UNIVERSITETET I OSLO
Institutt for informatikk

Ytelseanalyse av Froots og Dimension-Order

Masteroppgave
(60 studiepoeng)

Bjørn Arne Dybvik

01. 08 2006

FORORD

Denne oppgaven er en del av min mastergrad ved universitetet i Oslo.

Oppgaven er løst ved hjelp av en simulator som er utviklet ved ND-avdelingen på Simula Research Laboratory. Jeg har altså ikke laget hele simulatoren, men gjort noen modifikasjoner.

Jeg vil først og fremst takke mine foreldre, søster og bestemor for støtten jeg har fått til å kunne gjennomføre utdannelsen. De neste jeg vil takke er mine venner for moralsk støtte.
Til slutt vil jeg takke mine veiledere, Åshild Grønstad Solheim og Tarik Cicic, for en fantastisk tålmodighet og god hjelp.

Bjørn Arne Dybvik

Innhold

Kap.1 Introduksjon

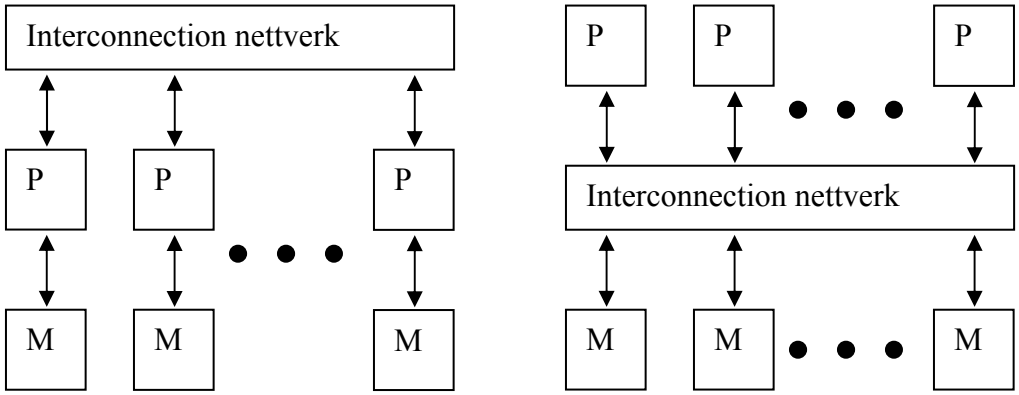
Værvarslingsystemer er eksempler på systemer hvor det ofte er store mengder data som skal behandles på kortest mulig tid. Dette krever ofte enormt med prosesseringskraft – gjerne mer enn vi kan oppdrive med dagens enkeltprosessorer. En tanke kan da være å satse på utvikling av enda kraftigere prosessorer. En har kommet frem til at det er en mindre bra løsning. Grunnen er at utviklingen av prosessorer er dyrt, i tillegg til at det er en liten kjøpergruppe som igjen medfører at det er kun de største firmaene som har råd til å utvikle prosessorer. En har med bakgrunn i dette kommet frem til at en bedre løsning er å utvikle parallelldatamaskiner.

I en parallelldatamaskin er flere prosessorer sammenkoblet ved hjelp av et interconnectionnettverk. En stor jobb kan da deles opp i mindre jobber som en distribuerer blant prosessorene, for til slutt å samle det til en ferdig jobb. Eksempelvis kan en på denne måten korte ned en jobb, som kanskje tar fire dager, til en dag.

Hyllevarekomponenter er designet for imøtekomme krav i nettverk som LAN og WAN (internett). Altså er de ikke tiltenkt å imøtekomme den størrelsesorden på dataoverføringene som for eksempel værvarslingssystemet vil produsere. Det vil medføre flaskehalser. Man kan imøtekomme økende prosesseringskraft ved å distribuere minne mellom prosessorene. En kan også bygge filsystemer som imøtekommer den økte prosesseringen ved å bruke RAID - teknologi, men interconnectionnettverket, som skal forbinde prosessorene, minne, disker og andre enheter, blir en flaskehals.

Kravene til interconnectionnettverket er avhengig av arkitekturen til parallelldatamaskinen og det finnes flere typer arkitekturer. Eksempler er multidatamaskin og Unified Memory Access (UMA) delt-minne multiprosessor, henholdsvis figur 1.1 a) og b):

.



a) Multidatamaskin b) UMA delt-minne multiprosessor

Figur 1.1: To arkitekturer for parallell datamaskiner.

Hvilket interconnectionnettverk en skal velge, vil være avhengig av flere faktorer; skalerbarhet,

hvor enkelt det skal være, ytelseskrav, avstand, fysiske begrensninger, pålitelighet og mulighet for reparasjoner, kostnadsbegrensninger, forventet arbeidslast, partisjonerbarhet og inkrementerbarhet. Partisjonerbarhet har som siktemål at trafikken til en bruker ikke skal påvirke en annen bruker. Dette blir løst ved at nettverket blir partisjonert i mindre systemer. Inkrementerbarhet betyr at en kan legge til enheter etter at en har kjøpt maskinen. Eksempelvis kan det være at en kunde ønsker å legge til flere prosessorer etter hvert.

Interconnectionnettverk kan kategoriseres i fire hovedkategorier, hvorav to av de er direkte- og indirekte nettverk. I direkte nettverk er hver prosesserende node også en svitsjende node, mens det i indirekte nettverk er skilt mellom de prosesserende nodene og de svitsjende nodene ved en link.

Ruting, topologi og svitsjing karakteriserer både direkte - og indirekte nettverk. Topologien bestemmer hvordan nodene er lagt ut og hvordan de er forbundet via linker. Rutingalgoritmen bestemmer stien som meldingene tar gjennom nettverket, mens det er svitsjemekanismen som bestemmer når en melding kan traversere en node. Vi har mange typer rutingalgoritmer og vi kan klassifisere disse etter kriterier som hvor rutingavgjørelsen blir tatt og hvordan algoritmen er implementert.

Stien til meldingen kan bestemmes før meldingen injiseres i nettverket eller stien kan lages mens meldingen traverserer nettverket. Det første tilfellet blir kalt kilderuting. Vi har to typer kilderuting; deterministisk og adaptiv. Deterministisk kilderuting vil si at samme stien alltid blir gitt mellom et kilde-destinasjonspar. Adaptiv kilderuting tar hensyn til tilstanden i nettverket før rutingavgjørelsen tas. Er en kanal blokkert, er det mulig for algoritmen å adaptivt velge en annen kanal. Det neste tilfellet kalles distribuert ruting. Distribuert ruting innebærer at hver mellomliggende node som meldingen må traversere, for å komme til sin destinasjon, må ta en avgjørelse basert på sin kunnskap om nettverket. Denne rutingtypen kan være basert på at hver node har lokal kunnskap om nettverket eller at rutingtabeller er lastet ned fra en sentral enhet som står for kalkulering av rutene. En siste rutingmulighet er flerfaseruting. Flerfaseruting vil si at vi ruter pakken i flere faser fra kildenode til destinasjonsnoden. I flerfaseruting regner kildenoden ut noen mellomliggende noder. Pakken kan sendes til alle nodene eller kun til destinasjonsnoden. Mellomliggende noder kan brukes for å unngå feil.
Sendes pakken til alle nodene, kaller vi det multicast ruting mens det siste kalles unicast ruting.

Viktige temaer innenfor interconnectionnettverk er throughput, latency og feiltoleranse.
En melding kan defineres som kommunikasjonsenheten sett fra programmererens synspunkt. En melding deles som regel opp i pakker før den sendes gjennom nettverket. Pakkene kan ha forskjellig størrelse. Eksempelvis kan de være på noen hundre bytes. Throughput er en metrikk som sier noe om hvor mange pakker som har nådd frem til destinasjonene sine per tidsenhet.

Hver pakke har en viss distanse den må traversere for å komme til målet sitt. Det vil alltid ta en tid å traversere denne strekningen og eksempelvis kan denne tiden måles i nanosekunder. Latency defineres som den tiden en pakke bruker fra kildenode til destinasjonsnode.

Av og til er det slik at enheter i nettverket feiler. Eksempelvis kan det være en node som plutselig feiler under kjøring. Det er da viktig at systemet er feiltolerant. Feiltoleranse defineres

som evnen et system har til å fortsette sin videre aktivitet når komponenter/enheter svikter i nettverket.

Et feiltolerant system bør kunne tåle at feil oppstår mens systemet kjører og at systemet ikke blir avbrutt av feilen. Det eneste systemet bør merke er nedgang i throughput. I tillegg bør det kunne settes inn ekstra komponenter uten at systemet trenger å stoppe sin virksomhet [Theiss]. I visse omgivelser kan det være vanskelig å reparere et system og/eller komponentene feiler til ulike tider. Et eksempel kan være system i fly; uten feiltoleranse kan det mest katastrofale inntreffe.

En av de viktigste oppgavene når en bygger et feiltolerant system, er å designe en feiltolerant rutingalgoritme. Feil i et nettverk kan føre til at meldinger ikke kommer frem til destinasjonsnoden. Flere kategorier feil kan oppstå i et nettverk som statiske, dynamiske og transiente feil. En statisk feil er en feil som er der hele tiden - den er der fra systemet startes opp. Dynamiske feil er feil som skjer under kjøring. Et eksempel på en dynamisk feil er hvis en node i nettverket feiler mens systemet kjører. Transiente feil er feil som kan forsvinne og komme tilbake.

Feil i et nettverk kan føre til at en rutingalgoritme ikke blir vranglåsfri lengre. En vranglås fører til at et sett med pakker i et nettverk blir låst for alltid hvis ikke vranglåsen blir løst opp. Vranglås blir forklart mer inngående i kapittel 2.

Med bakgrunn i det ovennevnte, skal jeg i denne oppgaven sammenligne ytelsen til to rutingalgoritmer. Innledningsvis ble det forklart at rutingalgoritmer blir klassifisert etter blant annet to kriterier - deterministisk og adaptiv. En godt kjent deterministisk rutingalgoritme er Dimension-Order rutingalgoritmen. Et eksempel på en adaptiv rutingalgoritme er FRoots. FRoots er i utgangspunktet feiltolerant, mens Dimension-Order i utgangspunktet ikke er feiltolerant. Men, som vi skal se er Dimension–Order algoritmen utvidet med en metode som gjør at den blir feiltolerant.

1.1 Oppbygging av oppgaven

I kapittel 2 blir temaer som vranglås, rutingalgoritmer og rekonfigurering beskrevet.
Kapittel 3 vil ta for seg problemområdet og siktemålet til oppgaven i tillegg til algoritmene som skal analyseres.

Kapittel 4 vil ta for seg den metoden som skal brukes, mens kapittel 5 vil ta for seg resultatene og en diskusjon av disse.

I kapittel 6 får vi konklusjonen.

Kap. 2 Bakgrunn

Flytkontroll kan brukes på flere nivåer som ende-til-ende flytkontroll eller for å hindre at en link overfylles med pakker. Ende-til-ende flytkontroll vil si at flytkontrollen foregår mellom sender og mottaker. Av og til er det også slik at hvis vi har flere subnett, kan eksempelvis et subnett sende pakker over til et annet subnett raskere enn subnettet kan ta unna. Ved et slikt tilfelle kan flytkontroll brukes til å hindre at subnettet får for mange pakker.

I interconnectionnettverk brukes flytkontroll på linknivå. Det betyr at flytkontroll brukes for å hindre overfulle buffer på linken og pakketap.

[Duato] deler inn kommunikasjonen mellom prosessorer inn i tre lag; rutinglaget, svitsjelaget og det fysiske laget. Det fysiske laget vet hvordan nodene er lagt ut og hvordan de skal rute bits og bytes over linkene. Svitsjelaget utnytter det fysiske laget til å sende meldinger gjennom nettverket. I rutinglaget foregår rutingen for å bestemme hvilken sti en melding skal bruke gjennom nettverket og hvilke utlinker den skal ta på hver mellomliggende svitsj. Rutingalgoritmene hører til i rutinglaget mens svitjsemekanismene hører til i svitjselaget.

Denne oppgaven skal ta for seg rutingalgoritmer og feiltoleranse. Det er linknivå flytkontroll som brukes i oppgaven.

Dette kapittelet er et bakgrunnskapittel som vil ta for seg emner som er viktig innenfor disse områdene.

2.1 Interconnectionnettverk

Et interconnectionnettverk er et nettverk hvor pakkene ikke kan kastes på grunn av fulle buffer, en link-nivå flytkontroll stopper datapakker for å forhindre overfulle buffer, det er et høyhastighetsnettverk og nettverket kan være cut-through rutet [Theiss]. Mer om svitsjeteknikker som cut-through ruting i kapittel 2.7

Eksempler på hvor interconnectionnettverk brukes er blant annet i teknologier som Autonet[Schroeder], ServerNet[Horst], MyriNet[Myrinet], Infiniband[inf] og PCI-Express AS[PCI].

2.2 Innledende begreper

Flere begreper er sentrale og brukes både i beskrivelsen av rutingalgoritmer og nettverk. Dette er begreper som noder, linker, virtuelle kanaler, buffere, virtuelle lag, pakker, flytkontroll og svitsj. Begrepene vil innledningsvis bli forklart nærmere.

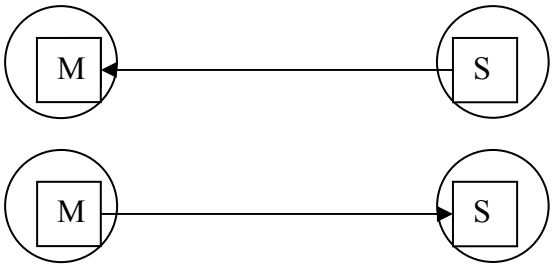
2.2.1 Endenoder, noder og linker

Det er endenodene i nettverket som genererer og tar imot meldinger. En endenode er en enhet

som kun står for generering av og mottak av pakker. Nodene sender meldingene generert av endenodene ut i nettverket mot sin destinasjon. I vårt tilfelle er noden en svitsj, men den kan også være en ruter [Duato].

Nodene i et nettverk blir forbundet via linker. Linkene er det mediet som meldingene sendes over. En link er et punkt-til-punkt kommunikasjonsmedium som forbinder to noder, og har en buffer på hver side [Theiss]. En link er illustrert i figur 2.2.1.

Det finnes linker som ikke er punkt-til-punkt kommunikasjonsmedium. Eksempel på dette kan være busser med mer enn to noder tilknyttet, men det er ikke viktig for denne oppgaven.

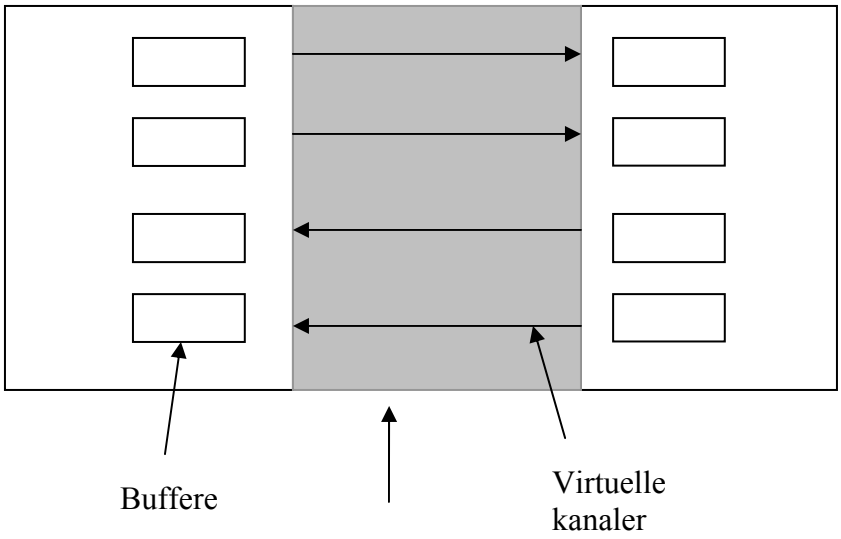


Figur 2.2.1: Link mellom to noder med tilhørende buffer. S står for sendebuffer og M står for mottakende buffer. I dette tilfellet kan sender utnytte hele linjen når den sender pakker.

Alle linkene i denne oppgaven kan tilby både sender og mottaker både å sende og motta pakker individuelt i tillegg til at de kan sende samtidig.

2.2.2 Virtuelle kanaler og buffere

En virtuell kanal vil si at den fysiske linken deles videre opp i flere linker(virtuelle). Linken vil da få tilhørende egress – og ingress buffer på hver side for hver virtuell kanal (figur 2.2.2 viser virtuelle kanaler). En virtuell kanal er uavhengig av de andre virtuelle kanalene, bortsett fra å dele den fysiske kanalen, og hver virtuell kanal oppfører seg som en selvstendig kanal. Dersom vi antar at den fysiske kanalen kan overføre 150 KB/sek., og vi deler den slik at det blir en ekstra kanal, kan det for eksempel være at hver kanal overfører 75 KB/sek. Men, det er ingenting i veien for at kanalene kan ha forskjellige overføringshastigheter.



Figur 2.2.2: Fysisk kanal som er delt opp i virtuelle kanaler – 2 i hver retning.

Virtuelle kanaler kan blant annet brukes til å øke nettverksgjennomstrømningen, minske latency eller løse opp vranglåser [Duato]. En måte å øke nettverksgjennomstrømmingen på er å bruke virtuelle kanaler til å løse ”head of line” blokkering av pakker. Anta at vi har en FIFO-kø; hvis da en pakke blir blokkert på i køen, vil den stanse alle pakker bak den. Dette kalles ”head of line” blokkering [Tannenbaum].

I denne oppgaven skal vi bruke virtuelle kanaler til både å løse opp vranglåser, unngå ”head of line”-blokkering og til feiltoleranse.

2.2.3 Virtuelt lag

Et virtuelt lag kan sees på som et virtuelt nettverk. Det virtuelle nettverket består av alle nodene i det fysiske nettverket, men bare en del av de virtuelle kanalene. Alle de fysiske linkene har en virtuell kanal som representant i det virtuelle laget.

Kanalene til ulike virtuelle nettverk er ulike. Eksempelvis ser vi i figur 2.2.3 b) at det virtuelle nettverket består av de virtuelle kanalene som går i X+ - og Y+ retning (altså positiv X og Y

retning).

Når en sender en pakke inn i et gitt virtuelt nettverk kan den nå noen destinasjoner, men som regel ikke alle.

a) Fysisk nettverk b) Virtuelt nettverk

Figur 2.2.3: Et fysisk nettverk til venstre og et virtuelt nettverk til høyre.

Flere rutingalgoritmer bruker virtuelle lag. For eksempel bruker FRoots det for å sikre pakker en sti til målet selv når det har oppstått en feil. Mer om dette i kapittel 3.

Senere i kapittelet skal vi også se eksempel på hvordan et virtuelt lag kan brukes til å forhindre vranglås.

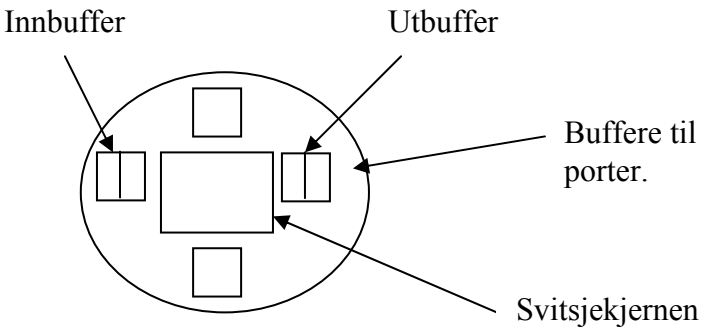
2.2.4 Pakker og flytkontroll

I kapittel 1 ble det forklart at en melding kan deles opp i pakker. Pakken har ofte en bestemt størrelse, i tillegg inneholder den rutinginformasjon. Rutinginformasjonen ligger som regel i det vi kaller pakkehodet som er de første bitsene av en pakke. Som et eksempel kan vi ta med at pakkestørrelsen kan være satt til 256 bytes hvorav 64 bytes defineres som pakkehodet. Rutinginformasjonen sier blant annet noe om hvilken node pakken skal til.

Flytkontroll er en synkroniseringsprotokoll mellom sender og mottaker. Den definerer hva som skal skje hvis bufferen på mottakernoden er fullt eller en feil har oppstått. Flytkontroll kan foregå på to nivåer hvorav det ene nivået er pakkenivå og det andre er flitnivå. En flit kan defineres som noen bits av en pakke, men en flit kan også defineres som en hel pakke. Det er viktig å merke seg at det ikke er flit som blir overført. Flits kan defineres som logiske informasjonsenheter. Det som fysisk overføres over linken, kalles phits og vi kan si at phits er den fysisk overføringsenheten.

2.2.5 Svitsj

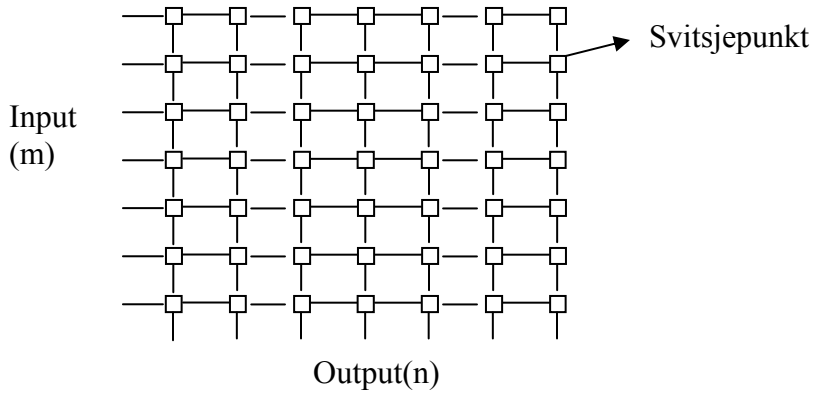
Svitsjen er altså den enheten i nettverket som står for videresendelse av pakker. Svitsjen består av et gitt antall porter og via portene kan svitsjen tilkobles andre svitsjer eller endenoder.



Figur 2.2.4: Svitsjen

Svitsjekjernen består blant annet av en svitsjemekanisme, inn- og utbuffere, en crossbar og en arbitreringshåndterer (figur 2.2.4). Svitsjemekanismen binder sammen bufferen til innkanaler (ingressbuffer) med bufferen til utkanaler (egressbuffer) slik at pakker kan videresendes. Svitsjemekanismen bestemmer også når videresendingen av pakken skal skje. Vi har flere typer svitsjemekanismer som vi altså vil komme tilbake til senere i kapittelet.

Svitsjekjernen inneholder også et nettverk som forbinder ingressbufferne til egressbufferne. Dette nettverket kaller vi en crossbar. En crossbar er et nettverk som kan tillate kommunikasjon mellom flere enheter samtidig. Det er vanlig å modellere en crossbar som vist i figur 2.2.5.



Figur 2.2.5: Crossbar.
Tegningen er hentet
fra [Duato].

Av figur 2.2.5 ser vi at crossbaren har m innganger og n utganger. Det betyr at den kan ha like mange innkanal til utkanal kommunikasjoner som den minste verdien av m og n (det er vanlig at de to verdiene er like).

Arbitreringshåndtereren er den enheten som velger hvilken av innportene som skal få sende til en utport. Det er flere strategier en kan bruke for å velge hvilken innport som skal få sende. Eksempelvis kan en bruke round-robin.

For å ta med et eksempel, viser vi en innbuffer og et utbuffer i figur 2.2.4. Begge bufferne har plass til to pakker. Hvis innbufferet skal sende pakker til utbufferet, må de sende via crossbaren og det er arbitreringshåndtereren som velger når det aktuelle bufferet skal få sende.

Egenskaper som det er interessant å skille mellom, med hensyn på svitsjer, er om de er blokkerende eller ikke–blokkerende og om de er rettferdige eller ikke–rettferdige [Theiss].

At de er ikke–blokkerende, vil si at svitsjen lar trafikken bli overført mellom ethvert portpar uavhengig av trafikken i en annen port. Dette skjer så lenge utporten ikke er blokkert av en pakke fra en den andre porten[Theiss].

En svitsj er rettferdig hvis ingen innport må vente uendelig lenge for å få sende pakker til en utport [Theiss].

2.3 Topologi

Nodene kan legges ut i nettverket på mange måter. Topologien viser hvordan nodene er lagt ut i nettverket og hvordan det er forbundet til hverandre via linker. Det finnes mange forskjellige topologier, og det er vanlig å skille mellom to hovedklasser av topologier; indirekte og direkte nettverk. Forskjellen på disse to nettverkstypene, er at i direkte nettverk, er hver node også en prosesserende node også en svitsjande node, mens det i indirekte nettverk er skilt mellom dee prosesserende nodene og rutingnodene. De har et nettverk mellom seg.

En topologi kan representeres som en graf G=(N, C) der N representerer rutingnodene og kantene C er kanalene.

Viktige egenskaper med topologier, er om de er sammenhengende, konnektiviteten til topologien og biseksjonsbredden.

Topologien er sammenhengende hvis rutingfunksjonen kan finne en sti mellom alle par av noder. Hvis ikke er den usammenhengende.

Hvilken konnektivitet en topologi har, sier noe om antall linker på hver node. Topologien har høyere konnektivitet desto flere linker det er på hver node. Eksempelvis kan vi ta med at i nettverk med få noder er hver enkelt node forbundet til et høyere antall noder enn i et nettverk med flere noder. I et nettverk med få noder, vil det være høyere konnektivitet enn i et nettverk med mange noder.

Biseksjonsbredden er en annen viktig egenskap. Det er en metrikk som sier noe om hvor få noder en kan fjerne fra et nettverk, før det ikke lenger er forbundet eller kan sees på som om det er delt i to partisjoner. Eksempelvis kan vi ta med at hvis en node feiler i et nettverk, og det ikke lenger er mulig å sette opp stier mellom alle nodene i nettverket, sier vi at biseksjonsbredden er 1. Hvis, derimot, nettverket fortsatt har forbindelse etter at en node har feilet, men mister forbindelsen etter to, sier vi at biseksjonsbredden er to. Dette er en viktig egenskap for feiltoleranse.

Det er vanlig å dele topologiene i to typer; regulær – og irregulær topologi.

2.3.1 Regulære topologier

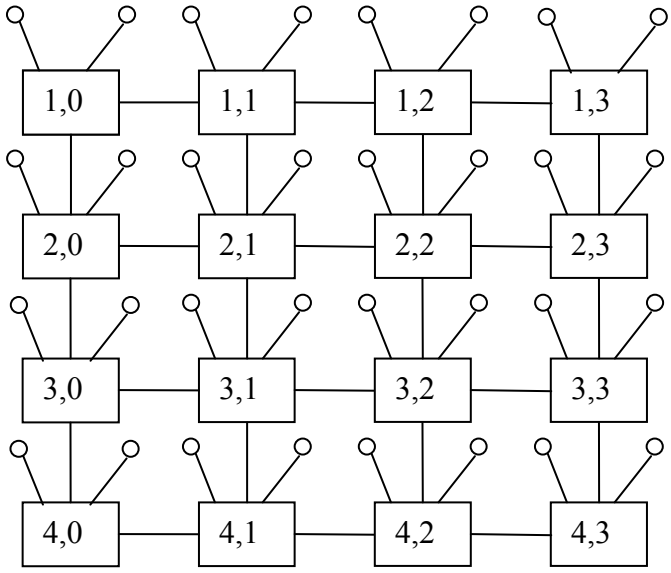
En regulær topologi er en topologi med en forutsigbar struktur. For å beskrive hva en regulær topologi er, kan vi bruke et eksempel. Eksempler på regulære topologier er blant andre mesh (rutenett) og k-ary n-cubes.

2.3.1.1 Rutenett

I en 2D mesh blir hver node representert som en koordinat. Bortsett fra endenodene og hjørnenodene, er alle nodene tilkoblet fire andre noder. Kantnodene er tilkoblet tre andre noder,

mens hjørnenodene er tilkoblet to andre noder. Vi ser en 2D mesh i figur 2.2.1.

[Duato]: Formelt har en n-dimensjonal mesh $k_0 \times k_1 \times \ldots\ldots \times k_{n-2} \times k_{n-1}$, k_i noder langs hver dimensjon, og $k \geq 2$, $0 \leq i \leq n-1$. Hver node er identifisert av n koordinater, $(x_{n-1}, x_{n-2}, \ldots\ldots, x_1, x_0)$ og $0 \leq x_i \leq k_i - 1$.



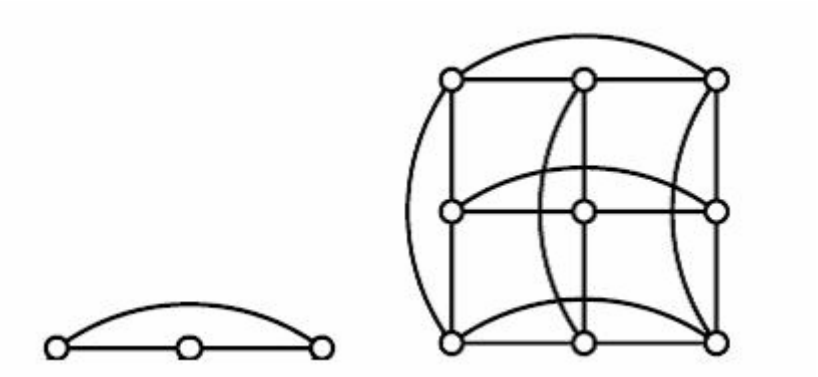
Figur 2.3.1: 2D mesh med 16 noder. I hver node er det angitt en koordinat som forteller hvilken posisjon den gitte noden har i nettverket.

Det er vanlig å betrakte en 2D mesh som et direkte nettverk. Til tross for det, blir den i denne oppgaven betraktet som et indirekte nettverk. Grunnen til det er at det er tilknyttet to endenoder til hver svitsj (figur 2.3.1). Endenodene er tilkoblet svitsjen via en link.

2.3.1.1 K-ary n-cube

En k-ary n-cube er en annen vanlig topologi. Vi har flere typer av denne topologien og de mest vanlige er 3-ary 2-cube, hyperkube og torus.

For å forklare hva en k-ary n-cube er, kan vi starte med en 1-ary 3-cube. Det kan vi se på som en ring med 3 noder (figur 2.3.2 a))



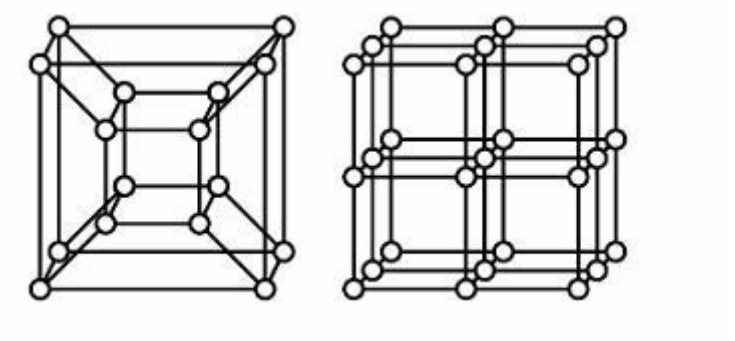
a) 1-ary 3-cube b) 3-ary 2-cube

Figur 2.3.2: Rundtkoblingslinker.

Av figur 2.3.2 b), ser vi at en 3-ary 2-cube er sammen satt av flere 1-ary 3-cuber og vi ser prinsippet med rundtkoblingslinker. Forskjellen mellom en k-ary n-cube og en n-dimensjonal mesh er at i den første topologien har alle naboene likt antall naboer.

Formelt kan vi skrive at i en k-ary n-cube er alle k_i lik k og nodene X og Y er naboer kun hvis $x_i = y_i$ for alle i, $0 \leq i \leq n-1$ bortsett fra en, j, der $y_j = (x_j + 1) \bmod k$. Det er forandringen til modulær aritmetikk som gjør at vi får rundtkoblingslinker. Alle nodene har n naboer hvis k=2 og 2n naboer hvis k>2. Som vist i 2.3.2 a) blir k-ary n-cube en ring med to retninger hvis k=1.
Det betyr at kantnodene og hjørnenodene er tilkoblet hverandre med ekstra linker.

Som et annet eksempel kan vi ta med hyperkube. Navnet hyperkube har den fått fordi det er en vanlig topologi. Det er to kuber som er satt sammen på en slik måte at vi har en kube som er større enn den andre, og den store kuben omslynger den mindre kuben (se figur 2.3.3a)).
Det er en n-dimensjonal mesh hvor $k_i = 2$ for $0 \leq i \leq n-1$. Men det kalles også en 2-ary n-cube[Duato].



a) Hyperkube (2-ary 4-cube)

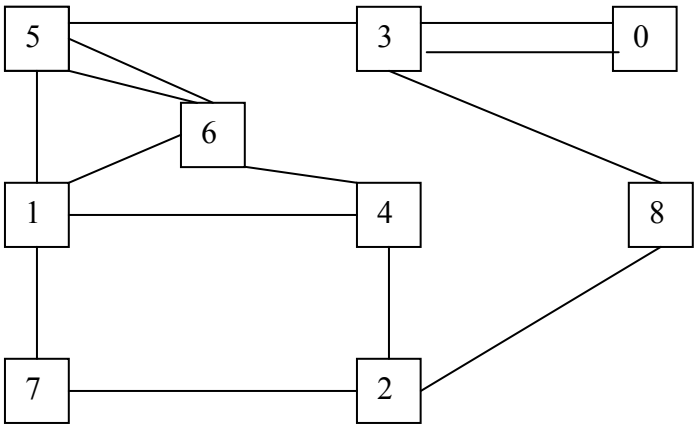
b) 3x3x3 mesh

Figur 2.3.3: Eksempler på forskjellige k-ary n-cuber. Bildene er hentet fra [Theiss]

2.3.2 Irregulære topologier

I irregulære topologier har vi ikke den samme strukturen som hos regulære topologier. En irregulær topologi følger ikke noe forhåndsdefinert mønster [Duato]. Denne typen topologier blir for eksempel brukt i networks of workstations (NOW).

Figur 2.3.4 viser et eksempel på en irregulær topologi..



Figur 2.3.4: Eksempel på en irregulær topologi [Duato]

En fordel med irregulære topologier er at det enkelt kan legges til en ny node. Dette er vanskeligere å gjøre i regulære topologier.

2.4 Rutingalgoritmer

Ovenfor forklarte vi hva en topologi er. Topologier kan vi si er det mediet som rutingalgoritmer opererer i. Rutingalgoritmen bestemmer hvilke noder og linker en pakke skal innom på sin reise mot destinasjonen.

En rutingalgoritme modelleres ved hjelp av to funksjoner: ruting- og seleksjonsfunksjonen. Rutingfunksjonen har et sett med noder og kanaler som den tar i betraktning når den skal gi tilbake utkanaler. Settet kaller vi domenet til rutingfunksjonen. Domenet kan variere og eksempler på domener er NxN->C. Det betyr at rutingfunksjonen tar i betraktning noden pakken er på for øyeblikket (nåværende node) og destinasjonsnoden for å gi tilbake en kanal (evt. et sett med flere kanaler). Et annet domene er CxN->C; da bruker rutingfunksjonen en "dummy"-kanal som den gir til rutingalgoritmen. Det betyr at $R(-k_k,k_d)$ gir første hoppet. Ved domenet NxN->C ville innparameterene sett slik ut: $R(k_k,k_d)$.

Definisjon 1 Rutingfunksjonen kan derfor defineres som R: NxN->C. R tar inn en gitt node (nåværende node) og en destinasjonsnode, og gir tilbake en kanal som går fra nåværende node og til en umiddelbar nabonode. Hvis vi kaller rutingfunksjonen R, så vil altså R (nåværendenode, destinasjonsnode) gi en kanal $nåværendenode_{kanal}nabonode$.

La oss ta med et eksempel; anta at vi bruker distribuert ruting; rutingfunksjonen tilbyr et sett av kanaler basert på den noden pakken er på(nåværende node) til noden en skal til (destinasjonsnoden). Fra det settet med kanaler som rutingfunksjonen tilbyr, velger seleksjonsfunksjonen en kanal. Det er den kanalen som eventuelt skal brukes og som blir tilbudt av rutingalgoritmen. Seleksjonsfunksjonen kan velge tilfeldig kanaler eller basere valget på tilstanden til kanalene.

Noen ganger kan det være at en ønsker å begrense rutingfunksjonen. Eksempler på situasjoner er hvis en skal løse opp en vranglås. Da kan det være hensiktsmessig å dele en rutingalgoritme inn i en begrenset rutingfunksjon og en seleksjonsfunksjon. Andre eksempler kan være hvis skal begrense transisjoner til noen kanaler. Vi skal se eksempler på begge tilfellene.

En begrenset rutingfunksjon, R_1, kan tilby alle kanalene som R tilbyr, men også bare en del av dem.

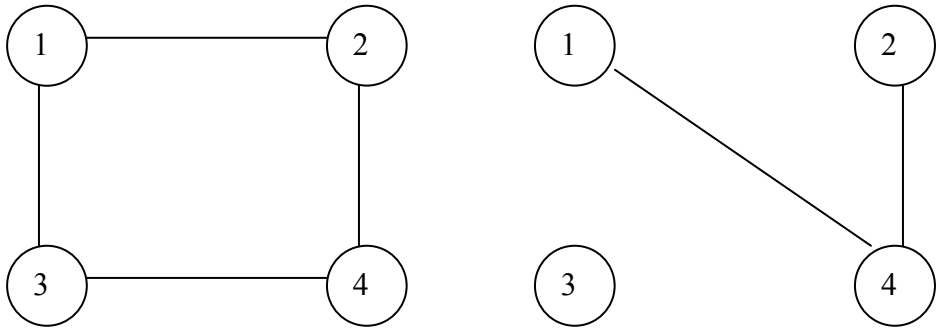
Definisjon 2 Formelt kan vi skrive at R_1 returnerer kanalsettet $\{x_cy_1, x_cy_2, ... x_cy_m\}$ og $m \geq 0$.

Det er viktig, og antatt i denne oppgaven, at rutingfunksjonen er sammenhengende. Det vil si at uansett hvilke nodepar en sender til den som kilde-destinasjonspar, klarer rutingfunksjonen å finne en sti.

Definisjon 3 En rutingfunksjon blir sagt å være sammenhengende hvis det eksisterer en sti mellom alle par av noder [Duato].

Til slutt kan vi ta med et eksempel på hvordan rutingfunksjonen fungerer; anta at vi bruker distribuert ruting og at vi skal sende en pakke fra node 1,0 til 4,0. Anta videre at domenet er NxN->C. Det første som skjer er at rutingfunksjonen får inn node 1,0 og 4,0 (R(1,0, 4,0)). Den vil da gi tilbake kanalen som går fra node 1,0 til 2,0 og pakken blir sendt til node 2. Neste gang rutingfunksjonen kalles på, sendes node 2,0 og 4,0 med som parametere. Rutingfunksjonen vil da gi tilbake kanalen som går fra node 2,0 til node 3,0. Den siste kanalen som gis tilbake er den fra node 3 til 4.

En konfigurasjon kan vi si er kombinasjonen av en topologi og en rutingalgoritme. For denne oppgaven, er det antatt at konfigurasjonen er sammenhengende. Dette til tross for at det skjer feil. Feil kan medføre at en topologi blir usammenhengende, eller ikke-forbundet. Det vil igjen medføre at konfigurasjonen blir usammenhengende. Hvis en konfigurasjon er sammenhengende, kan alle noder kommunisere med hverandre. Men, er topologien ikke–forbundet, er konfigurasjonen også usammenhengende. Som et eksempel kan vi ta med en forbundet og en ikke-forbundet topologi.



Figur 2.3.5: Eksempler på sammenhengende (a) og ikke-sammenhengende topologier(b).

Definisjon 4 En konfigurasjon er sammenhengende hvis topologien er sammenhengende. En kan da sette opp en sti mellom alle kilde-destinasjonspar.

Vi har mange typer rutingalgoritmer. Noen eksempler på rutingalgoritmer blir tatt med i kapittel 2.5.

2.4.1 Inndeling av rutingalgoritmer

Rutingalgoritmene klassifiseres etter flere kategorier. To av kategoriene er om algoritmen er deterministisk eller adaptiv og om algoritmen er minimal eller ikke-minimal. For denne oppgaven er det nok å skille mellom disse kategoriene.

2.4.1.1 Minimale, ikke-minimale og korteste sti rutingalgoritmer

Minimale rutingalgoritmer gir alltid den korteste lovlige stien mellom et nodepar. En minimal rutingalgoritme kan defineres slik: en minimal rutingalgoritme M velger en del av de korteste stiene i nettverket. En sti er en del av dette settet kun hvis den ikke bryter noen av begrensningene til rutingfunksjonen, og at det heller ikke finnes noen stier som er kortere.

En ikke-minimal rutingalgoritme velger en del av alle stiene i nettverket.

Og til slutt tar vi med forklaring på en korteste-sti rutingalgoritme. Den velger blant alle korteste-stiene i nettverket. I en korteste–sti rutingalgoritme R, er settet av lovlige ruter en del av alle korteste lovlige stiene i topologien. En sti er en lovlig rute til R kun hvis pakkene som følger denne stien ikke bryter noen rutingrestriksjoner, og det heller ikke er noen korteste stier som bryter noen rutingrestriksjoner.

2.4.1.2 Deterministiske rutingalgoritmer

Deterministiske rutingalgoritmer kan ikke legge om stien fra kildenoden til destinasjonsnoden. Stien er fastsatt i det pakken blir sendt ut i nettverket. Hvis rutingfunksjonen alltid tilbyr kun en sti mellom et nodepar, er den deterministisk.

Et eksempel på en deterministisk ruting er dimension-order. To eksempler på rutingalgoritmer som ruter dimension order, er e-cube algoritmen og XY ruting. Mer informasjon om disse algoritmene kan finnes i [Duato]. Siden dimension-order algoritmen blir nærmere forklart i kapittel 3, blir den ikke forklart nærmere her.

Deterministiske rutingalgoritmer brukes blant annet i Intel Paragon [Paragon], Cray T3D[Kessler] og MIT J-Machine [Noakes].

2.4.1.3 Adaptive rutingalgoritmer

Adaptive rutingalgoritmer tar hensyn til tilstanden i nettverket før rutingbestemmelsen blir tatt. Eksempelvis kan en adaptiv rutingalgoritme, så lenge den ikke er kilde adaptiv, velge en annen kanal på en mellomliggende node dersom det skulle vise seg at en kanal er blokkert. Den kan altså omlegge stien til en pakke mens den traverserer nettverket.

Ved bruk av kilderuting, er stien gjennom nettverket laget på forhånd, men også her blir det tatt hensyn til kanalenes tilstand.

Eksempler på adaptive rutingalgoritmer er planar–adaptiv ruting foreslått av [Chien] for n-dimensjonal mesher og hyperkuber, vest-først og P-kube.

Flere rutingalgoritmer, både deterministiske og adaptive, blir beskrevet mer utfyllende i 2.5. Men, først, blir begrepet vranglås forklart mer inngående.

2.5 Rutingalgoritmer og vranglås

Vranglås er et velkjent begrep innenfor dataverdenen og eksempelvis innenfor transaksjoner i databaseteknologien og i nettverk. Vranglås er viktig i interconnectionnettverk fordi at vi har link-nivå flytkontroll. Det vil si at pakker ikke blir kastet.

Vranglås vil si at vi har et sett med pakker og pakkene i settet venter gjensidig på hverandre på en syklisk måte. Vi sier da at vi har en vranglåskonfigurasjon. En konfigurasjon er alle pakkene i nettverket på et gitt tidspunkt.

Det er rutingfunksjonen som bestemmer om en rutingalgoritme er vranglåsfri eller ikke. Når rutingalgoritmer lager stier som pakker skal følge gjennom nettverket oppstår muligheten for vranglås. Vranglås medfører at pakker ikke kommer seg videre før vranglåsen er løst opp og blir den ikke det, må pakkene vente i all evighet.

For at det skal kunne oppstå vranglås må det minst dreie seg om to enheter (ressurser) som skal benyttes av andre enheter. I vårt tilfelle er det bufferne i svitsjene som er ressursene og det er pakkene som skal benytte bufferne.

Med hensyn på hvordan en håndterer en vranglås, er det forskjell på om vi har med en deterministisk – eller adaptiv rutingfunksjon å gjøre. Definisjon 5 er en nødvendig definisjon for å forhindre vranglås ved deterministisk ruting, men den er for streng for adaptiv ruting.

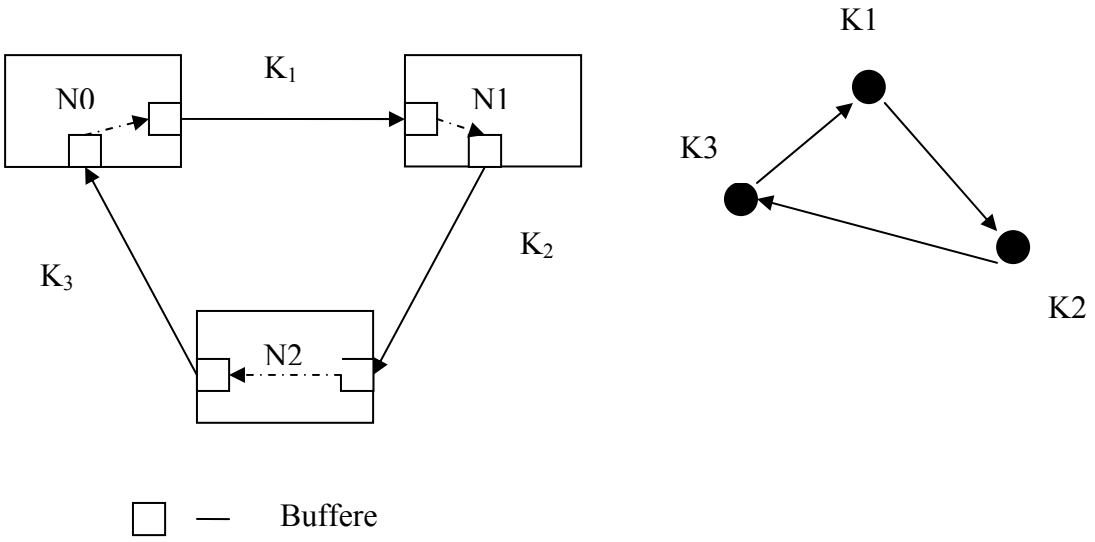
Definisjon 5: En (deterministisk) rutingfunksjon er vranglåsfri kun hvis det ikke eksisterer en sykel i kanalavhengighetsgrafen [Duato].

Grunnen til at det er skrevet deterministisk i parentes, er at adaptive rutingalgoritmer kan tillate sykler mellom noen kanaler. Grunnen er at de kan legge om stien til en pakke hvis en kanal er blokkert.

Først beskriver vi hva en kanalavhengighetsgraf er, før vi tar med et eksempel på en vranglåssituasjon. Deretter blir det forklart hvordan vi unngår vranglås ved bruk av deterministiske – og adaptive rutingfunksjoner.

Definisjon 6: Hvis en pakke kan bruke en kanal C_j etter en kanal C_i, sier vi at det er en avhengighet mellom kanalene C_j og C_i.

En kanalavhengighetsgraf er en graf hvor hver kanal, som er med, har en retning. Eksempelvis kan retningen være fra kanal K1 mot K2. Kanalavhegighetsgrafen, D = G (C, E), modelleres ofte på denne måten; C representerer linkene og E er avhengigheten mellom dem. Vi ser et eksempel på en kanalavhengighetsgraf i 2.5.1 b). Den gjelder for figur 2.5.1 a). Rutingfunksjonen for topologi 2.5.1 a) er som følger; hvis pakken har kommet frem til destinasjonsnoden, lagres pakken. Ellers bruker vi kanalene K_i. Avhengighetsgrafen viser at det kan oppstå en konfigurasjon som er i vranglås.



a) viser en konfigurasjon som er i vranglås. Pakkene sitter i bufferne hvor vi ser de stiplede pilene.

b) Kanalavhengighetsgrafen til a). Pilene viser hvilke kanaler det er avhengighet mellom.

Figur 2.5.1: Eksempel på en vranglåssituasjon.

Pakkene i bufferne ved starten av de stiplede pilene venter på at pakkene, ved slutten av pilene, skal komme seg videre. For å forklare vranglåssituasjonen, tar vi med følgende; anta at N1 skal sende til N0. Pakkene kan sendes over K_2, men må vente på K_3 fordi at N2 sender til N1. I det siste tilfellet kan pakkene sendes over link K_3, men må vente på K_1 fordi N0 sender til N2. Ved det siste tilfellet kan pakkene sendes over K_1, men må vente på kanal K_2. Av dette ser vi at alle pakkene venter på en kanal som er opptatt, og vi har skapt en vranglåssituasjon.

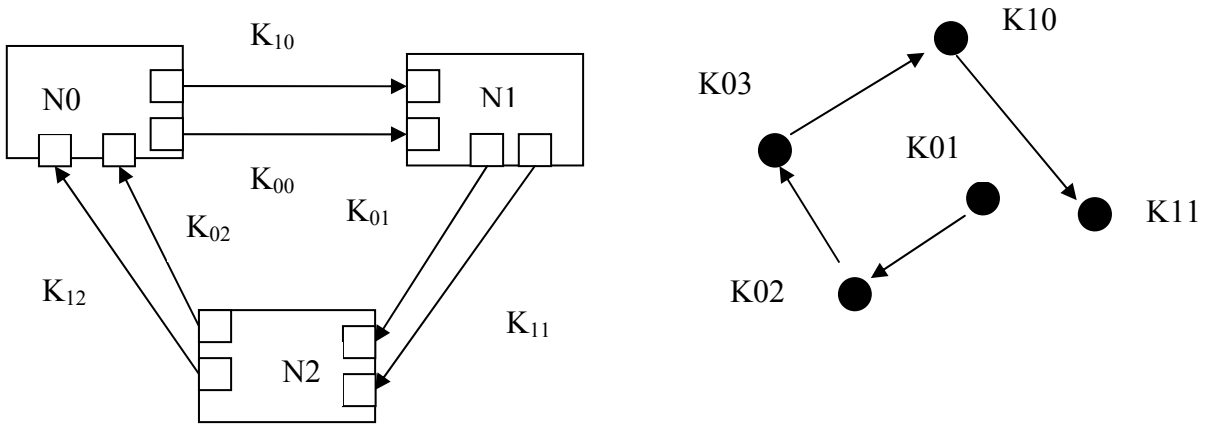
Det er to måter å løse problemet med vranglås; vranglås unngåelse og vranglås rekonvalesering. Siden vranglåsrekonvalesering ikke er viktig for oppgaven, blir det referert til [Duato] for mer kunnskap om dette emnet.

2.5.1 Vranglåsunngåelse

Vi skiller altså mellom vranglåsunngåelse for deterministisk og adaptiv ruting. Eksemplene som er tatt med er basert på eksemplene som gis i boken til [Duato].

2.5.1.1 Eksempel på vranglåsunngåelseteknikk ved bruk av deterministisk ruting

Ved vranglåsunngåelse er hensikten å hindre at vranglås oppstår. [Seitz] foreslo en metode for å løse opp sykler i kanalavhengighetsgrafen ved å bruke virtuelle kanaler. Først prøver en å fjerne kanaler fra kanalavhengighetsgrafen til den blir fri fra sykler, men rutingfunksjonen skal fortsatt være sammenhengende. Hvis en ikke klarer å fjerne syklene på denne måten, kan en legge til et ekstra sett med virtuelle kanaler. Det nye settet av virtuelle kanaler vil også få nye indekser som sikrer rekkefølgen mellom kanalene.



a) Oppløsing av en vranglåssituasjon ved å legge til ekstra virtuelle kanaler.

b) Kanalavhengighetsgraf for a).

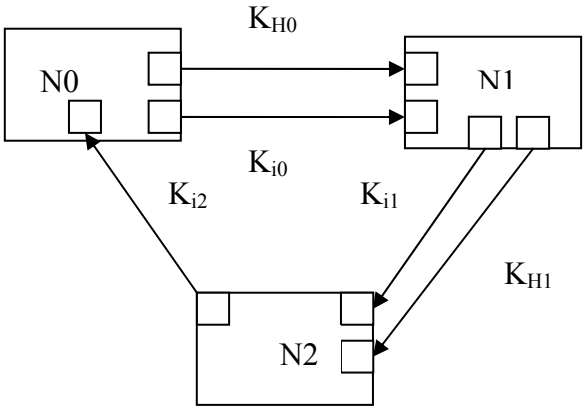
Figur 2.5.2

For å unngå vranglås i figur 2.5.1 kan en bruke metoden som beskrevet ovenfor. En deler opp hver kanal i to ekstra virtuelle kanaler og bruker en ny og begrenset rutingfunksjon; hvis en pakke har kommet frem til destinasjonsnoden, lagres pakken. Ellers brukes kanalene K_{0i} hvis destinasjonsnoden har lavere id enn noden pakken er på. Er nodeid større, brukes K_{1i} kanalene. Figur 2.5.2(b) viser kanalavhengighetsgrafen for figur 2.5.2 a), og vi ser at den ikke har noen sykler; altså kan det ikke oppstå vranglåssituasjoner. Dette kan forklares på følgende måte; Verken K_{00} eller K_{12} blir brukt; har en pakke kommet til N0, har alle andre nodene større id og

dermed kan vi ikke bruke K_{00} kanalen. Derfor er den altså ikke med i kanalavhengighetsgrafen. K_{12} er heller ikke med i kanalavhengighetsgrafen da eventuelle pakker som har kommet til N2, og skal videre, må på grunn av rutingfunksjonen bruke K_{02} kanalen.

2.5.1.2 Eksempel på vranglåsunngåelseteknikk ved bruk av adaptiv ruting

Som vi har sett tidligere, kan en adaptiv rutingalgoritme legge om stien hvis en kanal er blokkert. Dette kan vi utnytte for å løse opp vranglåsen i figur 2.4.1. Nøkkelen er å begrense rutingfunksjonen på en slik måte at vi har en rømningskanal. Et eksempel vil illustrere hva en rømningskanal er.



Figur 2.5.3:
Vranglåsunngåelse ved å bruke adaptiv ruting.

Rutingfunksjonen for figur 2.5.3 er som følger; har en kommet frem til destinasjonsnoden, lagres pakken. Ellers brukes kanalene K_{Hi} dersom inneværende node har en større id enn destinasjonsnoden. K_{ii} kanalene kan brukes for alle kanaler hvor inneværende node ikke er lik destinasjonsnoden. Av figuren ser vi at det må bli sykler i avhengighetsgrafen til K_{ii} – kanalene. Men, allikevel kan ingen vranglåssituasjoner oppstå.

For å forklare situasjonen, tar vi med følgende; en pakke som skal sendes fra N0 til N2 vil alltid komme frem da N2 har større id enn alle andre nodene, og dermed kan en ikke bruke K_{i2} etter N2. K_{Hi}- kanalene vil dermed være fri for pakker. Vi kan ta med et eksempel; anta at det er pakker i K_{H1} – kanalen. Disse pakkene må være ment for N2. De kan avansere da pakkene ikke kan bli låst uendelig ved å vente på bufferplass på N2. Det betyr at K_{H1} er fri for pakker. Dette kan vi si er den første grunnen til at det ikke kan oppstå vranglås.

Av begrensningene i rutingfunksjonen ser vi også at ingen av nodene kan sende pakker til seg selv. Dette er den andre grunnen til at det ikke kan oppstå vranglås (N0 kan ikke sende til seg selv).

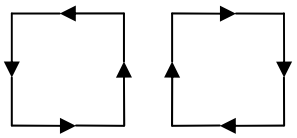
Dermed kan K_{ii} – kanalene bruke K_{Hi} kanalene for å unngå vranglås. Et eksempel vil illustrere det nærmere; la oss anta at det sitter pakker i bufferet til $k_{i1.}$ Pakkene må da enten være på vei til N2. Skulle det sitte pakker i K_{i2} ville de kunne avansere. Anta at det sitter en pakker i K_{i2} og at de venter på å bruke $K_{i0.}$ Ifølge rutingfunksjonen kan de, istedenfor å vente, bruke K_{H0} kanalen.

Siden det er utenfor denne oppgaven å gå noe mer inn på rømningskanaler, henvises det til [Duato].

2.5.2 Turnmodellen

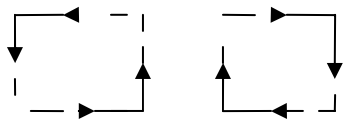
I tillegg til bruk av ekstra virtuelle kanaler er turnmodellen et alternativ til å unngå vranglås. Turnmodellen ble foreslått av Glass og Ni [Glass]. Som forklart, oppstår vranglås når det er sykler i avhengighetsgrafen til kanalene. Turnmodellen forsøker å forhindre så få svinger som mulig for å unngå vranglås.

For å vise hvordan turnmodellen fungerer, kan vi ta med et eksempel.



Figur 2.5.4: Svinger som kan tas i en 2D mesh

Av figur 2.5.4 a) ser vi alle svinger som finnes i en 2D mesh. Noen rutingalgoritmer hindrer noen av svingene for å unngå vranglås. XY-rutingalgoritmen er et eksempel på en slik algoritme. Den forbyr fire av svingene i en 2D mesh. Hvilke svinger vises i figur 2.5.5.



Figur 2.5.5: Forbudte svinger i xy – ruting. De stiplede linjene indikerer forbudte svinger.

En annen rutingalgoritme er vest-først rutingalgoritmen som tillater flere svinger enn XY, og samtidig hindrer vranglås. Som navnet tilsier er alle svingene mot vest ulovlige. Hvis en skal rute mot vest må en først gjøre det for deretter å rute de andre retningene. Vest–først algoritmen kan være både minimal og ikke–minimal.

2.6 Eksempler på rutingalgoritmer

Bortsett fra de rutingalgoritmene som er nevnt, blir det her tatt med ytterligere noen eksempler på rutingalgoritmer. Det blir gitt eksempler på rutingalgoritmer for både regulære og irregulære topologier.

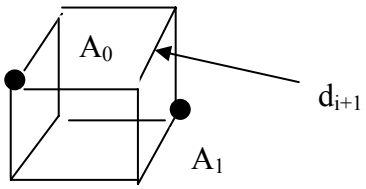
2.6.1 Rutingalgoritmer for regulære topologier

Det vil her bli tatt med noe eksempler på rutingalgoritmer for regulære topologier.

2.6.1.1 Planar-Adaptiv ruting

Tanken bak rutingalgoritmen er å rute i plan for å komme til destinasjonsnoden. Planar-adaptive ruting ble foreslått av [Chien].

Noen topologier, som en kube, er delt opp i flere plan. Et plan er satt sammen av to dimensjoner som vi kan kalle d_i og d_{i+1}. Eksempelvis, vil da et plan tilsvare en side i kuben. To plan, som vi kan kalle p og p+1, deler dimensjon d_{i+1}.



Figur 2.6.1: planar-adaptiv ruting i 3D nettverk

Algoritmen fungerer som følger; anta at vi har en kildenode og en destinasjonsnode i kuben (vist i figur 2.6.1). For å rute fra kildenoden til destinasjonsnoden vil da pakken først blir rutet i plan A_0 for så å rutes i plan A_1. Pakken kan adaptivt bruke alle stiene i hvert plan. Hvis d_i sin avstand er redusert til null mens en ruter i plan A_0 kan en skifte til å rute i neste plan (a_{i+1}). Hvis avstanden i d_{i+1} er redusert til null mens en ruter i plan A_0, vil neste plan ikke gi noen adaptivitet, og en kan hoppe over det.

Hvert plan består altså av to dimensjoner. Hvis vi antar at vi har med en mesh å gjøre, trenger vi en virtuell kanal i den første dimensjonen og to i den siste for å unngå vranglås. Hvis vi tar det settet som krysser en dimensjon i nettverket, kan vi dele det i to virtuelle nettverk. Det ene virtuelle nettverket har kanaler som går i positiv retning mens det andre har kanaler som går i negativ retning (i en gitt dimensjon). Grunnen til at vranglås unngås, er todelt: for det første krysses planene i økende rekkefølge og for det andre er det ingen forbindelse mellom de to nettverkene.

2.6.1.2 P-Kube

P-kube er en rutingalgoritme basert på turnmodellen [Glass] og det er en rutingalgoritme for hyperkube. Nodene i en hyperkube kan adresseres med et binært tall med n siffer, der hvert siffer indikerer posisjonen i hver dimensjon [Theiss 02]. Videre har vi et sett, S, som inneholder nummeret på alle dimensjonene hvor kildenoden og destinasjonsnoden er forskjellig. S deles videre i to sett; S_0 og S_1. Det første settet inneholder alle dimensjoner der kildenoden er 0 og destinasjonsnoden er 1. Sett S_1 inneholder alle dimensjoner der kildenoden er 1 og destinasjonsnoden er 1.

Pakkene rutes i to faser. I den første fasen rutes pakkene gjennom dimensjonene i sett S_0 og i den andre fasen i sett S_1. Er det første settet tomt, kan pakkene rutes direkte i sett S_1.

P-kube ruting unngår vranglås på følgende måte; for at det skal kunne oppstå en vranglås må det eksistere en kanal fra en node til en node med en lavere verdi. For eksempel kan det ikke oppstå vranglås når en kun ruter i S_0 da ingen av kanalene går fra en node med høyere id til en node med lavere id. På samme måte kan det heller ikke oppstå vranglås ved å rute kun i det andre settet. Siden algoritmen også forbyr svinger fra S_1 til S_0 unngår den at vranglås oppstår.

2.6.1.3 Hop-algoritmene

[Gopal] foreslo noen algoritmer kallet hop – algoritmene. De er basert på at bufferne deles opp i klasser. Vi har to hop – algoritmer; positiv - og negativ hop algoritme.

Positiv-hop algoritmen fungerer som følger: idet en pakke injiseres i nettverket, legges den automatisk i buffer 0. Når den hopper til neste node, forflytter den seg til et buffer med et høyere nummer. Når en pakke har avlagt en avstand a vil altså pakken befinne seg i klasse a. På denne

måten hindrer en altså vranglås.

Negativ-hop algoritmen tillater hopp mellom buffer av samme klasse. Nettverket blir først partisjonert i deler slik at en del ikke inneholder to nabonoder. Delene er nummererte fra 0 til d – 1. Hvis en pakke hopper til en node med et lavere id – nummer, vil den rykke opp en klasse ellers hopper den til en buffer i samme klassen.
Siden et hopp til en node med lavere id medfører at pakken hopper til en buffer av en klasse med høyere nummer, vil ikke sykler kunne oppstå.

Negativ-hop algoritmen bruker halvparten av bufferne som positiv hop algoritmen bruker. Enda en forbedring av hop – algoritmene kom i [Boppana]. Der foreslås det å bruke såkalte bonuskort.

2.6.1.4 Dynamisk algoritme

Den neste algoritmen vi kan ta med er den dynamiske algoritme [Dally]. Den er basert på konseptet om dimension reversal. Det er en av to algoritmer hvorav den dynamiske algoritme. Dimension reversal er et nummer som holder orden på hvor mange ganger en pakke er rutet til en kanal i en lavere dimensjon. Den tillater sykliske avhengigheter mellom kanaler, men ikke at pakker venter på en syklisk måte.

De virtuelle kanalene i en link deles i to klasser – adaptiv og deterministisk. Først brukes de adaptive kanalene. Mens pakken blir rutet i de adaptive kanalene, er det ingen begrensninger på antall DR en pakke kan foreta. Når en pakke vil foreta en transisjon fra en kanal til en annen kanal og alle kanalene er blokkert av pakker, sjekkes det om noen av pakkene har samme DR nummer – eller lavere. Er alle kanalene merket med DR-nummer likt eller lavere, går pakken over til å bli rutet deterministisk. Den kan da ikke rutes adaptivt igjen.

Den dynamiske algoritme hindrer altså vranglås ved å hindre at en pakke venter på en utkanal med et lavere eller samme DR merke som pakken selv har.

Begge algoritmene, basert på DR er adaptive.

2.6.1.5 Deflection ruting

Til slutt kan vi ta med en rutingalgoritme som er laget for å unngå vranglås i pakke – og VCT svitsjede nettverk; deflection ruting. Rutingalgoritmen fungerer på følgende måte: en node injiserer en pakke i nettverket. Er det flere ledige kanaler, brukes den kanalen som er en del av en minimal sti; ellers blir pakken misrutet (den bruker en kanal som ikke er en del av en minimal sti).

På hver node er det en minneport. Hvis da en node injiserer en pakke, og alle utkanaler er fulle, vil pakken legges I minneporten sammen med eventuelle innkommende pakker. Når da en utkanal blir ledig, vil først alle pakkene som har ankommet noden sendes på utkanalen før nye pakker injiseres.

Ulemper med deflection ruting, er at rutingteknikken ikke kan brukes for wormhole svitsjet nettverk. Det er på grunn av at hele pakken må lagres på noden ved blokkering av utkanaler. At også andre stier enn minimale stier brukes, kan medføre i tillegg til at en kan risikere at latency øker.

Livelock vil si at en pakke ikke kommer seg inntil en node på grunn av at alle kanalene inntil noden er blokkert av andre pakker.

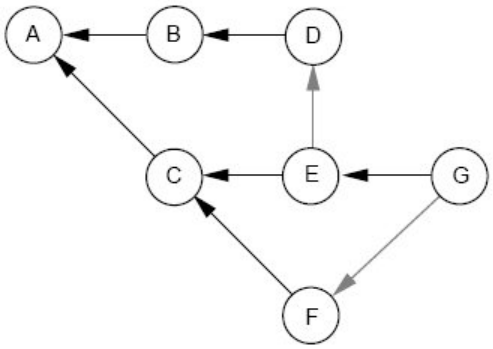
2.6.2 Rutingalgoritmer for irregulære topologier

Det har blitt utviklet flere rutingalgoritmer for irregulære topologier. Et eksempel på en algoritme er Up*/Down* - rutingalgoritmen. Men, Up*/Down*-algoritmen kan også brukes for regulære topologier.

2.6.2.1 Up*/Down*-rutingalgoritmen

Siktemålet til Up*/Down* - algoritmen er å gjøre alle linkene til up og down linker. Det gjør den ved å gi linker up og down retninger på en slik måte at ingen sykler kan oppstå. Etter at alle linkene er gitt retning, får vi en Up*/Down* - graf.

Et eksempel på en Up*/Down* - graf vises i figur 2.6.2.



Figur 2.6.2: Eksempel på en up*/down* - graf [Theiss].

Av figur 2.6.1 ser vi at det er piler mellom nodene og at pilene har en retning. En måte å gi retninger på, er å finne et såkalt spanning tre. Det vil si at vi finner det treet som omfavner alle nodene. Vi ser et spanningtre i figur 2.6.1. Vi ser av figur 2.6.1 at pilene i grafen har to farger; grå og svarte. De grå linkene er de linkene som ikke tilhører spanning treet, mens de sorte pilene tilhører et spanning tre fra node A. Node A er det vi kaller rotnode og alle up-retningene er satt mot rotnoden.

For å forklare litt om retningene i treet, kan vi ta med følgende: vi ser at mellom node G og E går det en pil fra G til E. Grunnen til at pilen har den retningen, er at node G er lengre borte fra rotnoden enn E. Det samme gjelder fra G til F. Et annet eksempler er pilen mellom D og E; på grunn av at node D og E er like langt vekke, ble det valgt å bruke alfabetet til å bestemme hvilken vei pilen skulle ha. Resultatet ble, som vi ser, fra E til D.

[Theiss]: "En konsistent Up*/Down* - graf er en graf med retning som verken har sykler av up - eller down-linker, roten kan nå alle andre noder ved å bruke down-linker, og alle noder kan nå roten gjennom up-linker."

Reglene som gjelder er at pakkene kan traversere alle retninger unntatt down->up. Altså tillates det transisjoner fra up->down, up->up og down->down. På denne måten forhindrer vi vranglås ved at vi unngår sykler. For å ta med et eksempel er figur 2.6.1 tatt med. I figuren er node A rotnode. Hvis en pakke skal fra node F til node B må den først sendes til node C så til A, før den kan sendes til B.

Som et eksempel på anvendelse av denne algoritmen kan vi ta med FRoots. FRoots er basert på Up*/Down*-ruting. Det vil si at den kun kan brukes med denne rutingalgoritmen. Mer om FRoots i kapittel 3.

2.7 Svitsjing

En melding som blir rutet av rutingalgoritmer fra en kildenode til en destinasjonsnode må i de fleste tilfeller traversere flere mellomliggende noder før den kommer til destinasjonsnoden. Hvilke noder meldingen skal traversere bestemmes altså av rutingalgoritmen.

Svitsjeteknikken setter opp forbindelse mellom innbuffer og utbuffer i tillegg til å bestemme når sendingen over forbindelsen skal foregå. Vi har flere typer svitsjeteknikker; pakkesvitsjing, linjesvitsjing, virtual cut-through svitsjing og wormhole svitsjing. Andre svitsjeteknikker som finnes, er mad-postman svitsjing og noen hybride svitsjeteknikker som buffret wormhole svitsjing, "pipelined" linjesvitsjing, "scouting" svitsjing.

Tidligere i dette kapittelet ble det sagt at et interconnection nettverk kan være cut-through rutet. Som en innledning vil først noen enklere rutingteknikker bli forklart, før cut-through ruting forklares. Vi har to typer cut-through ruting; virtual cut-through og wormhole svitsjing.

2.7.1 Linjesvitsjing

Ved linjesvitsjing blir en probe sendt fra kildenode til destinasjonsnode, for å sette opp forbindelse. Mens proben reiser gjennom nettverket, reserverer den ressurser. Når den har kommet frem til destinasjonsnoden, sender den tilbake et bekreftelsesignal på at den har kommet frem og at sendingen kan begynne. Kildenoden sender pakkene den skal sende, og etterpå blir forbindelsen tatt ned.

Fordelen med linjesvitsjing er at hele båndbredden kan utnyttes da ingen andre kan sende på linjen. En ulempe er hvis proben skulle bli blokkert på sin vei mot destinasjonsnoden. Linjen frem til blokkeringen vil da bli forhindret å bruke av andre sendere.

Linjesvitsjing kan med fordel brukes i de tilfeller der det tar kort tid å sett opp stien, mens det tar lang tid å sende selve meldingene.

Linjesvitsjing har blant annet blitt brukt i JPL Mark III binær hyperkube [Duato].

2.7.2 Pakkesvitsjing

En løsning på problemet med linjesvitjsing kan være å dele opp meldingene i pakker og buffre pakkene på ruterne mens en venter på at en link skal bli frigitt. Denne teknikken kalles pakkesvitsjing.

Ved pakkesvitsjing sender en hele pakker gjennom nettverket, og hver pakke blir rutet individuelt. Hver ruter henter ut pakkehodet fra hver pakke (pakkehodet er de første bytes i pakken), og bruker informasjonen til å bestemme hvilken kanal pakken skal sendes ut på. Denne svitsjeteknikken blir også kallet "store-and-forwarding (SAF)" fordi den lagrer pakken på hver ruter før den blir sendt videre.

Pakkesvitsjing er fordelaktig når pakkene er korte og forekommer ofte. Flere pakker som hører til den samme meldingen kan være i nettverket samtidig selv om den første pakken ikke har ankommet destinasjonsnoden. Ulempen er at det tar prosesseringskraft å dele oppe en melding i pakker i tillegg til at pakken må rutes på hver mellomliggende node.

Eksempler på hvor denne svitsjeteknikken er blitt brukt, er Intel iPSC/1 [Intel] og Cosmic Cube [Cosmic].

2.7.3 Virtual cut-through svitsjing

Tanken bak virtual cut-through svitsjing er at en ruter ikke trenger hele pakken for å kunne ta en rutingavgjørelse og sende pakken eller deler av pakken videre. Pakkehodet inneholder den rutinginformasjonen som trengs for å starte videresending av pakken. En pakke blir delt opp i flits, og ruteren trenger da kun de første flitsene for å kunne ta rutingavgjørelsen. Ruteren kan altså sende pakken videre før den har mottatt hele pakken, noe som er en fordel hvis en har en pakke som er større enn båndbredden til linken den skal sendes over. Pakken vil da bruke flere sykler for å komme over linken.

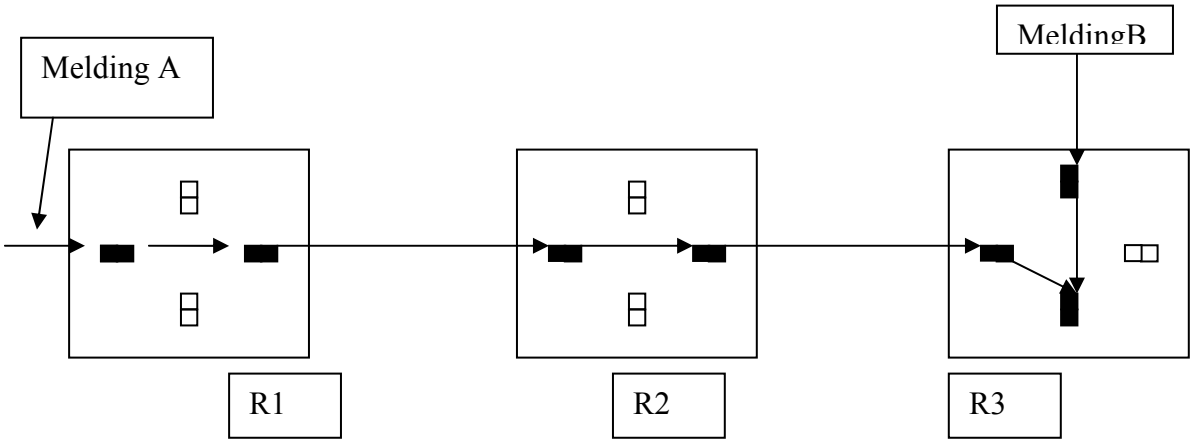
En ulempe med VCT-svitsjing viser seg ved høyt antall pakker i nettverket. Siden det er slik at når flits blir blokkert på en node blir hele pakken lagret på den noden, vil VCT, ved høyt antall pakker i nettverket, oppføre seg som pakkesvitsjing.

VCT – svitsjing brukes i simulatoren brukt under simuleringene i denne hovedoppgaven.

2.7.4 Wormhole svitsjing

I likhet med en virtual cut-through svitsjing blir pakken også her delt opp i flits. Vanligvis er bufferene i en ruter stor nok til å lagre noen få flits. Pakken blir sendt gjennom nettverket som flits.

Forskjellen i forhold til virtual cut-through svitsjing, er at hvis fliten med rutinginformasjon blir blokkert på en node, blir hele pakken blokkert på stedet. Fordi bufferne ikke er store nok til å holde en hel pakke, kan en blokkert pakke okkupere buffere på flere rutere. Illustrasjon ser vi i figur 2.7.1. Med hensyn på denne figuren; en melding blir sendt gjennom ruterne 1, 2 og 3. På ruter 3 krever melding en utkanal som blir brukt av melding B og meldingen blir dermed blokkert på stedet og alle flitsene tilhørende pakken blir stående på stedet.



Figur 2.7.1: Pakke som blir blokkert når en bruker wormhole Svitsjing. Vi ser at flitsene blokkeres på stedet. Tegningen er hentet fra [Duato].

En ulempe med wormhole svitsjing er altså at en blokkert melding kan skape avhengigheter over flere buffere. En fordel er at vi kan ha små buffere som fører til at vi kan få et enklere design av svitsjer.

2.8 Rutingalgoritmer og feiltoleranse

I interconnection nettverk, som i andre typer nettverk, kan det oppstå feil. Til tross for det oppstår feil, er det viktig at pakkene som blir rutet gjennom nettverket kommer frem til destinasjonsnodene. En feil i et nettverk kan for eksempel føre til at en rutingalgoritme, uansett adaptivitet, mister sin evne til vranglåsfrihet. Eksempler på feil som kan oppstå i et nettverk er en nodefeil i det systemet startes opp. En nodefeil kan føre til at nettverket blir usammenhengende, men det er utenfor rammene til denne oppgaven å betrakte denne typen tilfeller. Istedenfor antas det at topologien er sammenhengende etter at noder har feilet.

Før noen eksempler på feiltolerante rutingalgoritmer blir gitt, vil feilmodellen bli forklart.

2.8.1 Feilmodell

Feil i et nettverk kan ta mange former; det kan eksempelvis være linkfeil, nodefeil eller at noen prøver å avlytte nettverket. Denne oppgaven vil kun ta for seg nodefeil.

En nodefeil er en feil hvor noden svikter. Ved nodefeil blir også alle linkene på noden merket som at de har feilet. Ved linkfeil blir også de virtuelle kanalene regnet som feil. Vi har blant annet to typer nodefeil: den ene er at noden feiler med det samme systemet startes opp mens den andre er at noden feiler når systemet er oppe og går. Det første kalles for statisk feil, mens det siste kalles dynamisk feil.

Ved statisk feil er rutingtabellene regnet ut med kunnskap om feilen(-e). De vet derfor om dem før trafikken blir sluppet inn i nettverket.

Det er flere måter å håndtere dynamiske feil på. En måte er å bruke rekonfigurering. Vi har to typer rekonfigurering; statisk og dynamisk. Det blir forklart mer om statisk- og dynamisk rekonfigurering i 2.8.3.
Andre metoder som kan brukes ved dynamiske feil, er å bruke adaptive rutingalgoritmer eller backtracking. Ved backtracking kan pakken, hvis den treffer på en node som har feilet, trekke seg tilbake for å prøve å finne en alternativ sti. Adaptive rutingalgoritmer, kan (som tidligere forklart) bruke en annen sti hvis en skulle være blokkert.

Tiden det går før noder svikter i nettverket, måles i Mean Time To Repair (MTTR) og Mean Time Between Failure (MTBF). Eksempelvis kan vi ta med at komponenter i Myrinet har en MTBF som ligger rundt en million timer [Myrinet]. Men, denne tiden vil minske med større nettverk. Eksempelvis kan vi ta med at et nettverk med 32 000 svitsjer og 128 000 linker vil ha en MTBF som ligger på rundt 6 timer. Teknikker som skal gjelde for nett hvor komponentene har høy pålitelighet designes på en annen måte enn et nett der komponentene har lav pålitelighet.

Til slutt kan vi ta med at det er varierende hvor mye informasjon nodene i nettverket inneholder. Dess mer informasjon en node har dess lettere er det å ta mer kvalifiserte rutingavgjørelser. Det kan være at hver node i nettverket har informasjon om alle andre nodene i nettverket eller at de kun har informasjon om nabonodene.

Alt dette innebærer at vi har en måte å samle inn informasjon med jevne mellomrom. En diagnosealgoritme kan brukes til dette formålet. Diagnosealgoritmene er ulike med hensyn på om vi har å gjøre med en statisk eller dynamisk feil. Eksempelvis kan vi ta med at k–nabo diagnose innebærer at hver node har feilinformasjon innenfor en avstand på k-naboer. Mer om dette kan finnes i [Blough], [Wang].

I 2.8.2 vil det bli tatt med noen eksempler på feiltolerante rutingalgoritmer.

2.8.2 Feiltoleranse i nettverk

Det er utviklet flere teknikker for feiltolerant ruting i pakkesvitsjet nettverk. I pakkesvitsjede nettverk, kan ikke pakker hoppe fremover hvis ikke det er ledig bufferplass. Derfor blir vranglåshåndtering et spørsmål om bufferhåndtering.

[Chen] har foreslått en løsning for hyperkuber som går ut på å bruke spare dimensjon. En spare dimensjon er en dimensjon som ikke er en del av korteste stien til destinasjonsnoden, og den blir brukt i tilfelle pakken møter på blokkerte linker. Metoden fungerer som følger; først finner en forskjellen mellom kildenoden og destinasjonsnoden i antall bits. Denne verdien forteller hvor mange dimensjoner pakken vil traversere for å bruke korteste sti til destinasjonsnoden. Når en pakke kommer til en mellomliggende node, og alle utlinkene som er del av korteste sti til destinasjonsnoden er blokkert, kan pakken bruke spare dimensjonen for å komme seg videre. Pakken inneholder en koordinatsekvens som forteller hvilken dimensjon pakken kan sendes over. Dimensjonen som brukes blir så slettet fra sekvensen.

[Lee] foreslo en metode som baseres på at mer feilinformasjon er tilgjengelig på hver node. Metoden som bruker en spare dimensjon, forutsetter at nodene kun har lokal informasjon om nettverket. Metoden til [Lee] forutsetter global informasjon. Metoden går ut på å merke noder som nonfaulty, unsafe og faulty. Hver node har to lister; en liste over feilede noder i nærheten og en annen for unsafe noder. Hver node sjekker tilstanden til omliggende noder og forandrer status etter hvilken tilstand de har. Har en node to feilede naboer (eller flere) endrer den status til unsafe. Dette vil til slutt medføre rektangulære regioner av noder som er merket faulty eller unsafe. Rutingalgoritmen prøver først å rute til ikke-feilede noder før den ruter til noder som er merket som unsafe. En unsafe node innebærer en potensiell fare for at pakken kan kjøre seg fast hvis den blir rutet til den.

Til slutt kan vi ta med en løsning for wormhole svitja nettverk. Den er basert på at feilregionen er rektangulær. Vranglåshåndteringen er vanskeligere for wormhole svitsja nettverk, da en blokkert pakke kan blokkere buffere på flere noder.

[Chien] foreslår å bruke planar–adaptive ruting. Eksempelvis kan vi ta med en 2D mesh hvor e-cube ruting brukes. La oss anta at vi har en kvadratisk feilregion som dekker noder på midten av meshen, og at vi har en kildenode og en destinasjonsnode på hver side av denne feilregionen. La oss videre anta at feilregionen ikke ligger helt ut til kanten av meshen. Altså at det er noder rundt hele feilregionen (som ikke er en del av feilregionen). Stien til en eventuell pakke vil da gå forbi eller rundt regionen. Dette impliserer at vi, hvis kildenode og destinasjonsnode ligger slik til, må rute fra vertikal - til horisontal retning. Dette er ikke gunstig da det kan medføre sykliske avhengigheter mellom kanaler i de to retningene. Som et forsøk på å unngå vranglås, ble planar – adaptiv ruting foreslått.

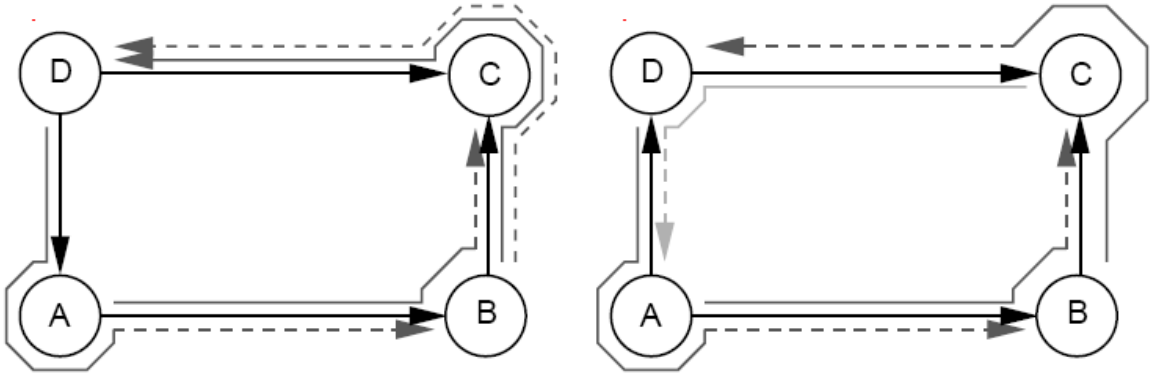
Ved å legge til en ekstra virtuell kanal i vertikal retning, kan en partisjonere opp meshen i to virtuelle nettverk hvor vi traverserer x-retning i henholdsvis i positiv – og negativ retning. På denne måten unngår en vranglås.

2.8.3 Rekonfigurering

I noen situasjoner er det slik at grunnlaget for å bruke en rutingfunksjon er borte. Det kan være mange årsaker til det og blant annet kan det være at topologien har endret seg. Eksempelvis hvis en node feiler mens systemet er oppe og kjører, bør en gå over til å bruke en ny rutingfunksjon.

Rekonfigurering vil si å gå over fra å bruke en rutingfunksjon i nettverket til å bruke en annen. Vi har to typer rekonfigurering som kalles statisk og dynamisk rekonfigurering. Begge typene kan brukes til å unngå Ghost-avhengighet.

Ghost avhengigheter vil si at gamle pakker gjør ulovlige vendinger i forhold til den nye rutingfunksjonen eller omvendt og Ghost-avhengighet kan føre til vranglås. Figur 2.8.1 illustrerer situasjonen.



a) Gammel Up*/Down* - ruting b) Ny Up*/Down* - ruting

2.8.1 Nettverk før og etter en feil (og etter rekonfigurering) har skjedd.

Et eksempel illusterer ghost – avhengighet som fører til vranglås; bilde 2.8.1 a) illusterer et nettverk som sender pakker. Det er kilderutet Up*/Down* ruting som brukes. Det har ennå ikke skjedd noen feil. Plutselig skjer en feil ut i nettverket, og det skjer en rekonfigurering. 2.8.1 b) viser nettverket etter at feilen har skjedd. Vi ser at linken mellom D og A har skiftet retning.

Vi ser at i a), er det flere pakker i nettverket, og alle er rutet etter den gamle rutingfunksjonen. Eksempelvis er det to pakker som skal fra node B til node D. Den ene pakken får komme frem til destinasjonsnoden. Problemet oppstår når den andre pakken skal ut på linken C-D; en annen pakke får nemlig lov til å bruke linken. Den nye pakken er rutet etter den nye rutingfunksjonen. Det fører til at den nye pakken må foreta en sving på node D som ikke er tillatt etter den gamle

rutingfunksjonen. Grunnen er at det ikke er tillatt med en sving fra down-up. Den vil altså okkupere bufferet på node D slik at den andre pakken fra B til D, blir låst på node C. Det neste vi observerer er at pakken fra D-B, som er rutet etter den gamle rutingfunksjonen, må foreta en sving på node A som ikke er lov etter den nye rutingfunksjonen (ikke tillatt med down-up). Dette medfører at pakken vil okkupere et buffer på node A. Når vi da har en pakke som venter på utlinken på node B (den er rutet etter den gamle rutingfunksjonen), har vi fått en vranglåssituasjon.
Dette er et eksempel på at ”Ghost”-avhengighet kan føre til vranglås.

2.8.3.1 Dynamisk rekonfigurering

Dynamisk rekonfigurering vil si at vi forandrer til en ny rutingfunksjon mens nettverket er oppe og kjører. Hovedutfordringen med dynamisk rekonfigurering ligger i å hindre vranglås ved å holde begrensninger på pakkeinjeksjoner og sending av pakker på et minimum [Lysne]. FRoots er en dynamisk rutingalgoritme. Et eksempel på en annen dynamisk rekonfigureringsmetode, er ”Simple Deadlock – Free Dynamic network reconfiguration”.

2.8.3.2 Statisk Rekonfigurering

Statisk rekonfigurering vil si at vi venter til alle pakkene er ute av nettverket før vi sender inn nye.

2.8.3.3 Andre rekonfigureringsmetoder

Double Scheme er en dynamisk rekonfigureringsmetode [Pang]. Metoden har flere sett med virtuelle kanaler, og kan nekte pakker adgang til et kanalsett mens den tømmer det for pakker. Det andre kanalsettet er i full bruk mens dette skjer. Vi har flere typer Double Schemes som Basic Double Scheme, Fully Adaptiv Double Scheme og Optimized Fully Adaptiv Double Scheme. Basic Double Scheme bruker kun to sett av virtuelle kanaler. Det ene settet brukes av den gamle rutingfunksjonen og det andre av den nye. Bortsett fra under rekonfigurering, brukes kun et av settene. Det andre settet tømmes for pakker og rekonfigureres. Dette skjemaet kan kun tilby delvis adaptiv ruting i beste tilfellet. En forbedring er Fully Adaptive Double Scheme. Det bruker tre sett av kanaler; to sett kun for rømningsruting for å unngå vranglås, mens det siste brukes for fullt adaptiv ruting.

Et annet eksempel er Simple Dynamic Network reconfiguration[Lysne]. Det er en metode som bruker et token for å skille pakker som bruker den gamle og den nye rutingfunksjonen på hver link.

Et problem med begge disse to rekonfigureringsmetodene er at de ikke kan garantere pakker stier under rekonfigureringen.

2.9 Oppsummering

I dette kapittelet har vi forklart en del begreper og sett på rutingalgoritmer og problemer knyttet til rutingalgoritmer. Vi har også sett på topologier, som ble beskrevet som det mediumet rutingalgoritmer ruter i. I tillegg til å ha sett på teknikker for å sende pakker videre gjennom svitsjene. Til slutt tok vi med noen eksempler på hvordan noen rutingalgoritmer oppfører seg ved tilstedeværelsen av feil i et nettverk og forklarte rekonfigureringsteknikker.

Kap.3 Siktemålet og algoritmene

Innledningsvis vil problemområdet og siktemålet til oppgaven bli beskrevet. Deretter blir algoritmene gjennomgått. I kapittel 4 gjennomgås forklares det hvilken forskningsmetode som er brukt for å analysere resultatene i tillegg til det som har blitt implementert i forbindelse med oppgaven.

3.1 Problemområdet

Interconnection nettverk stiller svært høye krav til throughput og latency. En mulig måte å øke throughput på og også minske latency, kan være å bruke korteste stier. Problemet med å bruke korteste stier ser vi når vranglås oppstår. De fleste rutingalgoritmer er laget slik at de har rutingrestriksjoner for å unngå vranglås. Det vil igjen ofte medføre at en ikke kan bruke korteste stier.

En annen måte å øke throughput på er å bruke adaptive rutingalgoritmer. Som sagt tidligere, kan en adaptiv rutingalgoritme legge om stien hvis en link er okkupert av en annen pakke. Det vil være med på å øke throughput. Problemet med å bruke adaptiv ruting, er at pakkene fra en melding kan ta forskjellige veier gjennom nettverket. Dette kan medføre at en pakke kommer før frem til kildenodene enn en annen pakke fra samme meldingen. På denne måten kan en ikke garantere at pakkene fra en melding blir levert i samme rekkefølgen som de forlot kildenoden. At pakkene ikke blir levert i rekkefølge medfører at destinasjonsnoden må bruke tid og prosesseringskraft på å sette sammen pakkene til rett rekkefølge – med fare for at latencyen øker.

FRoots er en adaptiv rutingalgoritme, mens Dimension-Order er en deterministisk rutingalgoritme.

3.2 Oppgavens siktemål

I oppgaven skal vi foreta en ytelseanalyse av FRoots og Dimension-Order.

Min generelle oppfatning i forkant av simuleringen, var at FRoots ville klart ha de beste egenskapene i forhold til Dimension-Order på grunn av dens kompleksitet.

Vi kan dele oppgaven inn i tre:

1. Algoritmene sin throughput (gjennomstrømming) og latency skal sammenlignes, både ved null feil og en feil.
2. Se nærmere hvordan Dimension-Order utvikler seg ved å kjøre den med flere virtuelle lag. I utgangspunktet blir den kjørt med ett lag, mens FRoots kjører med syv virtuelle lag.
3. Se nærmere hvordan FRoots utvikler seg ved flere feil i nettverket.

3.3 Algoritmene

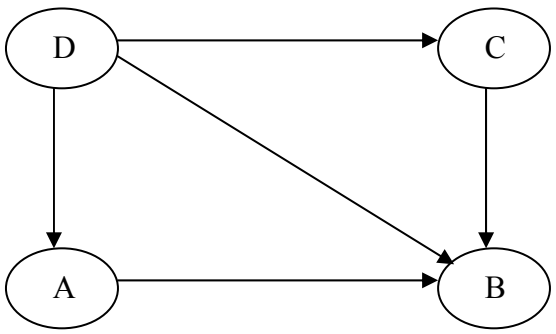
I oppgaven er det statiske feil som blir behandlet, men algoritmene er også i stand til å håndtere dynamiske feil. Algoritmene som forklares her, antar at nettverket fortsatt er forbundet etter et antall feil.

3.3.1 FRoots

3.3.1.1 Introduksjon

FRoots er en dynamisk rekonfigureringsmetode. Metoden bruker virtuelle lag og den kan kun brukes sammen med Up*/Down*-algoritmen. Hvis vi antar at en node feiler mens nettverket kjører, snakker vi altså om en dynamisk feil. Ved dynamiske feil, foretas det en rekonfigurering av nettverket. FRoots kan sikre pakker som ikke er ødelagte, en lovlig sti til destinasjonsnoden under rekonfigureringen. Det gjør den ved å bruke stiredundansen i bi-sammenhengende irregulære nettverk.

Dess høyere konnektivitet en topologi har, dess flere linker er det på hver node. Dette kan bety at det er flere stier mellom to noder i topologien. Eksempelvis kan vi ta med figur 3.1.



Figur 3.1: figuren viser en biforbundet topologi.

Figuren viser det vi kaller en biforbundet topologi. Det vil si en topologi som har to stier mellom alle nodepar. Topologier med flere forbindelser er viktig for feiltoleranse. Eksempelvis kan en

node i nettverket feile og vi har fortsatt en sti mellom to noder hvis vi har en biforbundet topologi.

Definisjon En k-sammenhengende topologi er en topologi som forblir sammenhengende selv etter at det har skjedd k-1 feil. Dette gjelder uansett hvilke noder eller linker som feiler [Theiss].

Definisjon En bisammenhengende konfigurasjon er en konfigurasjon til en k–sammenhengende topologi hvor k>o, og en rutingalgoritme, som forblir forbundet og sammenhengende etter hvilken som helst feil [Theiss].

Av det som er forklart ovenfor kan vi dra neste konklusjon:

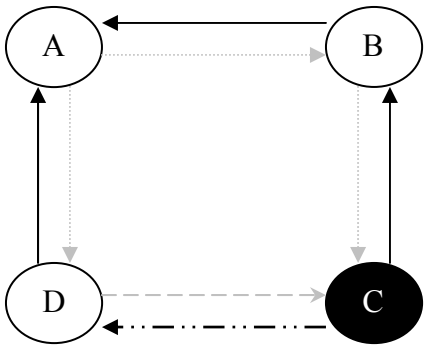
Definisjon En k-sammenhengende topologi har k ulike stier mellom alle kilde–destinasjonspar[whitney].

To stier er ulike hvis de kun har kildenode og destinasjonsnode til felles.

Et problem med dette er hvis vi har en rutingalgoritme som fungerer slik at selv om vi har to stier mellom et nodepar, så kan allikevel den ene stien ikke brukes. Et eksempel kan være at vi bruker minimal Up*/Down* ruting i figur 3.1. Skal vi da sende en pakke fra node A til node C, er det to stier å bruke, men allikevel bare en som er lovlig (dette altså på grunn av rutingrestriksjonene algoritmen setter). Den lovlige stien er AB, BC.

3.3.1.2 Virkemåten til FRoots

FRoots utnytter altså stiredundansen til topologier som er bisammenhengende. Her illustreres det hvordan.



Figur 3.2: FRoots med to lag.

Hvis en bruker to lag istedenfor et, ser vi (av figur 3.2) at vi sikrer at det er to lovlige ruter mellom alle kilde–destinasjonspar. Dette innbefatter at en bruker Up*/Down* ruting. Skal vi sende en pakke fra D-B, kan vi sende den enten via stien D-C-B (altså det lyse laget) eller D-A-B (det mørke laget).

Løvnoder er noder som kun har utgående up - linker. Av figur 3.2, ser vi at node C er en løvnode og node A er rotnode i laget med sorte piler. En løvnode har ingen bypass trafikk - trafikk som går gjennom noden. Noder som ikke har slik trafikk, kan fjernes fra nettverket uten problemer. Ingen trafikk skal verken til eller fra en feilet node (trafikk til noden må kastes uansett). Det laget hvor en node er løvnode kalles sikkerhetslaget til noden.

Froots fungerer på følgende måte; anta at node C, i figur 3.2, feiler. Hvis D skal sende en pakke til B, kan den bruke sikkerhetslaget til node C. Pakken vil dermed følge denne stien; D-A-B.

Kjernen i FRoots er å gjøre alle noder til løvnoder, i minst et virtuelt lag.

Definisjon: et virtuelt lag er det laget hvor en node er løvnode.

FRoots bruker denne algoritmen til å lage løvnoder:

Ta en kopi, K, av hele nettverkstopologien
Lag et sett, ikke-løv, som, i starten, består av alle nodene i K

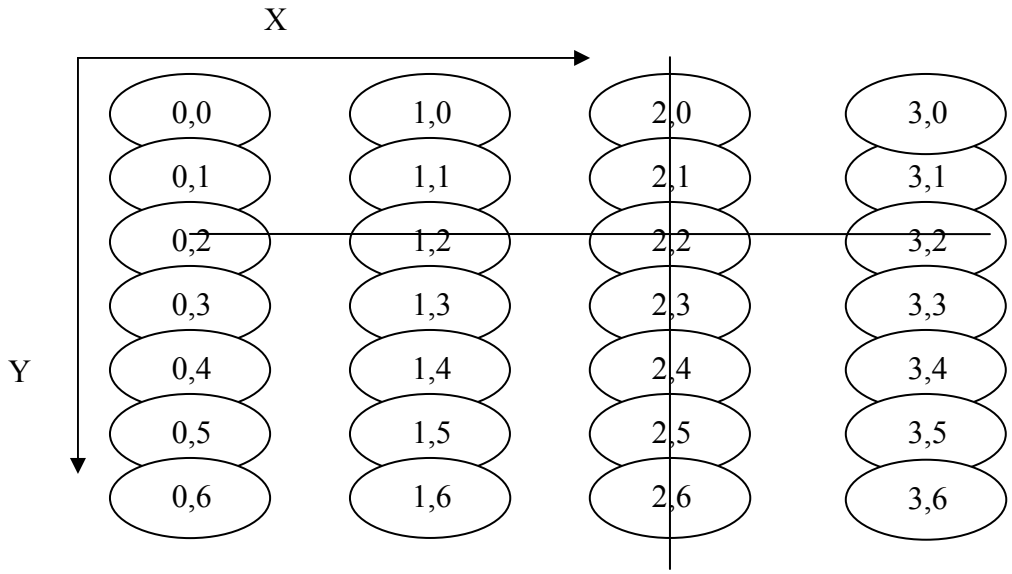
While ikke-løv ikke er tom
Finn et tidligere ubrukt lag, l , i nettverket
Velg en node n fra ikke-løv som laget l sin garanterte løvnode
Fjern node n fra K
Lag et tomt sett, løvkandidater
Flytt n fra ikke-løv til løvkandidater
Finn artikulasjonspunktene til K
For hver node d i ikke-løv hvor d ikke er en umiddelbar nabo
Til noen noder i løvkandidater og d heller ikke er et artikulasjonspunkt til noen noder i løvkandidater
fjern d fra K.
Finn nye artikulasjonspunkt til K
Lag en ny Up/Down* graf til K, men se bort fra danglende linker*
(linker som har vært tilknyttet noder i løvkandidater)
Danglende linker settes til å ha Up-retning
Fra nodene i løvkandidater til nodene i K
Legg tilbake alle nodene i løvkandidater tilbake til K
Kopier linkretningene til K til lag l

Figur 3.3: Pseudokode for å lage UpDown - graf i K
Koden er hentet fra [Theiss]

Ved å gjøre det, kan en hvilken som helst node i nettverket feile og vi kan bruke sikkerhetslaget til denne noden til å nå alle andre noder i nettverket.

3.3.2 Dimension order ruting

Dimension order rutingalgoritmen er skreddersydd for bruk i en 2D mesh. I en 2D mesh er nodene ordnet på en slik måte at avstanden mellom kildenode og destinasjonsnode lett kan regnes ut som avstanden i x - og y – retning.



Figur 3.4: 2D mesh. Krysset er tatt med for å vise hva som skjer når en node feiler. I eksempelet vil det være node 2,2 som har feilet. Dette resulterer i at alle nodene på korset blir disablet som kildenoder og destinasjonsnoder.

For å illustrere hvordan algoritmen fungerer, kan vi ta med et eksempel. Anta at vi skal rute mellom node 1,2 og 3,5 i figur 3.4. En må først regne ut avstanden i x-retning for så å finne avstanden i y-retning, som i dette tilfellet blir x = 2 og y = 3. Pakken må altså sendes to hopp i x-retning og 3 hopp i y-retning. Dette er tilfellet hvor ingen noder har feilet.

Dimension-Order er optimalisert med virtuelle lag, men et virtuelt lag er ikke det samme som hos FRoots. Istedenfor tilsvarer et virtuelt lag en virtuell kanal. Kjører en da med tre virtuelle lag, er det tre virtuelle kanaler å velge mellom når en skal sende pakker. Hvilket virtuelt lag en sender pakker på blir valgt tilfeldig for hver pakke, og det tas ingen hensyn til at et buffer på en kanal er fullt.

3.3.2.1 Metode for feiltoleranse

For å gjøre algoritmen feiltolerant ble en metode brukt som er nedskrevet i forbindelse med IBM sin Blue Gene/L. Metoden som er brukt er en av flere metoder en kan bruke for å gjøre Dimension-Order feiltolerant.

For å illustrere hvordan det hele fungerer, kan vi ta med et eksempel; anta at node 2,2 har feilet (i figur 3.4). Alle nodene som ligger på korset, i figur 3.1, blir disablet som kildenoder og destinasjonsnoder. Skal pakken sendes fra kildenode 1,2 til destinasjonsnode 3,5, vil altså pakken ikke komme frem til målet.

Det er flere noder som må disables. Hvis vi antar at node 2,2 har feilet, kan ikke nodene under

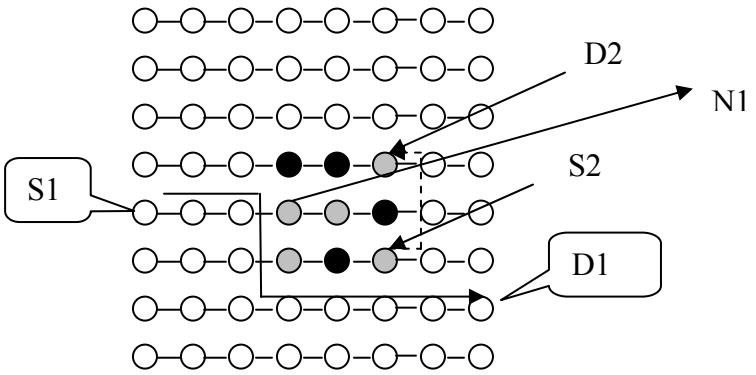
2,2 være destinasjonsnoder for 0,0, 1,0, 0,1, 1,1 , 0,2, 1,2, 3,0, 3,1, 3,2, 2,0, 2,1.

3.3.3 Fault–Tolerant Routing Using Unsafe Nodes

Fault–tolerant Routing Using Unsafe Nodes er en metode som ble foreslått av [**Lee**]. Metoden innbefatter at nodene i nettverket blir merket med unsafe, safe, faulty eller non-faulty, og at hver node i nettverket holder oversikt over sine naboer. I tillegg antar metoden at det brukes en metode for vranglåshåndtering. Metoden ruter pakker rundt feilområde.

Som en forkortelse for denne oppgaven, er metoden sitt navn satt til Fruun. Det presiseres at det er forfatter av denne oppgaven som har gitt metoden dette navnet.

Er en node nabo til to eller flere unsafe eller feilede noder, blir den satt til unsafe.



Figur 3.5: Stier som Fruun kan ta i et nettverk [Duato].

Når rutingalgoritmen skal rute en melding mellom en kildenode og en destinasjonsnode, prøver den først å rute meldingen til en ikke-feilet node som ligger på korteste stien til destinasjonsnoden. Eksisterer ikke en slik node, vil den prøve om det eksisterer en unsafe node før meldingen blir rutet til en hvilken som helst ikke–feilet nabo. En ikke–feilet nabo er garantert å eksistere så lenge det ikke er flere enn n/2 feilede noder i nettverket [Duato]. Figur 3.5 viser eksempler på to stier som pakker kan ta i et nettverk når en bruker Fruun.

3.4 Oppsummering

Vi har i kapittel 3 slått fast siktemålet til oppgaven og forklart hvilke algoritmer som skal sammenlignes. I tillegg har vi også forklart virkemåten til Fruun. Fruun var den rutingalgoritmen som vi ikke klarte å implementere.

Kap. 4 Forskningsmetode

Ved å simulere kan en sammenligne ytelsen til de to rutingalgoritmene FRoots og Dimension-Order. Simulering er en av tre metoder for å evaluere ytelse. De to andre metodene er analytisk modellering og målinger på et virkelig system. Ifølge [Carson] er simulering mest nyttig å bruke i tre situasjoner; når systemkomponenter kan defineres og en kan modellere en interaksjon mellom disse, når det er vanskelig eller umulig å forutsi effektene av de foreslåtte forandringer eller at det ikke finnes noen analytisk modell som er nøyaktig nok til analysere situasjonen. Valget har falt på simulering som metode for å evaluere ytelsen til algoritmene opp mot hverandre på bakgrunn av at det ikke finnes noe virkelig system eller analytisk modell som var anvendelig til dette formålet.

”Etter at modellen er utviklet må en bestemme hvor mange av startobservasjonene som må kastes for å sikre at simulatoren har nådd en stabil tilstand og hvor lenge en må kjøre simuleringen. Disse emnene kalles transient removal og stoppkriteriet.” [Jain].

Simulatoren er basert på diskrete hendelser. Det betyr at en hendelse skjer ved en gitt tid. Hvis vi tenker oss at vi har tidsaksen oppdelt i enheter, så skjer hendelsen ved en tidsenhet.

Simulatoren som er brukt, er bygd opp i simuleringsomgivelsen J-Sim.

Først en kort innføring i J-Sim og deretter vil et oversyn over modellen gis. Til slutt tas det med et implementasjonskapittel som forteller noe omhva forfatter har implementert.

Sentrale deler av simulatoren blir forklart for å gi lesere bedre forståelse av hva som foregår. All forklaring om simulatoren er hentet direkte fra koden.

4.1 Simuleringsomgivelse

J-Sim er en komponentbasert simuleringsomgivelse [J-Sim]. Basisentitene er altså komponenter og de er autonome. Oppførselen til de blir bestemt av en kontrakt og hver kan individuelt testes, implementeres, designes etc.

To lag med klasser er programmert på J-Sim. Den klassen, som vi kan si er det mer generelle laget, heter conan og har til hensikt å gjøre det lettere å implementere mer spesifikke teknologier. To eksempler på teknologier som kan implementeres er Advanced Switch Interconnect (ASI)[PCI] og Infiniband[inf]. Det er ASI som brukes i denne oppgaven.

Modellen støtter inntil 16 virtuelle kanaler; 8 kanaler OVC og 8 BVC (OVC står for Ordered Virtual Channel mens BVC står for Bypass Capable Virtual Channel). Hvor mange virtuelle kanaler en ønsker å kjøre med, kan bestemmes ved parametersetting. Dette er ASI-spesifikt og en annen teknologi bruker kanskje en annen måte.

Algoritmene beskrevet i kapittel 3, er implementert i en simulator. Simulatoren består av en

simulatordriver som er diskrete hendelsesdrevet og med flere nettverkselementer som genererer hendelser. Disse hendelsene er lagt i en kø, for til rett tid å bli kjørt av driveren. Simulatorkoden er skrevet i Java.

4.2 Simulatordriveren

Hver hendelse i simuleringen, som generering av pakker eller ankomst av pakker over en link, lagres i en lagringsstruktur med et tidsstempel for når hendelsen skal kjøres. Kjernen går gjennom lagringsstrukturen og kjører hendelsene. Når det ikke er flere hendelser å kjøre for en tidsenhet, skiftes den til neste tidsenhet og slik fortsetter det til lagringsstrukturen er tom.

Et problem med simulatoren oppstår når to hendelser blir kjørt samtidig. I et virkelig nettverk er det flere prosesserende enheter som genererer hendelser samtidig. Simulatoren kjøres på en maskin med kun en prosessor, så det er umulig å kjøre to hendelser samtidig. En løsning på problemet er å legge hendelsene som skal skje samtidig i en kø, og kjøre disse i tilfeldig rekkefølge. Det er denne løsningen som er valgt implementert.

4.3 Oppstart av simulatoren

Simuleringen startes opp på følgende måte: en kaller på den algoritmen en vil kjøre og sender med de rette parameterne. Det første som skjer er at topologifilen lages og leses inn. Topologifilen forteller hvordan nettverket skal se ut og hvordan nodene er forbundet til hverandre via linker – altså til hvilke porter endenodene er forbundet til svitsjene og hvilke porter to svitsjer kommuniserer via. I tillegg forteller topologifilen om antall noder som skal være i nettverket. Rutingen initieres og svitsjene og endenodene dannes. Disse enhetene bindes sammen via portene. Eksempelvis brukes en J-Sim metode som heter connect () til å forbinde to gitte porter til hverandre. De opprettede komponentene blir lagt til i kjøretid etter at simuleringen er initiert.

Simulatoren inneholder flere trafikkgeneratorer. Vi kan ta med at skal vi kjøre med uniformt trafikkmønster, bruker vi en generator laget spesielt for uniformt trafikkmønster til å finne en destinasjon for pakken.

I tillegg inneholder simulatoren kode for alle komponenter som er nødvendig for å simulere et nettverk som svitsjer, linker og buffere. Til hver svitsj er det knyttet to endenoder og det er endenodene som genererer pakkene som sendes ut i nettverket. Hvor mange pakker endenodene skal generere kan en styre ved hjelp av parameter som bestemmer hvor lang tid det skal gå mellom hver pakke som skal genereres. Simuleringsparameterne brukt ved kjøring av simuleringene, vil bli diskutert nærmere i kapittel 4.4.

4.4 Enhetene i simulatoren

4.4.1 Svitsjene

Svitsjene sender pakkene videre til destinasjonsnoden (en endenode) eller til neste svitsj.
De består av flere deler og svitsjeteknikken som brukes er virtual cut-through svitsjing.

4.4.1.1 Oppbygning av svitsjen

I svitsjen finner vi svitsjekjernen og portene til svitsjen. Kjernen består av scheduler, arbitration manager, crossbar og buffere. Kjernen har en lagringsstruktur som inneholder alle portene den kan sende til/fra. Hver innlink og utlink har dedikert aksess til crossbaren og den er et svitsjenettverk med N innganger og M utganger hvor N=M. Når crossbaren er ledig kan en sende via den til utlinken.

Hvilke buffer som skal sende, bestemmes av komponenten som heter arbitration manager. Rekkefølgen er longest wait round robin. Det vil si at hendelsene utføres slik at den som har ventet lengst blir behandlet først osv. Arbitration manageren sender forespørsler til en scheduler. En forespørsel tilsvarer en pakke som skal sendes gjennom svitsjen og forespørslene legges i en lagringsstruktur i scheduler. Scheduler velger hvilken port som skal sende og sjekker om det er plass til pakken i utbufferen. Etter det blir crossbaren sjekket for å finne ut om den er åpen, før hendelsen utføres.
Hvis flere meldinger etterspør den samme utlinken, velger altså arbitration manager hvem som skal få benytte utlinken.

Køene til svitsjene praktiserer FIFO (First-In-First-Out).

Som et eksempel kan vi ta med hva som skjer når svitsje kjernen blir satt til å våkne; den vil da sjekke køene til hver port sine virtuelle kanaler om det er noe å gjøre, finne ut hvilken kø som skal først behandles, og sende pakkeenheter gjennom kjernen. Når svitsjekjernen finner ut at det ligger en pakke i en av innportene sine virtuelle kanaler, blir den ”merket” slik at det går an å velge den som har ligget lengst.

4.4.1.1.1 Svitsjeporten

Svitsjeportene inneholder blant annet metoder for å legge pakker i køen til en gitt virtuell kanal eller fjerne pakker.

Hver port har en referanse til arbitration manageren.

4.4.2 Endenodene

Endenodene står altså for generering av pakker. I hver endenode er det en metode som lager en

ny pakke og legger den ut i en kø. Hvis køen er full, kastes pakken. Mellom hver pakke som skal genereres brukes en generator til å finne ut tidspunktet for når neste pakke skal genereres. Verdien sendes inn til en lagringsstruktur.

Så lenge simuleringen varer, vil endenodene generere pakker. En simuleringsparameter bestemmer når de skal slutte.

Endenodene starter også opp en analyse – klasse som blant annet sørger for skriving av verdier til en resultatfil. For hver simulering lages det en resultatfil. Denne filen inneholder all informasjon om simuleringen som alle parameterne og hvilke parametere som brukes. I tillegg inneholder filen simuleringsresultatene. Skriving til resultatfilen skjer ved bestemte mellomrom som en kan justere etter ønske. Eksempelvis kan en skrive ut ei linje i resultatfilen for hver 100 sykel.

Latency regnes ut på følgende måte; for hver gang endenoden får inn en pakke, sjekkes det først ut om pakken har kommet til riktig destinasjon. Har den det, regnes latency for den aktuelle pakken ut: pakken bærer med seg et nummer som forteller når pakken ble lagt i sendekøen på kildenoden. Dette nummeret kan vi kalle a. En finner deretter tiden pakken ankom destinasjonsnoden, som vi kan kalle b.

$$\text{Latency} = b - a$$

Deretter oppdaterer en antall pakker som har kommet frem til denne noden. Det siste som skjer er at gjennomsnittslatency til alle pakker som har kommet frem til til rett destinasjonsnode (for hele systemet), blir funnet. Det er denne verdien som er interessant.

Antall rutingfeil finnes ved at hver node holder telling på antall pakker som de har fått og som de ikke skulle hatt. Resultatet som skrives ut er totalt antall rutingfeil i nettverket i et gitt tidspunkt.

Throughput finner vi ved at en for hver simulering tar gjennomsnittet av totalt antall pakker som har kommet frem til nodene.

Eksempelvis kan vi ta med at hvis vi har satt utskriften til at den skal skje for hver 100 sykel, blir altså både gjennomsnittslatency, antall rutingfeil totalt og total throughput for systemet opptil det aktuelle tidspunktet, skrevet ut.

Til slutt kan vi ta med at før endenodene skriver et resultat til resultatfilen, sjekker de om systemet har stabilisert seg. Det gjøres ved å bruke regresjonsanalyse (mer om regresjonsanalyse senere i dette kapittelet). Etter det sjekkes det om det har gått så lang tid i simuleringen at det er siste rapportlinje, og dermed også samlet resultat for hele simuleringen som skal skrives til resultatfilen. Hvis det ikke er det blir det altså bare skrevet et resultat til resultatfilen for den gjeldende tiden. Er det den siste rapporten, skrives det siste resultatet ut til resultatfilen som inneholder hvor mange pakker som totalt er generert av endenodene, totaltantall pakker kommet

frem til destinasjonen og gjennomsnittlig latency. Det er siste rapport–linjen som er viktig i hver resultatfil og som brukes til å lage grafene som vises i kapittel 5.

4.5 Kredittbasert flytkontroll

Simulatoren bruker kredittbasert flytkontroll. For å forklare hva som menes med det, kan vi først ta med at en kreditt forstås som 64 bytes. Hvis da en pakke er 512 bytes tilsvarer det 8 kreditt.

Hvis en pakke skal sendes over en link, blir det først sjekket om bufferen på egress–linken har nok kreditt til å ta imot pakken. Har bufferen, som skal ta imot pakken, ikke nok ledig kreditt, kan heller ikke pakken sendes til denne bufferen. For å realisere dette er det til hver buffer tilknyttet variabler som henholdsvis holder styr på størrelsen til bufferen totalt (i kreditt), antall forbrukte kreditt til nå og antall kreditt den kan ta imot.

I tillegg kan vi ta med at kreditt også brukes til å styre sendingen av pakker. En node kan ha lov til å sende så og så mye kreditt før en annen må sende. På denne måten kan vi for eksempel sikre at node i nettverket ikke må vente uendelig lenge før de får anledning til å sende.

4.6 Diverse simuleringsparametere

4.6.1 Trafikkmønster

Trafikkmønsteret sier noe om hvordan trafikken flyter i nettverket. I et nettverk kan vi ha mange forskjellige typer trafikkmønster og trafikkmønsteret er avhengig av applikasjonen som kjører. Det er derfor viktig å kjøre simulering med flest mulig trafikkmønstre. Det er flere typer trafikkmønstre implementert i simulatoren som uniformt, parvis og hot spot.

Hot–spot trafikkmønster er et trafikkmønster som er passende å bruke hvis det er sannsynlig at det meste av trafikken er rettet mot en node i nettverket slik at noden blir en flaskehals.

4.6.2 Trafikkgenerering

Hensikten med en simulator er å få den så nært opp mot virkeligheten som mulig. Dette gjelder også for trafikken som blir generert i nettverket. Trafikken skal være mest mulig lik trafikken som oppstår i et virkelig nettverk og for å realisere dette kan en for eksempel bruke matematisk modellering. Men, en kunne også brukt såkalte tracer. Et eksempel på en matematisk modellering for å simulere virkelig nettverkstrafikk, er Poisson – fordelingen. Poisson - fordelingen [Emstad] kan forklares på denne måten; en prosess som genererer kun en type hendelser av gangen og aldri mer enn en hendelse av gangen, kalles en regulær punktprosess. Hvis vi antar at en regulær punktprosess er konstant og uavhengig av forrige hendelse, så har vi en Poisson - prosess.

4.6.3 Trafikklast

Hvor mye trafikk hver endenode skal genere bestemmes av en parameter. Parameteren bestemmer hvor lenge hver endenode skal vente mellom hver pakke som genereres. Trafikkgeneratoren bruker parameteren til å regne ut hvor lang tid det skal ta mellom hver enkelt pakke. Denne verdien er variert under simuleringene.
Hensikten med simuleringene er å finne hvor mye trafikk hver algoritme klarer å ta unna. En må derfor kjøre med forskjellig belastninger på nettverket. Belastningene forandres ved å endre på meanverdien. Eksempelvis kan mean – verdien settes til 98.5 for en kjøring og standardavviket til 9.85, altså 10 % av meanverdien.

Vi bruker normaltilnærmelsen til Poissonfordelingen.

4.6.4 Topologi

Når en skal kjøre simuleringer med rutingalgoritmer, må en bruke en nettverkstopologi som rutingalgoritmen kan operere i. I oppgaven blir det brukt en regulær topologi, og topologien som er valgt er en 2D mesh. Det blir kjørt med et rutemønster lignende figur 2.3.1. Det er ingen rundtkoblings(wraparound) - linker i topologien. Det blir kjørt med to størrelser på nettverket og det er 4x4 og 8x8.

4.6.5 Buffere

Størrelsen på bufferne kan varieres, men siden det er virtual cut – through svitsjing som brukes, må bufferne være store nok til å holde minst en pakke. Bufferne er satt til å kunne holde 3 pakker.

4.6.6 Tiden i simuleringen

Tiden blir målt i sykler. En sykel er en abstrakt tidsenhet.

For å finne throughput bruker vi følgende formel:

Formel 1: Throughput(Gbps) = antall pakker mottatt x pakkestørrelsen i bits
antall sykler x lengden på hver sykel i ns (nanosekund)

Resultatene kan overføres til ulike linkhastigheter hvis en bruker forskjellige lengder på en sykel. Formelen som brukes er:

Formel 2: Linkhastighet = Pakkestørrelsen i bits
overføringshastigheten x sykellengden(ns)

Overføringshastigheten sier noe om antall bits som kan overføres over linken per sykel. Eksempelvis kan det være at overføringshastigheten er 128 bytes. Dette vil medføre at det tar to sykler å overføre en pakke som er 256 byte. I tillegg kan vi regne ut linkhastigheten.

Vi kan ta med to eksempler som illustrerer sykellengden. Begge eksemplene antar at overføringshastigheten er 128 bytes per sykel og at pakkestørrelsen er 256 bytes.
I det første eksempelet er linkhastigheten 32 Gbps. Ifølge formel 2 vil da sykellengden bli 32 ns. Et annet eksempel er hvis linkhastigheten 64 Gbps; ifølge formel 2 vil da sykellengden bli 16 ns.

4.7 Antall simuleringer

Simulering med et sett parametere kjøres 16 ganger for å få resultatet tilstrekkelig statistisk nøyaktig. Siktemålet til simuleringene er å finne knekkpunktet til grafene og hvordan grafene ser ut litt etter knekkpunktet. Et punkt på grafen kan for eksempel tilsvare kjøring med en meanverdi på 55.5 og standardavvik 5.5. En må altså kjøre 16 simuleringer for å få et tilstrekkelig statistisk nøyaktig resultat med disse parameterene. Vil en da at grafen skal bestå av 5 punkter, vil det resultere i 80 kjøringer (5*16). Istedenfor å kjøre en og en simulering, kan en sende en jobb, som kan inneholde et antall simuleringer (eksempelvis da 80), til et kluster. Dette klusteret kalles condorklusteret. Det er det vi har gjort.

Man starter jobber på condorklusteret på følgende måte: en sender en kommando-fil som inneholder alle jobbene en vil kjøre. Resultatet til hver jobb, blir skrevet ut i en dump-fil (resultatfilen). Informasjonen i dump-filen leses ut av et perl – script.

4.8 Metrikker

Throughput ble definert innledningsvis. Det er flere måter å definere throughput på og i vårt tilfelle tas altså gjennomsnittet av antall pakker som har kommet frem til alle nodene i hver simulering. Altså finner vi først gjennomsnittlig antall pakker som har kommet frem til hver endenode i en simulering. Deretter finner vi antall pakker som har kommet frem i hver simulering og deler på antall simuerlinger i en jobb. Som vi husker tilsvarer en jobb 16 simuleringer.

Den andre metrikken som skal brukes, er latency. Latency er den tiden en pakke bruker fra kildenode til destinasjonsnode. For å finne latency finner vi, på samme måte som throughput, først gjennomsnittslatency til alle pakkene under hver simulering for til slutt å ta gjennomsnittslatency til alle simuleringene.

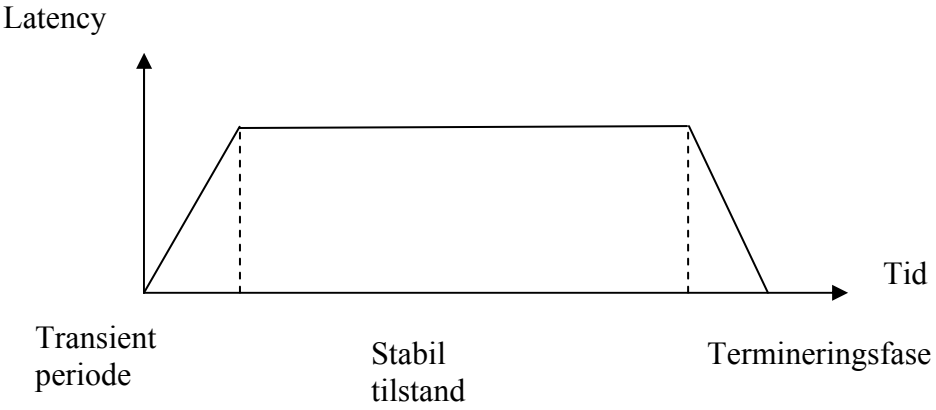
Ytelsen vil hovedsaklig bli målt med throughput som kriterium, men latency blir tatt med for å vise hvordan den blir påvirket når algoritmene begynner å kaste pakker.

4.9 Simuleringens faser

Det er tre hovedfaser i simulatoren (se figur 4.1). Den første fasen er den transiente fasen hvor alle bufferne fyller seg opp fra å være tomme til at det er pakker i dem. Den andre fasen kaller vi den stabile fasen. Det er i denne fasen at vi samler inn dataene. I den stabile fasen er det en representativ fyllingsgrad av pakker i bufferne. Den siste fasen kaller vi termineringsfasen. Da er pakkgenereringen avsluttet, bufferne tømmer seg og systemet kjører til det ikke skjer noen flere hendelser.

Sampleperiode er den perioden som det tas prøver fra nettverket for å skrive til resultatfilen. Lengden på hver simulering regnes som lengden på den transiente perioden + sampleperioden. Målinger i den transiente perioden bør ikke tas med da det kan gi et bedre resultat enn det som ellers vil bli gitt etter at systemet har stabilisert seg.

En utfordring ligger i å finne når den transiente perioden er ferdig. Til det brukes regresjonsanalyse.



Figur 4.1: De tre simuleringsfasene fremstilt ved hjelp av en graf.

4.9.1 Regresjonsanalyse

Regresjonsanalysen som er brukt, er hentet fra [Bhattacharyya].

Det er latency det som måles. Det tas prøver av latency i 10 intervaller og vi finner gjennomsnittslatency for hvert av disse intervallene. Deretter finner vi en linje som representerer punktene på en best mulig måte. Stigningstallet til denne linjen er avgjørende for om vi kan si at systemet er stabilt eller ikke.

Det som er hensikten med analysen er å finne en linje som ikke er stigende. Det gjøres ved å

finne stigningstallet til det vi kaller regresjonslinjen (som er den linjen som best representerer alle punktene). Ligningen til regresjonslinjen er y=ax+b. Dette er et matematisk uttrykk for en rett linje; variabelen a sier noe om hvor bratt linjen stiger – eller synker. Et negativt fortegn foran a, det er en linje som synker. Positivt fortegn det motsatte. Siktemålet blir da å finne når a ikke er større enn null. Det betyr at systemet har stabilisert seg (jfr. Figur 4.1).

Det blir kjørt regresjonsanalyse 100 ganger for å sjekke om systemet har stabilisert seg. Hvis ikke det har skjedd innen den tid, skrives det ut en feilmelding og ingen målinger er brukbare.

4.10 Implementasjonen

I oppgaven er det statiske feil som behandles. Statiske feil håndteres ved å bruke den statiske feilmodellen.

4.10.1 Statisk feilmodell

I den statiske feilmodellen er alle feil i systemet allerede oppstått før systemet starter opp. Topologifilen i FRoots bygges opp med en node mindre i nettverket. I og med at Dimension–Order disabler noder, er alle nodene fortsatt der selv om en node egentlig har feilet.

I FRootsimplementasjonen fungerer den statiske feilmodellen som følger; en sender inn som parameter det antall feil en ønsker å ha i nettverket, for eksempel to feil. Når topologien lages fjernes to noder.

I og med at FRoots og Dimension-Order skal sammenlignes for en feil, og at det da bør være samme node som er feil, måtte en liten justering foretas i FRoots. Det var en ganske enkel justering som gikk ut på endre på feiltopologien slik at en kunne sende inn, som parameter, noden som en ønsket skulle være feil.

Når vi bruker Dimension-Order sender vi med noden som en ønsker skal være feil under kjøringen, eksempelvis kan det være node 10. En løkke går gjennom alle nodene og sjekker om de ligger i en slik avstand/posisjon i forhold til den feilede noden at det er nødvendig for dem å bli disablet. Svitsjene og endenodene blir så opprettet og koblet sammen.

4.10.2 Problemet med Fruun

Vi prøvde å implementere Fruun med flere forskjellige algoritmer; Dimension-Order og vest-først-algoritmen. Nedenfor blir det forsøkt forklart hvorfor det ikke lykkes.

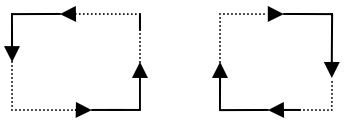
4.10.2.1 Dimension-Order

Den første algoritmen vi prøvde med, var Dimension-Order. For å forklare hva som skjedde, tar

vi i bruk turnmodellen.

Stiene som en pakke skal bruke gjennom nettverket er utregnet på forhånd og kan ikke omlegges underveis. Dessuten; når en pakke er ferdig med å rute i x-retning og har rutet i y – retning, kan ikke pakken rutes tilbake til y – retning, noe som er en alternativ rute i henhold til Fruun.

Dimension–Order er forhindret fra å foreta følgende svinger:

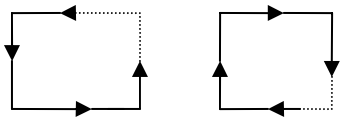


Figur 4.2: De stiplede linjene er forbudte svinger ved bruk av Dimension-Order algoritmen.

Eksempelvis blir da den ene svingen i ruten, fra S2 til D2 (i figur 3.5), umulig å foreta for Dimension–Order algoritmen. Det er på grunn av at det kun lovlig å foreta en sving (figur 4.2). Det andre er at algoritmen kan kjøre seg fast dersom den skulle finne på å ta en sti som går f.eks. via N1. En eventuell pakke vil låse seg hvis den bruker denne stien.

4.10.2.1 Vest-først-algoritmen

Den neste algoritmen vi prøvde var vest-først. Turnmodellen for vest-først vises i figur 4.9.

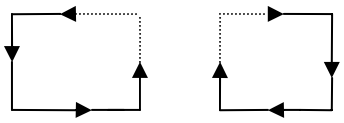


Figur 4.3: De stiplede linjene er forbudte svinger ved bruk av west-first algoritmen.

Det vi kan se (av figur 4.3), er at for å kunne rute i vest retning, må en først gjøre det. Ellers kan en aldri rute i den retningen. Hvis vi skulle gått fra node S2 til node D2 (i figur 3.5), ser vi at vi først måttet rutet mot øst for så å rute mot vest. Dette er ulovlig i forhold til vest-først.

4.10.2.1 North-last-algoritmen

Den tredje algoritmen som ble prøvd ut, var north-last. Algoritmen tillater ikke svinger fra nord mot øst og nord mot vest.



Figur 4.4: De stiplede linjene er forbudte svinger ved bruk av north-last algoritmen.

Ved å sammenligne de svingene som er forbudt i henhold til north-last algoritmen med stien fra S2 til D2, finner vi at det ikke vil gå. Stien krever at vi først foretar en sving fra øst mot nord og deretter fra nord mot vest. Siden nord mot vest er en forbudt sving, vil den altså ikke være brukbar.

4.10.3 Endringer i FRoots implementasjon

En variabel ble opprettet i koden for å holde på hvilken node som skal feile. Denne noden ble hentet ut og sendt over til topologien som skulle opprettes. Eksempelvis kan vi ta med at hvis vi vil fjerne en node fra nettverket, opprettes en topologi med en node mindre og vil vi fjerne to, blir to noder fjernet ved opprettelsen av topologien osv.

Kap.5 Simuleringsresultater

I kapittel 4 ble simulatoren gjennomgått. Det ble forklart nærmere hvordan den er bygd opp og hvilke parametere som er brukt. Dette kapittelet vil ta for seg resultatene en kom frem til etter kjøringene. Før selve presentasjonen av resultatene, blir parameterne brukt under kjøring presentert, i tillegg blir metrikkene brukt for å justere simulatoren forklart.

I kapittel 6 trekkes konklusjonen av resultatene som er presentert i dette kapittelet.

5.1 Simuleringsparametere

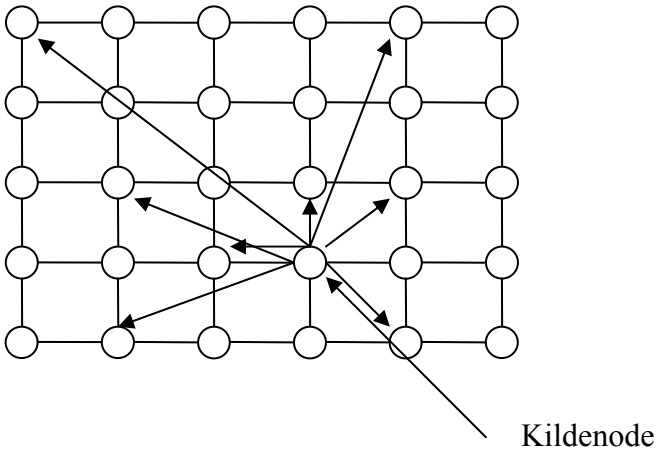
Det er flere parametere som vi anvender under kjøringene

5.1.1 Trafikkmønster

Trafikkmønsterne som ble brukt var uniformt og parvis trafikkmønster.

5.1.1.1 Uniformt trafikkmønster

Det enkleste trafikkmønsteret, er det uniforme trafikkmønsteret. Ved uniformt trafikkmønster har alle nodene like stor sannsynlighet for å bli valgt som destinasjonsnoder. Noden som genererer en pakke, plukker tilfeldig ut destinasjonsnoden.



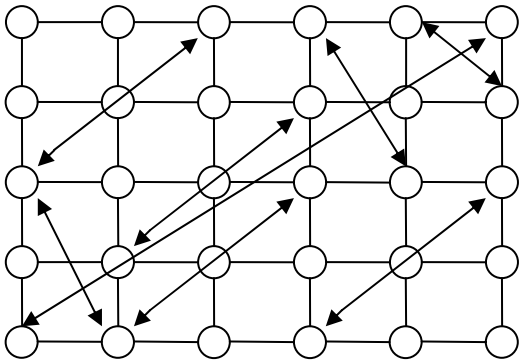
Figur 5.1: Eksempler på destinasjonsnoder ved uniformt trafikkmønster.

Figur 5.1 skal illustrere at node N kan velge ut alle noder som destinasjon for en pakke. For hver pakke blir det valgt ut en ny destinasjonsnode.

5.1.1.2 Parvis trafikkmønster

Parvis trafikkmønster passer til å modellere omgivelser hvor par av noder kommuniserer mye med hverandre.

I det simuleringen starter opp, velger hver node seg ut en tilfeldig partner som de kommuniserer med gjennom hele simuleringen.



Figur 5.2: Eksempler på kommuniserende noder ved parvis trafikkmønster.

Figur 5.2 viser eksempler på nodepar som kan kommunisere gjennom en hel simulering.

5.1.2 Trafikkgeneratoren

I vårt tilfelle er det normaltilnærmelse til Poisson – fordelingen som brukes. Poisson-fordelingen blir mye brukt, men er enkel og gjenspeiler ikke skikkelig virkelig trafikk. Årsaken til det, er at virkelig trafikk er self-similar [Leland]. Det betyr at trafikkmønsteret til en viss grad gjentar seg selv. Ifølge [Willinger] viser virkelig nettverkstrafikk samsvar over et bredt spekter av tidsskalaen, mens de fleste trafikkmodeller viser samsvar over et kort spekter av tidsskalaen. [Willinger] skriver om en modell som kan brukes for å simulere self-similar trafikk. Det henvises til [Willinger] for videre forklaring av denne modellen.

5.1.3 Topologier

I kapittel 2 ble det forklart hva en topologi er og det ble gitt eksempel på ulike topologier. Et av eksemplene var en 2D mesh. Simuleringene er kjørt med både 4x4 mesh og 8x8 mesh. Det vil si at meshen har henholdsvis 16/64 noder og til hver node er altså tilkoblet to endenoder via linker.

5.1.4 Trafikklast

Trafikklasten i nettverket vil si det antall pakker en kjører inn i nettverket under simuleringene (jfr. Kap. 4 om hvordan simuleringene er utført). Knekkpunktet til grafene forteller ved hvilken trafikklast systemet begynner å kaste pakker. Det er derfor viktig å generere lastverdier slik at et stort nok antall pakker blir sendt inn i nettverket. Det er forskjellig lastverdier som benyttes. Først for å finne knekkpunktet for så å generere et stort nok antall pakker slik at en finner ut hvordan grafen oppfører seg rundt, og litt etter, knekkpunktet.

Lastverdiene blir generert av et java – program. En sender inn som parameter starten på og slutten på intervallet der en tror at knekkpunktet vil ligge, i tillegg til at en setter hvor mange parametere en vil ha generert i dette intervallet. Som resultatet får du ut en fil som inneholder lastverdier fint fordelt over det aktuelle intervallet.

5.1.5 Overføringshastighet

Overføringshastigheten er satt til 128 bytes per sykel. Siden pakkestørrelsene er satt til 256 byte, tar det to sykler å overføre en pakke.

5.2 Resultatene

FRoots og Dimension-Order ruting vil først bli testet mot hverandre med en feil og med samme node feilet. FRoots kjøres med sju virtuelle lag. Der er derfor valgt å kjøre Dimension-Order med 1, 3 og 7 lag. Resultatet av dette vil bli vist i samme graf. Deretter vil det bli sett på hvordan FRoots utvikler seg når flere feil oppstår i nettverket

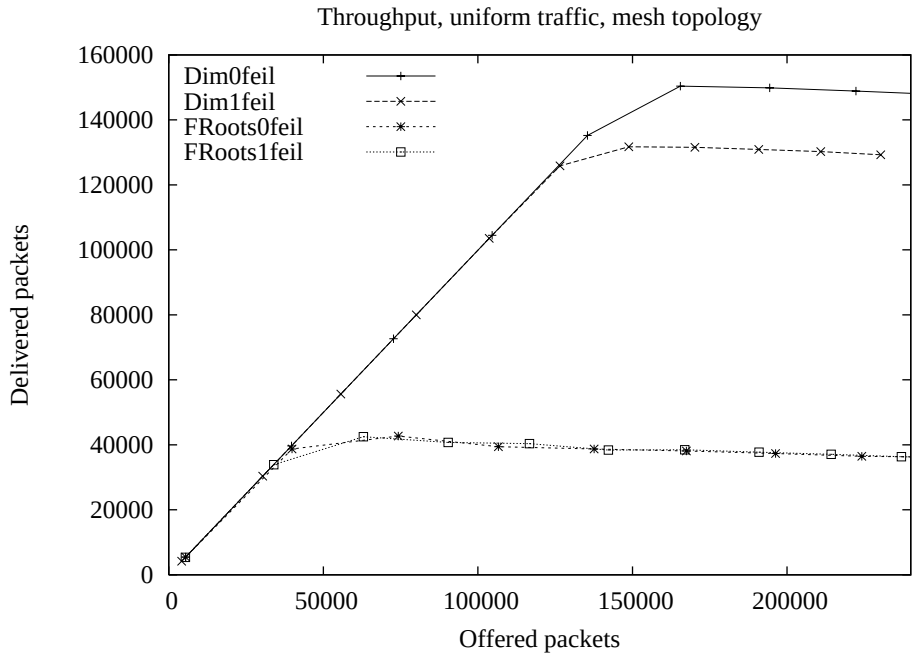
Grafene i denne ytelsesanalysen, er sammensatt av en x-akse og en y-akse. For grafene som viser throughput, har X-aksen tittelen ”offered packets” og den viser gjennomsnittlig antall pakker tilbudt i løpet av alle de 16 simuleringene. Y-aksen har tittelen ”Delivered packets” og den indikerer altså gjennomsnittlig antall pakker som har kommet frem til destinasjonsnodene i nettverket i løpet av de 16 simuleringene. Eksempelvis kan vi ta med at et punkt på en throughput-graf kan indikere at det ble gjennomsnittlig generert 50 000 pakker i de simuleringene og de kom frem gjennomsnittlig 40 000 pakker.

Latency – grafene har en annen betegnelse på y – aksen, nemlig ”clockticks”. Det er ikke tatt med noen referansekurve for disse grafene, men det ideelle er at de går jevnt, i en vannrett linje som går så lavt på y-aksen som mulig. Avvik fra den tenkte ideelle linjen (og da de avvikene som går oppover) er negative, da det tyder på at pakken bruker lengre tid på å komme seg frem til

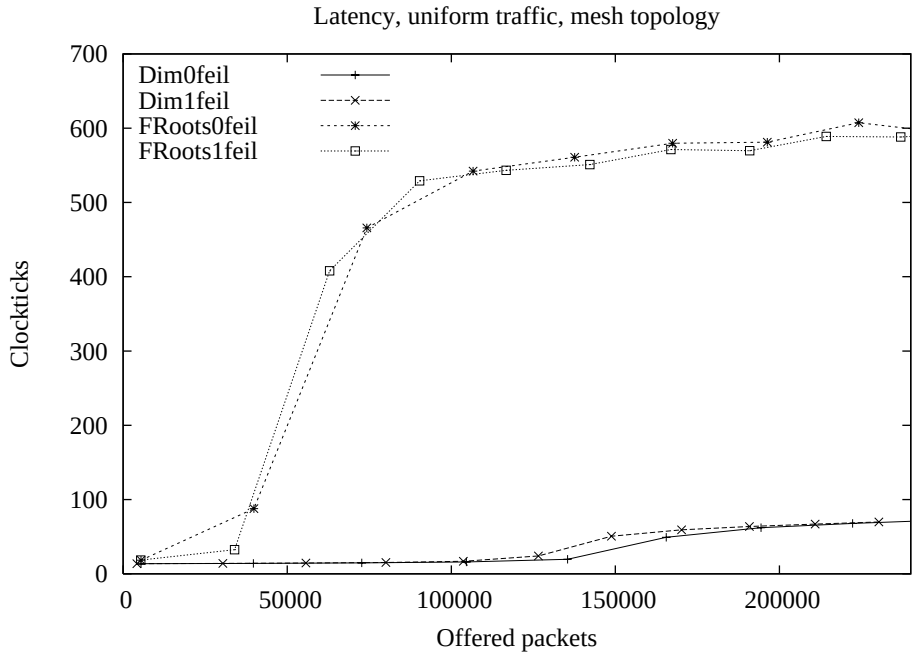
destinasjonsnoden.

Kapittel 4 gav en nærmere forklaring om hvordan en har funnet frem til alle disse verdiene.

Siden det ikke ser ut til å være noen forskjell på hvilke algoritme som er best (når det gjelder størrelsen på nettverket), ble det valgt å kun ta med resultatene for 64 noders nettverk.



Figur 5.4 (a): - Grafen viser throughput for Dimension-Order og FRoots med null – og en feil.

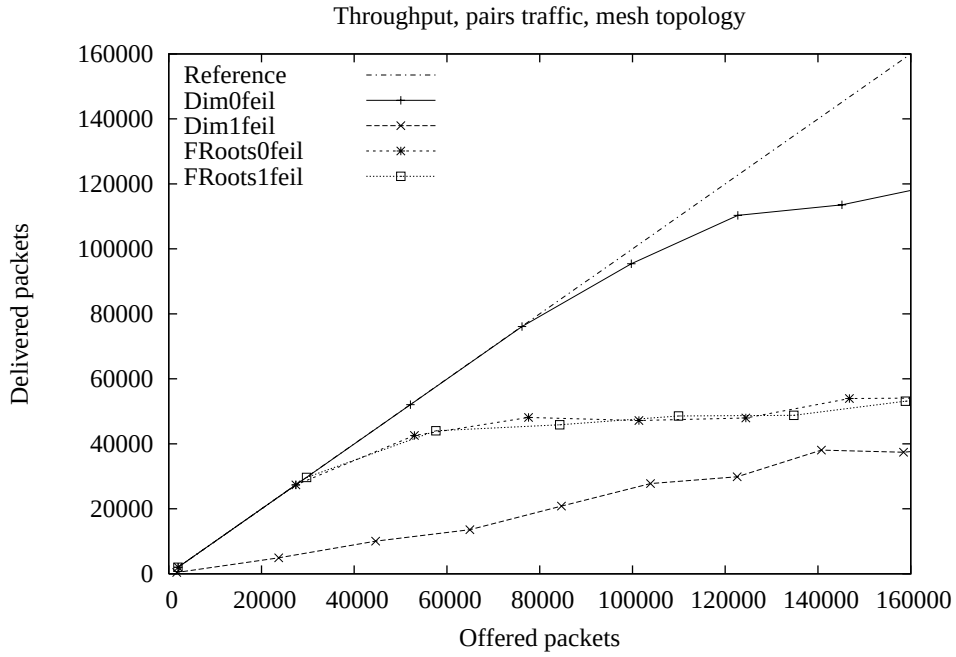


Figur 5.4 (b): Grafen viser latency for Dimension-Order og FRoots med null – og en feil.

I figur 5.4 a) ser vi throughput-grafen for Dimension-Order algoritmen med henholdsvis null og en feil og algoritmen blir kjørt med et virtuelt lag (altså en virtuell kanal). I tillegg viser grafen FRoots med null feil og en feil. 5.4 b) viser latency-grafene for de samme tilfellene som i a).

Vi ser at Dimension-Order har betydelig høyere ytelse enn FRoots når en bruker uniformt trafikkmønster. Grunnen til dette kan være oppbyggingen av algoritmen. Dimension-Order er en algoritme som er av lavere kompleksitet og den gir alltid korteste sti mellom to noder. FRoots med Up*/Down* vil ikke alltid gi korteste stier mellom et nodepar.

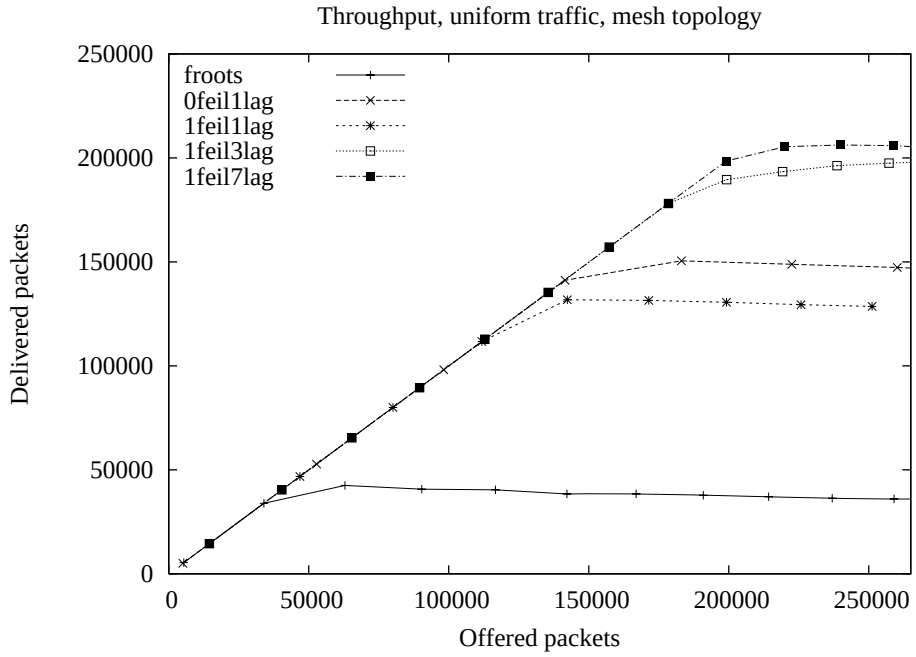
Av latency – figuren ser vi at tiden pakkene bruker, øker kraftig i det systemet begynner å kaste pakker (ved bruk av FRoots). Dimension-Order algoritmen, når den bruker et virtuelt lag, øker litt, men ikke noe i nærheten av det Froots gjør. Hva er grunnen til det? Etter min mening må en se på hvordan algoritmene bruker de virtuelle kanalene. FRoots sender pakker tilfeldig inn på et virtuelt lag og bruker tilfeldige stier. Dermed vil kanskje flere pakker i nettverket, og kanskje pakker fra flere linker, ”krangle” om den samme utkanalen på en node. Dette vil, i det en når punktet hvor systemet begynner å kaste pakker, medføre at pakkene vil måtte vente lenger før de kommer seg videre til destinasjonsnoden og latency vil dermed øke.



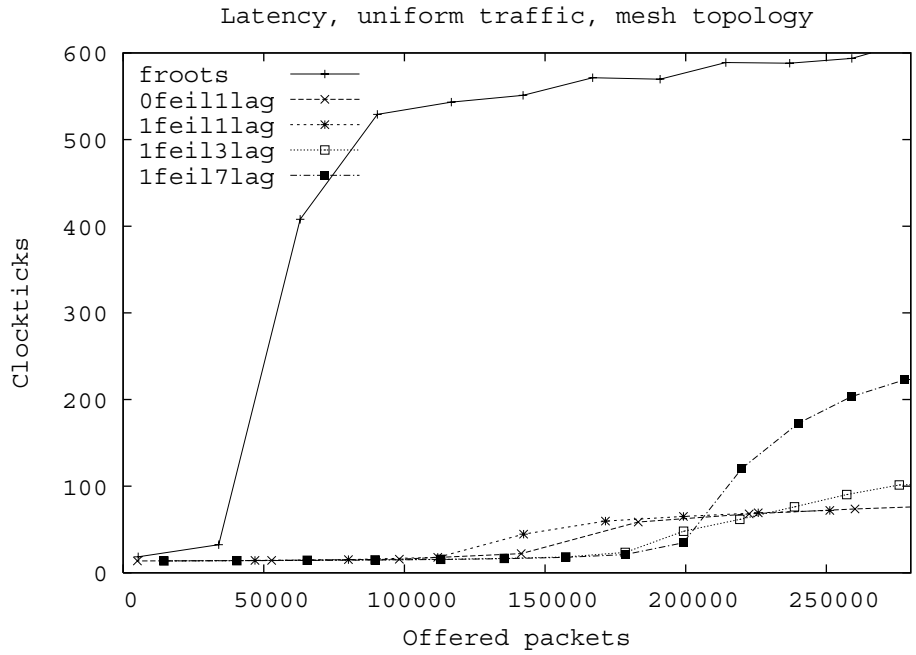
Figur 5.5: -Throughput for Dimension-Order
og FRoots med null – og en feil

Figur 5.5 viser Dimension-Order algoritmen med henholdsvis null og en feil og algoritmen blir kjørt med et virtuelt lag (altså en virtuell kanal). I tillegg viser grafen FRoots med null feil og en feil. Denne gangen er det parvis trafikkmønster som brukes.

Det mest påfallende med grafene er at situasjonen er snudd på hodet - FRoots yter bedre enn Dimension-Order. For å forklare resultatet, kan det igjen være en tanke å se tilbake på hvordan parvis trafikkmønster oppfører seg. Siden Dimension-Order alltid vil gi samme sti mellom to noder, mens FRoots kan bruke flere stier, vil sistnevnte yte bedre da den kan sende flere pakker ut i nettverket mot sine destinasjonsnoder.



Figur 5.6 (a) -Throughput for FRoots og Dimension-Order. FRoots kjører med en feil. Dimension-Order kjører med henholdsvis null feil og et lag, en feil og et lag, en feil og tre lag og en feil og syv lag.



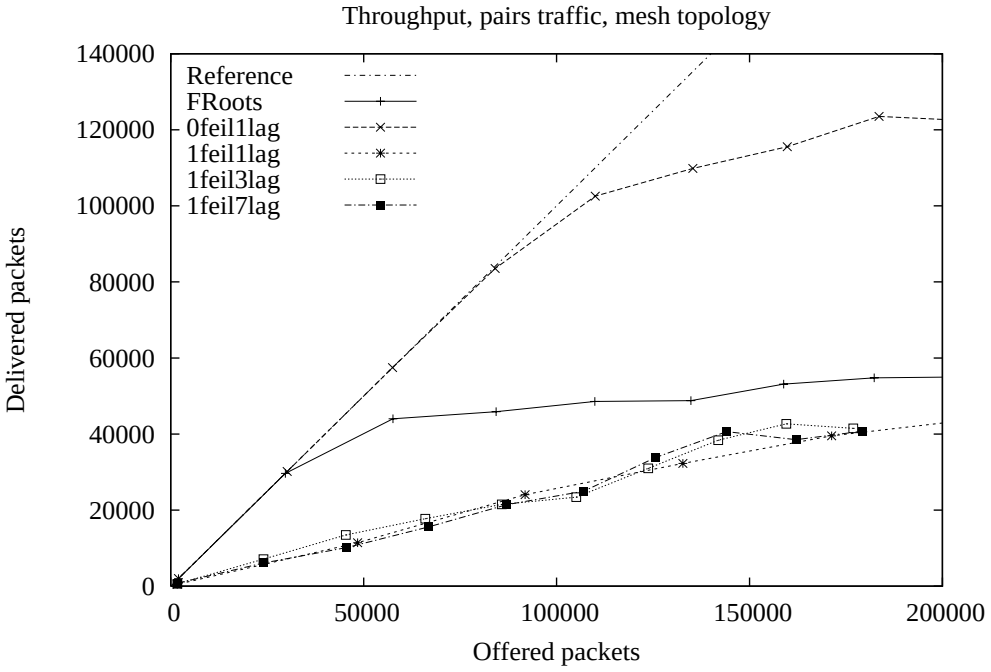
Figur 5.6 (b): Latency for FRoots og Dimension-Order. FRoots kjører med en feil. Dimension-Order kjører med henholdsvis null feil og et lag, en feil og et lag, en feil og tre lag og en feil og syv lag.

Siden FRoots kjører med syv lag fant vi altså ut at det kunne være nyttig å kjøre Dimension-Order med flere lag for å se om de nærmer seg hverandre når antall virtuelle lag blir likt. Dimension-Order med null feil og ett lag, en feil og et lag, en feil og tre lag og en feil og syv lag vises. Dessuten tas det feilfrie tilfellet og et virtuelt lag også med for å se forskjellen. Resultatene vises i figur 5.6. FRoots med en feil er også tatt med.

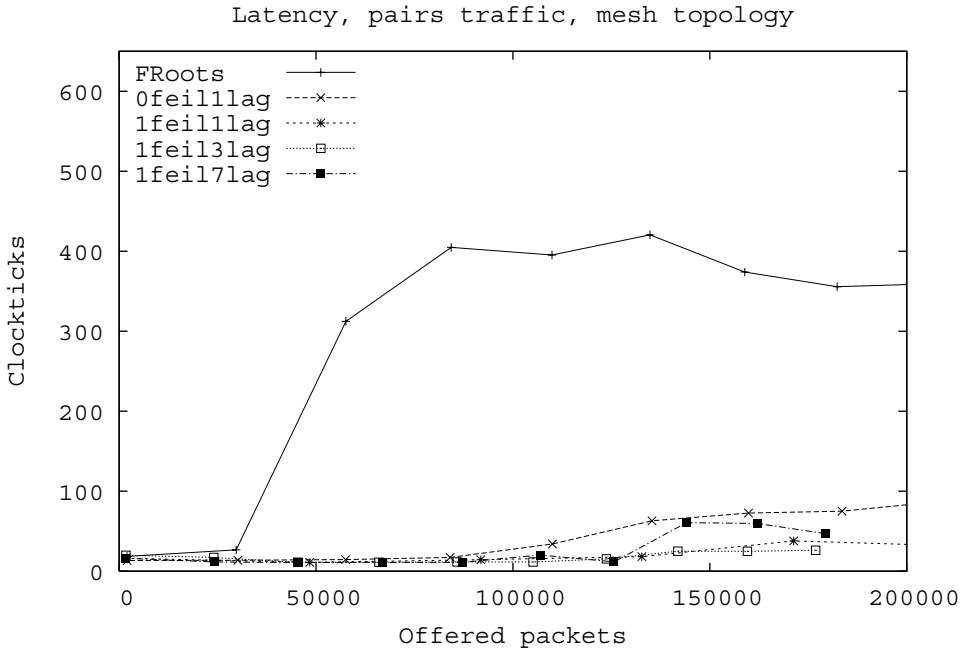
Vi ser av figur 5.6a) at Dimension-Order yter dårligere ved en feil og bruk av et virtuelt lag enn ved ingen feil og et virtuelt lag. Dette kan en vel kanskje si er selvforklarende. Det som kanskje ikke er så innlysende, er at vi ser en ganske kraftig økning i gjennomstrømning fra bruk av et virtuelt lag til bruk av tre virtuelle lag mens økningen fra å bruke tre virtuelle lag til syv virtuelle lag avtar mye. En mulig forklaring på dette kan være at i og med at et virtuelt lag, i tilfellet med Dimension-Order, tilsvarer en virtuell kanal, vil en ved å bruke tre virtuelle lag kunne bruke tre ekstra kanaler. En vil dermed klare å sende gjennom flere pakker enn kun ved å bruke et virtuelt lag (en unngår altså ”head of line” blokkering). Dette vil kunne tilsvare den kraftige økningen i gjennomstrømmingen.

Den lille økningen fra å bruke tre virtuelle lag til å bruke syv virtuelle lag, kan forklares i at det er begrenset hvor mange pakker linken klarer å ta unna i løpet av en simuleringsperiode. Hadde en kjørt på med flere lag, ville en kanskje, til slutt, sett at det ikke hadde noen betydning for gjennomstrømmingen om en økte antall virtuelle lag.

Vi kan av latency – grafen se at ved å bruke syv virtuelle lag, får Dimension-Order en lignende latencyoppførsel, som FRoots, bare ikke så dramatisk. Dette kan forklares i at det blir flere pakker å få frem til en destinasjonsnode og det vil som følge av det ta lengre tid for hver pakke å bli sendt videre når systemet er overfylt av pakker.



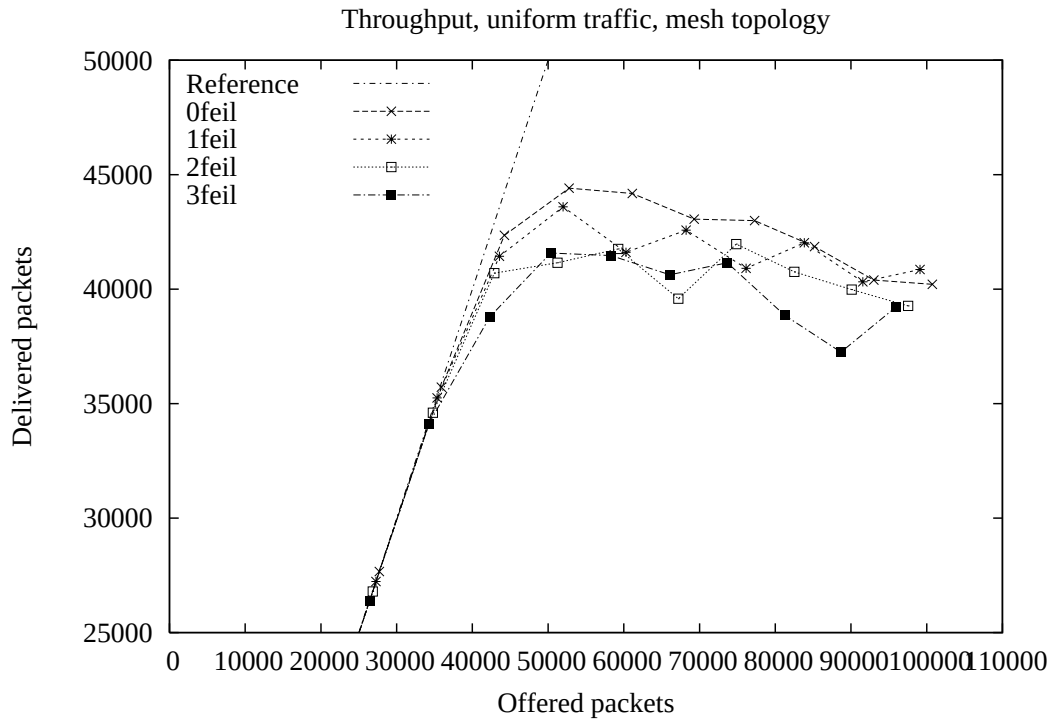
Figur 5.7 (a): Throughput for FRoots og Dimension-Order. FRoots kjører med en feil. Dimension-Order kjører med henholdsvis null feil og et lag, en feil og et lag, en feil og tre lag og en feil og syv lag.



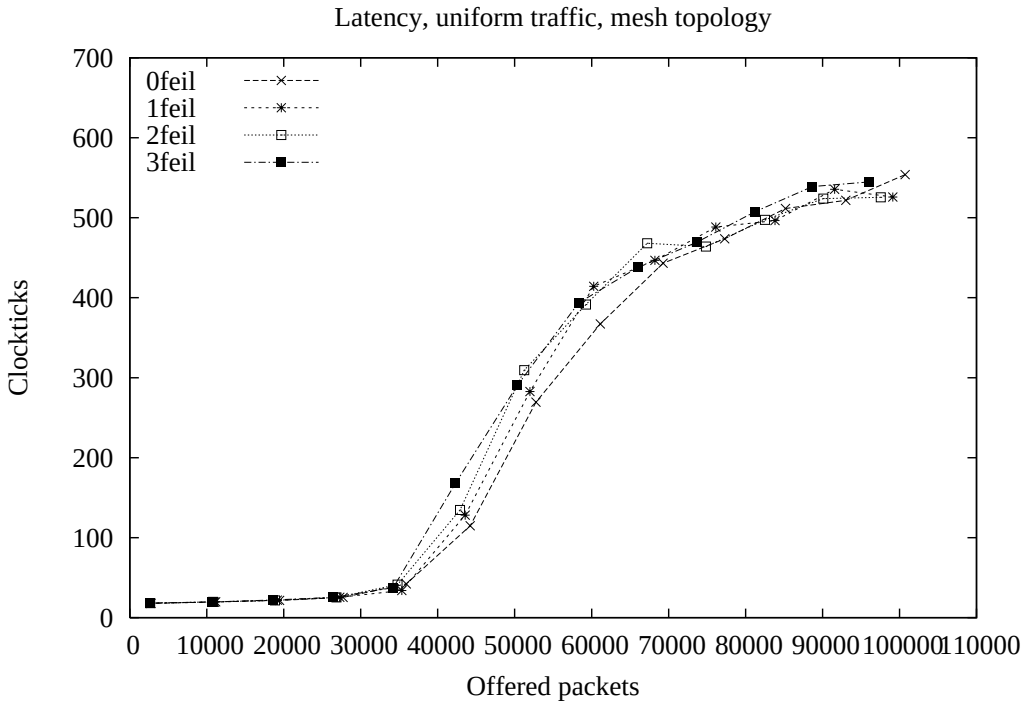
Figur 5.7 (b) – Latency for FRoots og Dimension-Order. FRoots kjører med en feil. Dimension-Order kjører med henholdsvis null feil og et lag, en feil og et lag, en feil og tre lag og en feil og syv lag.

Figur 5.7 viser for parvis trafikkmønster, og for de samme tilfellene som i figur 5.6. Altså at FRoots kjører med en feil og Dimension-Order kjører med henholdsvis null feil og et lag, en feil og et lag, en feil og tre lag og en feil og syv lag.

Vi kan her se en veldig dramatisk nedgang i gjennomstrømming for Dimension-Order algoritmen, fra at det er null feil i nettverket, til en feil. En forklaring er at en feil fører til at flere endenoder blir disablet som kilde- og destinasjonsnoder, og dermed blir det færre noder som både kan generere og ta unna pakkene. Vi ser at det har liten betydning hvor mange lag det kjøres med. Dimension-Order gir alltid samme stien mellom to noder og trafikken blir ikke spredt på samme måte ved parvis trafikkmønster og uniformt trafikkmønster.



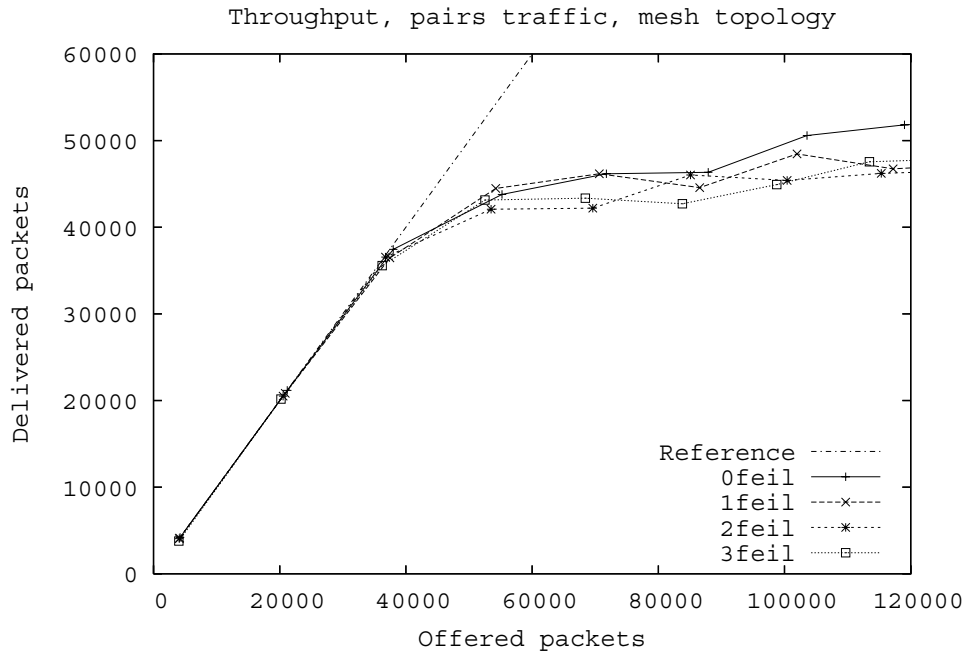
Figur 5.8 (a): Throughput for FRoots ved økende antall feil. Vi har tatt med resultater for null-, en-, to- og tre feil.



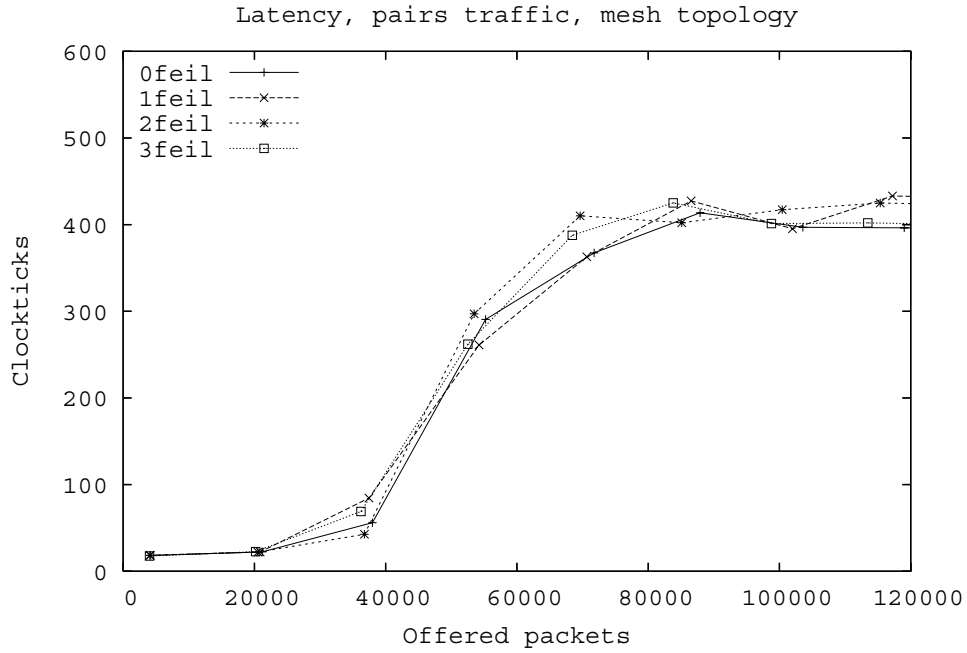
Figur 5.8 (b) – Latency for FRoots ved økende antall feil. Vi har tatt med resultater for null-, en-, to- og tre feil.

Figur 5.8 viser hvordan FRoots-algoritme nyter med flere feil i nettverket. 0feil står for FRoots med 0 feil, 1feil står for FRoots med 1 feil osv.

Av figur 5.8a) ser vi at Froots – algoritmen yter dårligere og dårligere, desto flere feil som er i nettverket. Med bakgrunn i det som er forklart om den statiske feilmodell i kapittel 4.8.1, kan dette forklares på følgende måte; en node mindre i nettverket, betyr at det er en node mindre i nettverket til å ta unna pakker. Nettverket vil starte før med å kaste pakker. Men, vi ser også at en node feilet ikke utgjør den helt store forskjellen. Latency-grafen (figur 5.8 b)), viser at pakkene bruker lengre tid fra kildenode til destinasjonsnode, når flere noder har feilet i nettverket. En node mindre i nettverket betyr færre stier som kan få frem pakkene til destinasjonsnodene, som medfører at pakkene bruker lengre tid.



Figur 5.9(a) –Throughput for FRoots ved økende antall feil. Vi har tatt med resultater for null-, en-, to- og tre feil.



Figur 5.9(b) –Latency for FRoots ved økende antall feil. Vi har tatt med resultater for null-, en-, to- og tre feil.

Figur 5.9 viser resultatene for et nettverk med parvis trafikk. Figuren viser viser hvordan FRoots-algoritmen yter med flere feil i nettverket. 0feil står for FRoots med 0 feil, 1feil står for FRoots med 1 feil osv.

Vi ser at det utgjør liten forskjell på gjennomstrømmingen å fjerne en node i nettverket. Ennå mindre enn når en bruker uniformt trafikkmønster. Grunnen til dette kan være at hver node kun kommuniserer med en annen node i en hel simuleringsperiode. Det vil derfor påvirke gjennomstrømmingen i nettverket mindre at en fjerner den fra nettverket. Av latency – grafen ser vi også at det påvirker tiden en pakke bruker fra kildenode til destinasjonsnode i mindre grad enn ved uniformt trafikkmønster.

5.3 Oppsummering

I dette kapittelet har vi tatt med parametere som vi bruker i simulatoren, presentert resultatene og diskutert de. Vi har også prøvd å komme med en forklaring på hvorfor resultatene ble som de ble.

Kapittel 6 Konklusjonen

I oppgaven har vi sett nærmere på ytelsen til rutingalgoritmer i et nettverk med forskjellige trafikkmønster.

6.1 Oppsummering

Vi har sett at både Dimension-Order og FRoots blir påvirket av størrelsen til nettverket på den måten at de yter bedre dess flere noder som er i nettverket. Vi kjørte begge algoritmene på ulike trafikkmønster og prøvde ut Dimension-Order med flere virtuelle kanaler. Det viste seg at den, ved uniformt trafikkmønster, gav en enda bedre ytelse enn med en virtuell kanal, mens ved parvis trafikkmønster gav antall kanaler nesten ikke noe utslag.

For begge algoritmene gjaldt også følgende; at det var best å holde seg under knekkpunktet for å få best ytelse. Vi så at det etter knekkpunktet ble nådd, brukte pakkene lengre tid til destinasjonen i tillegg til at flere pakker ble kastet.

Det siste som ble prøvet ut, var hvilken innvirkning flere feilede noder har for FRoots. Det viste seg at både ytelsen og latency sank dess flere noder som feilet.

6.2 Konklusjonen

Det ble nevnt i innledningen at forfatteren av denne oppgaven hadde en følelse av at FRoots kom til å overkjøre Dimension-Order totalt uansett hvilket tilfelle som skulle undersøkes.

Den første konklusjonen må bli at desto jevnere trafikken er fordelt utover nodene, dess bedre fungerer Dimension-Order algoritmen. Det er best for gjennomstrømmingen å bruke en adaptiv rutingalgoritme hvis trafikken er ujevn og en holder seg under knekkpunktet til algoritmen.

Det andre en kan si er at dess flere feil som eksisterer i topologien, dess dårligere yter rutingalgoritmene. Innledningsvis ble det sagt at det eneste systemet bør merke, ved feil, er at algoritmene får mindre gjennomstrømming. Det bør være en kontrollert nedgang i gjennomstrømming. En kan si at FRoots oppfyller dette kriteriet til veldig bra. Når vi fjerner en node fra nettverket, ser vi en forandring i gjennomstrømmingen, men ikke mye. Dessuten ser det ut til at gjennomstrømmingen blir redusert med like mye for hver node som feiler. Dette ser ut til å gjelde for begge trafikkmønsterne som er brukt. Etter å ha observert Dimension-Order,

I tillegg kan bør systemet operere under knekkpunktet.

Til slutt kan vi ta med at de som skulle ønske å bruke noen av disse rutingalgoritmene i værvarslingssystemet, bør tenke på hvilket trafikkmønster de bruker. Det ser ut som trafikkmønsteret har stor betydning for hvilken algoritme som er best.

6.3 Fremtidige utvidelser

Som fremtidige utvidelser i oppgaven kan en se for seg flere ting. For det første at en kjører med flere forskjellige trafikkmønster for å ”kvalitetssikre” konklusjonen. For det andre kan en se for seg å ta med flere algoritmer. I tillegg kan en bruke dynamiske feil istedenfor statiske feil.

Referanser

[Blough] D.Blough and S.Najand, ”Fault tolerant multiprocessor system routing using incomplete diagnostic information,” Proceedings of the 6th International Parallel Processing Symposium, pp. 398 – 402, April 1992.

[Boden] n.J Boden, D.Cohen, R.E Felderman, A.E Kulawik, C.L Seitz, J.N Seizovic and Wen-King SUu. Myrinet – a gigabit-per-second local – area network. IEEE MICRO, 1995

[Boppana] R.V Boppana and S.Chalasani, “A comparison of adaptive wormhole routing algorithms, ” Proceedings of the 20th International Symposium on Computer Architechture, pp.351-360, May 1993.

[Carson] “Modeling and Simulation World Views. I Proceedings of the 1993 Winter Simulation Conference, 1993”

[Chen] M.-S.Chen and K.G.Shin, “Adapitive fault-tolerant routing in hypercube multi computers,” IEEE Transactions on Parallel and Distributed Systems, vol.C-39, no. 12, pp.1406-1416, December 1990.

[Chien] "A.A.Chien and J.H. Kim, "Planar-adaptiv routing:Low-cost adaptive networks for multiprocessor networks," Proceedings of the 19th International Symposium on Computer Architecture, pp.268-277, May 1992."

[Cosmic] C.L Seitz, “The Cosmic Cube,” Communications of the ACM, vol.28, no.1,

[Dally] W.J Dally and H.Aoki, “Deadlock-free adaptive routing in multicomputer networks using virtual channels,” IEEE Transactions on Parallel and Distributed Systems, vol.4 no.4, pp.466-475, April 1993.

[Duato] Jose Duato, Sudhakar Yalamanchili and Lionel Ni “Interconnection Networks: An Engineering Approach,“ 2003 av Elsevier Science.

[Emstad] "TTM 4110 Pålitelighet og ytelse i informasjons – og kommunikasjonsteknologi”

[Glass] ”C.J.Glass and L.M.Ni ”The turnmodell for adaptiv ruting,” Proceedings of the 19th

International Symposium on Computer Architecture, pp.278-287, May 1992.

[Gopal] I.S Gopal, “Prevention of store-and-forward deadlock in computer networks, ” IEEE Transactions on Communications, vol.COM-33, no.12, pp.1258-1264, December 1985.

[Horst] R.Horst. A flexible ServerNet – based fault tolerant architechture. In proceedings 25th international Symposium on Fault tolerant Routing, 1995

[inf] Infiniband Architechture specification. Infiniband Trade Association.

[Theiss 02] "Evaluering av metodar for svitsja SCI"

[Intel] Intel, iPSC/1 Reference Manual, Beaverton, OR, 1986.

[Jain] R.Jain “The art of computer systems performance analysis, Techniques for

Experimental Design, Measurement, Simulation and Modeling." New York, John Wiley&Sons Inc. 1991

[J-Sim] http://www.j-sim.org/

[Kessler] R.E Kessler and J.L Schwarzmeier, "Cray T3D: Anew Dimension for Cray Research," Proceedings of Compcon, pp.188-195, August 1996.

[Lee] T.Lee and J.P.Hayes, "A fault-tolerant communication scheme for hypercube computers," IEEE Transactions on Computers, vol..C-41, no.10,pp1242-1256, October 1992.

[Leland] "On the Self-Similar Nature of Ethernet Traffic (Extended Version)"

[Lysne] Olav Lysne, Jose Miguel Montanana, Timothy Mark Pinkston, Jose Duato, Tor Skeie, Jose Flich Simple Deadlock-Free Dynamic Network Reconfiguration

[Myrinet] Myrinet Inc. Guide to Myrinet-2000 Switches and Switch Networks.

[Ni] L.M Ni og P.K McKinley. A Survey of wormhole routing techniques in direct

[Noakes] M.Noakes, D.A Wallach, and W.J. Dally, "The J-Machine multicomputer: An architechtural evaluation," Proceedings of the 20th International Symposium on Computer Architechture, pp. 224-235, May 1993.

[Pang] R.Pang, T.Pinkston and J.Duato. The Double Scheme: Deadlock-free Dynamic reconfiguration of cut-through networks. In Proceedings of 2000 International Conference on Parallel Processing (ICPP'00), Toronto, Canada, August 2000. Ohio State univ.

[Paragon] Intel, Pragon XP/S Product Overview, Supercomputer Systems Division, Beaverton, OR, 1991

[PCI] PCI-SIG. PCI-Express. http//www.pcisig.com/, 2003

[pfeiffer] "http://www.cs.nmsu.edu/~pfeiffer/classes/573/notes/topology.html"

[Schroeder] M.D Schroeder, A.D Birrell, M.Burrows, H.Murray, R.M NeedHam, T.L Rodeheffer, E.H Satterthwaite and C.P Parker. AutoNet: a high-speed, self configuring local area network using point to point links. SRC Research Report 59, Digital Equipment Corporation, 1990

[Seitz] W.J. Dally and C.L. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," IEEE Transactions on Computers , vol.C-36, no.5, pp.547-553, May 1987

[Tannenbaum] Andrew S. Tannenbaum. Computer Networks. Prentice Hall International, 1996.

[Theiss] "Modularity, Routing and Fault Tolerance in Interconnection Networks", Ingebjørg Theiss

[Wang] D.Blough and H.Wang, “Cooperative diagnosis and routing in fault-tolerant multiprocessor systems,” Journal of Parallel and Distributed Computing, vol.27, pp.205-211, June 1995.

[whitney] H.Whitney. Congruent graphs and the connectivity of graphs. Amer.J.Math, 54:150-168, 1932

[Willinger] Walter Willinger, Murad S.Taqqu, Robert Sherman and Daniel V.Wilson. ”Self-similarity through high-variability: statistical analysis of Ethernet LAN Traffic at the source level” IEEE/ACM Transactions on Networking, 5(1):71-86, 1997