UNIVERSITETET I OSLO
Institutt for informatikk

Eksperimentell analyse av kommunikasjons-ytelse for disk til nettverk

Jan Erik Askjellrud

Hovedfagsoppgave

25. juli 2001

Forord

Dette er en hovedfagsoppgave innenfor fagområdet kommunikasjonssystemer ved Institutt for Informatikk ved Universitetet i Oslo. Jeg vil takke mine veiledere Pål Halvorsen og Thomas Plagemann for inspirasjon, rettledning og gode tilbakemeldinger i løpet av studietiden. Jeg vil også takke Vigdis Haaseth for god innsats med korrekturlesing av oppgaven.

Innhold

1 Innledning **1**
1.1 Problemstilling . . . 1
1.2 Fremgangsmetode . . . 2
1.3 Språkvalg . . . 3
1.4 Oppgavens struktur . . . 3

2 Scenario **5**
2.1 Kritiske elementer i serverbaserte systemer . . . 5
2.2 INSTANCE . . . 8
2.2.1 ILP . . . 9
2.2.2 Dataflyt . . . 9
2.2.3 Ytelse . . . 11
2.2.4 Analytisk evaluering . . . 11
2.3 Relatert arbeid . . . 12
2.4 Ytelsesanalyse av TCP over ATM . . . 13
2.5 SunOS IPC og TCP/IP . . . 14
2.5.1 Design av målingene . . . 15
2.5.2 Resultater fra målingene . . . 15
2.6 Minnehåndtering for høyhastighetsprotokoller . . . 16
2.7 Egne målinger . . . 16

3 Chorus **17**
3.1 Lagdelt struktur . . . 17
3.2 Actorer . . . 18
3.3 Tråder . . . 19
3.4 Minnehåndtering . . . 20
3.5 Kjernestruktur . . . 21
3.6 Sanntidsapplikasjoner . . . 22

4 NetBSD **25**
4.1 Kjernen . . . 26
4.2 Prosesshåndtering . . . 27
4.3 Minnehåndtering . . . 28

5 Disk **29**
5.1 Strukturering av data på disken . . . 29
5.2 Grensesnitt mot disk . . . 30
5.3 Scheduleringsalgoritmer . . . 30
5.3.1 First Come First Served . . . 30

5.3.2 Shortest Seek Time First 31
5.3.3 SCAN 32
5.3.4 C-SCAN 33
5.3.5 Earliest Deadline First 34
5.3.6 Position Delay Reduction 34
5.4 Bufferhåndtering 35
5.5 Read-ahead 37
5.6 Henting av buffere 37
5.7 Ulemper ved bruk av cache 38
5.8 Blokkstørrelse 38
5.9 Filsystemer 39

6 Nettverk 43
6.1 Socketlaget 43
6.2 Protokollaget 44
6.3 Nettverkslaget 44
6.4 Internettprotokollene 45
6.4.1 IP 45
6.4.2 TCP 46
6.4.3 UDP 48
6.5 Mbuffere 49
6.5.1 Kjede av mbuffere 50
6.5.2 Mbuffer med cluster 50
6.5.3 Flyt av mbuffere gjennom protokollene 52
6.6 Dataflyt 54
6.6.1 Socket 55
6.6.2 Protokollene 60
6.6.3 Linklaget 62

7 Målinger på disk-til-nettverk veien 65
7.1 Målinger på datasystemer 65
7.2 Valg av riktig type måling 65
7.3 Evalueringsteknikker 67
7.4 Valg av målingsmetode 67
7.5 Nøyaktighet 68
7.6 Bakgrunnslast 68
7.7 Teknisk spesifikasjon av maskinvare 69

8 Minneytelse 71
8.1 Problembeskrivelse 71
8.2 Valg av målingsmetode 71

8.3 Utførelse . . . 71
8.4 Resultater . . . 72
8.5 Chorus . . . 73
8.6 NetBSD . . . 75
8.7 Andre systemer . . . 76

9 Diskytelse **77**
9.1 Problemstilling . . . 77
9.2 Valg av målingsmetode . . . 77
9.3 Utførelse . . . 77
9.4 Implementasjon . . . 78
9.5 Resultater . . . 79
9.6 Chorus . . . 80
9.7 NetBSD . . . 85

10 Nettverksytelse **89**
10.1 Loopback . . . 89
10.1.1 Problemstilling . . . 89
10.1.2 Loopbackgrensesnittet . . . 89
10.1.3 Utførelse . . . 90
10.1.4 Resultater . . . 91
10.1.5 Chorus . . . 92
10.1.6 NetBSD . . . 94
10.2 JProbe . . . 95
10.2.1 Valg av målingsmetode . . . 95
10.2.2 JProbe-prøver . . . 96
10.2.3 Prøvenes innvirkning på resultatet . . . 97
10.2.4 Plassering av prøver . . . 97
10.2.5 Aktivering og henting av data fra prøver . . . 98
10.2.6 Prøvestruktur . . . 99
10.2.7 Implementasjon . . . 99
10.2.8 Utførelse . . . 103
10.2.9 Resultater . . . 103
10.2.10 Chorus TCP . . . 104
10.2.11 NetBSD TCP . . . 107
10.2.12 NetBSD UDP . . . 109

11 Konklusjon **113**
11.1 Sammendrag . . . 113
11.2 Antagelser og begrensninger . . . 114
11.3 Mulige løsninger . . . 115

11.4 Videre arbeid 116
11.5 Samlet konklusjon 116

A Endringer i socket.h 1

B Endringer i socketvar.h 2

C Endringer i sosetopt() 3

D Endringer i sogetopt() 7

E Installasjon av Chorus 11
E.1 Tekniske data 11
E.2 Installasjon 12
E.3 IDE disk med UFS 13

Figurer

1 Oppgavens struktur 4
2 Kritisk vei i en server 6
3 INSTANCE-node 8
4 Core-RSD [1] 12
5 Gjennomstrømning avhengig av vindusstørrelse 13
6 Kø og prøver i SunOS 14
7 Lagdelt struktur i Chorus 18
8 Oppdeling av minne i Chorus 20
9 Kjernens struktur i Chorus 21
10 Prioritetsoppdeling 22
11 Prioritetsrekkefølge 23
12 Prosesshåndtering i NetBSD 27
13 Oppbygning av disk 29
14 FCFS 31
15 SSTF 32
16 SCAN 33
17 C-SCAN 33
18 EDF 34
19 Cache 36
20 MS-DOS filsystem 41
21 Ufs filsystem 41
22 Nettverkslag i BSD 43
23 IP-hode 46
24 Sliding window protokoll 47
25 Mbuffer 49
26 To mbuffere knyttet sammen 50
27 Mbuffer med ekstern cluster 51
28 Mbuffere i nettverkskoden 53
29 Dataflyt gjennom nettverkslagene 54
30 UIO-struktur før flytting av data 59
31 UIO-struktur etter flytting av data 60
32 Dataflyt i protokollaget 61
33 Dataflyt i linklaget 63
34 Tre mulige resultater av forespørsel etter tjeneste 66
35 Kopiering av minne, Chorus 73
36 Kopiering av minne, NetBSD 75
37 Lesing fra disk (read) 80
38 Lesing fra disk. (read, region 4) 82
39 Lesing fra disk. (re-read) 83

40 Lesing fra disk. (re-read, region 4) 84
41 Lesing fra disk (read) 85
42 Lesing fra disk. (read, region 4) 86
43 Lesing fra disk (re-read) 87
44 Lesing fra disk. (re-read, region 4) 88
45 Loopbackgrensesnittet i nettverkskoden 90
46 Loopbackresultater, Chorus 92
47 Loopbackresultater, NetBSD 94
48 Typisk maskinoppsett for Tcpdump 95
49 Plassering av JProbe-prøver i kjernen 98
50 Data ut fra socket- og nettverkskøen 104
51 Data i socket- og nettverkskøen 105
52 Data ut fra socket- og nettverkskøen 107
53 Data i socket- og nettverkskøen 108
54 Data ut fra socket- og nettverkskøen 109
55 Data i nettverk og socketkøen 110

1 Innledning

Multimediaapplikasjoner behandler store, komplekse, kontinuerlige og tidsavhengige dataelementer som for eksempel video, audio og animasjoner kombinert sammen med bilder, grafikk og tradisjonelle dataelementer, som for eksempel tekst [24]. Håndtering av slike datamengder i et multimediasystem stiller store krav til systemkomponentene. Datamengden i multimediaapplikasjoner er ofte stor og kompleks, og må derfor håndteres på en effektiv måte. Det gjelder for både endesystemene og nettverket dataene skal sendes over.

I den siste tiden har bredbånd vært i sterk fokus. Definisjonen av bredbånd er ikke klart definert, men det alle er enige om er at det gir en større båndbredde enn det som er vanlig i dag. Med introduksjon av ADSL og andre teknologier for bredbånd, blir flere og flere personer knyttet til internett med stadig større båndbredde. Dette medfører at et økende antall personer vil koble seg opp mot servere for henting av store mengder data. Eksempler på servere som sender store mengder data er VoD (Video On Demand), WWW (World Wide Web), realaudio og ftpservere. Servere vil derfor være en viktig ressurs for stadig flere brukere. I den forbindelse vil det være ønskelig å utnytte serverens kapasitet maksimalt. Dette kan gjøres ved å finne serverens svakeste ledd, for så å utbedre flaskehalsen som oppstår her. Denne hovedfagsoppgaven tar for seg denne problemstillingen. I oppgaven skal jeg utføre målinger på mikrokjerneoperativsystemet Chorus og operativsystemet NetBSD for å avdekke systemenes flaskehals, og på denne måten kunne forbedre flyten av data til multimediaservere som kjører på slike og lignende operativsystemer.

1.1 Problemstilling

Servere består av flere komponenter som jobber sammen for å betjene brukere av systemet. Data som skal sendes til brukeren hentes ofte først fra disk, behandles av multimediaapplikasjonen, sendes ut på nettverket og tas imot av maskinen til klienten. I denne kommunikasjonskanalen utgjør servere den enkle noden i en en-til-mange kommunikasjonskanal. Jo flere brukere som kommuniserer med serveren, jo mer belastet blir serveren. Servere har ikke ubegrenset med ressurser og det er en øvre grensen for hvor mange brukere den kan betjene. Når denne grensen er nådd er det fordi en komponent i veien fra der data hentes av serveren til der dataene sendes ut ikke har nok ressurser til rådighet. I en multimediaserver er disse komponentene organisert i en kjede, der dataene flyttes gjennom en komponent av gangen. Denne kjeden

starter med lesing av data fra disk og ender med å sende data ut på nettverket. Hver komponent i denne kjeden har begrenset med ressurser, og siden de er organisert i en kjede vil det svakeste leddet i denne kjeden bestemme den maksimale ytelsen for multimediaserveren. I denne oppgaven ser vi kun på ytelsen til multimediaserveren, og forutsetter derfor at nettverket ikke er en flaskehals.

For å sørge for at serveren kan betjene så mange brukere som mulig er det nødvendig å lokalisere punktet i serveren (flaskehalsen) som fører til redusert ytelse. Flaskehalsen vil befinne seg et sted mellom disken og nettverket. Dette punktet kan eksistere flere steder i serveren. Eksempler på punkter i serveren som kan skape en flaskehals er begrensninger i minnehastighet, diskhastighet, prosessorhastighet, køing av data og protokollbehandling. Målet med denne oppgaven er derfor å finne hvor i multimediaserveren denne flaskehalsen befinner seg. For å kunne finne dette punktet er det nødvendig å gjøre en analyse av komponentene i multimediaserveren. I denne oppgaven har jeg valgt å gjøre en eksperimentell analyse av disse komponentene. Disse resultatene kan senere brukes for å gi en formening om hvor god ytelsesforbedringer man kan oppnå ved å bygge om systemkomponentene i en multimediaserver, slik det er beskrevet i INSTANCE (avsnitt 2.2).

Denne oppgaven tar for seg multimediaservere som skal brukes under INSTANCE. Når denne oppgaven ble påbegynt var det valgt å bruke operativsystemet Chorus som utgangspunkt for multimediaserveren. Det har siden vist seg at det ikke er hensiktsmessig å kjøre en multimediaserver på dette operativsystemet. Grunnen til det er at det har kommet nye og raskere komponenter i maskinvaren ettersom tiden har gått. Disse komponentene viser det seg at Chorus ikke støtter, og en multimediaserver på Chorus vil derfor ikke være aktuelt. Det ble istedenfor valgt å bruke operativsystemet NetBSD, som er oversiktlig og har god støtte for nye systemkomponenter. I denne oppgaven har jeg derfor gjort en eksperimentell analyse av begge disse operativsystemene.

1.2 Fremgangsmetode

Strategien for å løse oppgavens problemstilling har bestått av både litteraturstudier, kildekodestudier, implementasjon og utførelse av målinger. Først har jeg satt meg inn i fagområdet ved å studere relevant faglitteratur. Litteraturen omfatter utførelse av målinger på datasystemer, operativsystemer, datakommunikasjon, disk, minnehåndtering, INSTANCE og lignende. Deretter studerte jeg artikler hvor det var utført ytelsesanalyser på datasys-

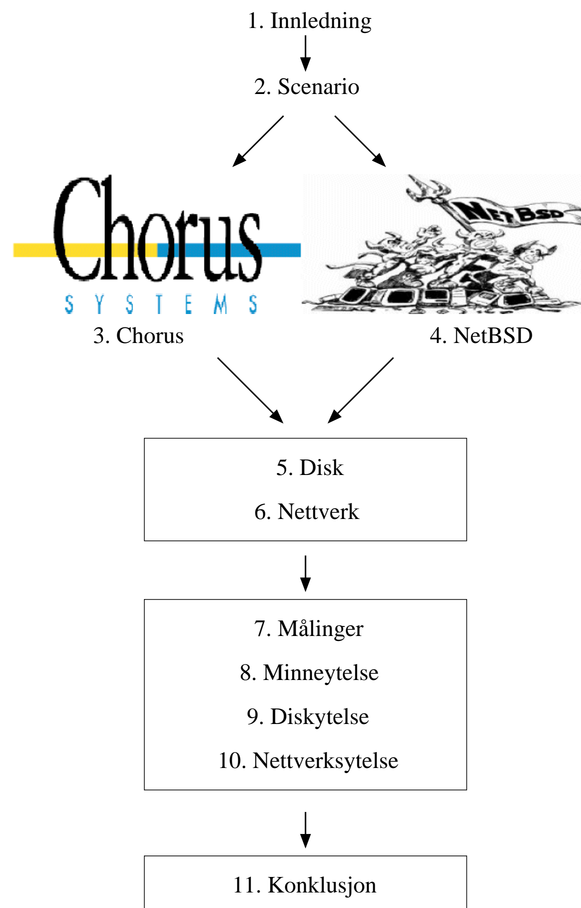
temer, og så på resultater som var funnet i disse artiklene. På denne måten kunne jeg finne ut hvordan jeg skulle utføre mine egne målinger. Når jeg hadde funnet ut hvilke målinger og hvordan jeg ville utføre målingene, ble kildekoden til kjernen i både Chorus og NetBSD studert. Kildekodestudiene ble fokusert på disk- og nettverkskommunikasjon, for å se hvordan målingene kunne implementeres. Målingene ble så implementert og utført. Resultatene fra målingene ble deretter samlet inn og behandlet. Til slutt undersøkte jeg resultatene jeg fikk, og sammenlignet de med resultater fra artiklene som tok for seg relaterte målinger.

1.3 Språkvalg

Fagområdet datakommunikasjon preges av bruk av engelsk fagterminologi. I arbeidet med denne oppgaven har jeg derfor måtte gjøre en avveining mellom forståelse og fornorsking av ord og uttrykk. Problemet oppstår fordi de engelske ordene brukes i litteraturen, og fordi det ofte ikke eksisterer noen god norsk oversettelse for termene. Selv om ordet lar seg oversette, er det likevel ikke sikkert at leseren vil gjenkjenne det norske ordet og assosiere det med den engelske betydningen. På tross av dette har jeg forsøkt å oversette de engelske begrepene så langt det lar seg gjøre.

1.4 Oppgavens struktur

Oppgaven er delt opp i 11 kapitler som vist i figur 1. Først kommer en introduksjon til oppgaven. Etter introduksjonen følger et kapittel som beskriver komponentene en multimediaserver er bygd opp av, INSTANCE og relatert arbeid. I de to neste kapitlene tar jeg for meg operativsystemene som jeg skal utføre målinger på. Operativsystemene er Chorus, som er et sanntids mikrokjerneoperativsystem, og NetBSD som er et UNIX-lignende operativsystem. Etter operativsystemene er beskrevet, gis en oversikt over mekanismer og komponenter som utgjør disk-til-nettverk veien i operativsystemet. Den første delen tar for seg disken og elementer som virker inn på diskens ytelse. Den andre delen beskriver elementer i nettverksdelen av operativsystemet, og hvordan disse virker inn på flyt av data fra applikasjon og ut på nettverket. Etter kapitlene som beskriver disk og nettverk, følger fire kapitler som inneholder en beskrivelse, utførelse og resultater fra målingene som ble gjort på de forskjellige komponentene i disk-til-nettverk veien. Til slutt følger et kapittel som oppsummerer og trekker en konklusjon fra målingene som ble gjort i de fire foregående kapitlene.



Figur 1: Oppgavens struktur

2 Scenario

Servere som tilbyr tjenester, som f.eks. World Wide Web (WWW) og Video on Demand (VoD), er ofte sentraliserte systemer ved at oppslag blir gjort mot kun en maskin. Dette gjør serveren til et svakt punkt, og serveren er derfor ofte en flaskehals i systemet. I serverbaserte systemer som tilbyr tjenester til flere brukere samtidig stilles det store krav til effektiviteten av serveren. Det er derfor viktig at serveren bruker minst mulig ressurser på overføring av informasjon fra server til bruker.

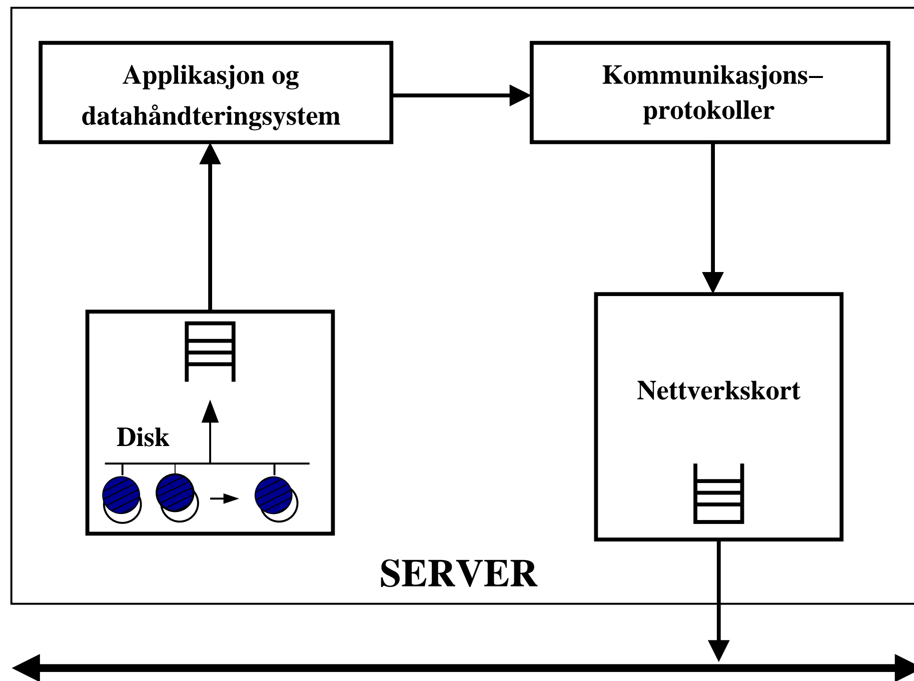
For at serveren skal bruke minst mulig ressurser på overføringen er det viktig at ingen av komponentene i disk-til-nettverk veien er en flaskehals. I dette kapittelet vil jeg beskrive disse komponentene, gi en beskrivelse av INSTANCE [1] og se på arbeid som er relatert til denne oppgaven. INSTANCE er en beskrivelse av en ny arkitektur for serverbaserte systemer.

2.1 Kritiske elementer i serverbaserte systemer

Den kritiske veien i en server er som nevnt veien fra lagringsmediet frem til nettverket. Det er viktig at serveren bruker så kort tid som mulig på å flytte data over denne veien. Ved å identifisere og forbedre det svakeste leddet i denne veien, kan man forbedre den maksimale gjennomstrømning av data. Den kritiske veien i serveren er vist i figur 2.

De viktigste elementene i denne veien er:

- Lagringsmediet
  Harddisk er det lagringsmediet som brukes mest i serverbaserte systemer. Skal man bruke serveren til VoD kreves det høy overføringshastighet fra harddisken. Kommersielle harddisker har begrenset overføringshastighet og kan være en flaskehals i et multimediasystem. Typiske overføringshastigheter på dagens disker ligger på mellom 15 og 50 MB/s [37], avhengig av hvilken type disk man velger og hvor på disken dataene leses fra. Siden moderne harddisker benytter "Zoned Bit Recording", vil harddisken være raskere ytterst enn innerst. På grunn av begrensninger i diskenes overføringshastighet brukes det ofte i multimediasystemer mange parallelle disker, såkalte "Redundant Array of Inexpensive Disks" (RAID) [13], for å øke hastigheten. Når dataene skal lagres blir de spredt over flere disker, slik at de kan hentes inn raskere enn hastigheten på den enkelte disken.



Figur 2: Kritisk vei i en server

– Datahåndteringssystemet
  Systemet holder rede på hvor forskjellige data er lagret på disken, og tilbyr et enkelt grensesnitt mot lagringsmediet for applikasjonen. Etter at en applikasjon har gjort en forespørsel etter data, kopierer datahåndteringssystemet dataene til applikasjonen.

– Applikasjonen
  Applikasjonen er den tjenesten som står for å velge hvilke data som skal hentes fra lagringsmediet, og å sende dataene videre. Den bearbeider dataene, hvis nødvendig, og kopierer dem så videre til kommunikasjonsprotokollene. Hastigheten til applikasjonen avhenger av hvordan dataene som er lagret på disk behandles før de sendes videre.

– Ende-til-ende protokoller
  Protokollene mottar data fra applikasjonen, splitter de opp i pakker og legger så til de tjenester applikasjonen krever. Det kan f.eks. være feilkontroll, flytkontroll og retransmisjon. Etter at ende-til-ende protokollene har delt opp data til pakker og lagt til kontrollinformasjon (PCI), kopieres pakkene til nettverkskortet. Programvareprotokoller kan være en stor flaskehalse i høyhastighetskommunikasjon, da dagens

nettverk ofte har større båndbredde enn serverens evne til å sende ut data [5].

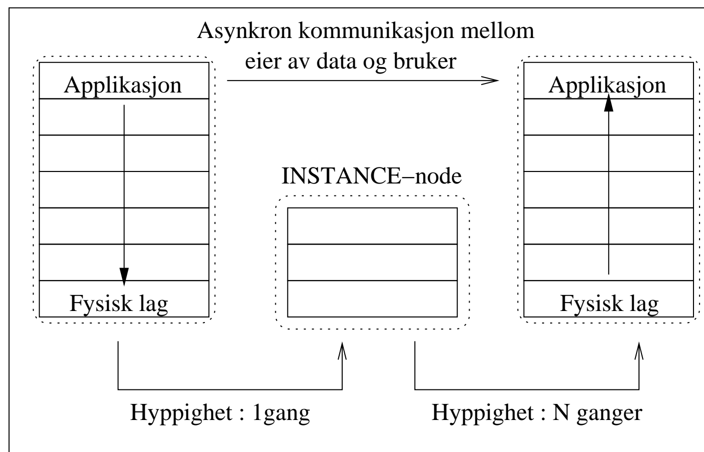
– Nettverkstilkobling
Nettverkskortet tar imot pakker fra ende-til-ende protokollen og står for å fysisk sende pakkene ut på nettverket. Ethernet er en type nettverkskort som brukes mye i dag, og følger ofte med som standard når man kjøper komplette maskiner. Disse kortene kommer i flere typer hastigheter. De vanligste er 10 Mbit/s kort, mens 100 Mbit/s kort blir stadig vanligere. Det finnes også 1 Gbit/s kort for ethernet selv om disse ikke er så vanlige, da de fortsatt koster en del mer enn 10 Mbit/s og 100 Mbit/s kortene.

2.2 INSTANCE

INSTANCE (Intermediate Storage Node Concept) er et prosjekt som startet på UNIK (Universitetsstudiene på Kjeller) i august 1997. Prosjektets mål er å kunne tilby asynkron kommunikasjon i serverbaserte systemer mer effektivt ved en ny struktur av serveren. I asynkron kommunikasjon sendes ikke informasjonen direkte til brukeren, men lagres midlertidig på en server for å kunne hentes ned eller sendes senere. Eksempler på servere som tilbyr asynkron kommunikasjon er WWW, VoD og distribuerte filsystemer.

I dette kapittelet beskriver jeg virkemåten til INSTANCE. Først skal vi se på hvordan systemet er tenkt bygd opp, og så ser vi på hvilke metoder INSTANCE bruker for å øke effektiviteten til serveren, som f.eks. "Integrated Layer Processing" (ILP) [11] .

I en vanlig klient-tjener arkitektur ser man på serveren som et fullstendig endesystem. Dette gjør at serveren må behandle alle lagene i kommunikasjonsprotokollen, fra applikasjonslaget til det fysiske laget. I INSTANCE derimot, ser man på serveren som en node mellom eier og bruker av data. Data blir sendt fra eieren til INSTANCE-noden kun en gang, hvor de mellomlagres, for så å kunne bli hentet ned flere ganger av brukeren. Denne forskjellen i tenkemåte gjør at en INSTANCE-node ikke trenger alle kommunikasjonslagene som en tradisjonell server har. Figur 3 viser dette.



Figur 3: INSTANCE-node

I OSI-referansemodellen [12] må en tradisjonell server behandle syv lag, mens en INSTANCE-node kun behøver å behandle de tre nederste lagene. Hvis pakkene ikke går over flere forskjellige nettverk vil det holde med å behandle de to nederste lagene. De øverste lagene, som en INSTANCE-node ikke bruker, er de mest tidkrevende å behandle. På denne måten slipper INSTANCE den prosessorkrevende ende-til-ende protokollbehandlingen, og reduserer dermed lasten på serveren.

I tradisjonelle servere flyter data fra disk til nettverkskortet gjennom ett lag av gangen, som vist i figur 2. Data kopieres fra lagringsmediet til datahåndteringssystemet, fra datahåndteringssystemet til applikasjonen, fra applikasjonen til ende-til-ende protokollene, og så til slutt fra ende-til-ende protokollene til nettverket. Dette medfører mange kopieringsoperasjoner for å flytte data fra lagringsmediet og ut på nettverket. I INSTANCE tar man i bruk en metode som reduserer antall kopieringsoperasjoner. Ideen er at datahåndteringssystemet, applikasjonen og ende-til-ende protokollene bruker delt hukommelse, og slipper på den måten å kopiere data seg imellom. Dette gjør at dataene bare behøver å kopieres fra lagringsmediet til det første laget (datahåndteringssystemet), og så ned til nettverkskortet. Dette gir INSTANCE en reduksjon på opptil tre kopieringsoperasjoner i forhold til tradisjonelle systemer.

2.2.1 ILP

Lagdelte protokoller implementeres ofte som isolerte moduler for hvert lag i protokollhierarkiet. Dette gir en oversiktlig og ryddig implementasjon, som følge av at man ikke trenger å tenke på andre lag enn det laget man implementerer. Det er kun grensesnittet mot de andre lag en må ta hensyn til. Selv om dette er ryddig, viser det seg at det ikke alltid er den mest effektive måten å implementere protokollene på. Ved å prosessere protokollene i en eller to omganger, uten å dele opp koden i forskjellige lag, kan man øke effektiviteten til protokollbehandlingen [38]. En annen fordel man oppnår ved å behandle protokollene på denne måten er at man slipper å skifte mellom flere prosesser, da hele protokollbehandlingen ligger i samme prosess.

2.2.2 Dataflyt

Når eieren av en mengde data skal sende dataene sine til en INSTANCE-node, overføres dataene på en annen måte enn i tradisjonelle systemer. Vi skal her ta for oss et eksempel der vi skal overføre en videofil til en videoserver, og så som bruker hente ned filmen fra serveren.

Dataene som representerer filmen vi skal legge på serveren blir først hentet fra kamera eller analog tape, og blir så digitalisert. Den digitaliserte filmen mottas av applikasjonen, som sender den videre til kommunikasjonslaget. I kommunikasjonslaget blir dataene først delt opp i mindre pakker, og så sendt til transportprotokollen (TCP). Der blir pakkene tilført protokollinformasjon (PCI) på 24 bytes. Pakkene kopieres så til IP, hvor det legges til 24 nye bytes med protokollinformasjon. I INSTANCE brukes ikke TCP/IP da disse protokollene ikke er fleksible nok. INSTANCE vil derfor ta i bruk et annet system kalt Da CaPo [47], eller sende pakkene over UDP. I Da CaPo kan man velge hvilken funksjonalitet man ønsker i protokollhierarkiet, og tilpasse disse etter behovet til applikasjonen. Men man trenger også PCI informasjon i Da CaPo, så vi antar her at den bruker 48 bytes som TCP/IP på PCI-informasjon.

Pakkene som ble generert ut fra videofilmen har ikke en tilfeldig størrelse. Størrelsen til pakkene blir generert ut fra tre kriterier. Dette er maksimal pakkestørrelse på nettet (MTU), størrelsen på blokkene i filbehandlingsystemet og størrelsen på de fysiske diskblokkene. Hvis vi i eksempelet vårt sender dataene over ethernettet, vil MTU være 1500 bytes. Vi antar videre at vi har en disk med fysisk blokkstørrelse på 4 KBytes, og en logisk størrelse på blokkene i filbehandlingssystemet på 1 KByte. Vi velger nå størrelsen på pakkene, inkludert PCI-informasjon, til en størrelse som passer alle disse tre kriteriene, altså 1 KByte. Grunnen til at pakkene tilpasses disse størrelsene er at de da kan flyttes raskere, siden de ikke går over flere buffere i noen av systemene. Når pakken senere skal hentes fra serveren, kan man kopiere en blokk fra disken inn til kun én buffer i filhåndteringsystemet og videre til én pakke ut på nettverket.

Før pakkene sendes til serveren genereres det en ekstra pakke som kun inneholder paritetsinformasjon. I vårt eksempel genererer vi en paritetspakke for hver tredje datapakke. Når data og paritetspakkene kommer frem til serveren blir de ikke endret, men lagret direkte til disk slik de så ut etter de var sendt, inkludert PCI-informasjon. I INSTANCE bruker man ikke vanlige disker for å lagre pakkene, men bruker isteden et RAID-system. Pakkene som kommer inn blir da i dette tilfellet spredt over fire forskjellige disker i RAID-systemet. På denne måten sørger paritetspakken vår for å kunne rette opp eventuelle diskfeil som kan oppstå i RAID-systemet. Vi har nå fått sendt filmdataene over til serveren.

Når vi nå som bruker skal hente ned denne filmen, hentes pakkene fra disk og sendes til protokollene. Det eneste som blir gjort med pakkene her

er at adresseinformasjonen i PCI-delen på pakkene blir endret til brukerens adresse. Resten av protokollinformasjonen ligger allerede i pakken lagret på disk, slik at dette ikke må fylles ut. Pakkene blir så sendt ut på nettverket frem til mottakeren. Der blir pakkene behandlet av protokollene opp til applikasjonslaget, og ved eventuelle feil i pakkene kan man ta i bruk paritetspakken. Pakkene kan nå settes sammen igjen og vises som en film hos brukeren.

2.2.3 Ytelse

Vi har nå sett på hvordan INSTANCE virker. For at vi skal kunne gjøre oss opp en formening om hva man kan tjene på å ta i bruk et INSTANCE-system, skal vi se på en ytelsesanalyse av systemet. Denne ytelsesanalysen baserer seg på målinger på SunOS IPC (Inter-Process Communication) og TPC/IP protokollen [3], som vi skal se på etter vi har sett på ytelsesanalysen av INSTANCE.

2.2.4 Analytisk evaluering

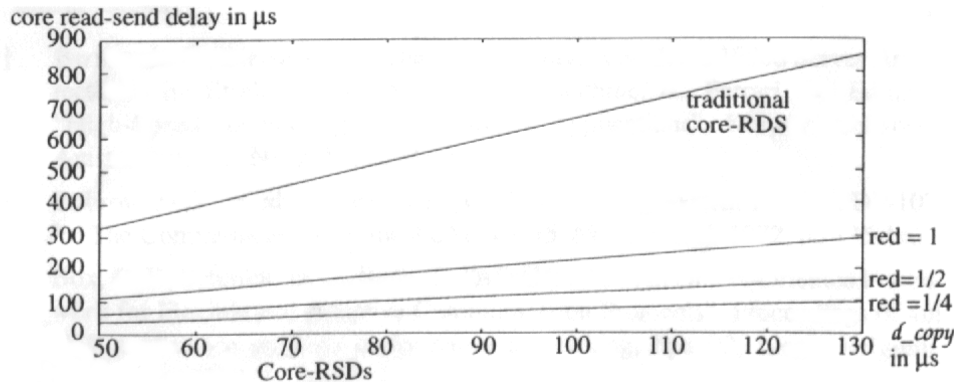
For å få en formening om ytelsen til INSTANCE, ser vi på tiden INSTANCE bruker på å flytte data fra disk til nettverk. I tradisjonelle servere leses data fra disk, kopieres til applikasjonen (d_copy), kopieres til socket og videre til protokollaget (d_socket), behandles av protokollene (d_prot) og sendes ut på nettverket. Vi kaller den tiden serveren bruker på å kopiere dataene fra disk til nettverket for read-send delay (RSD). I INSTANCE slipper vi d_copy, d_socket og d_prot, da vi bruker ILP. Vi kaller disse tre operasjonen for core-RSD, siden det er de som utgjør forskjellen fra en tradisjonell server. Men vi må også huske på at det tar ekstra tid å sette adressene i pakkene (d_addr), kopiere ekstra informasjon (red) som ligger i paritetspakken og lese PCI-informasjon. Dette gir oss følgende ligning for RSD-reduksjon i INSTANCE sammenlignet med en tradisjonell server :

$$RSD - reduksjon = d_copy + d_socket + d_prot - (d_addr + 2red * d_copy)$$

Alle disse faktorene er svært avhengig av det bestemte systemet serveren kjører på, inkludert maskinvare og operativsystem. For å få en idé av den mulige RSD-reduksjonen vi kan oppnå, bruker vi resultater fra målingene på SunOS. Følgene tider ble målt for en 1 KByte blokk : d_copy $= 130\mu$s, d_socket $= 280\mu$s og d_prot $= 443\mu$s. Videre finner vi ut at d_addr tilsvarer $\frac{1}{4} * d_copy$. Basert på at d_copy, d_socket og d_prot ikke endrer seg i

forhold til hverandre i raskere systemer, har vi at $d_socket = 2,15 * d_copy$ og $d_prot = 3,41 * d_copy$. Vi får da følgende core-RSD for INSTANCE og tradisjonelle servere :

Tradisjonell core-RSD : $6,56 * d_copy$
INSTANCE core-RSD : $\frac{1}{4} * d_copy + 2red * d_copy$



Figur 4: Core-RSD [1]

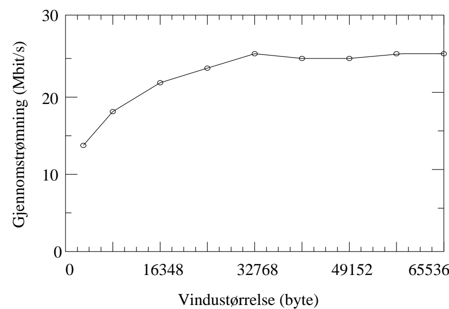
Ut i fra disse to funksjonene kan vi plotte en kurve for en tradisjonell server, og en kurve for de forskjellige verdiene av red i INSTANCE. I figur 4 er disse kurvene plottet for verdier av d_copy som går fra 50 til 130μs. Ut i fra disse kurvene kan vi få en antagelse av hvor mye det er å tjene på å ta i bruk INSTANCE. Vi må her huske at dette bare er en enkel analytisk modell, som ikke tar hensyn til alle detaljer.

2.3 Relatert arbeid

I avsnitt 2.4, 2.5 og 2.6 skal vi se på tre relaterte ytelsesanalyser av målinger som er utført på tre forskjellige endesystemer. Den første målingen ser på hvordan forskjellige typer konfigurasjon av maskinvare og programvare virker inn på dataflyten til endesystemet. Den andre ytelsesanalysen går mer i detalj. Den ser på dataflyten i nettverkskoden ved å modifisere operativsystemets kildekode, for å kunne hente ut informasjon knyttet til datastrømmen gjennom nettverkskoden. Den siste målingen undersøker ytelsen til et system som bruker delt minne for å redusere antall kopieringsoperasjoner gjennom kommunikasjonskanalen.

2.4 Ytelsesanalyse av TCP over ATM

Vi skal her se på en ytelsesanalyse [7] som er gjort på maskinene Sparc2 og Sparc10 med operativsystemet SunOS 4.1.x. Ytelsesanalysen fokuserer på hvordan ytelsen til TCP over ATM avhenger av endesystemets konfigurasjon for forskjellige typer maskinvare og programvare. I tillegg til å gjøre målinger med forskjellige typer komponenter, ble konfigurerbare parametre som minnestørrelsen til databufferet som skal sendes, størrelsen på socketens buffer og TCP-valg undersøkt.



Figur 5: Gjennomstrømning avhengig av vindusstørrelse

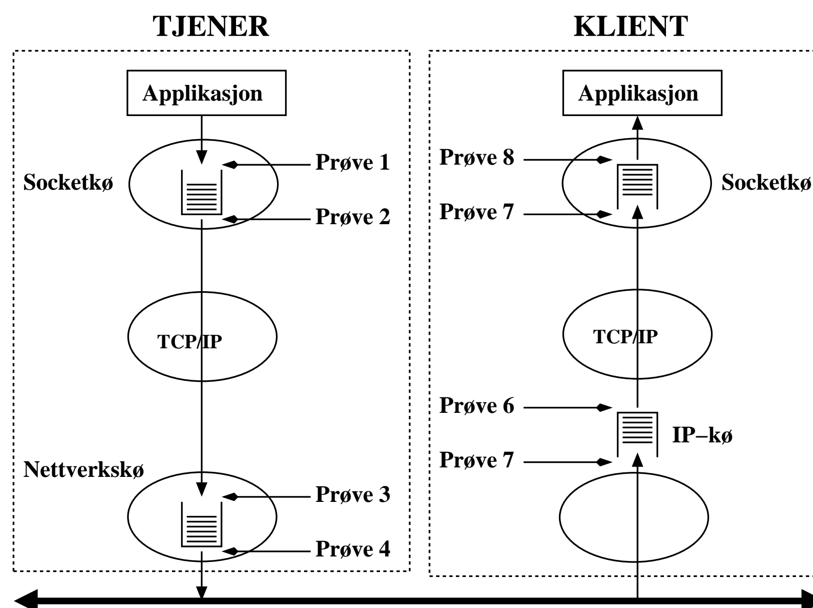
Fra målingene ble det funnet at den mest effektive vindusstørrelsen til TCP og dens gjennomstrømning er avhengig av både sender og mottakers socketbuffer. Når ingen av endesystemene er overbelastet er vindusstørrelsen, gitt av socketens buffer, en flaskehals. Ved å øke socketens buffer, øker gjennomstrømningen som vist i figur 5.

Størrelsen på datasegmentene som skal sendes avgjør antall systemkall som må gjøres for å sende en melding. Gjennomstrømningen av data synker betraktelig når størrelsen på datasegmentene minsker. For 128 bytes store datasegmenter er gjennomstrømningen på 4 Mbit/s, mot en gjennomstrømning på 26 Mbit/s for 8192 bytes store segmenter. For større segmenter flater gjennomstrømningen ut, og for segmenter fra 16384 til 32768 bytes synker gjennomstrømningen så ned til 24 Mbit/s.

Forskjellige TCP-valg viste seg å påvirke gjennomstrømningen av data. Å skru av Nagle-algoritmen fikk en positiv innvirkning på middels store vindusstørrelser, men for større vinduer minsket gjennomstrømningen. Videre viser det seg at en økning av MTU øker TCPs MSS (Maximum Segment Size), men ikke nødvendigvis gjennomstrømning av data.

2.5 SunOS IPC og TCP/IP

Vi skal nå se på en måling utført på operativsystemet SunOS 4.0.3 [6], som er basert på 4.3 BSD UNIX. I 4.3 BSD UNIX er IPC organisert i tre lag. Det første laget, socketlaget, er IPCs grensesnitt mot applikasjonen. Socketlaget tilbyr flere typer socketer, som f.eks. stream og datagram, hvor det i målingene ble brukt stream socket. Det andre laget er protokollaget, som tilbyr protokollbehandling som støtter de forskjellige sockettypene. Disse er gruppert i flere typer, og vi ser her på TCP/IP. Det siste laget er nettverkslaget, som inneholder drivere til maskinvaren for kommunikasjon. Disse tre lagene er vist i figur 6 for både sender og mottaker.



Figur 6: Kø og prøver i SunOS

Hver socket har en buffer for å sende og en for å motta data. Bufferne blir brukt til å mellomlagre data for sending og data mottatt fra protokollaget. Disse bufferne ligger i kjernen, og for å være fleksible og effektive, er de ikke kontinuerlige. De er bygd opp av såkalte mbuffere (Memory buffers), som er blokker på 128 bytes, og "cluster mbuffere" på 1024 bytes. Videre protokollbehandling blir gjort på disse mbufferne.

Når en applikasjon skal sende data, blir dataene først sendt til socketen, som deler opp dataene i mbuffere. Der blir de liggende i kø til protokollen er

klar til å sende dem. I en sikker forbindelse som TCP/IP vil dataene ligge der helt til de er bekreftet. Når protokollen mottar dataene fra socketkøen, splittes de opp i størrelser som passer nettverket, og nødvendige hoder legges til. Pakkene blir så sendt til nettverkslaget, hvor de blir lagret i en ny kø til driveren er klar til å sende dem. I målingene som blir utført undersøkes spesielt de to køene hvor pakkene blir mellomlagret. De er vist i figur 6 for både sender og mottakersiden.

2.5.1 Design av målingene

Målingene på SunOS er gjort ved at man designet spesielle funksjoner for å registrere informasjon om pakkene som kommer inn i køene. Dette er kodesegmenter som er plassert strategiske steder i kjernen, vist i figur 6. Når en prøve blir aktivert lagrer den informasjon om hvor mye data som passerer der prøven er satt, klokkeslettet og hvor prøven er tatt. Informasjonen man får av dette er minimal, men gir likevel god indikasjon på hvordan køene oppfører seg. For eksempel kan klokkeslettene gi informasjon om hvor lenge og hvor mange pakker det er i køen, hvor ofte pakker kommer inn i køen og hvor ofte de går ut av køen. Datalengden til pakkene gjør det mulig å måle gjennomstrømning, og finne ut hvor og hvordan data blir delt opp i pakker.

Et viktig poeng med prøvene er at de skal påvirke systemet så lite som mulig, slik at de ikke er med på å endre måleresultatene. Det må derfor brukes minst mulig ressurser på å kjøre prøvene, lagre prøveresultatene og aktivere prøvene. For å holde eksekveringstiden på et minimum må prøvene kun bruke tid på å registrere resultater. Prøveresultatene lagres så i en ringbuffer i kjernens virtuelle minne, som ble opprettet før målingene startet. Når målingene er over, hentes prøvene ut fra ringbufferet i kjernen av et annet program. På denne måten unngår man at prøvene virker inn på resultatene fra målingen.

2.5.2 Resultater fra målingene

Generelt er ytelsen til SunOS 4.0.3 IPC god i forhold til nettverkshastigheten på 10 Mb/s [6]. Eksperimentet viste at gjennomsnittlig overføringshastighet på 7.5 Mb/s er mulig, og det er ikke TCP/IP protokollen, men nettverksdriveren som er flaskehalsen. Forbedringer av IPC er likevel mulig. Målingene viste at ved å øke bufferstørrelsen til socketen økte gjennomstrømningen merkbart, men ulempen ved å øke socketens buffer er at det krever mye minne.

2.6 Minnehåndtering for høyhastighetsprotokoller

Denne ytelsesanalysen [27] ser på faktorer som påvirker ytelse og minnebehandling, som kan være en hindring for å oppnå høyhastighetskommunikasjon. Målinger er gjort på både minne- og nettverksytelsen.

Resultater fra minneytelsen viser at minne setter begrensninger på hastigheten til nettverkskoden. Fra resultatene ser vi at minneytelsen ligger på i overkant av 2 Gbit/s. Dette gir lite rom for ekstra kopieringsoperasjoner dersom man skal overføre data med hastigheten til et gigabit ethernet. Minnehastigheten kan derfor være en flaskehals i tradisjonelle servere, som benytter seg av flere kopieringsoperasjoner for å overføre data.

Målingene på nettverksytelsen fokuserer på hvordan forskjellige minnehåndteringsmetoder påvirker kommunikasjonshastigheten. Den ser på hvordan UVM-implementasjonen [49] i OpenBSD virker inn på gjennomstrømningen av data. Systemet som målingene blir utført på er modifisert, slik at det bruker UVMs "page loan". Dette tillater applikasjonen og kjernen å dele sider i minnet, slik at minneoperasjoner kan reduseres. Resultater viser at ved å ta i bruk er slikt system kan gjennomstrømningen øke med hele 46%, sammenlignet med bruk av vanlige kopieringsoperasjoner.

2.7 Egne målinger

Vi har nå sett på ytelsesanalyser som er relatert til de målingene jeg skal utføre senere i denne oppgaven. I siste kapittel skal jeg gå gjennom resultater fra mine egne målinger, og sammenligne resultatene med resultater fra ytelsesanalysene vi har sett på i dette kapittelet.

Den analytiske modellen av INSTANCE bruker resultater fra SunOS, men INSTANCE er tenkt kjørt på en annen type plattform. Resultatene fra SunOS er også fra en maskin som ikke lenger tilfredstiller kravene til en rask multimediaserver. Derfor vil denne oppgaven ta for seg to operativsystemer som har vært aktuelle for INSTANCE, med maskinvare som er betydelig raskere enn maskinen SunOS-målingene ble utført med. Disse to operativsystemene er Chorus og NetBSD.

3 Chorus

Utviklingen av Chorus startet på det franske forskningsinstituttet INRA i 1980 som et forskningsprosjekt innen distribuerte systemer. Prosjektet har siden utviklet seg fra et rent forskningsprosjekt til et kommersielt produkt, hvor hovedmålene er :

- Høyytelses UNIX-emulering.
  Mye arbeid er lagt i å få produktet til å følge UNIX-standardene, gjøre Chorus kompatibel med flere typer maskinvare og forbedre ytelsen.

- Bruk av distribuerte systemer.
  Chorus tillater UNIX-programmer å kjøre på flere maskiner koblet sammen i et nettverk.

- Sanntidsapplikasjoner.
  Chorus har støtte for sanntidsapplikasjoner, hvor programmene kjører delvis i kjernemodus og har direkte tilgang til mikrokjernen uten å gå gjennom ekstra programvare. Kontroll over interrupter og scheduleringsalgoritmer fra programmer som ikke kjører i kjernemodus er også mulig.

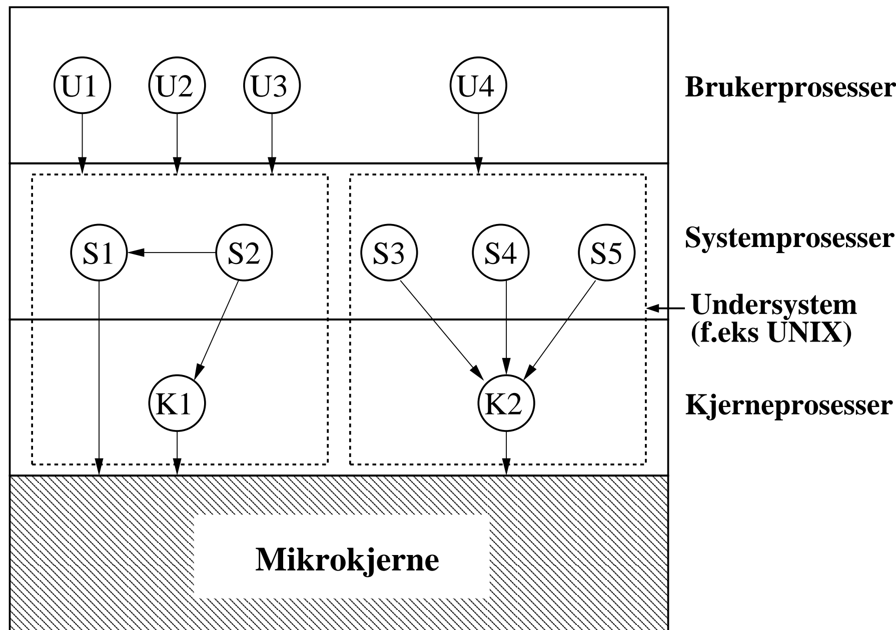
- Integrert objektorientert programmering.
  Målet er å introdusere objektorientert programmering i Chorus uten å endre på eksisterende undersystemer og applikasjoner.

3.1 Lagdelt struktur

Chorus er delt opp i flere lag (figur 7), med en mikrokjerne, undersystemer og brukerprosesser. I bunnen er mikrokjernen, som håndterer prosesser, tråder, minne og kommunikasjon. Over mikrokjernen kjører kjerneprosessene. Disse kan lastes dynamisk under kjøring, og er med på å utvide funksjonaliteten til kjernen uten å gjøre mikrokjernen større og mer kompleks.

Laget over inneholder systemprosessene. Disse kjører i brukermodus, men kan sende meldinger til kjerneprosessene og gjøre kall ned til mikrokjernen. En samling av kjerne og systemprosesser utgjør et undersystem, f.eks. UNIX, som gir brukeren et grensesnitt mot UNIX-systemkall. Med undersystemer er det mulig å bygge nye operativsystemer, og grensesnitt mot flere forskjellige operativsystemer kan eksistere på samme maskin samtidig. På toppen av Chorus-systemet er brukerprosessene, som må gjøre kall ned til undersystemet, som igjen kan gjøre kall til mikrokjernen. Brukerprosessene kan ikke

utføre kall direkte ned i mikrokjernen. Sanntidsprosesser kjøres derfor som systemprosesser, og kan på den måten utnytte mikrokjernen uten ekstra kall gjennom undersystemet.



Figur 7: Lagdelt struktur i Chorus

3.2 Actorer

I Chorus kalles en prosess for en actor. Chorus-actorer er essensielt det samme som en prosess i andre operativsystemer. En prosess i Chorus er en samling av aktive og passive elementer, som arbeider sammen for å utføre forskjellige typer utregninger. De aktive elementene er tråder. De passive elementene er adresserom som er delt opp i regioner som stack, data og program. En actor med en tråd, er det samme som en tradisjonell UNIX-prosess. En actor uten tråder kan ikke gjøre noe fornuftig, så de eksisterer som regel bare mens en prosess opprettes. I Chorus finnes det tre typer actorer, som skiller seg fra hverandre ved at de har forskjellige privilegier og om man må stole på dem eller ikke. Privilegiene går på om prosessene skal ha lov til å utføre I/O-operasjoner og andre instruksjoner som er beskyttet av kjernen. At man må stole på prosessen, vil si at den får lov til å kalle kjernen direkte.

Type	Må stole på?	Privilegier	Kjernemodus	Adresserom
Bruker	Nei	Nei	Bruker	Bruker
System	Ja	Nei	Bruker	Bruker
Kjerne	Ja	Ja	Kjerne	Kjerne

Kjerneprosessene er de kraftigste. De kjører i kjernemodus, og alle kjerneprosessene deler det samme adresserommet med mikrokjernen og de andre kjerneprosessene. De kan lastes inn og fjernes under kjøring, og kan tenkes på som en utvidelse av mikrokjernen. Kjerneprosesser kan snakke med hverandre ved hjelp av spesielle lettvekts RPC-kall, som ikke er tilgjengelig for andre prosesser enn kjerneprosesser.

Hver systemprosess har sitt eget adresserom. Systemprosesser er ikke privilegerte, og kan derfor ikke utføre I/O-operasjoner og andre begrensede instruksjoner direkte. Men systemprosesser må man stole på, og de kan derfor utføre systemkall direkte uten noe mellomledd.

Brukerprosesser er ikke privilegerte og skal ikke stoles på. De kan ikke utføre I/O-operasjoner direkte, og kan heller ikke gjøre kall mot kjernen med mindre de utføres gjennom undersystemet.

3.3 Tråder

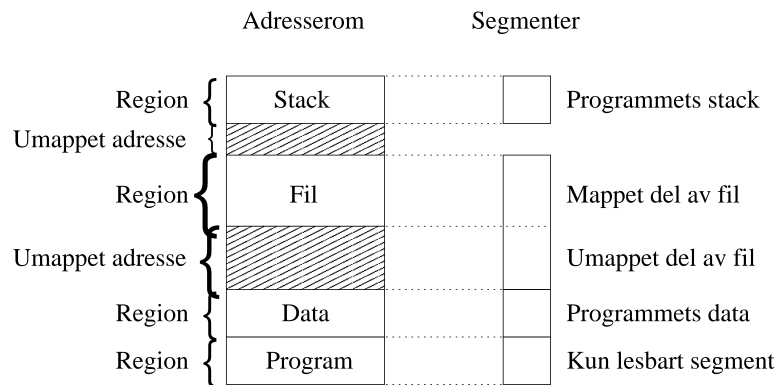
Inne i en prosess kan det eksistere en eller flere tråder. Hver tråd er lik en prosess ved at den har en egen stack, stackpeker, programteller og registre. Alle trådene som tilhører samme prosess deler det samme adresserommet og prosessens ressurser. I prinsippet er hver tråd i en prosess uavhengig av hverandre. På en multiprosessormaskin kan flere tråder kjøre samtidig. Alle de tre prosesstypene beskrevet over kan ha flere tråder. Tråder i Chorus er kjent for kjernen og kan scheduleres av kjernen. Opprettelse og sletting av tråder krever derfor kall ned til kjernen. En fordel med å støtte tråder i kjernen, er at når en tråd blokker for å vente på en hendelse, så kan kjernen schedulere andre tråder. En annen fordel er muligheten til å kunne kjøre forskjellige tråder på flere prosessorer når en multiprosessor er tilgjengelig. Ulempen med å ha støtte for tråder i kjernen, er den ekstra kjøretid som kreves for å behandle dem.

Kjernen i Chorus tilbyr to synkronisasjonsmetoder som tråder kan bruke. Den første er den tradisjonelle semaforen, med OPP og NED-operasjoner. Disse operasjonene er alltid implementert som systemkall, så de er derfor

kostbare. Den andre metoden er mutexer, som er essensielt det samme som en semafor, hvor verdiene er låst til verdiene 0 og 1.

3.4 Minnehåndtering

Minnebehandling i Chorus er basert på to konsepter, regioner og segmenter. Hver prosess har et adresserom som normalt går fra 0 til en maksimumsverdi, som f.eks. 2^{32} - 1. En region er et kontinuerlig område i det virtuelle adresserommet, f.eks. 4096 til 9063. I teorien kan en region begynne og slutte på en hvilken som helst adresse, men for å kunne være til nytte burde en region starte på begynnelsen av en side (page) og ha lengde lik et helt antall sider. Alle dataene i en region må ha de samme attributtene, f.eks. kun lesbar. Hver region er assosiert med et stykke data, som et program eller en fil. Regioner eies av prosesser, og alle tråder i en prosess har tilgang til de samme regionene. To regioner i samme prosess kan ikke overlappe hverandre. Regioner spiller en viktig rolle i minnebehandlingen i Chorus.



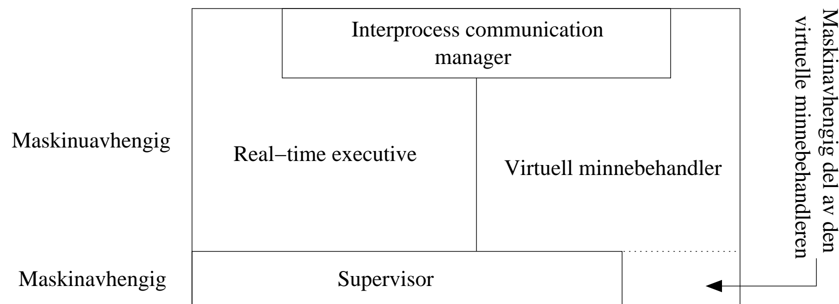
Figur 8: Oppdeling av minne i Chorus

Capability er navnet på en ressurs som brukes av undersystemet eller kjernen. Den gjenkjennes ved hjelp av en 64-bits unik identifikator. Et segment er en lineær sekvens av bytes identifisert ved en capability. Når et segment er mappet inn i en region, er dataene i segmentet tilgjengelig for trådene som tilhører regionens prosess. Programmer, filer og andre former for data er lagret som segmenter i Chorus og kan mappes i regioner. Et segment kan leses og skrives til ved systemkall, selv når de ikke er mappet til en region. Mappede segmenter er vanligvis demand-paged. Når en prosess først refererer til et nylig mappet segment vil man få en sidefeil og segmentsiden som tilhører

denne adressen vil bli hentet inn. På denne måten kan vanlig virtuelt minne implementeres, og i tillegg kan en prosess gjøre filer tilgjengelige i det virtuelle minnet så den kan få tilgang til filene direkte uten å skrive og lese dem ved hjelp av systemkall.

3.5 Kjernestruktur

Vi skal her se på hvordan kjernen til Chorus er strukturert internt. Kjernen består av fire komponenter som vist i figur 9.



Figur 9: Kjernens struktur i Chorus

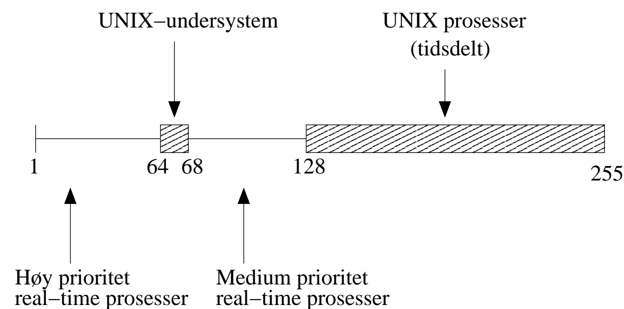
I bunnen av kjernen ligger supervisoren. Supervisoren behandler kommunikasjon med maskinens maskinvare, traps, exceptions, interrupter og context switching. Den er skrevet delvis i assembler for optimalisering, og må skrives om hvis den skal kunne kompileres for ny maskinvare. Over supervisoren finner vi den virtuelle minnebehandleren, som behandler lavnivådelen av sideutbyttingssystemet. Den største delen av denne modulen jobber med cache-sideutbytting og andre logiske konsepter, og er maskinuavhengig. En liten del av den må vite hvordan den skal laste og lagre MMU-registrene. Denne delen er maskinavhengig og må modifiseres hver gang Chorus skal implementeres på en ny plattform. Disse to delene av den virtuelle minnebehandleren gjør ikke alt arbeid med behandling av sideutbytting, en tredje del (mapperen) er utenfor kjernen og tar seg av høynivådelen. Kommunikasjonsprotokollen mellom den virtuelle minnebehandleren og mapperen er godt definert, slik at brukere kan bruke egne spesialiserte mappere.

Den tredje delen av kjernen er sanntidsbehandleren. Den er ansvarlig for behandling av prosesser, tråder og schedulering. Den tar også hånd om synkronisering av tråder. Til slutt har vi "interprocess communication manager", som tar seg av capabilityer, porter og sending av meldinger på en transparent måte. Den bruker tjenestene til sanntidsbehandleren og den virtuelle

minnebehandleren for å utføre jobben sin. Denne modulen er totalt uavhengig maskinvaren og trenger ikke å forandres når Chorus skal over på en ny plattform. Alle disse fire delene av kjernen er konstruert modulært, så endringer i en modul har vanligvis ingen effekt på de andre.

3.6 Sanntidsapplikasjoner

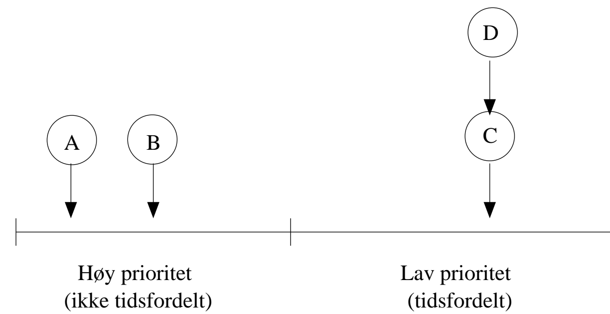
Chorus er designet for å kunne håndtere sanntidsapplikasjoner, både med og uten UNIX-undersystem. Prioriteter i Chorus er rangert fra 1 til 255. Jo lavere nummer, jo høyere prioritet vil prosessen få, som i UNIX. Vanlige UNIX-applikasjoner kjører med prioritet 128 til 255, som vist i figur 10. Ved disse prioritetsnivåene får hver prosess tildelt en tidskvote, og når en prosess har brukt opp denne tidskvoten, stoppes den og puttes bakerst i køen for sitt prioritetsnivå.



Figur 10: Prioritetsoppdeling

Prosessene som utgjør UNIX-undersystemet kjører med prioritet 64-68. På denne måten er et stort nummer av prioriteter, både lavere og høyere enn UNIX-undersystemet, tilgjengelig for sanntidsprosesser. Prosesschedulering av sanntidsprosesser gjøres ved å bruke prioriteter for hver enkel tråd. Hver prosess har en prioritet, og trådene i denne har en relativ prioritet i forhold til prosessen. Den absolutte prioriteten til en tråd er summen av prosessens prioritet og dens egne relative prioritet. Kjernen holder rede på prioriteten til hver tråd som er aktiv, og kjører tråden med den høyeste absolutte prioritet. På en flerprosessormaskin med N prosesser, vil trådene med de N høyeste prioritetene bli kjørt.

For å kunne tilpasse seg sanntidsprosesser, er Chorus utvidet med en ekstra algoritme. Algoritmen skiller mellom tråder som er over og under en gitt verdi. Trådene som er under denne verdien får høy prioritet, og de



Figur 11: Prioritetsrekkefølge

som er over får lav prioritet. Høyprioritetstråder, som A og B i figur 11, er ikke jevnt tidsfordelt. Med en gang en slik prosess (B) starter å kjøre, så kjører den til den frivillig gir fra seg prosessoren, til den må vente på I/O-operasjoner eller en prosess med høyere prioritet (A) blir aktiv. Den blir altså ikke stoppet selv om den har kjørt lenge. Når A og B er avsluttet eller ikke er i aktiv status, vil C kjøre. Men etter den har brukt opp tidskvoten, blir den puttet bakerst i køen for sin prioritet, og D vil så bli kjørt. I kampen om prosessoren vil trådene C og D ha prosessoren vekselvis i gitte intervaller. Systemkall er tilgjengelig for å skifte prioriteter, slik at applikasjoner kan fortelle hvilke tråder som er de viktigste og hvilke som ikke er så viktige. Denne mekanismen er generell nok for de fleste sanntidsapplikasjoner. Flere algoritmer for schedulering er tilgjengelig i Chorus for å støtte System V sanntid og systemprosesser.

4 NetBSD

NetBSD er et UNIX-lignende operativsystem tilgjengelig for mange forskjellige typer plattformer. NetBSDs rene design og avanserte egenskaper gjør den godt egnet i både produksjon og forskningsmiljøer.

Den første versjonen av NetBSD ble utgitt i 1993, og baserte seg på kildekode fra operativsystemet 4.3BSD-Lite og 386BSD. 4.3BSD-Lite er en versjon av UNIX utviklet ved Berkleyuniversitetet i California, og 386BSD er den første versjonen av BSD portet til Intels 386-prosessor. I de følgende år har modifikasjoner fra 4.4BSD-Lite blitt integrert inn i NetBSD. BSD-grenen av UNIX har vært viktig for, og påvirket utviklingen av NetBSD. Den har gitt NetBSD mange verktøy, idéer og forbedringer. Eksempler på dette er Berkley Fast File System, støtte for virtuelt minne og TCP/IP-implementasjon, som nå er standard i alle UNIX-systemer.

NetBSD-prosjektets fokus ligger i :

- Ren design.
  NetBSD fokuserer på ren og oversiktlig kode. På grunn av dette kan NetBSD støtte allerede eksisterende egenskaper senere enn andre systemer. Men etter hvert som tiden går vil koden bli lettere å håndtere, mens andre systemer som foretrekker nye egenskaper ovenfor kodekvalitet vil ha økende problemer med å håndtere og å unngå konflikter i koden.

- Bred plattformstøtte.
  En av hovedfokusene til NetBSD-prosjektet har vært å lage operativsystemet ekstremt portabelt. Dette har resultert i at NetBSD er portet til et vidt spekter av plattformer. NetBSD er designet for å utnytte den nyeste maskinvaren som er tilgjengelig for Alpha, PPC og PC-systemer, samtidig som den støtter eldre arkitekturer. Det gjøres en stor innsats for å holde en kategorisering av områder som er maskinavhengige, og områder som er maskinuavhengige i kildekoden. På denne måten får man en god oversikt, som gjør porting til andre typer plattformer enklere. Eksempler på plattformer som støttes av NetBSD er Digital Alpha, Commodore Amiga, i386 systemer, Apple Macintosh, Sun SPARC og Sun 3.

- Fritt tilgjengelig.
  NetBSD har full kildekode tilgjengelig, som kan hentes ned slik at flere personer skal kunne studere, videreutvikle og spesialdesigne NetBSD

slik de ønsker. Slik beskriver NetBSD sitt prosjekt : "The NetBSD Project provides a freely available and redistributable system that professionals, hobbyists, and researchers can use in whatever manner they wish... Also if you need a Unix system which runs consistently on a variety of platforms, NetBSD is probably your best (only) choice."

– Sikkerhet.
NetBSD er et av operativsystemene som det er rapportert minst feil på i åpne diskusjonsforumer [48]. Siden kildekoden er åpen, jobber flere personer sammen for å forsikre seg om at kildekoden til NetBSD er sikker og fri for feil.

– Stabilitet.
NetBSD er bygget på kode som stammer tilbake til de tidlige 80-årene på UC Berkeley, og har siden da vært tilgjengelig for korrigering av feil og mangler. Den rene designen til NetBSD skaper også mindre sjanser for feil og bidrar til økt stabilitet. NetBSD brukes i dag av NASA, der de krever en god og stabil kodebase som de kan bygge prosjekter på.

4.1 Kjernen

Kjernen til NetBSD er monolittisk. En monolittisk kjerne ble valgt for at kjernen skulle være enkel og ha god ytelse. I begynnelsen var kjernen liten, men innføring av komponenter som nettverksfunksjonalitet i kjernen har økt størrelsen. Nettverksfunksjonalitet tar i dag opp 35% av kjernen, og på grunn av dette, er det flere og flere som velger å legge nettverksfunksjonalitet utenfor kjernen.

Maskinavhengige aspekter av kjernen er isolert fra hovedkoden. Ikke noe av den maskinuavhengige koden inneholder egen kode for forskjellige typer arkitekturer. Når en maskinavhengig handling må utføres, kaller den maskinuavhengige koden en funksjon som finnes i den separate maskinavhengige koden.

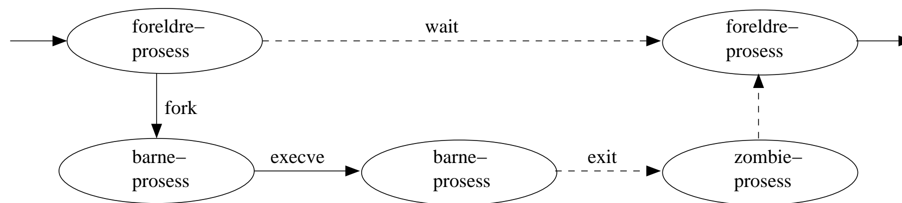
Grensen mellom kjerne og brukerkode overholdes av underliggende maskinvare. Kjernen opererer i et eget adresserom, som ikke er tilgjengelig for brukerprosesser. Privilegerte operasjoner, som å starte I/O-operasjoner, er tilgjengelig kun for kjernen. Brukerprosesser må bruke tjenester fra kjernen, gitt ved et sett lovlige systemkall. Parametre som sendes til systemkallene vil bli kontrollert av kjernen før de blir brukt. Denne kontrollen sørger for at ingen ulovlige parametre brukes, og sikrer integriteten til systemet.

Alle parametre blir også kopiert inn i kjernens adresserom, for å sørge for at de kontrollerte parametrene ikke endrer seg. Resultatet av systemkallet blir returnert til brukerprosessen, enten i registre, eller i brukerprosessens adresserom.

4.2 Prosesshåndtering

Hver prosess og tråd i NetBSD kalles for en prosess. Brukere i NetBSD kan opprette NetBSD-prosesser, ha kontroll over prosessens utførelse og få informasjon ettersom prosessens status endres. Hver prosess har en unik identifikator (PID). Denne verdien brukes av kjernen for å identifisere en prosess når den skal rapportere om prosessens status til bruker, og når brukeren skal referere til en prosess i et systemkall.

Kjernen oppretter prosesser ved å lage en kopi av innholdet til en annen prosess. Den nye prosessen kalles for et barn av den originale foreldreprosessen. Livssykelen til en prosess er beskrevet i figur 12. En prosess kan opprette en ny prosess, som er en kopi av den originale, ved hjelp av systemkallet fork. Selv om man noen ganger vil at prosessen skal være en kopi av foreldreprosessen, ønsker man ofte at den nye prosessen skal kjøre et annet program. En prosess kan derfor kopiere et annet program over sin egen kode ved hjelp av systemkallet execve. Prosessen kan velge å terminere utførselen ved å sende exit status til sin foreldreprosess.



Figur 12: Prosesshåndtering i NetBSD

Prosesser scheduleres for utførelse ut fra prosessens prioritet. Prioriteten prosessen starter med er bestemt av kjernens scheduleringsalgoritme. Brukere kan senere påvirke scheduleringen til en prosess ved å endre prosessens prioritet gjennom spesielle systemkall. Men det er fortsatt kjernen som bestemmer den underliggende scheduleringsalgoritmen.

4.3 Minnehåndtering

Hver prosess har sitt eget private adresserom. Adresserommet er delt opp i tre logiske segmenter, som er tekst, data og stack. Tekstsegmentet er kun leselig og inneholder maskininstruksjonene til programmet. Data og stacksegmentet er både leselige og skrivbare. Datasegmentet inneholder både initialisert og uinitialisert data som tilhører programmet, mens stacken holder på programmets runtimestack.

Hele prosessens adresserom behøver ikke å ligge i minne. Hvis en prosess refererer til et område av adresserommet som ikke ligger i minne, vil systemet page dette området inn i minne. Når minneressurser er knappe, vil kjernen først prøve å ta over minneområder fra andre prosesser, dersom de har minneområder som ikke er brukt på en stund. Hvis ingen slike minneområder finnes, vil kjernen swappe ut hele innholdet til en prosess ut på disk for å få mer plass.

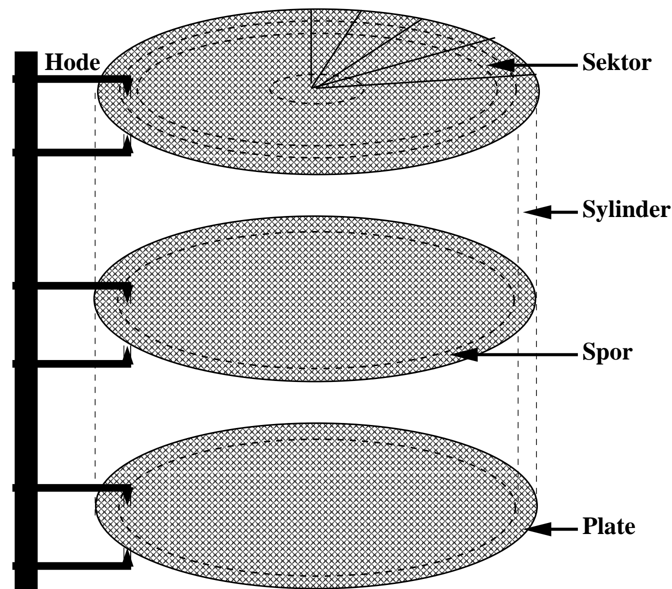
NetBSDs minnebehandler baserte seg tidligere på den gamle Mach-baserte, 4.4BSD virtuelle minnebehandleren. Den er senere blitt byttet ut med et nyere system for minnebehandling, kalt UVM. UVM er et nytt virtuelt minnesystem spesielt designet for å gi I/O og IPC-systemer et bredt spekter av fleksible mekanismer for flytting av data. UVM har gjort den gamle virtuelle minnebehandleren (4.4BSD VM) overflødig. UVM ble utviklet med fokus på hastighet, og har bedre ytelse i forhold til andre virtuelle minnehåndteringssystemer [49].

5 Disk

For å forstå faktorene som spiller inn på diskens ytelse, er det viktig å vite hvordan data er lagt ut på disken. Vi skal i dette kapittelet se på komponenter som kan spille inn på hastigheten til overføring av data fra disk til minne. Først ser vi på hvordan data er lagret på disken. Deretter ser vi på scheduleringsalgoritmer, bufferhåndtering, filsystemer og andre faktorer som påvirker diskens ytelse.

5.1 Strukturering av data på disken

En disk inneholder et sett av hurtige roterende plater, som på begge sider er belagt med et magnetisk belegg. På hver side av platene er det et lesehode som kan lese og skrive data i sirkulære spor. Lesehodene på disken er koblet sammen slik at de alltid er i samme sylinder, og i de fleste disker er det bare ett lesehode som er aktivt av gangen.



Figur 13: Oppbygning av disk

Den minste enheten som det er mulig å lagre på en disk er en sektor, som vanligvis har plass til 512 bytes data og feilkorrigeringsdata [4]. En sylinder er et sett av sektorer fra hver plate, som ligger i en gitt avstand fra platens sentrum. For eksempel inneholder den innerste sylinderen de innerste sporene

på hver plate. En fysisk posisjon på disken kan beskrives ved en overflate, en sylinder og en sektor.

5.2 Grensesnitt mot disk

De fleste disker blir i dag produsert med et eget kort på disken, som kontrollerer disken, for å gi et relativt enkelt grensesnitt mot disken. Protokoller som SCSI (Small Computer System Interface [15]) og IDE (Integrated Drive Electronics [10]) er velkjente protokoller som brukes av mange disker. SCSI er den mest fleksible protokollen, og den som støtter de raskeste hastighetene. Ultra2Wide-SCSI er en vanlig standard, som tillater opp til 15 disker i en SCSI-kjede, med en hastighet opp til 80 MB/s. En vanlig IDE-standard er ATA66, som tillater opp til fire disker og en hastighet opp til 66 MB/s. Ulempen med SCSI-disker er at de er relativt dyre, ofte opp til tre ganger så dyre som en IDE-disk. I tillegg må man ha et SCSI-kontrollerkort, i motsetning til IDE, hvor kontrollerkortet ofte er innebygd i hovedkortet. Hastigheten for kontinuerlig overføring av data fra selve disken er den samme for begge systemene, men hvis man skal bruke diskene i en RAID-konfigurasjon vil SCSI være den klart raskeste [14]. I SCSI og IDE-protokollene blir forespørsler etter data gitt ved et startende logisk blokknummer, og størrelsen på forespørselen. Detaljene i forespørslene ligger gjemt i elektronikken på disken, og avlaster håndtering av plasseringen av data på disken fra operativsystemet.

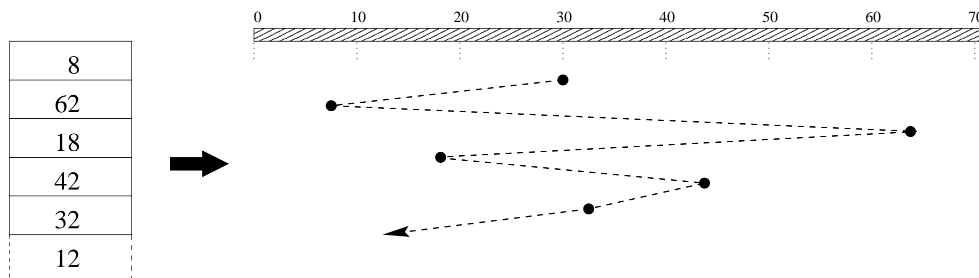
5.3 Scheduleringsalgoritmer

Forespørsler etter data fra disken kan bli forsinket på grunn av mekaniske forsinkelser. For det første må lesehodet flytte seg (søkeforsinkelse) til den sylinderen det skal leses fra. For det andre må man vente på at disken skal rotere rundt til lesehodet kommer til den sektoren det skal leses fra (posisjonsforsinkelse). For de fleste disker er det søkeforsinkelsen som tar lengst tid [29], så en reduksjon i gjennomsnittlig søkeforsinkelse kan øke diskytelsen betraktelig. I tillegg kommer mekaniske forsinkelser som oppstår når man skal lese flere spor eller sylindere av gangen. For å minimere den totale mekaniske forsinkelsen (søke og posisjonsforsinkelse), finnes det flere scheduleringsalgoritmer som dynamisk ordner køen av ventende diskforespørsler.

5.3.1 First Come First Served

First Come First Served (FCFS) er den enkleste algoritmen for håndtering av køen av forespørsler etter data på disken. FCFS godtar én forespørsel

om gangen, og utfører dem i samme rekkefølgen som de kommer inn. FCFS-diskschedulering resulterer ofte i dårligere ytelse [29] enn andre algoritmer, da det er lite som gjøres for å optimalisere ytelsen til disken. Den er lett å programmere, gir en god fordeling og sørger for at ingen må vente for lenge på en forespørsel. Ulempen med denne metoden er at den ikke er optimal med hensyn på strekningen lesehodet på disken tilbakelegger, da den ikke tar hensyn til andre forespørsler i køen. Den gjennomsnittlige søketiden i et system som bruker FCFS er som regel veldig høy.

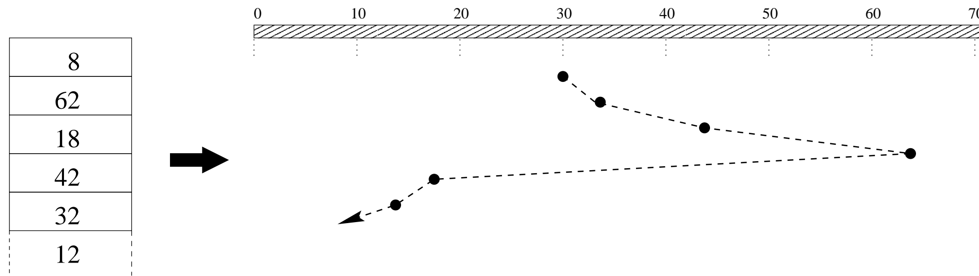


Figur 14: FCFS

I figur 14 ser vi forespørslene komme inn i køen til venstre. Til høyre ser vi strekningen lesehodet tilbakelegger ved lesing av blokkene i køen. Figuren viser det som er typisk for FCFS, lesehodet beveger seg unødvendig mye frem og tilbake og skaper store forsinkelser.

5.3.2 Shortest Seek Time First

SSTF er en algoritme som behandler forespørselen som til en hver tid er nærmest lesehodet. Søketiden er minimert, og algoritmen er optimal med hensyn på total søketid. Ulempen er at den ikke er rettferdig, da den fører til at noen forespørsler må vente i lang tid på andre. Når det gjøres forespørsler etter flere blokker på to forskjellige områder på disken, vil den som er nærmest lesehodet få første blokk. Deretter vil hodet holde seg på den delen av disken som den startet på, da blokker fra den andre delen aldri vil være nærmest lesehodet. Dette fører til at et område på disken får veldig høy prioritet, mens de andre må vente til det ikke er flere blokker som skal hentes fra dette området. Et annet problem er at lesehodet vil prioritere midten av disken i forhold til de innerste og ytterste områdene av disken.



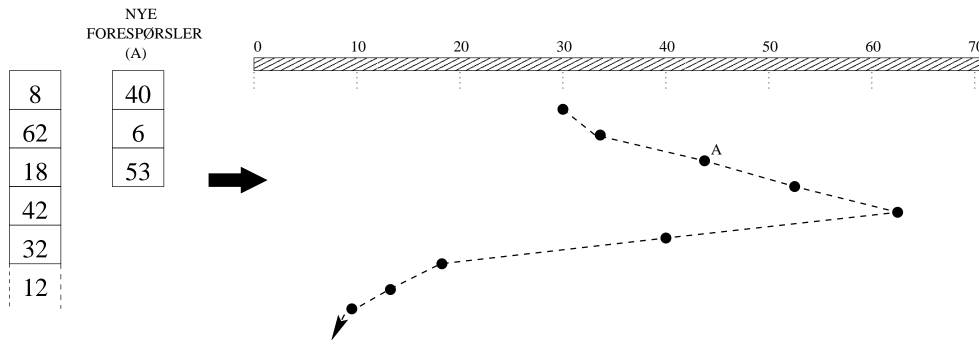
Figur 15: SSTF

I figur 15 ser vi en forespørsel med den samme køen som i FCFS (figur 14). Vi ser her at veien lesehodet tilbakelegger er betraktelig redusert i forhold til FCFS. En annen ting vi kan legge merke til er at blokk 8 som ligger først i køen blir hentet inn sist.

5.3.3 SCAN

SCAN-schedulering minimerer søketiden ved å sortere forespørslene etter retningen diskhodet forflytter seg. Først behandler den alle forespørslene i én retning inntil det ikke er flere forespørsler igjen i denne retningen. Så forflytter diskhodet seg i motsatt retning til det er tomt for forespørsler i den andre retningen. Slik fortsetter den til det ikke er flere forespørsler igjen. Denne metoden fungerer bra for henting av data som ikke er tidskritiske, men egner seg ikke for kontinuerlig henting av tidskritiske data, som f.eks. filmdata. I noen tilfeller vil en forespørsel måtte vente lang tid før den kan få blokken den har spurt etter. La oss si at diskhodet forflytter seg oppover og holder på å lese blokk 2. Det kommer så inn en forespørsel etter blokk 1, men denne kan ikke behandles før lesehodet har gått gjennom alle forespørsler frem til andre enden av disken og så tilbake igjen. Dette fører til at man må regne med å vente i opptil 2 ganger den tiden det tar å sveipe gjennom disken før man kan få blokken.

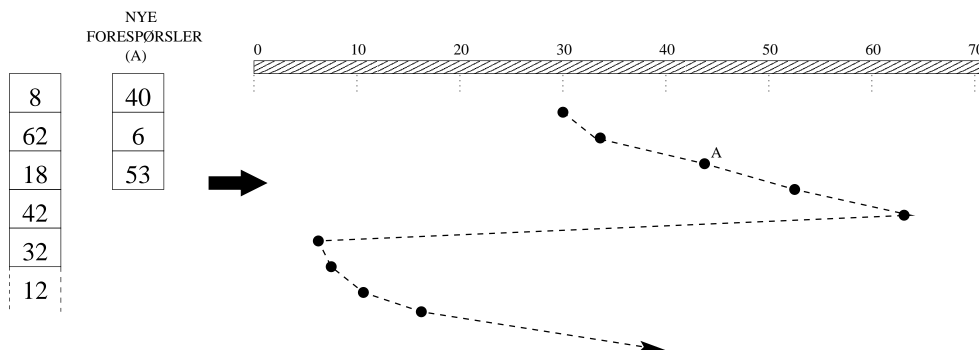
I figur 16 ser vi den samme køen som i FCFS, og et lesehode som forflytter seg mot høyre. Her kommer det inn nye forespørsler når lesehodet er kommet til punkt A. Disse forespørslene er vist i køen til høyre for den opprinnelige køen. Vi ser her at det kommer inn en forespørsel etter ny blokk 40, som er nærmest lesehodet på dette tidspunktet. Selv om denne er nærmest lesehodet vil den ikke bli hentet inn, for lesehodet skifter ikke retning så lenge det er flere blokker igjen i den retningen lesehodet beveger seg.



Figur 16: SCAN

5.3.4 C-SCAN

C-SCAN fungerer på samme måte som SCAN, foruten at diskhodet kun leser data mens den er på vei oppover eller nedover. Hvis diskhodet leser data på vei oppover, og har lest inn blokken med det høyeste nummeret i køen, vil den gå tilbake til den blokken i køen med det laveste nummeret uten å lese noen blokker på veien ned. Så vil den fortsette å lese blokkene i stigende rekkefølge, inntil den igjen kommer til blokken med det høyeste nummeret. I forhold til SCAN-algoritmen er den største mulige ventetiden for en blokk kortere, mens den totale søketiden er høyere. Dette gjør C-SCAN bedre egnet enn SCAN for henting av data som er tidskritiske, f.eks. filmdata.



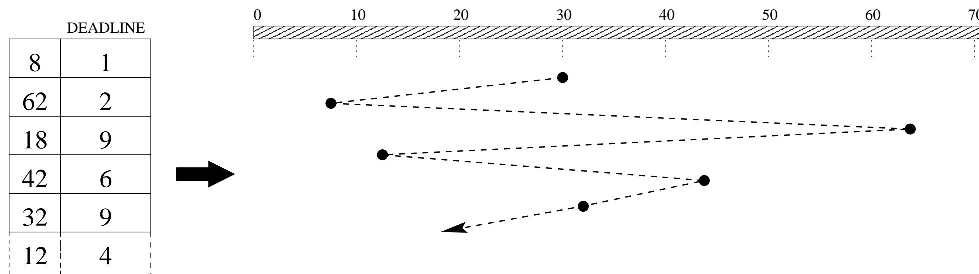
Figur 17: C-SCAN

I figur 17 ser vi de samme køene som i figur 16. Her ser vi at lesehodet ikke leser noen blokker fra den har hentet blokk 62 til den er kommet tilbake til blokk 6. En annen ting vi kan legge merke til er at blokk 8, som er den første

blokken i køen, ikke må vente på alle de andre før den blir behandlet, som den måtte i SCAN.

5.3.5 Earliest Deadline First

I multimediasystemer har man ofte gitte frister for når data må være tilgjengelige. I EDF blir en forespørsel gitt en deadline, og den forespørselen med kortest tid til deadline vil bli behandlet først. Denne algoritmen er ideell med tanke på sanntidsbehovet til kontinuerlige datastrømmer. Men i praksis viser det seg at den resulterer i en lav gjennomstrømning og høy søketid, da den ikke tar hensyn til retningen og posisjonen diskhodet har før neste blokk skal leses.



Figur 18: EDF

I figur 18 ser vi samme køen som i FCFS, men her har blokkene en prioritet knyttet til seg. Dette fører til at blokk 8, med kortest tid til deadline, vil bli behandlet først, mens blokk 32 og 18 som har lengst tid til deadline vil bli behandlet sist. Vi ser her en kø med tilhørende deadline som fører til at lesehodet beveger seg unødvendig mye frem og tilbake.

5.3.6 Position Delay Reduction

De algoritmene vi har beskrevet til nå har som mål å redusere søketiden. Reduksjon av den gjennomsnittlige søketiden krever bare kunnskap om den relative søkedistansen mellom de forskjellige forespørslene. For å også kunne ta hensyn til den tiden det tar for disken å rotere til den etterspurte blokken, trengs eksakt informasjon om plassering av datablokker på disken. I tillegg må man vite hvor diskhodet befinner seg. Ut fra disse opplysningene kan scheduleren velge den forespørselen med minst posisjonsforsinkelse (rotasjon

og søkeforsinkelse). Denne metoden har navnet Shortest Time First (STF).

For å minimere rotasjonsforsinkelsen er det lurt å legge blokker som tilhører samme fil sekvensielt etter hverandre. Når en fil blir lest sekvensielt er det en tidsforsinkelse fra blokken er lest til kjernen er klar for å lese inn neste blokk. Fordi disken roterer mens kjernen leser inn blokken, kan en eller flere sektorer ha passert lesehodet. Optimalisering av rotasjonsforsinkelse prøver derfor å finne ut antall sektorer man skal hoppe over slik at hodet er under den ønskede sektor ved neste lesing.

Ettersom diskteknologien utvikler seg, blir søkehastigheten stadig raskere, mens rotasjonshastigheten ikke øker i samme grad [16]. På noen disker er gjennomsnittlig søketid allerede mindre enn rotasjonsforsinkelsen. Hvis denne trenden fortsetter vil man tjene mer på å ta i bruk algoritmer som tar hensyn til rotasjonsforsinkelse.

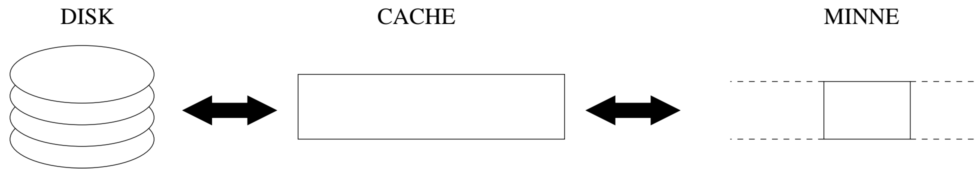
Valg av scheduleringsalgoritme avhenger av hvordan disken blir brukt. For disker som blir mye brukt er SCAN/LOOK algoritmene passende, da de tar hånd om forespørslene i en god rekkefølge. Det er ikke noen fare for at noen diskforespørsler skal bli nedprioritert så mye at de aldri får tilgang til disken. Hvis disken ikke er hardt belastet, kan EDF brukes for å sikre raskere tilgang for tidskritiske data.

5.4 Bufferhåndtering

Diskoperasjoner er en stor flaskehals i dagens maskiner. Tiden det tar å lese en blokk fra disk er i en størrelsesorden av noen få millisekunder. Tiden det tar å kopiere den samme mengden data fra et minneområde til et annet er i størrelsesorden av et par mikrosekunder. Kopiering fra minne til minne er over 100 ganger raskere enn å kopiere fra disk til minne. Det er derfor mye å tjene på å minimere antall diskoperasjoner, som kan gjøres ved å bruke bufferhåndtering. Bufferhåndtering er nødvendig i et filsystem av to grunner. For det første skal den sørge for at data blir hentet fra disk til minne for manipulering eller for å bli sendt ut på nettverket. For det andre skal den virke som en cache for nylig brukte diskblokker.

Flere typer algoritmer kan brukes for å behandle cachen. En vanlig algoritme er å sjekke alle forespørsler mot disk for å se om de er tilgjengelige i cachen. Hvis den er det, så returneres blokken fra cachen uten at det leses fra disken. Hvis den ikke er det, så leses den først inn i cachen, og blir så kopiert til den som spurte etter blokken. Hvis det senere kommer forespørsler

etter den samme blokken, kan blokken hentes direkte fra cachen. Data som skrives til disk blir også lagt i cache, slik at hvis det gjøres mange endringer på samme blokk, holder det å skrive blokken til disk kun en gang.



Figur 19: Cache

Når en blokk skal hentes inn og cachen er full, må en blokk fjernes for å få plass til den nye, og eventuelt skrives til disk hvis den er endret. Spørsmålet er hvilken blokk som skal skrives til disk. En mulighet er å skrive ut den blokken som har vært i cachen lengst uten å ha vært brukt. Ulempen med å ikke skrive data til disk med en gang, er at man ofte har data som er kritiske, og ikke burde ligge i cachen i tilfelle systemet går ned. Det gjelder f.eks. blokker som inneholder filstrukturen til filsystemet. Hvis maskinen går ned mens det ligger blokker som inneholder filstrukturen i cachen kan filsystemet bli inkonsistent. Cachen tar derfor også hensyn til:

1. Hvor sannsynlig det er for at blokken skal brukes igjen?
2. Er blokken viktig for å holde filsystemet konsistent?

Ut i fra disse punktene kan cachen lage en liste over hvilken blokk som skal skrives ut fra cachen. Er blokken viktig for filsystemet vil den bli plassert øverst på listen. Er blokken brukt i det siste, er det stor sannsynlighet for at den skal brukes igjen, og vil derfor bli plassert lenger ned på listen.

For at ikke blokker skal ligge i cachen til evig tid uten å bli skrevet til disk finnes det flere strategier. Den ene som UNIX bruker er at det startes en prosess under oppstart, som periodisk kaller funksjonen sync. Funksjonen sync sørger for at alle blokkene som er endret i cachen blir skrevet til disk. Den andre metoden, som blant annet MS-DOS brukte, var å skrive alle endringer i cachen rett til disk (write-through cache). Fordelen med write-through cache er at dataene lagres med en gang, som er fordelaktig når man f.eks. lagrer filer på flyttbare medier som diskettstasjonen. Ulempen er at den bruker mye ressurser på I/O-operasjoner.

5.5 Read-ahead

I en videoserver leses data fra filer inn sekvensielt. En metode for å gjøre sekvensiell lesing av blokker raskere er read-ahead. En read-ahead på en blokk er det samme som å lese en blokk, foruten at prosessen returnerer uten at den venter på at blokken skal bli lest.

Når en blokk skal leses, kan man først gjøre read-ahead på neste blokk. Prosessen vil så få kontrollen igjen, selv om blokken ikke er lest inn. Så kan prosessen hente inn blokken før read-ahead blokken. Read-ahead blokken blir så hentet inn sammen med blokken, og er klar når applikasjonen skal hente inn neste blokk. I mange operativsystemer gjøres dette i ett kall, slik at en forespørsel etter en blokk resulterer i forespørselen, inkludert en read-ahead på neste blokk. Dette systemkallet vil ta den originale og read-ahead blokken som parametere, og virker som beskrevet under.

1. Hvis den originale blokken ikke er i cache, gjør plass for den i en buffer og send en forespørsel etter diskblokken til diskdriveren.
2. Hvis read-ahead blokken ikke er i cache, gjør plass for den i en buffer og send en forespørsel etter diskblokken til diskdriveren.
3. Hvis den originale blokken er i cache, returner blokken.
4. Hvis ikke, vent på diskdriveren. Kallet blokkeres inntil blokken er lest fra disk.
5. Når blokken er lest vekkes prosessen og blokken returneres.
6. Prosessen fortsetter til den skal lese neste (read-ahead) blokk. Blokken returneres da fra cachen.

Det er også mulig å gjøre read-ahead på flere enn en blokk av gangen. Dette kan lønne seg i en videoserver hvis man har mye minne til rådighet for cachen. Er det sparsomt med minne og flere prosesser henter inn mange blokker samtidig, kan man risikere at read-ahead blokkene blir kastet ut fra cachen før man får lest dem.

5.6 Henting av buffere

Kompleksiteten i henting av buffere ligger ofte i at det er flere prosesser som konkurrerer om den samme diskblokken, som er representert ved en unik

buffer. Flere prosesser har også lov til å skrive til det samme bufferet samtidig under kontroll av kjernen. Prosedyren for buffertildeling i filsystemet er som følger:

1. En forespørsel etter en blokk fra en prosess er sendt til kjernen.
2. Kjernen allokerer en buffer for diskblokken, stenger bufferet for andre og returnerer bufferet til prosessen som spurte etter den. Hvis kjernen ikke klarer å allokere en buffer, går prosessen i søvn for å bli vekket opp senere.
3. Den etterspurte operasjonen på bufferet blir utført. Når den er utført blir bufferet returnert til kjernen, og kjernen åpner bufferet for andre. Kjernen sørger også for at bufferet blir frigjort.

Kjernen putter en prosess i søvn ved to tilfeller. Det ene tilfellet er når blokken ikke er funnet i buffer-cachen og listen over ledige buffere er tom. Det andre tilfellet er når blokken er funnet i buffer-cachen, men er opptatt. Prosessen må gå i søvn inntil andre prosesser ikke lenger bruker bufferet. Kjernen må derfor sørge for å vekke opp de prosessene som venter på ledige buffere.

5.7 Ulemper ved bruk av cache

Det er flere ulemper ved å bruke cache. For det første gjør forsinket skriving til disk maskinen sårbar når den går ned. En blokk i cachen representerer en fil, men dataene i blokken er ikke nødvendigvis skrevet til disk når maskinen går ned. Forsinket skriving for kritiske data i superblokker og i-nodeblokker kan føre til at filsystemet kommer i en inkonsistent status. Noen UNIX-varianter skriver cache til disk periodisk. Diskhastigheten blir dermed redusert, men det sikrer dataenes integritet til en viss grad.

For det andre, fører bruk av cache til at ekstra data må kopieres mellom kjernens og brukerens adresserom. Ved disklesing blir data lest fra disk til buffer-cache, og så kopiert til bufferet til applikasjonen. For overføring av store mengder data, som i en videoserver, kan ytelsen svekkes betraktelig.

5.8 Blokkstørrelse

Ytelsen til en disk er direkte avhengig av hvordan data lagres på disk. Det er to mulige måter å lagre data i filer på disk. Det kan settes av plass til hele filen på et område på disken, eller så kan filen splittes opp i flere blokker som

lagres forskjellige steder på disken. Ulempen med å plassere hele filen på ett sted, er at når filen vokser kan det gå tomt for plass der filen ble plassert. Disken må da omorganiseres for at filen skal få plass på disken, som resulterer i kostbare diskoperasjoner. På grunn av dette deler nesten alle filsystemer filer opp i en gitt størrelse (blokk), som kan legges hvor som helst på disken.

Når det er bestemt at man skal lagre filene i blokker, er spørsmålet hvor store blokkene skal være. Gitt måten data lagres på disken, er sektor, track og sylinder gode kandidater til blokkstørrelse. En stor blokkstørrelse på en sylinder fører til at selv en fil på én byte vil ta opp en hel sylinder. Hvis en sylinder er på 32 KBytes, vil et filsystem med gjennomsnittlig filstørrelse på 1 KByte kaste bort over 90% av disken. På den andre siden vil det å bruke små blokker dele store filer i mange små blokker. Lesing av mange små blokker fra disk påfører systemet ekstra tidsforbruk på diskrotasjon og søking etter blokker. Så lesing av en fil som består av mange små blokker vil redusere ytelsen. Det er her en konflikt mellom effektivitet i form av tid og plass.

I en videoserver er det tiden som er mest kritisk. Selv om man bruker store blokker i serveren, så vil det ikke gå mye plass tapt. Grunnen er at filene i en videoserver vanligvis er store. Hvis den gjennomsnittlige størrelsen på filene på serveren er på flere Mbytes, vil selv en blokkstørrelse på 32 KBytes gi ubetydelig med ubrukt plass. I en videoserver vokser ikke filene, det kan derfor være mulig å lagre filene sekvensielt på disken uten å splitte filene opp i blokker. På denne måten kan lesehodet holde seg i samme området på disken under lesing av samme fil. Dette vil redusere den totale søketiden for disken. En ulempe med å legge filene sekvensielt på disken er når det skal slettes og legges opp nye filer. Det oppstår da hull i disken der filene lå, og disken må omorganiseres hvis den nye filen ikke får plass i hullet. Hvis utbytting av filene på serveren skjer ofte, vil dette systemet fort bli tregt. Om det er lønnsomt å lagre filene sekvensielt eller i form av blokker, er altså avhengig av hvor ofte dataene på disken endres.

5.9 Filsystemer

Filsystemet er bygd opp av logiske strukturer og programrutiner, som fungerer som et grensesnitt mellom operativsystemet og kontrollen av harddisken. Hvilken type filsystem som brukes kan ha betydning for hvor raskt data kan skrives og leses fra harddisken, hvor bra lagringskapasiteten utnyttes og hvor stor harddisk som kan brukes.

Hvilket filsystem en datamaskin bruker, avhenger først og fremst av oper-

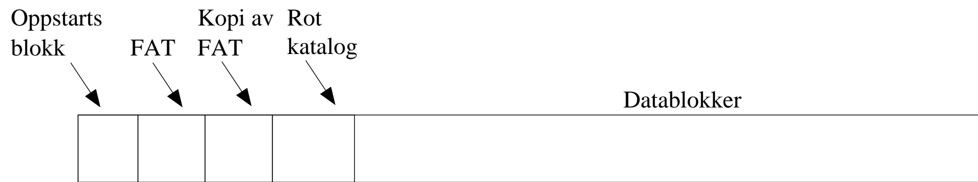
ativsystemet og ikke maskinvaren. En enkel datamaskin kan ha flere forskjellige typer harddisker, operativsystemer og filsystemer, fordi noen operativsystemer har støtte for mer enn ett filsystem.

Chorus støtter filsystemene NFS, FFS, MS-DOS og ramdisk. Ramdisk er den utvilsomt raskeste av disse, men er til gjengjeld mye dyrere enn disk. En annen ulempe med ramdisk er at all informasjon forsvinner når minnet ikke får tilførsel av strøm. Ramdisk er derfor uaktuelt som filsystem for en videoserver.

NFS (Network file system) er et filsystem utviklet for bruk over nettverk. Filsystemet er utviklet av SUN, og opprinnelig for bruk i UNIX-baserte arbeidsstasjoner. Hovedidéen bak NFS er å tillate en vilkårlig mengde klienter og servere å dele et felles filsystem. I de fleste tilfellene er alle klienter og servere på samme lokale nettverk, men det er ikke nødvendig. Den arkitektoniske karakteristikken til NFS er at servere eksporterer kataloger, og klienter kobler dem opp. Hvis to eller flere klienter kobler opp samme katalog samtidig, kan de kommunisere ved å dele filer i deres felles kataloger. Et program hos en klient kan opprette en fil, og et program på en annen klient kan så lese filen. Når katalogen er koblet opp trengs det ikke å gjøres noe spesielt for å kunne dele filene. Denne enkelheten gjør NFS attraktiv. Men hvis man skulle brukt NFS i en videoserver ville det kreve mye båndbredde, da filene først må overføres gjennom nettverket til applikasjonen på serveren, og så ut på nettet enda en gang til klienten.

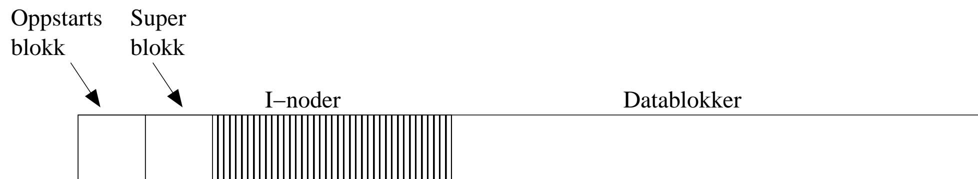
MS-DOS bruker et filsystem som ble utviklet tidlig på 80-tallet. Det lagrer filene på disken i blokker, som er en liten kontinuerlig del av disken. Blokkstørrelsen er gitt i oppstartssektoren, og kan være alt fra en til åtte sektorer. For å holde oversikt over hvilke filer som er i hvilke blokker, hvilke blokker som er brukt og hvilke som er ledige, har MS-DOS en FAT (File Allocation Table). FAT-tabellen inneholder én post for hver blokk på disken. For å sikre at denne informasjonen ikke går tapt har man en ekstra kopi av FAT-tabellen.

Alle disker som bruker FFS filsystemet er utformet som vist i figur 21. Den første blokken brukes ikke av UNIX, men er satt av for å brukes til kode for oppstart av maskinen. Superblokken inneholder kritisk informasjon knyttet til utformingen av filsystemet, inkludert antall i-noder, antall blokker og starten på listen over ledige blokker. Hvis superblokken ødelegges, vil hele



Figur 20: MS-DOS filsystem

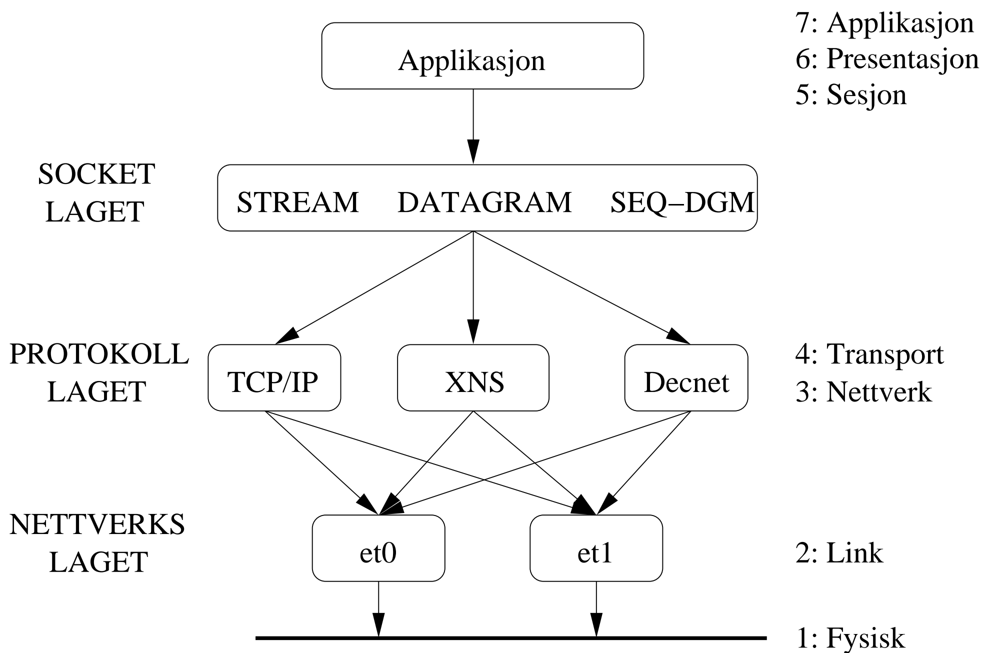
filsystemet bli uleselig. Etter superblokken ligger i-nodene, som er nummerert fra en og oppover. Hver i-node er 64 bytes stor, og inneholder beskrivelsen for en fil og filens posisjon på disken. Etter i-noden ligger datablokkene. Alle filer og kataloger er lagret i datablokkene. Hvis en fil tar mer plass enn én blokk, er det ikke nødvendig at blokkene ligger etter hverandre. Blokkene kan godt ligge spredt, og finnes ved å slå opp i tabeller i filens i-node.



Figur 21: Ufs filsystem

6 Nettverk

Dette kapittelet presenterer en oversikt over nettverksimplementasjonen i BSD (Berkeley Software Distribution), som Chorus og NetBSD baserer seg på. Beskrivelsen fokuserer på den interne strukturen og organisasjonen av IPC og de underliggende protokollene. Oversikten er nødvendig for å forstå hvordan protokollene fungerer, deres interaksjon med operativsystemet og for å identifisere køpunkter under en dataoverføring. IPC er i BSD-UNIX organisert i tre lag. Lagene er socket, protokoll og grensesnittlaget, som vist i figur 22. Til venstre i figuren ser vi de tre lagene i BSD-UNIX, og til høyre ser vi de tilsvarende lagene i OSI-referansemodellen.



Figur 22: Nettverkslag i BSD

6.1 Socketlaget

Socketlaget er et protokolluavhengig grensesnitt til det protokollavhengige laget under. Alle systemkall starter i det protokolluavhengige socketlaget. Socketlaget er derfor IPC-grensesnittet til applikasjonen. Hovedfunksjonene til socketlaget er:

- Identifisere datastrøm til forskjellige applikasjoner på samme maskin.
- Tilby buffer for innkommende og utgående datastrøm.

Socketlaget tilbyr applikasjoner socketer, som er abstrakte objekter som applikasjonen kan opprette for å sende og motta data. En socket representerer en ende i en kommunikasjonslink, og holder på all informasjon assosiert med linken. Denne informasjonen inkluderer hvilken protokoll som skal brukes, tilstandsinformasjon for protokollen, køer av innkommende forbindelser, databuffere og flagg for hvordan socketen skal oppføre seg. Det finnes flere typer socketer, som kan tilby forskjellige typer kommunikasjonsmetoder. En STREAM-socket tilbyr en sikker forbindelseorientert datastrøm. En DATAGRAM-socket tilbyr en forbindelseløs, usikker pakkeorientert dataforbindelse. I UNIX er en socket en type filbeskriver, slik at de samme operasjoner som kan utføres på filer, kan utføres på en socket.

6.2 Protokollaget

Protokollaget inneholder implementasjon av flere protokollfamilier. En protokollfamilie er en gruppe protokoller som hører sammen. Protokollene TCP, UDP og IP tilhører gruppen som heter internettprotokollene. Andre grupper som XNS og DECNET, inneholder egne protokoller som er funksjonelt like, men ikke kompatible med internettprotokollene. Avhengig av applikasjonens krav, kan applikasjonen velge de protokollene den ønsker fra en familie av protokoller.

6.3 Nettverkslaget

Nettverkslaget ligger rett over maskinens maskinvare, og består av maskinvaredrivere og deres grensesnitt, f.eks. ethernetdriveren og ethernetkortet. Nettverkslaget tilbyr et maskinvareuavhengig programmeringsgrensesnitt mellom nettverksprotokollene og driverne for nettverksenhetene som er koblet til systemet. Nettverkslaget tilbyr til alle nettverksenhetene:

- et definert sett av grensesnittfunksjoner
- et standardisert sett av statistikk og kontrollflagg for grensesnittet.
- en standard kømetode for utgående pakker.

Det er ikke noe krav om at nettverkslaget skal tilby en sikker overføring av pakker, bare en best mulig tjeneste. Høyere protokollag må kompensere for

mangelen på sikker overføring i nettverkslaget. En maskin kan være tilknyttet flere nettverk, og kan derfor ha mer enn ett nettverkskort. Protokollaget vil velge hvilket kort den skal sende ut pakker på, ut fra routingmekanismen i protokollen.

For å summere opp, så er nettverkskoden i BSD delt opp i tre lag. Socket laget som håndterer endepunkter som tilbyr et sett av tjenestetyper, protokollaget som består av protokollene som beskriver tjenesten og nettverkslaget som er en samling grensesnitt til nettverkene som maskinen er koblet til. Når en socket opprettes blir den bundet til en protokoll, som er bestemt av hva slags type socket som opprettes, og protokollen velger et nettverkskort som pakkene skal sendes ut på. Et eksempel på en slik konfigurasjon er en applikasjon som oppretter en sikker (STREAM) socket, i internettgruppen av protokoller, som velger TCP og IP-protokollen, som igjen velger å route pakkene over det lokale ethernetkortet.

I neste avsnitt i dette kapittelet beskriver jeg protokollene som jeg senere skal utføre nettverksmålinger på. Protokollene som brukes er TCP og UDP (med IP). Hastigheten til koden for sending av data fra applikasjon, gjennom disse protokollene og ut til nettverket er svært avhengig av hvordan data organiseres i koden. Flyten av data gjennom nettverkskoden i BSD baserer seg på bruk av mbuffere. Jeg vil derfor i kapittelet etter beskrive hva mbuffere er, hvordan de er oppbygd og hvordan de blir brukt. Til slutt i dette kapittelet vil jeg gå gjennom hvordan data flyttes gjennom koden til kjernen, på veien fra applikasjon og ut på nettverket.

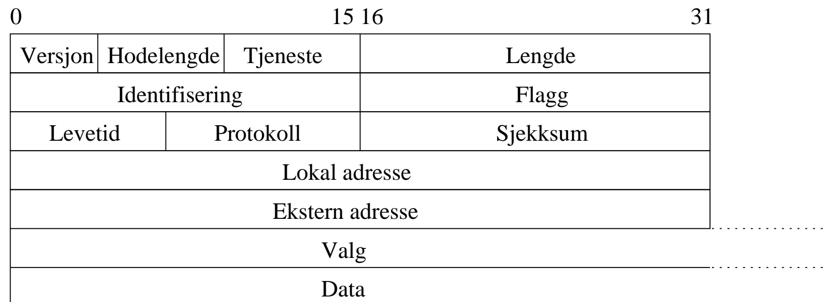
6.4 Internettprotokollene

I dette kapittelet skal vi se raskt på internettprotokollene IP, TCP og UDP. Vi starter med den enkleste protokollen IP, som TCP og UDP benytter seg av.

6.4.1 IP

Hvis alle maskiner var koblet til samme nettverk, ville alle kunne kommunisere direkte med alle de andre maskinene i nettverket. Uheldigvis er det mange forskjellige typer nettverk. Teknologien som knytter forskjellige nettverk sammen, til et virtuelt nettverk, kalles Internett. Internettjenesten (IP) er usikker, fordi leveranse av pakker ikke er garantert. Pakker kan mistes, dupliseres eller bli levert i feil rekkefølge. Tjenesten er kalt forbindelseløs, fordi hver pakke blir behandlet uavhengig av de andre. Forskjellige nettverks-

teknologier bruker forskjellig format på pakker og nettverksadresser er ikke globalt unike. Internettprotokollen løser disse problemene ved å definere en universell pakke kalt IP-datagram. Formatet til IP-datagrammet er vist i figur 23.



Figur 23: IP-hode

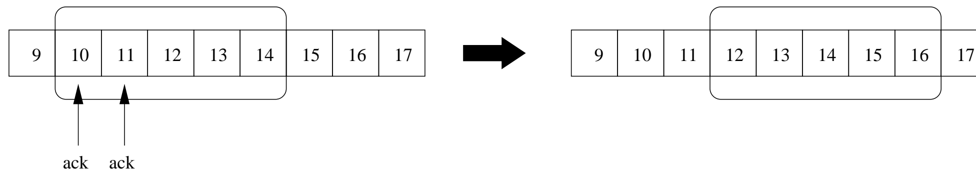
Datagrammene inneholder både sender og mottakers adresse. En 16-bit lang sjekksum gjør det mulig å oppdage feil i datagramhodet. Sjekksummen inkluderer ikke datadelen av datagrammet. Et 8-bit typefelt identifiserer hva slags type data datagrammet inneholder. Det 16-bit lange lengdefeltet beskriver lengden til datagrammet. Det tillater datagrammer opp til 65536 bytes, selv om de ofte er mye mindre siden forskjellige typer nettverksteknologier setter begrensning på størrelsen på pakkene. Hvis et datagram må sendes over et nettverk som ikke kan håndtere store pakker, kan IP-datagrammet derfor deles opp i flere mindre datagrammer. Et oppdelt datagram er fortsatt et komplett datagram, men mottakermaskinen må sette de oppdelte datagrammene sammen igjen før de leveres videre til høyere lag.

6.4.2 TCP

I forrige kapittel viste vi hvordan IP tilbyr et virtuelt pakkeorientert nettverk. TCP (Transmission Control Protocol) er en protokoll som omformer den datagramorienterte tjenesten til IP, hvor datagrammer kan mistes, dupliseres og leveres i feil rekkefølge, om til en fulldupleks sikker datastrøm. TCP retter opp i problemer som oppstår ved tap av pakker og sikrer integriteten til dataene.

TCP tilhører familien av protokoller som bruker sliding window, ved at den har et vindu som identifiserer data som senderen har sendt, men mottakeren ennå ikke har bekreftet mottatt. Når senderen mottar en bekreftelse på data som den tidligere ikke har fått bekreftelse på, skyves vinduet frem,

og nye data kan sendes. Figur 24. viser en sliding window protokoll, med fem pakker i vinduet. For å øke gjennomstrømningen av data kan sender øke størrelsen på vinduet, slik at det tillates flere ubekreftede pakker i forbindelsen på en gang. Gjennomstrømningen for sliding window protokoller er avhengig av vinduets størrelse, og tiden pakken bruker frem til mottaker, inkludert bekreftelsen av mottatt pakke.



Figur 24: Sliding window protokoll

En oppgave som alle transportprotokoller må ta hensyn til er flytkontroll. Ende-til-ende flytkontroll sikrer at mottakeren har nok bufferressurser til å kunne motta data sendt av senderen. TCP bruker flytkontrollvinduer for å løse dette problemet. TCP-hodet inneholder et vindusfelt, som spesifiserer hvor mange bytes mottakeren er klar til å motta. Hvis mottakeren går tom for buffere, setter den vindusfeltet til 0, og senderen slutter å sende data. Når mottakeren så har behandlet data, setter den opp størrelsen på vindusfeltet, og senderen kan begynne å sende igjen.

En av de viktigste oppgavene til TCP, er at mottakeren sender tilbake en bekreftelse på de pakkene som er levert korrekt, og senderen sender pakkene på nytt når den ikke har mottatt bekreftelse på pakken. Dette fungerer ved at senderen starter en tidtaker, som starter å telle ned fra en gitt tid når den sender ut pakken. Hvis tiden går ut, uten at det er mottatt bekreftelse på pakken, sendes pakken på nytt. Problemet er hvor lang denne tiden skal være. Er den for kort, vil senderen sende forsinkede pakker på nytt, selv om de ikke er tapt. Er den for lang, vil gjennomstrømningen av data synke når en pakke er tapt, fordi senderen må vente på at tiden skal gå ut før den kan sende på nytt. For å løse dette problemet har TCP et dynamisk estimat av hvor lang tid det tar for en pakke fra den blir sendt til bekreftelsen er mottatt. Dette er RTT (Round Trip Time). Fordi RTT varierer mye, har TCP en SRTT (Smoothed Round Trip Time), som beregner gjennomsnittlig RTT fra formelen:

$$SRTT = (\alpha * SRTT) + ((1 - \alpha) * RTT)$$

Her er alfa utjevningsfaktoren som bestemmer hvor stor vekting nye og gamle pakker skal ha. Typiske verdier for alfa ligger rundt 0.8 og 0.9. Siden RTT kan variere mye på grunn av kødannelse og overføringsforsinkelser beregner også TCP en MDEV (mean deviation) av RTT. MDEV er gjennomsnittet av forskjellen mellom RTT og SRTT. Ut i fra disse verdiene beregner TCP en RTO (Retransmission Timeout), som er tidsintervallet før en pakke som ikke er bekreftet skal sendes på nytt.

$$SMDEV = (\alpha * SMDEV) + ((1 - \alpha) * MDEV)$$

$$RTO = SRTT + 2 * SMDEV$$

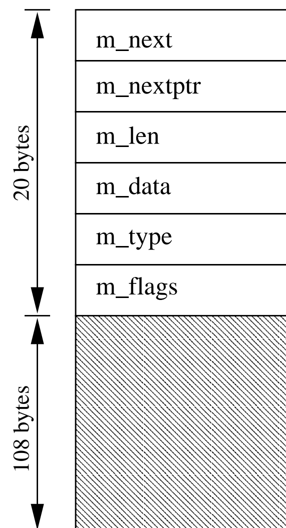
6.4.3 UDP

Ikke alle applikasjoner ønsker den sikre datastrømorienterte tjenesten til TCP. Noen applikasjoner ønsker datagramorientert kommunikasjon. UDP tilbyr en forbindelseløs, usikker kommunikasjon, ved å bruke IP for å frakte meldinger mellom maskiner. I tillegg til tjenestene IP tilbyr, sørger UDP for å skille mellom flere destinasjonsadresser på en maskin og tilbyr sjekksum for å sørge for dataintegritet. UDP er en enkel protokoll, som ofte brukes som byggeblokk for mer spesielle protokoller. Den støtter ikke sending av bekreftelse for mottatte pakker, retransmisjon av pakker, flytkontroll og opphopningskontroll, som TCP gjør.

6.5 Mbuffere

Nettverksprotokoller krever mye av minnehåndteringen som kjernen tilbyr. Dette inkluderer enkel manipulering av buffere av forskjellige størrelser, legge til data når lavere lag i nettverksprotokollen pakker inn data fra høyere lag, fjerne data ettersom data går oppover i protokollen og sørge for å minimere mengden av data som kopieres. Ytelsen til nettverksprotokollen er direkte avhengig av måten minnehåndteringen i kjernen utføres på.

Nettverksprotokollene bruker mbuffere (minnebuffere) i nettverkskoden for å lagre informasjon knyttet til protokollene, og for å sende data fra lag til lag gjennom nettverkskoden. For en TCP/IP-forbindelse er de viktigste lagene det overføres mbuffere mellom socket, TCP, IP og nettverkslaget. Mbuffere er alltid 128 bytes store, inkludert et hode på 20 bytes hvor informasjon om mbufferen er lagret. Figur 25 viser en enkel mbuffer.

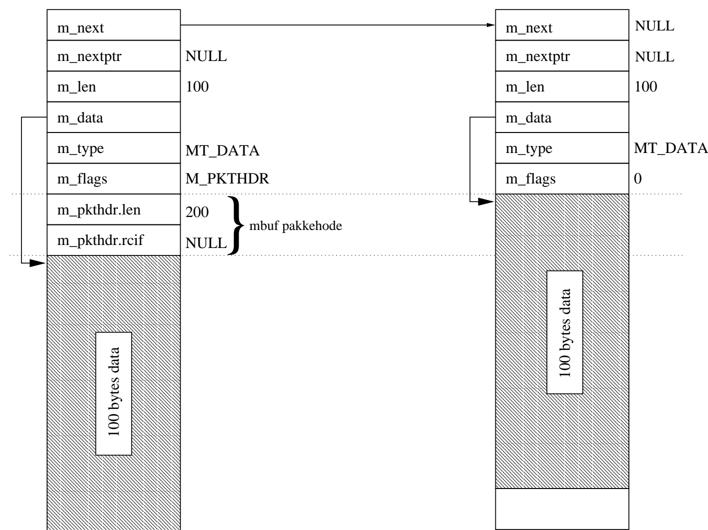


Figur 25: Mbuffer

`m_next` - Peker til neste mbuffer i kjeden.
`m_nextptr` - Peker til neste kjede av mbuffere.
`m_len` - Angir hvor mye data det er i mbufferen.
`m_data` - Peker til dataene lagret i mbufferen eller eksterne data.
`m_type` - Beskrivelse av innholdet i mbufferen (meldingshode, data osv.).
`m_flags` - Ekstrafelter i mbufferen, f.eks. meldingshodefelter.

6.5.1 Kjede av mbuffere

Hvis det skal lagres mer enn 100 bytes ved hjelp av mbuffere, er det ikke nok plass til dataene i én mbuffer. Man må da opprette en ny mbuffer som kan lagre resten av dataene. Hvis man skal lagre 200 bytes, vil man lagre de første 100 bytene i den første bufferet, og de neste 100 bytene i neste buffer. Man vil så sette `m_next` i den første mbufferen til å peke til den andre mbuffere. På denne måten blir disse to mbufferne knyttet sammen i en kjede av mbuffere. En kjede av mbuffere representerer en sammenhengende mengde av data, som kan være en IP-pakke med både pakkens meldingshode og data. En kjede av mbuffere kan i teorien være så lang man ønsker, men som vi skal se senere er ikke dette tilfellet. I figur 26 ser vi to mbuffer som er knyttet sammen.



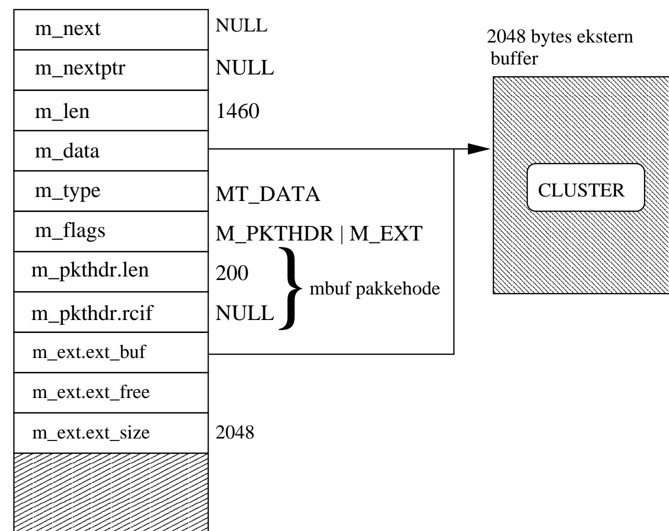
Figur 26: To mbuffere knyttet sammen

`m_pkthdr.len` - Total lengde på mbufferkjede.
`m_pkthdr.rcif` - Peker til grensesnitt for innkommende pakker.

6.5.2 Mbuffer med cluster

Siden mbuffere alltid er 128 bytes lange, den første mbufferen har plass til 100 bytes med data og de neste har plass til 108 bytes, så går det maksimalt

208 bytes i to mbuffere. Hvis datamengden overskrider 208 bytes brukes ikke tre mbuffere eller mer for å lagre dataene, men man bruker isteden en annen teknikk. Denne går ut på å lagre dataene i en større buffer, typisk 1024 eller 2048 bytes, kalt cluster. En cluster er en ekstern buffer. Hvis man bruker en cluster vil kun hodet i mbufferen, som er en peker til clusteret, brukes. En mbuffer med data lagret i en ekstern cluster ser ut som vist i figur 27.



Figur 27: Mbuffer med ekstern cluster

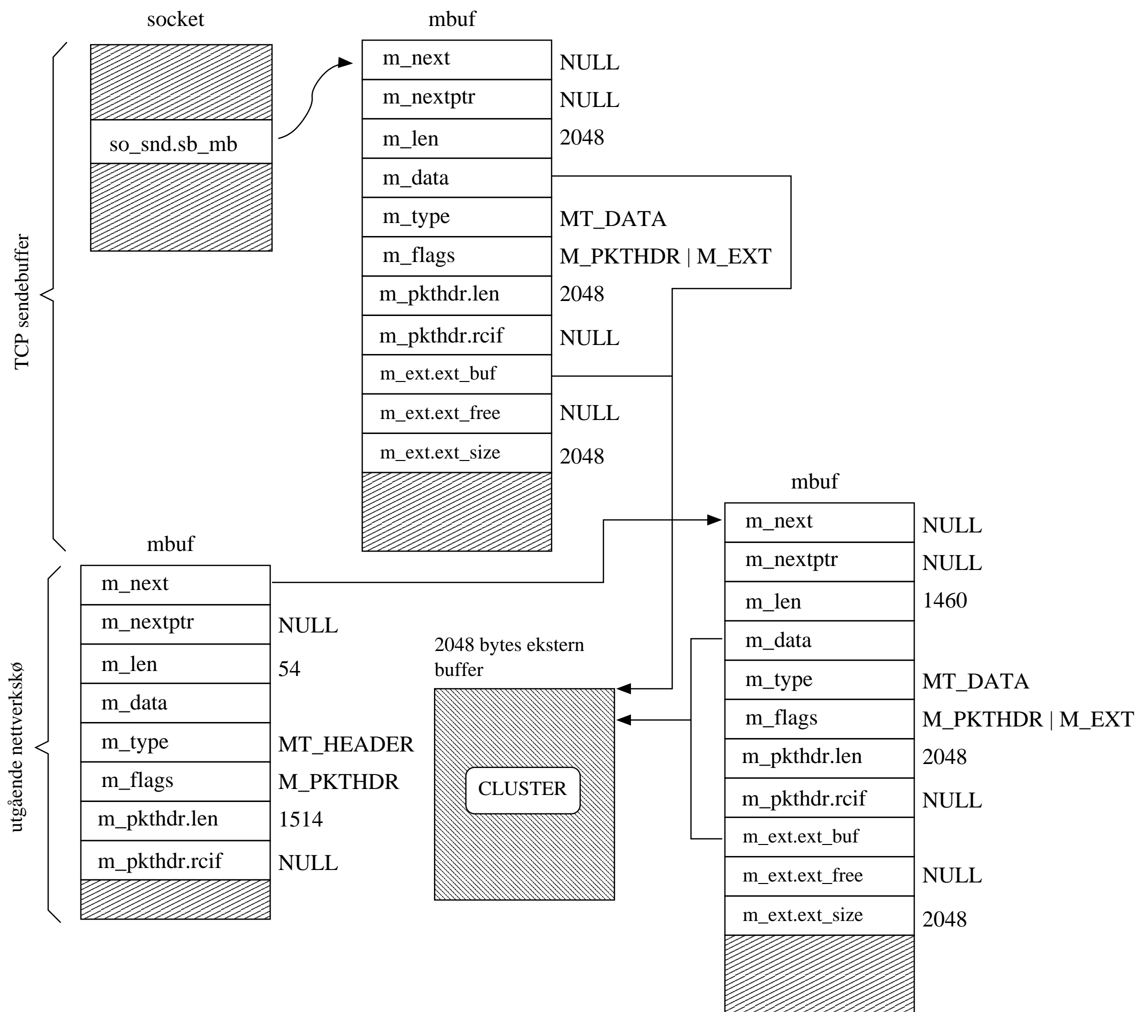
Fordelen med å bruke clustere, er muligheten til å redusere antall mbuffere som må brukes for å holde på store mengder data. Det å opprette mange mbuffere og dele dataene opp i flere mbuffere koster mye prosessorkraft, og er den største grunnen til at clustere brukes når datamengden bli stor. Hvis clustere ikke hadde vært brukt, ville man måtte ha 10 mbuffere for å ta vare på 1024 bytes med data. Den første med 100 bytes med data, de neste åtte med 108 bytes og til slutt en buffer med 60 bytes. Det er mer kostbart å allokere og linke sammen 10 mbuffere, enn det er å allokere en enkel mbuffer med en 2048 bytes stor cluster. En annen fordel med å bruke clustere er at man kan dele clustere mellom flere mbuffere, da mbuffere bruker en peker til dataene i clusteret. Kopiering av data fra en mbuffer til en annen, kan derfor reduseres til en kopiering av en peker istedenfor kopiering av alle dataene. En ulempe med clustere er bortkastet plass. Hvis man skal allokere plass til 1024 bytes, vil man med clustere bruke 2176 (128 + 2048) bytes, mens man ville brukt 1280 (10 * 128) bytes ved å ikke bruke clustere.

6.5.3 Flyt av mbuffere gjennom protokollene

For å sende data fra en socket til en annen må man ha et grensesnitt mot socketen man skal sende data fra. Et slikt grensesnitt er kallene `send`, `sendmsg` og `sendto`. Tilsvarende kall for å motta data er `recv`, `recvfrom` og `recvmsg`. Vi skal nå se på hvordan mbuffere flyter gjennom nettverkskoden fra den opprettes til den går inn i køen for utgående pakker. Vi starter med kallet `sendto`, som leverer data fra applikasjonen til socketkoden.

```
int sendto (int s, const char* msg, int len, int flags,
            const struct sockaddr* to, int tolen);
```

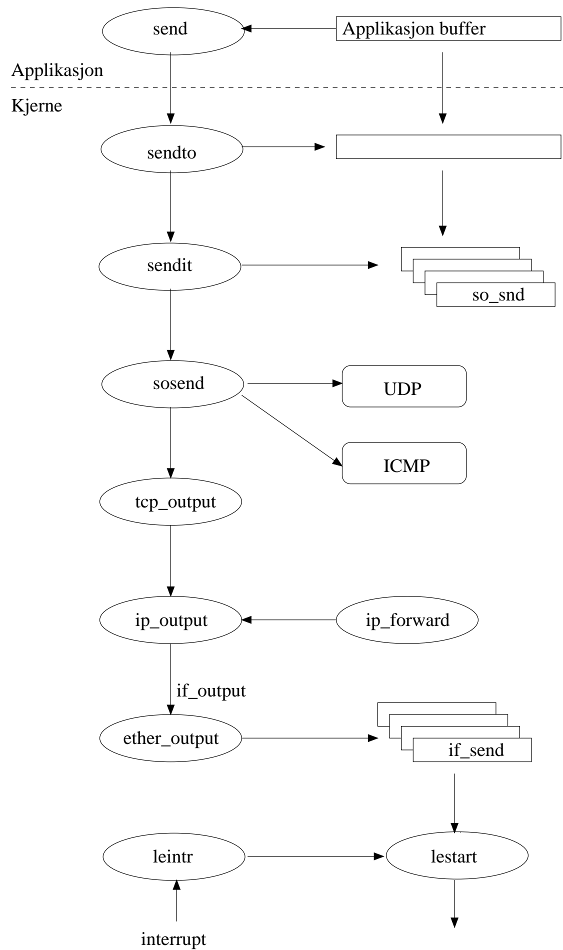
Funksjonskallet `sendto` starter med at socketlaget kopierer mottagerens adressestruktur (`to`) inn i en mbuffer, og data (`msg`) inn i en kjede av mbuffere. Deretter vil protokollaget som svarer til socketen (`s`) bli kalt, f.eks. TCP. Det opprettes en ny mbuffer, og i den blir det satt av nok plass til TCP og IP-hodene. Denne mbufferen blir lagt inn først i kjeden av mbuffere, som allerede inneholder dataene som skal sendes. Protokollaget vil så fylle ut TCP-hodet i den nye mbufferen, og gi mbufferne videre til IP-laget, som også fyller inn sitt hode i mbufferen. TCP og IP-hodet er lagret i slutten av den nye mbufferen, og i forkant av TCP og IP-hodet er det ledig plass. Dette tillater nye protokoller, under IP-protokollen, å legge sine hoder i forkant av TCP og IP-hodene uten å flytte på dataene i den utfylte mbufferen fra TCP og IP. IP-rutinen fyller ut de resterende feltene i IP-hodet, og finner ut hvor mbufferne skal sendes videre, f.eks. til ethernetkortet. Funksjonen som behandler utgående ethernetpakker konverterer IP-adressene til ethernetadresser gjennom ARP (Adress Resolution Protocol). Mens ARP-oppslaget gjøres, holdes mbufferne i påvente av svar. Ethernetrutinen fyller så inn sitt 14 bytes lange hode til den første mbufferen i kjeden. Mbufferne blir så lagt til på slutten av den utgående køen til ethernetkortet. Hvis kortet ikke er opptatt, blir rutinen som sender pakker kalt. Hvis kortet er opptatt, vil pakken bli sendt når den er ferdig med mbufferne som allerede er i køen. Når pakken er sendt og man har mottatt bekreftelse på at pakkene er mottatt, vil mbufferne som pakken har brukt gis tilbake til kjernen, og kan brukes for å sende nye pakker. Figur 28 viser hvordan mbuffere er organisert i den utgående nettverksforbindelsen.



Figur 28: Mbuffere i nettverkskoden

6.6 Dataflyt

Vi skal i dette kapittelet se på hvordan datapakker blir behandlet, fra de opprettes av applikasjonen til de blir sendt ut på nettverket. Vi starter med kallet fra applikasjonen, og følger pakken gjennom socket, protokoll og nettverkslaget. Vi ser her på oppdeling, kødannelse og behandling av pakkene. Figur 29 viser en grov skisse av hvordan data sendes gjennom de forskjellige lagene i nettverkskoden. Øverst har vi applikasjonen, og ned til `tcp_output` står socketen for behandling av pakkene. Fra `tcp_output` til `if_output` behandles pakkene av protokollene, før de sendes videre til link-laget som er den nederste delen av figuren. Først skal vi se på behandlingen av pakker i socketlaget.



Figur 29: Dataflyt gjennom nettverkslagene

6.6.1 Socket

En applikasjon som skal bruke nettverket har en mengde med data som skal sendes. For å sende dataene må applikasjonen legge det den skal sende i en buffer, som sendes videre til nettverket ved hjelp av I/O-systemkall. I BSD har man fire systemkall for å sende data over en nettverksforbindelse, `write`, `writev`, `sendto` og `sendmsg` [3]. Disse fire kallene er sendesystemkall, hvorav de tre første er enklere grensesnitt til det mest generelle kallet `sendmsg`. Alle sendesystemkall ender direkte eller indirekte med å kalle `sosend`. `sosend` er kjernen i socketlaget. Den behandler alle I/O-operasjoner mellom socket og protokollaget, som består i å kopiere data fra prosessen til kjernen, og å sende data videre til protokollen assosiert med socketen.

Før sendekallet utføres må det opprettes en socket som man kan bruke sendekallet på. En socket representerer en ende i en kommunikasjonslink, og holder på all informasjon assosiert med linken. Denne informasjonen er lagret i en socketstruktur. Under er en forenklet versjon av en socketstruktur, der de viktigste elementene er tatt med:

```
struct socket
{
         short so_type;
         short so_state;
       caddr_t so_pcb;
 struct socket *so_q0;
 struct socket *so_q;
         short so_qlimit;

 struct sockbuf
 {
  struct mbuf *sb_mb;
       u_long sb_cc;
         long sb_lowat;
       u_long sb_hiwat;
       u_long sb_mbmax;
 } so_rcv, so_snd;
}
```

- `so_rcv` og `so_snd`
  Hver socket har en buffer for sending (`so_snd`), og en buffer for å motta (`so_rcv`) data. Hver buffer inneholder kontrollinformasjon og pekere til data som er lagret i mbufferkjeden. `sb_mb` peker til den første mbufferen

i kjeden, og sb_cc er totalt antall bytes mbufferne inneholder. sb_lowat og sb_hiwat regulerer socketens algoritmer for flytkontroll. sb_mbmax er den øvre grense for hvor mye minne som kan allokeres som mbuffere til hver av socketens buffere. Standardverdier for denne grensen er satt hver gang en ny socket blir opprettet med socketsystemkallet. sb_lowat og sb_hiwat kan senere endres av prosessen, så lenge de ikke overstiger kjernens øvre grense (sb_max) på 262144 bytes for hver socketbuffer. Tabell 1 viser standardverdier for internettprotokollene. Tabellen viser at sb_hiwat for UDP er 9216. Så lenge prosessen ikke endrer sb_hiwat vil et forsøk på å sende et datagram større enn 9216 bytes resultere i en feilmelding.

	so_snd			so_rcv		
	sb_hiwat	sb_lowat	sb_mbmax	sb_hiwat	sb_lowat	sb_mbmax
UDP	9*1024	2048	2*sb_hiwat	40*(1024 + 16)	1	2*sb_hiwat
TCP	8*1024	2048	2*sb_hiwat	8*1024	1	2*sb_hiwat
IP	8*1024	2048	2*sb_hiwat	8*1024	1	2*sb_hiwat

Tabell 1: Standardverdier for internettprotokollene

- so_type
  Spesifiserer hva slags type socket som skal brukes ved sending av data. Dette kan være TCP, UDP eller ren IP.

- so_state
  Holder på informasjon knyttet til socketens status. Dette kan være at socketen ikke kan ta imot mer data, at socketen ikke kan sende mer data, at socketen holder på å koble seg opp mot en annen socket og annen tilstandsinformasjon knyttet til socketen.

- so_pcb
  Peker til en kontrollblokk som inneholder protokollavhengig statusinformasjon og parametere for socketen. Det finnes flere typer kontrollblokker. En kontrollblokk for UDP inneholder intern og ekstern IP-adresse, intern og ekstern port, datagram-flagg, IP-valg og annen protokollspesifikk informasjon. En kontrollblokk for TCP inneholder det samme som en kontrollblokk for UDP, og i tillegg sekvensinformasjon, informasjon for tidsberegninger, forbindelsestatus, mottakervindu, sendevindu og annen informasjon knyttet til TCP.

- so_q0, so_q og so_qlimit
  Forbindelser som ikke er opprettet (treveis håndtrykk ikke gjennomført) legges i so_q0 køen, mens forbindelser som er opprettet legges i so_q køen. Maks antall forbindelser i køene spesifiseres av so_qlimit som kan settes ved kallet listen.

Etter sendekallet har opprettet en socket kalles sendto og sendit, som gjør klar datastrukturer som skal brukes av sosend. sendit oppretter en UIO-struktur [3], som brukes for å samle opp utgående buffere spesifisert av prosessen inn i mbuffere i kjernen. UIO-strukturen består av flere iovec strukturer, som er pekere til data som skal hentes fra applikasjonens adresserom og inn i kjernen (se figur 30). UIO-strukturen brukes mest for lesing og skriving av data til devicedrivere, men er også nyttig ved kopiering av data mellom forskjellige typer adresserom.

```
struct uio
{
 struct iovec *uio_iov; ==> struct iovec
 ...                        {
                             char   *iov_base; // Baseadresse
}                            size_t iov_len;   // Lengde
                             }
```

Destinasjonsadresse og kontrollinformasjon blir så kopiert inn i mbufferne, og sendit finner ut hvilken socket den skal bruke. Socketen, destinasjonsadressen, UIO-strukturen og kontrollinformasjonen blir sendt videre til sosend. sosend er en av de mest kompliserte funksjonene i socketlaget. Det er sosend sin oppgave å gi data og kontrollinformasjon videre til protokollaget, med riktig type socket og med hensyn til buffergrensene spesifisert av socketen. sosend sin tolkning av sendebufferets grenser, so_lowat og so_hiwat, avhenger av om protokollen støtter sikker dataoverføring eller ikke. For sikre protokoller inneholder sendebufferet både data som ikke har blitt sendt og data som har blitt sendt men ikke er bekreftet mottatt.

Det er sosend sin oppgave å sørge for at det er nok plass i sendebufferet før data sendes til protokollaget og legges i sendebufferet. sosend overfører data til protokollen på to forskjellige måter, avhengig av flagget PR_ATOMIC.

- Hvis PR_ATOMIC er satt, må sosend beholde de samme meldingsgrensene spesifisert av prosessen til og gjennom protokollaget. I dette tilfellet vil sosend vente på at det skal bli nok plass for hele meldingen før den

sendes videre. Når det er nok plass, opprettes det en mbufferkjede med hele meldingen, som gis videre til protokollen gjennom ett enkelt kall. RDP og SPP er eksempler på protokoller av denne typen.

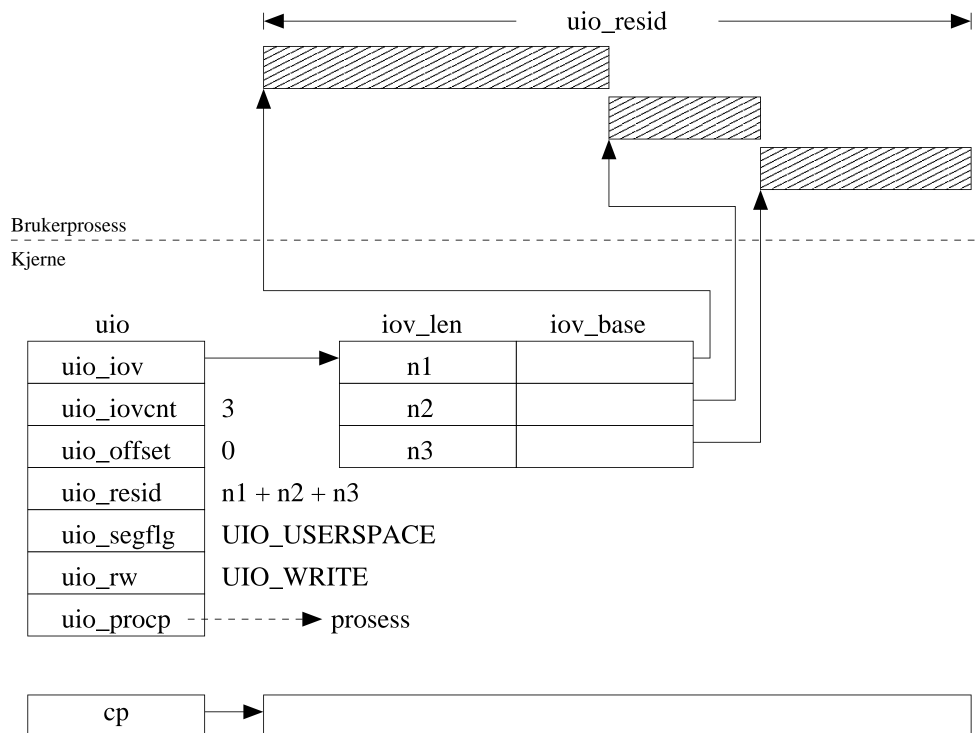
– Hvis `PR_ATOMIC` ikke er satt, vil `sosend` levere meldingen til protokollen ved å gi den en mbuffer av gangen. Den kan også dele opp mbuffere for å unngå å overstige `sb_hiwat`. Denne metoden brukes ved protokoller som TCP (`SOCK_STREAM`).

TCP-applikasjoner har ingen kontroll over størrelsen på utgående TCP-segmenter. For eksempel vil en melding på 4096 bytes, sendt over en TCP-socket, bli delt opp av socketlaget i to mbuffere. Disse to mbufferne vil ha en ekstern cluster på 2048 bytes hver, hvis det er nok plass i sendebufferet til begge (4096 bytes). Senere i protokollaget vil TCP segmentere dataene i henhold til den største tillatte segmentstørrelsen for forbindelsen, som vanligvis er mindre enn 2048 bytes.

Når en melding er for stor til å få plass i den tilgjengelige plassen i socketens sendebuffer og protokollen tillater å dele opp meldingen, så vil `sosend` likevel ikke levere data til protokollen før den ledige plassen i sendebufferet overstiger `sb_lowat`. For TCP er `sb_lowat` 2048, så denne regelen hindrer socketlaget i å gi TCP små mengder med data når sendebufferet er nesten fullt. Hvis denne regelen ikke hadde vært brukt, ville det ført til betraktelig reduksjon av ytelsen til protokollen [41].

For forbindelseløse protokoller (f.eks. UDP) lagres ikke dataene i sendebufferet, og ingen bekreftelse er forventet. Hver melding leveres med en gang til protokollen, hvor den blir satt i kø for overføring. I dette tilfellet er `sb_cc` alltid 0, og `sb_hiwat` spesifiserer den største verdien for et sendesystemkall og inndirekte den maksimale størrelsen på datagrammet.

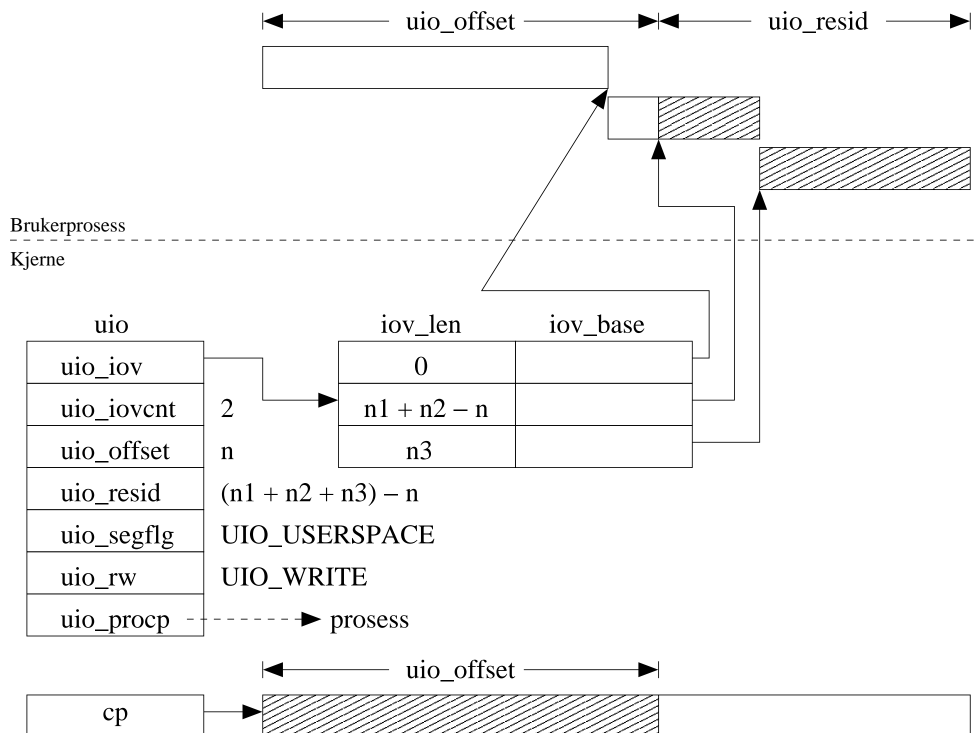
`sosend` er ansvarlig for å kopiere data fra applikasjonen til UIO-strukturen, som ble opprettet i kjernen av `sendit`. Før den gjør det, må det opprettes en mbuffer som den kan plassere dataene i. Hvis meldingen er stor nok til at opprettelse av en cluster er lønnsomt, opprettes et cluster som knyttes til mbufferen, hvis ikke opprettes en enkel mbuffer. Når mbufferen er klar, kopieres data med funksjonen `uiomove(cp, n, uio)` til den er full eller det ikke er mer data å kopiere. `cp` er her en peker til mbufferen i kjernen, `n` er antall bytes som skal kopieres og `uio` er UIO-strukturen. Denne kopieringen gjentas inntil applikasjonens buffer er tom. Figurene 30 og 31 viser UIO-strukturen før og under `uiomove` kallet utføres.



Figur 30: UIO-struktur før flytting av data

Figurene viser tre buffere som en applikasjon ønsker å sende. Disse tre bufferne er beskrevet med en iovec-struktur, som beskrevet tidligere i dette kapittelet. Iovec-strukturen angir buffernes baseadresse (`iov_base`) og buffernes lengde (`iov_len`). Bufferne blir kopiert over i en buffer (cp), som er en peker til en mbuffer eller en ekstern mbuffercluster i kjernen. Kopieringen starter med det første bufferet, som har lengde n1. Under kopieringen blir variabler i UIO-strukturen oppdatert slik at den hele tiden holder rede på hvor mye som er kopiert og hva som gjenstår. I figur 31 ser vi en kopiering som er påbegynt og de tilhørende oppdaterte variablene i UIO-strukturen. Når `uiomove` kallet er utført, vil alle data være hentet fra applikasjonens adresserom og inn i mbuffere i kjernen.

Data er nå fylt inn i mbuffere, og socketlaget er ferdig med å prosessere de utgående pakkene. Socketlaget overlater videre behandling av pakkene til protokollaget. I protokollaget skal vi se på hvordan dataene fortsetter videre gjennom TCP/UDP og IP-protokollene.



Figur 31: UIO-struktur etter flytting av data

6.6.2 Protokollene

Når protokollaget blir kalt, oppretter den valgte protokollen en PCB (Protocol Control Block). PCB-er brukes av protokollaget for å holde på forskjellig informasjon som er nødvendig for hver enkel UDP og TCP-socket. Internettprotokollene oppretter IP og TCP PCB-er. Siden UDP er forbindelseløs, er all informasjon den behøver for sitt endepunkt tilgjengelig i IP sin PCB, og UDP har derfor ingen egen PCB. IP PCB-en holder på informasjon knyttet til alle TCP og UDP-endepunkter. Det vil si intern og ekstern IP-adresse, internt og eksternt portnummer, prototype for IP-hodet, IP-opsjoner og en peker til routingtabellen for denne linken. TCP PCB-en holder på statusinformasjon som TCP må ta vare på for hver forbindelse. Det vil si sekvensnumre for begge retninger, vindustørrelse, retransmisjonstider og lignende.

All data som overføres over en TCP-forbindelse er tilknyttet et 32-bit langt sekvensnummer. I TCP-protokollen oppdateres hodet til pakken som skal sendes med dette nummeret. TCP-hodet inneholder sekvensnummeret til den første byten i pakken. TCP-hodet inneholder også et sekvensnummer for bekreftede pakker, som er det neste sekvensnummeret forventet fra

senderen av bekreftelsen. På denne måten kan protokollen sjekke om data den mottar er riktig, og om de kommer i riktig rekkefølge. Når disse verdiene er satt, oppretter TCP-protokollen et sendevindu, dersom det ikke allerede finnes et. Hvis det finnes et vindu vil vinduet flyttes for å gjøre rom for den nye pakken hvis det er mulig å flytte vinduet. Hvis det ikke er mulig å flytte vinduet, må pakken vente på å bli sendt til vinduet kan flyttes. Hvis pakken kan sendes, vil en retransmisjonstid bli satt. Hvis det ikke er mottatt bekreftelse når denne tiden utløper, vil samme data sendes på nytt. Verdien til den nye retransmisjonstiden regnes ut dynamisk, basert på roundtriptid, og hvor mange ganger dataene er sendt. Denne verdien vil alltid ligge mellom 1 og 64 sekunder, og sørger for at mottakeren får en ny pakke når man ikke kan vite om pakken er mottatt eller ikke. Hvis pakken som sendes ut er en tidlig pakke i forbindelsen kan den stoppes på grunn av Nagle-algoritmen, som sørger for at det ikke blir sendt ut for mange pakker i begynnelsen av en forbindelse. På denne måten unngår man til en viss grad at mottakeren ikke får mer pakker enn han kan behandle. Spesielle flagg i kildekoden til TCP gjør det mulig å overstyre denne funksjonen, som er praktisk når man på forhånd kan anta mengden pakker mottakeren kan behandle.

Pakken er nå klar for å sendes, og det opprettes en mbuffer som TCP og IP-hode informasjon kan kopieres inn i. Er det plass til dataene i denne mbufferen vil de kopieres direkte fra socketens sendebuffer inn i denne. Hvis det ikke er plass vil det opprettes enda en mbuffer som dataene kan kopieres inn i og linkes til mbufferen som inneholder hodet til TCP og IP. Husk at hvis dataene er i en ekstern cluster vil ikke dataene kopieres, men mbufferen vil kun opprette en peker til riktig mbuffer i socketens sendebuffer. Mbufferne blir så sendt videre til IP-protokollen gjennom funksjonen `ip_output`.

Figur 32: Dataflyt i protokollaget

TCP-laget gir data videre til IP-laget ved kallet til `ip_output` funksjonen. Dataene som den får inneholder et hode fra TCP-laget og data fra applikasjonen. IP-laget legger til sitt eget meldingshode til meldingen. Hvis det resulterende datagrammet er for stort for å sendes over nettverket, splitter IP-laget meldingen opp i flere fragmenter, som hver inneholder et meldingshode og en del av de opprinnelige dataene. IP-laget gjør bare endringer på

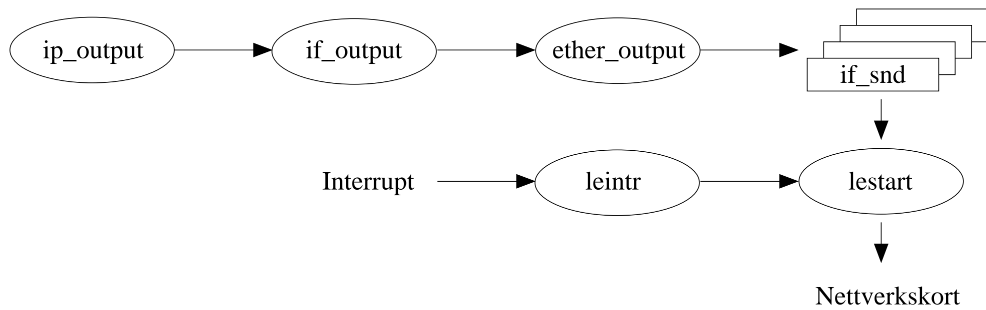
meldingshodet, og undersøker eller endrer ikke på noe av selve meldingen. IP-hodet inneholder blant annet formatet til pakken, adresse, routing og fragmenteringsinformasjon. Før meldingen blir sendt videre, sjekker IP-laget at adressen meldingen skal sendes til er gyldig. Hvis den er gyldig, gis pakken videre til neste lag som er linklaget.

6.6.3 Linklaget

Det første linklaget gjør er å opprette en `ifnet`-struktur for hvert nettverkskort. Denne strukturen inneholder en liste med adresser, som er assosiert med nettverkskortet. På denne måten kan et nettverkskort ha flere adresser knyttet til seg. `ifnet`-strukturen inneholder også pekere til funksjonskall, som angir hvilke kall som skal brukes for nettverkskortet. Er det et ethernetkort, vil f.eks. det generelle kallet `if_output` resultere i et kall til `ether_output`. Til slutt har `ifnet`-strukturen en kø hvor utgående pakker kan plasseres. Hvert nettverkskort har en egen `ifnet`-struktur, og har derfor også en egen kø. Denne køen er `if_snd`, som er en `ifqueue`-struktur. `ifnet`-strukturen inneholder også mye annen informasjon, men de jeg har nevnt her er de viktigste. Når all nødvendig informasjon er fylt ut i `ifnet`-strukturen, er nettverkskortet klargjort for sending av pakker.

```
struct ifnet {
 struct ifaddr *if_addrlist;
 int (*if_init) (...)
 int (*if_output) (...)

 struct ifqueue {
   struct mbuf *ifq__head;
   struct mbuf *ifq_tail;
 } if_snd;
}
```

Sending av pakker i linklaget starter med at `ip_output` kaller `if_output`. `if_output` kaller så riktig funksjon for å sende utgående pakker spesifisert av `ifnet`-strukturen. For ethernet er dette `ether_output`. `ether_output` tar datadelen av ethernetpakken, pakker den inn med et 14 bytes langt ethernethode, og plasserer den på `ifnet`-strukturens sendekø. Hvis det ikke er plass i sendekøen, vil pakken kastes og en feilmelding registreres. Det er så opp til protokollen å avgjøre hva den skal gjøre med denne feilmeldingen. Pakken ligger nå på `ifnet`-strukturens sendebuffer klar for sending.



Figur 33: Dataflyt i linklaget

Pakker i køen blir sendt ved å kalle lestart. Den tar den første pakken fra køen, og gir den til nettverkskortet. Når nettverkskortet er ferdig med å sende pakken, genererer den et interrupt som kaller `leintr`. Den sjekker om det er flere pakker å sende, og hvis det er tilfellet, vil den kalle `lestart` som vil sende en ny pakke. Dette gjentas helt til køen er tom for pakker. Nettverkskoden er nå ferdig med å behandle pakkene.

7 Målinger på disk-til-nettverk veien

Dette kapittelet beskriver det å gjøre målinger på datasystemer, målingene som er gjort på Chorus og NetBSD, valg av målingsmetoder, design av målingene, resultater og til slutt en drøfting av resultatene. Målingene er delt opp i tre forskjellige målinger. Den første ser på ytelsen til minne, som nettverkskoden er svært avhengig av. Neste måling tar for seg kopiering av data fra disk til applikasjon, mens den siste tar for seg flytting av data fra applikasjon til nettverk. Den siste målingen er mer omfattende, og ser på nettverksprotokollene i detalj med hensyn på køing og gjennomstrømning av data, ved hjelp av prøver plassert i kjernen til operativsystemet. Først skal vi se på fremgangsmetode og valg av analysemetode i forbindelse med målinger på datasystemer.

7.1 Målinger på datasystemer

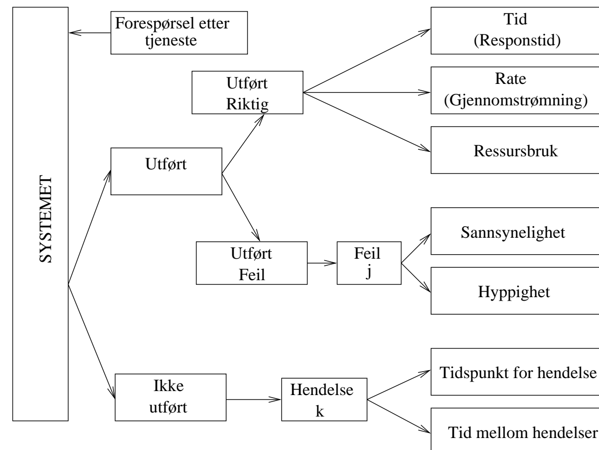
Når man skal gjøre en ytelsesanalyse av et datasystem [2] er det viktig å ha et mål med oppgaven og bestemme seg for hva som skal være med og hva som er utenfor systemet vi skal vurdere. Målet med oppgaven og valg av hva som skal være med i systemet, er med på å bestemme valg av målemetoder og å finne den riktige bakgrunnsbelastningen på systemet. Det er også viktig å gjøre rede for hvilke tjenester systemet målingene utføres på skal kunne tilby.

7.2 Valg av riktig type måling

Etter mål med oppgaven og valg av hva som skal inngå i systemet er bestemt, må det velges hvilke kriterier som skal brukes for sammenligning av ytelse. En måte å gjøre det på, er å først liste opp de tjenestene systemet kan tilby, vist som "Forespørsel etter tjeneste" i figur 34.

Utfallet av disse forespørslene kan deles inn i tre kategorier. Systemet utfører enten tjenesten korrekt, feil, eller utfører ikke tjenesten i det hele tatt. Et eksempel er en server som sender pakker fra disk til nettverk. Mulighetene den har er å sende pakken ut på nettverket korrekt, sende den ut med feil, eller serveren kan være nede slik at pakken ikke blir sendt i det hele tatt.

- Utført riktig
  Hvis systemet utfører tjenesten riktig, blir ytelsen beregnet ut fra hvor lang tid det tar å utføre tjenesten, hvor stor gjennomstrømningen er og hvor mye ressurser som er tatt i bruk for å utføre jobben. For



Figur 34: Tre mulige resultater av forespørsel etter tjeneste

nettverksytelse kan dette være den tid en pakke bruker ut til nettverket, hvor mange pakker som kommer ut per tidsenhet og hvor stor del av tiden ressursene i serveren er i bruk. Måling av ressursbruken i systemet er med på å bestemme systemets flaskehals, og en forbedring i dette punktet vil gi best ytelsesforbedring.

– Utført feil
  Hvis tjenesten blir utført feil, er det til god hjelp å klassifisere feilene og finne sannsynligheten for at de inntreffer.

– Ikke utført
  Det er også her til god hjelp å klassifisere tilfellene og finne sannsynligheten for at de inntreffer. For eksempel kan en server være nede 0,03% av tiden grunnet programvarefeil, og 0,01% grunnet maskinvarefeil.

Når hvilke målinger som skal utføres er valgt er det viktig å gi en beskrivelse av de systemdelene som kan påvirke resultatet av målingene. Det kan være arbeidslasten på serveren, hvilke maskinvarekomponenter den bruker og hvilken programvare den kjører på. Noen av disse komponentene forandrer seg fra gang til gang, som f.eks. arbeidslasten. Arbeidslasten må derfor deles opp i forskjellige typer arbeidslaster, og målinger må gjøres under alle disse forholdene.

7.3 Evalueringsteknikker

Det finnes tre typer teknikker for å analysere ytelsen til et system, analytisk modell, simulasjon og måling. Det er flere faktorer som spiller inn ved valg av teknikk. Vi skal her se på to av disse.

– Tid
  I de fleste situasjoner er man avhengig av å få resultatene av analysen så fort som mulig. Hvis det er tilfellet, er en analytisk modell trolig det eneste valget man har. Skal man bruke simulering vil man som oftest bruke lang tid. Målinger tar vanligvis lenger tid enn en analytisk modell, men kortere enn simulering. I målinger er det mye som skal stemme, og hvis noe kan gå galt er det også store sjanser for at noe går galt hvis man ikke er nøye med gjennomføring av målingene. Som et resultat av dette er det veldig varierende hvor lang tid en måling tar.

– Nøyaktighet
  I analytiske modeller kreves mange forenklinger og antagelser, modellene er derfor ofte unøyaktige. I simulasjon kan man legge inn flere detaljer og man krever færre antagelser enn i en analytisk modell. Simulasjon er derfor ofte nærmere virkeligheten enn en analytisk modell. Ved måling jobber man direkte mot systemet og det kan derfor være lett å tro at disse målingene må være korrekte. Men under målingene er det flere faktorer som spiller inn, som systemkonfigurasjon, last på maskinen og tidspunktet målingen ble tatt, og dette kan gi svært forskjellige resultater. Derfor kan nøyaktigheten til en måling variere.

Vi har i dette kapittelet sett på det å gjøre en ytelsesanalyse av et datasystem. I kapittel 8, 9 og 10 skal jeg gjøre en ytelsesanalyse av komponenter i operativsystemene Chorus og NetBSD. Komponentene inkluderer minnehåndtering, operasjoner på disk og behandling av data gjennom nettverkskoden.

7.4 Valg av målingsmetode

For å kunne finne flaskehalsen i disk-til-nettverk veien i Chorus og NetBSD, har jeg valgt å først dele opp målingene i tre målinger. En på minneytelse, en på diskytelse og en på nettverksytelse. Den første målingen ser på den maksimale gjennomstrømningen av data ved kopiering av data fra en posisjon i minne til en annen. Dette vil gi en indikasjon på om minnehastigheten setter begrensning på nettverkskoden. Den neste målingen ser på hvor lang tid det tar å hente data fra disk under forskjellige forutsetninger. Den ser også på hvordan bruk av cache virker inn på lesing av filer fra disk. Den siste

målingen undersøker gjennomstrømning av data fra applikasjon, gjennom nettverkskoden og ut nå nettverket. Nettverksmålingen er igjen delt opp i to målinger. Den første ser på den maksimale hastigheten som kan oppnås ved sending av data gjennom nettverkskoden. Den andre går mer i detalj og ser på flyt av data gjennom nettverkskoden inne i kjernen til operativsystemet.

7.5 Nøyaktighet

For å oppnå så nøyaktige målinger som mulig, må tiden som registreres i målingene være så nøyaktig som mulig. I de operativsystemene som ble brukt under målingene finnes det systemkall for å registrere tid med nøyaktighet ned til ett mikrosekund. Dette er ikke nøyaktig nok for målingene, da et mikrosekund tilsvarer 933 klokkesykler på en 933MHz rask maskin. Det vil si at det er en unøyaktighet på flere hundre instruksjoner for hver måling. For å kunne registrere nøyaktigere målinger tok jeg i bruk RDTSC-instruksjonen [22]. RDTSC-instruksjonen er en instruksjon som som ble introdusert i Pentium-prosessoren. Den gir som resultat antall instruksjoner som er utført siden maskinen ble startet. Det vil si at nøyaktigheten til en måling som bruker RDTSC vil variere ettersom hvor rask maskinen er. For en 933MHz rask maskin vil man få en nøyaktighet på 0.0011 mikrosekunder, mot 1 mikrosekund for en måling som registrerer tiden i mikrosekunder. Uansett maskinhastighet vil målinger som bruker RDTSC-instruksjonen ha nøyaktighet på en instruksjon.

Dette er ikke helt korrekt, da Pentium-prosessoren støtter out-of-order utførelse. Det vil si at instruksjonene i koden ikke nødvendigvis blir utført i den rekkefølgen de står i koden. Dette kan føre til at instruksjoner som er før RDTSC-instruksjonen i kildekoden blir utført etter RDTSC-instruksjonen, og instruksjoner etter RDTSC-instruksjonen kan bli utført før RDTSC-instruksjonen. Dette er ikke ønskelig, og for å forhindre at dette skal skje har Pentium-prosessoren en instruksjon CPUID [23]. CPUID er en instruksjon som sørger for at alt som står i koden før denne instruksjonen vil bli utført før denne instruksjonen, og alt som står etter vil bli utført etter. Ved å ta i bruk denne instruksjonen rett før RDTSC-instruksjonen, vil RDTSC-instruksjonen bli utført på rett sted i forhold til kildekoden.

7.6 Bakgrunnslast

Prøvene som ble tatt, ble tatt med hensyn på å finne den maksimale gjennomstrømningen til serveren. Prøvene ble derfor tatt uten bakgrunnslast og det ble sørget for at serveren ikke kjørte andre programmer i bakgrunnen

mens prøvene ble tatt. Cron-jobber og andre jobber som kjørte ble derfor skrudd av før målingene ble utført.

7.7 Teknisk spesifikasjon av maskinvare

Maskinene som målingene ble utført på var to forskjellige maskiner. Målingene på Chorus ble først utført på én maskin. Denne maskinen var THEBE, som beskrevet i tabell 3. Da målingene fra denne maskinen var ferdige fikk vi en ny maskin, NAG, som var kraftigere med tanke på disk, prosessorkraft og nettverkskort. Det ble samtidig bestemt at INSTANCE skulle bruke NetBSD, og målingene ble derfor kodet om til å kunne kjøres under NetBSD. Grunnen til at Chorus ikke ble installert på denne maskinen var at Chorus ikke støtter den maskinvaren som maskinen NAG er bygd opp med. Under følger teknisk spesifikasjon av disse to maskinene.

NAG: Dell Precision WorkStation 620	
Prosessor	
Type	Dual Intel Pentium III Xeon 933 MHz med front-side (minne) bus med ekstern hastighet på 133 MHz og intern matteprosessor
1st level cache	32 KB (16 KB datacache; 16 KB instruksjonscache)
2nd level cache	256 KB
Systeminformasjon	
Chipset	Intel 840i slot2 chipset
PCI bus hastigheter	32-bit, 33.3 MHz *og* 64-bit bus på 33.3 MHz and 66.6 MHz
Minne	
Type	PC800 feilsjekking og feilkorrigering (ECC) RDRAM i rambus in-line minnemodul (RIMM) slot
Størrelse	256 MB
Lagringssystem	
Disk	9.1 GB, 10.000 RPM SCSI harddisk
PCI SCSI controller	Integrert dual-channel Adaptec 7899 Ultra 160/M LVD (160 MB/s)
Kommunikasjonssystem	
Gbit NIC	3Com EtherLink server nettverkskort (3C985B-SX)
NIC (for eksterne forbindelser)	Integrert 10/100 3Com Ethernet kontroller (3C920 basert og 3C905-TX kompatibelt)

Tabell 2: Server NAG (NetBSD)

THEBE	
Prosessor	
Type	Pentium II 350MHz
Minne	
Størrelse	192 MB
Lagringsmedia	
Disk	Seagate Medalist 6531 AT (ST-36531A)
Sylindere	13.446
Hoder	15 (6 fysiske hoder)
Sektorer	63
Kapasitet	6505 MB
Buffer	128K
Kommunikasjonssystem	
Kort	SMC ethercard elite 16
Type	NE2000 kompatibelt, COMBO
Hastighet	10 Mb/s
Bus	ISA

Tabell 3: Server THEBE (Chorus)

8 Minneytelse

I dette kapittelet beskriver jeg målingen av minneytelsen under både Chorus og NetBSD. Først vil jeg forklare hvorfor målingene er utført. Så vil jeg beskrive hvordan de er utført og presentere resultater fra målingene. Til slutt drøfter jeg resultatene og sammenligner de med målinger som er gjort på andre systemer.

8.1 Problembeskrivelse

Målinger på minne ble gjort for å kunne ha en formening om hvor mye tid som brukes på minneoperasjoner, og hvilken rolle denne tiden spiller inn på operasjoner i forbindelse med behandling av data i de forskjellige komponentene som inngår i disk-til-nettverk veien. Minneoperasjoner blir utført ved henting av data fra disk, sending av data til socketen og ved kopiering og behandling av data gjennom nettverkskoden. Ytelsen for kopiering av minne er derfor viktig for ytelsen til de komponentene jeg skal gjøre målinger på senere.

Hastigheten til minne er begrenset, og hovedgrunnen til operativsystemets I/O-flaskehals er at økningen av hastigheten til minne ligger langt bak økningen av hastigheten til prosessoren og nettverket [25]. Måling av minnehastigheten er derfor viktig for å kunne fastslå om dette stemmer overens med de systemene jeg skal utføre målinger på.

8.2 Valg av målingsmetode

Det finnes flere kjente verktøy for å måle hastigheten til minne. Et eksempel på et slikt verktøy er STREAM [40]. STREAM er et enkelt verktøy, som måler kontinuerlig gjennomstrømning av data fra et område i minne til et annet. Fordelen med å bruke et slikt verktøy er at det minsker sjansen for programmeringsfeil, da programmet er godt testet. Likevel valgte jeg å ikke bruke et slikt verktøy av tre grunner. For det første ville ikke verktøyene kompilere under Chorus. For det andre ville jeg ta i bruk nøyaktige målinger, som beskrevet i avsnitt 7.5. For det tredje vil jeg ved å gjøre målingene selv, ha tilgang til alle resultatene fra målingene. På denne måten kan jeg undersøke målingene med de statistiske metodene jeg selv ønsker.

8.3 Utførelse

For å bestemme ytelsen til kopiering av minne valgte jeg å utvikle et program, som utførte kopieringsoperasjoner fra en minnelokasjon til en annen.

Programmet som utførte målinger på kopiering av minne startet med å opprette en buffer (BUF_ORIG), med størrelse BUF_SIZE, som skulle brukes for originaldataene. Deretter ble det opprettet en buffer som skulle brukes til å kopiere originaldataene inn i (BUF_COPY). På dette punktet i kildekoden ble det nåværende tidspunkt registrert i en tidsstruktur (T1). Data ble så kopiert fra BUF_ORIG til BUF_COPY. Etter at kopieringen var utført ble nåværende tidspunkt registrert i en tidsstruktur (T2). Registrering av tidspunktene og kopiering av dataene ble gjentatt i en løkke som ble utført 100 ganger. Grunnen til at kopieringen ble utført flere ganger, er at jeg da kan beregne minimumsverdi, maksimumsverdi, gjennomsnitt, standardavvik og konfidensintervall for målingene. Ut i fra disse dataene kan jeg finne den gjennomsnittlige tiden (GT) for en enkel minnekopiering av størrelse BUF_SIZE, samt at jeg kan finne hvor mye data som kan kopieres per tidsenhet (TP). Hele denne operasjonen ble så gjentatt med forskjellige størrelser på minneområdet (BUF_SIZE), for å se om størrelsen på bufferet har noen innvirkning på målingsresultatet. Valg av størrelser for minneområdet (BUF_SIZE) ble satt til $n * 2^{12}$, der $n \in \{1, 2, ..., 100\}$. Beregninger av total tid og hvor mye data som kunne kopieres per tidsenhet ble gjort ut fra formlene:

$$SINGLE = \texttt{T2} - \texttt{T1}$$

$$GT = \frac{\sum_1^{100} SINGLE}{100}$$

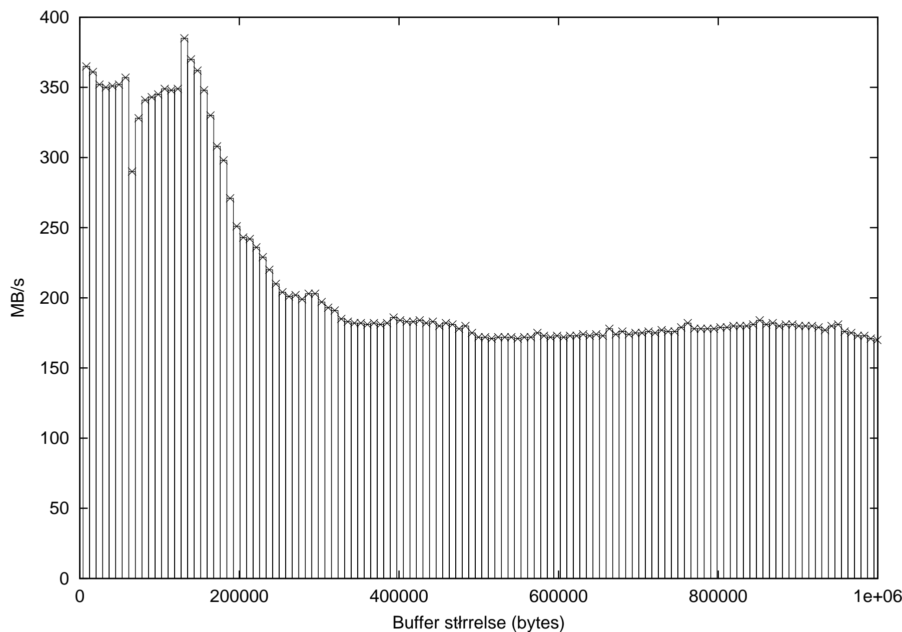
$$TP = \frac{\texttt{BUF_SIZE}}{GT}$$

8.4 Resultater

Under følger resultater for Chorus og NetBSD. Resultatene fra målingene er vist i stolpediagrammene i figur 35 og 36. Tiden det tok å allokere minne er ikke tatt med i målingene. Dette må legges til ved beregning av tid for kopiering, avhengig om minnet allerede er allokert eller ikke. For å ha en formening om nøyaktigheten til målingene, er statistiske resultater for en måling med bufferstørrelse på 400 KBytes vist i tabell 4 og 5.

8.5 Chorus

I figur 35 ser vi resultatene fra kopiering av minne. Vi ser ut fra grafen at kopiering av minne har en topp for bufferstørrelser under 180 KBytes. Her er overføringshastigheten på nesten 400 MB/s. Videre ser vi at grafen har en bunn, som strekker seg over et stort område, for bufferstørrelser som er større enn 300 KBytes. Denne todelingen av grafen viser innvirkningen prosessorens cache har på kopiering av buffere. For bufferstørrelser fra 180 til 300 KBytes ser vi at hastigheten synker, da mindre og mindre av bufferet som kopieres ligger i prosessorens cache.



Figur 35: Kopiering av minne, Chorus

I nettverkskoden er data organisert i mbuffere (se avsnitt 6.5). For å utveksle data mellom forskjellige lag i nettverkskoden kopieres mbufferne mellom lagene. Hvis disse mbufferne ligger i prosessorens cache, kan kopiering av disse utføres mye raskere enn hvis de ikke er i cachen. Vi ser fra figur 35 at det er mer enn en dobling av hastigheten for bufferstørrelser under 180 KBytes i forhold til bufferstørrelser over 300 KBytes.

I en server som sender ut mye forskjellige data, og samtidig utfører andre oppgaver, kan vi ikke regne med at mye av dataene ligger i prosessorens cache.

Serveren vil derfor ha en hastighet for minnekopiering på 180 KB/s. Denne hastigheten tilsvarer en hastighet på 1440 Kbit/s. Det vil si at denne maskinen ikke vil klare å sende ut nok data til å utnytte et gigabit-nettverk, med mindre data sendes ut ved bruk av kun en kopieringsoperasjon. I tradisjonelle operativsystemer sendes data ut ved bruk av flere kopieringsoperasjoner, og minnehastigheten vil derfor være en flaskehals for et slikt system.

Hvis vi tenker oss at vi skal ta i bruk et system som INSTANCE, der antall kopieringsoperasjoner minimeres, vil vi kunne utbedre denne flaskehalsen. Hvis antall kopieringsoperasjoner reduseres fra tre til en, vil det si at gjennomstrømningen av data kan økes fra 60 KB/s (180/3) til 180 KB/s. Her ser vi selvsagt bort fra at andre komponenter i systemet er en flaskehals. Vi ser også bort fra at det er andre komponenter som deler bruken av minne, som kan føre til redusert ytelse. Hvis dette hadde vært en videoserver, og filmer ble sendt ut med f.eks. 5 MB/s, ville dette ført til at serveren kunne betjene 24 flere brukere. En reduksjon i antall kopieringer, som beskrevet i INSTANCE, kan derfor vise seg å være svært lønnsomt hvis det ikke er nettverket som er flaskehalsen.

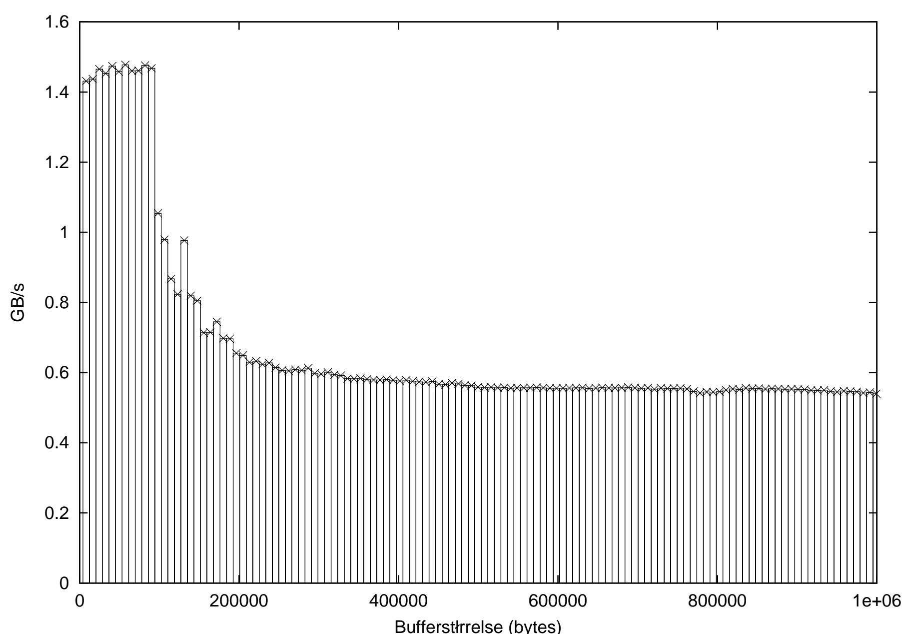
Minimum	187,54
Maksimum	188,61
Gjennomsnitt	188,05
Standardavvik	0,2598
Konfidensintervall (95%)	0,1470

Tabell 4: Resultater for kopiering med 400 KBytes buffer (MB/s)

I tabell 4 ser vi resultater fra kopiering av minne med bufferstørrelse på 400 KBytes. Fra tabellen ser vi at målingene ga jevne resultater. Dette var forventet, da kopiering av minne er en enkel operasjon og siden ingen bakgrunnslast ble brukt under målingene.

8.6 NetBSD

Figur 36 viser resultater for minnekopiering under NetBSD. Vi ser at grafen har en todeling, som i resultatene fra Chorus. Fra resultatene ser man en betraktelig forbedring sammenlignet med resultatene fra Chorus. Dette er på grunn av at maskinen som kjørte NetBSD er utstyrt med raskere maskinvare enn maskinen som ble brukt under målingene på Chorus. Resultatene viser en forbedring på omtrent fire ganger for bufferstørrelser under 200 KBytes. For større buffere er forbedringen på omtrent tre ganger. Maskinvaren til de to maskinene er beskrevet tidligere i tabell 2 og 3.



Figur 36: Kopiering av minne, NetBSD

Minimum	587,07
Maksimum	598,75
Gjennomsnitt	594,48
Standardavvik	2,8077
Konfidensintervall (95%)	1,6592

Tabell 5: Resultater for kopiering med 400 KBytes buffer (MB/s)

Tabell 5 inneholder resultater fra kopiering av minne med bufferstørrelse på 400 KBytes. Tabellen viser at også resultatene for NetBSD er relativt stabile.

8.7 Andre systemer

Under i tabell 6 ser vi resultater fra målinger av minneytelse utført på andre systemer [27]. Dette er resultater for kontinuerlig kopiering av minne, og kan sammenlignes med høyre del i grafene fra målingene på Chorus og NetBSD. Prosessorhastigheten i tabellen sier ikke noe om minneytelsen, da hastigheten for minne ofte er begrenset av hvilken type minne som brukes. Prosessorhastigheten er likevel tatt med da den indikerer fra hvilket år systemet er fra. Systemene i tabellen er PC-er basert på Intel-prosessorer og kloner (system 1-4), og Sun-servere (system 5-7). Systemene er basert på forskjellige typer arkitekturer, og er utstyrt med forskjellige typer minne.

System	**Prosessor (MHz)**	**Minne (MB)**	**Ytelse (MB/s)**
1. Intel Pentium II bastert PC	234	64	103,89
2. Pentium II basert PC	350	64	257,41
3. Pentium III basert PC	500	512	249,93
4. AMD K7 basert PC	550	128	418,54
5. Ross, RT626	150	256	67,54
6. SUNW, UltraSPARC-II	296	512	216,84
7. SUNW, UltraSPARC-IIi	333	128	193,88

Tabell 6: Minnehastighet for andre systemer

9 Diskytelse

I dette kapittelet beskriver jeg målingen av diskytelsen under Chorus og NetBSD. Først vil jeg forklare hvorfor målingene er utført. Så vil jeg beskrive hvordan de er utført, presentere resultater fra målingene og til slutt drøfte resultatene.

9.1 Problemstilling

En disk består av mekaniske deler som setter begrensninger på ytelsen til disken. Jeg skal derfor i denne delen gjøre målinger på disken for å få en formening om hvilke begrensninger denne mekaniske oppbygningen av disken setter på ytelsen. Målet med denne målingen er å kunne analysere hvor fort data kan overføres fra disk til applikasjonen, uten å sende dataene videre til nettverket. I denne målingen ser vi også på hvordan cache og plassering av data spiller inn på hastigheten.

9.2 Valg av målingsmetode

Som for målingene på minne valgte jeg av de samme grunnene å implementere et eget program som utfører diskmålingene. Likevel er mye av kildekoden for kopiering av data basert på målingsrutiner fra Iozone [20]. På denne måten kunne jeg bruke nøyaktige tidsmålinger, samtidig som jeg kunne bruke en godt testet kode for å utføre målingene. En annen grunn til at hele Iozone ikke ble brukt, var at den ikke ville la seg kompilere under Chorus. Kun idéer og deler av koden fra Iozone ble brukt. Iozone er et program for å gjøre målinger på flere forskjellige typer filoperasjoner. Operasjonene inkluderer read, re-read, backward read, strided og mange fler.

9.3 Utførelse

Målingene ble utført under Chorus og NetBSD, med filsystemet FFS (Fast File System). For at målingene skulle bli så korrekte som mulig ble det opprettet en ny tom partisjon på disken. En partisjon er en del av disken som er satt av til et bestemt filsystem, swap eller annen lagring av data. Filene som ble brukt under målingene ble så kopiert til den tomme partisjonen på disken. På denne måten ble hver fil plassert på samme område på disken, og ikke splittet opp i flere fragmenter på forskjellige fysiske posisjoner på disken. Grunnen til at filen ikke blir splittet opp, er at filsystemet alltid vil finne en ledig blokk rett etter den forrige blokken som ble allokert. Hadde filene i motsetning blitt plassert på en godt brukt partisjon, ville blokken i

filen havnet på forskjellige områder på disken, som ville økt diskforsinkelsen grunnet flytting av lesehodet. For å være sikker på at de ledige blokkene på disken ligger etter hverandre etter en formatering, er det nødvendig å sjekke dette.

Cache spiller inn på ytelsen ved overføring av data fra disk til minne. For å forsikre meg om at bruk av cache ikke skulle spille inn på målingene på diskytelsen, valgte jeg størrelsen på filene, som skulle brukes under testen, til større enn fysisk tilgjengelig minne på maskinen. På denne måten kunne jeg forsikre meg om at hele filen ikke lå i cachen under testen. Når testen så ble utført kunne jeg se hvilken verdi overføringshastigheten konvergerte mot, ved å velge stadig større størrelse på filene. Denne metoden gjør at det også går an å se hvilken innvirkning cache har på diskoperasjoner. Valg av størrelse på filene ble satt til verdier fra 2^6 til 2^{12} KBytes. Grunnen til at større filer ikke ble valgt, var at vi så at resultatene stabiliserte seg når filene kom opp i denne størrelsen.

Målingene på serveren er gjort for å se hvor mye data serveren kan levere fra seg, og derfor testet jeg bare lesing fra disk. Skriving til disk ble ikke testet, da servere skriver filer til disk sjelden i forhold til hvor ofte de leses. Når data er flyttet inn i minnet etter leseoperasjonene tas de ikke vare på, men skrives over av neste leseoperasjon. Det er ingen grunn til å ta vare på dataene, da jeg kun skal gjøre målinger på lesing fra disk, og ikke sende dataene videre ut på nettverket.

9.4 Implementasjon

Programmet som utfører målingene starter med å opprette en buffer (`BUF`), som dataene fra filen skal leses inn i. Størrelsen på dette bufferet ble variert fra 2^2 KBytes og opp til 2^{12} KBytes, for å se om størrelsen på buffere har noen innvirkning på måleresultatet. Måleresultatene er derfor variert med både størrelsen på bufferet dataene leses inn i, og størrelsen på filen.

API-et til Chorus tilbyr støtte for å registrere et tidspunkt med få instruksjoner, ved hjelp av kallet `KnChorusTime`. Dette kallet er brukt for å registrere tiden rett før og rett etter diskoperasjonene er utført. Diskoperasjonene utføres med bibliotekskallet `read`, og for hver blokk som leses inn fra filen oppdateres `inData`, som hele tiden reflekterer antall bytes som er lest inn fra filen. Ved hjelp av `inData` og tidsstrukturene som registrerer tiden før og etter målingen (`T1` og `T2`), ble så gjennomstrømningen av data regnet ut. Dette ble så gjentatt for forskjellige verdier for både bufferstørrelse og filstørrelse

som nevnt tidligere. Resultatet ble lagret i minne, og etter at alle målingene var utført ble de skrevet til en kommaseparert (CSV) fil, for å lett kunne fremstille resultatene grafisk. Resultatene ble skrevet til CSV-filen etter at alle målingen var utført, slik at skriving av CSV-filen ikke skulle ha noen innvirkning på resultatene. CSV-filen ble så importert i Excel, for å kunne fremstille målingsresultatene grafisk.

Kjernen i koden for målingene er som følger:

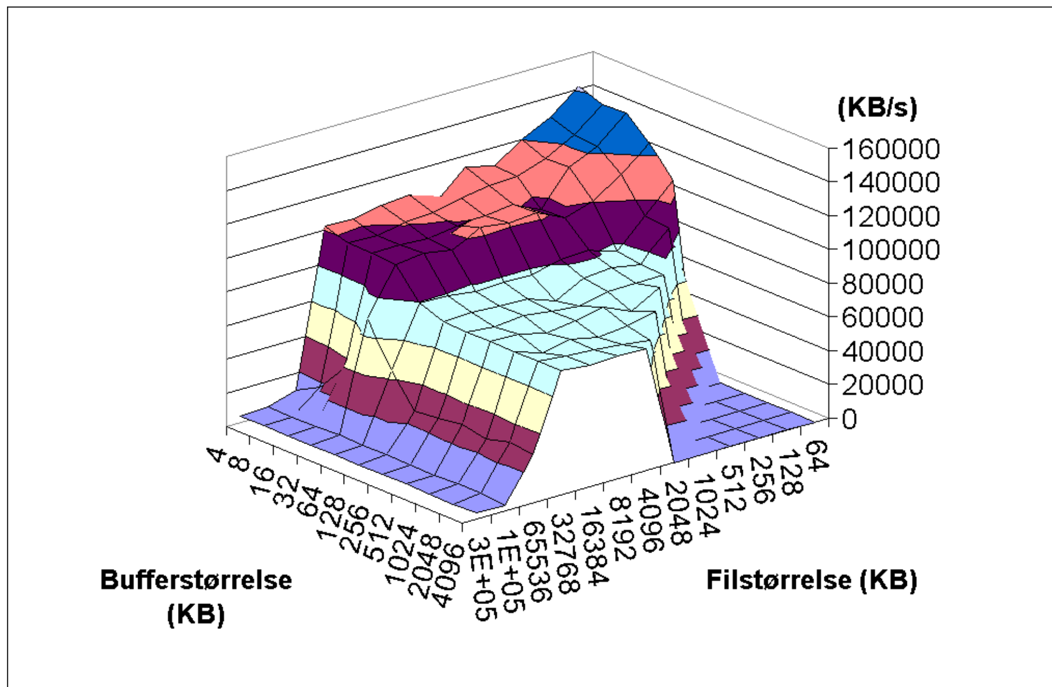
```
inData = 0;
KnChorusTime(T1);
while (!eof(probeFile))
{
  inData += read(probeFile, BUF, BUF_SIZE);
}
KnChorusTime(T2);
probeTime = TimeDiff(T1, T2);
probeThroughput = inData / probeTime;
```

9.5 Resultater

Resultatene for målingene presenteres som et sett av fire grafer for både Chorus og NetBSD. Først kommer to grafer som viser diskhastigheten for lesing av filer (read). Filene er her skrevet til disk rett før de leses inn. Disse to grafene er generert fra samme data, men siden det er vanskelig å se regionen hvor ikke noe av dataene ligger i cache i den første grafen, er denne regionen tatt ut i en egen graf, som er den andre grafen. De to neste grafene er like de to første, men her er hele filen lest en gang før målingen (re-read), og ikke bare skrevet til disk som i forrige måling. Forskjellen i ytelse til disse to målingene kan være betydelig, avhengig av hvilken metode algoritmen for cache-utskiftning bruker for lesing og skriving til disk. Andre faktorer som kan spille inn er hvordan read-ahead og write-behind utføres.

9.6 Chorus

Grafen i figur 37 viser resultater for lesing av filer fra disk til minne. X-aksen, nede til høyre i figuren, viser størrelsen på filen som ble brukt under målingen. Y-aksen, nede til venstre, viser størrelsen på minnebufferet (`BUF_SIZE`) som data fra filen ble hentet inn i. Fra figuren kan vi se at den er delt opp i flere regioner som har tilnærmet lik overføringshastighet. Vi kan se fem slike regioner i figuren. Regionene er som følger:



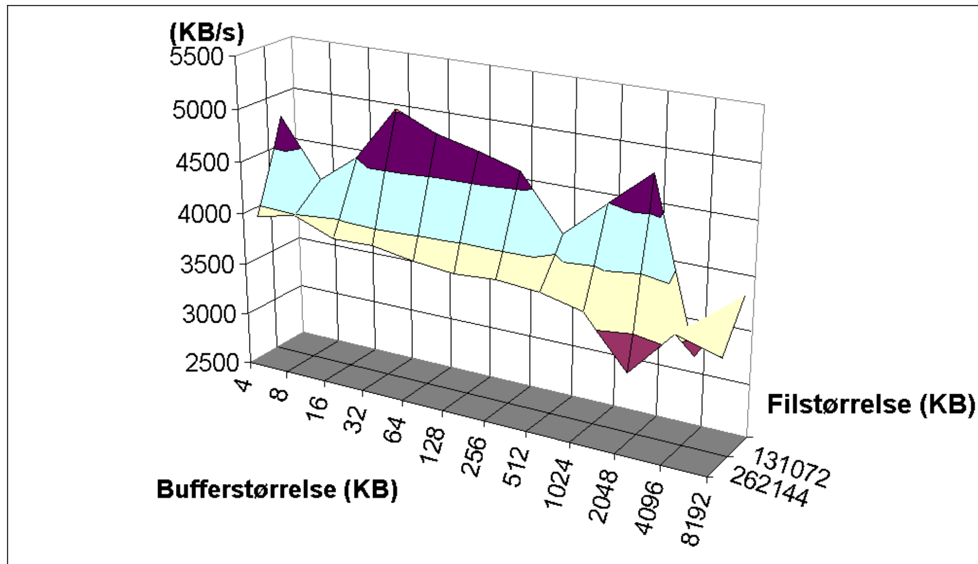
Figur 37: Lesing fra disk (read)

- Region 1 kan vi se som en topp bakerst i figuren. Regionen har den største gjennomstrømningen av data, med hele 140 KB/s. Dette er små filer, og av den grunn vil mye av dataene i disse filene ligge i cachen. En annen grunn til at man får så gode verdier her, er at deler av filen også kan ligge i prosessorens cache, som er en veldig rask cache.
- Region 2 er platået til venstre i figuren avgrenset av bufferstørrelser opp til 64 KBytes. I denne regionen ser vi at største gjennomstrømning av data ligger på 110 KB/s. Hvis vi ser på dette i forhold til region 1, så har region 1 nesten 30% større gjennomstrømning av data enn denne regionen. Årsaken til det er at mindre av filen befinner seg i prosessorens cache, som er raskere enn minnecachen.

- Region 3 er platået midt i figuren avgrenset av bufferstørrelser fra 64 KBytes og oppover. I denne regionen ser vi en ytterligere reduksjon i gjennomstrømningen. Her ligger gjennomstrømningen på ca. 70 KB/s. Region 2 er over 50% raskere enn region 3, og region 1 er 100% raskere. Grunnen til at region 3 har lavere gjennomstrømning enn region 2, kommer av at minnekopiering på dette systemet var tregere for store buffere enn små. Ser vi tilbake på grafen fra minnekopiering (figur 35), kan vi se likhetstrekk mellom denne og overgangen mellom region 2 og 3.

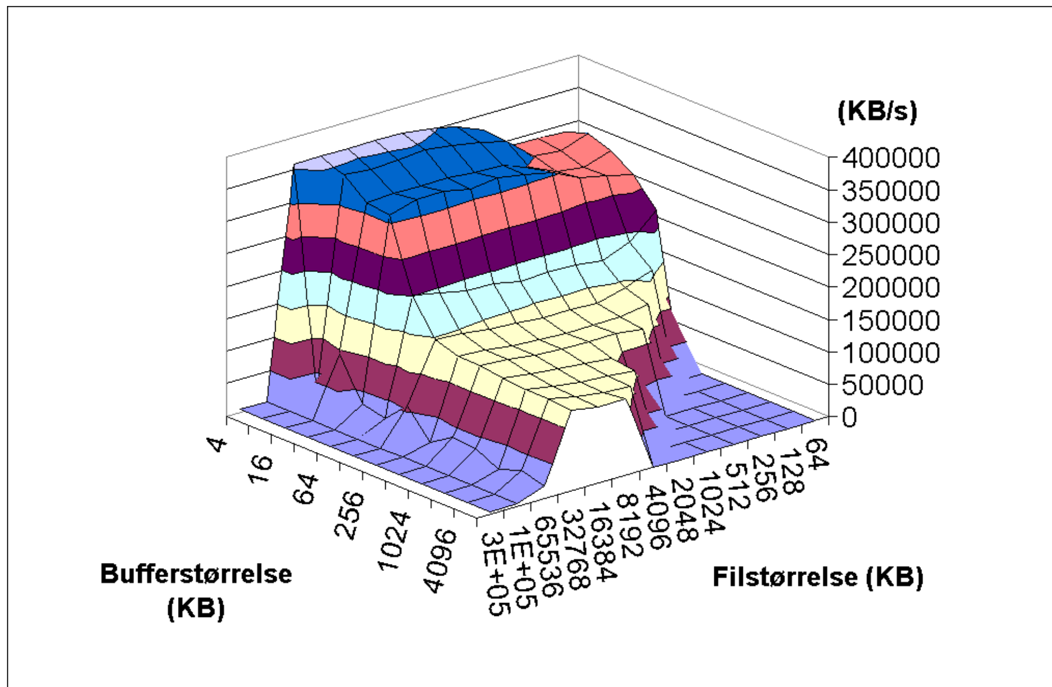
- Region 4 er avgrenset av platået nede til venstre i figuren. Fra region 2 og 3, og ut mot region 4 kan vi se at grafen faller betraktelig. Her er en mindre og mindre andel av filen tilgjengelig i cachen. Grunnen til dette er at filen blir større og større, og tilgjengelig cache er konstant på grunn av begrensningen på fysisk tilgjengelig minne. Det er vanskelig å se verdiene til resultatene i region 4, siden resultatene her er så lave i forhold til resten av grafen. Jeg har derfor plukket ut resultatene fra denne regionen og presenterer de i en egen graf (figur 38).

- Region 5 er avgrenset av platået nede til høyre i figuren. Region 5 i grafen er ikke data, siden jeg i testen unnlot å teste filer som var mindre enn bufferstørrelsen, da dette ikke er et interessant dataområde. Disse verdiene er derfor satt til 0 i grafen.



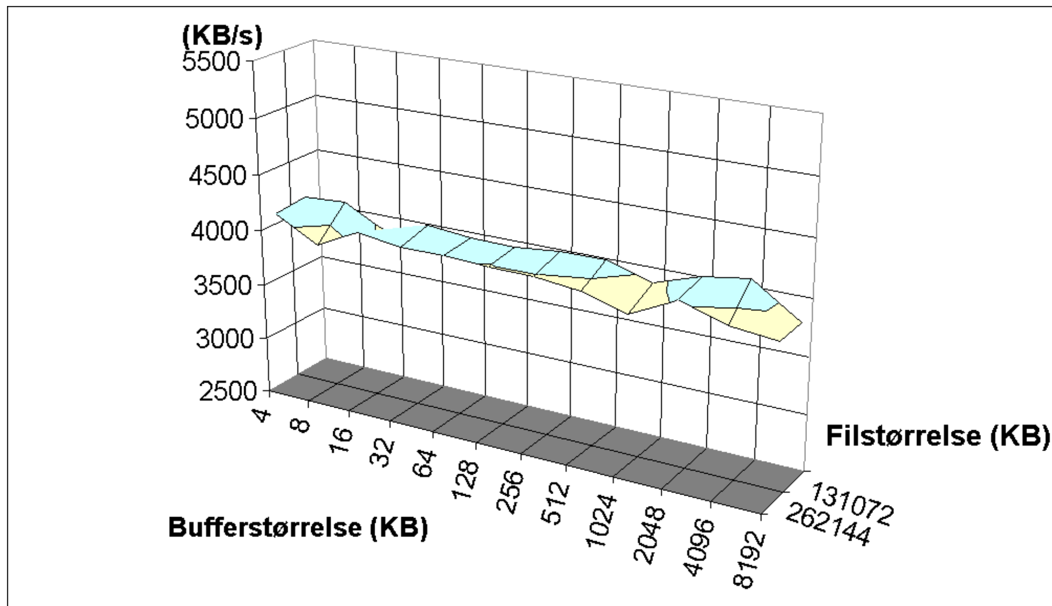
Figur 38: Lesing fra disk. (read, region 4)

I figur 38 ser vi region 4 fra den første grafen (figur 37). Vi ser her at gjennomstrømningen av data ligger mellom 3 MB/s og 5 MB/s. Når størrelsen på filen settes til 262 MBytes, ser vi at gjennomstrømningen ligger like under 4 MB/s. Her er filen større enn fysisk tilgjengelig minne, og derfor vil lite av filen være tilgjengelig i cachen. Prøver vi med enda større filer holdt resultatet seg på samme nivå. Det vil si at når filen ikke ligger i cache, vil overføringshastigheten til disken ligge på i underkant av 4 MB/s. Grunnen til at resultater for større filer ikke er tatt med i grafen er at disklesing av de største filene tok lang tid. Gjennomstrømning ble likevel testet for disse filene, men kun for 8 og 64 KBytes store buffere. En hastighet på 4 MB/s tilsvarer 32 Mbit/s. Med denne hastigheten kan vi si at disken er en betydelig flaskehals, dersom dette systemet hadde benyttet seg av nettverk som 100Mbit og gigabit-ethernet.



Figur 39: Lesing fra disk. (re-read)

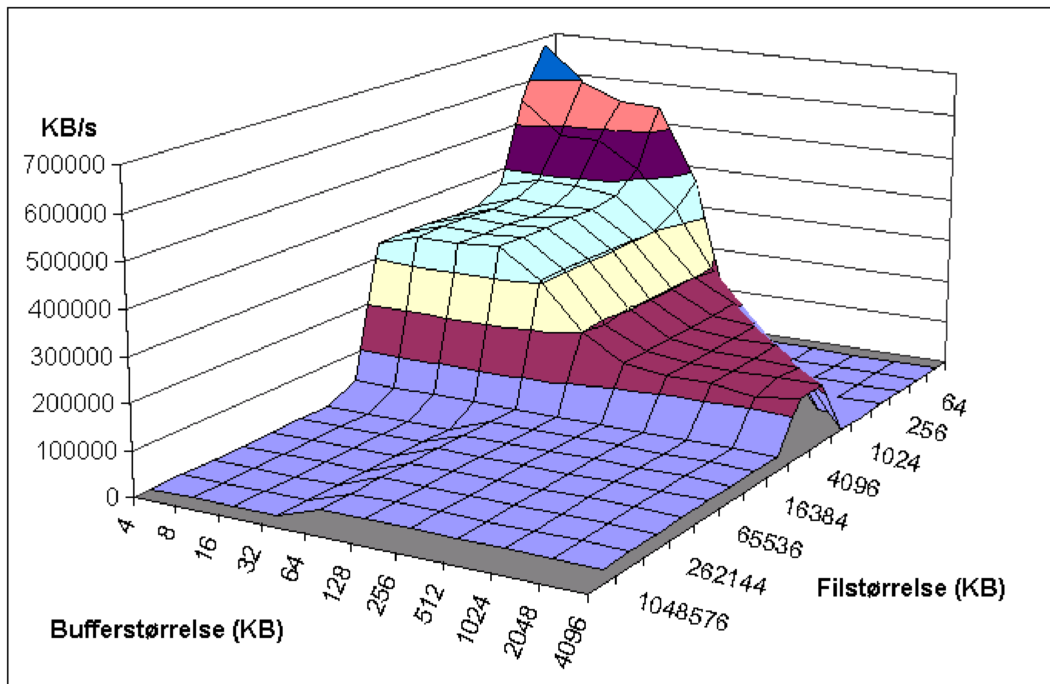
I de to forrige grafene ble filen lest rett etter at den var skrevet til disk. Disse to grafene representerer derfor lesing av en fil som nettopp er skrevet til disk. Vi skal nå se på resultater for lesing av filer som istedenfor å nettopp være skrevet er nettopp lest. På denne måten kan vi se hvordan hastigheten til disklesing er avhengig av algoritmene for cache-utskiftning og read-ahead. Figur 39 viser denne grafen. Her ser vi at gjennomstrømningen har økt betraktelig. Gjennomstrømningen er mer enn det dobbelte av gjennomstrømningen i den forrige målingen (figur 37). Ser vi nøyere på grafen kan vi se at gjennomstrømningen ligger nesten like høyt som resultatene for minnekopiering. Det kan tyde på at kopiering fra cachen tilsvarer omtrent en kopieringsoperasjon. Det viser at minnebussen avgjør hastigheten til disklesing når hele filen ligger i cache.



Figur 40: Lesing fra disk. (re-read, region 4)

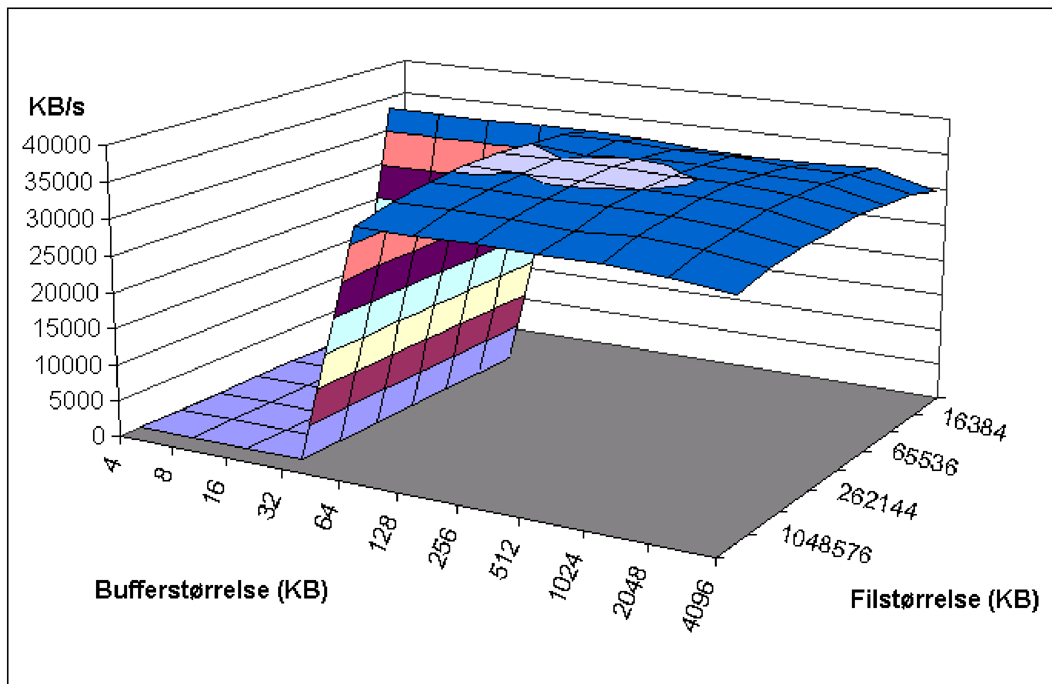
I grafen i figur 40 ser vi region 4 fra grafen i figur 39. Vi ser her at resultatene stabiliserer seg på samme verdi som i forrige måling (figur 38). Resultatene i denne grafen ser ut til å stabilisere seg raskere. Det kan tyde på at algoritmen for cache-utskiftning tar vare på blokker bedre under lesing fra disk enn for skriving til disk.

9.7 NetBSD



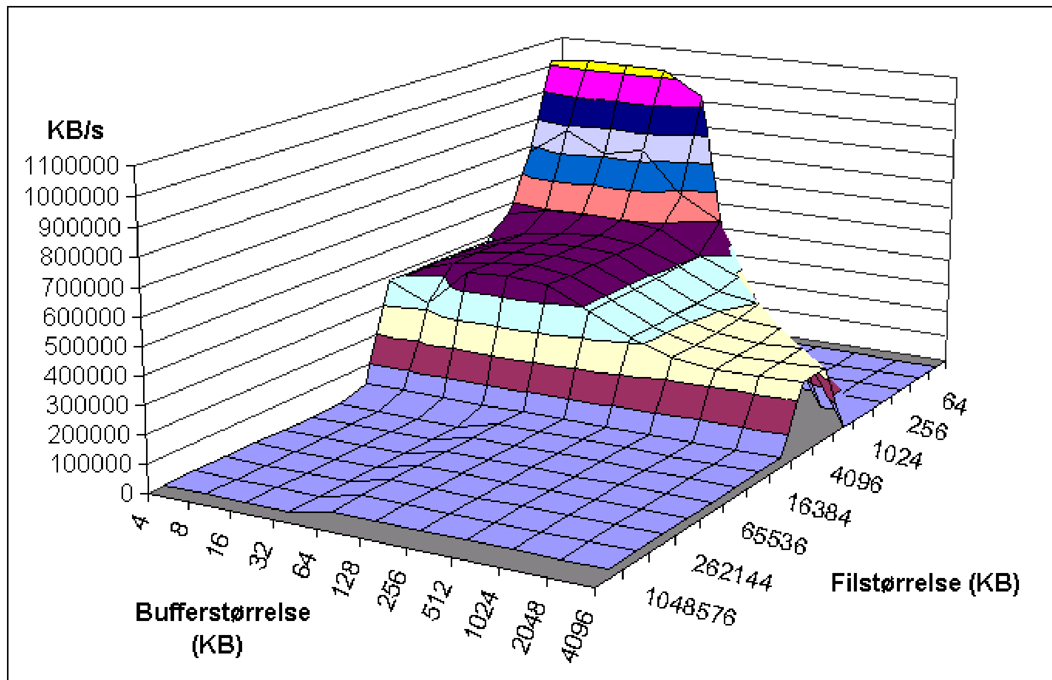
Figur 41: Lesing fra disk (read)

Figur 41 viser resultater for disklesing under NetBSD. Vi ser her at gjennomstrømningen av data har økt betraktelig i forhold til resultatene fra Chorus (figur 37). Grunnen til denne økningen i forhold til Chorus er ikke på grunn av operativsystemet, men på grunn av at den underliggende maskinvaren er forskjellig på disse to systemene. Maskinvaren som de to systemene er bygd opp av er vist i tabell 2 og 3. Formen på grafen er lik de resultatene som jeg fikk under Chorus. Eneste forskjellen er at målingene også ble gjort for større filer, da disken under NetBSD var rask nok til å utføre disse målingene.



Figur 42: Lesing fra disk. (read, region 4)

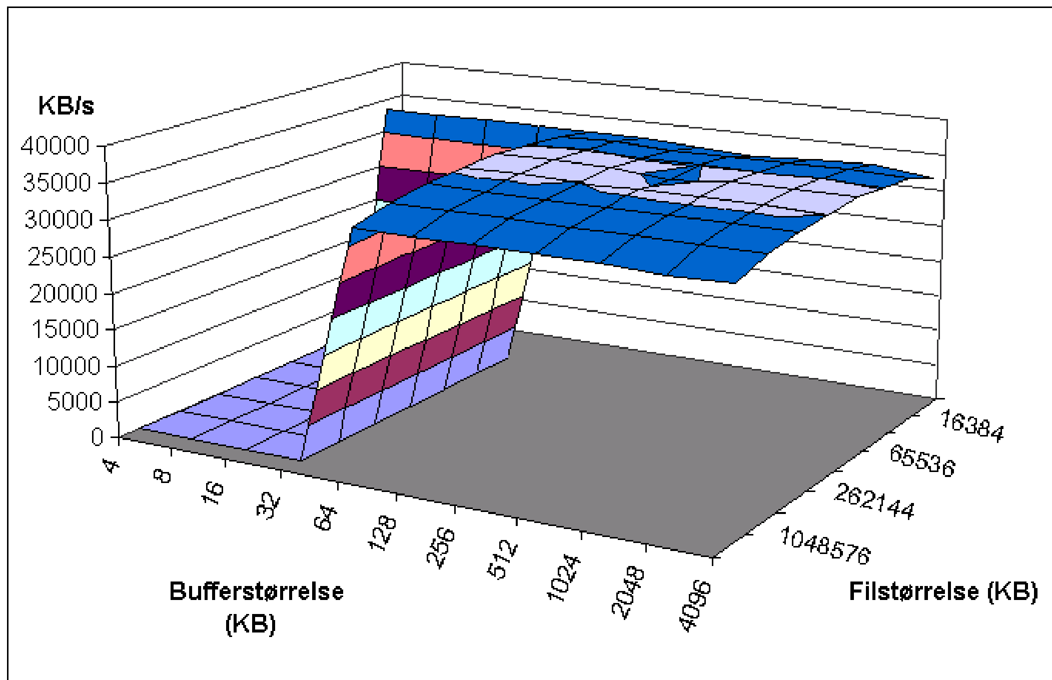
Figur 42 viser region 4 fra grafen i figur 41, som er den regionen hvor filen ikke er tilgjengelig i cachen. Her ser vi at overføringshastigheten ligger i overkant av 30 MB/s. Dette er en god forbedring fra resultatene på Chorus, hvor resultatene lå i underkant av 4 MB/s. Dette viser at disken som ble brukt under Chorus ikke var særlig rask. Resultatet på 30 MB/s tilsvarer 240 Mbit/s. Dette er nok til å utnytte et 100Mbit ethernet fullt ut, men under et gigabit-ethernet vil disken i dette systemet være en betydelig flaskehals.



Figur 43: Lesing fra disk (re-read)

I figur 43 ser vi resultater fra lesing fra disk, hvor vi først har lest filen fra disk en gang etter den er skrevet til disk. Vi kan se at i region 1 ligger overføringshastigheten på i underkant av den maksimale minneytelsen til NetBSD. Vi ser videre at grafen synker betraktelig ned mot region 2. Resultatene her ligger i underkant av verdiene på den høyre siden av den todelte figuren (36) for minneytelsen til NetBSD. Dette viser at mye av diskdataene ligger i minnecachen. Videre ser vi at ytelsen synker ned mot region 4, hvor diskdataene ikke hentes fra cache, men direkte fra disk.

Resultatene fra denne grafen viser en god forbedring av gjennomstrømning av data i forhold til Chorus. Dette er på grunn av maskinvaren systemene er bygd opp av, som nevnt tidligere.



Figur 44: Lesing fra disk. (re-read, region 4)

Grafen i figur 44 viser region 4 fra grafen i figur 43, som er gjennomstrømning av data når dataene ikke er tilgjengelig i cachen. Vi ser her at gjennomstrømningen stabiliserer seg på i overkant av 30 MB/s som i forrige måling (figur 42).

10 Nettverksytelse

Målinger på nettverksytelsen er delt opp i to forskjellige målinger. Den første målingen (avsnitt 10.1) er en måling hvor jeg prøver å finne den maksimale ytelsen til nettverkskoden uten å være avhengig av hastigheten til nettverket som systemet opererer under. For å kunne gjøre en slik måling benytter jeg meg av loopbackgrensesnittet, som jeg beskriver i mer detalj senere i kapittelet. Den neste målingen (avsnitt 10.2) tar hensyn til hastigheten til nettverket, og ser på hvordan begrensningene nettverket setter på hastigheten spiller inn på behandling av data som sendes ut fra applikasjonen. Denne målingen ser også mer detaljert på flyten av data gjennom de forskjellige lagene i nettverkskoden. Den ser på hver enkel komponent i nettverkskoden, som beskrevet i kapittelet om dataflyt gjennom nettverket.

10.1 Loopback

I denne målingen ser vi på hastigheten til nettverkskoden uten å være avhengig av hastigheten til det underliggende nettverket. Vi skal først se på hvorfor og hvordan jeg skal utføre en slik måling. Til slutt vil jeg gå gjennom resultater fra målingene.

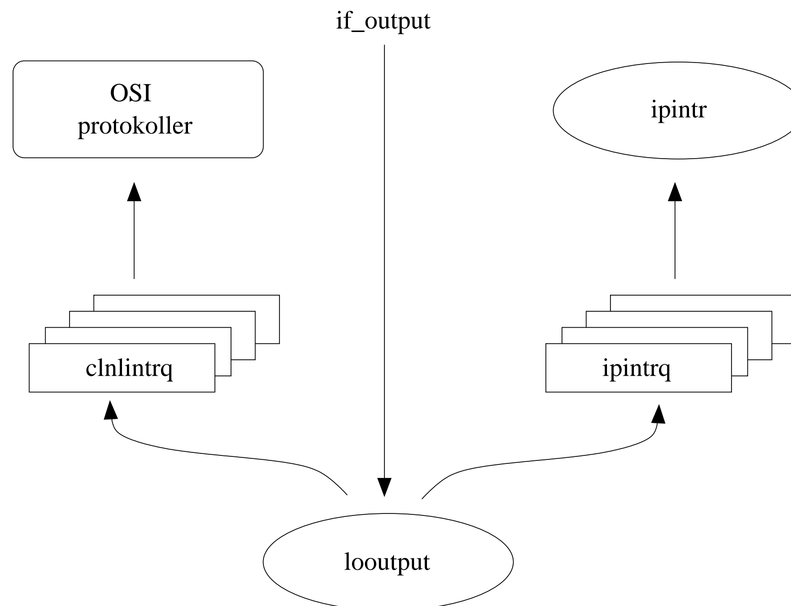
10.1.1 Problemstilling

For å kunne vite hvor mye data nettverkskoden i operativsystemet er i stand til å sende ut kan man sende data fra minnet og ut på nettverket så fort man klarer. Problemet med dette er når nettverkskoden ikke klarer å sende fort nok på grunn av begrensninger i hastigheten på nettverket. Løsningen er da å oppgradere nettverket til et raskere nettverk, f.eks. fra 10 Mb/s ethernet til 100 Mb/s ethernet. Under Chorus var ikke dette noe valg, da den ikke støttet denne type nettverkskort, og selv med 100 Mb/s (12,5 MB/s) kunne nettverket blitt en flaskehals. For å likevel kunne avgjøre hvor mye data nettverkskoden er i stand til å sende ut tok jeg i bruk loopbackgrensesnittet.

10.1.2 Loopbackgrensesnittet

De fleste implementasjoner støtter et loopbackgrensesnitt, som tillater en klient og tjener på samme maskin å kommunisere med hverandre ved bruk av TPC/IP. Klasse A nettverk 127 er reservert for loopbackgrensesnitt, og de fleste systemer knytter IP-adresse 127.0.0.1 opp mot denne og gir den navnet localhost. Et IP-datagram som sendes til loopbackgrensesnittet skal ikke gå ut på nettverket. Alle pakker som sendes til loopbackgrensesnittet blir med

en gang lagt til i køen for innkommende pakker, som vist i figur 45. De blir altså ikke lagt til i køen for utgående pakker. Loopbackgrensesnittet er implementert i programvare, og bruker ingen form for maskinvare (nettverkskort). Selv om vi kunne tenkt oss at transportlaget kunne oppdaget at den andre enden av forbindelsen er en loopbackadresse, og dermed kuttet ut å sende pakken gjennom transportlaget, er det ikke slik det gjøres. De fleste implementasjonene utfører hele behandlingen av pakken i transport- og nettverkslaget, og sender pakken tilbake til sin egen innkommende kø når den når bunnen av nettverkslaget. Selv om det kan virke lite effektivt å behandle loopbackpakker gjennom hele transport og nettverkslaget, forenkler det designet, siden loopbackgrensesnittet opptrer som et hvilket som helst annet linklag for nettverkslaget.



Figur 45: Loopbackgrensesnittet i nettverkskoden

10.1.3 Utførelse

For målingene av loopbackytelsen ble det opprettet en buffer (`BUF`) med størrelse `BUF_SIZE`. Så ble det opprettet en socket til adressen 127.0.0.1, som er loopbackgrensesnittet. Sendebufferet til socketen ble så satt til verdien `SS` (socket size), der $\texttt{SS} = 2^n$ og $n \in \{12, 13, ..., 18\}$. Grunnen til at forskjellige størrelser på socketens sendebuffer ble brukt var for å se om dette kunne

spille inn på ytelsen til nettverkskoden. Tiden ble så registrert i en tidsstruktur (T1). Data ble så sendt til en socket opprettet av et annet program på samme maskin i størrelser på BUF_SIZE bytes. Når dataene var sendt ble tiden registrert i en ny tidsstruktur (T2). Dette ble så gjentatt i en løkke, for å kunne avgjøre hvor stabile målingene var. Målingene ble også kjørt for forskjellige verdier av BUF_SIZE, for å se om dette hadde noen innvirkning på resultatene.

Selv om kallet for å sende dataene er utført, er det ikke sikkert at alle data er sendt og mottatt av klienten. For å sørge for at dette ikke skulle ha noe å si, ble det sendt så store mengder data at dette ikke påvirket resultatet.

Kjernen i målingene er som følger:

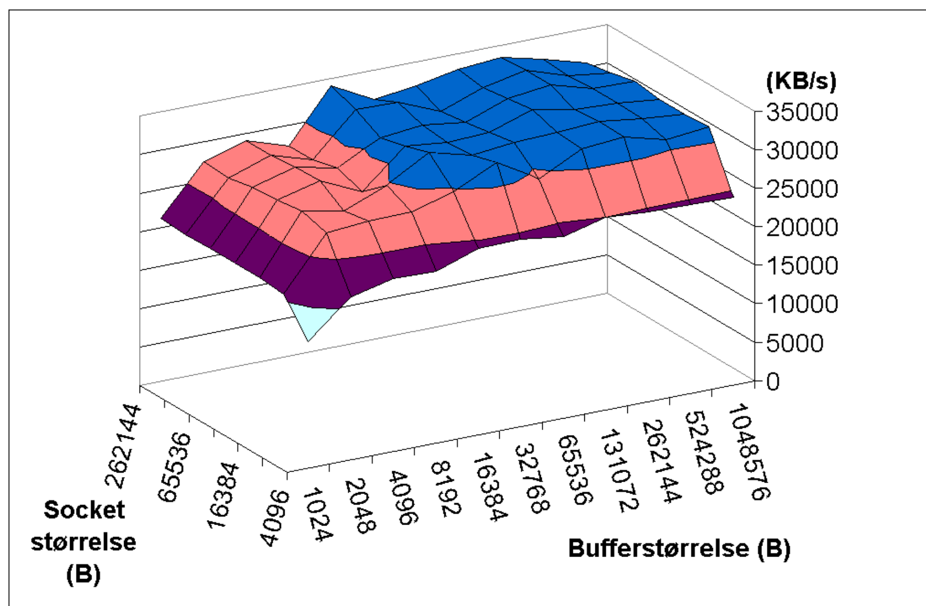
```
int sent = 0;
int sentNow = 0;

KnChorusTime(T1);
while(sent < sendSize)
{
  sentNow += write(sock, BUF, BUF_SIZE);
  if (sentNow != -1)
    sent += sentNow;
  else
    sendError();
}
KnChorusTime(T2);
```

10.1.4 Resultater

Vi skal nå se på resultatene fra loopbackmålingene for Chorus og NetBSD. Resultatene er presentert i en graf som viser maksimal gjennomstrømning av data. Grafen viser gjennomstrømning for forskjellige størrelser på socketbufferet og bufferet som brukt for sending av data (BUF).

10.1.5 Chorus

Figur 46 viser resultater fra loopbackmålingen, variert med størrelsen på socketens sendebuffer og størrelse på bufferet med dataene som skal sendes. Vi ser her at resultatet ligger på ca. 30 MB/s. Dette er mye raskere enn resultatene fra disklesingen som lå på 4 MB/s. Men ser vi på disklesing med cache, som lå på 70 MB/s og oppover, var disklesing raskere enn å sende pakker gjennom nettverkskoden. Dette avhenger selvsagt av hvor mye av filen som ligger i cachen. I en webserver som har sider som brukes ofte, vil man ha mange filer i cachen, mens det i en videoserver vil være færre treff i cachen. Hvis vi sammenligner hastigheten med minnekopiering, ser vi at nettverkskoden tilsvarer ca. 6 (180/30) minnekopieringer. Vi må her huske på at data ble sendt både ned til nettverkskoden og opp til applikasjonen igjen. En annen ting vi må huske på, er at pakkene ikke legges i utgående kø, men legges direkte inn i den innkommende køen. Vi skal derfor i målingen i neste kapittel se på hastigheten på nettverkskoden kun en vei, og ser mer detaljert på hva som skjer med pakkene inne i nettverkskoden. På denne måten kan vi gjøre oss opp en formening om hvor mye det har å si at pakkene i denne testen ikke legges i den utgående køen for nettverkskortet.

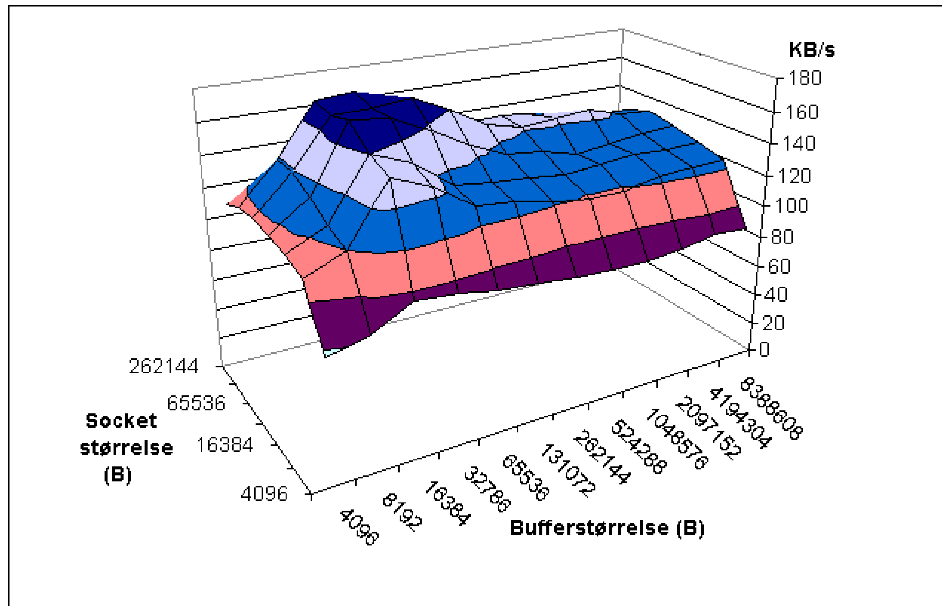


Figur 46: Loopbackresultater, Chorus

I grafen kan vi også se hva som skjer når vi endrer størrelsen på socketens sendebuffer og størrelsen på bufferne som sendes. Av grafen ser vi at når vi øker socketens sendebuffer, så øker gjennomstrømningen av data helt opp til størrelser på 16 KBytes. For større verdier på socketens sendebuffer er økningen av gjennomstrømningen mye mindre. Det kan derfor være lurt å sette socketens sendebuffer til 16 KBytes. Settes den til mer vil det brukes mer minne på hver forbindelse, selv om forbedringen av hastigheten ikke øker noe særlig. Disse resultatene avhenger selvsagt av hvor god forbindelsen er, og avstanden frem til mottakeren av dataene. Er avstanden lang og båndbredden høy, kan det være lurt med større sendebuffer, men for vårt tilfelle var 16 KBytes et godt valg.

Hvis vi ser på nettverkskodens hastighet i forhold til bufferstørrelsen til dataene som ble sendt, ser vi at grafen ikke stemmer overens med grafen for kopiering av minne. Her ser det ut som det er lurt å bruke større blokker når man skal sende dataene. Grunnen til dette er at nettverkskoden splitter opp dataene opp i flere mbuffere. Når datamengden som sendes til nettverkskoden bare tilsvarer et fåtall mbuffere, vil det være mange mbuffere som ikke er fulle når de legges i den utgående køen. F.eks. hvis vi sender 3 KBytes med data, vil disse få plass i to mbufferclustere på 2 KBytes. Hvis vi istedenfor hadde sendt 17 KBytes, ville de fått plass i ni mbufferclustere på 2 KBytes. I det første eksempelet ville vi kastet bort 25% av sendebufferets plass, mens vi i det andre eksempelet ville kastet bort 6% av plassen.

10.1.6 NetBSD



Figur 47: Loopbackresultater, NetBSD

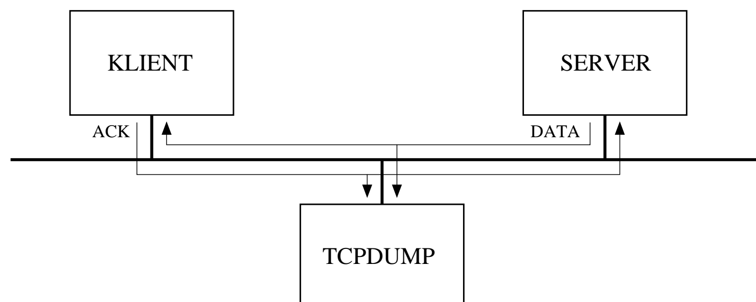
Fra grafen i figur 47 kan vi se at den maksimale gjennomstrømningen for loopbackgrensesnittet under NetBSD ligger i underkant av 170 MB/s. Dette er 1300 Mbit/s, som er raskere enn hastigheten til et gigabit-ethernet. Dette viser at kjernen kan sende ut pakker raskere enn hastigheten til nettverket. Dette er hvis serveren bruker alle minneressurser og all prosessorkraft på sending av pakker, og ikke på noe annet. Derfor kan vi anta at man i en multimediaserver, som også må bruke tid å overføre data fra disk og behandle dataene, likevel ikke kan være sikker på å oppnå hastigheter over grensen til nettverket.

10.2 JProbe

For å få en mer detaljert oversikt over hvordan data flyter gjennom kjernen når den går fra applikasjon til nettverk, har jeg lagt inn prøver i kjernen til Chorus og NetBSD, som gjør målinger i koden til nettverksdelen i kjernen. Systemet som utfører disse målingene har jeg valgt å kalle JProbe. I dette kapittelet beskriver jeg hvorfor jeg har valgt å bruke JProbe, hvordan jeg vil utføre målingene, implementasjon, resultater og til slutt en drøfting av resultatene.

10.2.1 Valg av målingsmetode

Det finnes mange verktøy for å måle ytelsen til nettverket. Kjente verktøy er Tcpdump [34], Netmon [6] og Netperf [21]. Tcpdump er et verktøy for å fange opp pakkehoder på nettverket og samle opp informasjon om de pakkene som tilfredstiller valgte kriterier. På denne måten kan man overvåke hvordan data flyter mellom to eller flere noder i et nettverk. For å unngå forstyrrelser forårsaket av Tcpdump, er det vanlig å kjøre programmet på en egen maskin som ikke er involvert i den overvåkede kommunikasjonskanalen. Figur 48 viser en typisk konfigurasjon for Tcpdump.



Figur 48: Typisk maskinoppsett for Tcpdump

Tcpdump har to fordeler. For det første forstyrrer den ikke målingene da den kjører på en egen maskin. For det andre er det ikke nødvendig å gjøre noen endringer i kildekoden til serveren for å kunne utføre målingene på overføring av data. Ulempen med Tcpdump er at man ikke får noen informasjon om hva som skjer gjennom nettverkskoden, kun informasjon knyttet til pakkene etter at de er sendt ut på nettverket. Man får ikke noen informasjon om hvordan pakkene blir køet internt i serveren før de sendes ut, og man kan derfor ikke finne ut hva det er som setter begrensninger på

hastigheten til overføringen.

Netmon er et verktøy for å hente ut protokollinformasjon for overvåkning og analyse. Netmon setter prøver i kjernens kode for å hente ut informasjon om aktiviteten i koden til protokollene. En brukerprosess blir så satt opp for å hente ut denne informasjonen, og lagre den til disk. Netmon overvåker kun informasjon om IP-delen av protokollkoden og informasjon om nettverkskøen. Ingen prøver overvåker høyere lag i nettverkskoden, og Netmon gir derfor ikke en fullstendig oversikt over dataflyten. Netmon var også utviklet for å hente ut informasjon som ikke er nødvendig for å gjøre målinger på nettverkskoden, slik at den bruker unødvendig tid under målingene. Dette kan ha innvirkning på måleresultatene og er ikke ønskelig.

Netperf er et verktøy som utfører målinger av prosessorforbruk og kontinuerlig nettverksytelse for flere typer protokoller, inkludert TCP og UDP. Netperf bruker en klient-tjener arkitektur. Klientprogrammet startes på en maskin og kobler seg opp mot tjeneren. Gjennomstrømningen måles så ved at klienten kaller `send` gjentatte ganger. Resultatet av overføringen gis så ut med statistikk med beregnet gjennomsnitt og konfidensintervall.

Selv om disse verktøyene er veldig nyttige for å gjøre målinger på ytelsen til nettverket, så viser det seg at de ikke er brukbare for å undersøke interne hendelser i selve nettverkskoden. Dette var grunnen til at et eget system (JProbe), for å utføre målinger på ytelsen til nettverkskoden, ble utviklet.

10.2.2 JProbe-prøver

Verktøy som Tcpdump og Netmon fokuserer på overvåkning av spesielle lag i kommunikasjonskanalen, og ser derfor ikke på kommunikasjonsmekanismen som en helhet. Et verktøy som overvåker alle kommunikasjonslagene samtidig vil gi bedre innsikt i den interne behandlingen av kommunikasjonsforbindelsen. Et slikt verktøy kan undersøke samhandlingen mellom de forskjellige komponentene, og utdype aspekter som ikke er synlige når man undersøker komponentene hver for seg.

JProbe er utviklet for å kunne overvåke dataflyten gjennom nettverkskoden ved å plassere prøver på strategiske punkter i kjernen. En prøve er en liten kodebit som plasseres i kjernen for å registrere data som flyter gjennom dette punktet i koden. Prøvene har egne datastrukturer for å kunne lagre informasjonsflyten, og rutiner for å behandle disse dataene. Vi skal i de neste kapitlene se på prøvenes innvirkning på resultatet, plassering, aktivering og

beskrive hvordan prøvene er implementert.

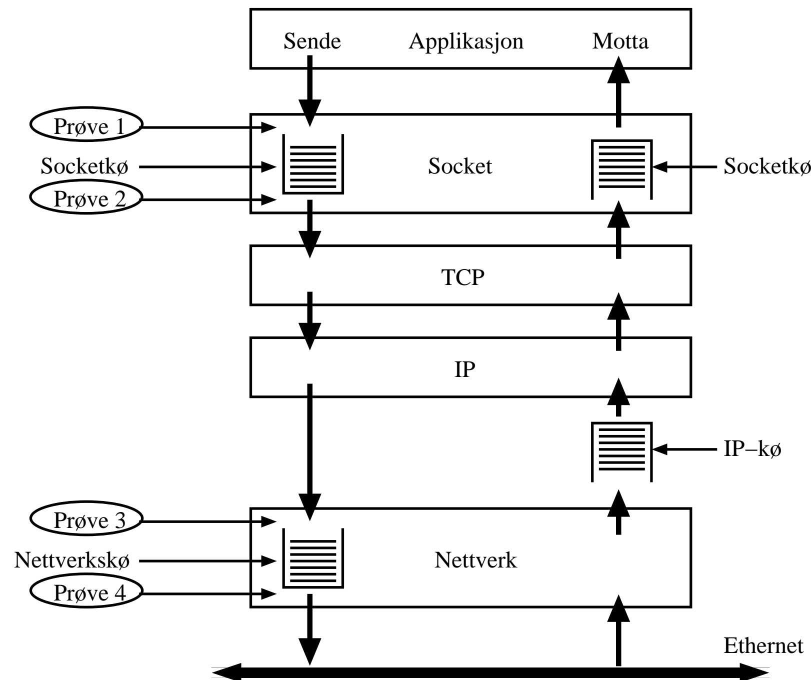
10.2.3 Prøvenes innvirkning på resultatet

Et viktig poeng med målingene er at prøvene skal ha så liten innvirkning som mulig på resultatene. Det er da viktig å minimere antall operasjoner som utføres av prøvene. Det er tre steder man kan redusere prøvenes innvirkning på resultatene, der prøvene blir utført, der prøvene blir aktivert og der resultatene fra prøvene hentes ut. For å minimere tidsbruken til utføringen av prøvene, registreres kun et fåtall variabler i prøvene. Det som registreres er klokkeslettet prøven er tatt og lengden på dataene som passerer prøven. For å redusere innvirkningen av aktiveringen av prøvene, blir prøvene nullstilt før hver måling ved hjelp av egendefinerte kall til nettverkskoden. For å sørge for at logging av dataene skal ta minst mulig tid, ble det satt av fire buffere i kjernen hvor hver av prøvene kunne legge resultatene sine før de ble hentet ut og lagret på disk. Grunnen til at det ble brukt fire buffere, og ikke en ringbuffer som det ble brukt under målingene på SunOs [6], er at det er mer effektivt med fire kontinuerlige buffere enn en ringbuffer. Grunnen til det er at når dataene ble lagret i kun én ringbuffer, må man også lagre hvor prøven er tatt. Ved å bruke fire buffere kan prøven plasseres i det bufferet som er assosiert med stedet prøven er tatt. Man kan f.eks. si at alle prøver som er tatt rett før socketkøen skal plasseres i buffer nummer en.

10.2.4 Plassering av prøver

To prøver brukes for å identifisere en kø i nettverkskoden. Den første brukes for å registrere data som blir lagt til i køen, og den andre registrerer data som blir fjernet fra køen. Valg av plassering av prøvene er vist i figur 49.

I figuren ser vi fire prøver som er knyttet til socket- og nettverkskøen i den utgående kommunikasjonskanalen til serveren. Prøve 1 og 2 lagrer informasjon om pakker som går inn og ut av socketkøen, mens prøve 3 og 4 lagrer informasjon om pakker i nettverkskøen. Når prøvene er aktivert vil hver prøve lagre et tidsstempel og hvor mye data som passerer prøven. Informasjonen som lagres er minimal, men kan likevel gi god informasjon om køene. Tidstempelet kan gi informasjon om hvor lenge pakkene ligger i køen, hvor lang køen er og hvor fort pakker går inn og ut av køen. Datalengden gir informasjon om gjennomstrømningen og hvor pakker blir fragmentert.



Figur 49: Plassering av JProbe-prøver i kjernen

10.2.5 Aktivering og henting av data fra prøver

For å aktivere prøvene kaller applikasjonen `setsockopt`. Dette er et systemkall for å sette egenskaper til socketen, som f.eks. størrelsen på socketens sendebuffer. Dette systemkallet ble utvidet med et ekstra valg, `SO_JPROBE`, som kan brukes for å aktivere og nullstille prøvene. Ved f.eks. å kalle `setsockopt` med parameteren `SO_JPROBE` og verdien 1, vil prøvene bli nullstilt. På denne måten kan prøvene nullstilles mellom hver måling.

For å hente ut prøver kaller applikasjonen `getsockopt`. Dette er også et systemkall for socketens egenskaper, men med dette kallet henter man ut egenskaper istedenfor å sette egenskaper. Denne funksjonen ble også utvidet med valget `SO_JPROBE`, slik at når `getsockopt` blir kalt med parameteren `SO_JPROBE` og verdien `jAction`, vil resultater fra prøvene hentes ut. Forskjellige verdier av `jAction` brukes for å angi hva man vil hente ut, for eksempel om man vil hente ut datamengden eller tidsstempelet. Andre verdier av `jAction` gir muligheten for å velge hvilken prøve man vil hente data fra, og for å finne ut hvor mange prøver som er registrert. På denne måten kan prøvene hentes ut og lagres på disk for videre behandling etter at målingene

er gjennomført.

10.2.6 Prøvestruktur

```
if (jProbeCounter < jProbeSize) // probe is activated?
   {
     s = splimp(); // make sure nobody disturb us
     sysTime(&jProbe[jProbeCounter]); // record time
     jProbeLen[jProbeCounter] = len;  // datalength
     jProbeCounter++;                 // next probe
     splx(s);
   }
```

Over ser vi et eksempel på en prøve. Først sjekker den om prøven er aktivert, ved at `jProbeCounter` er satt til en verdi som er mindre enn `jProbeSize`. Denne testen sørger også for at prøven ikke registrerer informasjon utenfor maksimumsgrensen for antall registreringer det er plass til i prøven. Etter denne testen utføres `splimp`. `splimp` er en funksjon som sørger for å blokke for interrupter som kan forstyrre målingene som blir tatt. `splx` som brukes lenger nede i prøven opphever `splimp`, slik at interrupter igjen kan utføres. `splimp` og `splx` er ikke nødvendig hvis man sjekker at prøvene som er tatt er i kronologisk rekkefølge. Grunnen til at man kan gjøre det slik er at hver prøve har en egen buffer, og prøver kan dermed kun forstyrre egne målinger. Hvis man så sjekker at dataene er i rekkefølge med hensyn på tid, vet man at prøvene er korrekte. Hadde alle prøvene vært i en ringbuffer, som man gjorde i SunOs målingene, kunne man ikke gjort det slik. Grunnen til det er at alle prøvene kan forstyrre hverandre ved at de kan bli avbrutt av et interrupt midt inne i registreringen av prøven. `splimp` og `splx` ble derfor ikke tatt med, for å holde tiden prøvene bruker nede på et minimum, selv om de er vist i eksempelet over.

Etter `splimp` funksjonen er utført, blir tiden og datalengden registrert i `jProbe` og `jProbeLen`. Til slutt økes `jProbeCounter`, slik at neste gang en prøve registreres, vil den havne på riktig plass i bufferne for tidsregistrering og datalengder. Kodebiten over er lik for de tre andre prøvene, bortsett fra at datalengden knyttet til prøvene må hentes fra forskjellige steder ettersom hvor prøven befinner seg i koden.

10.2.7 Implementasjon

I dette kapittelet skal vi se på hvilke endringer som ble gjort i kildekoden til Chorus og NetBSD for å implementere JProbe i kjernen. Her gis også en

beskrivelse av hvordan prøvene i kjernen opererer, henter ut data og leverer resultater videre til applikasjonen (serverprogrammet). Serverprogammet er en egen applikasjon som utfører overføringen av data til en klient, slik at JProbe kan registrere resultater fra målingen på nettverkskoden. Applikasjonen som henter ut dataene fra JProbe, kan så lagre resultatene til disk for å senere behandle de, slik at de kan presenteres gjennom grafer og tabeller.

For å gi en oversikt over hvordan prøvene ble implementert følger en beskrivelse av endringer som ble gjort i kjernen for å tilpasse kildekoden prøvene. Under følger filene i kjernen som det ble gjort endringer på, samt en beskrivelse av hvorfor disse endringene ble gjort og hvordan de opererer sammen for å utføre innsamling av tidsmålinger når data passerer gjennom kjernens nettverkskode. Her er kun det viktigste tatt med, og for full oversikt vil jeg henvise til appendiks.

`soket.h`

`socket.h` er filen som definerer konstanter, strukturer og funksjonskall knyttet til en socket. Følgende endringer ble gjort:

```
/* Option flags per-socket. */
#define SO_JPROBE        0x0400   /* Reserved for JProbe */
#define SO_GETJPROBE     0x0800   /* Get probe results */

#define jProbeSize1      4000     /* Size of probe 1 */
#define jProbeSize2      4000     /* Size of probe 2 */
#define jProbeSize3      4000     /* Size of probe 3 */
#define jProbeSize4      4000     /* Size of probe 4 */
```

`SO_JPROBE` brukes for å kunne bestemme hvilke funksjoner som skal utføres i `sosetopt` og `sogetopt`. Hvis `sosetopt` kalles med `SO_JPROBE`, vil den skjønne at det er handlinger på prøvene i JProbe som skal utføres. `SO_JPROBE` er global og er dermed tilgjengelig i all kildekode som inkluderer kildekode for socketer. `SO_GETPROBE` fungerer likt som `SO_JPROBE`, men brukes for å hente ut verdier fra prøven etter at alle dataene for målingen er sendt.

`jProbeSize1` bestemmer den øvre grense for hvor mange resultater som kan lagres i prøve nummer en. De andre bestemmer den øvre grensen for de andre prøvene.

soketvar.h

`socketvar.h` er filen der socketstrukturen er definert. Følgende endringer ble gjort:

```
struct KnTimeVal jProbe[jProbeSize];
long   jProbeLen[jProbeSize];
long   jProbeCounter;

int    jAction;
```

I koden over ser vi en av de fire prøvene som brukes til å hente ut data. Prøvene er lagret i minne for å hindre skriving til disk under målingene. I `jProbe` lagres tidene, og i `jProbeLen` lagres datalengdene. `jProbeCounter` er en teller som holder rede på hvilken plass i prøven resultatene skal legges i.

jAction er en variabel som kan settes slik at forskjellige verdier av denne vil utføre forskjellige handlinger i `sosetopt` og `sogetopt`. På denne måten kan man hente og sette forskjellige verdier knyttet til prøvene.

uipc_socket.h

`uipc_socket.h` inneholder funksjonene `sogetopt` og `sosetopt`, som brukes for å sette og hente informasjon knyttet til en bestemt socket. `sosetopt` og `sogetopt` utføres gjennom systemkallene `setsockopt` og `getsockopt`. Følgende endringer ble gjort i disse to funksjonene:

```
sosetopt(so, level, optname, m0)
  ...
  switch (optname) {
    case SO_JPROBE:
    jAction = *mtod(m, int *);
    if (jAction == 1)
    {
      jProbeCounter1 = 0;
      jProbeCounter2 = 0;
      jProbeCounter3 = 0;
      jProbeCounter4 = 0;
    }
    if (jAction == 10) getjProbeCounter++; /* set jProbe pointer */
    if (jAction > 100) jAction -= 100; /* set jAction */
```

Endringene i `sosetopt` brukes for å sette forskjellige variabler i forbindelse med aktivering og henting av prøver. Vi ser her at hvis `optname` er satt til `SO_JPROBE` vil `sosetopt` utføre forskjellige handlinger i JProbe avhengig av `jAction`. Er `jAction` satt til 1 vil den nullstille prøvene, er `jAction` satt til 10 settes pekeren for hvilken prøve som skal hentes ut med `sogetopt` til neste prøve. Hvis `jAction` er over 100, vil `jAction` settes til `jAction` - 100 slik at man kan bruke `jAction` i `sogetopt`. For i `sogetopt` har man ikke mulighet til å sette variabler, kun hente ut data.

```
sogetopt(so, level, optname, mp)
  ...
  switch (optname) {
    case SO_JPROBE:
      if (jAction == 1) *mtod(m, int *) = jProbeSize1;
      if (jAction == 2) *mtod(m, int *) = jProbeSize2;
      if (jAction == 3) *mtod(m, int *) = jProbeSize3;
      if (jAction == 4) *mtod(m, int *) = jProbeSize4;

    case SO_GETJPROBE:
      if (jAction == 1)
      {
        if (getjProbeCounter < jProbeSize1)
          *mtod(m, long *) = jProbe1[getjProbeCounter].tmSec;
        else
          *mtod(m, long *) = 0;
      }
      if (jAction == 2)
      {
        if (getjProbeCounter < jProbeSize1)
          *mtod(m, long *) = jProbe1[getjProbeCounter].tmNSec;
        else
          *mtod(m, long *) = 0;
      }
      if (jAction == 3)
          *mtod(m, long *) = jProbeBufLen1[getjProbeCounter];
```

Endringene i `sogetopt` brukes for å hente ut prøvene etter at målingene er utført. `jAction` kan settes med `sosetopt`, og ut fra forskjellige verdier av denne utføres forskjellige handlinger. Hvis `optname` er satt til `SO_JPROBE` og `jAction` er 1, vil man få antall prøver som er registrert i prøve nummer en. Hvis `optname` er satt til `SO_GETPROBE` og `jAction` er lik 1 eller 2, vil man få

returnert tidspunktet data passerte prøven, og hvis `jAction` er tre vil man få returnert størrelsen til dataene. Ved riktig bruk av disse funksjonene kan man så hente ut prøvene fra kjernen.

10.2.8 Utførelse

For å utføre målingen utviklet jeg en egen klient og tjener som i Netperf. Tjeneren ble satt opp på maskinen målingene skulle utføres på, og klienten ble satt opp på en annen maskin knyttet til nettverket. Når klienten starter, spør den tjeneren om å få sendt over en gitt mengde data. Før tjeneren begynner å sende dataene nullstiller den JProbe-prøvene. Så sender tjeneren den gitte mengden data over til klienten. Når dataene er sendt til klienten, venter tjeneren en stund for å forsikre seg om at alle dataene er sendt. Tjeneren tar så i bruk funksjonene i `sosetopt` og `sogetopt` for å hente ut prøvene fra kjernen. Prøvene lagres så til en fil for å kunne behandles og presenteres ved hjelp av tabeller og grafer.

10.2.9 Resultater

Dataene fra prøvene presenteres for tre målinger som ble utført. Først presenteres resultater fra Chorus. Disse resultatene er fra TCP-protokollen. Senere ble det avgjort at man skulle bruke UDP for INSTANCE. Resultatene fra NetBSD inneholder derfor resultater for både TCP og UDP.

I hver av disse tre målingene ser vi på flere resultater fra prøvene. Først ser vi på hvor mye data protokollen klarer å sende ut på nettverket. Deretter ser vi på fire grafer som beskriver hvor mye data som går ut av socket- og nettverkskøen, og hvor mye data som er i socket- og nettverkskøen under tiden målingene ble foretatt. Til slutt ser vi på resultater som beskriver hvor lenge data ligger i socket- og nettverkskøen, og hvor lang tid protokollene bruker på å prosessere dataene.

Målingene på Chorus er gjort med maskinvare spesifisert i tabell 3 på et 10Mbit ethernet-nettverk. Målingene på NetBSD er gjort med maskinvare spesifisert i tabell 2 på et gigabit ethernet-nettverk. Resultatene er av den grunn forskjellige.

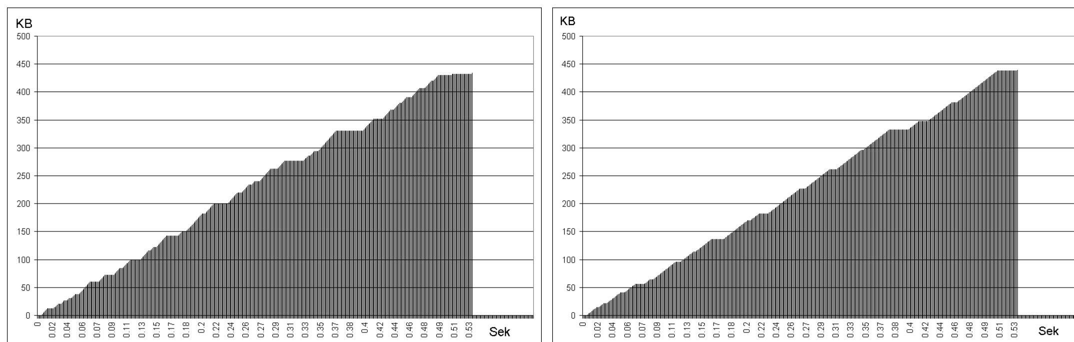
10.2.10 Chorus TCP

Tabell 7 viser gjennomstrømning av data for TCP-protokollen under Chorus. Vi ser her at gjennomstrømningen er avhengig av størrelsen på socketens sendebuffer. Øker vi socketens sendebuffer fra 8 til 64 KBytes, gir dette en forbedring av gjennomstrømning av data på hele 530 Kbit/s. Samme type resultat ble observert under målingene gjort på SunOS [6]. Vi ser fra tabellen at maksimal gjennomstrømning er 7,36 Mb/s, som vil si at 73% av båndbredden til nettverket er utnyttet. Vi må her huske at alle pakker som sendes har et pakkehode, som gjør at det ikke vil være mulig å utnytte nettverket 100% til data. Ser vi tilbake på resultatene fra loopbackmålingen kan vi fastslå at det ikke er protokollbehandlingen som er årsaken til at ikke hele båndbredden utnyttes. For å finne ut hva som er årsaken til at hele båndbredden ikke utnyttes, skal vi undersøke hvordan data flyter gjennom nettverkskoden.

Socketsize	**8KB**	**32KB**	**64KB**
Gjennomstrømning	6,83 Mb/s	7,23 Mb/s	7,36 Mb/s

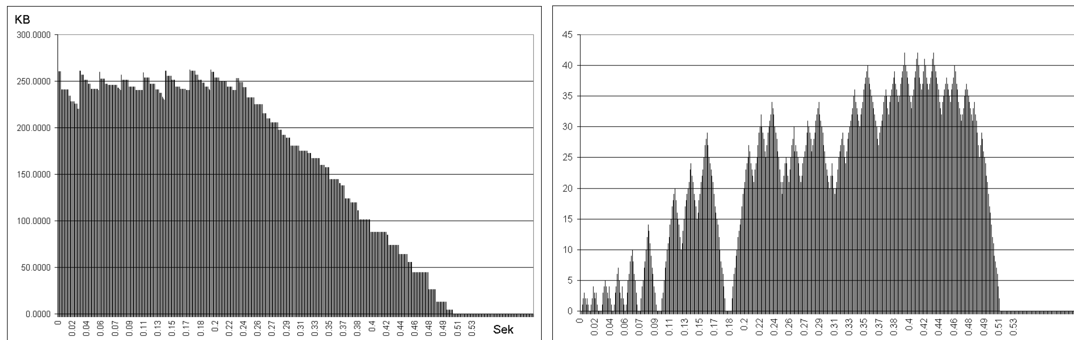
Tabell 7: Gjennomstrømning av data for TCP

I figur 50 ser vi hvordan data sendes ut av socket- og nettverkskøen. Vi ser at datastrømmen for socketkøen er jevnt stigende, med enkelte små pauser i datastrømmen. Disse pausene representerer perioder hvor protokollen ikke prosesserer data fordi den venter på bekreftelse på mottatt data, slik at TCP-protokollens vindu kan oppdateres og nye pakker sendes. Områdene mellom disse pausene, der hvor grafen er jevnt stigende, representerer perioder hvor protokollen er opptatt med å sende ut et nytt vindu med data. I grafen over



Figur 50: Data ut fra socket- og nettverkskøen

data ut fra nettverkskøen kan vi se at disse pausene er mindre på grunn av at protokollen klarer å produsere pakker fortere enn nettverket klarer å sende dem.



Figur 51: Data i socket- og nettverkskøen

Grafen til venstre i figur 51 viser hvor mye data som er i socketkøen til en hver tid under målingene. Vi ser at socketkøen synker ettersom tiden går, men ikke i et jevnt tempo. Dette skyldes TCP-protokollens vindusalgoritme. Merk at når socketkøen er tom betyr det at alle data er ute av socketkøen, men ikke at de er sendt ut på nettverket. I grafen til høyre ser vi nettverkskøen. Vi ser at nettverkskortet ikke klarer å holde tempoet til protokollen, som resulterer i at pakker køer seg opp i nettverkskøen. Vi ser at i begynnelsen av dataoverføringen er nettverkskøen liten, mens den øker ettersom tiden går. Dette skyldes TCP-protokollens algoritmer for opphopningskontroll, som sørger for at dataraten økes langsomt.

	Socketkø	**Protokollbeh.**	**Nettverkskø**
Minimum	12,793μs	24,374μs	3,627μs
Maksimum	14,368μs	60,521μs	4,080μs
Gjennomsnitt	13,372μs	39,965μs	3,759μs
Standardavvik	0,501μs	14,628μs	0,129μs
Konfidensintervall (95%)	0,284μs	8,276μs	0,073μs

Tabell 8: Tidsbruk i nettverkskoden

Tabell 8 viser hvor lang tid data bruker på å passere gjennom de forskjellige komponentene i nettverkskoden. Disse resultatene fikk jeg ved å sende

pakker med størrelse på 1 KByte gjennom nettverkskoden. På denne måten ble ikke dataene delt opp før de ble sendt, og jeg kunne derfor registrere når disse dataene passerte de forskjellige prøvene i kjernen. For å kunne skille mellom pakkene ble de sendt ut med en pause mellom hvert datasett.

Vi ser at pakkene bruker i gjennomsnitt 13,372 μs i socketkøen. Dette er tiden fra pakkene legges inn i socketkøen til de sendes videre til protokollen. Dette inkluderer ikke den tiden det tar å kopiere pakken fra applikasjonen til kjernen. Tiden for protokollbehandlingen ser vi ligger på 39,965 μs i gjennomsnitt, og det er her pakkene bruker mest tid. Vi kan også se at det er dette tidsintervallet hvor tiden varierer mest, helt fra 24,374 μs til 60,521 μs. Til slutt ser vi at pakkene bruker 3,627 μs i nettverkskøen før de sendes ut på nettverket.

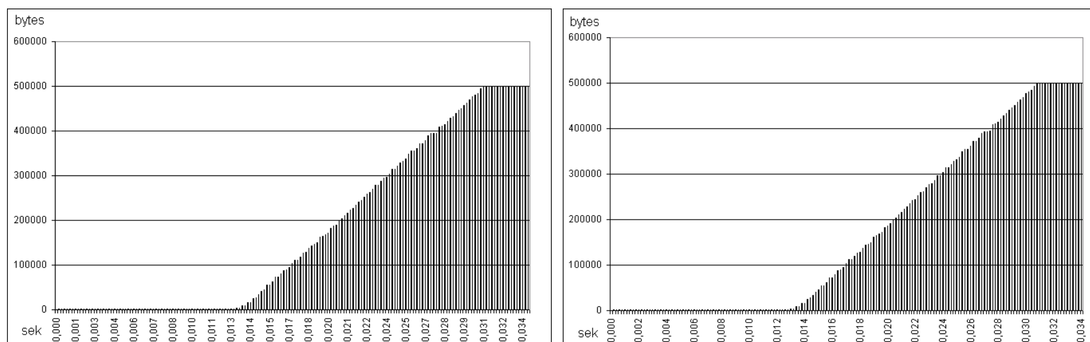
10.2.11 NetBSD TCP

Tabell 9 viser gjennomstrømning av data for NetBSD over TCP. Målingene er utført på et gigabit ethernet. Ser vi på den maksimale gjennomstrømningen av data, som ligger på 253,47 Mb/s, ser vi at dette er en dårlig utnyttelse av kapasiteten til nettverket. Resultatene viser at kun 25% av nettverkets kapasitet utnyttes. Hvis vi ser tilbake på loopbackmålingen, så indikerte den at det ville være mulig å komme opp i hastigheten til gigabit-ethernettet, men likevel ser vi at dette ikke skjer. Eneste forskjell fra loopbackmålingene er at vi nå bruker et fysisk nettverkskort istedenfor et nettverkskort implementert i programvare. Dette fører til at pakker ikke blir sendt ut like fort, og at pakker bruker tid på å bli bekreftet før nye pakker kan sendes. Dette fører altså til en betraktelig reduksjon i ytelsen.

Socketsize	**8KB**	**32KB**	**64KB**
Gjennomstrømning	121,23 Mb/s	222,53 Mb/s	253,47 Mb/s

Tabell 9: Gjennomstrømning av data for TCP

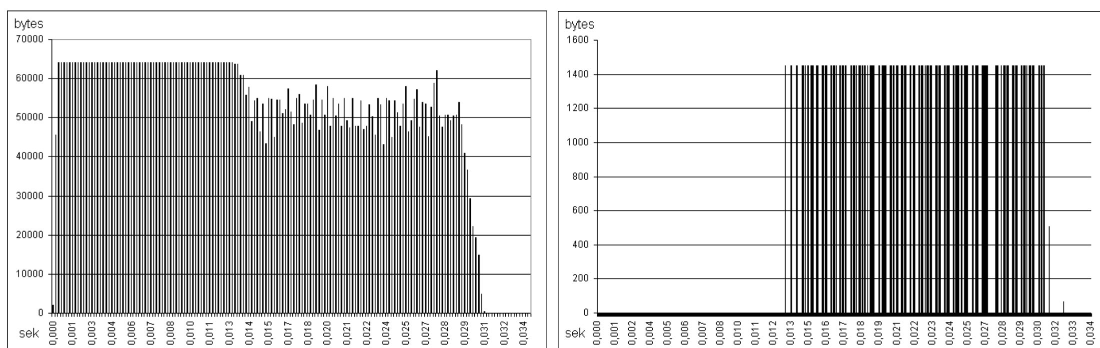
Figur 52 viser flyt av data ut fra socket- og nettverkskøen. Vi ser her at det er en jevnere flyt enn under Chorus. Dette skyldes at pakker behandles og sendes ut mye raskere, da NetBSD-målingene ble utført med hurtigere maskinvare. Siden disse målingene er for et kortere tidsintervall enn under Chorus, kan vi tydelig se at TCP-protokollen ikke sender ut data med en gang, men venter enn stund før de første dataene blir sent ut.



Figur 52: Data ut fra socket- og nettverkskøen

Figur 53 viser hvor mye data det er i socket- og nettverkskøen under målingene. Vi ser her at socketkøen blir fylt opp ved starten av målingene. Så

går det 13 millisekunder før de første pakkene sendes. Når pakkene begynner å sendes, kan vi se at socketkøen hele tiden blir fylt opp. Nettverkskoden har altså ingen problemer med å klare å fylle opp socketkøen med data. Ser vi på nettverkskøen, ser vi at det også her går 13 millisekunder før pakker blir sendt. Denne tiden viste seg å variere, og resultater fra gjentatte målinger viste at denne tiden lå mellom 10 og 25 millisekunder. Videre ser vi fra grafen at nettverkskøen ikke ligner den nettverkskøen vi så under Chorus. Her ser vi at pakker ikke køes opp i nettverkskøen, men sendes videre før nye pakker kommer inn i køen. Dette tyder på at pakker ikke blir levert fort nok til nettverkskøen fra protokollene, siden nettverkskortet hele tiden klarer å tømme nettverkskøen.



Figur 53: Data i socket- og nettverkskøen

I tabell 10 ser vi hvor lang tid pakkene bruker gjennom de forskjellige elementene i nettverkskoden til NetBSD. Vi ser fra tabellen at som for Chorus er det protokollbehandlingen som tar tid. Videre ser vi at tiden pakkene ligger i nettverkskøen er minst, dette viser at nettverket klarer å håndtere mengden av data som den får fra protokollen.

	Socketkø	**Protokollbeh.**	**Nettverkskø**
Minimum	2,555 μs	7,916 μs	0,834 μs
Maksimum	2,729μs	18,775 μs	0,938 μs
Gjennomsnitt	2,660 μs	13,235 μs	0,865 μs
Standardavvik	0,095 μs	4,763 μs	0,031 μs
Konfidensintervall (95%)	0,062 μs	3,112 μs	0,020 μs

Tabell 10: Tidsbruk i nettverkskoden

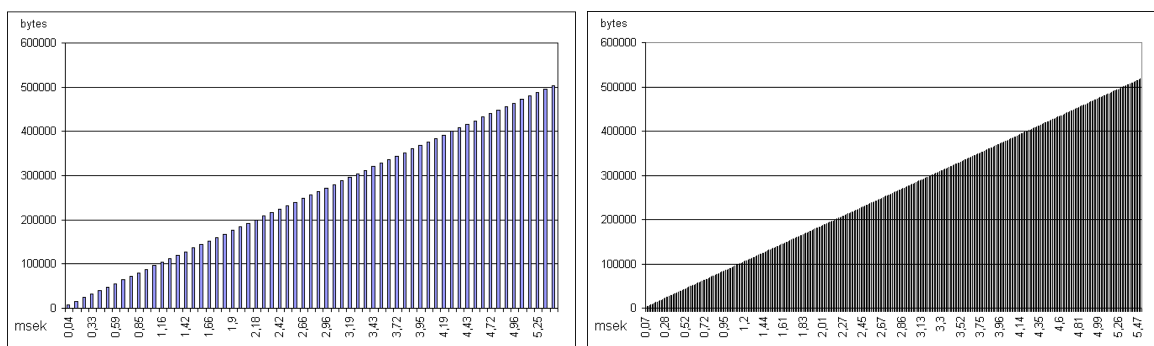
10.2.12 NetBSD UDP

Tabell 11 viser gjennomstrømning av data, i NetBSD, for UDP-protokollen. Vi ser her at den maksimale gjennomstrømningen er på 812,23 Mb/s. Dette er en god forbedring i forhold til TCP-protokollen, som hadde en gjennomstrømning på 253,47 Mb/s. Dette er en økning på hele 315%, som viser at TCP-protokollen setter store begrensninger på ytelsen. Resultatet på 812,23 Mb/s viser at UDP-protkollen utnytter nettverkshastigheten godt, men ikke fullt ut. Det er heller ikke mulig å utnytte nettverkshastigheten fullt ut, da pakker som sendes også inneholder pakkehoder i tillegg til dataene. En annen årsak til at hastigheten ikke kan benyttes fullt ut er mulige kollisjoner av pakker på ethernettet. Likevel viser det seg at det er mulig å oppnå høyere hastighet enn 812,23 Mb/s [18], ved å øke størrelsen på den maksimale pakkestørrelsen i kommunikasjonskanalen (MTU).

Socketsize	**8KB**	**32KB**	**64KB**
Gjennomstrømning	715,63 Mb/s	803,42 Mb/s	812,23 Mb/s

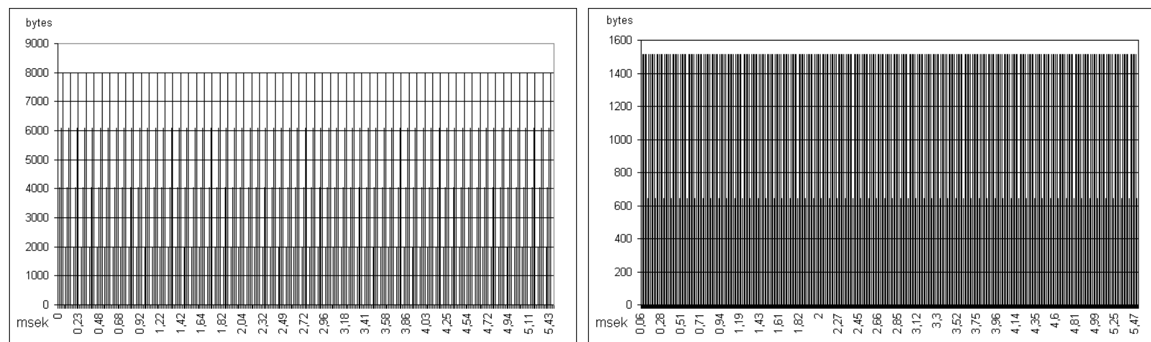
Tabell 11: Gjennomstrømning av data for UDP

I figur 54 ser vi dataflyten ut av socket- og nettverkskøen. Vi ser her at flyten av data er jevnere og raskere enn for TCP-protokollen. Vi ser også at UDP-protokollen begynner å sende data med en gang, i motsetning til TCP-protokollen. Fra figuren over data ut fra socketkøen, ser vi at data sendes ut i større størrelser enn fra nettverkskøen, dette vil vi se bedre når vi nå skal se på hvor mye data som ligger i socketkøen under målingene.



Figur 54: Data ut fra socket- og nettverkskøen

Figur 55 viser hvor mye data som ligger i socket- og nettverkskøen til en hver tid under målingene. I socketkøen ser vi at data samles opp i porsjoner på 8 KBytes før de sendes til protokollene for videre behandling. Socketkøen fylles ikke opp med mer enn 8 KBytes før data sendes videre, i motsetning til TCP-protokollen som fylte socketkøen opp til 64 KBytes. Fra nettverkskøen kan vi se at dataene fra socketkøen er delt opp for å få plass i et datagram, med maks størrelse spesifisert av MTU. Vi ser her at protokollen deler opp dataene fra socketkøen i fem mbuffere på 1500 bytes (inkludert pakkehode), og en mbuffer på 640 bytes. Disse kommer så inn i nettverkskøen en av gangen, og sendes ut på nettverket før den neste rekker å komme inn i nettverkskøen. I de målingene som ble foretatt er det brukt standardverdier for UDP-protokollen, som vil si at MTU er satt til 1500. Størrelsen på MTU er derfor en mulig flaskehals for denne forbindelsen.



Figur 55: Data i nettverk og socketkøen

I tabell 12 ser vi hvor lang tid en pakke bruker gjennom de forskjellige elementene i nettverkskoden for overføring av data med UDP-protokollen. Vi ser at tiden pakken bruker i protokollen er redusert betraktelig i forhold til resultatene fra TCP-protokollen. Her er gjennomsnittlig tid en pakke bruker på protokollprosessering nede i 5,574 μs. Protokollbehandlingen under UDP er mindre komplisert enn i TCP, og vi ser derfor at resultatene for protokollhåndteringen er jevnere for UDP enn TCP. Sammenligner vi tiden pakken er i socketkøen, kan vi se at denne er relativ lik i forhold til TCP. Nettverkskøen for UDP-protokollen mottar pakker fra protokollen raskere enn i TCP-protokollen. I resultatene for nettverkskøen ser vi derfor at pakkene ligger i køen lenger i forhold til TCP-protokollen.

	Socketkø	**Protokollbeh.**	**Nettverkskø**
Minimum	2,816μs	5,406μs	4,260μs
Maksimum	3,063μs	5,649μs	4,406μs
Gjennomsnitt	2,922μs	5,574μs	4,326μs
Standardavvik	0,074μs	0,075μs	0,053μs
Konfidensintervall (95%)	0,042μs	0,042μs	0,030μs

Tabell 12: Tidsbruk i nettverkskoden

11 Konklusjon

I denne oppgaven har jeg studert dataflyten fra disk til nettverk for to operativsystemer, Chorus og NetBSD, med forskjellig maskinvare. I dette kapittelet trekker jeg frem resultater fra målingene som er gjort, og trekker en konklusjon ut fra disse målingene. Først vil jeg gi et sammendrag av resultatene, og så en beskrivelse av antagelser og begrensninger ved målingene. Deretter vil jeg beskrive forslag til videre arbeid, og til slutt gi en endelig konklusjon.

11.1 Sammendrag

Vi skal her se på resultatene som ble funnet under målingene på Chorus og NetBSD. Vi ser først på Chorus. Maskinen Chorus kjørte på er bygd opp med eldre maskinvarekomponenter enn maskinvarekomponentene som maskinen NetBSD kjørte på er bygd opp av. Resultatene er av den grunn langt dårligere i forhold til resultatene fra maskinen med NetBSD.

Fra målingene på Chorus kan vi se at det er diskytelsen som er lavest, med en overføringshastighet i underkant av 4 MB/s. Med denne hastigheten er det ikke engang mulig å utnytte et 100Mbit (12,5 MB/s) nettverk fullt ut. Videre ser vi at minnehastigheten også er begrenset. Med en hastighet på 188 MB/s (1504 Mbit/s) er det kun tid til en kopieringsoperasjon fra disk til nettverk under et gigabit-nettverk. I tradisjonelle systemer bruker man flere kopieringsoperasjoner, og minnehastigheten vil derfor være en flaskehals i et slikt system. Videre ser vi fra nettverksmålingene at det er i nettverkskøen pakkene hoper seg opp. Dette er på grunn av at behandling av pakker gjennom nettverkskoden går hurtigere enn nettverket, på 10 Mbit/s, klarer å sende ut pakker. Vi ser også at mye av tiden pakkene bruker gjennom nettverkskoden går til protokollbehandling.

Målinger fra NetBSD viser en betraktelig forbedring i forhold til målingene på Chorus. Fra resultatene av diskmålingene under NetBSD kan vi se at diskytelsen er i overkant av 30 MB/s. Dette er en forbedring på 800%, noe som viser at disken som ble brukt under Chorus ikke var særlig rask. En hastighet på 30 MB/s tilsvarer 240 Mbit/s, som viser at disken helt klart vil være en flaskehals under et system som bruker et gigabit-nettverk. Videre ser vi at også minnehastigheten er betraktelig forbedret i forhold til resultatene fra Chorus. Minnehastigheten var her i underkant av 600 MB/s, en forbedring på 316% i forhold til Chorus. En hastighet på 600 MB/s tilsvarer 4800 Mbit/s. Dette gir mulighet for inntil fire kopieringsoperasjoner for flytting av data fra

disk til nettverk hvis et gigabit-nettverk skal utnyttes fullt ut. Dette vil si at minnehastigheten ikke nødvendigvis er en flaskehals i et slikt system. Dette kommer selvsagt an på hvor mye minneressurser behandling av andre elementer enn kopiering av data gjennom nettverkskoden bruker. Resultatet fra nettverksmålingene på loopback-grensesnittet bekrefter at minnehastigheten ikke nødvendigvis behøver å være en flaskehals. Vi ser her at det er mulig å oppnå hastigheter på 1300 Mbit/s, som er godt over hastigheten til et gigabit-nettverk. Likevel viser det seg at resultater fra målinger med TCP/IP-protokollen, på et gigabit-nettverk, ikke klarer å utnytte hele kapasiteten til nettverket. Resultatene viser at overføringshastigheten ligger på 250 Mbit/s, som er 25% av kapasiteten til nettverket. Ved å unngå begrensningene protokollbehandlingen til TCP/IP setter på forbindelsen, og isteden bruke UDP, ser vi at gjennomstrømningen øker til over 800 Mbit/s.

Ser vi mer detaljert på resultatene jeg fikk fra målingene på nettverket, ser vi at socketens størrelse på sendebufferet virker inn på den maksimale hastigheten for gjennomstrømning av data. Ved å bruke større sendebuffer øker gjennomstrømningen, men ikke når bufferet blir større enn 32 KBytes. Sammenligner vi dette med resultatene fra målingene som ble utført på Sparc10 (avsnitt 2.4) og SunOS (avsnitt 2.5), kan vi se de samme resultatene her. Videre ser vi at gjennomstrømningen også øker når det blir brukt større buffere for dataene som sendes, siden dette resulterer i færre systemkall og mindre oppdeling av dataene i mbuffere. I resultater fra målingene på Sparc10 ser vi samme resultat. Det viser seg at det også er mulig å få en ytterligere økning i gjennomstrømningen ved å øke størrelsen på den største dataenheten som kan sendes over nettverket (MTU). I målinger gjort på OpenBSD [27], kan vi også se dette resultatet.

11.2 Antagelser og begrensninger

Resultater fra målingene i denne oppgaven er for det meste gjort med verktøy som jeg selv har utviklet. For å forsikre meg om at dataene som er målt med disse verktøyene er korrekte, er det viktig å kontrollere at resultatene er riktige. Jeg vil her gi en beskrivelse, for hver av målingene, av hva som er gjort for å sørge for at resultatene er så korrekte som mulig.

For målingene som ble gjort på minne er resultatene sammenlignet med andre målinger som er gjort på lignende systemer med verktøyet STREAM [40]. STREAM er et verktøy som brukes av mange, og er derfor et godt utprøvd verktøy. Målingene som ble gjort på disk ble også utført med egenutviklet programvare. For å forsikre meg om at koden skal være fri for feil, valgte jeg å

bruke allerede eksisterende kodebiter fra Iozone. Iozone er også et et verktøy som er godt utprøvd og testet av mange. De siste målingene jeg gjorde med egenutviklet kode var nettverksmålingene. For å forsikre at disse resultatene var korrekte brukte jeg Netperf, som beskrevet tidligere, for å kontrollere resultatene.

11.3 Mulige løsninger

Vi har sett at disken er den største flaskehalsen i de to systemene vi har sett på. Dette var antatt, og ikke et uventet resultat. For å kunne øke ytelsen til disken er det flere mulige løsninger. En mulig løsning er å sørge for at mesteparten av dataene som sendes ligger i cache. Dette er ikke alltid mulig, særlig i en videoserver som består av et fåtall store filer. Siden filene er store og størrelsen på cachen er begrenset, vil det være begrenset med data som ligger tilgjengelig i cachen. En annen løsning kan da være å ta i bruk et RAID-system. Grunnen til at et RAID-system kan øke gjennomstrømningen av data, er at det ikke er bussen fra disken som begrenser hastigheten til disken, men selve diskens lesehastighet. Ved å lese fra flere disker samtidig, kan man øke gjennomstrømningen til serveren. I en SCSI-konfigurasjon kan man ha opptil 15 disker i en SCSI-kjede, og i en IDE-konfigurasjon kan man ha 4 disker. Teoretiske overføringshastigheter for diskene i et slikt system er vist i tabell 13. Verdiene i tabellen er kun en antagelse av at ytelsen dobles når en ekstra disk settes inn i systemet. For å finne ut hva de virkelige hastighetene er, er det nødvendig å gjøre en analyse av slike systemer.

Antall disker	1	2	3	4	5	10	15
Chors IDE	4	8	12	16	-	-	-
Chorus SCSI	4	8	12	16	20	40	60
NetBSD IDE	32	64	96	128	-	-	-
NetBSD SCSI	32	64	96	128	160	160	160

Tabell 13: Diskytelse med disker i kjede.

Hvis vi ser bort fra disken som en flaskehals, er det mulig å øke gjennomstrømningen ved å justere størrelsen på socketens og applikasjonens sendebuffer. Andre mulige løsninger for å øke gjennomstrømning av data, er å endre standardverdier som angir maksimal pakkestørrelse på nettet (MTU).

Videre har det vist seg at minnehastigheten ligger på grensen til å klare å utnytte hastigheten til et gigabit-nettverk. Ved å installere en annen type minne, som er raskere, kan man forsikre seg om at minnehastigheten ikke skaper en flaskehals. Dette er selvsagt ikke nødvendig før andre større flaskehalser er fjernet, som f.eks. flaskehalsen disken utgjør.

11.4 Videre arbeid

Målingene som er utført på nettverket er ikke testet grundig med forskjellige verdier av MTU. For å se hvilken rolle forskjellige størrelser av MTU spiller inn på flyten av data, kan det være aktuelt å gjøre nærmere undersøkelser på dette området. Videre kan man se på hvordan et system, med en RAID-konfigurasjon som overgår nettverkshastigheten, vil fungere. På denne måten kan det undersøkes nærmere hvordan minneytelsen setter begrensninger på overføring av data fra disk til nettverk i nettverkskoden, da vi kan gjøre målinger som inkluderer minnekopieringer fra disk til applikasjon uten at disken er en flaskehals. I denne oppgaven har vi sett på hver komponent i operativsystemet isolert. Hvis man har en diskytelse som overgår nettverket, kan man også se på systemet samlet, og undersøke om det fortsatt er mulig å utnytte hele nettverkskapasiteten.

For å vite hvilken flaskehals man skal satse på å utbedre, må man følge med på hvilke typer maskinvare som utvikler seg raskest. Det er altså ikke sikkert at det som i dag er en flaskehals vil være en flaskehals i fremtiden.

11.5 Samlet konklusjon

Resultater fra målingene som er gjort har vist at det som antatt er disken som er den største flaskehalsen. Videre har vi sett at minneytelsen kan begrense hastigheten, men at stadig raskere minne gjør det mulig å overgå hastigheten til et gigabit-nettverk. En reduksjon i antall kopieringsoperasjoner, som beskrevet i INSTANCE, kan likevel vise seg å være lønnsomt. Dette er på grunn av at minneytelsen ligger på grensen til å klare å utnytte nettverket, selv uten at minneoperasjoner i forbindelse med disklesing er iberegnet. Vi har også sett at valg av protokoll spiller en stor rolle for hastigheten. Under NetBSD var det mulig å oppnå over tre ganger så stor gjennomstrømning av data med UDP, enn det var med TCP. Videre har vi sett at feil valg av størrelsen på socketens og applikasjonens sendebuffer setter begrensninger på gjennomstrømning av data. Riktige valg kan øke gjennomstrømningen betraktelig og fjerne en mulig flaskehals. Til slutt har vi også sett at verdien

til MTU begrenser den maksimale gjennomstrømningen av data, og ved å øke MTU er det mulig å oppnå større hastigheter.

A Endringer i socket.h

```
/*
 * Option flags per-socket.
 */
#define SO_DEBUG  0x0001          /* turn on debugging info recording */
#define SO_ACCEPTCONN 0x0002 /* socket has had listen() */
#define SO_REUSEADDR 0x0004     /* allow local address reuse */
#define SO_KEEPALIVE 0x0008     /* keep connections alive */
#define SO_DONTROUTE 0x0010    /* just use interface addresses */
#define SO_BROADCAST 0x0020    /* permit sending of broadcast msgs */
#define SO_USELOOPBACK 0x0040 /* bypass hardware when possible */                10
#define SO_LINGER 0x0080          /* linger on close if data present */
#define SO_OOBINLINE 0x0100      /* leave received OOB data in line */
#define SO_REUSEPORT 0x0200     /* allow local address & port reuse */
#define SO_JPROBE 0x0400          /* jana probes */

#define jProbeSize1 2000           /* Size of probe #1 */
#define jProbeSize2 2000           /* Size of probe #2 */
#define jProbeSize3 2000           /* Size of probe #3 */
#define jProbeSize4 2000           /* Size of probe #4 */
                                                                                                    20
/*
 * Additional options, not kept in so_options.
 */
#define SO_SNDBUF 0x1001         /* send buffer size */
#define SO_RCVBUF 0x1002         /* receive buffer size */
#define SO_SNDLOWAT 0x1003     /* send low-water mark */
#define SO_RCVLOWAT 0x1004     /* receive low-water mark */
#define SO_SNDTIMEO 0x1005     /* send timeout */
#define SO_RCVTIMEO 0x1006     /* receive timeout */
#define SO_ERROR  0x1007          /* get error status and clear */                    30
#define SO_TYPE     0x1008          /* get socket type */
#define SO_GETJPROBE 0x1009    /* Get a probe */
```

B Endringer i socketvar.h

```
// jProbe
int  jAction;                          /* Action to perform in sosetopt and sogetopt */
long getjProbeCounter;                 /* Counter for receiving probe #getjProbeCouner */

/* This is probe 1 */
struct KnTimeVal jProbe1[jProbeSize1]; /* Time result for probe */
long   jProbeBufLen1[jProbeSize1];     /* Size result for probe */
long   jProbeCounter1;                 /* Counter to tell which probe to insert results */

/* This is probe 2 */                                                                     10
struct KnTimeVal jProbe2[jProbeSize2]; /* Time result for probe */
long   jProbeBufLen2[jProbeSize2];     /* Size result for probe */
long   jProbeCounter2;                 /* Counter to tell which probe to insert results */

/* This is probe 3 */
struct KnTimeVal jProbe3[jProbeSize3]; /* Time result for probe */
long   jProbeBufLen3[jProbeSize3];     /* Size result for probe */
long   jProbeCounter3;                 /* Counter to tell which probe to insert results */

/* This is probe 4 */                                                                     20
struct KnTimeVal jProbe4[jProbeSize4]; /* Time result for probe */
long   jProbeBufLen4[jProbeSize4];     /* Size result for probe */
long   jProbeCounter4;                 /* Counter to tell which probe to insert results */
```

C Endringer i sosetopt()

```
sosetopt(so, level, optname, m0)
        register struct socket *so;
        int level, optname;
        struct mbuf *m0;
{
        int error = 0;
        register struct mbuf *m = m0;


        if (level != SOL_SOCKET) {                                          10
                if (so->so_proto && so->so_proto->pr_ctloutput)
                        return ((*so->so_proto->pr_ctloutput)
                                  (PRCO_SETOPT, so, level, optname, &m0));
                error = ENOPROTOOPT;
        } else {
                switch (optname) {

                case SO_JPROBE:
                  if (m == NULL || m->m_len < sizeof (int))
                    {                                                       20
                      error = EINVAL;
                      goto bad;
                    }

                  jAction = *mtod(m, int *);

                  if (jAction == 1)
                    {
                      jProbeCounter1 = 0;
                      jProbeCounter2 = 0;                                   30
                      jProbeCounter3 = 0;
                      jProbeCounter4 = 0;
                    }

                  if (jAction == 2) so->so_options |= optname;
                  if (jAction == 3) so->so_options &= ~optname;

                  if (jAction == 10) getjProbeCounter = 1;
                  if (jAction == 11) getjProbeCounter--;
                  if (jAction == 12) getjProbeCounter++;                   40

                  // Only for testing
                  if (jAction == 21) sysTime(&jProbe1[1]);
                  if (jAction == 22) sysTime(&jProbe1[2]);
                  if (jAction == 23) sysTime(&jProbe1[3]);
                  if (jAction == 24) sysTime(&jProbe1[4]);
```

```
        if (jAction == 31) jProbeCounter1 = 1;
        if (jAction == 32) jProbeCounter2 = 1;
        if (jAction == 33) jProbeCounter3 = 1;                          50
        if (jAction == 34) jProbeCounter4 = 1;

        // Set jAction to jAction - 100
        // Use this for setting jAction for sogetopt
        if (jAction > 100) jAction -= 100;

        break;
    case SO_LINGER:
            if (m == NULL || m->m_len != sizeof (struct linger)) {
                    error = EINVAL;                                     60
                    goto bad;
            }
            so->so_linger = mtod(m, struct linger *)->l_linger;
            /* fall thru... */

    case SO_DEBUG:
    case SO_KEEPALIVE:
    case SO_DONTROUTE:
    case SO_USELOOPBACK:
    case SO_BROADCAST:                                                  70
    case SO_REUSEADDR:
    case SO_REUSEPORT:
    case SO_OOBINLINE:
            if (m == NULL || m->m_len < sizeof (int)) {
                    error = EINVAL;
                    goto bad;
            }
            if (*mtod(m, int *))
                    so->so_options |= optname;
            else                                                        80
                    so->so_options &= ~optname;
            break;

    case SO_SNDBUF:
    case SO_RCVBUF:
    case SO_SNDLOWAT:
    case SO_RCVLOWAT:
            if (m == NULL || m->m_len < sizeof (int)) {
                    error = EINVAL;
                    goto bad;                                           90
            }
            switch (optname) {

            case SO_SNDBUF:
            case SO_RCVBUF:
```

```
                if (sbreserve(optname == SO_SNDBUF ?
                    &so->so_snd : &so->so_rcv,
                    (u_long) *mtod(m, int *)) == 0) {
                        error = ENOBUFS;
                        goto bad;                                   100
                }
                break;

        case SO_SNDLOWAT:
                so->so_snd.sb_lowat = *mtod(m, int *);
                break;
        case SO_RCVLOWAT:
                so->so_rcv.sb_lowat = *mtod(m, int *);
                break;
        }                                                           110
        break;

case SO_SNDTIMEO:
case SO_RCVTIMEO:
    {
        struct timeval *tv;
        short val;

        if (m == NULL || m->m_len < sizeof (*tv)) {
                error = EINVAL;                                     120
                goto bad;
        }
        tv = mtod(m, struct timeval *);
        if (tv->tv_sec > SHRT_MAX / hz - hz) {
                error = EDOM;
                goto bad;
        }
        val = tv->tv_sec * hz + tv->tv_usec / tick;

        switch (optname) {                                          130

        case SO_SNDTIMEO:
                so->so_snd.sb_timeo = val;
                break;
        case SO_RCVTIMEO:
                so->so_rcv.sb_timeo = val;
                break;
        }
        break;
    }                                                               140

default:
        error = ENOPROTOOPT;
        break;
```

```
		}
		if (error == 0 && so->so_proto && so->so_proto->pr_ctloutput) {
			(void) ((*so->so_proto->pr_ctloutput)
				(PRCO_SETOPT, so, level, optname, &m0));
			m = NULL;	/* freed by protocol */
		}                                                          150
	}
bad:
	if (m)
		(void) m_free(m);
	return (error);
}
```

D Endringer i sogetopt()

```
sogetopt(so, level, optname, mp)
        register struct socket *so;
        int level, optname;
        struct mbuf **mp;
{
        register struct mbuf *m;
        int jTemp;

        if (level != SOL_SOCKET) {
                if (so->so_proto && so->so_proto->pr_ctloutput) {
                        return ((*so->so_proto->pr_ctloutput)
                                  (PRCO_GETOPT, so, level, optname, mp));
                } else
                        return (ENOPROTOOPT);
        } else {
                m = m_get(M_WAIT, MT_SOOPTS);
                m->m_len = sizeof (int);

                switch (optname) {

                  // JPROBE BY Jan Erik Askjellrud
                case SO_JPROBE:
                  // For kompatibelt med gammelt...
                  if (jAction == 1) *mtod(m, int *) = jProbeSize1;
                  if (jAction == 2) *mtod(m, int *) = jProbeSize2;
                  if (jAction == 3) *mtod(m, int *) = (int) jProbe1[1].tmSec;
                  if (jAction == 4) *mtod(m, int *) = (int) jProbe1[1].tmNSec;
                  if (jAction == 5) *mtod(m, int *) = (int) getjProbeCounter;
                  if (jAction == 6) *mtod(m, int *) = (int) jProbeCounter1;
                  if (jAction == 7) *mtod(m, int *) = (int) jProbeCounter2;
                  if (jAction == 8) *mtod(m, int *) = (int) jProbeCounter3;
                  if (jAction == 9) *mtod(m, int *) = (int) jProbeCounter4;
                  // Ny!
                  if (jAction == 11) *mtod(m, int *) = jProbeSize1;
                  if (jAction == 12) *mtod(m, int *) = jProbeSize2;
                  if (jAction == 13) *mtod(m, int *) = jProbeSize3;
                  if (jAction == 14) *mtod(m, int *) = jProbeSize4;

                  if (jAction > 20 || jAction < 1) *mtod(m, int *) = jAction;
                  break;

                case SO_GETJPROBE:

                  /*********************
                    PROBE SEC
                  *********************/
```

```
if (jAction == 1)
  {
    if (getjProbeCounter < jProbeSize1)
      *mtod(m, long *) = jProbe1[getjProbeCounter].tmSec;          50
    else
      *mtod(m, long *) = 0;
  }
if (jAction == 2)
  {
    if (getjProbeCounter < jProbeSize2)
      *mtod(m, long *) = jProbe2[getjProbeCounter].tmSec;
    else
      *mtod(m, long *) = 0;
  }                                                                60
if (jAction == 3)
  {
    if (getjProbeCounter < jProbeSize3)
      *mtod(m, long *) = jProbe3[getjProbeCounter].tmSec;
    else
      *mtod(m, long *) = 0;
  }
if (jAction == 4)
  {
    if (getjProbeCounter < jProbeSize4)                            70
      *mtod(m, long *) = jProbe4[getjProbeCounter].tmSec;
    else
      *mtod(m, long *) = 0;
  }

/********************
  PROBE NSEC
********************/
if (jAction == 11)
  {                                                                80
    if (getjProbeCounter < jProbeSize1)
      *mtod(m, long *) = jProbe1[getjProbeCounter].tmNSec;
    else
      *mtod(m, long *) = 0;
  }
if (jAction == 12)
  {
    if (getjProbeCounter < jProbeSize2)
      *mtod(m, long *) = jProbe2[getjProbeCounter].tmNSec;
    else                                                           90
      *mtod(m, long *) = 0;
  }
if (jAction == 13)
  {
    if (getjProbeCounter < jProbeSize3)
```

```
        *mtod(m, long *) = jProbe3[getjProbeCounter].tmNSec;
      else
        *mtod(m, long *) = 0;
    }
  if (jAction == 14)                                              100
    {
      if (getjProbeCounter < jProbeSize4)
        *mtod(m, long *) = jProbe4[getjProbeCounter].tmNSec;
      else
        *mtod(m, long *) = 0;
    }

  /********************
    PROBE LEN
  ********************/                                           110

  if (jAction == 21) *mtod(m, long *) = jProbeBufLen1[getjProbeCounter];
  if (jAction == 22) *mtod(m, long *) = jProbeBufLen2[getjProbeCounter];
  if (jAction == 23) *mtod(m, long *) = jProbeBufLen3[getjProbeCounter];
  if (jAction == 24) *mtod(m, long *) = jProbeBufLen4[getjProbeCounter];

  break;

case SO_LINGER:
        m->m_len = sizeof (struct linger);                        120
        mtod(m, struct linger *)->l_onoff =
                so->so_options & SO_LINGER;
        mtod(m, struct linger *)->l_linger = so->so_linger;
        break;

case SO_USELOOPBACK:
case SO_DONTROUTE:
case SO_DEBUG:
case SO_KEEPALIVE:
case SO_REUSEADDR:                                                130
case SO_REUSEPORT:
case SO_BROADCAST:
case SO_OOBINLINE:
        *mtod(m, int *) = so->so_options & optname;
        break;

case SO_TYPE:
        *mtod(m, int *) = so->so_type;
        break;
                                                                  140
case SO_ERROR:
        *mtod(m, int *) = so->so_error;
        so->so_error = 0;
        break;
```

```
                case SO_SNDBUF:
                        *mtod(m, int *) = so->so_snd.sb_hiwat;
                        break;

                case SO_RCVBUF:                                                    150
                        *mtod(m, int *) = so->so_rcv.sb_hiwat;
                        break;

                case SO_SNDLOWAT:
                        *mtod(m, int *) = so->so_snd.sb_lowat;
                        break;

                case SO_RCVLOWAT:
                        *mtod(m, int *) = so->so_rcv.sb_lowat;
                        break;                                                     160

                case SO_SNDTIMEO:
                case SO_RCVTIMEO:
                    {
                        int val = (optname == SO_SNDTIMEO ?
                             so->so_snd.sb_timeo : so->so_rcv.sb_timeo);

                        m->m_len = sizeof(struct timeval);
                        mtod(m, struct timeval *)->tv_sec = val / hz;
                        mtod(m, struct timeval *)->tv_usec =                       170
                            (val % hz) / tick;
                        break;
                    }

                default:
                        (void)m_free(m);
                        return (ENOPROTOOPT);
                }
                *mp = m;
                return (0);                                                        180
        }
}
```

E Installasjon av Chorus

Dette dokumentet beskriver hvordan jeg installerte Chorus 3.2 på maskinen thebe.unik.no. Først har jeg listet opp tekniske data, så kommer en beskrivelse av hvordan man installerer Chorus og til slutt en beskrivelse av hvordan en IDE-disk settes opp ned UFS.

E.1 Tekniske data

Under har jeg listet opp tekniske data for maskinen Chorus skal ligge på (target), og maskinen som brukes under utvikling av Chorus applikasjoner (host). I mitt tilfelle er thebe target og odin host.

TARGET

– Maskin
  Navn : thebe.unik.no
  IP adresse : 193.156.96.103
  Prosessor : Pentium II 350MHz
  Minne : 192 MB

– Ethernet kort
  Navn : SMC ethercard elite 16
  Type : NE2000 kompatibelt, COMBO
  Hastighet : 10 Mb/s
  Bus : ISA

– Disk
  Produsent : Seagate
  Navn : Medalist 6531 AT (ST-36531A)
  Cylindere : 13.446
  Hoder : 15 (6 fysiske hoder)
  Sektorer : 63
  Kapasitet : 6505 MB
  Buffer : 128K

HOST

- Maskin
  Navn : odin.unik.no
  IP adresse : 193.156.96.7
  Platform : solaris
- Environment
  USER : jana
  CLX : /share/chorus/ChorusOS-3.2.1
  MERGEDIR : $CLX/merge/$USER

E.2 Installasjon

For å kunne starte opp maskinen med Chorus må det lages et bootimage med Chorus. Bootimaget finnes i katalogen $MERGEDIR. Hvis du ikke har et bootimage, kan du lage et som beskrevet under. Pass på at du bruker gnu sin gcc når du skal kompilere Chorus (se etter MAKE= i filen $MERGEDIR/config).

```
%cd $CLX
%mkmerge -f profiles/server -t $MERGEDIR
%cd $MERGEDIR
%make check_install
%make
```

Etter en god stund er kompileringen ferdig og du skal ha følgende melding på skjermen:

```
=== autonomous Chorus Kernel build completed at ... ===
```

Nå er kjernen til Chorus lagd og for å boote den må vi lage en bootdisk. Bootdisken må konfigureres med riktig parametrer for nettverkskortet, target maskin og host maskin. Med tekniske data som beskrevet i tabellene over, lages bootdisken med følgende kommandoer:

```
%cd $CLX
%mkenv NBT/netboot -d ND_NE_0_ITLEVEL=11
%mkenv NBT/netboot -d ND_NE_0_BASEIO=0x220
%mkenv NBT/netboot -d LOCAL_INADDR=193.156.96.103
%mkenv NBT/netboot -d CONFIGSERVER=193.156.96.7
%mkenv NBT/netboot -d CONFIGFILE=C19C6067
%mkdf  NBT/netboot thebe-boot.dd
```

De to øverste linjene forteller at thebe har et NE2000 kompatibelt nettverkskort som kan brukes med IRQ 11 og BASEIO 0x220. Neste linje er IP adressen til thebe. De to neste linjene forteller at imaget til bootdisken skal bruke janus.unik.no (193.156.96.7) som bootserver med parametrene i filen /tftpboot/C19C6067 (på janus). Navnet på konfigurasjonsfilen (C19C6067) er IP-adressen til thebe i heksadesimalt. Konfigurasjonsfilen ser slik ut :

```
BOOTSERVER=193.156.96.46
BOOTFILE=chorus
INPUT_TIMEOUT=10
AUTOBOOT=yes
TFTP_TIMEOUT=10
```

Linjen med BOOTFILE forteller at bootimaget til Chorus finnes i filen /tftpboot/chorus og BOOTSERVER forteller at den ligger på janus. Så for at den skal boote med riktig Chorus image, må imaget som man ønsker at Chorus skal bootes med kopieres til /tftpbot/chorus på janus.

For å skrive bootdisken til en diskett kan du bruke kommandoen

```
%dd of=/dev/rfd0 if=thebe-boot.dd
```

Hvis du ikke har tilgang til diskettstasjon fra UNIX, kan du skrive bootimaget fra NT/Windows 98 med programmet rawrite2.exe.

Nå skulle alt være klart, og når du nå setter bootdisken inn i thebe skal den boote opp Chorus. Hvis det ikke virker, ta en titt på "Installation guide" fra Chorus.

E.3 IDE disk med UFS

Her følger en kort beskrivelse av hvordan man kan koble opp en disk med UFS under Chorus.

Kjør følgende kommando for å få support for IDE disk i Chorus:

```
%mkconfig -a IDE_DISK
```

Support for UFS i Chorus:

```
%mkconfig -a UFS
```

Rekompiler (make) for å aktivere endringene.

1. Mount $MERGEDIR med NFS fra maskinen embla

```
%/usr/ucb/rsh thebe mount 193.156.96.168:$CLX/merge/$USER
```

2. mkmod
Gå til $MERGEDIR/dev katalogen :

```
embla:/export/chorus/ChorusOS-3.2.1/merge/$USER/dev
```

```
%rlogin embla
%cd /export/chorus/ChorusOS-3.2.1/merge/$USER/dev
%mkdir dev
%cd dev
%mknod hd0a b 0 0
%mknod hd0b b 0 1
%mknod hd0c b 0 2
%mknod rhd0a c 3 0
%mknod rhd0b c 3 1
%mknod rhd0c c 3 2
```

da får du følgende spesialfiler:

```
brw-r--r--   1 chorus   other      0,   0 Dec  6 14:09 hd0a
brw-r--r--   1 chorus   other      0,   1 Dec  6 14:09 hd0b
brw-r--r--   1 chorus   other      0,   2 Dec  6 14:09 hd0c
crw-r--r--   1 chorus   other      3,   0 Dec  6 14:11 rhd0a
crw-r--r--   1 chorus   other      3,   1 Dec  6 14:11 rhd0b
crw-r--r--   1 chorus   other      3,   2 Dec  6 14:11 rhd0c
```

3. Disklabel, finn disklabel på disk.

```
%/usr/ucb/rsh thebe arun /bin/disklabel -r hd0
```

Skrive inn disklabel.

Dette puttet jeg i $MERGEDIR/etc/disktab:

```
thebeDisk1:\
:dt=ST506:ty=fixed:se#512:nt#15:ns#63:nc#13446:sf \
:pa#2000000:oa#0:ta=4.2BSD:
```

Linje 3 forteller at jeg setter av en partisjon på 2.000.000 sektorer, som hver er på 512 bytes.

Aktiver med :

```
%/usr/ucb/rsh thebe arun /bin/disklabel -r -w hd0 thebeDisk1 Newlabel
```

Sjekk om alt gikk bra ved å lese disklabelen. Bruker du thebeDisk1 vil du få:

```
1 partitions:
#        size   offset    fstype   [fsize bsize   cpg]
a:    2000000        0    4.2BSD        0     0     0   # (Cyl.    0 - 2116*)
```

Formatere disken:

```
%/usr/ucb/rsh thebe arun /bin/newfs /dev/rhd0a
```

Sjekke at formateringen gikk bra:

```
%/usr/ucb/rsh thebe arun /bin/fsck /dev/rhd0a
```

Mounte disken:

```
%/usr/ucb/rsh thebe mount -t ufs /dev/hd0a /part1
```

Nå skal disken være klar for bruk. Hvis det er noe som ikke fungerer som det skal, se i "File Systems User's Guide" fra Chorus.

Spesifikasjon av disken på thebe finnes på Seagate sine sider, prøv en av disse adressene :

```
http://www.seagate.com:80/cgi-bin/view.cgi?/at/st36531a.txt
ftp://ftp.seagate.com/techsuppt/at/st36531a.txt
```

Referanser

[1] Thomas Plagemann, Vera Goebel, “INSTANCE: The Intermediate Storage Node”, Universitetet i Oslo, ResearchIndex.

[2] Raj Jain, “The art of computer systems performance analysis”, Wiley-Interscience, New York, NY, April 1991.

[3] Gary R. Wright, W. Richard Stevens, “TCP/IP Illustrated, Volume 2 The Implementation”, Addison-Wesley, 1995.

[4] Margo Seltzer, Peter Chen, John Ousterhout, “Disk Scheduling Revisited“, In Proceedings of the Winter 1990 USENIX Conference, pages 313–324, January 1990.

[5] Jos’e Carlos Brustoloni, “Interoperation of Copy Avoidance in Network and File I/O”, In Proc. of the IEEE Infocom Conference, New York, Mar. 1999.

[6] Christos Papadopoulos, Gurudatta M. Parulkar, “Experimental evaluation of SunOS IPC and TCP/IP protocol implementation”, IEEE/ACM Transactions on Networking Vol.1, No.2 April 1993.

[7] Kjersti Moldeklev, “Performance Analysis and issues of end systems attached to high-speed networks”, Doktor Ingeniøravhandling, NTH

[8] Andrew Gallatin, Jeff Chase, Ken Yocum, “Trapeze/IP: TCP/IP at Near-Gigabit Speeds”, In Proceedings of the 1999 USENIX Technical Conference (Freenix Track), June 1999.

[9] Igor Curcio, Antonio Puliafito, “Architecture of an I/O Subsystem for Video Servers”, IEEE Transactions on Computers Volume 47.

[10] Friedhelm Schmidt ,“The SCSI bus and IDE interface : protocols, applications and programming”, Addison-Wesley, Second Edition.

[11] David D. Clarc and David L. Tennenhouse, “Architectual Considerations for a New Generation of Protocols”, Proc. SIGCOMM’10 Symposium, Sept. 1990, pp. 200–208.

[12] L.L. Peterson og B.S. Davie: "Computer Networks - A Systems Approach, Second Edition", October 1999.

[13] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz and D. A. Patterson, "RAID: HighPerformance, Reliable Secondary Storage", ACM Computing Surveys, 26(2):145– 185, 1994.

[14] Leonard Chung, Jim Gray, Bruce Worthington, "Windows 2000 Disk IO Performance", June 2000, Microsoft Research Advanced Technology Division

[15] Kenneth Yun, "A High-Performance Asynchronous SCSI Controller", In Proc. International Conf. Computer Design (ICCD), pages 44–49, October 1995.

[16] Andrew S. Tanenbaum, "Modern Operating Systems", Prentice Hall International, 1992.

[17] Andrew S. Tanenbaum, "Distributed Operating Systems", Prentice Hall International, 1995.

[18] Karl-Andre Skevik, Thomas Plagemann, Vera Goebel, Pål Halvorsen, "Experimental Analysis of a Zero-Copy Protocol Implementation", Universitetet i Oslo.

[19] Peter A. Steenkiste, Design, "Implementation, and Evaluation of a Single-Copy Protocol Stack" Software - Practice and Experience,1998.

[20] William D. Norcott, Don Capps, "Iozone Filesystem Benchmark".

[21] Hawlett-Packard Company, "Netperf: A Network Performance Benchmark", revision 2.0, 1996.

[22] Intel Corporation, "Using the RDTSC Instruction for Performance Monitoring", ftp://download.intel.com/software/idap/ media/pdf/rdtscpm1.pdf, 1998.

[23] Intel Corporation, "Intel Architecture Software Developer's manual - Volume 2: Instruction Set Reference", Order Number 243191, ftp://download.intel.com/design/pentiumii/manuals/243191.htm, 1999.

[24] Steinmetz, R., Nahrstedt, K, "Multimedia: Computing, Communications and Applications", Prentice Hall, 1995

[25] John K. Ousterhout, "Why aren't operating systems getting faster as fast as hardware?" Summer USENIX, pp. 247-256, June 1990.

[26] Marshall Kirk McKusick, William N. Joy, "A Fast File System for UNIX", ACM Transactions on Computer Systems 2, 3 (August 1984), 181-197.

[27] Karl-André Skevik, "Memory Management for High Speed Protocols", Hovedfagsoppgave, Universitetet I Oslo, August 2000.

[28] Peter Druschel. "Operating System Support for High-Speed Networking: techniques to eliminate processing bottlenecks in high speed networking", Communications of the ACM, 39(9):41-51, September 1996.

[29] Bruce L. Worthington, Gregory R. Ganger, "Scheduling for Modern Disk Drives and Non-Random Workloads", SIGMETRICS, 1994, pp. 241-251 and Technical Report CSE-TR-194-94, 1994.

[30] Vivek S. Pai. "Io-lite: A Copy-free Unix I/O System", Hovedoppgave, Rice University, Januar 1997.

[31] J. Postel. RFC 791: "Internet Protocol", September 1981.

[32] J. Postel. RFC 793: "Transmission Control Protocol", September 1981.

[33] J. Postel. RFC 768: "User Data Protocol", August 1980.

[34] Steve McCanne, Craig Leres, Van Jacobson, "Tcpdump 3.4", Lawrence Berkeley National Laboratory Network Research Group, 1998.

[35] W. Richard Stevens. "UNIX Network Programming", volume 1, Prentice Hall PTR, 2nd edition 1998.

[36] Stephen G. Kochan and Patrick H. Wood "Unix Networking", Hayden Books, 1989.

[37] IBM, "Ultrastar 36Z15, 15.000 rpm 36-GB hard disk drive", http://www.storage.ibm.com/press/hdd/20010130.htm

[38] T. Braun and C. Diot, "Protocol Implementation Using Integrated Layer Processing", 151-161, ACM SIGCOMM OCTOBER 95

[39] R. Dean and F. Armand. "Data Movement in Kernelized Systems", USENIX Workshop on Micro-Kernels and Other Kernel Architectures, 243-261, April 1992.

[40] John D. McCalpin, "Memory bandwidth and machine balance in current high performance computers", IEEE TCCA Newsletter, Desember 1995.

[41] Crowcroft J., Wakeman I., Wang Z., "Is Layering Harmful?", IEEE Network Magazine, Vol. 6, Nr. 1, 20-24, Januar 1992.

[42] Chorus Systems, "Chorus/Classix r3.1 Programming Overview", CS/TR-96-188.2, 1996.

[43] Chorus Systems, "ChorusOS User's Guide", CS/TR-96-33.8, Release 3, 1998.

[44] Andrew Gallatin, Jeff Chase, and Ken Yocum, "Trapeze/IP: TCP/IP at Near-Gigabit Speeds", Duke University.

[45] P. J. Denning "Effects of scheduling on file memory operations", Proc. AFIPS SJCC , 31, 1967.

[46] McKusick, William Joy, Laffler, Fabrym, "A Fast Filesystem for UNIX", ACM Transactions on Computer Systems, 2(3):181-197.

[47] A. Gotti. "The Da Capo Communication System", Swiss Federal Institute of Technology, Zuerich, Switzerland, June 1994.

[48] Marshall Kirk McKusick et al. "The Design and Implementation of the 4.4BSD Operating System". Addison-Wesley', 1996.

[49] Charles D. Cranor, Gurudatta M. Parulkar, "The UVM Virtual Memory System", Proc. of 1999 USENIX Annual Technical Conf., Monterey, CA, USA, June 1999.

[50] C. Cranor. "Design and Implementation of the UVM Virtual Memory System" PhD thesis, Washington University, August 1998.