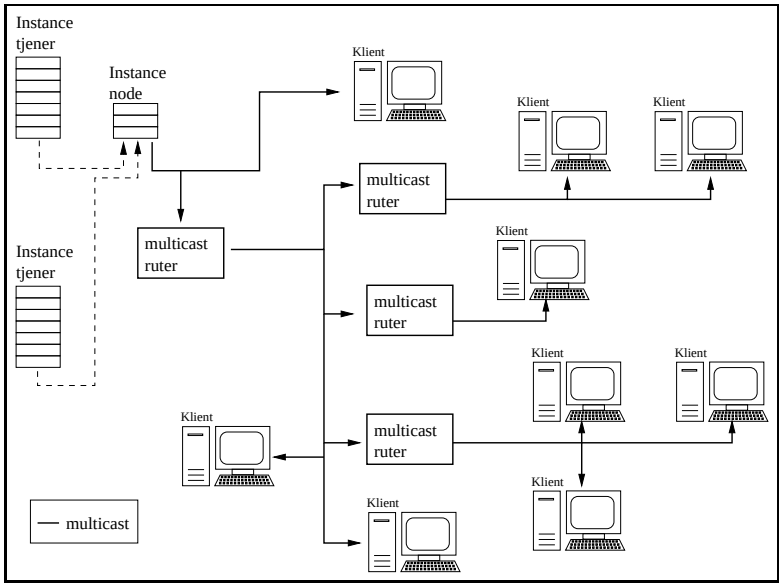
Figur 7.9: En INSTANCE-tjener kan fordele data på flere INSTANCE-noder.

7.4 Skalering av INSTANCE

Som vi så i seksjon 7.2 er belastningen på INSTANCE-noden konstant uavhengig av antallet klienter. Den eneste ekstra belastningen noden blir utsatt for er når INSTANCE-tjeneren oppdaterer noden. Denne belastningen kan imidlertid forutsees og eventuelt begrenses, siden hele poenget med INSTANCE-konseptet er at dataene ikke går i sanntid mellom tjener og klient. På denne måten oppnår vi at belastningen på noden er forutsigbar og vi kan lett fastslå maksimal belastning og dimensjonere noden etter dette.

Klientene har ingen direkte kontakt med noden. De bare mottar data distribuert i nettverket. Det spiller dermed ingen rolle hvor understrømmene kommer fra, så lenge nettverket distribuerer dem. Det betyr at en Klient samtidig eller over tid kan motta data fra flere forskjellige noder. Understrømmene i en og samme datastrøm kan dermed gjerne fordeles over flere noder uten at dette påvirker klientene. Figur 7.9 illustrerer dette. På samme måte kan flere tjenere benytte en og samme INSTANCE-node til å distribuere sine data.

Denne muligheten for å øke antallet INSTANCE-noder betyr at INSTANCE -konseptet er fullstendig skalerbart. Vi har vist at både antallet noder og tjenerne kan økes uten at det påvirker klientene. Vi antar også at nettverket lar seg skalere på en god måte. Vi vil derfor hevde at INSTANCE-konseptet er fullstendig skalerbart.



Figur 7.10: En INSTANCE-node kan distribuere data fra flere INSTANCE-tjenere.

7.5 Sammenligning med andre løsninger

Vi ønsker nå å sammenligne våre konsepter med de konseptene vi tok for oss i kapittel 6. Vi ønsker å se på likheter og forskjeller mellom vår bufferhåndteringsmekanisme og de mekanismene som er brukt i de andre prosjektene i kapittel 6.

Etter vår mening er den viktigste grunnen til at vårt konsept skiller seg fra de andre konseptene at vårt konsept er tilpasset INSTANCE-noden og de forutsetninger dette gir. Vi har ønsket å lage en optimalisert løsning basert på de gitte forutsetningene.

Vi har laget en buffermekansime som er spesialisert, ikke generell. Buffermekanismen er laget med det ene formål å effektivt håndtere varige kontinuerlige datastrømmer. Dette har påvirket hva denne bufferhåndteringen er, og hva den ikke er. Den er ikke ment å gi et grensesnitt lik de ordinære eksisterende mekanismer i NetBSD. Den gir heller ikke copy semantikk slik CoW gjør. Den er laget for å betjene datastrømmer som mellomlagres i INSTANCE-noden, slik at egenskaper ved strømmene er kjent på forhånd. Dersom en datastrøm plutselig skulle trenge buffere for å håndtere en datarate som er høyere enn bufferkapasiteten allokert i forhold til den på forhånd målte maksimale dataraten for strømmen vil dette konseptet håndtere dette dårlig. Eneste måten å øke bufferkapasiteten på er ved at INSTANCE-bufferne må frigjøres og nye større buffere allokeres. Andre konsepter håndterer slike endringer bedre, men det har ikke vært hensikten her.

Minnehåndteringen vår er basert på at filsystemet og nettverkssystemet deler et

minneområde. Vi skiller oss dermed fra CoW løsninger som IO-Lite[33] og Genie[9][7][8][6] ved at vi ikke beholder en copy semantikk. Dersom nettverkssystemet endrer på dataene vil de også endres for filsystemet. Her kommer vi inn på spesialiseringen igjen. Vi *vet* at filsystemet ikke skal lese dataene det har skrevet til bufferet. Vi behøver dermed ikke beskytte disse dataene for endring i tilfelle disksystemet skulle gi disse dataene også til en annen prosess.

Vi bygger bufferhåndteringen dels på konseptet I/O-Pathway fra Container Shipping[34]. Igjen fordi vi baserer oss på en delt minne mellom disksystemet og nettverkssystemet skiller vi oss fra Container Shippings move semantikk.

Det er store likheter mellom vårt konsept og konseptet mmbuf[10]. Vår bufferhåndtering skiller seg mest fra mmbuf ved at vi ikke har laget en like fleksibel mekanisme som mmbuf. Ved at mmbuf-er allokeres ved behov vil minneforbruket ved bruk av mmbuf-er bli mindre enn vårt siden vi allokerer minne for hver datastrøm basert på det maksimale behovet datastrømmen kan ha. Som nevnt er vi avhengige av at dette maksimale behovet ikke overskrides.

En Fordel ved vårt konsept er at vi bruker langt mindre ressurser på å allokere og deallokere buffere enn i mmbuf-konseptet. I vårt konsept allokerer og deallokerer vi buffere bare ved start og stopp av en datastrøm, mens mmbuf-er allokeres ved hver leseoperasjon. Det foregår arbeid i INSTANCE-gruppen med å implementer INSTANCE-buffere og mmbuf på samme maskin. Det vil være interessant å se resultatene av en sammenligning av disse to konseptene.

Kapittel 8

Implementasjon

I dette kapittelet går vi inn på hvordan vi i detalj har valgt å realisere konseptene i kapittel 7. Selve kildekoden finnes i tillegg B på side 123. Kildekoden er passet inn i NETBSD kjernen. Kildekoden til denne er hentet fra NETBSD prosjektets hjemmeside[31]. Fordi det er vårt ønske at implementasjonen skal kunne brukes videre har vi valgt å skrive kildekoden på engelsk. Funksjonsnavn, navn på datastrukturer og deres elementer samt kommentarene i koden er på engelsk. Vi håper dette ikke skaper problemer for leseren. Navn på konsepter i denne oppgaven er norske. For å forenkle forståelsen er konsepter skrevet med vanlig font, mens ord tatt fra kildekoden som systemkall og datastrukturer er skrevet med `skrivemaskin font`.

Vi går først i avsnitt 8.1 inn på premissene for implementasjonen. Hvilke valg vi har tatt og hvordan de har påvirket hvordan vi har implementert bufferhåndteringen i INSTANCE-noden. Vi ser videre i avsnitt 8.2 på hvordan datastrukturene som implementerer konseptene INSTANCE-strøm og INSTANCE-buffer fra avsnitt 7.3.7 ser ut. Videre i avsnitt 8.3 ser vi på hvilke systemkall som vi gjør tilgjengelig for applikasjonene på en INSTANCE-node. Avsnitt 8.4tar for seg detaljer i håndteringen av `buf` og `mbuf` strukturene. Etter dette, i avsnitt 8.5.2, beskriver vi *tilstandene* et INSTANCE-buffer kan være i. Vi ser også på hvordan INSTANCE-strømmen håndterer fordelingen av lese og sende kall til riktig INSTANCE-buffer.s Til slutt ser vi i avsnitt 8.6 i detalj på hva som skjer i INSTANCE-strømmen når dataene flyter gjennom systemet.

8.1 Underliggende valg

Vi går først kort inn på hvilke valg definerer rammene for implementasjonen. Hvorfor vi har gjort disse valgene og hvilke rammer det gir for implementasjonen.

8.1.1 Maskinvare

Vi ønsker om mulig å basere oss på standard hyllevare datamaskiner. Vi har derfor tatt utgangspunkt i den maskinvaren vi har hatt tilgjengelig. For utviklingen i denne oppgaven har vi hatt en Intel Pentium II baser PC med UltraWide SCSI kort og Ultra 2 Wide SCSI disk. Vi har videre brukt et Intel Pro 100+ nettverkskort. Om ikke veldig billig er dett tross alt relativt rimelige komponenter.

8.1.2 Operativsystem

Vi har valgt å basere implementasjonen vår på NETBSD versjon 1.4.1. NETBSD er et fritt tilgjengelig operativsystem basert på 4.4BSD UNIX. NETBSD er under kontinuerlig utvikling og har et aktivt miljø som driver med dette.

Selve implementasjonen er gjort ved å legge til fire systemkall i kjernen til `NetBSD`, samt endre minimalt på et femte. Noen sentrale variabler har også blitt endret.

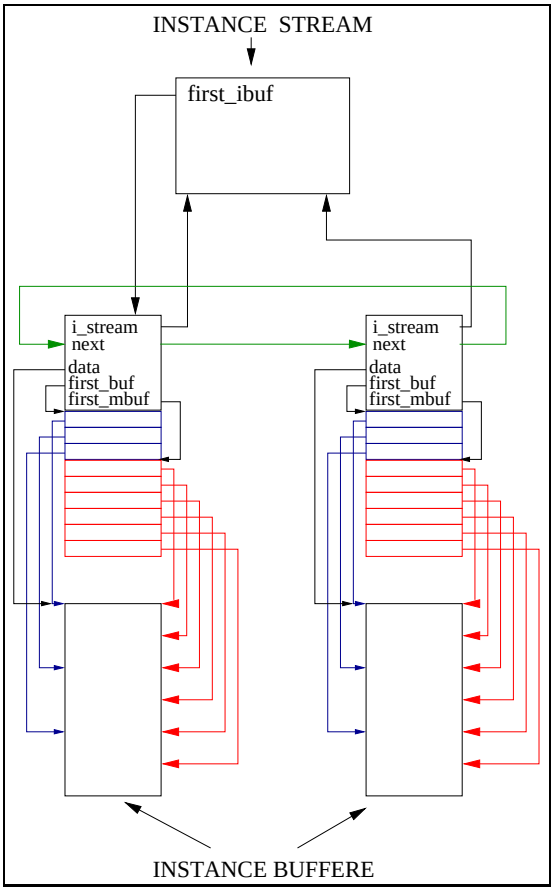
For å kunne bruke ferdige sett av header- og ekstern-`mbuf`er har vi måttet endre noe på kildekoden får håndtering av `mbuf`ene. Koden for håndtering `mbuf` strukturer er basert på å allokere ved behov og deallokere etterpå. Koden måtte derfor modifiseres noe for at den ikke skulle slette mbuf strukturene i INSTANCE-bufferne. Dette har vi gjort ved å innføre et nytt flagg i `mbuf` strukturene og endre noe på oppførselen i funksjonen `udp_output` og `m_freem`. Funksjonen `udp_output` tar seg av å sende udp pakker ut på nett, mens `m_freem` frigjør `mbuf`-ene etter bruk. Koden til `udp_output` finnes i avsnitt B.3.4 på side 161. `m_freem` finnes i avsnitt B.3.3 på side 161.

8.2 Datastrukturer

For å håndtere datastrømmene i INSTANCE-noden har vi laget to datastrukturer, `instance_stream` og `instance_buffer`. Til hver datastrøm hører en instance_stream og instance_buffer. Datastrukturen instance_stream inneholder informasjon om datastrømmen som er felles for de to bufferne. Instance_buffer inneholder data nødvendig for intern håndtering av dataflyten gjennom bufferet.

8.2.1 INSTANCE-stream

Figur 8.1 viser forenklet hvordan en INSTANCE-stream ser ut. Kildekoden til INSTANCE-stream finnes i tillegg B.2.7. Vi ser at en instance stream har to



Figur 8.1: Illustrasjon av en INSTANCE-stream og dens forhold til INSTANCE-bufferne

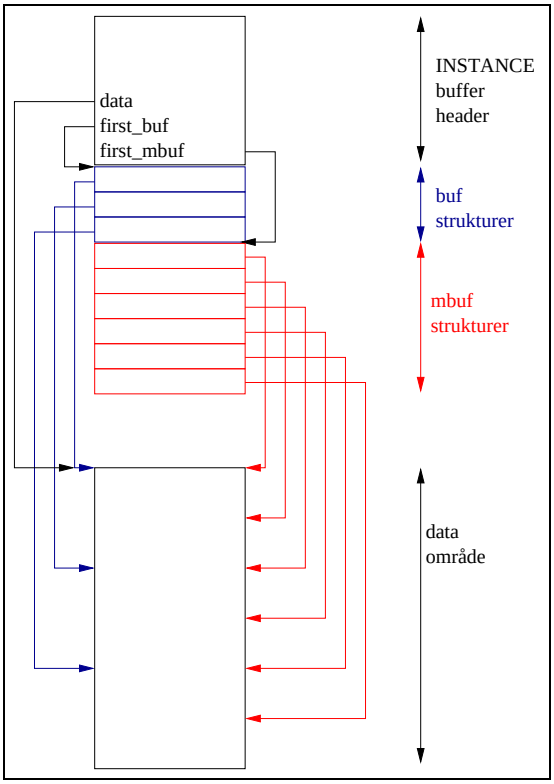
`first_ibuf` pekere til det første INSTANCE-bufferet. Begge de to INSTANCE-bufferne har en `next` peker som peker på det andre bufferet. Bufferne har også en `i_stream` peker til "eieren" av bufferet, med andre ord den tilhørende INSTANCE-stream.

Figur 8.2 viser koden til `instance_stream`. Vi har en peker `p` til prosessen som opprettet strømmen slik at vi kan sjekke rettigheter ved kall. Vi har størrelsen på de selve dataområdene i bufferet, `buf_size`. Vi har tre peker til instance buffere, `first_ibuf`, `read_ibuf` og `send_ibuf`. `first_ibuf` er en fast pekere til det første bufferet. `read_ibuf` og `send_ibuf` peker til enhver tid på det bufferet det skal leses eller skrives til ved *neste* lese eller skriveoperasjon. Strømmen har også pekere til filen som buffres og nettverks socket-en som dataene sendes ut på. Til filen har vi en vnode peker (`vnode`) og en fil peker (`file_fp`). Vi har også en peker til filsystemet filen ligger på, `file_system`. Ved lesing trenger vi også vite antallet `buf` strukturer, `num_buf`. For å håndtere lesing med variabel bitrate har vi

```
struct instance_stream {
  struct proc *p;
  int buf_size;
  off_t end_of_file;
  struct instance_bufer *first_ibuf;
  struct instance_buffer read_ibuf;
  struct instance_buffer send_ibuf;
  struct file *file_fp;
  struct vnode *vnode;
  int file_fdesc;
  struct fs *file_system;
  int num_buf;
  off_t prev_read_arg_offset;
  size_t buf_data_size;
  struct sockaddr_in *addr;
  struct socket *socket;
  struct file *socket_fp;
  int socket_fdesc;
  int num_mbuf;
  off_t prev_send_arg_offset;
  int mbuf_data_size;
  int last_mbuf_size;
};
```

instance_syscalls.h

Figur 8.2: `struct instance_stream`



Figur 8.3: Illustrasjon av et INSTANCE-buffer

`prev_read_arg_offset` som inneholder offset argumentet ved forrige kall på `instance_read`.

8.2.2 INSTANCE-buffer

Figur 8.3 Illustrerer oppbygningen av et INSTANCE-buffer. Kildekoden til INSTANCE-buffer finnes i tillegg B.2.6. Vi ser hvordan et INSTANCE-buffer består av en header, et antall `buf` strukturer og et antall `mbuf` strukturer. I tillegg kommer et dataområde som inneholder selve dataene i bufferet. Både `buf` strukturene og `mbuf` strukturene inneholder pekere inn i dataområdet.

Instance bufferet inneholder en både `buf` og `mbuf` strukturer som peker inn i data-område. For `buf` strukturer er dette relativt rett frem. mbuf-ene er består av `mbuf` par, en header `mbuf` og en ekstern `mbuf` som peker inn i dataområdet. De to `mbuf`-ene i paret er lenket sammen i en kjede. For å håndtere `buf`-ene har vi en peker til første `buf`, `first_buf`, og et antall, `num_buf`, i tilhørende INSTANCE-strøm. Vi leser alltid alle `buf` strukturene “samtidig”, og etter det er bufferet klar til lesing.

```
struct instance_buffer {
  struct instance_stream *i_stream;
  struct instance_buffer *next;
  void *data;
  short state;
  short error;
  int io_pending;
  int io_being_called;
  struct buf *first_buf;
  struct mbuf *first_mbuf;
  struct mbuf *curr_mbuf;
  struct mbuf *last_mbuf;
  off_t start_of_data_area;
};
```

instance_syscalls.h

Figur 8.4: `struct instance_buffer`

Vi har tre pekere for å håndtere `mbuf` strukturene. En `first_mbuf` som peker til første mbuf, en `curr_mbuf` som peker på den neste header-`mbuf`-en som skal sendes. `last_mbuf` peker på den siste header-`mbuf`-en. `curr_mbuf` traverserer nedover gjennom header-`mbuf`-ene ettersom sendingen foregår. Når siste `mbuf`-par er sendt er INSTANCE-bufferet klart for sending.

Vi beskriver håndteringen av `buf` og `mbuf` mer i detalj i avsnittene 8.3.3 og 8.3.4.

8.3 Systemkall

Vi ser nå på de systemkallene INSTANCE-systemet tilbyr brukerprosessene. Dette er `instance_alloc`, `instance_free`, `instance_read` og `instance_send`. `instance_alloc` og `instance_free` kalles bare en gang hver i forbindelse med opprettelsen og avslutningen av en INSTANCE-strøm. `instance_read` og `instance_send` kalles en gang hver tidsperiode, normalt hvert sekund. `instance_read` og `instance_send` har som hoved parameter offset i filen som distribueres og kallene garanterer at data frem til minimum offsetet blir lest eller sendt.

`instance_read` og `instance_send` operer som par forskjøvet i tid. Ved opprettelsen av en strøm må først et `instance_read` kall sendes slik at data er klar til lesing. Neste tidsperiode sendes først et kall på `instance_send`, slik at data fra forrige kall blir sendt ut, deretter et nytt kall på `instance_read` for å lese

inn data før neste kall. Normalt returnerer `instance_read` og `instance_send` med en gang, mens lesing og sending foregår i bakgrunnen. I enkelte situasjoner må allikevel kallene vente på hverandre. Mer om dette i avsnitt 8.6 som beskriver dataflyten gjennom noden i større detalj.

8.3.1 instance_alloc

Kallet `instance_alloc` allokerer alle de nødvendige strukturene og dataområdene som er nødvendig for å betjene er INSTANCE-strøm. Det allokeres en `instance_stream`, to `instance_buffer` med hvert sitt dataområde. Kildekoden til `instance_alloc` finnes i tillegg B.1.1.

Ved kall på `instance_alloc` må brukerprosessen allerede ha bestemt seg for hvilken INSTANCE-strøm som skal betjenes. Dette fordi hver `instance_stream` tilpasses behovet til en strøm. Vi må vite:

- Maksimal bitrate
- Hvilken fil vi skal buffre
- Hvor dataene skal sendes

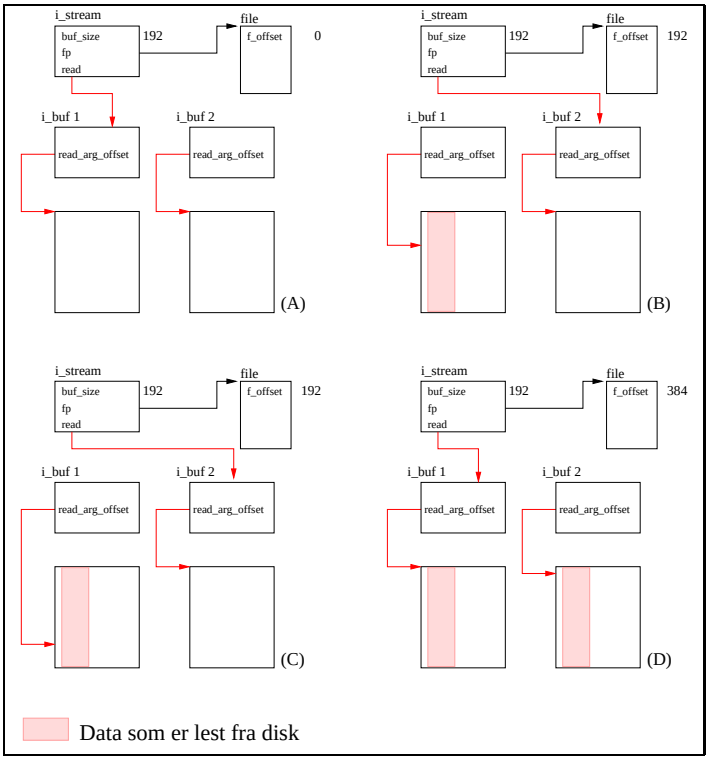
8.3.2 instance_free

Kallet `instance_free` frigjør alle strukturer og dataområder tilordnet en INSTANCE-strøm. Kildekoden til dette kallet finnes i tillegg B.1.2 Kallet tar en peker til den `instance_stream` som skal frigjøres. Kallet vil normalt vente på at allerede igangsatte lese og sendeoperasjoner blir ferdige før minnet frigjøres.

8.3.3 instance_read

Kallet `instance_read` sendes en gang for hver tidsperiode. Kallet tar et offset inn i filen som parameter i tillegg til en peker til den `instance_stream` det skal leses fra. Kallet garanterer at det minimum er lest frem til offset når det returnerer. Kallet leverer alle `buf` strukturene i INSTANCE-bufferet videre til disksystemet og overlater den fysiske lesingen til dette. Disksystemet står fritt til å optimalisere rekkefølgen på den fysiske lesingen fritt for å optimalisere lesingen. Kildekoden til dette kallet finnes i tillegg B.1.3.

`instance_read`'s leseoperasjoner er alltid på hele INSTANCE-buffere. Figur 8.5 illustrerer dette. Figur 8.5(A) viser situasjonen rett etter at bufferne er allokert.



Figur 8.5: Illustrasjon av kallet `instance_read`

Figur 8.5(B) viser hva som skjer når et read kall på 64KB er sendt. Som vi ser fører dette til at et helt INSTANCE-buffer leses fra disk. I dette tilfellet er det på 192KB. I figur 8.5(C) er et nytt kall med offset 144KB sendt. Som vi ser er disse dataene allerede lest. Kallet returnerer dermed suksessfullt uten at nye data er lest. Til sist i figur 8.5(D) er `instance_read` kalt med offset på 208KB. Dette er utenfor området som allerede er lest, så nye 192KB leses inn. Hele tiden peker `read_buffer` på det bufferet det skal leses til neste gang det *faktisk* leses fra disk.

8.3.4 instance_send

Et kall på `instance_send` må alltid følge etter et tilsvarende kall på `instance_read`. Nødvendigvis må de som skal sendes leses først. Som read kallet tar `instance_send` en peker til en `instance_stream` og et offset inn i filen. Mens read kallet leser et helt buffer om gangen ville tilsvarende løsning for dette kallet føre til en veldig variable bitstrøm, spesielt ved lave rater. I stedet sender vi så mange `mbuf` strukturer som nødvendig. Det betyr likevel at dersom det datarate-ne er lavere enn størrelsen på `mbuf` strukturene vil kun hele antall `mbuf` strukturer sendes.

For å håndtere hva som skal sendes har hvert `instance_buffer` tre `mbuf` pekere, `first_mbuf`, `last_mbuf` og `curr_mbuf`. `first_mbuf` og `last_mbuf` peker på henholdsvis første og siste `mbuf`. Pekeren `curr_mbuf` i INSTANCE-bufferet brukes til å holde orden på hvilke `mbuf` som skal sendes. Den peker alltid på det neste `mbuf` som skal sendes.

Etter at vi har beregnet hvilke `mbuf` strukturer som skal sendes må kall sendes videre ned i systemet for å sette i gang den reelle dataoverføringen til nettverkskortet.

8.4 Håndtering av `buf` og `mbuf` strukturer

Vi tar nå for oss enkelte viktige detaljer i implementasjonen. Vi ser på hvordan vi håndterer `buf` og `mbuf` strukturene i INSTANCE-bufferne. Spesielt hva vi gjør med disse slik at de ikke skal bli frigjort av disksystemet eller nettsystemet slik det normalt skjer.

8.4.1 Håndtering av buf strukturer

Som vi har beskrevet ser INSTANCE-bufferne våre ut som `buf` strukturer for disksystemet. Disse `buf`-ene blir håndtert som vanlig av disksystemet.

Som vi beskrev i avsnitt 6.1.1 blir `buf` strukturene normalt frigjort etter bruk. Dette skjer ved at en bestemt funksjon pekt til i strukturen, `b_iodone`. Normalt er denne pekeren satt til **NULL**. Da blir en standardfunksjon kalt for å frigjøre bufferet. Dersom denne pekeren peker på noe, blir den funksjonen kalt i stedet. Denne setter vi til å peke på vår egen versjon av I/O-Done (mer om I/O-Done i avsnitt 8.4.3). I tillegg har vi endret den orginale `buf` strukturen til å inneholde en `void *` peker. Denne brukes som argument av vår egen I/O-Done funksjon[1]. Kildekoden med den endrede `buf` strukturen finnes i tillegg B.3.2 Vi skriver mer om I/O-Done i avsnitt 8.4.3.

8.4.2 Håndtering av mbuf strukturer

Håndteringen av `mbuf` strukturer ligner mye på håndteringen av `buf` strukturer. Vi innfører også her vår egen I/O done funksjon. I tillegg har vi lagt til et ekstra flagg i `mbuf` strukturen, `M_NODELETE` (se tillegg B.3.1). Vi setter dette flagget i "våre" `mbuf` strukturer. Vi har endret på kildekoden som frigjør `mbuf` strukturene, slik at strukturer med `M_NODELETE` flagget satt ikke slettes eller puttes på en liste over ledige `mbuf`-er (se tillegg B.3.3).

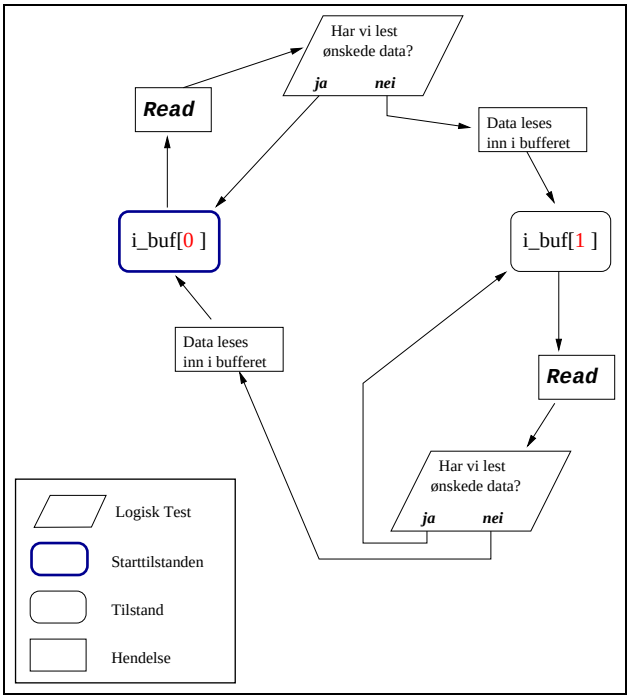
8.4.3 I/O-Done

I et INSTANCE-buffer er både `buf` og `mbuf` strukturene fast allokert. Dette i motsetning til vanlig NETBSD der disse frigjøres etter bruk. Som beskrevet i avsnittene 8.4.1 og 8.4.2 har vi byttet ut I/O-Done funksjonene i både `buf` og `mbuf` strukturene med vår egen INSTANCE I/O-Done. Denne funksjonen frigjør ikke bufferet. I stedet oppdateres telleren `instance_io_pending` i INSTANCE-bufferet. Denne telleren økes med én for hver lese eller sendeoperasjon som settes i gang. Når så I/O-Done kalles reduseres telleren med en. Når telleren når null endres status for bufferet. Buffertilstanden er beskrevet i avsnitt 8.5.2. Kildekoden til I/O-done finnes i tilleggene B.1.6 og B.1.7.

8.5 Håndtering av bufferpekere og buffertilstander

Vi går nå inn på detaljer i håndteringen av INSTANCE-bufferne. I avsnitt 8.5.1 ser vi på hvordan vi håndterer pekerne `read_ibuf` og `send_ibuf` i INSTANCE-strømmen. Pekerne som forteller hvilket av de to bufferne det til enhver tid skal leses til eller skrives fra. For å sikre oss mot at det for eksempel ikke sendes fra et buffer der lesingen ikke er avsluttet eller at det skrives til et buffer mens sending

[1] Dette ekstra argumentet tilsvarer `mbuf` strukturens `void *ext_arg`.



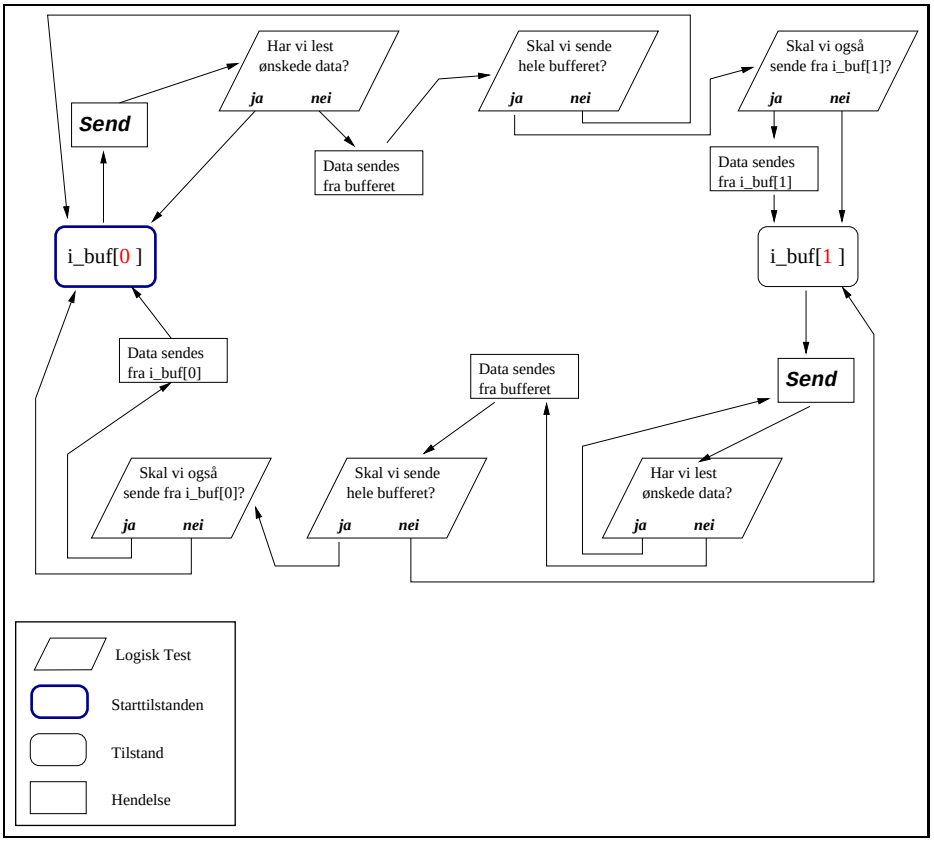
Figur 8.6: Håndteringen av pekeren `read_ibuf`

pågår har hvert INSTANCE-buffer et *tilstand.* Hvilke tilstander vi har og hvordan disse blir håndtert tar vi for oss i avsnitt 8.5.2.

8.5.1 `read_ibuf` og `send_ibuf`

Som beskrevet i avsnitt 8.2.1 har hver INSTANCE-strøm to pekere, `read_ibuf` og `send_ibuf` som holder rede på hvilket buffer som skal håndtere systemkallene `instance_read` og `instance_send`. Pekeren koderead_ibuf peker til enhver tid på det bufferet som skal håndtere neste *fysiske* lesing. Vi bruker ordet *fysisk lesing* fordi vi av optimaliseringsgrunner ofte leser mer enn det applikasjonen ber om, siden vi alltid leser et helt INSTANCE-buffer om gangen (se avsnitt 8.3.3). På samme måte peker `instance_send` på det bufferet som neste fysiske sendeoperasjon skal foregå fra. Pekerne peker ofte på hvert sitt INSTANCE-buffer, men ikke alltid. Vi kommer mer inn på dette i avsnitt 8.6 som beskriver dataflyten gjennom INSTANCE-strømmen og dermed også håndteringen av `read_ibuf` og `send_ibuf` pekerne.

Figur 8.6 viser et tilstandsdiagram for pekeren `read_ibuf`. Tilstandene beskriver hvilket INSTANCE-buffer pekeren peker på. Hendelsene “read” representerer et kall fra applikasjonen på systemkallet `instance_read`. Ved hvert kall vurderes det



Figur 8.7: Håndteringen av pekeren `send_ibuf`

om kallet ber om data som ikke er lest. Dersom det er tilfellet leses data inn i bufferet pekt på av `read_ibuf` og `read_ibuf` settes så til å peke på det neste bufferet.

Håndtering av `send_ibuf` er noe mer komplisert enn håndteringen av `read_ibuf`. Figur 8.7 viser dette. I motsetning til lesekall som enten fører til ingen fysisk lesing, eller til lesing av et helt INSTANCEbuffer, sendes data i hele multiplum av `mbuf` strukturer (se avsnitt 8.3.4). Dermed kan vi sende deler av dataene, men ikke alle. Det betyr at vi etter et kall på send og sending av deler av bufferets data fremdeles skal sende fra samme buffer ved neste send. Etter slike `instance_send` kall fortsetter vi dermed tilbake til å peke på samme buffer og tilstanden endres ikke.

Det at deler av et buffer kan sendes ved et sende-kall gir grunnlag for en spesialsituasjon. Enkelte ganger resulterer et kall på `instance_send` i at deler at et buffer sendes og også deler av det neste. Dette skjer ved at det gjøres et rekursivt internt i systemkallet for å sende også fra neste buffer. Så settes pekeren til å peke på neste

INSTANCE-buffer før systemkallet returnerer.

Som beskrevet i avsnitt 8.3.4 sender vi i multiplum av hele `mbuf`-er. Hvis et sendekall på for eksempel bare 100 byte blir mottat, og disse dataene allerede er sendt fordi det forrige sendkallet ble rundet opp til nærmeste antall hele `mbuf`-er, vil kallet bare returnere uten å sende noen data.

8.5.2 Buffertilstand

Hver av INSTANCE-bufferne befinner seg til enhver tid i en *tilstand.* Denne tilstanden settes i variabelen `state`. Tilstandene er:

`INSTANCE_READ_OK` Klart for lesing.

`INSTANCE_SEND_OK` Klart for sending.

`INSTANCE_READING` Det leses til bufferet.

`INSTANCE_SENDING` Det sendes fra bufferet

`INSTANCE_READ_WAITING` Det sendes fra bufferet, samtidig ligger det et lese kall og venter på at bufferet er klart for sending.

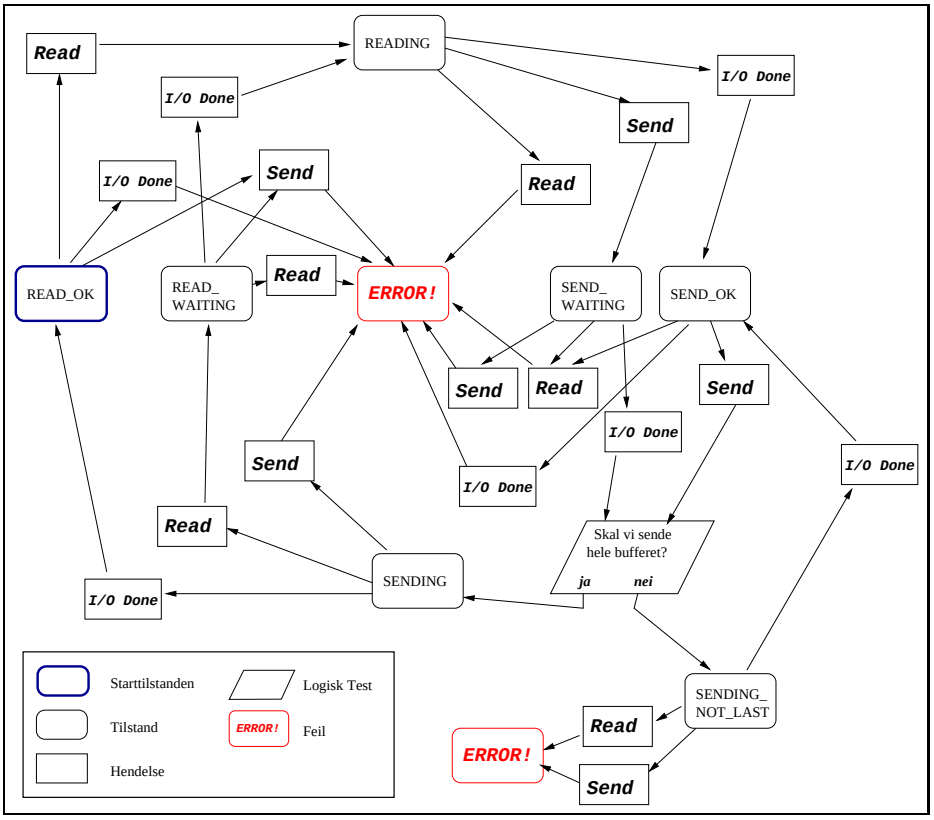
`INSTANCE_SEND_WAITING` Det leses og kall venter på å kunne sende.

`INSTANCE_SENDING_NOT_LAST` Det sendes, men det vil fremdeles være data igjen etter sending som ikke er sendt.

Tilstanden beskriver bufferets status. Tilstandene `INSTANCE_READ_OK` og `INSTANCE_SEND_OK` er ventetilstander. Bufferet er enten tomt [2] og klart for lesing eller at bufferet nettopp er fylt etter en leseoperasjon og klar for sending. `INSTANCE_READING` og `INSTANCE_SENDING` betyr at lese eller skriveoperasjoner er satt i gang, men ikke er ferdig. Dersom `instance_read` kalles mens bufferet er i tilstanden `INSTANCE_SENDING`, settes tilstanden til `INSTANCE_READ_WAITING` for å indikere at en leseoperasjon venter på å få begynne. I denne tilstanden vil I/O-Done funksjonen sette i gang leseoperasjonen når sendeoperasjonen er ferdig. `INSTANCE_SEND_WAITING` vil normalt ikke inntreffe, da det er en forutsetning at det som sendes er lest forrige tidsperiode. Denne tilstanden vil dermed bare inntreffe dersom det oppstår større forsinkelser i noden.

I motsetning til en leseoperasjon som alltid fyller hele bufferet, kan en sendeoperasjon foregå på bare deler av bufferet. Dermed trenger vi tilstanden

[2] Med tomt mener vi at vi ikke er interessert i innholdet. Det kan bety at vi har sendt hele innholdet ut på nettet eller at bufferet nettopp er allokert.



Figur 8.8: Tilstandsdiagram for et INSTANCE-buffer

`INSTANCE_SENDING_NOT_LAST` når vi sender, men neste operasjon på bufferet er en ny send, og ikke en read. Etter `INSTANCE_SENDING_NOT_LAST` vil I/O-done endre tilstanden tilbake til `INSTANCE_SEND_OK`.

Figur 8.8 viser et tilstandsdiagram for INSTANCE-bufferet mens bufferet er i bruk. Tilstanden er til enhver tid lagret i bufferets `state` variabel. Før allokering eksisterer ikke bufferet og under allokering er tilstanden udefinert. Under frigjøringen kan bufferet ha en tilstand helt til siste data er sendt ut med mindre applikasjonen spesifiserer at INSTANCEbufferet skal frigjøres uten hensyn til tilstand. Figuren viser ikke spesialtilfellene allokering eller frigjøring, men vanlig bruk med kall på `instance_read` og `instance_send`. Figuren viser i tillegg til tilstanden hendelser som gir feil i bufferet. Eksempler på slike feil er dersom applikasjonen forsøker å sende data fra et buffer før det er lest data inn i bufferet eller dersom applikasjonen forsøker å lese nye data inn i et buffer før de gamle er sendt ut. Slike feilsituasjoner fører til at bufferet blir korrupt. Feilstatus settes da i variabelen `error`.

Figuren viser de *tilstander* INSTANCE-bufferet kan befinne seg i. Pilene viser hvordan bufferet kan bevege seg fra en tilstand til en annen dersom det oppstår *hendel-*

ser. Hendelser kan være systemkall fra applikasjonen som `instance_read` eller `instance_send`. Når disk- eller nettverks-systemet er ferdige med å håndtere data kalles `INSTANCE_io_done`. Når `INSTANCE_io_done` funksjonen finner at alle kalte I/O operasjoner er ferdige, endres tilstanden. Hendelsen io_done i figur 8.8 referer altså til den "endelige" io_done da all kalt I/O er ferdig.

Starttilstanden i et INSTANCE-buffer er `READ_OK`. I denne tilstanden er det bare et kall til `instance_read` som er korrekt. Alle andre kall eller hendelser fører til feil.

8.6 Datastrømmen gjennom INSTANCE-noden

I dette avsnittet skal vi gå i detalj på hva som skjer i INSTANCE-strømmen og INSTANCE-bufferne når data strømmer gjennom noden. Vi beskriver her implementasjonen av konseptene i avsnitt 7.3.2 og går dypere inn i håndteringen av sende og leseoperasjoner beskrevet i figur 7.7 (side 76).

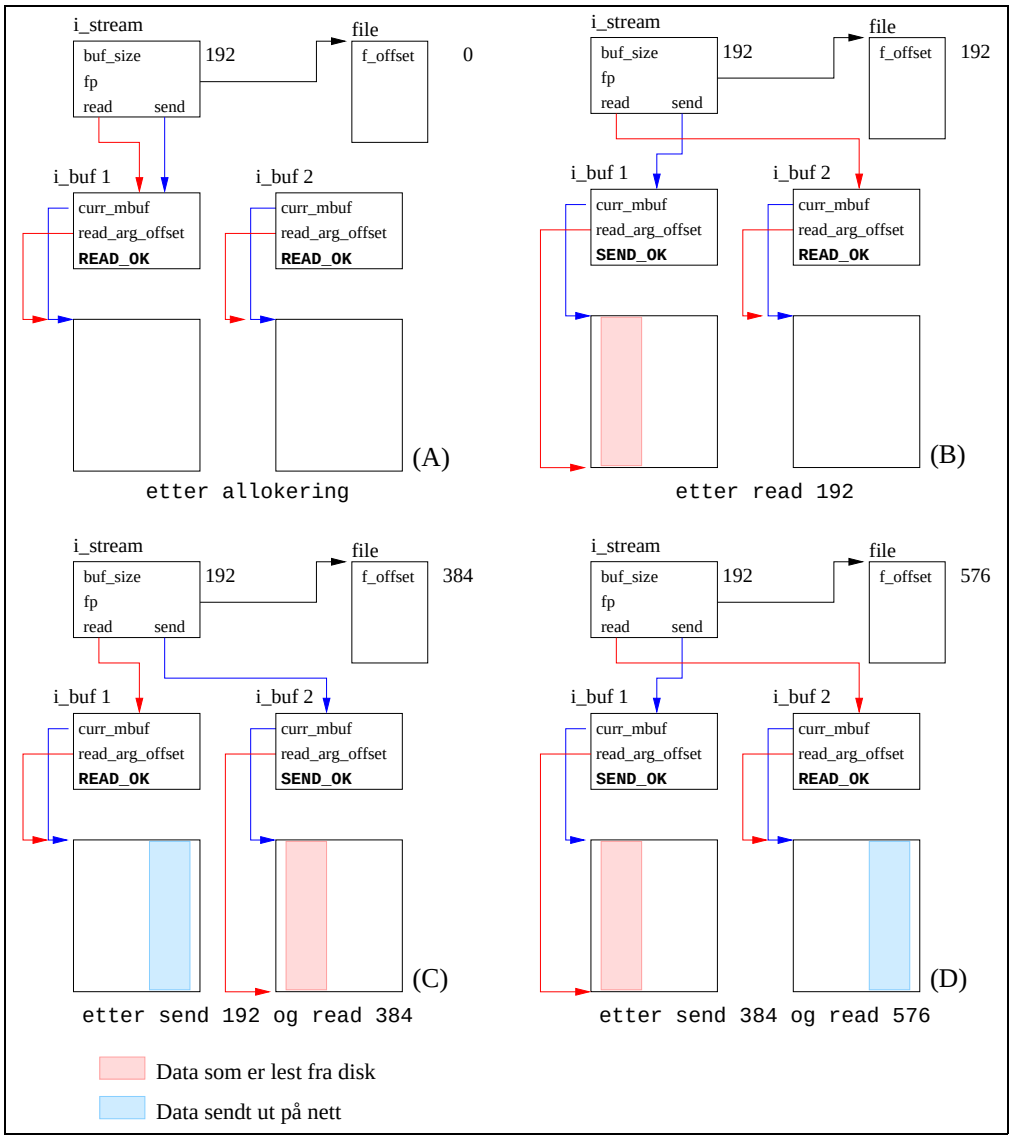
Konseptuelt forgår lese og skriveoperasjonene samtidig. Dette er mulig selv om kallene nødvendigvis må skje i en rekkefølge. Årsaken er at lese og skrivekallene setter i gang lesing og skriving i underliggende systemer og returnerer før de fysiske operasjonene er ferdige. Normalt foregår også lesing og skriving parallelt. Imidlertid finnes det tilfeller der lese kallet på vente på at sending av deler av bufferet er ferdig. Dette er noe vi ser nøyere på i avsnitt 8.6.2.

Først ser vi på et eksempel med konstant bitrate. Vi ser på hvordan situasjonen er med variabel bitrate. Ikke overraskende er det noe vanskeligere å håndtere dette.

8.6.1 Konstant bitrate

Figur 8.9 illustrere hvordan pekere og verdier ser ut når data strømmer gjennom noden men en kontinuerlig hastighet. Hastigheten er også identisk med bufferstørrelsen. Dette er den enkleste situasjonen å håndtere. I første tidsperiode sendes et lese kall på med bufferstørrelsen som offset. Neste periode kommer et sendekall på med samme offset og et nytt lesekall for neste datamengde.

Figur 8.9 (A) viser situasjonen rett etter allokering. Etter første lesekall med offset 192 ser situasjonen ut som figur 8.9 (B). Nå er vi klare for å sende. Etter kall på send (offset 192) og read (384) kommer vi til figur 8.9 (C). Til sist etter nok et sett av kall på send (offset 384) og les (offset 576) får vi situasjonen vist i figur 8.9 (D). Figurene viser hele tiden status *etter* at kallet og I/O operasjonene de setter i gang er fullført. Vi ser hvordan `read_ibuf` og i `instance_stream` til enhver tid peker på det bufferet som *neste* gang skal leses. Det samme gjelder `send_ibuf` og



Figur 8.9: Dataflyt i INSTANCE-buffere, konstant bitrate

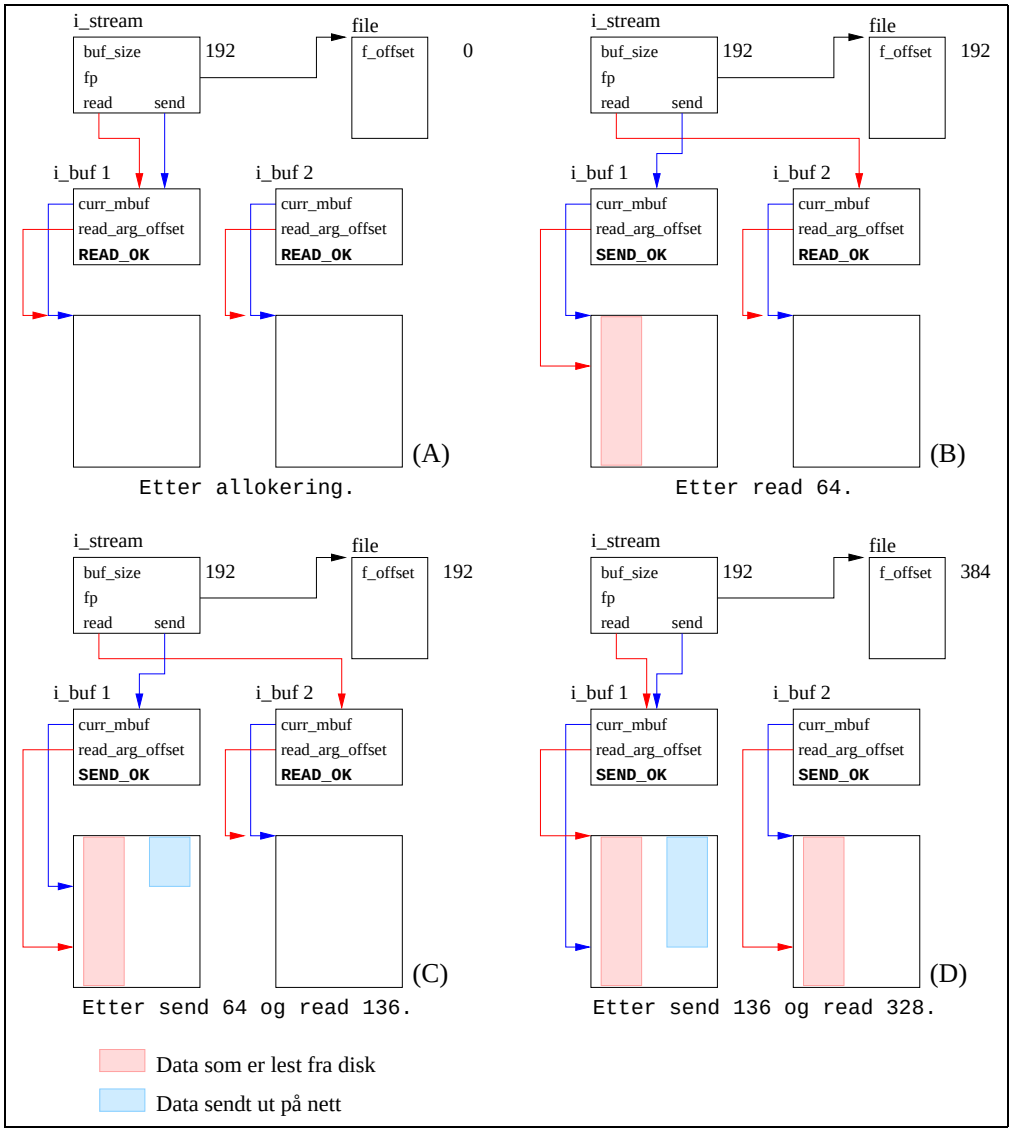
sending. Data om hvor langt inn i filen vi er kommet ligger i `file` strukturen som blir pekt på av INSTANCE-strømmen.

8.6.2 Variabel bitrate

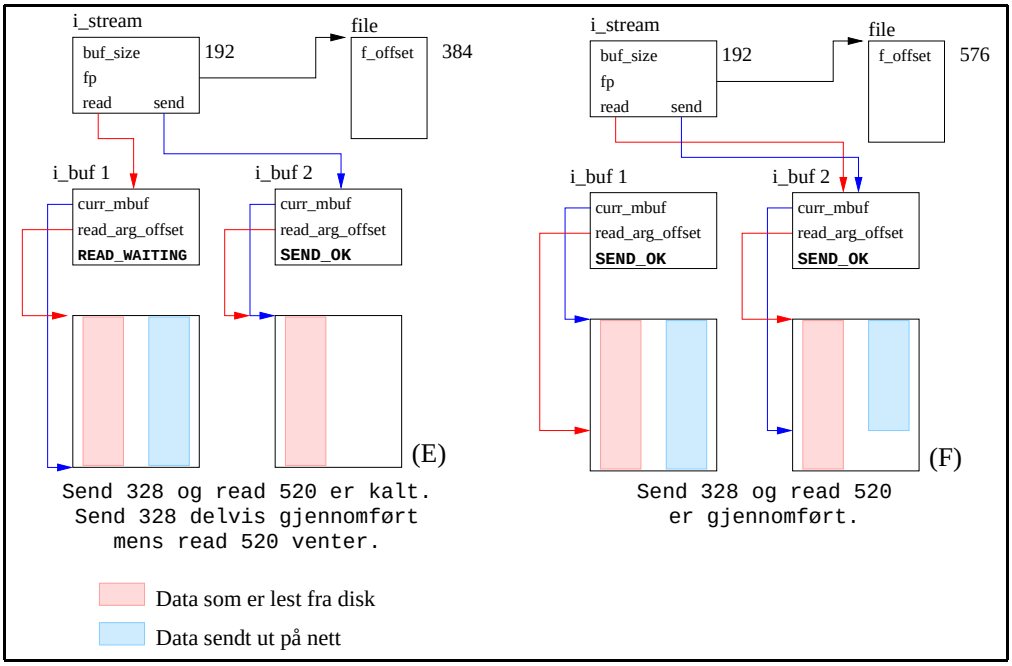
Nå ser vi på hvordan dataflyten tar seg ut med en variabel bitrate. Forutsetningen er fremdeles at vi alltid sender det vi leste forrige tidsperiode. Vi vil vise at variabel bitrate byr på flere problemer enn eksempelet i avsnitt 8.6.1. Det hender at et lesekall må vente på at et samtidig sendekall gjør seg ferdig.

Figurene 8.10 og 8.11 illustrerer dataflyten med variable bitrate. Etter allokeringen er situasjonen den samme som ved konstant datarate (figur 8.10 (A)). Figur 8.10 (B) viser situasjonen etter read (offset 64). Her leser vi det samme som med konstant datarate (figur 8.9 (B)), men offsetet `read_arg_offset` viser 64. Når vi neste gang får et sendekall på 64 og et lesekall på 136 sendes 64KB. Ved å sammenligne offsettet med offsettet i filen, ser vi at det som ønskes lest allerede er lest. Vi leser dermed ingenting, i stedet flytter vi leseoffsetet i INSTANCE-bufferet til 136 (figur 8.10 (C)). I neste tidsperiode kommer et sendekall på forrige periodes offset 136 samt et nytt lesekall med offset 328. Vi sender det nødvendige og leser et nytt buffer (figur 8.10 (D)).

Figur 8.11 (E) viser et problem. Sendekallet med offset 328 må sende data fra begge bufferne. Lesekallet må lese buffer nummer 1 for å nå sitt offset på 520. Siden vi alltid kaller `instance_send` før `instance_read`, vil sendingen settes i gang først. `instance_read` finner at bufferet er i tilstand `INSTANCE_SENDING`, endrer denne til `INSTANCE_READ_WAITING` og kaller `tsleep` som får leseprosessen til å "sove". Når så I/O-done finner at sendingen i buffer 1 er ferdig og at tilstanden er `INSTANCE_READ_WAITING`, vekker den leseprosessen slik at dette fortsetter. Figur 8.11 (E) viser situasjonen i det sendeprosessen er ferdig, men før I/O-done har satt leseprosessen i gang igjen. Figur 8.11 (F) viser situasjonen når både sendeprosessen og leseprosessen er avsluttet.



Figur 8.10: Dataflyt i INSTANCE-buffere, variable bitrate (1 av 2)



Figur 8.11: Dataflyt i INSTANCE-buffere, variable bitrate, (2 av 2)

Kapittel 9

Resultater

I dette kapittelet presenterer vi resultater fra målinger vi har utført på implementasjonen vår. I avsnitt 9.1 lister vi opp hvilken hardware som er benyttet ved målingene. Vi beskriver så målemetodene i avsnitt 9.2 før vi presenterer resultatene i avsnitt 9.3

9.1 Underliggende Hardware

Maskinen vi har brukt til målingene har hatt følgende spesifikasjoner:

Hovedkort ASUS P2B (Intel 440BX chipset)

CPU Pentium II 350MHz

SCSI kort Adaptec AHA-2940 UltraWide SCSI kort

Disk Seagate Cheetah Ultra2Wide SCSI disk 10.000rpm

Nettverkskort Intel Pro 100+ full duplex 100Mb/s

Switch CNet CNSH-800 full duplex 10/100Mb/s

Disken er formatert spesielt for målingene, slik at filen i minst mulig grad skulle være fragmentert. Partisjonen det er lest fra er formatert med diskblokker på 64kB.

9.2 Målemetoder

Vi har målt hastigheten av buffermekanismene på følgende måte: Vi har lagt en meget stor fil på den nyformaterte partisjonen. Filen har vært på 2150888559 byte, litt over 2GB. Denne har vi så sendt fra disk og ut på nett. Vi har inkludert to eksempler på slike program i tillegg B.4. Programmene sender data ut så raskt som mulig.

Vi har målt tidsforbruk med BSD rutinen `gettimeofday`. Den gir tidspunkt med høy nøyaktighet. Vi har målt en gang før vi begynner å sende filen, en gang etter og regner ut differansen. Vi har kjørt alle målingene flere ganger og regnet ut snitt, minimum, maksimum og median av både tid og bitrate. Disse tallene og grafer over tidsforbruk finnes i tillegg C.

Vi har også målt forbruk av CPU-tid. BSD rutinen `getrusage` returnerer blant annet CPU-tid for prosessen, og CPU-tid brukt av kjernen på å utføre kall fra prosessen. Disse målingene finnes også i tillegg C.

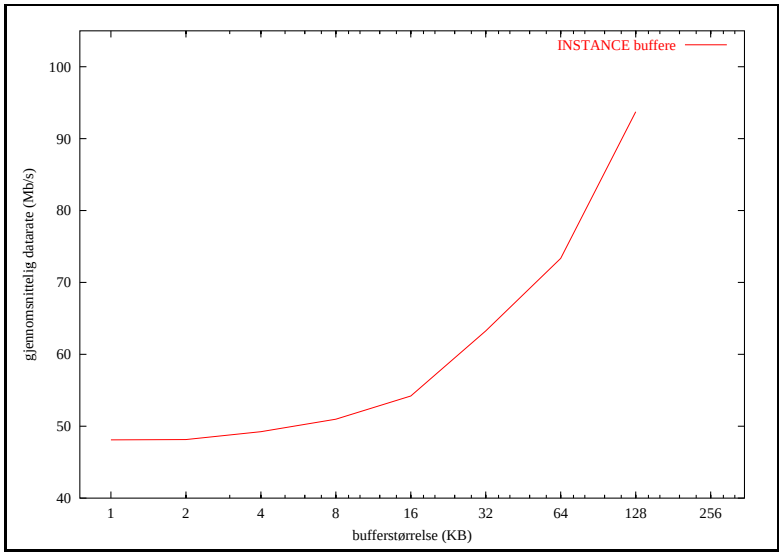
9.3 Måleresultater

Vi har testet både distribusjon med INSTANCE-buffere og de tradisjonelle socket-kallene `read` og `sendto`. Vi har først testet ut at konseptet virker ved å sende data med en lavere hastighet enn den fulle. Vi satte opp en klient på en annen maskin som tok imot filene korrekt. Vi har dermed vist at konseptet med INSTANCE-buffere virker.

Vi har testet med forskjellige bufferstørrelser fra 1kB til 2MB. Minste bufferstørrelse for et INSTANCE-buffer er lik størrelsen på diskblokken. I dette tilfellet har det vært 64kB. Når vi har testet INSTANCE-bufferet har vi dermed brukt buffere på minimum 64kB, mens størrelsen på lese og sende systemkallene tilsvarer den angitt bufferstørrelsen. I NETBSD tar ikke UDP laget i mot pakker på over ca. 9100 byte. Ved testing med tradisjonelle systemkall og størrelser over 8kB har vi derfor måttet bruke buffer og read kall med gitt størrelse, mens kall på `sendto` har vært splittet opp i kall på 8kB.

9.3.1 Test over 100Mb/s nettverk

Vi har testet ved å sende data fra testmaskinen over et 100Mb/s full duplex nettverk. Nettverket hadde ingen andre noder enn testmaskinen og den som tok i mot dataene. Figur 9.1 viser resultatet.



Figur 9.1: INSTANCE-buffere testet over 100Mb/s full duplex nettverk

Som vi ser har vi ingen datarate for den tradisjonelle buffermekanismen. Dette skyldes at de tradisjonelle systemkallene fikk nettverkskortet til å feile, slik at den beregnede dataraten ble over 100Mb/s (se tillegg C.1). Vi har ikke med resultater for INSTANCE-buffere over 128kB buffere. Vi når et snitt på 93,74MB/s ved buffer på 128kB, etter det forteller systemloggen at nettverkskortet svikter også her. Nettverkskortet ble dermed flaskehalsen i dette testoppsettet.

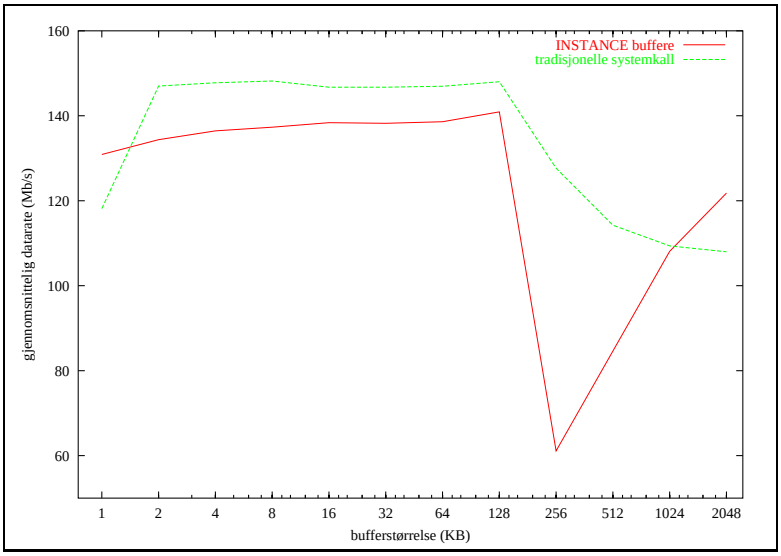
Vi har desverre ikke noen testmaskin som klarer å motta data med en datarate på over 93Mb/s. Vi antar alikevel utfra måleresultatene og det faktum at vi ikke får meldinger i systemloggen at dataene kommer riktig ut på nettverket.

9.3.2 Test til loopback adresse

I forrige avsnitt så vi at nettverkskortet vårt ble en flaskehals. Vi endret derfor testoppsettet noe. I stedet for å sende til en mottager over nettet satt vi opp en lokal mottager. Vi sendte så data til denne mottageren via nettverkssystemets loopback adresse. Figur 9.2 viser resultatet av denne testen.

Figur 9.2 viser noe skuffende at vår zero-copy datapath med INSTANCE-buffere kun gir høyere hastighet ved en bufferstørrelse på 1kB. Over 128kB buffer inntreffer noe som *kan* være en overbelastning i nettverkssystemet. Vi tviler derfor noe på de resultatene vi får ved disse størrelsene.

At en zero-copy datapath gir lavere ytelse enn kopiering antyder at det ikke er kopieringen av data som er flaskehalsen i testsystemet vårt. Vi antar at det er disken



Figur 9.2: INSTANCE-buffere testet over 100Mb/s full duplex nettverk

som er flaskehalsen. Da vi ikke hadde tid til å teste med et raskere disksystem, så vi etter andre løsninger. Vi valgte først å teste båndbredden til minnekopieringen.

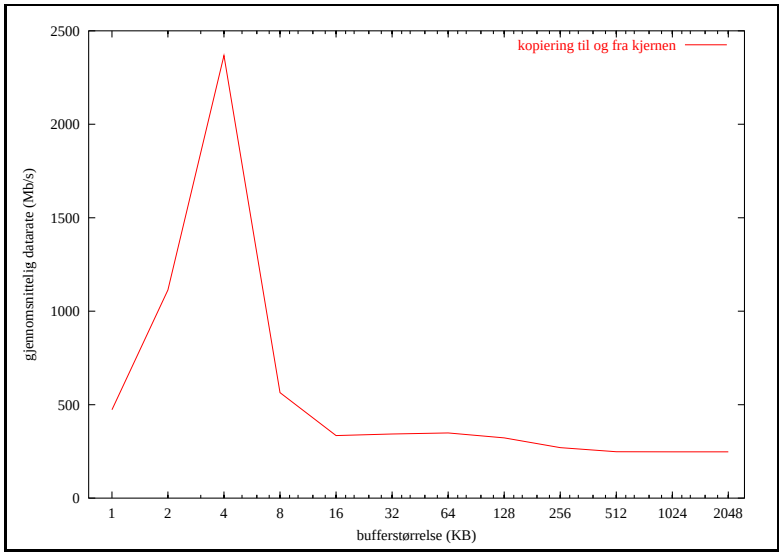
9.3.3 Test av kopiering i minnet

Vi har laget et nytt systemkall for testing av båndbredden ved minnekopiering. Dette kallet allokerer et buffer i kjernen Så kopieres data fra et buffer i brukerområdet inn i bufferet i kjernen. Dataene kopieres så tilbake til en nytt buffer i brukerområdet. Vi mener dette tilsvarer den kopiering som foregår ved skriving av lesing. Ved bruk av tradisjonelle systemkallene får vi først en kopiering fra diskbuffer og inn i applikasjonen, så en kopiering til et buffer i nettverkssystemet.

Vi har testet dette kallet med bufferstørrelser fra 1kB til 2MB. Vi har kopiert data tilsvarende 2150888559 byte, rundet opp til nærmeste hele antall kopieringsoperasjoner. Figur 9.3 viser resultatet.

Figur 9.3 viser en datarate langt høyere enn den vi oppnådde ved sending til loopbackadressen i figur 9.2. Vi mener derfor at det er svært sannsynlig at det er disken som begrenser datarateen ved sending til en loopback adresse.

INSTANCE-bufferet gir dårligere resultater enn de tradisjonelle kallene. Dette tyder på at vår implementasjon har en dårligere håndtering av disksystemet og nettverkssystemet enn den de tradisjonelle socketkallene har. Vi antar at dette skyldes



Figur 9.3: Kopiering av 2150888559 byte til og fra kjernen.

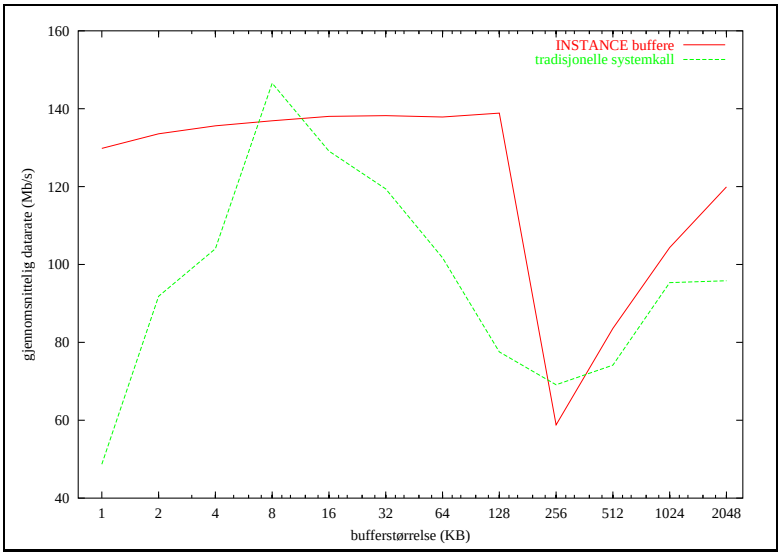
at INSTANCE-bufferne er delvis blokkernede i denne testsituasjonen. Etter at kallene `instance_read` og `instance_send` har "overlevert" dataene til disk eller nettverkssystemet, blokkeres bufferet inntil I/O-Done returnerer. I dette testoppsettet kan dette begrense ytelsen noe. I en antatt normalsituasjon vil vi ha mange datastrømmer støttet av flere INSTANCE-buffere. De vil i en normal situasjon ikke blokkere, da man oftest leser og skriver fra hvert sitt buffer. En forsinkelse i et enkelt INSTANCE-buffer vil heller ikke skade så lenge vi buffrer hos klientene. Vi er derfor usikre på om vi Vi er derfor usikre på om vi får en tilsvarende dårlig ytelse i håndteringen av disksystemet og nettverkssystemet i en reell implementasjon av en INSTANCE-node.

Det at INSTANCE-bufferet blokkerer betyr også at vi i de fleste situasjoner kan garantere at dataene blir sendt ut på nettverket. Vi venter jo på I/O-Done før vi går videre. Dermed får vi en hvis støtte for QoS i bufferhåndteringen.

9.3.4 Test til loopbackadresse ved høy CPU-last

Vi ønsket så å teste INSTANCE-bufferne i en situasjon der kopiering i kjernen var en begrensende faktor. Vi oppnådde dette ved å sette i gang en prosess som ikke gjorde annet enn å regne på flyttall. Denne førte til at lasten på testmaskinen gikk opp i over 1. Dermed måtte testprosessene våre konkurrere om CPU-en.

Figur 9.4 viser resultatet av dette forsøket. Vi ser at INSTANCE-bufferet med ett er raskere for de fleste bufferstørrelser.



Figur 9.4: INSTANCE-buffere testet over 100Mb/s full duplex nettverk

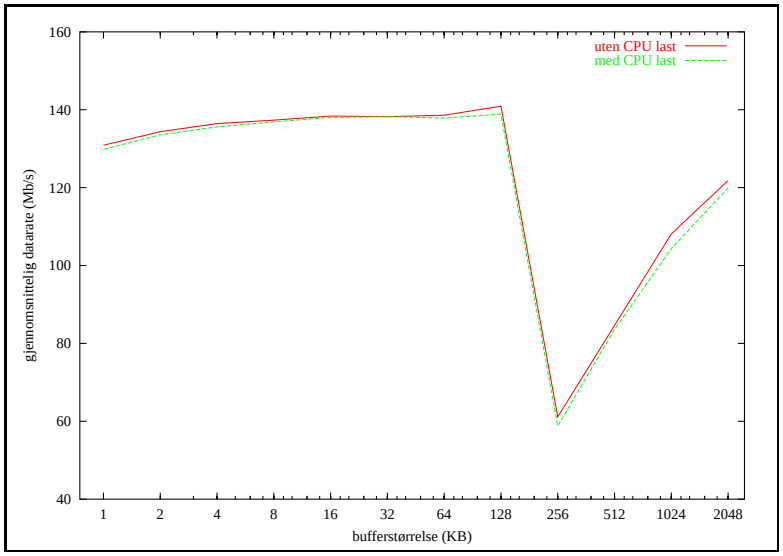
9.3.5 Sammenligning av resultater med og uten CPU-last

Vi sammenligner nå resultatet for hver av teknikkene med og uten CPU-last. Vi ser at INSTANCE-bufferets ytelse knapt påvirkes av at lasten på testmaskinen øker (se figur 9.5). For de tradisjonelle systemkallene faller ytelsen dramatisk (se figur 9.6).

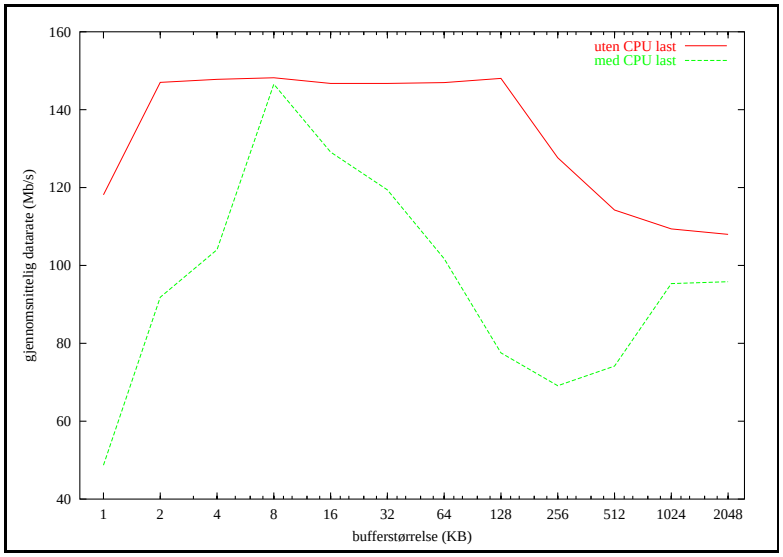
9.3.6 CPU tidsforbruk

Vi ser nå på hvor mye CPU-tid INSTANCE-bufferet og de tradisjonelle systemkallene bruker. Kjernen i NETBSD holder rede på hvor mye reell CPU tid hver prosess bruker. NETBSD er et multiprosesserings OS, og mange prosesser kjører i parallell. Det vi måler her er hvor mye tid prosessen reelt kjører på CPU-en.

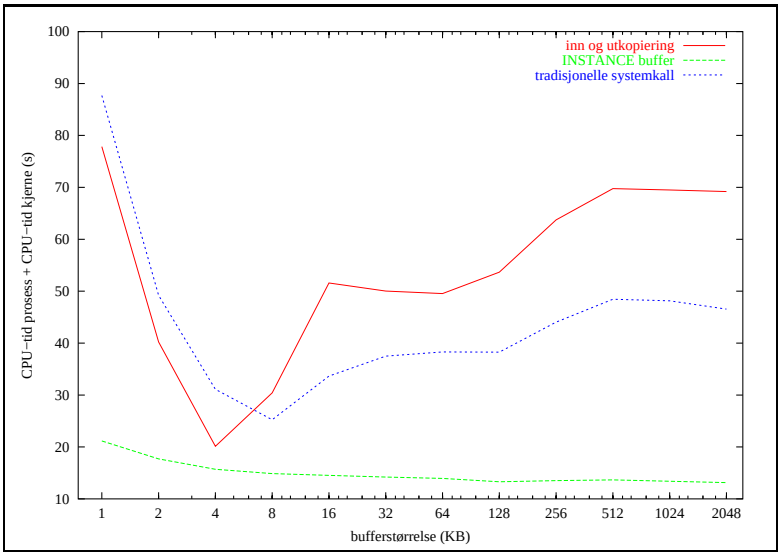
Figur 9.7 viser summen av CPU-tid brukt av prosessen og den CPU-tiden kjernen har brukt som følge av kall fra prosessen. Vi viser CPU-tid forbruket for INSTANCE-bufferet, de tradisjonelle systemkallene og kopiering inn og ut av kjernen. Figuren viser to ting. Den ene er at bruk INSTANCE-bufferne tar lite ressurser i form av CPU-tid. Det var ventet siden vi ikke belaster CPU-en med kopiering. Den andre tingen er at vi ser at variasjonene i forbruket til de tradisjonelle systemkallene ser ut til å ligne mye på variasjonene i CPU-tid brukt ved ut og innkopiering. Denne figuren viser dermed tydelig hva vi sparer ved å unngå kopiering internt i minnet.



Figur 9.5: INSTANCE-buffere med og uten CPU-last.



Figur 9.6: Tradisjonelle systemkall med og uten CPU-last



Figur 9.7: CPU-tid brukt av INSTANCE-bufferet, tradisjonelle systemkall og kopiering inn og ut av kjernen

Kapittel 10

Konklusjoner og videre arbeid

I dette kapittelet har vi først i avsnitt 10.1 en rask oppsummering av målet med oppgaven. I avsnitt 10.2 evaluerer vi resultatene vi har oppnådd. I avsnitt 10.3 ser vi på hva som gjenstår av arbeide med konsepter og implementasjon for å kunne utvikle en ferdig INSTANCE-node.

10.1 Oppsummering

Vi har i denne oppgaven ønsket å designe en ny buffermekanisme for de dedikert MoD-tjener kalt en INSTANCE-node. INSTANCE-noden mellomlagrer MM-data på vei mellom INSTANCE-tjeneren og klientene. INSTANCE-noden er optimalisert og skal raskest mulig sende data fra disken og ut på nettverket. INSTANCE-noden skal også tilby et visst QoS-nivå til klientene.

Som vi har beskrevet tidligere kan kopiering fra buffer til buffer være en begrensende faktor for ytelsen til en tjener som utfører mange og store I/O operasjoner. Sannsynligvis vi en stor andel av dataene i INSTANCE-noden være kontinuerlige datatyper som video og lyd. Kontinuerlige datatyper kjennetegnes gjerne ved at de gir mange og store I/O operasjoner. Målet med denne oppgaven har derfor vært å lage en zero-copy bufferhåndteringsmekanisme spesielt tilpasset INSTANCE-noden og med gode egenskaper når det gjelder å håndtere kontinuerlige datatyper.

10.2 Evaluering og tolkning av resultater

Målingene vi har gjort i vårt testsystem viser at vi oppnår en maksimal gjennomsnittlig datarate på 93,74Mb/s gjennom et 100Mb/s dull duplex nettverkskort. At

INSTANCE-bufferet er delvis blokkerende antyder at alle disse dataene kommer ut på nettet i riktig rekkefølge. Det samme gjør analysen av systemloggen. Vi har dessverre ikke noe testsystem som klarer å ta i mot data med en så høy rate så vi har ikke hatt muligheter til å verifisere dette.

Vi har sett at vår nye bufferhåndteringsmekanisme kun blir minimalt mindre effektiv når vi belaster CPU-en i testmaskinen. Hvis vi derimot gjør det samme når vi bruker de tradisjonelle systemkallene `read` og `sendto` faller ytelsen betydelig. Målinger av forbrukt CPU-tid viser også at vår nye INSTANCE-buffer mekanisme bruker langt mindre CPU-tid en de tradisjonelle systemkallene.

Vi finner dessverre at dataraten ut fra disken er en begrensende faktor i testoppsettet vårt. Målinger av kopiering inn og ut av kjernen antyder at vi ikke vil nå begrensinger i form av manglende kapasitet for bufferkopiering før vi når en datarate på et sted mellom 250Mb/s og 500Mb/s avhengig av bufferstørrelsen. Disse målingene er foretatt uten at andre prosesser belastet CPU-en i særlig grad. Derfor vil vi trolig nå denne grensen tidligere i en reell implementasjon av INSTANCE-noden. I den endelige versjonen vil vi også måtte ha andre systemer i noden som belaster CPU-en.

For å få realistiske målinger av vårt nye konsept vil vi trolig trenge både et raskere nett som gigabit Ethernet og en raskere disksystem som for eksempel et disk RAID. Først da vil vi virkelig ha muligheter for å måle den reelle maksimale ytelsen for bufferhåndteringsmekanismen vår.

Målingene viser at buffermekanismen vår virker og at den belaster CPU-en minimalt. Vi mener at vi har belegg for å si at vi har utviklet en effektiv zero-copy, in-kernel bufferhåndteringsmekanisme. På grunn av svakheter ved måleoppsettet vårt har vi dessverre ikke vært i stand til å måle nøyaktig hvor effektiv mekanismen er.

10.3 Videre arbeid

Vi har delt avsnittet om videre arbeid i to deler. Avsnitt 10.3.1 tar for seg sider ved INSTANCE-konseptet som som inngår i bufferhåndteringen eller som vil kunne virke inn på denne. I avsnitt 10.3.2 ser vi på problemer ved den implementasjonen vi har gjort og hva vi mener gjenstår her før INSTANCE-noden kan settes i drift. Til slutt i avsnitt 10.3.3 beskriver vi testoppsett vi gjerne skulle ha gjennomført.

10.3.1 Konsepter

Vi ser her på deler av konseptet vi mener trenger mere arbeid. Vi ser først på de delene av konseptet som direkte inngår i bufferhåndteringen. Vi ser så på andre

sider ved INSTANCE-noden vi mener kan tenkes å påvirke designet av bufferhåndteringen.

10.3.1.1 Dataflyten mellom noden og klientene

Flyten av data mellom INSTANCE-noden og klientene må vurderes videre. Vi vet at mye av dataene vil består av kontinuerlige datastrømmer. Endelig valg av metode, eller kanskje mer sansynlig et sett av flere metoder venter til senere.

10.3.1.2 Håndtering av diskrete data

Vi har gjort et bevist valg ved å designe INSTANCE-noden for håndtering av kontinuerlige datastrømmer. Det betyr at vi må håndtere diskrete data som tekst og bilder på en annen måte. Dette kan enten av andre mekanismer for bufferhåndtering enn den vi har utviklet, eller overlates til tradisjonelle webtjenere som allerede er optimalisert for denne type data. Vi føler imidlertid at dette er et område som må vurderes nøyere før INSTANCE-noden settes i drift.

10.3.1.3 Forholdet mellom tjener og node

Overføringer fra INSTANCE-tjener til INSTANCE-node trenger mer arbeide. Vi har i oppgaven vår basert oss på å beregne egenskaper ved datastrømmene idet data overføres fra INSTANCE-tjeneren til INSTANCE-noden. Hvordan dette skjer, og hvilke verdier vi beregner vil kunne være viktig for bufferhåndteringen.

10.3.1.4 Tilbakeflyt av data

Hvordan vi håndterer tilbakeflyt av data fra klientene til INSTANCE-tjeneren er ikke avklart. Det beste ville trolig være å basere seg på en form for tre-topologi der meldingene samles opp før de når noden, slik at ikke alle klientene aksesserte INSTANCE-tjeneren hver for seg. Da ville jo denne tilbakemeldingen bli en flaskehals i systemet. Foreløpig baserer vi oss på et system uten tilbakemeldinger. Ulempen er at vi da ikke har kunnskaper om dataenes popularitet.

10.3.1.5 Schedulering

Schedulering vil være viktig for å få INSTANCE-noden til å fungere optimalt. Standard NETBSD har dessverre ingen støtte for real-time schedulering. Det betyr at oppgaven med å holde en jevn datarate ved lesing og utsending av data må løses med ad-hoc løsninger. Ideelt skulle vi hatt real-time schedulerings egenskaper i INSTANCE-noden. Dette faller imidlertid utenfor oppgavens tema, og må derfor vurderes av andre.

10.3.1.6 Andre sider av INSTANCE-konseptet

Kapittel 2 beskriver at INSTANCE-noden skal lagre ferdig nettverkspakker på disk. Hvorvidt dette vil øke hastigheten eller ikke og hvilken påvirkning dette vil ha på fleksibiliteten i INSTANCE-noden er viktige spørsmål. Dette er problemer andre i INSTANCE-gruppen ser på. Det er også arbeide på gang med å slå sammen feilkorrigerende algoritmer i RAID-disker og i nettverket. Begge disse problemene faller dessverre utenfor problemstillingen i denne oppgaven. Vi venter i spenning på resultatene.

10.3.2 Implementasjon

På grunn av begrenset tid gjenstår det å implementere en del konsepter. Muligens kunne også noe vært implementert mer effektivt. Det hadde selvsagt vært en fordel vi hadde rukket å gjøre dette. Vi tror dette hadde gitt oss et bedre system, og muligens også noe bedre måleresultater. Det er dessverre også en del kontrollsystemer rundt bufferne som kontroll av pekere og håndtering av minnelekasjer som gjenstår. Alt dette er ting vi ville implementert dersom tiden hadde strukket til.

I tillegg er det en del mangler ved sikkerheten i koden. INSTANCE-noden er ment å skulle være en selvstendig enhet uten “uvedkommende” brukere og prosesser. Det betyr at muligheten for ondsinnet utnyttelse av mangler i implementasjonen er mindre, men det hadde allikevel vært en klar fordel om disse problemene ble rettet.

10.3.2.1 Manglende kontroll med INSTANCE-buffer pekere

Det er dessverre noen sikkerhetshull i implementasjonen slik den er i dag. Det kanskje største er pekeren til INSTANCE-strømmene som brukerprosessen mottar ved kall på `instance_alloc` og sender ved kall på `instance_read` og `instance_send`. Hvis pekeren peker på et INSTANCE-strøm, sjekkes det at prosessen er “eier” av bufferet. Den feilen som derimot ikke fanges opp er dersom

pekeren *ikke* peker på et INSTANCE-buffer. Dette vil gi kræsj i kjernen. Dette skyldes at det er opp til prosessen å holde orden på pekerne. Kjernen har ingen tabell eller tabeller over buffer pekerne.

En mulig løsning ville være å benytte indirekte pekere slik det gjøres med fildeskriptorer. En slik indirekte peker er en index i en fildeskriptor tabell som håndteres av kjernen. Dette gjør det mulig å holde kontroll med pekerne slik at kjernen kan sjekke om pekeren er en gyldig peker før operasjonen med pekeren igangsettes.

10.3.2.2 Minnelekasje

Et annet problem som oppstår fordi vi ikke har en sentral oversikt over instance buffere er muligheten for minnelekasjer.

Slik implementasjonen er i dag er det ingen som kontrollerer at prosessene frigjør INSTANCE-bufferne når de ikke lenger er i bruk. Det er heller ikke noe system i kjernen som fjerner bufferne dersom prosessen skulle dø. Begge disse problemene vil kunne forårsake lekkasje av minne, dvs. minne som ikke er i bruk men heller ikke er frigjort slik at det kan tas i bruk igjen.

En løsning som skissert i avsnitt 10.3.2.1 med en peker tabell i kjernens prosess struktur vil kunne løse dette problemet slik dette i dag løses med frigjøring av filer ved prosess-kræsj.

10.3.3 Målinger

Det hadde vært ønskelig om vi kunne fått flere måleresultater. Først og fremst skulle vi gjerne hatt tid til å måle på konseptet for bufferhåndtering med et raskere nettverk og raskere disker. Først da vil vi kunne få bekreftet det vi tror om at disse bufferhåndteringsmekanismene er mere effektive enn de tradisjonelle systemkallene.

Vi kunne også tenke oss andre måleoppsett enn å måle maksimal datarate i én datastrøm. Vi kunne tenke oss mange prosesser på INSTANCE-noden som hver sender ut MM-data med datarater mellom 1Mb/s og 10Mb/s. Vi ville også gjerne hatt mange klienter i et testoppsett slik at vi kunne kontrollere at alle dataene kom frem. For å få et bra resultat måtte vi trolig ha en Real-Time schedulering i INSTANCE-noden. Siden denne ikke er klar må slike målinger vente.

Det foregår arbeid med å implementere en ny prototype på en INSTANCE-node på en maskin med gigabit Ethernet. På den maskinen er det ment å testes både vår eget bufferhåndteringskonsept og mmbuf[10]. Dette arbeidet vil først være ferdig etter at denne oppgaven er levert. Vi venter i spenning på resultatet av disse testene.

Del III

Tillegg

Tillegg A

Forkortelser og enheter

Vi har her listet opp forkortelser med norsk og engelsk forklaring. For mer informasjon, slå opp i indeksen på side 185.

A.1 Forkortelser

A.2 Enheter

Vi gir her en liten oversikt over enheter brukt i datamaskiner. En faktor som definitivt skaper forvirring er bruken av begrepene kilo (k), mega (M) og giga (G). Egentlig står disse enhetene henholdsvis for tallene $10^3 = 1.000$, $10^6 = 1.000.000$ og $10^9 = 1.000.000.000$. Siden addresseringen i en datamaskin foregår i det binære tallsystem, er ofte enhetene byttet ut med $k = 2^{10} = 1024, M = 2^{20} = 1024x1024$ og $G = 2^{30} = 1024x1024x1024$. Problemet er at dette ikke *alltid* gjelder. Når en produsent ønsker et stort tall for å selge et produkt brukes gjerne 1 GB om $10^9 = 1.000.0000.000byte$. Andre steder er igjen $1GB = 2^{30} = 1.073.741.824byte$.

Forkortelse	Engelsk	Norsk
ADU	Application Data Unit	
CBR	Constance Bit Rate	konstant datarate
CDL	Constant Data Length	konstant datalengde
CoW	Copy-on-Write	kopiering ved skriving
CPU	Central Processing Unit	prosessoren
CTL	Constanct Time Lengt	konstat tidslengde
DMA	Direct Memory Access	direkt minneaksess
DRAM	Dynamic Random Access Memory	
GDB	Greedy Discconserving Broadcasting	
HNI	Host Network Interface	
ILP	Integrated Layer Processing	
INSTANCE	Intermediate Storage Node Concept	
IO	Input Output	
ISO	International Standards Organization	
MM	Multimedia	multimedia
MoD	Multimedia on Demand	
NVoD	Near Video on Demand	
OS	Operating System	Operativsystem
OSI	Open Systems Interconnection	
QVoD	Quasi Video on Demand	
QoS	Quality of Service	tjenestekvalitet
RSVP	Resource Reservation Protocol	
SRAM	Static Random Access Memory	
TCoW	Trancient Copy-on-Write	midlertidig kopiernig ved skriving
UNIK		Universitetsstudiene på Kjeller
UVM		
VBR	Variabel Bit Rate	variabel bitrate
VoD	Video on Demand	

Tabell A.1: Forkortelser med norsk og engelsk forklaring.

Enhet	Betydning	Forklaring
b	bit	Et binært siffer, enten 1 eller 0.
B	byte	1 byte = 8 bits.
k	kilo	$10^3 = 1000$, eller $2^{10} = 1024$.
kb	kilobit	10^3 bits.
kB	kilobyte	2^{10} bytes
kbskbps	kilobitssek	Kilobits per sekund.
M	mega	10^6=1.000.000, eller $2^{20} = 1024x1024 = 1.048.576$.
Mb	megabit	10^6 bits.
MB	megabyte	2^{20} bytes.
Mbsmbps	megabitssek	Megabits per sekund.
G	giga	$10^9 = 1.000.000.000$, eller $2^{30} = 1024x1024x1024 = 1.073.741.824$.
Gb	gigabit	10^9 bits
GB	gigabyte	2^{30} bytes.
GbsGbps	gigabitssek	Gigabits per sekund.

Tabell A.2: Oversikt over enheter brukt i oppgaven

Tillegg B

Kildekode

B.1 instance_syscalls.c

```
/*****************************************************************************/
/* instance_syscalls.c                                                       */
/*                                                                           */
/* (C) 2000 Espen Jorde                                                      */
/*                                                                           */
/* This is free software; you can redistribute it and/or                     */
/* modify it under the terms of the GNU Library General Public License as    */
/* published by the Free Software Foundation; either version 2 of the        */
/* License, or (at your option) any later version.                           */
/*                                                                           */
/* This file is distributed in the hope that it will be useful,              */
/* but WITHOUT ANY WARRANTY; without even the implied warranty of            */
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU         */
/* Library General Public License for more details.                          */
/*                                                                           */
/* You should have received a copy of the GNU Library General Public         */
/* License along with the GNU C Library; see the file COPYING.LIB.  If not,  */
/* write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,  */
/*   Boston, MA 02111-1307, USA.                                             */
/*                                                                           */
/*****************************************************************************/


#ifdef INSTANCE

#include "instance_syscalls.h"
```

B.1.1 sys_instance_alloc

```
/***************************/
/***************************/
/******** SYS_ALLOC *********/
```

```
/***************************/
/***************************/

/* Allocate an instance_stream and two instance_buffers       */
/* Take name of file and a internet address as parameters.    */
/* That way, the calling process has neither direct access to */
/* the file nor the socket. This makes a lot of checking for  */
/* each read and send unneccesary since the process cannot    */
/* change the state of the file and socket.                   */

/*****************************************************************************/
/*                        PARAMETERS                                         */
/*                                                                           */
/* IN:  filename    = name of file to read from                              */
/*      address     = ip-address and port number to send to                  */
/*      arg_bufsize = required size of each of the two buffers               */
/* OUT: pointer to instance_stream containing buffers, mbuf's, buf's etc.    */
/*                                                                           */
/*                                                                           */
/*                           RULES                                           */
/*                                                                           */
/* filename:    Must refer to a file which can be opened read only by the    */
/*                calling process. The file _must_ be on a local             */
/*                filesystem                                                 */
/* address:     Must be a legal IP address refering two twe reciving         */
/*                IP address and UDP port. Can be a broadcast or multicast   */
/*                address.                                                   */
/* arg_bufsize: Muste be greater than zero. Note that the true buffersize    */
/*                is rounded up two the nearest disk block size.             */
/*                                                                           */
/*****************************************************************************/

/* ARGSUSED */
int
sys_instance_alloc(p, v, retval)
     struct proc *p;
     void *v;
     register_t *retval;
{
     struct sys_instance_alloc_args /* {
         syscallarg(char *) file_path;
         syscallarg(struct sockaddr_in *) address;
         syscallarg(size_t) arg_bufsize;
         syscallarg(struct instance_stream **) return_stream;
     } */ *uap = v;


  char *file_path;                  /* Path of file to read */
  struct sockaddr_in *address;      /* Address to send to */
  size_t arg_bufsize;               /* required (minimum) buffer size */
  size_t bufsize;
  int file_fdesc;                   /* File file descriptor reference */
  int sock_fdesc;                   /* Socket file descriptor reference */
  struct filedesc *fdp;             /* File descriptor pointer */
  struct file *file_fp;             /* File file pointer */
  struct file *socket_fp;           /* Socket file pointer */
  struct socket *socket;            /* Socket pointer */
  struct vnode *vnode;              /* vnode pointer */
  struct instance_buffer *i_buf[1];
                                    /* instance_buffer array */
  struct instance_stream *i_stream; /* */
  struct inode *inode;              /* inode pointer */
```

```
struct fs *file_sys;              /* File system pointer */
int flags;
int error;
int creat_mode;                   /* Creation mode of file */
struct nameidata namei;           /* Used for lookup of names */
size_t fs_blocksize;              /* Blocksize of filesystem */
struct mbuf *mbuf;                /* mbuf pointer */
struct mbuf *prev_mbuf = NULL;
int s;
int num_buf;
int num_mbuf;
size_t buf_data_size;
size_t mbuf_data_size;
void *data_area;
struct buf *buf;
int i, j;
size_t size;
size_t last_mbuf_size;
int interface_mtu;
struct instance_stream **return_stream;


/*********************/
/* RETRIVE ARGUMENTS */
/*********************/

file_path =  SCARG(uap, file_path);
address = SCARG(uap, address);
arg_bufsize = SCARG(uap, arg_bufsize);
return_stream = SCARG(uap, return_stream);

ALLOC_DBG(("instance_alloc called\n"));


/* Return NULL if not successfull */
*return_stream = NULL;

error = 0;

/*******************/
/* CHECK ARGUMENTS */
/*******************/

if (address == NULL)
  {
    /* ERROR */
    error = EINSTANCE_NO_ADDRESS;
    ALLOC_DBG(("Error NO ADDRESS\n"));
    return error;
  };

if (file_path == NULL)
  {
    /* ERROR */
    error = EINSTANCE_NO_FILE;
    ALLOC_DBG(("Error NO FILE\n"));
    return error;
  };


if (address->sin_family != AF_INET)
  {
```

```
      /* ERROR */
      error = EINSTANCE_WRONG_SOCK_FAMILY;
      ALLOC_DBG(("Error WRONG SOCK FAMILY\n"));
      return (error);
    };



  /* Get the file descriptor table of the calling process */
  fdp = p->p_fd;



  /* All buffers must be round_ups of pagesize */
  bufsize = round_page(arg_bufsize);


/*    ALLOC_DBG(("bufsize: %d, arg_buf_size: %d\n", bufsize, arg_bufsize)); */

  /*******************************************************/
  /* CHECK FILEPATH, OPEN FILE AND GET A FILE DESCRIPTOR */
  /*******************************************************/


  /* Open file en READ ONLY mode */
  flags = FREAD;

  /* Allocate file struct and process file descriptor */
  error = falloc(p, &file_fp, &file_fdesc);

  if (error)
    {
      /* ERROR */
      ALLOC_DBG(("error file alloc\n"));
      error = EINSTANCE_FILE_ALLOC;
      return (error);
    };

  /* Set mode for filedes creation */
  creat_mode = ((0 &~ fdp->fd_cmask) & ALLPERMS) &~ S_ISTXT;

  /* Initialize nameidata struct */
  NDINIT(&namei, LOOKUP, FOLLOW, UIO_USERSPACE, file_path, p);

  p->p_dupfd = -file_fdesc - 1;

  /* Open file */
  error = vn_open(&namei, flags, creat_mode);
  ALLOC_DBG2(("vnode opened\n"));

  if (error)
    {
      /* ERROR */
      ALLOC_DBG(("error vnode_open\n"));

      /* Release file and reset tables */
      error = EINSTANCE_VNODE_OPEN;
      goto release_file;
    };


  /* Fix process pointer */
```

```
p->p_dupfd = 0;

/* Get vnode poiter */
vnode = namei.ni_vp;
ALLOC_DBG2(("vnode fra namei: 0x%x\n", (u_int)vnode));

/* Set values in file pointer */
file_fp->f_flag = flags & FMASK;
file_fp->f_type = DTYPE_VNODE;
file_fp->f_ops = &vnops;    /* Pointer to operations on a vnode */
file_fp->f_data = (caddr_t) vnode;

ALLOC_DBG2(("Vnode u f_data: 0x%x\n", (u_int)file_fp->f_data));

/* unlock vnode */
VOP_UNLOCK(vnode, 0);
ALLOC_DBG2(("vnode etter unlock: 0x%x\n", (u_int)vnode));

/* Get inode pointer */
inode = VTOI(vnode);
ALLOC_DBG2(("inode: 0x%x\n", (u_int)inode));
ALLOC_DBG2(("vnode etter VTOI: 0x%x\n", (u_int)vnode));
file_sys = inode->i_fs;  /* Associated FS of the inode (FFS). */
                         /*If NOT a (local?) FFS, an error occurs */

/* Get size of blocks in filesystem */
fs_blocksize = file_sys->fs_bsize;

if (fs_blocksize != INSTANCE_BLOCK_SIZE)
  {
    /* Trying to use an INSTANCE_BUFFER on a filesystem where block size */
    /* Is diffrent from INSTANCE_BLOCK_SIZE */
    /* return (NULL); */
  };



/*********************************/
/* ALLOCATE AND OPEN A UDP SOCKET */
/*********************************/

/* Allocate socket file pointer and process socket file descriptor */
error = falloc(p, &socket_fp, &sock_fdesc);
if (error)
  {
    /* ERROR */
    error = EINSTANCE_SOCK_ALLOC;
    ALLOC_DBG(("error alloc socket"));

    /* Release file and reset tables */
    goto release_file_and_vnode;
  };
ALLOC_DBG2(("socket allocated\n"));

/* Set socket values */
socket_fp->f_flag = FREAD | FWRITE;
socket_fp->f_type = DTYPE_SOCKET;
socket_fp->f_ops = &socketops;   /* Pointer to operations on a socket */
ALLOC_DBG2(("Socketops: 0x%x\n",(u_int) &socketops));

/* Create socket */
```

```
error = socreate(AF_INET, &socket, SOCK_DGRAM, 0);

if (error)
  {
    /* ERROR */
    error = EINSTANCE_SOCK_CREATE;
    ALLOC_DBG(("SOCKET CREATE\n"));

    /* Release file and socket. Reset tables */
    goto release_file_and_vnode;
  };



socket_fp->f_data = (caddr_t) socket;



/*  */

error = sockargs(&mbuf, (struct sockaddr *) address,
                 sizeof(struct sockaddr), MT_SONAME);


if (error)
  {
    /* ERROR */
    error = EINSTANCE_SOCK_ARGS;
    ALLOC_DBG(("Error SOCKET ARGS\n"));

    /* Release file and socket. Reset tables */
    goto release_file_vnode_and_socket;
  };


error = soconnect(socket, mbuf);

if (error)
  {
    /* ERROR */

    m_freem(mbuf);
    ALLOC_DBG(("Error SOCONNECT %d\n", error));
    error = EINSTANCE_SOCONNECT;

    /* Release file and socket. Reset tables */
    goto release_file_vnode_and_socket;
  };

if ((socket->so_state & SS_NBIO) && (socket->so_state & SS_ISCONNECTING))
  {
    /* ERROR */
    m_freem(mbuf);
    error =  EINPROGRESS;

    goto release_file_vnode_and_socket;
};

s = splsoftnet();
while ((socket->so_state & SS_ISCONNECTING) && socket->so_error == 0)
  {
    error = tsleep((void *)&socket->so_timeo, PSOCK | PCATCH, netcon, 10);
```

```
      if (error)
        {
          break;
        };
    };

  if (error == 0)
    {
      error = socket->so_error;
      socket->so_error = 0;
    };

  splx(s);
  socket->so_state &= ~SS_ISCONNECTING;
  m_freem(mbuf);
  if (error == ERESTART)
    {
      ALLOC_DBG(("Erestart\n"));

      error = EINTR;
    };

  /* SUCSESS SOCKET CONNECTED */
  ALLOC_DBG2(("Socket connected!\n"));


  size = (sizeof (struct instance_stream));
  /* ALLOCATE INSTANCE STREAM */
  MALLOC(i_stream, struct instance_stream *, sizeof (struct instance_stream), M_INSTANCE, M_NOWAIT);

  ALLOC_DBG2(("i_stream allocated\n"));


  if (i_stream == 0)
    {
      /* ERROR */

      error = EINSTANCE_STREAM_ALLOC;
      ALLOC_DBG(("ERROR STREAM alloc!\n"));

      goto release_file_vnode_socket_and_more;
    };

/*    ALLOC_DBG(("alloc i_stream 0x%x-0x%x\n", (u_int) i_stream, (u_int) i_stream + size)); */


  /* Calculate size of buf data area and mbuf size */
  interface_mtu = INSTANCE_MTU;

  buf_data_size = fs_blocksize;

  mbuf_data_size = interface_mtu - sizeof(struct udpiphdr);

  ALLOC_DBG2(("MTU: %d, mbuf_data_size: %d\n", interface_mtu,sizeof(struct udpiphdr)));
  /* Calculate number of bufs */
  num_buf = (bufsize / buf_data_size);

  if (bufsize % buf_data_size)
    {
      num_buf++;
    };
```

```
  /* Final calculation of bufsize in whole number of bufs */
  bufsize = num_buf * buf_data_size;

  /* Calculate number of mbufs */
  num_mbuf = (bufsize / mbuf_data_size);

  if (bufsize % mbuf_data_size)
    {
      num_mbuf++;
    };



  /*********************************/
  /* SET VALUES IN INSTANCE STREAM */
  /*********************************/

  /* General */
  i_stream->p = p;
  i_stream->buf_size = bufsize;
  i_stream->end_of_file = 0;

  /* File handeling */
  i_stream->file_fp = file_fp;
  i_stream->vnode = vnode;
  i_stream->file_fdesc = file_fdesc;
  i_stream->file_system = file_sys;

  /* buf handeling */
  i_stream->num_buf = num_buf;
  i_stream->prev_read_arg_offset = -1;        /* 0 is not read */
  i_stream->buf_data_size = buf_data_size;

  /* Socket handeling */
  i_stream->addr = address;
  i_stream->socket = socket;
  i_stream->socket_fp = socket_fp;
  i_stream->socket_fdesc = sock_fdesc;

  /* mbuf handeling */
  i_stream->num_mbuf = num_mbuf;
  i_stream->prev_send_arg_offset = -1;        /* 0 is not read */
  i_stream->mbuf_data_size = mbuf_data_size;


  ALLOC_DBG2(("2mbuf_data_size: %d\n", i_stream->mbuf_data_size));


  /* ALLOCATE INSTANCE BUFFERS */
  for (i=0; i<2; i++)
    {

        MALLOC(data_area, void *, bufsize, M_INSTANCE, M_NOWAIT);
/*        data_area = (void *) uvm_km_alloc(kmem_map, bufsize); */
      ALLOC_DBG2(("Data area allocated ibuf[%d]: 0x%x\n", i, (u_int)data_area));

      if (data_area == 0)
        {
          /* ERROR */
          error = EINSTANCE_BUFFER_ALLOC;
          ALLOC_DBG(("Error buffer alloc!\n"));
```

```
    goto release_file_vnode_socket_and_more;
  };


size = sizeof(struct instance_buffer)
  + (sizeof(struct buf) * num_buf)
  + (sizeof(struct mbuf) * 2 * num_mbuf); /* two mbuf per data area */

MALLOC(i_buf[i], struct instance_buffer *, size, M_INSTANCE, M_NOWAIT);

ALLOC_DBG2(("ibuf[%d] allocated: 0x%x\n", i, (u_int) i_buf[i]));

if (i_buf[i] == NULL)
  {

    /* ERROR */
    if (i == 1)
      {
          FREE(i_buf[0], M_INSTANCE);
          FREE(i_buf[0]->data, M_INSTANCE);
      };

      FREE(data_area, M_INSTANCE);
    error = EINSTANCE_IBUF_ALLOC;
    ALLOC_DBG(("ERROR Ibuf alloc\n"));


    goto release_file_vnode_socket_and_more;

  };


/* buf points to the buf region of the instance buffer */
buf = (struct buf *) (((int )i_buf[i]) + sizeof(struct instance_buffer));


/* mbuf points to the mbuf region of the instance buffer */
mbuf = (struct mbuf *) (((int)i_buf[i]) + sizeof(struct instance_buffer)
                        + (sizeof(struct buf) * num_buf));


/***************************************/
/* Setting values in the instance_buffer */
/***************************************/

/* General values */
i_buf[i]->i_stream = i_stream;

/* next is set later */
i_buf[i]->data = data_area;
i_buf[i]->state = INSTANCE_READ_OK;
i_buf[i]->error = 0;
i_buf[i]->io_pending = 0;
i_buf[i]->io_being_called = 0;

/* buf handeling */
i_buf[i]->first_buf = buf;

/* mbuf handeling */
i_buf[i]->first_mbuf = mbuf;
i_buf[i]->curr_mbuf = mbuf;
```

```
      /* Values used when sending: */
      i_buf[i]->start_of_data_area = 0;



      /**********************/
      /* ALL BUF STRUCTURES */
      /**********************/

      for (j=0;j < num_buf ; j++, buf++)
        {
          buf->b_dev = NODEV;
          buf->b_error = 0;
          buf->b_resid = 0;
          buf->b_dirtyoff = 0;
          buf->b_dirtyend = 0;
          buf->b_validoff = 0;
          buf->b_validend = 0;
          buf->b_rcred = NOCRED;
          buf->b_wcred = NOCRED;
          buf->b_proc = NULL;
          buf->b_vp = i_stream->vnode;
          ALLOC_DBG2(("buf->b_vp: 0x%x\n", (u_int)i_stream->vnode));
          buf->b_data = (caddr_t) ((u_int) data_area + (j * (u_int)buf_data_size));
          ALLOC_DBG2(("b_data: 0x%x\n", (u_int)buf->b_data));
          buf->b_iodone = instance_read_done;
          buf->b_bufsize = i_stream->buf_data_size;
          ALLOC_DBG2(("buf->b_bufsize: %u\n", (u_int) buf->b_bufsize));

          /* This variable is introduced to buf by INSTANCE */
          buf->iodone_args = i_buf[i];
          ALLOC_DBG2(("buf->b_saveaddr: %u\n", (u_int) buf->b_saveaddr));

        };

      PRINT_BUF(buf);

      ALLOC_DBG2(("First mbuf: 0x%x\n", (u_int)mbuf));

      /***********************/
      /* ALL MBUF STRUCTURES */
      /***********************/

      ALLOC_DBG2(("Data_area: 0x%x\n", (u_int)data_area));
/*        ALLOC_DBG2(("num_mbuf: %d\n",num_mbuf)); */
      for (j = 0; j < num_mbuf; j++)
        {
/*           ALLOC_DBG(("header mbuf: 0x%x\n", (u_int)mbuf)); */

          /* HEADER MBUF */

          mbuf->m_type = MT_DATA;
          mbuf->m_data = mbuf->m_pktdat;
          mbuf->m_flags = M_PKTHDR | M_INST_MBUF;
          mbuf->m_len = 0;
          mbuf->m_pkthdr.rcvif = NULL;
          mbuf->m_pkthdr.len = i_stream->mbuf_data_size;

          prev_mbuf = mbuf;
          mbuf++;
```

```
          /* Link header mbuf and ext mbuf in mbuf chain */

          prev_mbuf->m_next = mbuf;
/*            ALLOC_DBG(("ext mbuf: 0x%x\n", (u_int)mbuf)); */

          /* PACKET MBUF (EXT MBUF) */

          mbuf->m_type = MT_DATA;
          mbuf->m_flags = M_EXT | M_INST_MBUF;
          mbuf->m_data = (caddr_t) ((u_int)data_area +
                                    ( j * (u_int)i_stream->mbuf_data_size ));
          mbuf->m_len = i_stream->mbuf_data_size;
          mbuf->m_next = NULL;
          mbuf->m_nextpkt = NULL;
          mbuf->m_ext.ext_buf = mbuf->m_data;
          mbuf->m_ext.ext_free = instance_send_done;
          mbuf->m_ext.ext_type = M_INST_MBUF;
          mbuf->m_ext.ext_arg = (void *) i_buf[i];
          mbuf->m_ext.ext_prevref = mbuf;
          mbuf->m_ext.ext_nextref = mbuf;

          mbuf++;
        };

      PRINT_MBUF_H(prev_mbuf);
      PRINT_MBUF_E(prev_mbuf->m_next);


      /* Last mbuf pair has not neceserrely the same size as the rest */

      if ((last_mbuf_size = (size_t) (u_int)bufsize % (u_int)mbuf_data_size))
        {
          prev_mbuf->m_pkthdr.len = last_mbuf_size;
          (--mbuf)->m_len = last_mbuf_size;

          /* Also the last header mbuf must be correct */
          (--mbuf)->m_pkthdr.len = last_mbuf_size;
          i_stream->last_mbuf_size = last_mbuf_size;
        }
      else
        {
          i_stream->last_mbuf_size = 0;
        };

      ALLOC_DBG2(("Last mbuf: 0x%x\n", (u_int)mbuf));

      /* Last_mbuf points to last header mbuf, which is */
      /* the one before the last mbuf */
      i_buf[i]->last_mbuf = mbuf;


    }; /* End ibuf loop */



  /* Link the two buffers */
  /* Simplyfies jumping from one i_buf to the other */
  i_buf[0]->next = i_buf[1];
  i_buf[1]->next = i_buf[0];
```

```
  /*********************************/
  /* SET VALUES IN INSTANCE STREAM */
  /*********************************/


  /* Instance buffer */
  i_stream->read_ibuf = i_buf[0];
  i_stream->send_ibuf = i_buf[0];
  i_stream->first_ibuf = i_buf[0];



  /* Retur successfully */
  *return_stream = i_stream;

  ALLOC_DBG(("Success!!!\n"));

  return (0);


  /*******************************/
  /* END IF SUCCESFULL EXECUTION */
  /*******************************/


  /*********************************/
  /* RELEASE POINTERS ETC. IF ERROR */
  /*********************************/

 release_file_vnode_socket_and_more:
  soo_close(socket_fp, p);

 release_file_vnode_and_socket:
  /* Release file and reset tables */
  ffree(socket_fp);
  fdp->fd_ofiles[sock_fdesc] = NULL;

 release_file_and_vnode:
  /* Close vnode */
  vn_close(vnode, file_fp->f_flag, file_fp->f_cred, p);

 release_file:
  /* Release file and reset tables */
  ffree(file_fp);
  fdp->fd_ofiles[file_fdesc] = NULL;

  ALLOC_DBG(("Error in instance_alloc\n"));

  *retval = error;

  return (error);

};
```

B.1.2 sys_instance_free

```
/**************************/
/**************************/
/******** SYS_FREE *********/
/**************************/
/**************************/

/* Free an instance stream including both */
/* instance buffers, socket and vnode     */

/*************************************************************************/
/*                        PARAMETERS                                     */
/*                                                                       */
/* IN:  i_stream = instance stream wich shoud be freed                   */
/*      flags    = How free should operate                               */
/*                 Possible values:                                      */
/*                    INSTANCE_NO_WAIT: Free the buffers no matter what      */
/* OUT: zero if successfull                                              */
/*                                                                       */
/*                                                                       */
/*                           RULES                                       */
/*                                                                       */
/* i_stream: Must point to an instance_stream owned by the calling       */
/*             process.                                                  */
/* flags:    There is a serious risk the kernel will crash if this flag  */
/*             is set. Haven't yet decided how to handle that...         */
/*                                                                       */
/*************************************************************************/


/* ARGSUSED */
int
sys_instance_free(p, v, retval)
     struct proc *p;
     void *v;
     register_t *retval;
{
     struct sys_instance_free_args /* {
         syscallarg(struct instance_stream *) i_stream;
         syscallarg(short) flags;
     } */ *uap = v;


/*    void *buffer; */
  struct instance_stream *i_stream;
  struct instance_buffer *i_buf;
  int error;
  int return_error = 0;
  int i;
  short flags;



  /*********************/
  /* RETRIVE ARGUMENTS */
  /*********************/

  i_stream = SCARG(uap, i_stream);
  flags = SCARG(uap, flags);


  /*******************/
  /* CHECK ARGUMENTS */
```

```
/******************/


if (i_stream == NULL)
  {
    /* OBS! No check possible to check if pointer really points to an */
    /* INSTANCE buffer yet possible */
    /* ERROR */
    FREE_DBG(("Error no stream\n"));
    return (EINSTANCE_NO_STREAM);
  };


/* SECURITY */
if (i_stream->p != p)
  {
    /* ERROR */
    FREE_DBG(("Error not owner\n"));
    return (EINSTANCE_NOT_OWNER);
  };


i_buf = i_stream->first_ibuf;


for (i = 0; i < 2; i++)
  {
    struct instance_buffer *next_ibuf = NULL;
    FREE_DBG(("i: %d\n", i));


    if (i == 0) {
      next_ibuf = i_buf->next;
    };


    /* CHECK THAT THE BUFFER IS FINISHED SENDING OR */
    /* READING WAITING TO SEND */

    if (!(flags & INSTANCE_NO_WAIT))
      {
        while ((i_buf->io_pending || i_buf->io_being_called)
               && !(i_buf->error & INSTANCE_FATAL_ERROR))

          {
            i_buf->error
              = i_buf->error | INSTANCE_CLOSING;
            FREE_DBG(("Have to wait before freeing...\n"));


            /* If I/O Operations is still pending in one of the buffers */
            /* and the buffer has no fatal error we wait for the operation */
            /* to finish */

            tsleep ((void *) i_stream, PRIBIO
                           , "close_waiting", 50);
          };
      }
    else
      {
        /* Warn possible sending processes that something is going on! */
        i_buf->error
```

```
            = i_buf->error | INSTANCE_FATAL_ERROR
            | INSTANCE_FORCED_CLOSING;
          FREE_DBG(("Setting INSTANCE_FORCED_CLOSING.\n"));
        };


      /* RELEASE BUFFER AREA */
      FREE(i_buf->data, M_INSTANCE);
      FREE_DBG2(("ibuf data_area freed\n"));

      /* RELEASE INSTANCE BUFFERS */
      FREE(i_buf, M_INSTANCE);
      FREE_DBG2(("ibuf freed\n");)

      i_buf = next_ibuf;
    };


  /* CLOSE SOCKET */
  error = soo_close(i_stream->socket_fp, i_stream->p);
  if (error)
    {
      FREE_DBG(("Error when closing socket\n"));
    };


  /* RELEASE SOCKET */
  ffree(i_stream->socket_fp);
  FREE_DBG2(("freed socket fp\n"));


  /* CLOSE VNODE */
  error = vn_close(i_stream->vnode, i_stream->file_fp->f_flag,
           i_stream->file_fp->f_cred, i_stream->p);
  if (error)
    {
      FREE_DBG(("Error when closing vnode\n"));
    };


  /* RELEASE FILE */
  ffree(i_stream->file_fp);
  FREE_DBG2(("Freed file fp\n"));


  /* CLEAR THE PROCESS'S FILE DESCRIPTOR TABLE */
  i_stream->p->p_fd->fd_ofiles[i_stream->socket_fdesc] = NULL;
  i_stream->p->p_fd->fd_ofiles[i_stream->file_fdesc] = NULL;
  FREE_DBG2(("File descriptor table cleared\n"));


  /* FREE THE INSTANCE STREAM */
  FREE(i_stream, M_INSTANCE);
  FREE_DBG2(("istream freed\n"));

  return (return_error);
};
```

B.1.3 sys_instance_read

```
/**************************/
/**************************/
/******** SYS_READ ********/
/**************************/
/**************************/


/* Read the required data if not already read. If some of the data */
/* requested is not read, read a whole instance buffer. If requested */
/* data is read already by a previous read call, do nothing */

/*****************************************************************************/
/*                        PARAMETERS                                         */
/*                                                                           */
/* IN:  i_stream = instance stream wich shoud be read                        */
/*      offset   = How far we will read                                      */
/* OUT: zero if successfull                                                  */
/*                                                                           */
/*                                                                           */
/*                           RULES                                           */
/*                                                                           */
/* i_stream: Must point to an instance_stream owned by the calling           */
/*             process.                                                      */
/* offset:   Must be > 0.                                                    */
/*           Must be greater than last read offset                           */
/*           The required read size (this offset - last offset) must not     */
/*             be bigger than the buffer size                                */
/*                                                                           */
/*****************************************************************************/

/* ARGSUSED */
int
sys_instance_read(p, v, retval)
     struct proc *p;
     void *v;
     register_t *retval;
{
     struct sys_instance_read_args /* {
          syscallarg(struct instance_stream *) i_stream;
          syscallarg(off_t) offset;
     } */ *uap = v;


  struct instance_stream *i_stream;
  struct instance_buffer *read_ibuf;
  off_t arg_offset;
  off_t file_offset;
  off_t end;
  size_t length;
  int num_blocks;
  int error;
  int x;
  struct inode *inode;
  struct vnode *vnode;
  struct buf *buf;
  long start_lblk_no;
  long end_lblk_no;
  long logic_block;
```

```
/*********************/
/* RETRIVE ARGUMENTS */
/*********************/

i_stream = SCARG(uap, i_stream);
arg_offset = SCARG(uap, offset);



/*******************/
/* CHECK ARGUMENTS */
/*******************/

if (i_stream == NULL)
  {
    /* OBS! No check possible to check if pointer really points to an */
    /* INSTANCE buffer yet possible */
    /* ERROR */
    return (0);
  };


/************/
/* SECURITY */
/************/

if (i_stream->p != p)
  {
    /* ERROR */
    return (0);
  };


if (istream->read_ibuf->error & INSTANCE_FATAL_ERROR)
  {
    /* ERROR */
    return (0);
  };


if ((arg_offset - i_stream->prev_read_arg_offset) > i_stream->buf_size)
  {
    /* ERROR */
    /* REQUESTING A READ LARGER THAN BUFFERSIZE */
    /* BUFFERSIZE MUST BE >= LARGEST READ */
    return (0);
  };

/* For use i next read call: */
i_stream->prev_read_arg_offset = arg_offset;


/* If offset already read, do nothing */
file_offset = i_stream->file_fp->f_offset;

if (arg_offset < file_offset)
  {
    /* We have already read what is requested. Do nothing */
    return (0);
  }
```

```
/* WE NOW KNOW THAT WE WILL READ AN INSTANCE BUFFER */
/* CHECK THAT THE BUFFER IS READY FOR READING BEFORE WE START */
read_ibuf = i_stream->read_ibuf;

x = splimp(); /* Block interrupts while handeling buffer state*/
switch (read_ibuf->state)
  {

  case INSTANCE_SENDING:
    {
      /* Still reading, will sleep until done */
      read_ibuf->state = INSTANCE_READ_WAITING;
      splx(x);   /* Release lock */
      while (read_ibuf->state == INSTANCE_READ_WAITING)
        {
          tsleep(read_ibuf, PRIBIO | PCATCH, "read_waiting", 10);
        };
      /* Continue reading */
      /* Fall through */
    }

  case INSTANCE_READ_OK:
    {
      read_ibuf->state = INSTANCE_READING;
      splx(x);
      break;
    }

  default:
    {
      /* All other states */
      /*ERROR*/
      read_ibuf->error = read_ibuf->error | INSTANCE_FATAL_ERROR
        | INSTANCE_READ_ERROR;
      splx(x);   /* Release lock */
      return (0);
    }
  };

/* EVERYTHING IS OK. WE WILL READ! */

/* Data needed when sending: */
read_ibuf->start_of_data_area = file_offset;


vnode = read_ibuf->vnode;
inode = VTOI(vnode);
buf = read_ibuf->first_buf;


if ( file_offset + i_stream->bufsize > inode->i_ffs_size) {
  /* We have reached EOF */
  /* We will still read, but must handle this */
  end = inode->i_ffs_size - 1;
  i_stream->end_of_file = end;
}
else {
  end = file_offset + i_stream->bufsize - 1;
};

start_lblk_no = lblkno (i_stream->file_system, file_offset);
```

```
  end_lblk_no = lblkno (i_stream->file_system, end);

  /* Just to make sure that the i_buf changes state while we are */
  /* requesting reads. This _could_ happen if all requested reads */
  /* finish while there are still requests not sent. This is not */
  /* very likely, but still... */
  i_buf->io_pending++;

  for (logic_block = start_lblk_no; logic_block <=  end_lblk_no;
       lblock++, buf++)
    {
      i_buf->io_pending++;

      buf->b_lblkno = logic_block;

      /* Logical to physical block mapping */
      VOP_BMAP(vnode, logic_block, NULL, &buf->b_blkno, NULL);

      /* Call disk driver */
      /* When finished, instance_io_done is called */
      read_ibuf->io_pending++;
      buf->b_flags = (buf->b_flags | B_BUSY | B_CALL | B_READ | B_ASYNC | \
                      B_MMBUF) & ~B_DONE;
      VOP_STRATEGY(buf);
    };

  /* Finished requesting reads: */
  i_buf->io_pending--;



  /* CHANGE VALUES AFTER READ */

  file_fp->f_offset += i_stream->bufsize;
  i_stream->read_ibuf = i_stream->read_ibuf->next;


  return (0);
};
```

B.1.4 sys_instance_send

```
/**************************/
/**************************/
/******** SYS_SEND *********/
/**************************/
/**************************/

/* Send data from instance stream. Starts at first mbuf after last send */
/* and stops at mbuf containg offset */

/*****************************************************************************/
/*                        PARAMETERS                                         */
/*                                                                           */
/* IN:  i_stream = instance stream wich shoud be sent                        */
/*      offset   = How far we will send                                      */
/* OUT: zero if successfull                                                  */
/*                                                                           */
```

```
/*                                                                      */
/*                            RULES                                     */
/*                                                                      */
/* i_stream: Must point to an instance_stream owned by the calling      */
/*             process.                                                 */
/* offset:   Must be > 0.                                               */
/*           Must be greater than last sent offset                      */
/*           The data to be sent must have been read, or a call to read */
/*             the data must have been sent                             */
/*           The required read size (this offset - last offset) must not*/
/*             be bigger than the buffer size                           */
/*           The data to be sent can be devided over two instance buffers.*/
/*             This requires special handeling.                         */
/*                                                                      */
/************************************************************************/


/* ARGSUSED */
int
sys_instance_send(p, v, retval)
     struct proc *p;
     void *v;
     register_t *retval;
{
     struct sys_instance_send_args /* {
            syscallarg(struct instance_stream *) i_stream;
            syscallarg(off_t) offset;
     } */ *uap = v;

  struct instance_stream *i_stream;
  struct instance_buffer *send_ibuf;
  off_t arg_offset;
  int error;
  long length;
  off_t end_of_file = 0;



  /*********************/
  /* Retrive arguments */
  /*********************/

  i_stream = SCARG(uap, i_stream);
  arg_offset = SCARG(uap, offset);



  /*******************/
  /* Fault handeling */
  /*******************/

  if (i_stream == NULL) {
    /* OBS! No check possible to check if pointer really points to an */
    /* INSTANCE buffer yet possible */
    /* ERROR */
    SEND_DBG(("Error no stream\n"));
    return (EINSTANCE_NO_STREAM);
  };



  /************/
```

```
/* SECURITY */
/************/

if (i_stream->p != p) {

  /* ERROR */
  SEND_DBG(("Error not owner\n"));
  return (EINSTANCE_NOT_OWNER);
};


send_ibuf = i_stream->send_ibuf;



/*****************/
/* Check for EOF */
/*****************/

if (i_stream->end_of_file &&
    ((u_int)arg_offset) > ((u_int)i_stream->end_of_file))
  {
    SEND_DBG2(("EOF reached\n"));
    end_of_file = i_stream->end_of_file;
    arg_offset = end_of_file;
  };


length = arg_offset - i_stream->prev_send_arg_offset;


if (length > i_stream->buf_size)
  {
    /* ERROR */
    /* REQUESTING A SEND LARGER THAN BUFFERSIZE */
    /* BUFFERSIZE MUST BE >= LARGEST READ */
    SEND_DBG(("Error arg oversize!\n"));
    return (EINSTANCE_ARG_OVERSIZE);
  };

if (length < 0)
  {
    /* ERROR */
    /* REQUESTING TO SEND A NEGATIVE AMOUNT */
    SEND_DBG(("Error arg undersize!\n"));
    return (EINSTANCE_ARG_UNDERSIZE);
  };


/* For use i next send call: */
i_stream->prev_send_arg_offset = arg_offset;


error = instance_buffer_send(send_ibuf, arg_offset);

if (end_of_file)
  {
    SEND_DBG2(("EOF REACHED ONCE MORE\n"));
  };

SEND_DBG(("Finished sending\n"));
return (error);
```

```
};
```

B.1.5 instance_buffer_send

```
/***********************/
/***********************/
/******** SEND *********/
/***********************/
/***********************/

/* Called from sys_instance_send. Send watherever necessary from a single */
/* instance buffer. If more to send, call itself recursivley with next */
/* instance buffer and how much more to send as parameters */

/*****************************************************************************/
/*                          PARAMETERS                                       */
/*                                                                           */
/* IN:  i_buf = intance buffer from which to send                            */
/*      offset   = How far we will aend                                      */
/* OUT: zero if successfull                                                  */
/*                                                                           */
/*                                                                           */
/*                             RULES                                         */
/* i_buf : Must point two a instance_buffer. Since this procedure is only */
/*           called by sys_instamce_read, it is safe not to check this.   */
/* offset: Note that this offset _can_ point outside the end of the       */
/*           instance buffer. In this case the rest of the buffer is sent */
/*           and a recursive call to this procedure is sent with a        */
/*           pointer to the nex instance buffer.                          */
/*                                                                           */
/*****************************************************************************/


int
instance_buffer_send(i_buf, arg_offset)
     struct instance_buffer *i_buf;
     off_t arg_offset;
{
  off_t offset;
  struct instance_stream *i_stream;
  struct mbuf *curr_mbuf;            /* Always points two the header mbuf */
                                     /* of the chain to send next. */
  struct mbuf *last_mbuf;
  struct socket *socket;             /* Socket where we send */
  int i;
  int x;
  int s;
  int more_sending = 0;
  int ibuf_done = 0;
  int error = 0;
  off_t end_of_file;

  SEND_DBG(("Internal send called, ibuf: 0x%x\n", (u_int)i_buf));

  i_stream = i_buf->i_stream;

  /* Offset into i_buf */
```

```
offset = arg_offset - i_buf->start_of_data_area;

curr_mbuf = i_buf->curr_mbuf;


if (i_stream->mbuf_data_size == 0) {
  return (EINSTANCE_DIV_ZERO);
}

i = ((u_int)offset + 1) / i_stream->mbuf_data_size;
SEND_DBG2(("Send until mbuf no: %u\n", i + 1));
i *= 2;

last_mbuf = (i_buf->first_mbuf + i);
SEND_DBG2(("Last mbuf: 0x%x\n", (u_int)last_mbuf));

if (last_mbuf >= i_buf->last_mbuf)
  {
    /* We have finished sending this ibuf */
    ibuf_done = 1;
    last_mbuf = i_buf->last_mbuf;
    SEND_DBG2(("Last mbuf: 0x%x\n", (u_int)last_mbuf));
    SEND_DBG2(("We send the last of this ibuf: 0x%x\n", (u_int)i_buf));


    /* Next time we will send from the next ibuf */
    SEND_DBG(("send_ibuf = NEXT\n"));
    i_stream->send_ibuf = i_buf->next;


    /* Clear up i_buf for next send */
    i_buf->curr_mbuf = i_buf->first_mbuf;

    if ((u_int) offset + 1 > (u_int) i_stream->buf_size)
      {
        SEND_DBG2(("We will send more than this ibuf\n"));
        more_sending = 1;
      };
  };


if (last_mbuf < curr_mbuf)
  {
    /* We have already sent what we want to send */
    /* do nothing */
    SEND_DBG(("Do nothing\n"));
    return (0);
  };

if (i_stream->end_of_file &&
    ((i_stream->end_of_file - i_buf->start_of_data_area)
     < i_stream->buf_size))
  {
    size_t last_size;

    SEND_DBG2(("EOF REACHED ONCE MORE2\n"));
    /* Handle EOF */
    SEND_DBG2(("i_stream: 0x%x\n", (u_int)i_stream));
    end_of_file = i_stream->end_of_file + 1;
    if ((last_size = (end_of_file % i_stream->mbuf_data_size)))
      {
        struct mbuf *mbuf;
```

```
        /* The size of the last mbuf should be less than */
        /* mbuf_data_size */
        mbuf = last_mbuf;
        SEND_DBG2(("last_mbuf: 0x%x, last_size: %d\n", (u_int)mbuf, last_size));
        mbuf->m_pkthdr.len = last_size;
        mbuf++;
        mbuf->m_len = last_size;
      };
  };


/* EVERYTHING IS NOW READY FOR SEND */
/* CHECK THAT BUFFER IS ALREADY FILLED AND READY FOR SENDING. */
/* IF NOT, WAIT FOR RESD TO FINISH */


SEND_DBG(("Block...\n"));
x = splimp(); /* Block interrupts while handeling buffer state*/

/* Handle buffer state */
switch (i_buf->state)
  {

  case INSTANCE_READING:
  case INSTANCE_SENDING_NOT_LAST:
    {
      /* Still sending, will sleep until done */
      i_buf->state = INSTANCE_SEND_WAITING;
      splx(x);   /* Release lock */
      SEND_DBG2(("ibuf: 0x%x Must sleep\n", (u_int)i_buf));
      while (i_buf->state == INSTANCE_SEND_WAITING)
        {
          tsleep((void *)i_buf, PRIBIO , "send_waiting", 20);
        };

      i_buf->state = INSTANCE_SEND_OK;

      /* ibuf could have changed when reading! */
      /* Do recursive call to get all values correct! */
      SEND_DBG(("Do recursive call on ibuf 0x%x after sleeping", (u_int)i_buf));
      error = instance_buffer_send(i_buf, arg_offset);
      SEND_DBG(("Return from recursive call on ibuf 0x%x\n", (u_int)i_buf));
      return error;
    }

  case INSTANCE_SEND_OK:
    {
      if (ibuf_done)
        {
          SEND_DBG2(("i_buf: 0x%x Changing state to SENDING\n", (u_int)i_buf));
          i_buf->state = INSTANCE_SENDING;
        }
      else
        {
          SEND_DBG2(("ibuf: 0x%x Changing state to SENDING_NOT_LAST\n", (u_int)i_buf));
          i_buf->state = INSTANCE_SENDING_NOT_LAST;
        };
      splx(x);
      break;
    }

  default:
```

```
    {
      /* All other states */
      /*ERROR*/
      splx(x);   /* Release lock */

      SEND_DBG(("i_buf: 0x%x ERROR! Wrong state: 0x%x\n",(u_int)i_buf, (u_int) i_buf->state));
      return (EINSTANCE_IBUF_STATE);
    }
  };

SEND_DBG(("Start actual sending\n"));

/************************/
/* DO THE ACTUAL SENDING */
/************************/

/* curr_mbuf points to the first header mbuf to send. last_mbuf to the */
/* last header mbuf to send in this instance buffer. There may be more */
/* mbuf to send in the next instance buffer, but this is handled in a */
/* new call to this function with curr_mbuf pointing to the first mbuf */
/* in the next instance buffer, and last_mbuf mbuf to the last mbuf to */
/* send. */
socket = i_stream->socket;
SEND_DBG2(("socket: 0x%x\n", (u_int)socket));

/* Just to make sure that the i_buf changes state while we are */
/* requesting sends. This _could_ happen if all requested sends */
/* finish while there are still requests not sent. This is not */
/* very likely, but still... */
i_buf->io_being_called = 1;

SEND_DBG2(("BEFORE: "));
PRINT_MBUF_H((i_buf->first_mbuf));
PRINT_MBUF_E((i_buf->first_mbuf->m_next));

for (;curr_mbuf <= last_mbuf; curr_mbuf += 2)
  {
    struct mbuf *mbuf;

    mbuf = curr_mbuf;

    mbuf++;



    curr_mbuf->m_data = curr_mbuf->m_pktdat;
    curr_mbuf->m_len = 0;
    if ((curr_mbuf == i_buf->last_mbuf) && (i_stream->last_mbuf_size))
      {
        curr_mbuf->m_pkthdr.len = i_stream->last_mbuf_size;
      }
    else
      {
        curr_mbuf->m_pkthdr.len = i_stream->mbuf_data_size;
      };


    i_buf->io_pending++;

    s = splsoftnet();                       /* Like "sendit" */

    error = (*(socket->so_proto->pr_usrreq))
```

```
        (socket, PRU_SEND, curr_mbuf, NULL ,NULL ,i_stream->p);

        splx(s);

    };


  /* Finished requesting sends: */

  x = splimp(); /* Block interrupts */
  i_buf->io_being_called = 0;

  if (i_buf->io_pending == 0)
    {
      INSTANCE_IO_DONE(i_buf, x);
    }
  else
    {
      splx(x);
    };

  SEND_DBG2(("AFTER: "));
  PRINT_MBUF_H((i_buf->first_mbuf));
  PRINT_MBUF_E((i_buf->first_mbuf->m_next));



  /* Setting values before next sendcall */
  if (more_sending)
    {

      SEND_DBG2(("Recursive call\n"));
      error = instance_buffer_send(i_stream->send_ibuf, arg_offset);

      SEND_DBG(("Last sending\n"));

    }
  else if (!ibuf_done)
    {
      /* We have already sent curr_mbuf, */
      /* so we will not send the same next time */
      i_buf->curr_mbuf = curr_mbuf;
    };


  return (error);
};
```

B.1.6 instance_read_done

```
/************************/
/************************/
/******** IO_DONE ********/
/************************/
/************************/


/************************/
/*        READ_DONE      */
```

```
/************************/

void
instance_read_done (buf)
     struct buf *buf;
{
  struct instance_buffer *i_buf;
  int x;

  i_buf = (struct instance_buffer *) buf->iodone_args;
  IODONE_DBG(("read done called, i_buf:%x!\n", (u_int)i_buf));
  IODONE_DBG(("read done called, io_pending:%d!\n", i_buf->io_pending));

  i_buf->io_pending--;
  IODONE_DBG(("after --, io_pending:%d!\n", i_buf->io_pending));


  x = splimp(); /* block */

  if (i_buf->io_pending == 0 && i_buf->io_being_called == 0)
    {
      INSTANCE_IO_DONE(i_buf, x);
    };

  IODONE_DBG(("(read)makro done, io_pending:%d!\n", i_buf->io_pending));

  return;
};
```

B.1.7 instance_send_done

```
/************************/
/*        SEND_DONE      */
/************************/


void
instance_send_done (ext_buf, ext_size, arg_i_buf)
     caddr_t ext_buf;
     u_int ext_size;
     void *arg_i_buf;
{
  struct instance_buffer *i_buf;
  int x;

  IODONE_DBG(("send done called!\n"));


  i_buf = (struct instance_buffer *) arg_i_buf;

  IODONE_DBG(("send done called, io_pending:%d!\n", i_buf->io_pending));

  i_buf->io_pending--;
  IODONE_DBG(("after --, io_pending:%d!\n", i_buf->io_pending));

  x = splimp(); /* block */

  if (i_buf->io_pending == 0 && i_buf->io_being_called == 0)
```

```
    {
      INSTANCE_IO_DONE(i_buf, x);
    }
  else
    {
      splx(x);
    };

  IODONE_DBG(("(send)makro done, io_pending:%d!\n", i_buf->io_pending));

  return;
};

#endif /* ifdef INSTANCE */
```

B.2 instance_syscalls.h

```
/*****************************************************************************/
/* instance_syscalls.h                                                       */
/*                                                                           */
/* (C) 2000 Espen Jorde                                                      */
/*                                                                           */
/* This is free software; you can redistribute it and/or                     */
/* modify it under the terms of the GNU Library General Public License as    */
/* published by the Free Software Foundation; either version 2 of the        */
/* License, or (at your option) any later version.                           */
/*                                                                           */
/* This file is distributed in the hope that it will be useful,              */
/* but WITHOUT ANY WARRANTY; without even the implied warranty of            */
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU         */
/* Library General Public License for more details.                          */
/*                                                                           */
/* You should have received a copy of the GNU Library General Public         */
/* License along with the GNU C Library; see the file COPYING.LIB.  If not, */
/* write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,*/
/*   Boston, MA 02111-1307, USA.                                             */
/*                                                                           */
/*****************************************************************************/



#ifdef INSTANCE
#ifndef INSTANCE_SYSCALL_H
#define INSTANCE_SYSCALL_H
\end{verbatim
\end{scriptsize}

\subsection{Include files}
\begin{scriptsize}
\begin{verbatim}
/*********************/
/*    INCLUDES       */
/*********************/

#ifdef _KERNEL

#include "opt_compat_netbsd.h"
```

```
#include "opt_compat_43.h"
#include <sys/types.h>
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/dkstat.h>
#include <sys/callout.h>
#include <sys/kernel.h>
#include <sys/proc.h>
#include <sys/resourcevar.h>
#include <sys/signalvar.h>
#include <vm/vm.h>
#include <sys/sysctl.h>
#include <sys/timex.h>
#include <sys/sched.h>
#include <sys/dirent.h>
#include <sys/file.h>
#include <sys/filedesc.h>
#include <sys/kernel.h>
#include <sys/malloc.h>
#include <sys/mbuf.h>
#include <sys/mman.h>
#include <sys/mount.h>
#include <sys/namei.h>
#include <sys/param.h>
#include <sys/protosw.h>
#include <sys/proc.h>
#include <sys/queue.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/stat.h>
#include <sys/syscallargs.h>
#include <sys/sysctl.h>
#include <sys/systm.h>
#include <sys/uio.h>
#include <sys/vnode.h>
#include <ufs/ufs/quota.h>
#include <ufs/ufs/inode.h>
#include <ufs/ffs/fs.h>
#include <uvm/uvm_extern.h>
#include <vm/vm.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/in_systm.h>
#include <netinet/ip.h>
#include <netinet/ip_var.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <netinet/tcpip.h>
#include <netinet/in_pcb.h>
#include <netinet/udp_var.h>
#include <sys/syslog.h>
#include <vm/vm_kern.h>
```

B.2.1 Defenitions

```
#ifndef INSTANCE_BLOCK_SIZE
#define INSTANCE_BLOCK_SIZE 65536
#endif
```

```
#ifndef INSTANCE_MTU
#define INSTANCE_MTU 1500
#endif


\subsection{Macros}
\begin{scriptsize}
\begin{verbatim}
/****************/
/****************/
/**** MACROS ****/
/****************/
/****************/

struct instamce_stream ;
struct instance_buffer ;

/************************/
/*** INSTANCE_IO_DONE ***/
/************************/

#define INSTANCE_IO_DONE(i_buf, x)                                      \
                                                                        \
    switch ((i_buf)->state)                                             \
      {                                                                 \
                                                                        \
      case INSTANCE_READING:                                            \
      case INSTANCE_SENDING_NOT_LAST:                                   \
        {                                                               \
          (i_buf)->state = INSTANCE_SEND_OK;                            \
          splx(x);                                                      \
          break;                                                        \
        }                                                               \
                                                                        \
      case INSTANCE_SENDING:                                            \
        {                                                               \
          (i_buf)->state = INSTANCE_READ_OK;                            \
          splx(x);                                                      \
          break;                                                        \
        }                                                               \
                                                                        \
      case INSTANCE_READ_WAITING:                                       \
        {                                                               \
          i_buf->state = INSTANCE_READ_OK;                              \
          splx(x);                                                      \
          wakeup (((void *)i_buf));                                     \
          break;                                                        \
        }                                                               \
                                                                        \
      case INSTANCE_SEND_WAITING:                                       \
        {                                                               \
          i_buf->state = INSTANCE_SEND_OK;                              \
          splx(x);                                                      \
          wakeup (((void *)i_buf));                                     \
          break;                                                        \
        }                                                               \
                                                                        \
      default:                                                          \
        {                                                               \
          /* All other states */                                        \
          /*ERROR*/                                                     \
          (i_buf)->error = ((i_buf)->error | INSTANCE_FATAL_ERROR);     \
```

```
          splx(x);                                                    \
          break;                                                      \
        }                                                             \
      };                                                              \
    x = splimp();                                                     \
    if ((i_buf)->error & INSTANCE_CLOSING)                            \
      {                                                               \
        splx(x);                                                      \
            wakeup ((void *)(i_buf)->i_stream);                       \
      }                                                               \
    else                                                              \
      {                                                               \
        splx(x);                                                      \
      };
```

B.2.2 Other definirions

```
/***************************/
/****** DEBUG MESSAGES ******/
/***************************/
#define _STREAM_DBG 0
#define _ALLOC_DBG 0
#define _ALLOC_DBG2 0
#define _FREE_DBG 0
#define _FREE_DBG2 0
#define _READ_DBG 0
#define _READ_DBG2 0
#define _SEND_DBG 0
#define _SEND_DBG2 0
#define _IODONE_DBG 0
#define _IODONE_DBG2 0
#define _PRINT_MBUF_H 0
#define _PRINT_MBUF_E 0
#define _PRINT_BUF 0

#if _STREAM_DBG == 1
#define STREAM_DBG(text)                        \
{                                               \
  printf("instance: ");                         \
  printf text;                                  \
}
#else
#define STREAM_DBG(text)
#endif


#if _ALLOC_DBG == 1
#define ALLOC_DBG(text)                       \
{                                             \
  printf("i_alloc: ");                        \
  printf text;                                \
}
#else
#define ALLOC_DBG(text)
#endif


#if _ALLOC_DBG2 == 1
```

```
#define ALLOC_DBG2(text)                        \
{                                               \
  printf("i_alloc: ");                          \
  printf text;                                  \
}
#else
#define ALLOC_DBG2(text)
#endif

#if _FREE_DBG == 1
#define FREE_DBG(text)                         \
{                                              \
  printf("i_free: ");                          \
  printf text;                                 \
}
#else
#define FREE_DBG(text)
#endif


#if _FREE_DBG2 == 1
#define FREE_DBG2(text)                         \
{                                               \
  printf("i_free: ");                           \
  printf text;                                  \
}
#else
#define FREE_DBG2(text)
#endif

#if _READ_DBG == 1
#define READ_DBG(text)                          \
{                                               \
  printf("i_read: ");                           \
  printf text;                                  \
}
#else
#define READ_DBG(text)
#endif


#if _READ_DBG2 == 1
#define READ_DBG2(text)                         \
{                                               \
  printf("i_read: ");                           \
  printf text;                                  \
}
#else
#define READ_DBG2(text)
#endif


#if _SEND_DBG == 1
#define SEND_DBG(text)                          \
{                                               \
  printf("i_send: ");                           \
  printf text;                                  \
}
#else
#define SEND_DBG(text)
#endif
```

```
#if _SEND_DBG2 == 1
#define SEND_DBG2(text)                          \
{                                                \
  printf("i_send: ");                            \
  printf text;                                   \
}
#else
#define SEND_DBG2(text)
#endif


#if _IODONE_DBG == 1
#define IODONE_DBG(text)                         \
{                                                \
  printf("io_done: ");                           \
  printf text;                                   \
}
#else
#define IODONE_DBG(text)
#endif


#if _IOSONE_DBG2 == 1
#define IODONE_DBG2(text)                         \
{                                                 \
  printf("io_done: ");                            \
  printf text;                                    \
}
#else
#define IODONE_DBG2(text)
#endif



#if _PRINT_MBUF_H == 1
#define PRINT_MBUF_H(m)                                                  \
{                                                                        \
  printf("mbuf: 0x%x\n", (u_int) m);                                     \
  printf("mbuf.m_next: 0x%x\n", (u_int)m->m_next);                       \
  printf("mbuf.m_nektpkt: 0x%x\n", (u_int)m->m_nextpkt);                 \
  printf("mbuf.m_data: 0x%x\n", (u_int)m->m_data);                       \
  printf("mbuf.m_len: %d\n", m->m_len);                                  \
  printf("mbuf.m_type: 0%o\n", (u_int)m->m_type);                        \
  printf("mbuf.m_flags: 0x%x\n", (u_int)m->m_flags);                     \
  printf("mbuf.m_pkthdr.rcvif: 0x%x\n", (u_int)m->m_pkthdr.rcvif);       \
  printf("mbuf.m_pkthdr.len: 0x%x\n", (u_int)m->m_pkthdr.len);           \
                                                                         \
  /* make sure the writing is done... */                                 \
  tsleep((void *) m, PRIBIO | PCATCH, "", 50);                           \
                                                                         \
};
#else
#define PRINT_MBUF_H(m)
#endif

#if _PRINT_MBUF_E == 1
#define PRINT_MBUF_E(m)                                                   \
{                                                                         \
  printf("mbuf: 0x%x\n", (u_int) m);                                      \
  printf("mbuf.m_next: 0x%x\n", (u_int)m->m_next);                        \
```

```
  printf("mbuf.m_nektpkt: 0x%x\n", (u_int)m->m_nextpkt);                  \
  printf("mbuf.m_data: 0x%x\n", (u_int)m->m_data);                        \
  printf("mbuf.m_len: %d\n", m->m_len);                                   \
  printf("mbuf.m_type: 0%o\n", (u_int)m->m_type);                         \
  printf("mbuf.m_flags: 0x%x\n", (u_int)m->m_flags);                      \
  printf("mbuf.m_ext.ext_buf: 0x%x\n", (u_int)m->m_ext.ext_buf);          \
  printf("mbuf.m_ext.ext_free: 0x%x\n", (u_int)m->m_ext.ext_free);        \
  printf("mbuf.m_ext.ext_type: %d\n", m->m_ext.ext_type);                 \
  printf("mbuf.m_ext.ext_arg: 0x%x\n", (u_int)m->m_ext.ext_arg);          \
  printf("mbuf.m_ext.ext_prevref: 0x%x\n", (u_int)m->m_ext.ext_prevref);  \
  printf("mbuf.m_ext.ext_nextref: 0x%x\n", (u_int)m->m_ext.ext_nextref);  \
                                                                          \
  /* make sure the writing is done... */                                  \
  tsleep((void *) m, PRIBIO | PCATCH, "", 50);                            \
                                                                          \
};
#else
#define PRINT_MBUF_E(m)
#endif

#if _PRINT_BUF == 1
#define PRINT_BUF(b)                                                      \
{                                                                         \
  printf("buf: 0x%x\n", (u_int) b);                                       \
  printf("buf.b_dev: 0x%x\n", (u_int)b->b_dev);                           \
  printf("buf.b_error: 0x%x\n", (u_int)b->b_error);                       \
  printf("buf.b_resid : 0x%x\n", (u_int)b->b_resid );                     \
  printf("buf.b_dirtyoff: %d\n", (u_int)b->b_dirtyoff);                   \
  printf("buf.b_dirtyend: %d\n", (u_int)b->b_dirtyend);                   \
  printf("buf.b_validoff: %d\n", (u_int)b->b_validoff);                   \
  printf("buf.b_validend: %d\n", (u_int)b->b_validend);                   \
  printf("buf.b_rcred: %d\n", (u_int)b->b_rcred);                         \
  printf("buf.b_wcred: %d\n", (u_int)b->b_wcred);                         \
  printf("buf.b_proc: %d\n", (u_int)b->b_proc);                           \
  printf("buf.b_bcount: %d\n", (u_int)b->b_bcount);                       \
  printf("buf.b_vp: 0x%x\n", (u_int)b->b_vp);                             \
  printf("buf.b_data: 0x%x\n", (u_int)b->b_data);                         \
  printf("buf.b_iodone: 0x%x\n", (u_int)b->b_iodone);                     \
  printf("buf.b_flags: 0x%x\n", (u_int)b->b_flags);                       \
  printf("buf.b_bufsize: 0x%x\n", (u_int)b->b_bufsize);                   \
  printf("buf.iodone_args: 0x%x\n", (u_int)b->iodone_args);               \
                                                                          \
  /* make sure the writing is done... */                                  \
  tsleep((void *) b, PRIBIO | PCATCH, "", 50);                            \
};

#else
#define PRINT_BUF(b)
#endif
```

B.2.3 External structs

```
/************************/
/*   EXTERNAL STRUCTS   */
/************************/


extern struct fileops socketops;  /* Pointer to operations on files */
extern struct fileops vnops;      /* Pointer to operations on files */
```

B.2.4 Function Prototypes

```
/*****************************/
/*****************************/
/**** FUNCTION PROTOTYPES ****/
/*****************************/
/*****************************/

int instance_buffer_send __P((struct instance_buffer *, off_t));
void instance_read_done __P((struct buf *));
void instance_send_done __P((caddr_t, u_int, void *));


#endif /* ifdef _KERNEL */
```

B.2.5 Flags

```
/*************************/
/*          FLAGS        */
/*************************/


/* FLAGS FOR INSTANCE BUFFER STATE */

#define INSTANCE_READ_OK          0x0001
#define INSTANCE_READ_WAITING     0x0002
#define INSTANCE_READING          0x0004
#define INSTANCE_SEND_OK          0x0008
#define INSTANCE_SEND_WAITING     0x0010
#define INSTANCE_SENDING          0x0020
#define INSTANCE_SENDING_NOT_LAST 0x0040



/* ERROR FLAGS */

#define INSTANCE_CLOSING        0x0001
#define INSTANCE_FATAL_ERROR    0x1000
#define INSTANCE_READ_ERROR     0x2000
#define INSTANCE_SEND_ERROR     0x4000
#define INSTANCE_FORCED_CLOSING 0x0002


/* FLAGS USED WHEN CALLING FREE */

#define INSTANCE_NO_WAIT      0x0001


/* RETURN VALUES */
#define EINSTANCE_NOT_OWNER           1
#define EINSTANCE_FREE_NOT_COMPLETE   2
#define EINSTANCE_NO_STREAM           3
#define EINSTANCE_IBUF_FATAL          4
#define EINSTANCE_ARG_OVERSIZE        5
#define EINSTANCE_ARG_UNDERSIZE       6
#define EINSTANCE_WRONG_SOCK_FAMILY   7
#define EINSTANCE_FILE_ALLOC          8
#define EINSTANCE_VNODE_OPEN          9
```

```
#define EINSTANCE_SOCK_ALLOC         10
#define EINSTANCE_SOCK_CREATE        11
#define EINSTANCE_SOCK_ARGS          12
#define EINSTANCE_SOCONNECT          13
#define EINSTANCE_STREAM_ALLOC       14
#define EINSTANCE_BUFFER_ALLOC       15
#define EINSTANCE_IBUF_ALLOC         16
#define EINSTANCE_IBUF_STATE         17
#define EINSTANCE_EOF                18
#define EINSTANCE_NO_FILE            19
#define EINSTANCE_NO_ADDRESS         20
#define EINSTANCE_DIV_ZERO           21

/***********************/
/*  INTERNAL STRUCTS  */
/***********************/

struct instamce_stream ;
```

B.2.6 struct instance_buffer

```
/*************************/
/*************************/
/**** INSTANCE BUFFER ****/
/*************************/
/*************************/


struct instance_buffer {

  /* GENERAL */

  struct instance_stream *i_stream; /* Owner */
  struct instance_buffer *next;     /* Next instance_buffer */
  void *data;                       /* Pointer to the data area */

  short state;                      /* State of buffer */
  short error;                      /* Error flags */
  int io_pending;                   /* Number of I/O operations pending */
  int io_being_called;              /* TRUE if I/O operatrions are beeing */
                                    /* called*/


  /* bufs for READING */

  struct buf *first_buf;            /* Pointer to the first buf */


  /* mbufs for SENDING */

  struct mbuf *first_mbuf;          /* Pointer to the first mbuf */
  struct mbuf *curr_mbuf;           /* Pointer to the next mbuf to send */
  struct mbuf *last_mbuf;           /* Pointer to the last mbuf */

  off_t start_of_data_area;         /* File offset corresponding to */
                                    /* the start of the data area. */
};
```

B.2.7 struct instance_stream

```
/************************/
/************************/
/**** INSTANCE STREAM ****/
/************************/
/************************/


struct instance_stream {

  /* GENERAL */
  struct proc *p;                  /* Owner of the instance_buffer */
  int buf_size;                    /* size of data-area */
  off_t end_of_file;               /* Set to end of file when reached */
                                   /* Else Zero */


  /* INSTANCE BUFFERS */

  struct instance_buffer  *first_ibuf; /* The instance buffers  */
  struct instance_buffer *read_ibuf;  /* Current READ i_buf */
  struct instance_buffer *send_ibuf;  /* Current SEND i_buf */



  /* FILE HANDLING */

  struct file *file_fp;            /* Pointer to file beeing bufferd */
  struct vnode *vnode;             /* Vnode */
  int file_fdesc;                  /* Index in process's descriptor table */
  struct fs *file_system;          /* File system */

  int num_buf;                     /* number of buf structures */
  off_t prev_read_arg_offset;      /* End of previous _requested_ read. */
  int buf_data_size;               /* Amount of data buffered by each buf */



  /* SOCKET HANDLING */

  struct sockaddr_in *addr;        /* Address we are Sending to */
  struct socket *socket;           /* Socket for sending */
  struct file *socket_fp;          /* Socket file pointer */
  int socket_fdesc;                /* Index in process's descriptor table */

  int num_mbuf;                    /* Number of mbufs */
  off_t prev_send_arg_offset;      /* End of previous _requested_ send. */
  int mbuf_data_size;              /* Amount of data buffered by each mbuf */
  int last_mbuf_size;              /* The size of the last mbuf can differ
                                      From the others */
};
```

B.3 Endringer andre steder i kjernens kildekode

B.3.1 `mbuf.h`

I `mbuf.h` er flagget `M_NODELETE` lagt til for å signalisere at `m_freem` ikke skal slette denne `mbuf`-en. Vi inkluderer deler av `mbuf.h`:

```
/* mbuf flags */
#define M_EXT        0x0001 /* has associated external storage */
#define M_PKTHDR     0x0002 /* start of record */
#define M_EOR        0x0004 /* end of record */
#define M_CLUSTER    0x0008 /* external storage is a cluster */

/* INSTANCE INCLUDED FLAGS */
#define M_NODELETE   0x0010 /* This mbuf should not be */
                            /* freed after use */@

/* mbuf pkthdr flags, also in m_flags */
#define M_BCAST       0x0100 /* send/received as link-level broadcast */
#define M_MCAST       0x0200 /* send/received as link-level multicast */
#define M_CANFASTFWD  0x0400 /* used by filters to indicate */
                             /* packet can be fast-forwarded */
#define M_LINK0       0x1000 /* link layer specific flag */
#define M_LINK1       0x2000 /* link layer specific flag */
#define M_LINK2       0x4000 /* link layer specific flag */
```

B.3.2 `buf.h`

I `buf.h` har vi føyd til en `void *` peker for å inneholde et argument til I/O-Done funksjonen vår. Vi inkluderer den delen av `mbuf.h` som viser `buf` strukturen:

```
/*
 * The buffer header describes an I/O operation in the kernel.
 */
struct buf {
        LIST_ENTRY(buf) b_hash;         /* Hash chain. */
        LIST_ENTRY(buf) b_vnbufs;       /* Buffer's associated vnode. */
        TAILQ_ENTRY(buf) b_freelist     /* Free list position if not active. */
        struct  buf *b_actf, **b_actb;  /* Device driver queue when active. */
        struct  proc  *b_proc;          /* Associated proc; NULL if kernel. */
        volatile long b_flags           /* B_* flags. */
        int        b_error;             /* Errno value. */
        long       b_bufsize;           /* Allocated buffer size. */
        long       b_bcount;            /* Valid bytes in buffer. */
        long       b_resid;             /* Remaining I/O. */
```

```
        dev_t      b_dev;              /* Device associated with buffer. */
        struct {
                caddr_t b_addr;  /* Memory, superblocks, indirect etc. */
        } b_un;
        void       *b_saveaddr;        /* Original b_addr for physio. */
        daddr_t    b_lblkno;           /* Logical block number. */
        daddr_t    b_blkno;            /* Underlying physical block number. */
                                       /* Function to call upon completion. */
        void        (*b_iodone) __P((struct buf *));


        void *iodone_args; /* Added by INSTANCE */



        struct     vnode *b_vp;        /* Device vnode. */
        int        b_dirtyoff;         /* Offset in buffer of dirty region. */
        int        b_dirtyend;         /* Offset of end of dirty region. */
        struct     ucred *b_rcred;     /* Read credentials reference. */
        struct     ucred *b_wcred;     /* Write credentials reference. */
        int        b_validoff;         /* Offset in buffer of valid region. */
        int        b_validend;         /* Offset of end of valid region. */
        off_t      b_dcookie;          /* Offset cookie if dir block */
};
```

B.3.3 m_freem

I filen uipc_mbuf.c er prosedyren m_freem endret for å håndtere flagget M_INST_MBUF. mbuf-er med dette flagget skal ikke frigjøres, men iodone skal kalles.

```
void
m_freem(m)
    struct mbuf *m;
{
  struct mbuf *n;

  if (m == NULL)
    return;

  if (m->m_flags & M_NODELETE)
    return;
  ;

  do {
    MFREE(m, n);
    m = n;
  } while (m);
}
```

B.3.4 udp_output

udp_output er en funksjon i filen udp_usrreq.c.

```
udp_output(m, va_alist)
        struct mbuf *m;
        va_dcl
{
        register struct inpcb *inp;
        register struct udpiphdr *ui;
        register int len = m->m_pkthdr.len;
        int error = 0;
        va_list ap;

        va_start(ap, m);
        inp = va_arg(ap, struct inpcb *);
        va_end(ap);

        /*
         * Calculate data length and get a mbuf
         * for UDP and IP headers.
         */



       /* CHANGED BY ESPEN JORDE 2000 */
       /* FOR USE WITH FIXED ALLOCATED MBUF'S */
       if (!(m->m_flags & M_INST_MBUF))
         {
           M_PREPEND(m, sizeof(struct udpiphdr), M_DONTWAIT);
           if (m == 0) {
             error = ENOBUFS;
           goto release;
         }
       }
        else

            int len;

            len = sizeof(struct udpiphdr);
            if (len < MHLEN)
              MH_ALIGN(m, len);
            m->m_len = len;
          ;
        /* END OF INSTANCE RELATED CODE */




        /*
         * Compute the packet length of the IP header, and
         * punt if the length looks bogus.
         */
        if ((len + sizeof(struct udpiphdr)) > IP_MAXPACKET) {
                error = EMSGSIZE;
                goto release;
        }

        /*
         * Fill in mbuf with extended UDP header
         * and addresses and length put into network format.
         */
        ui = mtod(m, struct udpiphdr *);
        bzero(ui->ui_x1, sizeof ui->ui_x1);
```

```
        ui->ui_pr = IPPROTO_UDP;
        ui->ui_len = htons((u_int16_t)len + sizeof (struct udphdr));
        ui->ui_src = inp->inp_laddr;
        ui->ui_dst = inp->inp_faddr;
        ui->ui_sport = inp->inp_lport;
        ui->ui_dport = inp->inp_fport;
        ui->ui_ulen = ui->ui_len;

        /*
         * Stuff checksum and output datagram.
         */
        ui->ui_sum = 0;
        if (udpcksum) {
            if ((ui->ui_sum = in_cksum(m, sizeof (struct udpiphdr) + len)) == 0)
                ui->ui_sum = 0xffff;
        }
        ((struct ip *)ui)->ip_len = sizeof (struct udpiphdr) + len;
        ((struct ip *)ui)->ip_ttl = inp->inp_ip.ip_ttl;        /* XXX */
        ((struct ip *)ui)->ip_tos = inp->inp_ip.ip_tos;        /* XXX */
        udpstat.udps_opackets++;
        return (ip_output(m, inp->inp_options, &inp->inp_route,
            inp->inp_socket->so_options & (SO_DONTROUTE | SO_BROADCAST),
            inp->inp_moptions));

release:
        m_freem(m);
        return (error);
}
```

B.4 Programmer for å teste throughput

Dette er to av programmene brukt til å teste hastigheten til INSTANCE-bufferne. De andre programmene er bygget over samme lest.

B.4.1 `instance_test.c`

```
#include <stdio.h>
#include <sys/instance_syscalls.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <sys/time.h>
#include <errno.h>


int
main(int argc, char *argv[])
{
  struct sockaddr_in *address;
  char *filename;
  struct instance_stream *i_stream;
  int error, end = 0;
  short port = 9876;
```

```
long ip_address;
int i, j;
size_t bufsize;
size_t readsize;
size_t sendsize;

struct timeval tv, tv2;
time_t t;
long m;
float f1, f2, f3;

if(argc!=4)
  {
    printf("Usage: %s serveraddr serverport\n\n", argv[0]);
    exit(1);
  };

bufsize = 1024 * atoi(argv[1]);
readsize = 1024 * atoi(argv[2]);
sendsize = 1024 * atoi(argv[3]);

address = (struct sockaddr_in *) malloc ( sizeof (struct sockaddr_in));
filename = (char *) malloc (64);
ip_address = 0xc0a80001; /* HARALD */
/*  ip_address = 0x7f000001;   LOOPBACK */

bzero(address, sizeof (address));
address->sin_family = AF_INET;
address->sin_port = htons(port);
address->sin_addr.s_addr = htonl(ip_address);

filename = "/mars2/BIGALIVE.mp3";

gettimeofday(&tv, NULL);
error = instance_alloc(filename, address, bufsize, &i_stream);
error = instance_read(i_stream,  readsize -1);

for (i=1; !end ; i++)
  {
    if (error) end = 1;

    error = instance_send(i_stream, (i * sendsize)-1);
    if (!error)
      {
        error = instance_read(i_stream, (((i+1) * readsize)- 1));
      };

  };

error = instance_free(i_stream, 0);

gettimeofday(&tv2, NULL);

t = tv2.tv_sec - tv.tv_sec;
m = tv2.tv_usec - tv.tv_usec;

if (m < 0)
  {
    m = 0 - m;
    t--;
  };
```

```
  f1 = t;
  f2 = m;
  f2 = f2 / 1000000;
  f1 = f1 + f2;

  printf(", %.3f\n", f1);

}
```

B.4.2 tradisjonell_test.c

```
#include <stdio.h>
#include <sys/instance_syscalls.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>

#include <stdlib.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>


int
main(int argc, char *argv[])
{
  struct sockaddr_in *address;
  char *filename;
  struct instance_stream *i_stream;
  int error;
  short port = 9876;
  long ip_address;
  size_t bufsize;
  size_t sendsize;
  int fd, sfd;
  void *buffer;
  size_t  readsize;
  int i, j, c, lastsize;
  int retval;

  u_int64_t end;
  struct timeval tv, tv2;
  time_t t;
  long m;
  float f;
  float f1, f2, f3;

  if(argc!=4)
    {
      printf("Usage: %s serveraddr serverport\n\n", argv[0]);
      exit(1);
```

```
  };

bufsize = 1024 * atoi(argv[1]);
readsize = 1024 * atoi(argv[2]);
sendsize = atoi(argv[3]);

address = (struct sockaddr_in *) malloc ( sizeof (struct sockaddr_in));
filename = (char *) malloc (64);
ip_address = 0xc0a80001; /* harald */

bzero(address, sizeof (address));

address->sin_family = AF_INET;
address->sin_port = htons(port);
address->sin_addr.s_addr = htonl(ip_address);

filename = "/mars2/BIGALIVE.mp3";

gettimeofday(&tv, NULL);

buffer = (void *) malloc (bufsize);

if  (!(fd = open(filename, O_RDONLY, 0)))
  {
    printf("Can't open file\n\n");
    exit(1);
  };

if ((sfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
  printf("Can't create datagram socket\n\n");
  exit(1);
}

if (connect(sfd, (struct sockaddr *) address, sizeof(*address)) < 0)
  {
    printf("ERROR: connect failed on socket %d\n", sfd);
    exit(1);
}

retval = 1;

while (retval > 0)
  {
    retval = read(fd, buffer, readsize);
    if (retval == -1)
        {
          exit(0);
        };

      c = retval / sendsize;
      if ((lastsize = retval % sendsize))
        {
          c++;
        };


    for (i = 0; i < (c-1); i++)
      {
           error = sendto(sfd, buffer + i * sendsize, sendsize, 0, NULL, 0);
        while (error != sendsize)
          {
            error = sendto(sfd, buffer + i * sendsize, sendsize,
```

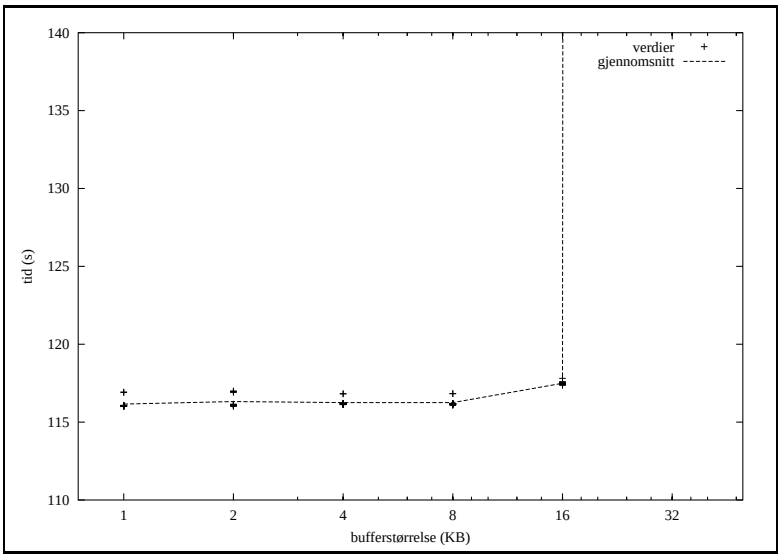
```
                              0, NULL, 0);
            };

        };

      error = sendto(sfd, buffer + i * sendsize, lastsize, 0, NULL, 0);
      while (error != lastsize)
        {
          error = sendto(sfd, buffer + i * sendsize, lastsize,
                         0, NULL, 0);
        };

    };

  close(fd);
  close(sfd);
  free(address);
  free(filename);
  free(buffer);

  gettimeofday(&tv2, NULL);

  t = tv2.tv_sec - tv.tv_sec;
  m = tv2.tv_usec - tv.tv_usec;

  if (m < 0)
    {
      m = 0 - m;
      t--;
    };

  f1 = t;
  f2 = m;
  f2 = f2 / 1000000;
  f1 = f1 + f2;

  printf("%d, %.3f\n", atoi(argv[1]), f1);

}
```

Tillegg C

Målinger

I dette kapitlet har vi samlet flere måleresultater fra tester på implementasjonen vår. Dett er ment som bakgrunnsmateriale for å se detaljene i måleresultatene som blir presentert i kapittel 9

C.1 Tradisjonelle systemkall over 100Mb/s nettverk

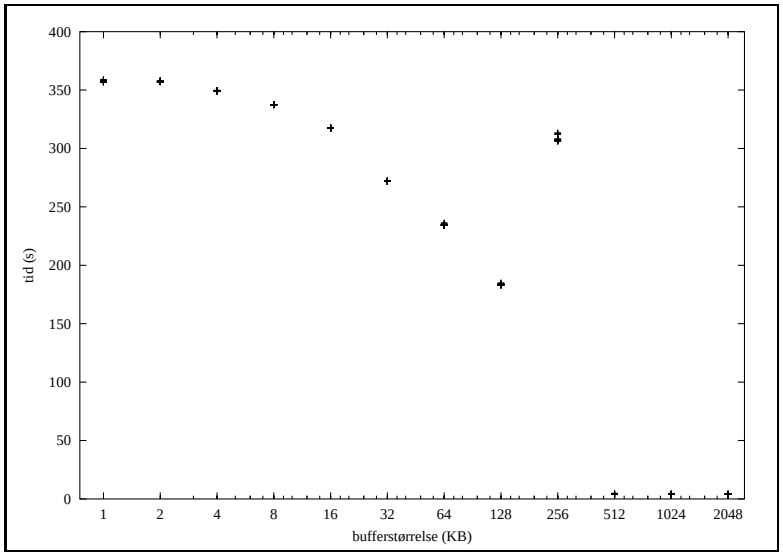


Figur C.1: Tradisjonelle systemkall over 100Mb/s nettverk.

Size (kB)	$\bar{t}$ **(s)**	$\hat{t}$ **(s)**	**min(t)** (s)	**max(t)** (s)	$\bar{d}$ **(Mb/s)**	$\hat{d}$ **(Mb/s)**	**min(d)** (s)	**max(d)** (s)
1	116.16	116.05	116.01	116.94	N/A	N/A	N/A	N/A
2	116.32	116.11	116.02	117.00	N/A	N/A	N/A	N/A
4	116.25	116.18	116.12	116.83	N/A	N/A	N/A	N/A
8	116.25	116.17	116.07	116.83	N/A	N/A	N/A	N/A
16	117.49	117.45	117.36	117.81	N/A	N/A	N/A	N/A
32	7606.40	7605.01	6950.16	8264.05	N/A	N/A	N/A	N/A
64								
128								
256								
512								
1024								
2048								

Tabell C.1: Tradisjonelle systemkall over 100Mb/s nettverk.

C.2 INSTANCE-buffere over 100Mb/s nettverk

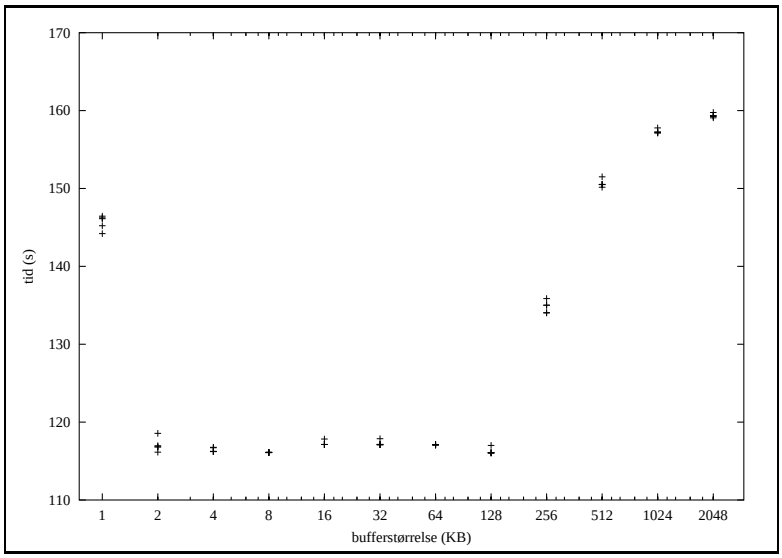


Figur C.2: INSTANCE-buffere over 100Mb/s nettverk.

Size (kB)	$\bar{t}$ **(s)**	$\hat{t}$ **(s)**	**min(t)** (s)	**max(t)** (s)	$\bar{d}$ **(Mb/s)**	$\hat{d}$ **(Mb/s)**	**min(d)** (s)	**max(d)** (s)
1	357,65	357,74	357,01	358,86	48,11	48,10	47,95	48,20
2	357,37	357,25	357,03	358,16	48,15	48,17	48,04	48,19
4	349,42	349,33	349,27	349,68	49,24	49,26	49,21	49,27
8	337,50	337,50	337,49	337,51	50,98	50,98	50,98	50,99
16	317,44	317,34	317,32	317,68	54,21	54,22	54,17	54,23
32	272,02	272,02	272,01	272,02	63,26	63,26	63,26	63,26
64	234,61	234,32	234,18	236,27	73,34	73,43	72,83	73,48
128	183,57	183,25	183,24	185,02	93,74	93,90	93,00	93,90
256	308,38	307,43	306,02	313,36	55,80	55,97	54,91	56,23
512	4,05	4,01	4,01	4,27	N/A	N/A	N/A	N/A
1024	4,00	4,01	4,00	4,01	N/A	N/A	N/A	N/A
2048	4,00	4,01	4,00	4,01	N/A	N/A	N/A	N/A

Tabell C.2: INSTANCE-buffere over 100Mb/s nettverk.

C.3 Tradisjonelle systemkall til loopback adresse

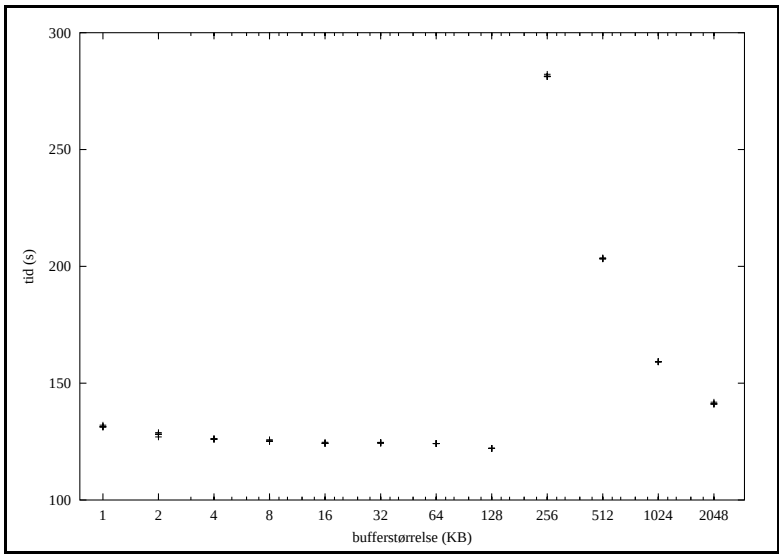


Figur C.3: Tradisjonelle systemkall til loopback adresse.

Size (kB)	$\bar{t}$ **(s)**	$\hat{t}$ **(s)**	**min(t)** (s)	**max(t)** (s)	$\bar{d}$ **(Mb/s)**	$\hat{d}$ **(Mb/s)**	**min(d)** (s)	**max(d)** (s)
1	145,64	146,12	144,20	146,45	118,15	117,76	117,50	119,33
2	117,06	116,82	116,15	118,55	147,00	147,29	145,14	148,15
4	116,44	116,25	116,21	116,76	147,78	148,02	147,37	148,07
8	116,11	116,12	116,08	116,12	148,20	148,19	148,18	148,23
16	117,27	117,13	117,12	117,83	146,73	146,91	146,03	146,91
32	117,26	117,11	117,10	117,88	146,74	146,94	145,98	146,94
64	117,09	117,09	117,05	117,13	146,96	146,95	146,91	147,01
128	116,25	116,07	116,02	117,01	148,02	148,25	147,06	148,32
256	134,79	135,01	134,01	135,86	127,66	127,45	127,65	128,40
512	150,64	150,51	150,17	151,50	114,23	114,33	113,58	114,59
1024	157,31	157,24	157,13	157,78	109,38	109,43	109,06	109,51
2048	159,35	159,32	159,07	159,75	107,98	108,00	107,71	108,17

Tabell C.3: Tradisjonelle systemkall til loopback adresse.

C.4 INSTANCE-buffere til loopback adresse

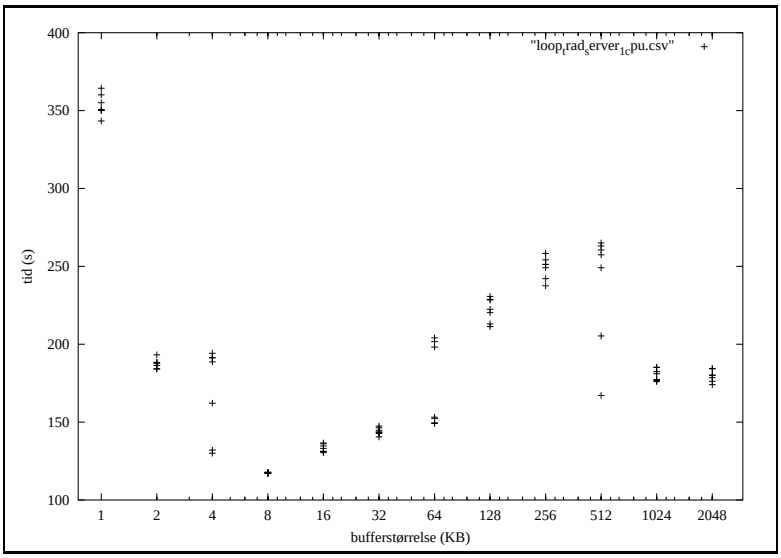


Figur C.4: INSTANCE-buffere til loopback adresse.

Size (kB)	$\bar{t}$ **(s)**	$\hat{t}$ **(s)**	**min(t)** (s)	**max(t)** (s)	$\bar{d}$ **(Mb/s)**	$\hat{d}$ **(Mb/s)**	**min(d)** (s)	**max(d)** (s)
1	131,46	131,30	131,21	132,04	130,90	131,05	130,32	131,14
2	128,06	128,15	127,01	128,86	134,37	134,28	133,53	135,48
4	126,11	126,12	126,00	126,17	136,45	136,43	136,39	136,56
8	125,30	125,18	125,11	125,80	137,32	137,46	136,78	137,53
16	124,35	124,27	124,23	124,70	138,38	138,47	137,99	138,51
32	124,48	124,37	124,35	124,66	138,23	138,35	138,03	138,38
64	124,15	124,15	124,15	124,16	138,60	138,60	138,59	138,60
128	122,11	122,11	122,10	122,12	140,92	140,92	140,91	140,93
256	281,46	281,29	281,16	282,19	61,13	61,17	60,98	61,20
512	203,35	203,29	203,21	203,66	84,62	84,64	84,49	84,68
1024	159,17	159,17	159,15	159,21	108,10	108,11	108,08	108,12
2048	141,27	141,12	141,10	141,85	121,81	121,93	121,30	121,95

Tabell C.4: INSTANCE-buffere til loopback adresse.

C.5 Tradisjonelle systemkall ved høy CPU load

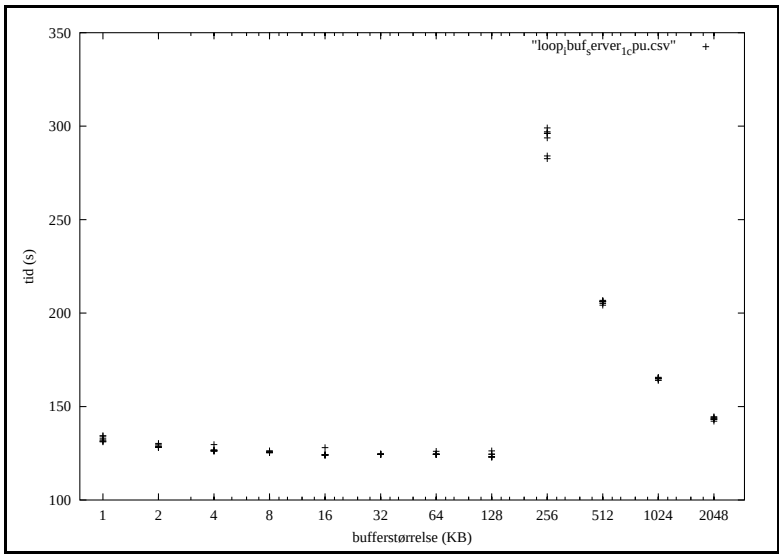


Figur C.5: Tradisjonelle systemkall til loopback adresse ved høy CPU load.

Size (kB)	$\bar{t}$ **(s)**	$\hat{t}$ **(s)**	**min(t)** (s)	**max(t)** (s)	$\bar{d}$ **(Mb/s)**	$\hat{d}$ **(Mb/s)**	**min(d)** (s)	**max(d)** (s)
1	353,41	350,67	343,35	364,35	48,71	49,07	47,23	50,11
2	187,43	187,52	184,01	193,21	91,83	91,76	89,06	93,51
4	169,98	188,70	130,07	194,18	104,05	91,19	88,61	132,29
8	117,41	117,41	117,02	117,86	146,56	146,56	146,00	147,05
16	133,33	133,15	130,38	136,58	129,10	129,23	125,99	131,98
32	144,13	143,75	140,50	147,44	119,41	119,70	116,70	122,47
64	172,61	153,19	149,12	204,12	101,73	112,33	84,30	115,39
128	222,13	222,42	211,31	230,74	77,55	77,36	74,58	81,43
256	249,11	251,18	237,52	258,32	69,12	68,51	66,61	72,45
512	238,24	257,41	167,21	265,04	74,15	66,85	64,92	102,96
1024	180,56	181,09	176,01	185,27	95,34	95,02	92,88	97,76
2048	179,67	180,04	174,06	184,46	95,81	95,57	93,28	98,86

Tabell C.5: Tradisjonelle systemkall til loopback adresse ved høy CPU load.

C.6 INSTANCE-buffere ved høy CPU load

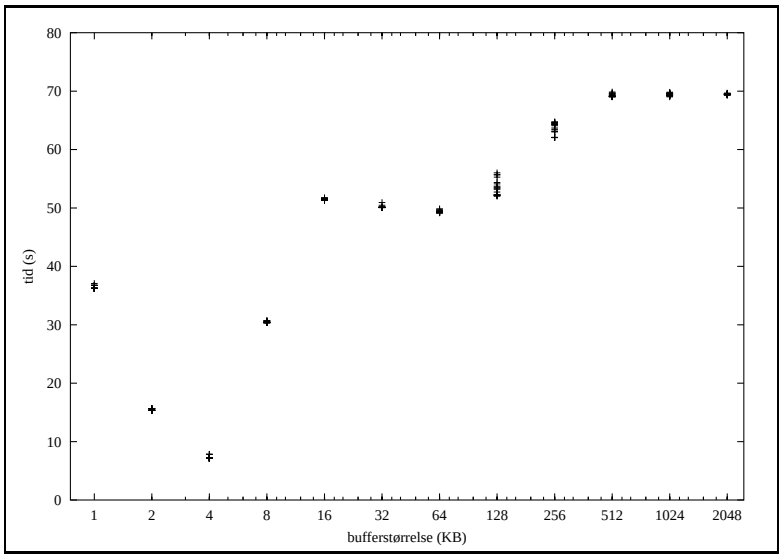


Figur C.6: INSTANCE-buffere loopback ved høy CPU load.

Size (kB)	$\bar{t}$ **(s)**	$\hat{t}$ **(s)**	**min(t)** (s)	**max(t)** (s)	$\bar{d}$ **(Mb/s)**	$\hat{d}$ **(Mb/s)**	**min(d)** (s)	**max(d)** (s)
1	132,56	132,16	131,12	134,35	129,82	130,20	128,08	131,23
2	128,83	128,72	128,15	130,26	133,57	133,68	132,10	134,28
4	126,89	126,36	126,23	129,67	135,61	136,18	132,70	136,31
8	125,69	125,65	125,35	126,30	136,90	136,95	136,24	137,28
16	124,68	124,12	124,10	128,01	138,03	138,64	134,42	138,65
32	124,48	124,42	124,38	124,62	138,23	138,30	138,08	138,34
64	124,70	124,42	124,38	126,05	137,88	138,29	136,51	138,34
128	123,93	123,07	123,02	126,31	138,86	139,82	136,23	139,87
256	292,69	296,08	282,63	299,04	58,82	58,12	57,54	60,88
512	205,86	206,13	204,20	206,79	83,59	83,48	83,21	84,27
1024	164,91	165,17	164,06	165,65	104,34	104,18	103,88	104,88
2048	143,50	143,42	142,15	144,51	119,91	119,97	119,08	121,05

Tabell C.6: INSTANCE-buffere til loopback adresse ved høy CPU load.

C.7 Inn og utkopiering fra kjernen

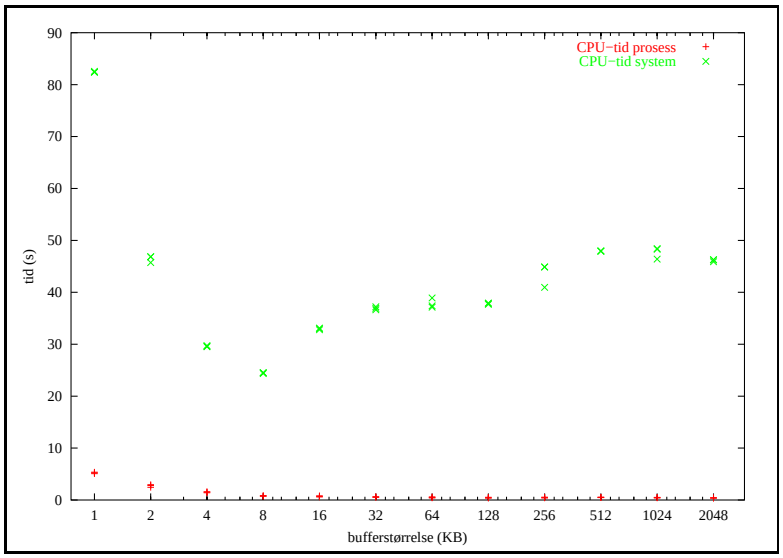


Figur C.7: Tidsforbruk ved inn og utkopiering fra kjernen

Size (kB)	$\bar{t}$ **(s)**	$\hat{t}$ **(s)**	**min(t)** (s)	**max(t)** (s)	$\bar{d}$ **(Mb/s)**	$\hat{d}$ **(Mb/s)**	**min(d)** (s)	**max(d)** (s)
1	36,38	36,24	36,22	37,07	472,97	474,80	464,19	475,06
2	15,46	15,36	15,34	15,65	1113,28	1120,25	1099,21	1121,64
4	7,27	7,18	7,17	7,83	2367,78	2396,87	2197,87	2399,21
8	30,46	30,37	30,37	30,63	564,57	566,55	561,79	566,56
16	51,44	51,35	51,32	51,68	334,51	335,12	332,95	335,29
32	50,15	50,10	50,06	50,93	343,05	343,47	337,83	343,72
64	49,36	49,21	49,19	49,81	348,61	349,22	345,45	349,86
128	53,43	53,22	52,00	56,02	322,27	323,13	307,17	330,89
256	63,67	64,08	62,05	64,74	270,30	268,19	265,78	277,33
512	69,26	69,13	69,01	69,85	248,46	248,91	246,34	249,34
1024	69,44	69,48	69,01	69,81	247,82	247,63	246,46	248,25
2048	69,47	69,45	69,32	69,67	247,71	247,73	246,99	248,25

Tabell C.7: Tidsforbruk ved inn og utkopiering fra kjernen

C.8 Forbruk av CPU-tid for tradisjonelle systemkall

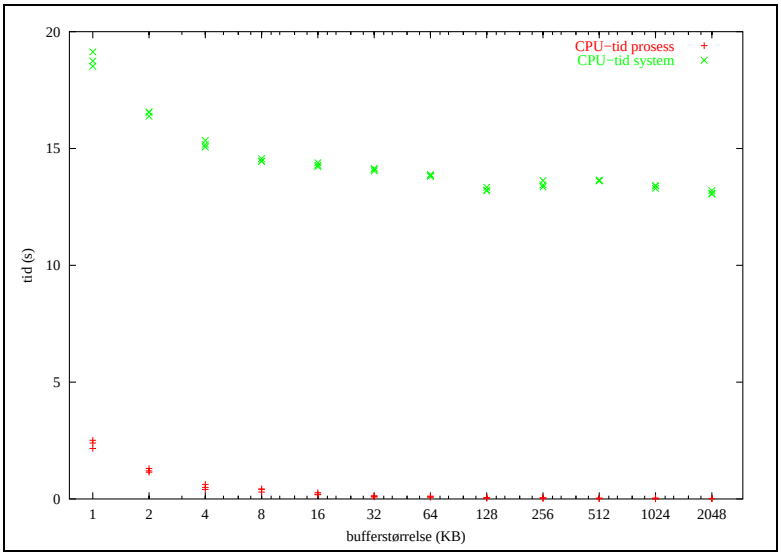


Figur C.8: Forbruk av CPU-tid for tradisjonelle systemkall.

Size (kB)	$\overline{\mathbf{cpu_p}}$ (s)	$\widetilde{\mathbf{cpu_p}}$ (s)	$\mathbf{min(cpu_p)}$ (s)	$\mathbf{max(cpu_p)}$ (s)	$\overline{\mathbf{cpu_s}}$ (s)	$\widetilde{\mathbf{cpu_s}}$ (s)	$\mathbf{min(cpu_s)}$ (s)	$\mathbf{max(cpu_s)}$ (s)
1	5.23	5.23	5.08	5.37	82.45	82.40	82.39	82.56
2	2.73	2.79	2.44	2.95	46.48	46.84	45.73	46.86
4	1.50	1.56	1.39	1.56	29.61	29.61	29.51	29.71
8	0.78	0.75	0.75	0.84	24.47	24.51	24.36	24.53
16	0.69	0.69	0.58	0.80	32.96	32.98	32.80	33.11
32	0.58	0.57	0.53	0.63	36.92	36.91	36.64	37.23
64	0.49	0.46	0.40	0.61	37.82	37.41	37.15	38.92
128	0.44	0.41	0.38	0.52	37.82	37.87	37.68	37.92
256	0.47	0.45	0.41	0.55	43.58	44.87	40.97	44.90
512	0.51	0.51	0.47	0.55	47.94	47.95	47.92	47.95
1024	0.45	0.44	0.40	0.50	44.71	48.30	46.39	48.42
2048	0.40	0.39	0.37	0.44	46.14	46.23	45.90	46.29

Tabell C.8: Forbruk av CPU-tid for tradisjonelle systemkall.

C.9 Forbruk av CPU-tid med INSTANCE-buffere

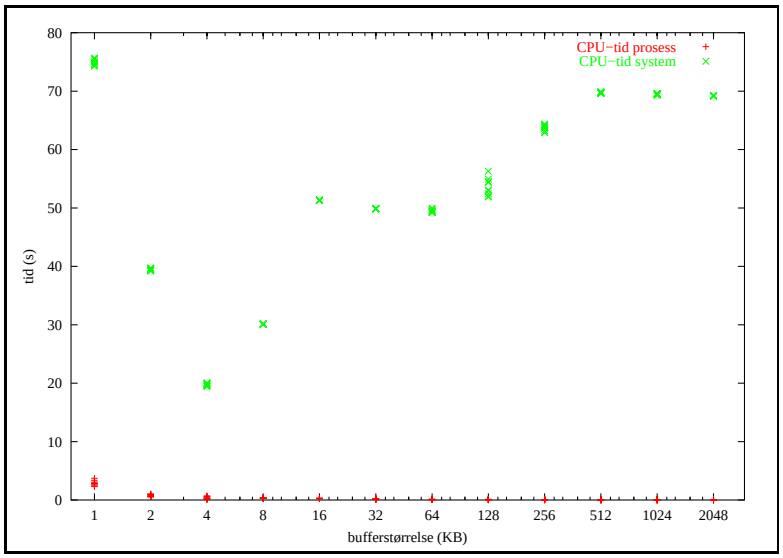


Figur C.9: Forbruk av CPU-tid med INSTANCE-buffere.

Size (kB)	$\overline{\mathbf{cpu_p}}$ (s)	$\widehat{\mathbf{cpu_p}}$ (s)	$\mathbf{min(cpu_p)}$ (s)	$\mathbf{max(cpu_p)}$ (s)	$\overline{\mathbf{cpu_s}}$ (s)	$\widehat{\mathbf{cpu_s}}$ (s)	$\mathbf{min(cpu_s)}$ (s)	$\mathbf{max(cpu_s)}$ (s)
1	2.36	2.40	2.16	2.51	18.80	18.74	18.52	19.14
2	1.22	1.20	1.15	1.30	16.50	16.55	16.38	16.56
4	0.51	0.50	0.41	0.62	15.19	15.17	15.07	15.34
8	0.38	0.42	0.30	0.43	14.49	14.46	14.45	14.56
16	0.22	0.20	0.18	0.27	14.30	14.28	14.22	14.39
32	0.11	0.10	0.09	0.14	14.10	14.11	14.04	14.15
64	0.10	0.09	0.08	0.12	13.85	13.86	13.80	13.88
128	0.04	0.04	0.04	0.05	13.25	13.22	13.19	13.34
256	0.04	0.05	0.02	0.05	13.48	13.44	13.36	13.63
512	0.02	0.02	0.01	0.04	13.63	13.64	13.61	13.64
1024	0.02	0.01	0.01	0.04	13.37	13.39	13.29	13.43
2048	0.02	0.02	0.01	0.02	13.11	13.09	13.05	13.20

Tabell C.9: Forbruk av CPU-tid med INSTANCE-buffere.

C.10 Forbruk av CPU-tid inn og utkopiering fra kjernen



Figur C.10: Forbruk av CPU-tid inn og utkopiering fra kjernen.

Size (kB)	$\overline{\mathbf{cpu_p}}$ (s)	$\widetilde{\mathbf{cpu_p}}$ (s)	$\mathbf{min(cpu_p)}$ (s)	$\mathbf{max(cpu_p)}$ (s)	$\overline{\mathbf{cpu_s}}$ (s)	$\widetilde{\mathbf{cpu_s}}$ (s)	$\mathbf{min(cpu_s)}$ (s)	$\mathbf{max(cpu_s)}$ (s)
1	2.89	2.82	2.30	3.72	74.96	74.99	74.24	75.67
2	0.79	0.84	0.54	1.05	39.45	39.38	39.20	39.71
4	0.34	0.27	0.11	0.71	19.79	19.86	19.41	20.04
8	0.31	0.32	0.26	0.38	30.10	30.11	30.04	30.16
16	0.26	0.26	0.19	0.31	51.32	51.32	51.27	51.38
32	0.16	0.16	0.09	0.20	49.87	49.86	49.82	49.93
64	0.08	0.08	0.05	0.12	49.47	49.45	49.23	49.97
128	0.06	0.05	0.03	0.12	53.61	53.07	51.87	56.28
256	0.05	0.05	0.02	0.08	63.68	63.74	62.89	64.36
512	0.02	0.01	0.00	0.05	69.73	69.74	69.60	69.89
1024	0.02	0.02	0.00	0.03	69.47	69.53	69.27	69.59
2048	0.00	0.00	0.00	0.01	69.18	69.19	69.10	69.26

Tabell C.10: Forbruk av CPU-tid inn og utkopiering fra kjernen.

Bibliografi

[1] E. L. Abram-Profeta og K. G. Shin. Scheduling video programs in near video-on-demand systems. I *Proceedings of the Fifth ACM International Conference on Multimedia (MULTIMEDIA '97)*, side 359–370, Seattle, Washington, USA, november 1997. ACM Press/Addison-Wesley.

[2] Aftenpostens, may 1999. Morenutgaven 19. mai.

[3] M. Bär, C. Griwodz, og L. Wolf. Long-term movie popularity models in video-on-demand systems. I *Proceedings of the Fifth ACM International Conference on Multimedia (MULTIMEDIA '97)*, side 349–358, Seattle, Washington, USA, november 1998. ACM Press/Addison-Wesley.

[4] R. Braden og L. Zhang. RFC 2209: Resource ReSerVation Protocol (RSVP) — version 1 message processing rules, september 1997. Status: INFORMATIONAL.

[5] R. Braden, L. Zhang, S. Berson, S. Herzog, og S. Jamin. RFC 2205: Resource ReSerVation Protocol (RSVP) — version 1 functional specification, september 1997. Status: PROPOSED STANDARD.

[6] J. C. Brustolini. Interoperation of copy avoidance in network and file I/O. I *Proceedings of the 18th IEEE Conference on Computer Communicatons (INFOCOM '98)*, New York, New York, USA, mars 1999. IEEE.

[7] J. C. Brustolini og P. Steenkiste. Evaluation of data passing and scheduling avoidance. I *Proceedings of the Seventh International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 96)*, side 101–111, St. Louis, Missouri, USA, mai 1997.

[8] J. C. Brustolini og P. Steenkiste. User-level protocol servers with kernel-level performance. I *Proceedings of the 17th IEEE Conference on Computer Communicatons (INFOCOM '98)*, San Fransisco, California, USA, mars 1998. IEEE.

[9] J. C. Brustoloni og P. Steenkiste. Effects of buffering semantics on i/o performance. I *Proceedings of the Second Symposium on Operating Systems Design*

and Implementation, side 277–291, Seattle, Washington USA, oktober 1996. USENIX.

[10] M. M. Buddhikot, X. J. Chen, D. Wu, og G. M. Parulkar. Enhancements to 4.4 BSD UNIX for efficient networked multimedia in project MARS. I *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS '98)*, Austin, Texas, USA, juni 1998.

[11] M. M. Buddhikot og G. M. Parulkar. Efficient data layout, scheduling and playout control in MARS. I *Proceedings of the Fifth International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 95)*, volume 1018, side 318–??, Durham, New Hampshire, USA, april 1995. Springer. Lecture Notes in Computer Science, Vol. 1018.

[12] M. M. Buddhikot, G. M. Parulkar, og J. R. Cox Jr. Design of a large scale multimedia storage server. *Journal of Computer Networks and ISDN Systems*, 27:503–517, desember 1994.

[13] E. Chang og A. Zakhor. Variable bit rate mpeg video storage on parallel disk arrays. I *First International Workshop on Community Networking Integrated Multimedia Services to the Home*, side 127–137, San Fransisco, USA, juli 1994.

[14] D. D. Clark og D. L. Tennenhouse. Architectural considerations for a new generation of protocols. *ACM SIGCOMM Computer Communication Review*, 4:200–208, september 1990.

[15] C. D. Cranor. *Design and implementation of the UVM virtual memory system*. Doktorgradsoppgave, Washington University, Saint Louis, Missouri, aug 1998.

[16] C. D. Cranor og G. M. Parulkar. The UVM virtual memory system. I *Proceedings of the 1999 USENIX Annual Technical Conference (USENIX-99)*, side 117–130, Berkeley, CA, juni 1999. USENIX Association.

[17] A. Dan, D. Sitaram, og P. Shahabuddin. Scheduling policies for an on-demand video server with batching. I *Proceedings of the Second ACM International Conference on Multimedia (MULTIMEDIA '94)*, side 15–24, San Fransisco, California, USA, oktober 1994. ACM Press.

[18] S. Deering og D. Cheriton. Multicasting routing in datagram internetworks and extended LANs. *ACM Trans. on Computer Systems*, 8(2):85–110, mai 1990.

[19] "P. J. Denning". "virtual memory". *"ACM Computing Surveys"*, "28"("1"):"213–216", mar "1996".

[20] DivX digest homepage, oct 2000. "http://www.divx-digest.com/".

[21] P. Druschel og L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. I Barbara Liskov, redaktør, *Proceedings of the fourtennth ACM Symposium on Operating Systems Principles (SIGOSP' 93)*, side 189–202, Asheville, North Carolina, USA, desember 1993. ACM Press.

[22] L. Gao, J. Kurose, og D. Towsley. Efficient schemes for broadcasting popular videos. I *Proceedings of the Eighth International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 98)*, Cambridge, Storbritannia, juli 1998. IEEE Computer Society.

[23] S. Glassman. A caching relay for the World-Wide Web. I *Proceedings of the 1st International WWW Conference*, Geneve, Sveits, mai 1994.

[24] K. A. Hua og S. Sheu. Skyscraper broadcasting: A new broadcasting scheme for metropolitan video-on-demand systems. I *Proceedings of the ACM SIGCOMM Conference : Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '97)*, volume 27,4 av *Computer Communication Review*, side 89–100, Cannes, Frankrike, september 1997. ACM Press.

[25] The internet engineering task force(IETF). `http://www.ietf.org/`, oktober 2000. The official IETF homepage.

[26] ISO. Information processing system - open systems interconnection - basic reference model, 1984. International Standard ISO-7498.

[27] M. R. Izquierdo og D. S. Reeves. A survey of statistical source models for variable-bit-rate compressed video. *Multimedia Systems*, 7(3):199–213, 1999.

[28] A. Luotonen og K. Altis. World-wide web proxies. I *Proceedings of the 1st International WWW Conference*, Geneva, Switzerland, mai 1994. Elsevier Science Publishers B.V.

[29] A. Mankin, F. Baker, B. Braden, S. Bradner, M. O'Dell, A. Romanow, A. Weinrib, og L. Zhang. RFC 2208: Resource ReSerVation Protocol (RSVP) — version 1 applicability statement some guidelines on deployment, september 1997. Status: INFORMATIONAL.

[30] M. K. McKusick, K. Bostic, M. J. Karels, og J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison Wesley, 1996.

[31] The netBSD project. `http://www.netbsd.org/`, oktober 2000. The official NetBSD Project homepage.

[32] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? I *Proc. 1990 Summer USENIX Conf.*, Anaheim, juni 11-15 1990.

[33] V. Pai, P. Druschel, og W. Zwaenepoel. IO-lite: A unified I/O buffering and caching system. I *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, Lousiana, USA, februar 1999. USENIX.

[34] J. Pasquale, E. Anderson, og P. K. Muller. Container shipping: Operating system support for I/O-intensive applications. *IEEE Computer*, 27(3):84–93, mars 1994.

[35] D. A. Patterson og J. L. Hennessy. *Computer organization and design: the hardware / software interface*. Morgan Kaufmann Publishers, 1994.

[36] T. Plagemann og V. Goebel. INSTANCE: The intermediate storage node concept. *Lecture Notes in Computer Science*, 1345:151–??, 1997.

[37] A. W. Read, redaktør. *The New International Webster's Comprehensive Dictionary of the English Language*. Trident International Press, deluxe encyclopedic edition utgave, 1996.

[38] R. Steinmetz og K. Nahrstedt. *Multimedia: Computing, Communications and Applications*. Prentice Hall, Upper Saddle River, New Jersey, USA, 1995.

[39] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, New Jersey, 3 utgave, 1996.

[40] H. Vin, P. Goyal, A. Goyal, og A. Goyal. A statistical admission control algorithm for multimedia servers. I *Proceedings of the Second ACM International Conference on Multimedia (MULTIMEDIA '94)*, side 33–40, San Fransisco, California, USA, oktober 1994. ACM Press.

[41] S. Viswanathan og T. Imielinski. Metropolitan area video-on-demand service using pyramid broadcasting. *Multimedia Systems*, 4(4):197–208, august 1996.

[42] A. Vogel, B. Kerhervé, G. von Bochmann, og J. Gecsei. Distributed multimedia and qoS: A survey. *IEEE Multimedia*, 2(2):10–19, 1995.

[43] G. R. Wright og W. R. Stevens. *TCP/IP Illustrated*, volume 2. Addison Wesley, 1995.

[44] "L. Zhang, S. Deering, og D. Estrin". "RSVP: A new resource ReSerVation Protocol. novel design features lead to an Internet protocol that is flexible and scalable". *IEEE network*, 7(5):8–18, september 1993.

Register