Universitetet i Oslo
Institutt for informatikk

En studie av flytkontroll og bufferstørrelser i Ethernet som tett koblet nettverk

Ernst Gunnar Gran

Hovedoppgave

22. mai 2007

Forord

Denne oppgaven er skrevet som en del av mitt studium for graden candidatus scientiarum ved Det matematisk-naturvitenskapelige fakultet ved Universitetet i Oslo. Arbeidet med oppgaven er dels gjennomført ved Institutt for Informatikk, og dels ved Simula Research Laboratory, hvor min veileder Olav Lysne har sitt hovedvirke.

Jeg ønsker først og fremst å takke min veileder for inspirasjon, gode råd og usedvanlig tålmodighet i en periode hvor arbeidet med denne oppgaven ble lagt til side til fordel for jobb, kone og barn (nevnt i vilkårlig rekkefølge). Videre vil jeg takke min arbeidsgiver Simula Research Laboratory og min sjef Åsmund Ødegård for å ha vist stor fleksibilitet. Uten en forståelsesfull arbeidsgiver ville arbeidet med denne oppgaven blitt vanskeliggjort. Mange takk skal også Sven-Arne Reinemo ved Simula Research Laboratory ha for spennende diskusjoner og nyttige kommentarer til oppgaven. En stor takk til min far, Lasse Gran, for korrekturlesing, og sist, men ikke minst: En stor takk til min kone Gunn Marit og min sønn Isak. Dere er mitt muntrasjonsråd!

Ernst Gunnar Gran

Oslo, 22. mai 2007

Innhold

Kapittel 1

Introduksjon

Dette kapittelet inneholder en kort introduksjon til datakommunikasjon og datanettverk. Det gis en lett beskrivelse av bakgrunn for valg av problemstillinger samt definering av disse. Terminologi og begreper introduseres, mens de vil bli ytterligere presisert i neste kapittel. Valg av metode blir begrunnet mot slutten av kapittelet.

1.1 Det allestedsnærværende nettverk

1941: Konrad Zuse (1910-95) fullfører Z3*, en av verdens første, fullt funksjonsdyktige programstyrte datamaskiner*[1]*. Z3 klarte å håndtere om lag 20 til 60 operasjoner i minuttet, avhengig av operasjonenes kompleksitet.*
2007: IBMs BlueGene/L *hos Lawrence Livermore National Laboratory, USA, troner øverst på listen over verdens raskeste datamaskiner. Maskinen ble installert i 2005, men har etter den tid blitt utvidet til å omfatte nå totalt 131.072 prosessorer med en samlet teoretisk maksimal ytelse på 367 Tflops*[2] *(april, 2007).*

De to eksemplene ovenfor illustrerer hvilken enorm utvikling elektronisk databehandling har hatt i løpet av om lag 65 år. Tiden da én datamaskin alene dekket behovet for datakraft og tjenester hos en bedrift eller institusjon er i vesentlig grad forbi. Vi ser stadig oftere at et *nettverk* av maskiner eller prosessorer utfører de oppgavene som tidligere ble utført av én. Vi har beveget oss fra én prosesserende enhet til et sofistikert nettverk av samarbeidende enheter.

Forskjellige typer nettverk har blitt utviklet, avhengig av hvilke oppgaver man har ønsket å løse. Noen nettverk har blitt skreddersydd for å

[1] Den mer kjente ENIAC ble utviklet i USA først fra 1943, og ble tatt i bruk ved University of Pennsylvania i 1946.

[2] teraflops: billioner flyttallsoperasjoner per sekund.

oppnå svært høy regnekraft ved hjelp av massiv, parallell prosessering. Andre nettverk har blitt utviklet med det for øyet å sikre størst mulig grad av pålitelighet og tilgjengelighet for kritiske applikasjoner og tjenester. Atter andre igjen har primært blitt konstruert for utveksling av informasjon og deling av ressurser innad i en bedrift eller organisasjon.

I de senere år har det blitt svært vanlig å koble forskjellige, autonome nett sammen til enda større nettverk. Slike *internettverk*, hvorav det vi idag kaller Internett er det mest opplagte eksempelet, har gjort det mulig med nye former for mellommenneskelig kommunikasjon. Foruten elektronisk post har blant annet telefoni, videokonferanser og fjernundervisning fått en ny og effektiv distribusjonskanal. Formidling av informasjon har fått nye muligheter, særlig gjennom fremveksten av World Wide Web (WWW). Tekst, lyd, bilde og video er nå selvfølgeligheter i det nye "verdensbiblioteket" som WWW utgjør. Som en del av Internetts utvikling ser vi også en økende tendens til at programvare på forskjellige datamaskiner samarbeider med hverandre på kryss av bedrifter, organisasjoner og geografiske avstander, uten at mennesker er innblandet i kommunikasjonen.

Men utviklingen innen datanettverk har ikke bare gitt oss nettverk med enorm geografisk utstrekning, slik som Internett. Nettverksteknologi finner stadig nye bruksområder hvor den fysiske størrelsen må holdes på et minimum. For eksempel blir datamaskiner som produseres i dag i større grad enn tidligere orientert rundt en egen, intern nettverksstruktur. Den tradisjonelle buss-arkitekturen er i ferd med å måtte vike for teknologi som er både raskere, mer fleksibel og mindre plasskrevende. Komponenter kobles sammen i et nettverk for til sammen å utgjøre det vi tradisjonelt forbinder med én datamaskin. Et annet eksempel på nettverk i miniatyr finner vi om vi søker enda dypere inn i datamaskinen – helt til maskinens hjerne, prosessoren. Litt forenklet kan vi si at man frem til nylig har hatt prosessorer som består av én kjerne. Denne kjernen utfører de beregninger som datamaskinen til en hver tid arbeider med. For å kunne fortsette å lage raskere og bedre prosessorer, på tross av de fysiske begrensningene man er i ferd med å nå, søker prosessorprodusentene å integrere flere kjerner internt i én og samme prosessor. Intel og AMD er i ferd med å gjøre to kjerner til standard selv blant billigere prosessorer. Sun lanserte sin prosessor med åtte kjerner alt i siste kvartal 2005[34], mens IBM med sin PowerPC- og Cell-arkitektur ikke bare leverer flerkjerneprosessorer til tjenermarkedet, men også til spillkonsollene Xbox360 og PlayStation3[8, 32]. I fremtiden ser man for seg at antall kjerner per prosessor øker betraktelig. Intel viste i februar 2007 frem en prototyp på en prosessor med 80 kjerner. En slik konstruksjonen avhenger av effektiv kommunikasjon mellom kjernene internt i prosessoren. Vi snakker her om behov for et svært kompakt nettverk internt i én prosessor, et nettverk internt i én brikke.

Eksemplene ovenfor er ikke ment å gi et komplett bilde av alle områder hvor nettverksteknologi blir, eller kan bli, brukt, men de illustrerer bruk

av nettverk på forskjellige nivåer. Vi har nettverk av nettverk, nettverk av datamaskiner, nettverk for å bygge opp én datamaskin og nettverk internt i én brikke. I tillegg finner vi en rekke varianter av nettverk innenfor hvert av de nevnte nivåene. Nett på forskjellige nivåer, og innen forskjellige bruksområder, har naturlig nok forskjellige krav til funksjonalitet og underliggende teknologi. I denne oppgaven skal vi ha hovedfokus på nettverk som er *små, men kravstore*. Hva vi mener med dette skal vi presisere i løpet av dette og neste kapittel. La oss starte med å kikke på noen klassifiseringer av nettverk for å tydeliggjøre hva vi skal jobbe med.

1.2 Klassifisering av nettverk

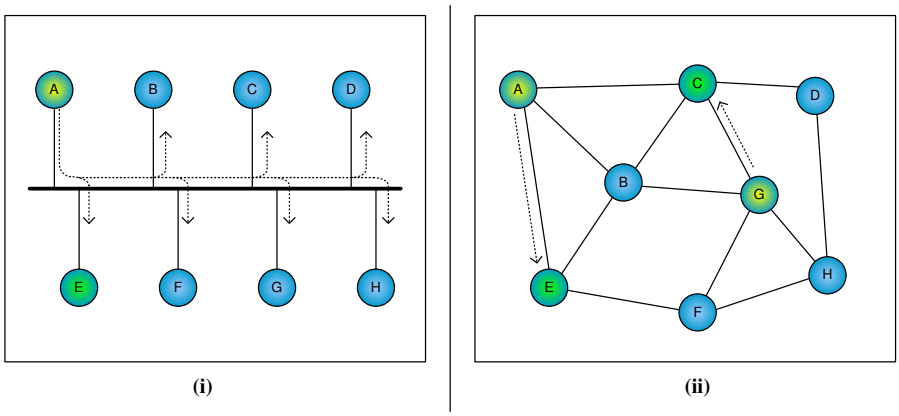
Det finnes flere forskjellige klassifiseringsskjema for datanettverk. Hvilket av disse man benytter avhenger i stor grad av hvorfor man ønsker å dele nettverk inn i grupper, samt ønsket abstraksjonsnivå. Vi skal ta en rask titt på to klassifiseringsskjema som er hensiktsmessige for å sirkle inn de typer nettverk hvor resultater presentert i denne oppgaven vil være av størst interesse.

1.2.1 Klassifisering ut fra type kommunikasjonsmedium

Det først klassifiseringsskjemaet deler inn nettverk i to hovedgrupper ut fra hvordan enhetene, eller *nodene*, i nettverket kommuniserer med hverandre. Man skiller mellom *kringkastings-* og *punkt-til-punkt-nettverk*. I et kringkastingsnettverk deler alle enhetene i nettverket det samme transmisjonsmediet. Det vil si at alle nodene snakker sammen via den samme kommunikasjonskanalen. Når en node sender en melding, gjerne kalt en *pakke* eller en *ramme*, ut på kanalen, vil alle nodene i nettverket motta pakken. Som en del av pakken ligger en *adresse* som identifiserer den noden pakken er tiltenkt. Denne noden vil behandle pakken videre, mens de andre nodene kan ignorere den. Figur 1.1(i) illustrerer, ved de stiplede pilene, hvordan en pakke node A sender til node E også vil nå alle andre noder i nettet. De fleste kringkastingsnettverk har en spesiell, reservert kringkastingsadresse som identifiserer alle noder i nettverket. Ved å bruke denne adressen kan en node på en enkel måte gi alle andre noder den samme beskjeden ved å sende en pakke kun én gang[3].

Hva nå om flere enheter i nettverket forsøker å kommunisere samtidig? Om A forsøker å sende til E samtidig som G sender til C, vil de to nodene A og G snakke "i munnen på hverandre". De fleste kringkastingsnettverk

[3]Mange kringkastingsnettverk støtter også det som på engelsk går under betegnelsen *multicast*. Her er det snakk om spesielle multicast-adresser som identifiserer grupper av noder i nettverket. Ved å sende til en slik adresse kan en node sende en pakke til en gruppe andre noder.



Figur 1.1: Kommunikasjon ved hjelp av kringkasting (i) og punkt-til-punkt-forbindelser (ii).

takler ikke en slik situasjon[4]. En felles kommunikasjonskanal impliserer et felles kollisjonsdomene; om to eller flere noder sender pakker ut i nettverket og pakkene overlapper hverandre i tid, vil pakkene kollidere og ødelegge hverandre. De nodene som sender pakker involvert i en kollisjon, må forsøke å sende pakkene på nytt ved et senere tidspunkt. Det er viktig at man innlemmer en grad av tilfeldighet når man bestemmer tidspunkt for neste forsøk. Fravær av tilfeldighet vil kunne føre til at noder involvert i en kollisjon til stadighet opplever nye kollisjoner da alle nodene venter like lenge mellom hvert nye forsøk.

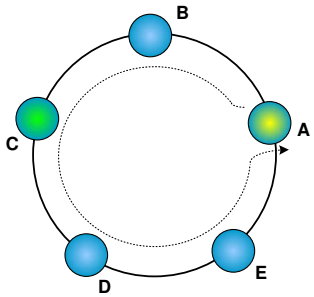
I figur 1.1(ii) har vi koblet sammen nodene fra (i) i et punkt-til-punkt-nettverk. Den felles kommunikasjonskanalen har blitt byttet ut med et sett dedikerte kanaler, gjerne kalt *linker*, som forbinder to og to noder. Med en slik løsning er det ikke lenger noe problem at A snakker til E samtidig som G snakker til C. Hvorvidt kommunikasjonen kan være toveis, slik at informasjonspakker kan sendes i begge retninger over linkene, avhenger av den konkrete teknologien som er i bruk. Et krav til punkt-til-punkt-nettverk er at nettverket ikke er splittet i to disjunkte deler. Med det menes følgende: Uansett hvordan man deler nodene i nettverket inn i to grupper, S og T, så finnes det minst én node s fra S, og minst én node t fra T, slik at s og t har en dedikert kommunikasjonskanal seg imellom. Dersom dette ikke skulle være tilfelle, så har vi i prinsippet å gjøre med to eller flere separate punkt-til-punkt-nettverk, eventuelt enkeltnoder som ikke lenger har kontakt

[4]Ingen regel uten unntak. CDMA[44] er et eksempel på et kringkastingsnettverk hvor flere kan snakke sammen samtidig. Dette er imidlertid ikke det vanlige innen denne typen nettverk.

med resten av nettverket.

Om vi nå vender tilbake til figur 1.1(ii), kan vi merke oss at A ikke lenger kan snakke direkte med G. Denne utfordringen kan vi løse ved enten å legge til en ekstra link mellom A og G, eller sende pakker fra A til G via en eller flere andre noder i nettverket. Generelt sett vil den første løsningen kreve at alle noder i nettverket har en direkte link til alle andre noder. En slik løsning er mindre vanlig da den etter hvert som antallet noder øker krever svært mange linker. Nettverket blir dyrere og fysisk vanskeligere å implementere. Det vanlige er derfor at nodene i nettverket har evnen til å *videresende* pakker som ikke er tiltenkt dem selv. Om A ønsker å sende en pakke til G, kan A sende pakken adressert G til B. B ser at pakken er tiltenkt G og videresender den ut på korresponderende link.

I denne oppgaven skal vi holde oss til punkt-til-punkt-nettverk hvor nodene har støtte for videresending. Før vi går videre til neste klassifiseringsskjema, skal vi nevne en type nettverk som går på tvers av de to hovedgruppene vi har nevnt ovenfor. Et *ringnettverk* er et punkt-til-punkt-nettverk i den forstand at hver node i nettet er koblet til nøyaktig to andre noder med hver sin dedikerte link. På den andre siden er et ringnettverk et kringkastingsnettverk siden hver node i nettverket vil motta alle pakker som blir sendt i nettet. Dette er illustrert ved figur 1.2. Den stiplede linjen viser en pakke som sendes fra node A til node C. C mottar pakken, men fjerner den ikke fra nettverket. Dette vil først skje når pakken er tilbake hos A. Vi skal ikke dvele lenger ved denne typen nettverk, annet enn å la det være en illustrasjon på hvordan et klassifiseringsskjema ikke nødvendigvis passer for alle typer nettverk.



Figur 1.2: Ringnettverk.

1.2.2 Klassifisering etter fysisk størrelse

Det andre klassifiseringsskjemaet vi skal se på deler nettverk inn i grupper etter nettverkets fysiske størrelse. Grunnen til å klassifisere nettverk etter størrelse på denne måten, er det faktum at hvilke teknologier som egner seg, og hva som er fysisk mulig, avhenger av hvor stor avstand man har mellom noder i nettverket[44]. Tabell 1.1 gir en oversikt over de forskjellige klassene. Tallene for nodeavstand er ikke ment som absolutte grenser, men antyder typiske avstander mellom noder i nettverk innen de forskjellige klassene[5].

[5]I litteraturen vil man tidvis kunne finne at begrepene i tabell 1.1 brukes noe forskjellig. For denne oppgaven vil imidlertid den noe overfladiske gjennomgangen av begrepene på påfølgende sider være tilstrekkelig for å illustrere det vi ønsker.

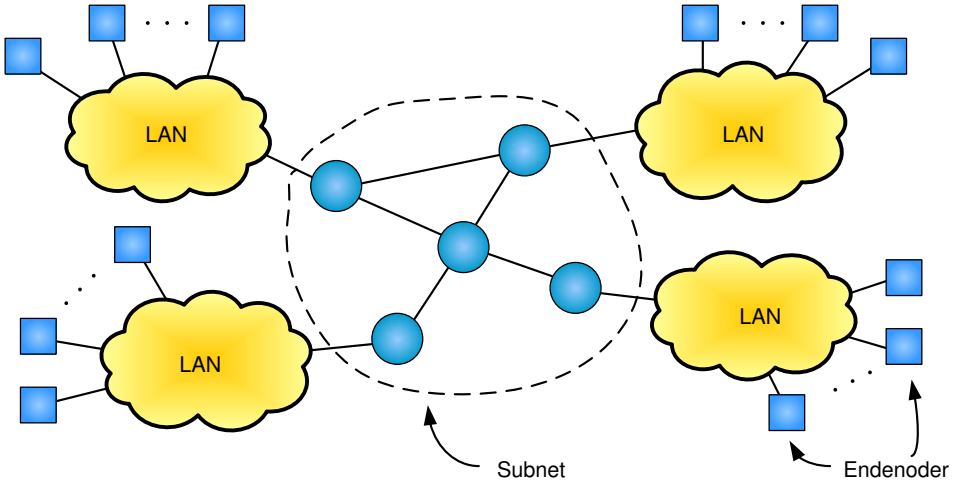
Nodeavstand	**Klasse**	**Akronym**
$<$ 1 cm	Brikkenettverk	NoS (Network on Chip)
5-15 cm	Kretskortnettverk	CBN (Circuit Board Network)
15-200 cm	Systemnettverk	SAN (System Area Network)
2-1000 m	Lokalnett	LAN (Local Area Network)
1-50 km	Bynettverk	MAN (Metropolitan Area Network)
50-1000 km	Langdistansenett	WAN (Wide Area Network)
$<$ globalt	Internett	

Tabell 1.1: Nettverk klassifisert etter fysisk utstrekning.

Hva gjelder fysisk størrelse representerer *brikkenettverk* og *kretskortnettverk* de aller minste formene for datanettverk. Her snakker vi om et nettverk hvor alle noder og kommunikasjonskanaler er samlet på en brikke eller et kretskort. Slike nettverk er som regel stabile hva gjelder antall noder og infrastruktur nodene imellom. Ved konstruksjon av nettverket sitter man på detaljert kunnskap om ønsket funksjonalitet, samt kontroll over dets infrastruktur ned til minste detalj. Dette gir rom for høy grad av optimalisering. Samtidig legger den begrensede fysiske størrelsen strenge føringer for hva man kan velge av tekniske løsninger. Disse faktorene gjenspeiles i at spesielt brikkenettverk i stor grad er preget av skreddersøm. I kapittel 1.1 nevnte vi IBMs Cell-prosessor. Brikkenettverket i denne prosessoren tar form av et skreddersydd ringnettverk[32].

Systemnettverk (SAN) er den fleksible storebroren til kretskortnettverk. I et SAN tillater man større avstand mellom nodene enn hva man kan få til innenfor en brikke eller et kretskort. Økt nodeavstand medfører at man kan ha flere noder i nettverket, siden man ikke lenger har like stramme fysiske begrensninger. Hva gjelder fleksibilitet er SAN-løsninger gjerne utviklet med tanke på at noder og kommunikasjonskanaler vil kunne falle fra eller bli lagt til i løpet av nettverkets levetid. Det er da gunstig at nettverksteknologien ikke stiller strenge krav til hvordan nodene er koblet sammen. Dette står i kontrast til brikke- og kretskortnettverk hvor man ofte finner en rigid infrastruktur. *InfiniBand*[10] er et eksempel på en SAN-teknologi som har fått mye oppmerksomhet.

For menigmann er nok begrepet *lokalnett* og forkortelsen *LAN* blant de best kjente nettverksbegrepene. Dette skyldes antagelig bruksområdet for denne typen nett; LAN har tradisjonelt blitt satt opp for utveksling av informasjon og deling av ressurser innad i en bedrift eller organisasjon. Følgelig består nodene i et LAN gjerne av datamaskiner i form av klienter og tjenere, samt andre delte ressurser som for eksempel skrivere. I kjølvannet av Internetts utbredelse ser vi også at stadig flere setter opp små lokalnett hjemme med det for øyet å kunne dele Internett-forbindelsen mellom boligens data-



Figur 1.3: Langdistansenett (WAN)

maskiner. Innen LAN er det én teknologi som gjennom sin dominans står i en særstilling: Ethernet[1, 42]. Ethernet har røtter helt tilbake til 1973 som eksperimentelt nettverk hos selskapet Xerox Corporation i USA. Siden den tid har Ethernet blitt standardisert og videreutviklet gjennom IEEE[6] 802.3-familien av spesifikasjoner. Ethernet ser ut til å ha en egen evne til å videreutvikle seg, og man søker stadig nye bruksområder. For eksempel arbeider en egen IEEE-gruppe, IEEE P802.3ap[27], med spesifisering av Ethernet over bakplanet.

Et *bynettverk* (MAN) brukes til å koble sammen datamaskiner, eller andre typer noder, spredd utover en by, kommune eller et annet område med tilsvarende størrelse. MAN er et lite brukt begrep, og i følge Andrew S. Tanenbaum er hovedgrunnen til at man i det hele tatt snakker om MAN som en egen nettverksklasse, at man har definert en egen standard for denne typen nettverk[44]. Standarden kalles Distributed Queue Dual Bus og er definert gjennom IEEE 802.6[6].

Et *Langdistansenett* (WAN) er et punkt-til-punkt-nettverk med noder spredd utover et stort geografisk område, gjerne på størrelse med et land eller et kontinent. Innen denne typen nettverk deler man nodene inn i to grupper: endenoder og kommunikasjonssubnett. Dette er illustrert ved figur 1.3. Her ser vi separate LAN med tilhørende endenoder, koblet sammen ved hjelp av et kommunikasjonssubnett. Subnettet består av spesielle noder, gjerne kalt *svitsjer* eller *rutere*, som har som oppgave å videresende pakker mellom

[6]Institute of Electrical and Electronics Engineers Inc.(IEEE) er, for å sitere dem selv, "...verdens ledende profesjonelle forening for fremme av teknologi" (http://www.ieee.org). De står blant annet bak en rekke spesifikasjoner av nettverksteknologi.

LANene. Tradisjonelt baserte man LAN på kringkastings-teknologi, mens WAN var basert på punkt-til-punkt-teknologi. På 90-tallet ble det imidlertid stadig vanligere å bygge opp LAN ved hjelp av svitsjer. Hver (ende)node i LANet fikk ved denne overgangen en dedikert kommunikasjonskanal til en svitsj, og infrastrukturen i et LAN gikk fra å være kringkastingsorientert til å bli punkt-til-punkt-orientert. Et slikt punkt-til-punkt-LAN er på mange måter et WAN i miniatyr.

Som nevnt i kapittel 1.1 er et *internett* en sammenkobling av autonome nettverk. Om de forskjellige autonome nettverkene er basert på forskjellig nettverksteknologi, må man kunne oversette mellom de forskjellige teknologiene man ønsker å koble sammen. Dersom vi i figur 1.3 tillater endenoder innenfor kommunikasjonssubnettet, slik at subnettet i realiteten er et WAN, så vil figuren gi et bilde av hvordan et internett kan se ut.

1.3 Tett koblede nettverk

I denne oppgaven skal vi ta utgangspunkt i en gruppe nettverk vi skal la gå under betegnelsen *tett koblede nettverk*[7] (TKN). Uten å være for kategorisk må man kunne si at TKN har sitt hovedsete innen brikke-, kretskort- og systemnettverk. Når det gjelder kommunikasjonsorientering omfatter TKN løsninger basert på både mer tradisjonelle buss-arkitekturer, kringkastings- og punkt-til-punkt-nettverk. For denne oppgavens del skal vi begrense oss til TKN organisert som punkt-til-punkt-nettverk, og da med utgangspunkt i nett med en fysisk størrelse på linje med SAN.

En av drivkreftene bak utviklingen av TKN er den stadig økende etterspørsel etter regnekraft. Prosessorene blir raskere for hvert år, men krevende anvendelsesområder innen forsvar, meteorologi, romfart og annen naturvitenskapelig forskning innbefatter problemer som krever teraflops-ytelse i tusener av timer i strekk. Et slikt ytelseskrav er langt utenfor rekkevidde, selv for dagens mest avanserte prosessorer. For å kunne oppnå høyere ytelse enn det man til enhver tid får ved hjelp av én enkelt prosessor, lar man derfor flere prosessorer jobbe sammen om oppgavene som skal løses. For at et slikt samarbeid skal gi ønsket ytelsesøkning trenger man en sofistikert måte å koble sammen både prosessorer, hukommelse og I/U-enheter[8] på. Det er som slik sammenkoblingarkitektur TKN har hatt sin utvikling. Teknologien åpner blant annet døren til teraflops-ytelse ved konstruksjon av parallelle datamaskiner, og til massive lagringsløsninger bygget opp av et tett koblet nettverk av lagringsenheter.

[7] På engelsk brukes betegnelsen *Interconnection Network*.

[8] Funksjonsenheter som håndterer inn- og utdata til og fra prosessor og hukommelse. Det kan for eksempel være snakk om platelager og nettverksenheter.

For å kunne bygge parallelle datamaskiner og lagringsnettverk som gir et godt forhold mellom pris og ytelse, er det ønskelig å ta i bruk hyllevare; prosessorer, hukommelse og lagringsenheter som produseres i store kvanta er langt billigere enn skreddersydde komponenter. For eksempel har prosessordesignet nå blitt så sofistikert, og videreutviklingen så dyr, at kun et fåtall firma i verden er i stand til å håndtere det å utvikle nye prosessorer. En dyr utvikling krever igjen tilsvarende stort salg, eller høy pris, for å kunne forsvares økonomisk. Dermed blir skreddersøm en lite gunstig løsning til tross for at en skreddersydd komponent kanskje vil kunne yte bedre enn hva som er tilgjengelig av hyllevare.

Når det gjelder selve sammenkoblingsarkitekturen, det tett koblede nettverket, har man ikke i samme grad som for prosessor, hukommelse og platelagre kunnet benytte seg av hyllevare[21]. Man har utviklet og tatt i bruk standarder som InfiniBand[10] og Fibre Channel[9], men ingen av disse standardene har en utbredelse som kan sammenliknes med en teknologi som Ethernet. Siden LAN, i form av Ethernet, er svært utbredt og billig kunne man ønske seg å benyttet LAN som TKN. Men LAN-teknologi er i utgangspunktet utviklet for å tilfredsstille andre krav enn dem man stiller til et TKN. Dette fikk man erfare da man tidlig forsøkte å konstruere parallelle datamaskiner ved hjelp av Ethernet; nettverket ble en flaskehals.

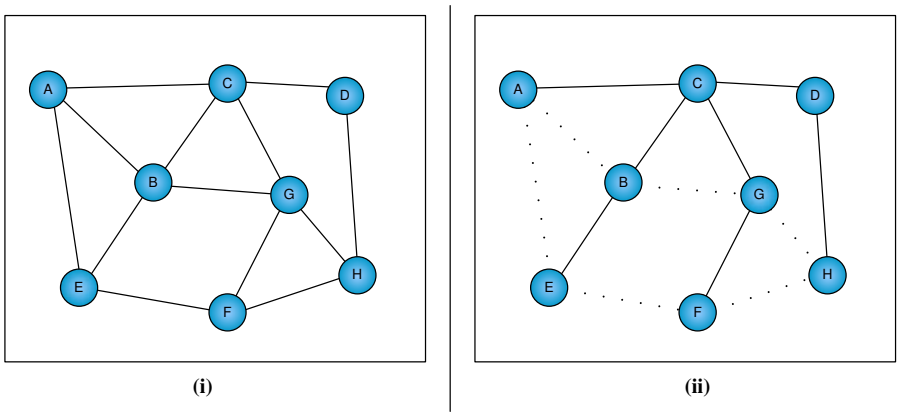
Ethernet er imidlertid i stadig utvikling der de siste spesifikasjonene tillater hastigheter helt opp til 10Gb/s[9][4]. Dette gjør det interessant å på nytt se på Ethernet som en alternativ TKN-teknologi. Selv om 10Gb/s Ethernet er svært raskt, har Ethernet en grunnleggende svakhet som må utfordres dersom teknologien skal egne seg for TKN: Ethernet lar en link i nettverket ligge uvirksom dersom linken som aktiv ville dannet en løkke i nettverket. Vi skal illustrere denne svaketen, samt definere problemstillingene vi skal jobbe med, ved hjelp av et lite eksempelnettverk.

1.3.1 Et lite tett koblet nettverk

Figur 1.4(i) viser punkt-til-punkt-nettverket fra kapittel 1.2.1. Vi skal anta at alle nodene i nettverket har videresendingsfunksjonalitet og dermed kan opptre som svitsjer. Linkene kan benyttes i begge retninger samtidig. Enhver link er med andre ord *full dupleks*. En link kobles til en node ved hjelp av et nettverksgrensesnitt. Selve sammenkoblingspunktet kaller vi gjerne for en *port*. Når vi videre i dette delkapittelet henviser til figur (i) og (ii), refererer dette til de to tilsvarende merkede delene av figur 1.4.

Nettverket i figur (i) er modellert ved hjelp av en *graf* [26]. Grafen består av et sett *noder* og *kanter* som abstrakt sett gjenspeiler henholdsvis noder og

[9] En *bit* representerer et binært tall i en datamaskin. En link med en hastighet på 1b/s kan sende én bit per sekund. 10Gb/s tilsvarer da 10 milliarder bit per sekund.



Figur 1.4: (i) representerer et tett koblet nettverk hvor alle linker er i bruk, mens (ii) viser samme nett med seks uvirksomme linker (stiplede kanter).

linker i nettverket. Vi vil fortsette å modellere nettverk ved hjelp av grafer i denne oppgaven.

En graf, og det representerte nettverk, inneholder en *løkke* dersom man fra en node S ved å følge én eller flere kanter i grafen kan nå tilbake til node S igjen. Vi forutsetter at man ikke benytter samme kant mer enn én gang, ei heller i motsatt retning. Grafen i figur (i) inneholder flere løkker. La $l_{N_1N_2}$ betegne kanten som forbinder to noder N_1 og N_2 i grafen. Da finner vi en løkke ved å følge kantene l_{AB}, l_{BC} og l_{CA}. Et eksempel på en litt større løkke dannes av følgende kanter: l_{BC}, l_{CD}, l_{DH}, l_{HF}, l_{FG} og l_{GB}.

Ethernet, løkker og veivalg

Ethernet tillater ikke at man har løkker i nettverket. Dersom svitsjene i et Ethernet oppdager en løkke, vil de fjerne løkken ved å legge en av løkkens linker uvirksom. Om man hadde benyttet Ethernet for nettverket representert ved figur (i), ville flere linker i nettverket forbli ubrukt. Et mulig resultat er nettverket illustrert ved figur (ii), hvor inaktive linker er markert som stiplede linjer. En slik fjerning av aktive linker er akseptabelt innen LAN fordi man innen dette bruksområdet i utgangspunktet sjelden implementerer nettverk som inneholder løkker[10]. Innen tett koblede nettverk er situasjonen en annen. Vi skal i kapittel 2.3.1 se at et TKN typisk inneholder svært mange løkker. Flere linker i et nettverk betyr både at flere noder har en direkte link mellom seg samtidig som man potensielt sett har flere linker å fordele

[10] Om dette igjen delvis kan forklares ved at den dominerende LAN-standarden Ethernet ikke tillater løkker skal vi ikke diskutere nærmere.

trafikken i nettverket på. At en link ikke kan brukes fordi den vil føre til en løkke i nettet kan innen TKN være svært uheldig. La oss sammenlikne trafikk mellom F og G for de to nettverkskonfigurasjonene representert ved (i) og (ii). I (i) kan de to nodene kommunisere direkte, mens tilsvarende trafikk i (ii) må sendes via hele tre andre noder. Dette til tross for at man faktisk har en direkte link mellom F og H i begge tilfeller. Om Ethernet skal kunne fungere som et TKN må vi kunne håndtere løkker i nettverket på en mer effektiv måte.

Så hva er det som avgjør hvilke linker i nettverket som blir brukt? La oss holde oss til figur (i). Om A ønsker å sende en pakke til H er det i utgangpunktet flere veier, eller *ruter*, å velge igjennom nettverket. Selv om vi begrenser oss til de rutene som gir kortest vei målt i antall mellomliggende noder, her to, så finnes det fire forskjellige alternativer. Ruten bestående av linkene l_{AB}, l_{BG} og l_{GH} er en mulighet. Hvilke rute en pakke faktisk tar avhenger av nettverkets *rutealgoritme*[11]. En rutealgoritme definerer et sett med regler som bestemmer hvordan man finner veier fra kilde til destinasjon gjennom et nettverk. For at man skal slippe å beregne mulige ruter for hver nye pakke lagrer man gjerne gyldige ruter i en *rutetabell*. Når en svitsj skal sende en pakke videre i nettverket, slår den opp i rutetabellen for å finne mulige veier. Merk at det i utgangspunktet kan være mer enn én lovlig vei videre.

Vi skal i kapittel 2.6 se nærmere på hvordan en Ethernet-svitsj bygger opp rutetabellen sin og legger linker uvirksomme. Det vi kan merke oss foreløpig er at om vi tar kontroll over rutetabellene i Ethernet, så kan vi samtidig sørge for at de linkene som i (ii) ble lagt uvirksomme også benyttes. Dette kan vi gjøre ved å la svitsjene benytte en annen rutealgoritme enn det som er vanlig i Ethernet. Den nye rutealgoritmen skal imidlertid ikke velges vilkårlig blant kjente rutealgoritmer. Ethernet stiller ikke spesielle krav til hvordan nodene i nettverket er koblet sammen. Dette gir stor fleksibilitet i forhold til antall noder og linker man benytter i nettverket. Denne egenskap ønsker vi å beholde i det vi innfører en ny rutealgoritme. Dermed kan heller ikke vår nye rutealgoritme stille krav til hvordan nodene er koblet sammen. Samtidig ønsker vi at alle linker i nettverket skal benyttes. En av de mest kjente rutealgoritmene som tilfredsstiller disse to kravene kalles *up*/down*-ruting*[14]. Innføring av up*/down* innebærer kun en endring i rutehåndteringen hos svitsjene. Ethernet kan ellers holdes uforandret, noe som er viktig for å kunne dra nytte av masseproduserte Ethernet-komponenter. For eksempel kan nodenes nettverksgrensesnitt og Ethernets pakkeformat beholdes. Vi formulerer vår første problemstilling:

[11] *Algoritmi* er en latinifisering av navnet til den arabiske matematikeren al-Khwarizmî. Influert av det greske ordet for tall, *arithmós*, danner Algoritmi trolig bakgrunnen for den latinske termen *algorithmus* (brukt av blant andre Leibnitz). På norsk har vi begrepet *algoritme* som refererer til "...nøyaktig sett av regler for fremgangsmåten ved problemløsning"[47].

P1: *Hvilken ytelsesforbedring kan vi forvente fra Ethernet som tett koblet nettverk om vi bytter ut Ethernets tradisjonelle rutetabeller med rutetabeller bygget opp ved hjelp av up*/down*-ruting?*

Ytelsen til et nettverk skal vi måle ved å se på hvor mye trafikk i form av antall pakker vi klarer å sende gjennom nettverket i løpet av en gitt tid. Vi skal også se på hvor lang tid pakkene gjennomsnittlig bruker på å bevege seg fra kilde til destinasjon. For å kunne relatere de to alternative Ethernet-implementasjonene til andre rutealgoritmer skal vi i tillegg sammenlikne dem med et tredje alternativ: *korteste-vei-ruting*. Både up*/down*- og korteste-vei-ruting vil bli gjennomgått i kapittel 2.3.2.

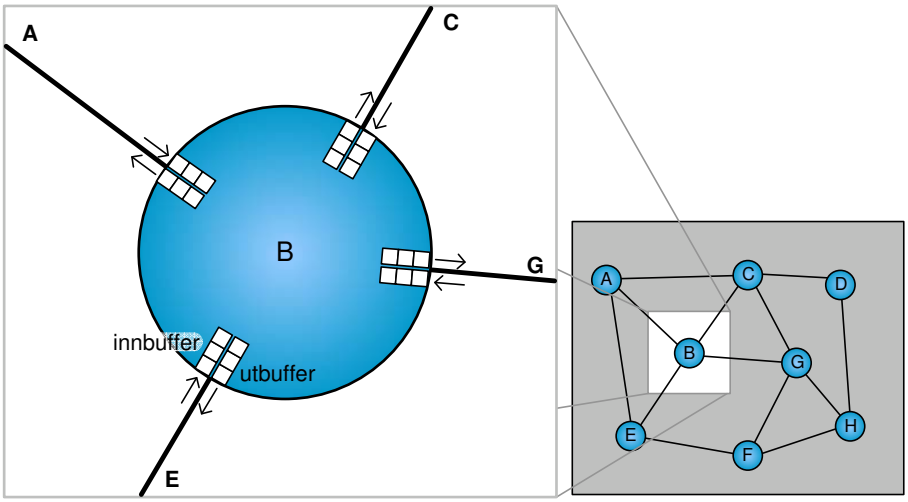
Svitsjer og pakkelagring

Vi skal i denne oppgaven holde oss til svitsjer som er av typen *lagre-og-videresend* (LOV). En LOV-svitsj tar i mot en hel pakke før den sender den videre. En svitsj S vil altså ikke starte videresending av første delen av en pakke før siste del av samme pakke har ankommet S. For at man skal kunne ta imot pakker fra alle svitsjens forskjellige linker samtidig betyr dette at S for hver link som bærer trafikk inn til svitsjen minimum må kunne lagre én hel pakke. Den lagerplassen som settes av for å lagre pakker kaller vi gjerne et *pakkebuffer*, eventuelt bare et *buffer*. Vi skal snart se at det kan være flere grunner til at vi ønsker å lagre pakker i svitsjene, men la oss først ta en titt på hvordan våre svitsjer organiserer sine buffer.

Våre svitsjer benytter seg av *kantbuffer*. Det vil si at enhver link har et reservert buffer, og at ethvert buffer er reservert en gitt link. Svitsjene har med andre ord ikke buffer som dynamisk kan allokeres til trafikk fra forskjellige linker etter behov. Om linken er toveis vil hver retning ha sitt eget buffer. En pakke som kommer inn til svitsjen via en gitt link, legges først i den aktuelle linkens *innbuffer*. Svitsjen slår opp i rutetabellen for å finne rett vei videre før pakken flyttes gjennom svitsjen til det *utbuffer* som er assosiert med den linken som korresponderer med veien videre for pakken. I utbufferet ligger pakken i påvente av å bli sendt videre. Figur 1.5 viser en svitsj med reserverte inn- og utbuffer for hver enkelt toveis-link. Hvert buffer har her plass til tre pakker. En pil indikerer retningen til trafikk knyttet til hvert buffer, og dermed også om det aktuelle bufferet er et inn- eller et utbuffer.

Alle pakkebuffer vi skal arbeide med i denne oppgaven opptrer som først-inn-først-ut-køer (FIFU-køer). I en FIFU-kø skal den pakken som har ligget lengst i køen alltid være den pakken som forlater køen først. Om pakkene P_m og P_n ligger i det samme bufferet, og pakken P_m ble lagt i bufferet først, kan altså ikke pakken P_n forlate bufferet før etter at P_m har gjort det.

La oss vende tilbake til nettverket i figur 1.4(i). Dette er det samme nettverket som vi finner i forminsket utgave i figur 1.5. Anta nå at alle linkene

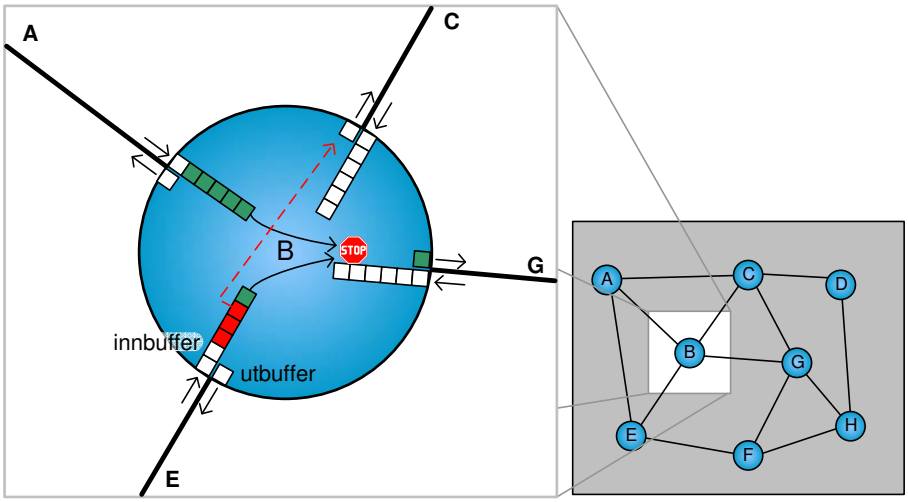


Figur 1.5: Det forstørrede utsnittet illustrerer svitsjen B med inntegnede kantbuffer. Hvert buffer i figuren har plass til tre pakker. Siden alle linkene er toveis har hver link både et inn- og et utbuffer. Pilene viser retningen til trafikken knyttet til hvert enkelt buffer.

---

i nettverket har den samme kapasiteten. Anta videre at A og E ønsker å sende trafikk til G, og at de begge benytter seg av den korteste ruten som går via B. Dersom A og E sender hver sin pakke som begge kommer frem til B samtidig, oppstår det en konflikt hos B siden kun én av pakkene kan sendes videre til G om gangen. Dersom pakken fra E blir valgt for videresending har B tre muligheter for pakke fra A: B må enten kaste pakken, ta vare på den til linken ut mot G blir ledig, eller sende den ut på en annen link enn den som kobler B til G. Trafikken i et tett koblet nettverk er gjerne av en slik type at man ønsker å unngå at pakker kastes. Samtidig skal vi i denne oppgaven forholde oss til rutealgoritmer som ikke tillater at en pakke rutes via alternative veier. I utgangspunktet bør B derfor lagre pakken, men hvor? Skal pakken fra A lagres i innbufferet, eller skal vi flytte den over til utbufferet som korresponderer med linken i retning G[12]? Dersom vi hovedsakelig ønsker å lagre pakker i innbufferet, kan vi ta hensyn til dette ved å la en relativt stor andel av en svitsjs totale bufferplass være reservert innlinkene. Om vi

[12] At man må vente i et utbuffer kan ved første øyekast virke unødvendig, men dette kan skyldes to ting. Om svitsjens indre arbeider raskere enn hastigheten til linkene, vil pakker fra flere forskjellige innbuffer kunne forflyttes gjennom svitsjen til samme utbuffer raskere enn hva utlinken klarer å sende. En annen mulighet er at utlinken for øyeblikket ikke har anledning til å sende trafikk videre. Begge tilfeller vil føre til at pakker blir lagret i utbufferet.



Figur 1.6: Køhodeblokkering: Pakker på vei til C, her merket med fargen rød, må vente til tross for at utbuffer i retning C er ledig. Ventingen skyldes at en blokkert pakke på vei til G, merket med grønt, ligger forran de røde pakkene i innbufferet.

---

på den andre siden ønsker at pakker primært skal lagres ved utportene, bør en tilsvarende stor andel av svitsjenes buffer være reservert som utbuffer. Et tredje alternativ er å la alle inn- og utbuffer være like store.

Så tidlig som i 1987 publiserte Mark J. Karol med flere en artikkel hvor de ved hjelp av formelle medtoder og simulering viser at man i en svitsj bør organisere buffer som utbuffer fremfor innbuffer[31]. Resultatet er kanskje ikke så overraskende. Rent intuitivt kan det virke fornuftig å flytte pakken igjennom svitsjen så raskt som mulig. Må man vente, kan man like gjerne gjøre det ved utporten. Samtidig minker mulighetene for *køhodeblokkering* ved bruk av små innbuffer. Køhodeblokkering oppstår i det en pakke ikke kan sendes igjennom svitsjen til ledig utbuffer fordi en annen pakke som tidligere ankom samme innbuffer er blokkert. Figur 1.6 illustrerer dette problemet. Her har vi utstyrt vår svitsj B med innbuffer som har plass til 6 pakker, mens hvert utbuffer kun har plass til én pakke. Hvite buffer er tomme, mens røde og grønne buffer representerer pakker som skal videresendes til henholdsvis C og G. De røde pakkene blir utsatt for køhodeblokkering; til tross for at utbufferet i retning C er ledig må de røde pakkene vente på at den grønne pakken først i innbufferet har blitt flyttet over til rett utbuffer. Når den grønne pakken forlater innbufferet, avhenger både av når utbufferet i retning G blir ledig, og av hvordan svitjsen vil fordele tilgangen til dette utbufferet blant de pakker som ønsker å benytte det. Vi ser at det befinner seg grønne

pakker i innbuffer fra både svitsj A og svitsj E.

Innen nettverkslitteratur finner man ofte igjen denne argumentasjonen for hvorfor utbuffer er å foretrekke fremfor innbuffer[39, 42, 44]. Felles for de referanser vi har gitt her, inklusive artikkelen i forrige avsnitt, er at de alle argumenterer med utgangspunkt i hvordan én enkelt svitsj oppfører seg. I et tett koblet nettverk har vi imidlertid flere svitsjer koblet sammen ved hjelp av et sett linker. Det å sende en pakke fra en svitsj S_1 til en annen svitsj S_2, kan man abstrakt se på som en forflytting av en pakke innad i et tenkt buffer bestående av utbuffer hos S_1, selve linken og innbuffer hos S_2. Vi forutsetter da at pakker ikke kastes innad i dette abstrakte bufferet. Skal man unngå å kaste pakker, kan man ikke sende trafikk over linken med mindre det er plass i innbuffer hos S_2. Om innbufferet er fullt må S_1 vente med å sende mer trafikk over linken. Skal S_1 opptre som ønsket fordrer dette at S_2 holder S_1 underrettet om når S_1 kan få lov til å sende pakker over linken; S_2 må styre aktivitet på linken ved hjelp av *flytkontroll*. Vi skal i neste avsnitt utdype begrepet flytkontroll, men la oss først fullføre idéen om det abstrakte buffer. Dersom S_1 stoppes av flytkontrollen, vil utbufferet hos S_1 kunne fylles. Med små innbuffer og store utbuffer vil man minske sjansene for køhodeblokkering i innbufferet hos S_2, men man forplanter samtidig blokkeringen raskere til utbufferet hos S_1. Etter denne observasjonen er det ikke lengre like opplagt at utbuffer er å foretrekke fremfor innbuffer i et nettverk med flere svitsjer. Vi er nå klare for å formulere vår problemstilling nummer to. Ytelsen i nettverket skal vi måle på samme måte som forklart ved defineringen av vår første problemstilling.

> P2: *For én svitsj med kantbuffer er det, for å unngå køhodeblokkering, gunstig at kun en liten andel av svitsjens buffer benyttes som innbuffer. Kan vi si det samme om tilsvarende svitsjer i et tett koblet nettverk med flytkontroll, eller har ikke køhodeblokkering i et innbuffer her samme effekt i forhold til nettverkets totale ytelse?*

Flytkontoll

Det finnes flere tilnærminger til flytkontroll, men essensen er alltid den samme: Man ønsker å kontrollere ressursbruk innad i nettverket[19]. For vår del skal flytkontroll forstås som en mekanisme for å sørge for at en sender av trafikk ikke sender raskere enn hva mottager tillater. Dette kan vi oppnå ved å la mottager styre ressursallokering av linken og dermed bruk av eget innbuffer. Behovet for flytkontroll oppstår i det en node i nettverket ikke klarer å hente unna pakker fra et innbuffer like raskt som bufferet fylles. Dette kan for eksempel skyldes at en pakke først i innbufferet hos en svitsj venter på at et utbuffer skal bli ledig, eller at en endenode ikke klarer å hente unna pakker fra nettverket raskt nok. Dersom et innbuffer har gått fullt, må man

enten stoppe tilhørende link eller kaste eventuelle nye pakker som kommer inn over linken. Som nevnt tidligere søker vi å unngå å kaste pakker i et tett koblet nettverk.

Den formen for flytkontroll vi skal konsentrere oss om i denne oppgaven behandler hver link uavhengig av andre linker i nettverket. Flytkontrollen over en gitt link styres kun ut fra tilstanden til tilhørende innbuffer. Hvert innbuffer hos en svitsj monitoreres derfor separat. Ut fra et innbuffers tilstand sendes kontrollpakker til den noden som er koblet til andre enden av tilhørende link. Nøyaktig hva slags informasjon pakkene inneholder vil avhenge av hva slags type flytkontroll som benyttes. Vi skal forholde oss til en variant av flytkontroll hvor en kontrollpakke inneholder en beskjed om at avsender enten må stoppe all trafikk på linken eller eventuelt kan benytte linken ved full hastighet. Vi skal altså arbeide med en flytkontroll som definerer klare av- og på-perioder for trafikk på linken. For Ethernet er en tilsvarende flykontroll definert gjennom IEEE 802.3x[5, 42]. Som LAN benytter man ofte Ethernet uten flytkontroll siden man har et lite anstrengt forhold til pakketap. Dersom man skal benytte seg av Ethernet-teknologi ved konstruksjon av tett koblede nettverk er det imidlertid en nødvendighet at flytkontroll er på plass for at pakker ikke skal kastes unødvendig.

Det er opp til eier av et innbuffer å administrere flytkontrollen over korresponderende link. Når man velger å skru av og på flytkontrollen vil ha innvirkning på utnyttelsesgraden av innbufferet. Skrur man på flytkontrollen for tidlig, risikerer man at man aldri fyller innbufferet. Om så er tilfelle kunne man like gjerne benyttet seg av et mindre innbuffer. Skrur man derimot på flytkontrollen for sent, risikerer man at pakker kastes. Når man skal skru flytkontrollen av igjen, er det viktig at dette ikke skjer så sent at innbufferet blir stående tomt mens pakker venter på å bli sendt over den tilhørende linken. Hvor tidlig man velger å skru av igjen flytkontrollen vil samsvare med hvor mye av innbufferet man ønsker å tømme før man tillater at tilhørende link kan bære trafikk igjen. Vi ønsker å studere om det har vesentlige implikasjoner for trafikken i et nettverk når vi skrur av og på flytkontrollen, og om dette igjen kan gi oss kunnskap som hjelper oss å avgjøre hvor store buffer det er hensiktsmessig å benytte i et tett koblet nettverk. La oss nå formulere vår tredje og siste problemstilling:

> P3: *Når er det hensiktsmessig å skru på og av flytkontroll, og har kunnskap om dette implikasjoner for hvor store buffer svitsjene i et tett koblet nettverk bør ha?*

Legg til slutt merke til at flytkontrollinformasjonen må kommuniseres i motsatt retning av den trafikkstrømmen man ønsker å kontrollere. Effektiv bruk av egne flytkontrollpakker fordrer derfor at man enten benytter linker som er toveis eller eventuelt at to noder alltid er koblet sammen ved hjelp av to linker, en link i hver retning.

1.4 Oppgavens problemstillinger

La oss oppsummere de tre problemstillingene vi søker å besvare i løpet av denne hovedoppgaven:

> P1: *Hvilken ytelsesforbedring kan vi forvente fra Ethernet som tett koblet nettverk om vi bytter ut Ethernets tradisjonelle rutetabeller med rutetabeller bygget opp ved hjelp av up*/down*-ruting?*
>
> P2: *For én svitsj med kantbuffer er det, for å unngå køhodeblokkering, gunstig at kun en liten andel av svitsjens buffer benyttes som innbuffer. Kan vi si det samme om tilsvarende svitsjer i et tett koblet nettverk med flytkontroll, eller har ikke køhodeblokkering i et innbuffer her samme effekt i forhold til nettverkets totale ytelse?*
>
> P3: *Når er det hensiktsmessig å skru på og av flytkontroll, og har kunnskap om dette implikasjoner for hvor store buffer svitsjene i et tett koblet nettverk bør ha?*

Svarene på disse tre problemstillingene vil kunne ha innflytelse på design og konstruksjon av svitsjer tenkt brukt i tett koblede nettverk. Dersom vi får en betydelig ytelsesforbedring i Ethernet ved bruk av up*/down*, indikerer dette at up*/down*-modifiserte Ethernet-svitsjer kanskje kan stå frem som et prisgunstig alternativ for tett koblede nettverk.

Ved P2 setter vi spørsmålstegn ved om det er vesentlig for nettverkets ytelse hvorvidt en svitsj primært benytter seg av utbuffer eller innbuffer. Det har vært en generell oppfatning at utbuffer er å foretrekke. Dersom det skulle vise seg at dette ikke i samme grad gjelder for et nettverk av svitsjer, vil det samtidig medføre at man ved konstruksjon av svitsjer kan velge den løsningen som gir den enkleste, og dermed gjerne billigste, logikken.

I et tett koblet nettverk er vi avhengig av flytkontroll for ikke å kaste pakker unødvendig. Ved å studere en variant av den formen for flytkontroll som er spesifisert for Ethernet søker vi økt innsikt i nettverktrafikkens dynamikk sett i forhold til bruk av link og innbuffer. Vi ønsker å danne oss et bilde av hvordan trafikken påvirkes av når man velger å skru på og av flytkontroll. Dersom det skulle vise seg at man ikke taper vesentlig ytelse på å skru flytkontrollen på tidlig, vil dette samtidig ha implikasjoner for hvor store buffer det er hensiktsmessig å benytte. Mindre buffer betyr enklere logikk, noe som igjen fører til en rimeligere konstruksjon.

1.5 Metode

Arbeidet med denne oppgaven startet med et litteraturstudium. Innen fagområdet for tradisjonelle tett koblede nettverk gir boken *Interconnection*

Networks, an Engineering Approach[21] en omfattende innføring i de grunnleggende prinsipper. På tilsvarende måte gir boken *The Switch book*[42] en god presentasjon av teknologier tilknyttet Ethernet. Disse to bøkene ble derfor et naturlig utgangspunkt for mitt arbeide.

Vi ønsker i løpet av denne oppgaven å studere flere forskjellige aspekter ved tett koblede nettverk. For å kunne sammenlikne ulike nettverkskonfigurasjoner og deres ytelse kan vi se for oss tre ulike metodiske tilnærmelser: *eksperiment*, *formell resonnering* og *simulering*. I påfølgende avsnitt følger en rask introduksjon til de tre metodene. Vi skal se at simulering er den metoden som det i vårt tilfelle er mest hensiktsmessig å benytte.

Eksperiment

Ønsker man å bruke eksperiment som metode, setter man opp et testnettverk som utsettes for trafikk mest mulig lik den man ville hatt ved reell bruk av et tilsvarende nettverk. Målinger gjort ved et slikt eksperiment har potensiale til å gi data som ligger svært tett opp til hva man ville opplevd i et nettverk i produksjon. Slike *empiriske* data ville gitt de mest nøyaktige resultatene, og således gitt det beste utgangspunktet for å kunne besvare våre problemstillinger.

Et eksperiment fordrer imidlertid at man har et testmiljø som er stort nok til at man kan etterlikne de ønskede nettverkskonfigurasjonene. Samtidig må rutealgoritmene med alle tilhørende mekanismer, den ønskede form for flytkontroll med mere, implementeres fullt ut. Dette gjelder også de deler av programvaren som eventuelt må implementeres i maskinvare. Et så krevende implementasjonsarbeid er lite hensiktsmessig for nye idéer som skal testes ut. Det lar seg heller ikke realisere innenfor de rammene et hovedfag har. Da testmiljøet heller ikke er tilgjengelig, er det uaktuelt med eksperiment som metode.

Formell resonnering

Ved bruk av formell resonnering konstruerer man først matematiske og statistiske modeller for det nettverket man ønsker å studere nærmere. Disse modellene kan så brukes til å analysere nettverkets oppførsel og ytelse. Formell resonnering vil gi gode og nøyaktige resultater dersom de underliggende modellene gir en god beskrivelse av nettverket. I vårt tilfelle er imidlertid systemet vi ønsker å modellere, og resonneringen vi ønsker å gjøre rundt modellene, av en slik karakter at de er lite egnet for matematisk eller statistisk beskrivelse. Dette fører til at modellene ville blitt svært kompliserte. Uten tung matematisk bakgrunn ville bruk av formell resonnering i vårt tilfelle innebære så mange forenklinger og antagelser at de analytiske resultatene ville blitt svært upresise. På grunn av dette vil ikke formell resonnering bli benyttet som metode for å besvare våre problemstillinger.

Simulering

Ved bruk av simulering som metode ønsker man å etterlikne virkeligheten uten å bygge opp et reelt nettverk. Dette kan gjennomføres ved at virkeligheten speiles i programvare; man implementerer en *simulator*. Komponenter fra virkeligheten søkes da implementert i tilsvarende kodebiter i programvare. Når simulatoren kjøres, må så programkodebitene interagere med hverandre slik at de gjenspeiler de vesentlige aspekter ved virkeligheten. En gunstig egenskap ved en godt implementert simulator er dens fleksibilitet; ved å endre på parametre til programkoden vil man kunne forandre på egenskaper som for eksempel antall svitsjer og linker i nettverket, mengden trafikk eller hvilken rutealgoritme som benyttes.

En viktig del av det å implementere en simulator består i å validere programvaren man utvikler. Simulatorens presisjon, og dermed de påfølgende resultatene, vil avhenge både av hvilke forenklinger man gjør i forhold til virkeligheten, og av eventuelle feil i programvaren. Selv små feil i programvaren vil kunne gi store utslag i resultatet av en simulering.

For å lette arbeidet ved oppbygging av en presis simulator er det viktig å velge et velegnet programmeringsspråk. De objektorienterte språkene er spesielt godt egnet for simuleringer av den typen vi skal benytte i denne oppgaven. Siden vi i tillegg har mulighet til å ta utgangspunkt i et rammeverk for nettverkssimulering skrevet i det objektorienterte programmeringsspråket Java[25], vil dette gi et godt utgangspunkt for simulering som metode. Simulering lar seg også gjennomføre med de ressursene man har til rådighet innenfor et hovedfag. Kapittel 3 inneholder en grundig gjennomgang av den simulatoren vi har implementert som en del av denne oppgaven. Der vil vi også kommentere de grep som er gjort for å validere simulatoren.

1.6 Språk

Denne oppgaven er skrevet på norsk, mens all faglitteratur lest i løpet av litteraturstudiet er skrevet på engelsk. Alt kildematerialet det henvises til i løpet av teksten er også engelskspråklig.

Rent språklig sett burde alle engelske termer oversettes til norsk når man først velger norsk som skriftspråk. En slik tilnærming var også utgangspunktet for denne oppgaven. Det har imidlertid vist seg at det å oversette engelsk fagterminologi i enkelte situasjoner kan virke lite naturlig og til tider forvirrende. Dette skyldes at det i utgangspunktet finnes flere begreper fra engelsk nettverksterminologi som også benyttes i norsk muntlig tale. Disse engelske begrepene har gjennom muntlig tale blitt innarbeidet i norsk fagterminologi uten at en norsk oversettelse har funnet sted. Det å da skulle komme opp med norske oversettelser av disse mye brukte engelske begrepene i en oppgave som denne vil, spesielt for lesere med god kjennskap til gjeldende fagområde, kunne virke forvirrende og dermed lite hensiktsmessig. Det vil

derfor forekomme enkelte engelske termer i løpet av oppgaven.

Enkelte steder i teksten er engelske faguttrykk blitt oversatt til norsk, mens tilhørende akronymer har blitt bevart i sin opprinnelige, engelske form. Dette gjelder for eksempel alle akronymer knyttet til klassifisering av nettverk etter fysisk størrelse. For eksempel er den engelske betegnelsen *Local Area Network* oversatt til *lokalnett*, mens den engelske forkortelsen LAN er beholdt. Slik delvis oversettelse skyldes at både den norske betegnelsen og den engelske forkortelsen er godt innarbeidet i norsk fagterminologi, uten at en tilsvarende norsk forkortelse foreligger.

Innen Ethernet-terminologi benytter man seg gjerne av termene *ramme* og *adressetabell* for å navngi det vi har omtalt som henholdsvis pakke og rutetabell. Vi skal imidlertid holde oss til pakke og rutetabell også når vi snakker om Ethernet, siden dette er de vanlige termene å benytte når man snakker om tett koblede nettverk.

I kapittel 3 vil vi blant annet komme inn på hvilke klasser simulatoren består av og disse klassenes egenskaper. Å snakke om en klasses egenskaper er strangt tatt litt upresist siden en klasse i seg selv ikke besitter disse egenskapene. I de fleste tilfeller er det slik at en klasse *definerer* de aktuelle egenskaper. Instanser av klassen, i form av objekter, besitter disse egenskapene. For ikke unødvendig å vanskeliggjøre språket i kapittel 3 har vi likevel valgt å snakke om klasser og deres egenskaper selv om dette altså er litt unøyaktig.

I følge Tanums store rettskrivningsordbok[47] er ordet *buffer* et hannkjønnsord. Dette gjelder også dersom ordets betydning skal forstås som "lagerenhet i datamaskin...". Som fagutrykk innen informatikk er det imidlertid ikke uvanlig å benytte ordet som et intetkjønnsord. Vi har i denne oppgaven derfor valgt å bøye ordet buffer som et intetkjønnsord til tross for at dette da strider med Norsk språkråds anbefaling. Ordet bøyes av oss slik:

et buffer - bufferet - buffer - bufferne

1.7 Videre organisering

I kapittel 2 skal vi utdype og presisere begreper vi til nå kun har introdusert. Vi skal se på hva som karakteriserer tett koblede nettverk, og kikke nærmere på teknologi tilknyttet Ethernet. Kapittel 3 inneholder en grundig gjennomgang av den simulatoren vi benytter for å besvare våre problemstillinger. En vesentlig del av arbeidet med denne oppgaven har bestått i å implementere den programvaren som utgjør simulatoren. I kapittel 4 skal vi gå igjennom et sett konkrete simuleringer, analysere resultatene, og vurdere dem i lys av oppgavens problemstillinger. Kapittel 5 inneholder en kort oppsummering av vår funn, samt forslag til videre arbeid. Tillegg A inneholder en del resultater fra simuleringer som støtter opp om de konklusjoner vi trekker i kapittel 4, men som vi ikke fant plass til å kommentere ytterligere i løpet av teksten.

Kapittel 2

Tett koblede nettverk og Ethernet

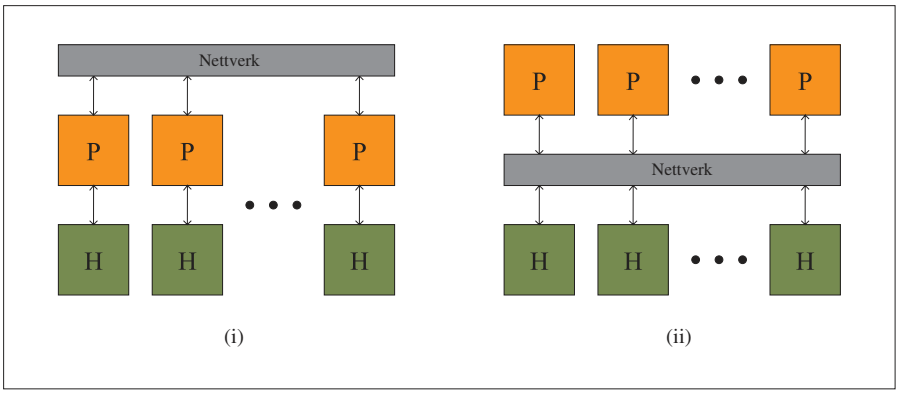
Dette kapittelet inneholder utdypning og presisering av begreper introdusert i kapittel 1. Vi skal se på hva som karakteriserer tett koblede nettverk, vi skal ta en titt på begrepene vranglås, utsulting og kontinuerlig omruting, og vi skal kikke nærmere på teknologi tilknyttet Ethernet.

2.1 Tett koblede nettverk og hyllevare

Idéen om å bruke hyllevare til å bygge parallelle datamaskiner førte i starten av 1980-årene til utviklingen av *multiprosessorer med distribuert hukommelse* (MDH)[21]. Masseproduserte prosessorer, hver med sin egen lokale hukommelse, ble koblet sammen ved hjelp av billig nettverksteknologi. Det skulle imidlertid vise seg at de masseproduserte nettverkskomponentene, som for eksempel kunne være basert på datidens Ethernet-teknologi, ble en stor flaskehals. For bedre å kunne utnytte potensialet til MDH ble derfor sammenkoblingsnettverket etter hvert preget av skreddersydde løsninger. Figur 2.1(i) gir en enkel skisse over MDH-arkitekturen.

Multiprosessorer med felles hukommelse (MFH)[21] er en alternativ arkitektur til MDH. Her ligger sammenkoblingsnettverket mellom prosessorene og hukommelsen. Dette medfører en uniform tilgang til hukommelsen, hvilket gir alle prosessorene samme tilgang til hele minneområdet. En enkel skisse over MFH-arkitekturen er gitt ved figur 2.1(ii). En slik arkitektur forenkler deling av data sett i forhold til en MDH. Den store ulempen med MFH er imidlertid dårlig skaleringsevne. Tiden det tar å aksessere minnet innbefatter forsinkelsen som oppstår på grunn av nettverket. Nettverksforsinkelsen vil øke med størrelsen på systemet.

For å oppnå både god skaleringsevne og et system som er relativt enkelt å programmere, har man utviklet hybrider av MDH og MFH. *Multiprosessorer med distribuert felles hukommelse*[21] har hukommelsen fysisk distribuert



Figur 2.1: Parallelldatamaskin med (i) distribuert eller (ii) felles hukommelse (P=prosessor, H=hukommelse).

blant prosessorene, men likefullt et felles minneområde. Meldingsutveksling for tilgang til hukommelse som ikke ligger lokalt initieres av minneaksess og ikke som i MDH ved å kalle en systemfunksjon. Den lave forsinkelsen til lokal hukommelse øker skalerbarheten. Videre har hver prosessor flere nivå med hurtigminne for å minske forsinkelse mot den delen av hukommelsen som ikke er lokal.

Uavhengig av om man konstruerer en parallell datamaskin som en tradisjonell MDH, MFH eller som en eller annen hybrid, så vil et skreddersydd nettverk være et fordyrende element. For å bedre forholdet mellom pris og ytelse er det ønskelig å kunne ta i bruk mer generelle løsninger også for nettverket. En generell teknologi kan benyttes innen flere områder, produseres i større kvanta og vil dermed gjerne også bli billigere. Utfordringen ligger i det å kombinere det generelle med god ytelse. Utviklingen innen LAN på 90-tallet førte til at *nettverk av arbeidsstasjoner* (NAS) stod frem som et prisgunstig og svært fleksibelt alternativ til tradisonelle parallelle datamaskiner. Kommersielle svitsjer basert på både LAN- og SAN-teknologi[1] ble produsert hvor nettop NAS var en målgruppe [14, 15, 23, 28]. De senere år har vi sett en videre utvikling av generell SAN-teknologi som har god nok ytelse til å kunne benyttes i parallelle datamaskiner, lagrinsnettverk og andre beslektede bruksområder med tilsvarende krav til det underliggende nettverk. InfiniBand er et godt eksempel på en teknologi som er utviklet med tanke på generell SAN-bruk. Både NAS-svitsjer, InfiniBand og andre SAN-

[1] Vi skal være forsiktige med å skape inntrykk av et kategorisk skille mellom SAN- og LAN-teknologi. Selv om man gjerne sikter til forskjellige bruksområder med forskjellige krav til nettverket, er det også slik at de to typene nettverk overlapper hverandre, og at felles bruk av teknologiske prinsipper delvis visker ut markante forskjeller.

teknologier er imidlertid alle for nisje-produkter å regne sammenliknet med Ethernet. Selv 1Gb/s Ethernet er nå så billig å produsere at teknologien er på vei inn i de aller rimeligste personlige datamaskinene. Ethernets enorme utbredelse og dertil lave pris er hovedmotivasjonen bak det å se på Ethernet-teknologi som et alternativ ved konstruksjon av tett koblede nettverk.

2.2 Tjenestekvalitet

Når noder kommunisere med hverandre via et nettverk, er det alltid for å utveksle en eller annen form for informasjon. Nøyaktig hva slags informasjon det er snakk om vil ha innvirkning på hvilke krav man stiller til det nettverket man benytter. Begrepet *tjenestekvalitet*[44] henspiller på de ulike egenskaper man kan ønske fra nettverket. Det kan for eksempel være snakk om egenskaper som ønsket hastighet på linker og svitsjer, nettverkets *båndbredde*; krav relatert til hvor lang tid pakker bruker på å forflytte seg igjennom nettverket, *forsinkelse*; et ønske om begrenset spredning i forsinkelse, *jitter*; vurdering av hvor viktig det er å unngå pakketap, grad av *pålitelighet*; og hvorvidt det stilles strenge krav til *sikkerhet*. For eksempel vil en banktransaksjon stille strenge krav til pålitelighet og sikkerhet. Man ønsker at all informasjon kommer frem korrekt uten at andre kan blande seg inn i kommunikasjonen. Da er det antagelig mindre viktig om informasjonspakkene som tilhører transaksjonen kommer i rykk og napp; det er uproblematisk om kommunikasjonen skulle bli utsatt for jitter. Om man derimot skulle benytte et nettverk til videotelefoni er det andre krav som gjelder. Det viktigste vil da være at både forsinkelse og jitter er lave nok til at samtalen flyter godt. I tillegg ønsker man så stor båndbredde at kvaliteten på lyd og bilde oppleves som tilstrekkelig god. Disse kravene vil delvis overskygge kravet om pålitelighet. Dersom man skulle miste en pakke i nettverket, vil man foretrekke at man dropper denne pakken fra videostrømmen, fremfor at man venter på at pakken skal sendes på nytt fra avsender. En forringelse av kvaliteten på lyd og bilde et brøkdels sekund er som regel å foretrekke fremfor at samtalen for en liten periode fryser.

Kravene til tjenestekvalitet innen tett koblede nettverk kan variere noe avhengig av bruksområde, men felles for alle er at man gjerne ønsker så høy båndbredde, så lav forsinkelse og så stor grad av pålitelighet som mulig. Jo høyere båndbredde desto mer data kan nodene utveksle seg i mellom innen en gitt tid. Økt båndbredde kan også ha en positiv effekt på forsinkelsen, spesielt når det er mye trafikk i nettverket. Et nettverk med høy båndbredde kan håndtere større menger data raskere, og man kan til en viss grad forvente at dette også innebærer mindre venting for pakker på sin vei fra svitsj til svitsj. Mindre venting gir mindre forsinkelse. Forsinkelsen består imidlertid også av en komponent det er vanskelig å gjøre noe med. Signaler bruker tid på å forplante seg. Selv med lysets hastighet tar det tid å forflytte seg fra en

node til en annen.

Trafikken i et tett koblet nettverk er som regel av en slik art at det å kaste pakker er lite heldig. En pakke som for eksempel inneholder resultatet fra en beregning, eller deler av en fil, er høyst sannsynlig kritisk i den forstand at pakken må sendes på nytt dersom den kastes i nettverket. Vi ønsker derfor høy grad av pålitelighet. Dersom vi tvinges til å kaste en pakke er dette uheldig på flere måter. Siden pakken må sendes på nytt vil noden som venter på pakkens innhold oppleve økt forsinkelse. Samtidig vil den pakken som blir kastet faktisk ha lagt beslag på ressurser i nettverket helt til den kastes. Økt forsinkelse og bortkastede ressurser søker vi å unngå. Merk at det i utgangspunktet er umulig å sikre at pakker aldri kastes. Dersom feil oppstår i en pakke vil man være nødt til å sende den på nytt. Vi kan imidlertid unngå å kaste pakker unødvendig. I den forbindelse er det vesentlig at vi benytter oss av en form for flytkontroll som minimerer sjansene for pakketap på grunn av fulle buffer.

Andre former for tjenestekvalitet, som vurdering av behov for sikkerhet eller pålitelighet vurdert ut fra det tett koblede nettverkets oppetid, har vi valgt å ikke komme nærmere inn på siden de ikke er knyttet til de problemstillinger vi søker å besvare. Vi skal imidlertid se at våre rutealgoritmer vil sørge for at de pakker som sendes fra en node til en annen, og faktisk kommer frem, vil komme frem i den rekkefølge de ble sendt. Denne egenskapen kan man også anse å være en form for tjenestekvalitet.

2.3 Karakteristikk av tett koblede nettverk

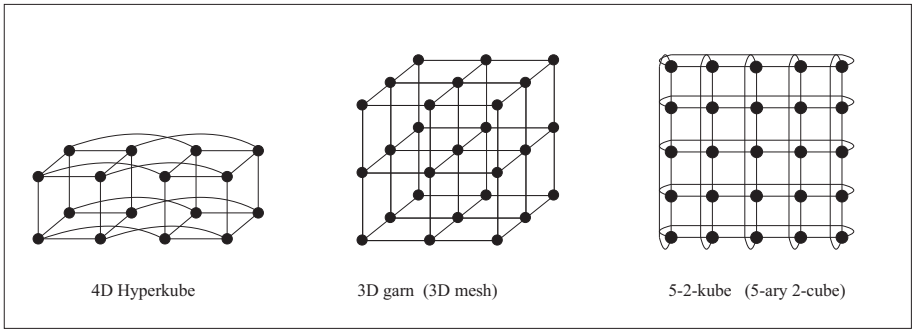
Tett koblede nettverk kan karakteriseres ved hjelp av fire egenskaper: *topologi*, *ruting*, *flytkontroll* og *svitsjing*.[2]

2.3.1 Topologi

Et nettverks topologi beskriver hvordan de forskjellige nodene er koblet sammen. Siden vi ikke skal forholde oss til kringkastingsnettverk tar dette delkapittelet kun for seg topologier knyttet til punkt-til-punkt-nettverk.

I et punkt-til-punkt-nettverk kobler man ved hjelp av linker noders utganger til andre noders innganger. Slik avgjøres nettverkets topologi på en entydig måte. Siden linker gjerne er toveis kan et tilkoblingspunkt hos en node fungere som både en inngang og en utgang. Som vi tidligere har vært inne på modellerer man gjerne topologien til et nettverk ved hjelp av en graf.

[2] I litteraturen vil man finne at enkelte velger å se på svitsjing og ulike typer svitsjeteknikker som en del av et utvidet flytkontroll-begrep[19]. Andre vurderer i større grad svitsjing som en egen karakter ved et nettverk[37, 42]. I denne oppgaven har vi valgt en forholdsvis snever definisjon av flytkontroll. Flytkontroll og svitsjing omhandles derfor i hvert sitt delkapittel.



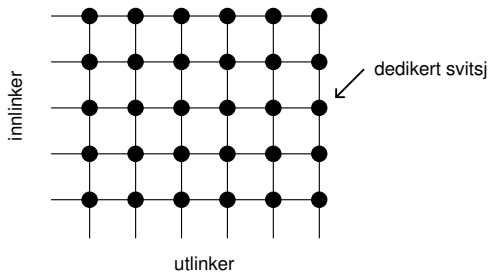
Figur 2.2: Tre regulære topologier for direkte nettverk.

Å gi alle noder direkte kobling til alle andre noder er i de fleste tilfeller lite hensiktsmessig, eller umulig, grunnet økonomiske og fysiske begrensninger. Vi benytter oss derfor som regel av en topologi som ikke er *komplett*, til tross for at dette vil medføre et behov for videresending av pakker. Uten videresendingsfunksjonalitet vil vi ikke kunne oppnå alle-til-alle-kommunikasjon i en ikke-komplett topologi.

Tett koblede nettverk kan grovt sett deles inn i *direkte* og *indirekte nettverk*, *regulære* og *irregulære topologier*. Direkte nettverk refererer til topologier hvor alle nodene er prosesseringsnoder. Nodene når hverandre ved å gå via null eller flere mellomliggende noder. For at dette skal fungere effektivt må hver node ha en dedikert svitsje-enhet som tar seg av videresending av pakker som ikke er tiltenkt noden selv. Tre vanlige topologier for direkte nett er illustrert ved figur 2.2.

Indirekte nettverk refererer til topologier hvor alle prosesseringsnoder er koblet sammen via mellomliggende, dedikerte svitsjer. Prosesseringsnodene selv har ikke videresendingsfunksjonalitet. Derfor kaller vi dem også gjerne endenoder. Ingen endenode er koblet til en annen endenode uten via en eller flere svitsjer. Sammenliknet med direkte nettverk finner vi altså et mer markant skille mellom svitsjer og endenoder i et indirekte nettverk. Figur 2.3 viser et indirekte nettverk. Endenoder kobles til inn- og utlinker, mens de interne svitsjene utgjør det man på engelsk kaller en *crossbar*.

Vi skal i denne oppgaven arbeide med nettverk som består av endenoder koblet sammen ved hjelp av dedikerte svitsjer. I utgangspunktet ville vi kalle et slikt nettverk for et indirekte nettverk. I våre nettverk vil imidlertid en svitsj alltid ha minimum én endenode koblet direkte til seg. Dermed kan vi se på hver svitsj med tilhørende endenoder som en prosseseringsnode med svitsjefunksjonalitet. Våre nettverk kan med andre ord sees på som både direkte eller indirekte nettverk, avhengig av ønsket abstraksjonsnivå. Vi vil derfor heller ikke eksplisitt markere endenoder i de grafer vi bruker for å representere ulike nettverk, men la det være gitt at hver node representerer



Figur 2.3: Et indirekte nettverk i form av en crossbar. Endenoder kobles til inn- og utlinker.

---

en svitsj og minimum én endenode. I kapittel 4 hvor vi vurderer resultater fra ulike simuleringer, vil det alltid gå klart frem hvor mange endenoder som er koblet til hver enkelt svitsj.

Ved bruk av regulære topologier foreligger det alltid rigide regler for hvordan man tillater at noder kobles sammen. Det kan for eksempel være snakk om at alle svitsjer skal ha det samme antall linker eller at topologien skal ta form av et binærtre[21]. Et mye brukt sett av regulære topologier krever at alle noder kan ordnes i forhold til et ortogonalt n-dimensjonalt rom, hvor hver link tilsvarer forflytning innenfor én dimensjon. De tre topologiene i figur 2.2 er alle ortogonale, regulære topologier.

Når man forholder seg til irregulære topologier, finnes det ingen tilsvarende regler for hvordan nodene kobles sammen. Et irregulært nettverk kan med andre ord ha en vilkårlig topologi. Dette medfører stor grad av fleksibilitet, men bringer samtdig med seg ekstra utfordringer. Ved bruk av en regulær topologi kan man utnytte kunnskap om topologien når man for eksempel utformer rutealgoritmen man ønsker å benytte. Noe tilsvarende er ikke på samme måte mulig når man arbeider med irregulære topologier. Irregulære topologier krever generelle løsninger. Om vi setter sammen en tilfeldig irregulær topologi kan denne sammenfalle med en regulær topologi. Normalt vil vi da faktisk referere til topologien som regulær, men dette illustrerer samtidig hvordan en regulær topologi kan sees på som et spesialtilfelle av en generell irregulær topologi. Løsninger utviklet for irregulære topologier vil derfor også kunne brukes for regulære topologier. Det motsatte er generelt ikke tilfelle.

I denne oppgaven skal vi forholde oss til irregulære topologier. Ethernet stiller ingen krav til hvordan nodene i nettverket kobles sammen. Denne egenskapen ønsker vi å beholde om vi skal benytte Ethernet ved konstruksjon av tett koblede nettverk. Irregularitet medfører fleksibilitet, skalerbarhet og god støtte for inkrementell utvidbarhet av et nettverk. Om man ønsker å

utvide et irregulært nettverk, har man stor frihet til å velge antall nye noder og linker i henhold til gjeldende behov. I et regulært nettverk må en utvidelse skje uten at man bryter de regler for sammenkobling som nettverket er basert på. Merk også at om man mister en link eller en node innen en regulær topologi, kan dette føre til at nettverket nå tar form av en irregulær topologi. En slik situasjon medfører en ekstra utfordring dersom man har utnyttet topologiens egenskaper ved utvikling av benyttet rutealgoritme. Noe tilsvarende vil ikke på samme måte være et problem i en irregulær topologi, siden rutealgoritmen i utgangspunktet tar høyde for at topologien er vilkårlig.

2.3.2 Ruting

Som vi var inne på i kapittel 1.3.1 handler ruting om å finne vei gjennom et nettverk med en ikke-komplett topologi slik at noder som ikke er koblet sammen ved hjelp av en direkte link likevel kan kommunisere med hverandre. Rutealgoritmen består av et sett med regler som entydig definerer hvilke lovlige ruter en pakke kan benytte på sin vei fra kilde til destinasjon.

I litteraturen vil man finne foreslått et stort antall rutealgoritmer[21]. Mange av rutealgoritmene er imidletid knyttet til regulære topologier. For eksempel kan vi nevne *dimension-order*-familien[21] av rutealgoritmer for ortogonale topologier. Her rutes pakker i henhold til én og én dimensjon, etter en streng ordning av dimensjonene. En annen familie rutealgoritmer som gjerne benyttes for regulære topologierer baserer seg på *the turn model*[21]. Igjen ordnes linkene inn i dimensjoner. Overgangen fra en dimensjon til en annen refereres til som en *turn*, altså en sving. I utgangspunktet tenker man seg gjerne at alle svinger er tillatt, før man så fjerner et sett med lovlige svinger for å hindre at *vranglås*[3] kan oppstå i nettverket. Siden vi har valgt å forholde oss til irregulære topologier kan vi naturlig nok ikke benytte oss av rutealgoritmer skreddersydd for regulære nettverk. Vi skal derfor heller ikke komme nærmere inn på flere slike algoritmer.

Rutealgoritmer kan deles inn i *deterministiske* og *adaptive* algoritmer. I tillegg skiller vi mellom *sentraliserte*, *distribuerte* og *kilderuting*-baserte rutealgoritmer (i tillegg til hybrider av disse klassene). Merk at selv om vi nå kun skal forholde oss til rutealgoritmer for irregulære topologier så gjelder de samme inndelingene også for algoritmer beregnet på regulære topologier.

En deterministisk rutealgoritme vil alltid tilordne en og samme rute for alle pakker som sendes fra en gitt kilde til en gitt destinasjon. Dette medfører at det for en pakke som ankommer en svitsj alltid kun finnes én gyldig vei videre. En adaptiv rutealgoritme tillater på sin side bruk av flere ruter. Adaptive rutealgoritmer er fleksible i den forstand at de kan ta hensyn til nettverkets tilstand, og benytte alternative veier dersom områder av nettverket er tungt belastet eller ute av funksjon.

[3] Vi skal forklare begrepet vranglås i kapittel 2.4.

Sentralisert ruting baserer seg på at en sentral enhet i et nettverk tar alle ruteavgjørelser. En slik tilnærming til ruting er uvanlig for irregulære topologier[21]. Den sentrale enheten som skal stå for alle ruteavgjørelser vil potensielt utgjøre en flaskehals i nettverket.

Ved bruk av kilderuting avgjør kilden selv den ruten en pakke skal ta gjennom nettverket. Hver pakke inneholder en komplett veibeskrivelse som hver svitsj langs veien forholder seg til. Myrinet[15] benytter seg av kilderuting og har støtte for irregulære topologier. Dersom vi skulle bruke kilderuting for Ethernet ville dette imidlertid kreve at pakke-formatet til Ethernet ble utvidet til å kunne inneholde en komplett veibeskrivelse. Siden vi ønsker å beholde Ethernets pakkeformat er ikke kilderuting et alternativ for oss.

Ved distribuert ruting avgjør hver enkelt svitsj veien videre for pakker som ankommer svitsjen. Gyldige ruter lagres i en rutetabell slik at svitsjen til en hver tid kan slå opp i tabellen når pakker skal videresendes. Rutetabellene kan enten fylles manuelt eller som en del av et nettverks automatiske konfigurasjon. Distribuert ruting og bruk av rutetabeller er kompatibelt med Ethernets virkemåte, og blir da et naturlig valg for oss når vi skal se på Ethernet ved konstruksjon av et tett koblet nettverk.

Vi skal i tillegg holde oss til deterministiske rutealgoritmer. Deterministisk ruting er enklere enn adaptiv ruting og gir derfor gjerne en enklere og raskere svitsje-arkitektur[21]. I kapittel 2.6 skal vi se at Ethernet er av determinitisk natur; det finnes kun én rute som fører frem til en pakkes destinasjon. Ved å holde oss til deterministisk ruting søker vi å minimere antall nødvendige forandringer av Ethernet-svitsjer tiltenkt tett koblede nettverk. Deterministisk ruting sikrer også at pakker som sendes fra en kilde til en gitt destinasjon kommer frem i samme rekkefølge som de blir sendt. Det er da en forutsetning at pakker ikke har blitt kastet, og at alle buffer i mellomliggende svitsjer følger først-inn-først-ut-prinsippet. *Kontinuerlig omruting* er heller ikke et problem når man benytter seg av deterministisk ruting. Vi skal forklare begrepet kontinuerlig omruting sammen med vranglås i kapittel 2.4[4].

En av de mest kjente distribuerte rutealgoritmene for irregulære nettverk er *up*/down*-ruting*[14]. Rutealgoritmen lar seg benytte sammen med Ethernet ved at man tar kontroll over Ethernet-svitsjenes rutetabeller. I motsetning til Ethernet tillater up*/down*-ruting at man benytter seg av alle linker i nettverket. Up*/down*-ruting står derfor frem som en naturlig kandidat for å forbedre Ethernets ytelse som tett koblet nettverk. Vi har i tillegg valgt oss en tredje distribuert rutealgoritme, *korteste-vei-ruting*, for å sammenlinke med de to foregående. Korteste-vei-ruting sender alltid en pakke korteste vei gjennom nettverket. En slik tilnærming virker tilsynelat-

[4]Håndtering av vranglås i vår simulator forenkles også betraktelig av at vi holder oss til deterministisk ruting. Vi skal imidlertid ikke la dette stå som et argument for hvorfor vi ønsker å benytte oss av deterministisk ruting.

ende forlokkende, men vi skal se i kapittel 2.4 at den samtidig fører meg seg problemer som gjør den mindre egnet for reelle nettverk.

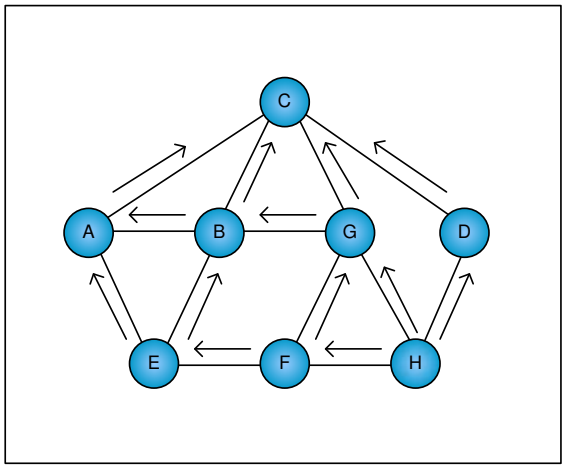
La oss nå gjennomgå de to rutealgoritmene up*/down*- og korteste-vei-ruting. Ethernet vil vi behandle i kapittel 2.6. Vi skal se at både up*/down*- og korteste-vei-ruting i utgangspunktet er adaptive. Siden vi ønsker å forholde oss til deterministiske rutealgoritmer, gjennomgår vi også noen små forandringer for å transformere de to algoritmene til deterministiske rutealgoritmer.

Up*/down*-ruting

Up*/down*-ruting ble intodusert som en del av Autonet[14]. Ved bruk av denne rutealgoritmen gjennomgår nettverket først en konfigurasjonsfase hvor svitsjene beregner et felles *bredde-først-spenntre*[50]. Som en del av dette blir alle svitsjene i nettverket enige om en felles rot. Hver link i nettverket blir så tilordnet en retning. La avstanden fra en node N til roten måles som det minste antallet linker man må følge fra N for å nå roten. En link mellom to noder N_1 og N_2 har retning fra N_2 til N_1 dersom N_1 ligger nærmere roten enn N_2. Om det motsatte skulle være tilfelle vil linken ha motsatt retning. Dersom de to nodene N_1 og N_2 i nettverket har samme avstand fra roten, vil linken mellom dem få retning fra noden med høyest ID til noden med lavest. En pakke som sendes over en link i samme retning som linken sies å benytte linken i opp-retningen. Dersom en pakke sendes i motsatt retning av linken benytter den linken i ned-retningen. Pakker i nettverket kan nå rutes fritt i nettverket så lenge up*/down*-invarianten overholdes: *En lovlig rute fra kilde til destinasjon medfører traversering av null eller flere linker i opp-retningen, etterfulgt av null eller flere linker i ned-retningen.* En pakke har altså ikke anledning til å forflytte seg "oppover" etter at den har forflyttet seg "nedover". Merk at denne begrensningen aldri forhindrer to noder som er koblet direkte til hverandre i å benytte linken imellom seg når de skal kommunisere. Siden en pakke da kun traverserer én link spiller ikke retningen på linken noen rolle.

Figur 2.4 viser et nettverk hvor svitsjene benytter up*/down*-ruting. Noden C har blitt valgt som rot. Noder med samme avstand til roten ligger på samme horisontale nivå. Pilene viser hvilken retning hver enkelt link har fått. Merk hvordan det eksisterer mange lovlige ruter fra for eksempel noden H til A. For å oppnå en deterministisk utgave av up*/down* må vi begrense antallet ruter i nettverket til én per kilde/destinasjons-par. Man kan tenke seg dette gjort på flere måter. Vi har valgt to tilnærminger som resulterer i hver sin deterministiske utgave av up*/down*.

Den første og enkleste utgaven har vi valgt å kalle *deterministisk up*/down** (DUD). En DUD-svitsj S bygger opp rutetabellen sin ved å følge følgende prinsipp: Dersom det finnes en eller flere ruter til destinasjonen D som kun benytter linker med ned-retning, legger man den korteste



Figur 2.4: Up*/down*-ruting. Pilene viser linkenes retning.

inn i rutetabellen. Finnes det flere ruter som er like korte velges en av dem. Dersom det ikke finnes noen rute til D som kun bruker linker i ned-retning, legger man inn i rutetabellen den korteste ruten fra S til roten. Finnes det flere ruter som er like korte velges en av dem. Merk at dette ikke nødvendigvis betyr at en pakke må sendes helt opp til roten for å komme frem til destinasjonen D. Neste svitsj pakken kommer til etter å ha blitt sendt over en link i opp-retning kan ha en rute til D som kun består av linker i ned-retning. Om så er tilfelle vil denne bli brukt videre. Uavhengig av om ruten videre fra S til D starter med en link i opp- eller ned-retning vil kun én rute legges inn i rutetabellen for hver enkelt destinasjon. Dermed sikrer man at DUD holdes deterministisk. Merk at det alltid finnes en lovlig rute som kun benytter linker i ned-retning fra roten til alle andre noder. Dette sikrer at alle noder i nettverket kan snakke med alle andre noder: om man ikke finner en annen og bedre rute, vil pakker sendes oppover til de når roten og derfra ned til rett destinasjon.

Den andre varianten av up*/down*-ruting vi skal benytte har vi valgt å kalle *deterministisk korteste-vei-up*/down** (DKVUD). En DKVUD-svitsj S vil, for alle destinasjoner D, beregne korteste rute fra S til D som består av null eller flere linker i opp-retning, etterfulgt av null eller flere linker i ned-retning. Korteste rute for hver enkelt destinasjon legges så inn i rutetabellen. Finnes det flere ruter som alle er like korte velges én av dem. Her må man imidlertid være litt forsiktig. DKVUD er optimalisert for alltid å velge korteste vei uten å bryte up*/down*-invarianten. Man må med andre ord passe på at en pakke som kommer inn til S via en link i ned-retning ikke sendes

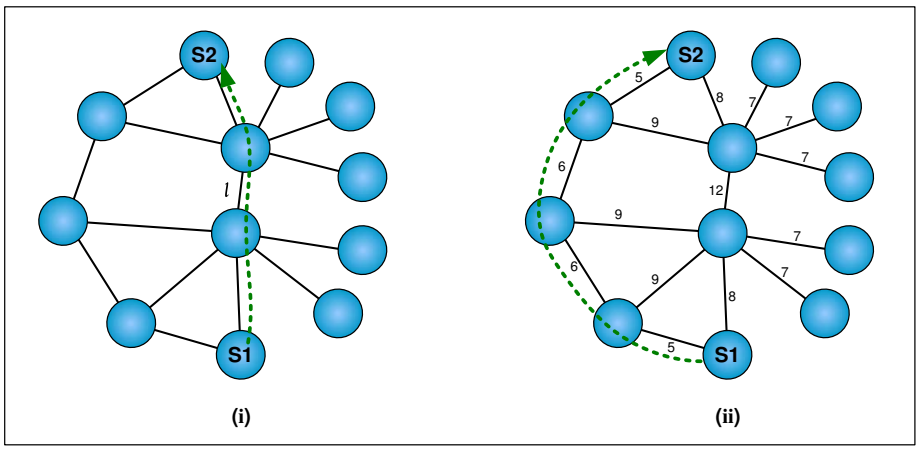
videre ut på en link i opp-retning. Spørsmålet blir da om man ved bruk av DKVUD kan oppleve følgende: La ruten R være korteste up*/down*-vei fra svitsjen S til destinasjonen D. La S' være en svitsj som ligger et sted mellom S og D langs ruten R. Er det da mulig at ruten R fra og med linken inn til S' (eller tidligere) kun består av linker i ned-retning, mens korteste rute fra S' til D faktisk starter med en eller flere linker i opp-retning? Svaret på dette spørsmålet er dessverre ja. Med andre ord vil DKVUD slik den er beskrevet ovenfor kunne føre til at up*/down*-invarianten brytes. For å rette på dette er det nødvendig at man i rutetabellen til en svitsj S ikke bare legger inn neste steg for korteste rute fra S til D, men også legger inn neste steg for en eventuell rute som kun bruker linker i ned-retningen. Om dette gjøres kan S, ved å ta hensyn til retningen på den linken en pakke kommer inn via, videresende pakken uten å bryte up*/down*-invarianten. Merk at dette ikke bryter med at DKVUD skal være en deterministisk rutealgoritme. Pakker som sendes fra kilden K til destinasjonen D vil alltid velge den samme ruten, til tross for at en svitsj S på veien tilsynelatende har to gyldige steg videre. Dette skyldes at pakkene fra D alltid vil komme inn til S via den samme linken. Dermed er veien videre også entydig bestemt. Merk også at DKVUD på samme måte som DUD tillater kommunikasjon mellom alle nodene i nettverket. Om ingen bedre vei finnes vil man sende pakker opp til roten og videre ned til rett destinasjon.

Mens DKVUD potensielt sett er en mer effektiv rutealgoritme ved at metoden er flinkere enn DUD til å finne korteste vei mellom kilde og destinasjon, vil metoden også legge beslag på større ressurser både under konfigurasjonsfasen, samt for hvert ruteoppslag. DKVUD vil også kreve at en eventuell Ethernet-svitsj ved ruteoppslag er i stand til å ta hensyn til linken en pakke kommer inn via. De to deterministiske implementasjonenene av up*/down* har med andre ord begge fordeler og ulemper.

Korteste-vei-ruting

Hovedidéen bak korteste-vei-ruting er svært enkel: Hver pakke som sendes mellom to noder benytter seg av den korteste ruten mellom de to nodene. Nøyaktig hvordan man definerer hvilken rute som er kortest vil kunne variere fra rutealgoritme til rutealgoritme. Dersom det er mer enn én rute som kan sies å være kortest, velger man gjerne den som ut fra nettverkets tilstand tilsynelatende er det beste valget. Med andre ord er korteste-vei-ruting i utgangspunktet en familie med adaptive rutealgoritmer. Vi skal begrense våre korteste-vei-algoritmer ved at vi alltid kun legger inn én rute i rutetabellen for hvert kilde/destinasjons-par, selv de gangene hvor det finnes mer enn én korteste vei. Dermed vil våre korteste-vei-algoritmer holdes deterministiske.

Den enkleste formen for korteste-vei-ruting ser på alle linker i nettverket som et hopp mellom to svitsjer. Alle hopp anses å være likeverdige. Korteste rute fra en svitsj S til en destinasjon D er da den ruten som består av



Figur 2.5: Figuren viser hvordan SKV og VKV, illustrert ved henholdsvis (i) og (ii), har forskjellig oppfatning av hva som er korteste vei fra S1 til S2.

---

færrest hopp. Er det flere enn én rute som kan sies å være kortest, velges en av dem. Denne formen for korteste-vei-ruting skal vi referere til som *simpel korteste-vei-ruting* (SKV).

Ved bruk av SKV står man i fare for å utsette en eller noen få linker i nettverket for svært mye trafikk. Dette skyldes at en link kan være del av korteste-vei for relativt mange ruter gjennom nettverket. Figur 2.5(i) illustrerer dette. Her er link l spesielt utsatt. En stor andel av korteste-vei-rutene i nettverket vil passere l, noe som lett fører til at l blir en flaskehals. Mer konkret ser vi at veien fra svitsj S1 til svitsj S2 går via l. Kan det tenkes at denne veien, til tross for at den inneholder færrest hopp, likevel ikke er den beste, nettopp fordi l er tungt belastet? Vi ser at det finnes alternative veier fra S1 til S2, men hvilken av disse skal man velge om man ikke ønsker å bruke linken l?

I et forsøk på å avlaste link l i figur 2.5(i), og dermed bedre balansere trafikken i nettverket, introduserer vi en alternativ algoritme for korteste-vei-ruting: *vektet korteste-vei-ruting* (VKV). Ved bruk av VKV tilordnes alle linker i nettverket en vekt. Tilordningen skjer på følgende måte: La antall linker knyttet til en node N, gjerne kalt *graden* til N, være gitt ved $g(N)$. Vekten v til linken $l_{N_1N_2}$ som forbinder de to nodene N_1 og N_2 er gitt ved summen av graden til de to nodene:

$$v(l_{N_1N_2}) = g(N_1) + g(N_2) \tag{2.1}$$

Videre definerer vi vekten v_r til ruten R som summen av vekten til de linker

som utgjør ruten:

$$v_r(R) = \sum_{l \in R} v(l) \tag{2.2}$$

Den korteste ruten R_{min} fra en svitsj S til en destinasjon D defineres nå i VKV som den ruten som har lavest vekt:

$$R_{min} = min(v_r(R')), R' \in \Re \tag{2.3}$$

hvor $\Re$ er mengden av alle mulige ruter fra S til D. Er det flere enn én rute som kan sies å være kortest velger vi en av dem for å oppnå en deterministisk rutealgoritme.

Figur 2.5(ii) illustrerer effekten VKV kan ha på et nettverk. Alle linkene i nettverket har blitt vektet av funksjonen (2.1). Legg spesielt merket til at linken l fra (i) nå har fått vekten 12 og dermed er den tyngste linken i nettverket. Dette har ført til at trafikken fra svitsj S1 til S2 ved bruk av VKV vil følge en rute som ikke inneholder linken l. Det vil fortsatt være mange ruter som bruker linken l, men vektingen kan potensielt sett gjøre at trafikken blir bedre fordelt i nettverket.

2.3.3 Flytkontroll

Generelt definerer flytkontroll hvordan man skal administrere ressurser i et nettverk[19]. For å oppnå effektiv kommunikasjon mellom to noder i et tett koblet nettverk innebærer dette styring av hvor raskt en avsender kan sende pakker til mottager uten at mottager, eller eventuelt mellomliggende svitsjer, får problemer med å håndtere trafikken.

Flytkontroll kan utøves på flere plan. For eksempel kan endenoder i et nettverk benytte seg av flytkontroll seg imellom, uavhengig av mellomliggende svitsjer. De to endenodene benytter seg av *ende-til-ende-flytkontroll*. Samtidig kan noder i et nettverk benytte seg av flytkontroll som opererer på link-nivå. Her benyttes flytkontrollen til å påse at to noder koblet sammen ved hjelp av en direkte link ikke sender pakker til hverandre raskere enn det motparten klarer å håndtere. En slik form for flytkontroll kaller vi gjerne for *punkt-til-punkt-flytkontroll*. Vi skal i denne oppgaven forholde oss til flytkontroll på link-nivå, uten ytterligere å vurdere ende-til-ende-flytkontroll. Før vi går videre kan vi imidlertid merke oss at det finnes forskning som tyder på at en kombinasjon av flytkontroll på flere nivå kan ha en gunstig effekt på trafikken i et nettverk[11, 49].

Som vi var inne på i kapittel 1.3.1 har vi behov for punkt-til-punkt-flytkontroll i et tett koblet nettverk for å minimere sjansen for at pakker kastes. En mottager av trafikk kontrollerer da bruk av link og tilhørende innbuffer ved å sende kontrollpakker til avsender. Skal flytkontrollen i én retning av en toveis-link være uavhengig av flytkontrollen i motsatt retningen over samme linken, er man avhengig av at flytkontrollpakker ikke selv

utsettes for flytkontroll. Nøyaktig hva flytkontrollpakkene inneholder vil avhenge av den spesifikke flytkontroll-mekanismen som er i bruk.

Kredittbasert flytkontroll er mye brukt innen SAN. En mottager av trafikk gir avsender en kreditt som tilsier hvor mange pakker avsender får lov til å sende. Hver gang avsender sender en pakke senkes kreditten. Når kreditten når null, må avsender vente med å sende nye pakker til mottager har gitt ny kreditt. Kreditten hos avsender økes enten ved at mottager eksplisitt sender over en ny kreditt, eller ved at mottager på annen måte gir beskjed om at pakker avsender tidligere har sendt ikke lengre opptar ressurser hos mottager. Både Fibre Channel og InfiniBand er eksempler på teknologier som benytter kredittbasert flytkontroll.

For Ethernet finnes det ingen spesifisert standard for kredittbasert flytkontroll. Ethernet har på sin side fått støtte for *på/av-flytkontroll* gjennom IEEE 802.3x[5, 42]. Ved bruk av på/av-flytkontroll vil en eier av et innbuffer, altså mottager av trafikk, underrette avsender i andre enden av korresponderende link om når det er tillatt og benytte linken. Dersom mottagers innbuffer er i ferd med å gå fullt, sender mottager en flytkontrollpakke til avsender med en beskjed om at trafikk på linken i retning fra avsender til mottager må stoppes. Etter hvert som innbufferet tømmes igjen må mottager vurdere å gi en ny beskjed til avsender om at det på nytt er tillat å benytte linken. Slik styrer mottager flytkontroll over en link ut fra tilstand i korresponderende innbuffer. Merk at selv om flytkontrollen for hver link ene og alene styres ut fra tilstanden til tilhørende innbuffer, vil aktivering av flytkontroll på en link indirekte kunne påvirke flytkontroll ved andre linker i nettverket. Dette skyldes følgende: Når en svitsj S på grunn av flytkontroll blir forhindret fra å sende pakker ut på en link vil tilhørende utbuffer gradvis kunne fylles. I det utbufferet går fullt vil ikke lengre pakker fra innbuffer hos S kunne flyttes over til dette utbufferet. Dermed vil S kunne oppleve at innbufferne gradvis fylles, noe som igjen kan føre til at flytkontroll aktiveres på tilhørende linker.

Siden Ethernet kun støtter link-basert på/av-flytkontroll er dette den formen for flytkontroll man må benytte dersom Ethernet-teknologi skal tas i bruk for tett koblede nettverk. Det er derfor interessant å se nærmere på denne typen flytkontroll, og når det kan være fornuftig å skru flytkontrollen på og av.

Triggerverdier for på/av-flytkontroll

Det er opp til eier av et innbuffer å avgjøre når flytkontroll skal skrus på og av. Det er imidlertid viktig å huske på at disse avgjørelsene ikke vil ha effekt umiddelbart. La linken $l_{N_1N_2}$ bære trafikk mellom nodene N_1 og N_2. Dersom N_2 ønsker å sende en flytkontrollpakke til N_1 vil dette ta tid. Først skal pakken genereres hos N_2. Så må pakken vente på tilgang til $l_{N_1N_2}$ i retning N_1. Når linken er ledig må pakken forflyttes over til N_1, før N_1 til slutt reagerer i henhold til pakkens innhold. Denne tregheten i effektuering

av flytkontroll har implikasjoner både i forhold til når man skrur på og når man skrur av flytkontrollen. La oss anta at innbufferet reservert $l_{N_1N_2}$ hos N_2 er i ferd med å bli fullt. For at pakker ikke skal kastes må N_2 generere og sende en flytkontrollpakke mens det fortsatt er plass i innbufferet: Innbufferet må kunne ta i mot pakker ikke bare frem til N_1 mottar flytkontrollpakken, men faktisk helt frem til siste pakke N_1 startet å sende til N_2 før flytkontollpakken kom frem til N_1, har ankommet N_2. Når N_2 ved et senere tidspunkt ønsker å skru av flytkontrollen igjen, er det gunstig at dette gjøres så tidlig at innbufferet ikke tømmes i påvente av at N_1 skal sende mer trafikk. Et tomt innbuffer indikerer i denne sammenhengen at vi har holdt igjen linken unødvendig lenge. N_2 må generere og sende en flytkontrollpakke så tidlig at N_2 fortsatt har nok pakker i innbufferet til at det ikke tømmes i påvente av at N_1 mottar flytkontrollpakken, gjenopptar bruk av $l_{N_1N_2}$, og at første nye pakke fra N_1 kommer frem til N_2 (forutsatt at N_1 faktisk har noe å sende).

La oss anta at vi skrur flytkontrollen på i det bufferet fylles opp over en grense $t_{på}$, mens vi skrur den av igjen når bufferet blir tømt under en grense t_{av}. $t_{på}$ og t_{av} fungerer som triggere for flytkontrollen. Størrelsen på svitsjenes innbuffer vil naturlig nok diktere hvilke verdier vi kan benytte for $t_{på}$ og t_{av}. Små innbuffer gir liten frihet til å sette de to triggerverdiene, mens store innbuffer gir større frihet. For en gitt bufferstørrelse vil forrige avsnitt gi oss et grunnlag for å sette en øvre grenser for $t_{på}$, t_{maks}, og en nedre grense for t_{av}, t_{min}. Vi har imidlertid ingen tilsvarende nedre grense for $t_{på}$ eller øvre grense for t_{av}. For at flytkontrollen skal fungere som forventet er det imidlertid gitt at $t_{på} \geq t_{av}$.

Vi ønsker å studere hvilke implikasjon det har på ytelsen til nettverket dersom vi varierer $t_{på}$ og t_{av}, men slik at de holder seg innenfor det man må kunne anta er fornuftige yttergrenser: $t_{maks} \geq t_{på} \geq t_{av} \geq t_{min}$. Om $t_{på}$ holdes nær t_{maks} mens man varierer t_{av}, vil ytelsesmålinger kunne gi indikasjoner på hvor vidt det er gunstig at man skrur flytkontrollen raskt av igjen i det innbufferet får ledig plass, eller om man bør vente til innbufferet er nesten tomt. På den andre siden er det også interessant å se på hvordan trafikken i nettverket påvirkes av en varierende $t_{på}$ mens t_{av} holdes nær t_{min}. Merk at man etter hvert som man senker $t_{på}$ potensielt skrur på flytkontrollen så tidlig at innbufferet aldri fylles opp. Dersom deler av et innbuffer aldri benyttes, indikerer dette igjen at man kunne ha klart seg med et mindre innbuffer.

2.3.4 Svitsjing

I det en pakke ankommer en svitsj vil gjeldende rutealgoritme avgjøre hvilken link pakken skal sendes videre ut på. Rutealgoritmen sier imidlertid ingen ting om når eller hvordan pakken skal forflyttes internt i svitsjen. Det å forflytte en pakke gjennom svitsjens indre refererer vi til som *svitsjing*. Merk at dette medfører å flytte en pakke gjennom et eget internt nettverk i

svitsjen, et nettverk som knytter svitsjens innporter sammen med svitsjens utporter. Nøyaktig hvordan svitsjingen utføres bestemmes av den benyttede *svitsje-teknikken*[21]: Svitsje-teknikken avgjør når og hvordan svitsjens indre nettverk settes opp for å koble svitsjens innporter til svitsjens utporter, og når (deler av) en pakke skal flyttes igjennom det indre nettverket. For en svitsj med kantbuffer vil med andre ord valg av svitsje-teknikk avgjøre når og hvordan en pakke, helt eller delvis, forflyttes fra innbuffer til utbuffer gjennom svitsjens indre nettverk.

For denne oppgaven skal vi ikke diskutere nærmere hvordan svitsjens interne nettverk er konstruert. Vi forutsetter imidlertid at svitsjene er feilfrie. En pakke vil ikke feilaktig få endret sitt innhold på sin vei gjennom svitsjen, og den videresendes ut på korrekt link, bestemt av rutealgoritmen. I tillegg forutsetter vi at det interne nettverket er av en slik karakter at svitsjen kan forflytte pakker fra alle innbuffer parallelt gjennom svitsjens indre til korrekt utbuffer, uten at pakkestrømmene forstyrrer hverandre. Det er da en forutsetning at ingen av pakkene skal til det samme utbuffer. Et slikt *ikke-blokkerende nettverk*[21] kan for eksempel konstrueres ved hjelp av en crossbar (figur 2.3).

For irregulære topologier kan svitsje-teknikk deles inn i fire hovedtyper: *linje-svitsjing*, *pakke-svitsjing*, *virtual cut-through* og *wormhole-svitsjing*[21]. Ved bruk av linje-svitsjing reserveres en vei gjennom hele nettverket fra kilde til destinasjon før man starter sending av data. Slik sikrer man rask ruting for etterfølgende data, samt fravær av datablokkering siden ressurser er reservert på forhånd. En ulempe ved denne svitsje-teknikken er at det tar tid og ressurser å reservere en vei i gjennom nettverket. Det er også uheldig at en reservert vei legger beslag på ressurser i nettverket selv mens veien ikke er i bruk.

Ved bruk av pakke-svitsjing rutes og sendes data som selvstendige, uavhengige pakker gjennom nettverket. Hver svitsj vil ta i mot hele pakken før den sendes videre. Derfor refereres denne svitsje-teknikken ofte til som *lagre-og-videresend*, som er det begrepet vi benyttet i kapittel 1.3.1 da vi vurderte behovet for pakkelagring i svitsjer. Pakke-svitsjing krever mer bufferplass hos svitsjene enn linje-svitsjing, og medfører ekstra administrasjon både ved at en datastrøm må delse opp i pakker samt at hver pakke vil kreve individuell ruting hos hver svitsj den passerer i nettverket. På den andre siden vil pakke-svitsjing potensielt utnytte ressurser i nettverket bedre enn linje-svitsjing, siden man unngår at reserverte ressurser ligger ubrukt.

Virtual cut-through (VCT) har mye til felles med pakke-svitsjing. VCT utnytter imidlertid det faktum at man innen mange nettverksteknologier finner destinasjonsadressen til en pakke helt først i pakken, som en del av pakkens *hode*. Så fort en svitsj har tatt imot nok av en pakke til å vite destinasjonsadressen, kan svitsjen rute pakken og faktisk starte videresending av den før hele pakken har ankommet svitsjen. Dette reduserer forsinkelse til en pakke igjennom nettverket sett i forhold til pakke-svitsjing, forutsatt at

pakken ikke blokkeres hos svitsjene. Om en pakke blokkeres må den lagres i sin helhet hos blokkerende svitsj. Dersom en pakke blokkeres i hver svitsj gjennom nettverket, vil med andre ord VCT oppføres seg på samme måte som pakke-svitsjing.

Ved bruk av pakke-svitsjing eller virtual cut-through vil flytkontroll og bufferstørrelser være relatert til pakkestørrelse. Om en node som holder på å sende en pakke P over en link samtidig får beskjed om at flytkontroll på den aktuelle linken skal aktiveres, vil noden fortsatt fullføre sendingen av P. Man har ingen mekanisme for å holde igjen en pakke man har startet å sende. Dermed er det også naturlig at en mottager kan ta imot et antall hele pakker, noe som igjen kan kreve mye bufferplass hos svitsjen. For å redusere behovet for buffer-plass hos svitsjene kan man benytte seg av wormhole-svitsjing. Ved bruk av wormhole-svitsjing deles hver pakke inn i et antall *flitter*. Starten på pakken, altså en av de første flittene, inneholder fortsatt destinasjonsadressen slik at man kan starte videresending av en pakke så raskt som mulig. Den store forandringen ligger imidlertid i at man omdefinerer flytkontrollen til å arbeide på flitt-nivå. Dermed er det tilstrekkelig at en svitsj har buffer til å lagre et ønsket antall flitter. Når flytkontroll aktiveres, vil en pakke som sendes over den aktuelle linken stoppes ved neste fullførte flitt. Med buffer som tilsvarer et lite antall flitter vil en pakke bukte seg som en orm gjennom nettverket og dermed være spredd utover flere svitsjer og linker, også dersom pakken blokkeres. Ved hjelp av wormhole-svitsjing kan man oppnå lav forsinkelse gjennom nettverket samtidig som svitsjene kan holdes små og raske. Ved blokkering vil imidlertid en pakke kunne blokkere flere linker i nettverket samtidig. En slik uheldig situasjon kan delvis løses ved å dele hver link inn i et sett med *virtuelle kanaler*[5], da til en pris av økt kompleksitet i svitsjene.

Selv om wormhole-svitsjing har vært mye brukt innen tett koblede nettverk skal vi i denne oppgaven fokusere på nettverk som benytter lagre-og-videresend (LOV). Ethernet er nemlig en teknologi som baserer seg på nettopp LOV. Det finnes riktignok Ethernet-svitsjer som også baserer seg på virtual cut-through[42], men vi har likevel valgt å begrense oss til en av de to teknikkene. Som tidligere nevnt vil to nettverk som benytter seg av henholdsvis virtual cut-through og LOV oppføre seg mer og mer likt etter hvert som trafikken, og dermed sannsynligheten for blokkering, øker i et nettverk. Ved lite trafikk vil virtual cut-through ha tilnærmet de samme egenskaper som wormhole-svitsjing, noe som potensielt medfører lavere forsinkelse for pakker gjennom nettverket. Vi kan imidlertid merke oss at den del av en pakkes totale forsinkelse på sin vei gjennom nettverket som skyldes at pakker må lagres i sin helhet før videresending i LOV, vil minke etter hvert som båndbredden til linkene øker, forutsatt at man ikke øker pakkestørrelsen tilsvarende. Ved bruk av 10Mb/s Ethernet vil det ta $1,2ms$ å motta en pakke

[5]Vi gir en lett introdukson til virtuelle kanaler i kapittel 2.5

på 12000 bit[6], mens man ved bruk av 10Gb/s Ethernet kun vil bruke 1/1000 av tiden på å motta samme pakke, altså $1,2\mu s$. Dette står i kontrast til den delen av forsinkelse som er forårsaket av tiden det tar å flytte signaler over en link. Signalhastigheten er oppad begrenset av lysfarten, og påvirkes ikke av økt båndbredde.

2.4 Utsulting, kontinuerlig omruting og vranglås

Hovedoppgaven til et tett koblet nettverk er å sørge for at pakker som sendes ut i nettverket kommer trygt frem til rett destinasjon, fortrinnsvis med så liten forsinkelse som mulig. En forflytting av en pakke gjennom nettverket krever ressurser som kun finnes tilgjengelig i en begrenset mengde. Som en del av det å oppnå en hensiktsmessig, og ikke minst rettferdig, ressurserallokering i nettverket er det tre situsjoner man søker å unngå: *utsulting*, *kontinuerlig omruting* og *vranglås*. Alle tre situasjonene kan resultere i at pakker aldri kommer frem til sine destinasjoner.

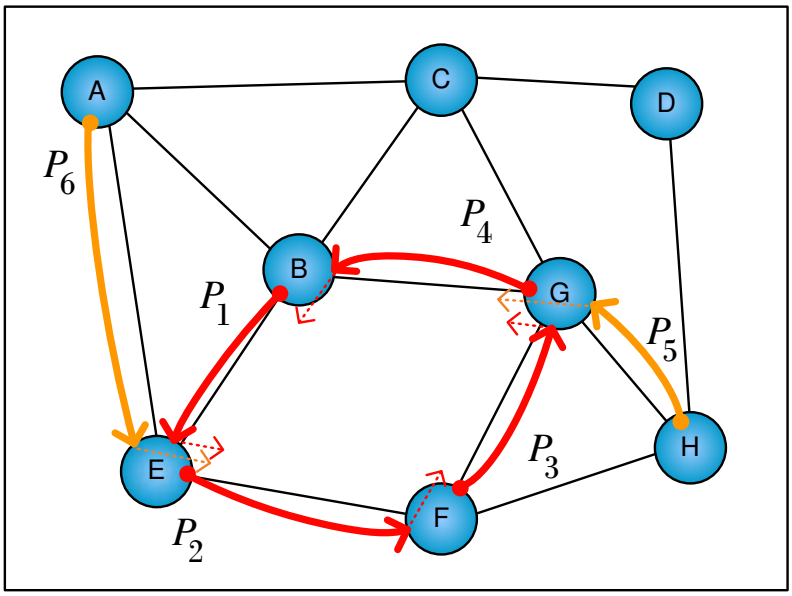
Utsulting oppstår i et nettverk dersom en pakke etterspør en ressurs, men alltid må vente fordi andre pakker som etterspør samme ressurs blir prioritert. For vår del vil utsulting kunne finne sted i en svitsj om en pakke først i et innbuffer aldri får tilgang til ønsket utbuffer fordi svitsjen alltid lar andre pakker som ønsker samme utbuffer gå forran. En slik situasjon kan vi unngå ved å påse at enheten i svitsjen som fordeler tilgang til utbuffer gjør dette på en rettferdig måte.

Kontinuerlig omruting oppstår i et nettverk når en pakke gang på gang blir sendt rundt på alternative ruter uten å noen gang komme frem til sin rette destinasjon. En slik situasjon kan oppstå ved bruk av adaptiv ruting dersom linker som fører frem til destinasjonen alltid er opptatt av andre pakker i det pakken utsatt for kontinuerlig omruting etterspør dem. Kontinuerlig omruting kan imidlertid aldri oppstå ved bruk av deterministiske rutealgoritmer, og er dermed heller ikke et problem for de nettverk vi arbeider med i denne oppgaven.

Den siste av det tre situasjonene man ønsker å unngå, vranglås, er ikke bare den som bringer med seg de største utfordringene, men det er også den eneste av de tre som kan føre til at all trafikk i hele nettverket stopper opp. La oss illustrere vranglås ved hjelp av et eksempel.

La oss nå anta at vi har et operativt nettverk som i figur 2.6. For enkelhets skyld antar vi at alle inn- og utbuffer har plass til én pakker hver, og at flytkontroll over linkene aktivers umiddelbart ved behov. Hver node i nettverket kan sende og motta egne pakker, samt videresende pakker for andre noder. Følgende fire situasjoner oppstår tilnærmet samtidig: B ønsker å sende pakker til F, E ønsker å sende pakker til G, F ønsker å sende pakker

[6] 12000 bit = 1500 byte, som igjen tilsvarer maksimal nyttelast for en tradisjonell Ethernet-pakke.



Figur 2.6: Fire linker har gått i vranglås i nettverket. To andre linker har indirekte blitt involvert.

---

til B og G ønsker å sende pakker til E. Vi antar at hver pakkestrøm er på minimum to pakker. Rutealgoritmen avgjør at pakkene skal sendes på følgende måte: Pakkestrømmen P_1 fra B til F skal sendes via E, pakkestrømmen P_2 fra E til G skal sendes via F, pakkestrømmen P_3 fra F til B skal sendes via G, og pakkestrømmen P_4 skal sendes fra G til E via B. Alle de fire nodene starter å sende pakker tilnærmet samtidig. La oss starte med å kikke på hva som skjer med pakker tilhørende P_1. Disse skal forflyttes via Bs utbuffer, over linken og til Es innbuffer, vist med rød pil merket P_1 i figur 2.6. Når første pakke fra P_1 når frem til E er imidlertid ønsket utbuffer opptatt av pakker fra pakkestrøm P_2. Første pakke fra P_1 blokkeres derfor i Es innbuffer i påvente av ledig utbuffer. Dette fører til at flytkontroll aktivers over linken mellom B og E, og at andre pakke i pakkestrømmen P_1 blir liggende i Bs utbuffer i påvente av at flytkontrollen skal skrus av igjen. Flytkontrollen vil på sin side være aktivert helt til første pakke fra P_1 kan forflyttes over til rett utbuffer hos E. I dette utbuffer ligger imidlertid andre pakke fra P_2 i påvente av at flytkontrollen over linken mellom E og F skal skrus av. Linken mellom E og F har nemlig også fått aktivert flytkontroll fordi første pakke i P_2 ligger blokkert i innbufferet hos F i påvente av ledig utbuffer, et utbuffer holdt av pakkestrøm P_3. Mens P_1 er blokkert av P_2, er altså P_2 blokkert av P_3. Dessverre har den samme situasjonen oppstått for P_3 og P_4. De er begge blokkert av henholdsvis pakkestømmene P_4 og P_1. Totalt har vi da at P_1

venter på P_2, P_2 venter på P_3, P_3 venter på P_4 og P_4 venter på P_1 igjen. Dette er illustrert i figur 2.6 hvor den stiplede fortsettelsen av en pil viser hvilken link en pakkestrøm ønskes videresendt ut på, og dermed også hvilken pakkestrøm man venter på. Siden ventingen er sirkulær er det ingen utsikter til progresjon for de fire pakkestrømmene; deler av nettverket har gått i vranglås. Vranglåsen vil etter hvert kunne spre seg til resten av nettverket i det pakker fra utsiden ønsker å benytte en av linkene som er blokkert. I figur 2.6 er dette illustrert ved at to nye pakkestrømmer, P_5 og P_6, indirekte har blitt involvert i vranglåsen. P_5 på vei til B har blitt stanset av P_4, mens P_6 på vei til F har blitt stanset av P_2.

Om man ukritisk tillater at enheter i et nettverk holder ressurser mens de etterspør nye, kan man oppleve vranglås slik vi så i eksempelet i forrige avsnitt. I det vranglås oppstår risikerer man at hele nettverket blir liggende uvirksomt. En slik situasjon er det naturlig nok essensielt at man unngår i ethvert nettverk. Merk at det var en forutsetning at nettverket benyttet seg av flytkontroll. Uten flytkontroll vil pakker kastes, noe som forhindrer vranglås i å oppstå.

Det er tre generelle tilnærminger til vranglåshåndtering[21]: Man kan *forhindre* vranglås, man kan *unngå* vranglås, eller man kan forsøke å *løse opp* vranglås etter at den har funnet sted. Ved forhindring av vranglås sørger man for at ressurser tilordnes pakker på en slik måte at en ressursforespørsel aldri kan føre til vranglås. Dette oppnår man typisk ved å reservere alle nødvendige ressurser gjennom nettverket før pakken sendes. For at selve reservesjonsforespørselen i seg selv ikke skal gå i vranglås er det viktig at man frigjør reserverte ressurser i nettverket dersom et forsøk på reservasjon ved en svitsj stopper opp. Linje-svitsjing håndterer gjerne vranglås ved forhindring.

Den andre tilnærmingen, å unngå vranglås, oppnår man ved at man tilordner ressurser til en pakke på dens vei gjennom nettverket, men kun på en slik måte at nettverket ikke kan gå i vranglås etter at den nye ressursen er tatt i bruk av pakken. For å forsikre seg om at nettverket ikke kan gå i vranglås etter en ressursallokering begrenser man typisk rutealgoritmen. En rutealgoritme som på denne måten unngår vranglås i et nettverk sier vi er en *vranglåsfri rutealgoritme*.

Den tredje måten å håndtere vranglås på er den mest optimistiske. Her forholder man seg til vranglås først etter at den eventuelt har oppstått. Dette krever da at man på en eller annen måte overvåker nettverket for å oppdage vranglås. Eventuelt kan man benytte seg av heuristikk for å vurdere om vranglås kan ha oppstått. Dersom en vranglås oppdages, eller man tror at nettverket kan ha gått i vranglås, løser man opp vranglåsen. Vranglåsen løses ved at man tar tak i en eller flere pakker direkte involvert i vranglåsen, og enten kaster dem eller sender dem ut langs alternative ruter i nettverket.

For oss er det de to siste tilnærmingene til vranglås som er av interesse. For å avgjøre om en (deterministisk) rutealgoritme er vranglåsfri for et gitt nettverk er det tilstrekkelig å kontrollere at tilhørende *kanalavhengighetsgraf*

ikke inneholder løkker[21]. Nodene i en kanalavhengighetsgraf representerer enveis-linker i det aktuelle nettverket[7]. En toveis-link splittes altså opp i to enveis-linker i kanalavhengighetsgrafen. Om de to nodene N_{L_1} og N_{L_2} representere linkene L_1 og L_2, så går det en rettet kant fra N_{L_1} til N_{L_2} i kanalavhengighetsgrafen dersom N_{L_1} *avhenger av* N_{L_2}. Litt forenklet kan vi si at N_{L_1} avhenger av N_{L_2} dersom en pakke kan holde igjen trafikk langs N_{L_1} mens den etterspør tilgang til N_{L_2}.

Up*/down*-ruting er vist å være en vranglåsfri rutealgoritme[14]. Ved å følge up*/down*-invarianten innfører man den nødvendige begrensning i rutealgoritmen for å unngå løkker i kanalavhengighetsgrafen, uavhengig av topologi. Vi har riktignok modifisert litt på up*/down* i det vi gjør rutealgoritmen deterministisk, men det å fjerne gyldige ruter fra en vranglåsfri rutealgortime kan ikke introdusere vranglås[21]. I Ethernet kan vranglås heller aldri oppstå siden nettverket i seg selv ikke tillater løkker. Da kan heller ikke en tilhørende kanalavhengighetsgraf inneholde løkker. Varianter av korteste-vei-ruting er derimot generelt utsatt for vranglås. Om vi vender tilbake til figur 2.6, kan vi observere at alle de brukte rutene kunne vært lovlige ruter innen simpel korteste-vei-ruting definert i kapittel 2.3.2. Ved implementasjon av korteste-vei-ruting må vi med andre ord holde øye med trafikken i nettverket for å avdekke og løse opp eventuell vranglås. Vi skal i kapittel 3.3.3 se hvordan vi ved hjelp av en *kanal-venter-på-graf* avdekker vranglås i nettverket, hvorpå vi for å løse opp vranglåsen kaster en eller flere av de pakkene som inngår i den.

2.5 Fysiske linker og virtuelle kanaler

I et tett koblet nettverk består de fysiske linkene gjerne av metalliske ledere eller optiske fibre. For de problemstillinger vi skal se på i denne oppgaven spiller det liten rolle nøyaktig hva slags linker som benyttes. Vi fortusetter imidlertid at linkene er feilfrie, slik at bit-feil ikke oppstår i det en pakke flyttes fra en node til en annen. Dette er for oss en akseptabel tilnærming til virkeligheten siden man for eksempel i Ethernet på bit-nivå gjerne opplever en feilrate i størrelsesorden $1 * 10^{-12}$[42]. Med en pakkestørrelse på 12000 bit tilsvarer dette at kun én av hundre millioner pakker inneholder feil påført av transmisjonsmediet.

En fysisk link kan være delt inn i et sett med *virtuelle kanaler*. De virtuelle kanalene har egne, private buffer, men deler den fysiske linken mellom seg. Ved et gitt tidspunkt er det derfor aldri mer enn én virtuell kanal som faktisk kan sende noe ut på linken. Tilgang til linken blir gitt på pakke- eller flitt-basis avhengig av benyttet svitsje-teknikk. Innføring av virtuelle kanaler fører til økt kompleksitet i svitsjene, men kan forsvares ved at de gir økt flek-

[7] Om man benytter seg av virtuelle kanaler i nettverket vil hver virtuelle kanal utgjøre en node i kanalavhengighetsgrafen. Virtuelle kanaler forklares i kapittel 2.5

sibilitet. Ved hjelp av virtuelle kanaler kan man minske ulempen medført av at en pakke blokkerer flere linker samtidig ved bruk av wormhole-svitsjing. Om en pakke blokkerer en virtuell kanal, kan andre virtuelle kanaler tilknyttet samme fysiske link fortsatt benytte linken for fullt. Virtuelle kanaler kan også benyttes til å kombinere flere rutealgoritmer slik at man oppnår mer effektiv bruk av nettverket eller unngår vranglås[21].

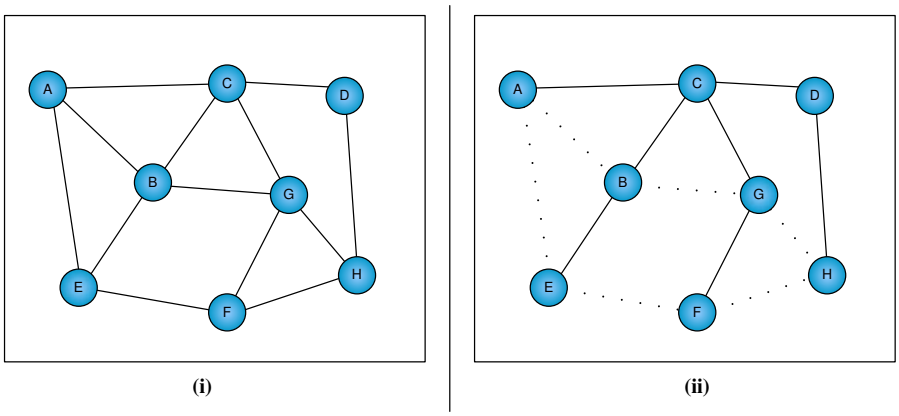
Vi har ikke behov for å benytte oss av virtuelle kanaler for å besvare våre problemstillinger, men siden simulatoren vi skal presentere i kapittel 3 har støtte for slike kanaler, er det likevel naturlig å introdusere begrepet.

2.6 Ethernet

Om noen nettverksteknologi kan sies å ha vokst inn i himmelen, så må det være Ethernet. Som vi alt har vært inne på har ikke Ethernet bare en enorm utbredelse innen lokalnett, men teknologien har også spredd seg til andre områder. En grundig gjennomgang av Ethernet er utenfor rekkevidde av denne oppgaven. Vi skal imidlertid gjennomgå tre sider ved Ethernet som alle er viktige sett i forhold til det å benytte Ethernet som teknologi for tett koblede nettverk. Vi skal gjennomgå hvordan Ethernet-svitsjer bygger opp rutetabellene sine, vi skal se hvordan Ethernet legger linker uvirksomme for å unngå løkker i nettverket, og vi skal gjennomgå den støtten Ethernet har for flytkontroll.

2.6.1 Adresselæring og flom

Med mindre man manuelt har fylt rutetabellen til en Ethernet-svitsj vil tabellen til svitsjen i utgangspunktet være tom. Tabellen bygges så opp ved at svitsjen noterer seg avsenderadressen til de pakker den mottar. Når en svitsj S mottar en pakke P med avsenderadresse A via link l, antar man at man ved et senere tidpunkt kan nå noden med adresse A ved å benytte nettopp linken l igjen. Dette noterer S i sin rutetabell ved å legge inn eller oppdatere ruteinformasjon tilknyttet adressen A. Dersom S skulle få inn en pakke med en destinasjonsadresse den ikke kjenner, og dermed heller ikke vet hvor den skal videresendes, vil svitsjen sende pakken ut på alle sine linker med unntak av den linken pakken kom inn på. Ved å sende en slik *flom* av pakker ut i nettverket oppnår man to ting. Først og fremst vil den noden pakken er tiltenkt motta pakken, forutsatt at destinasjonsadressen er gyldig, destinasjonsnoden er nåbar i nettverket og bit-feil ikke oppstår. Samtidig vil alle svitjser som ser pakken, også de som ikke ligger på veien mot destinasjonen, få oppdatert rutetabellen sin i henhold til avsender av pakken. Merk at rutetabellen alltid oppdateres i forhold til avsenderadresse, og ikke mottageradresse. Det er jo avsender av pakken som har vist aktivitet og derved gitt beskjed om hvor den befinner seg og at den er i live. Dersom en svitsj ikke mottar en pakke med avsenderadresse A innen en gitt tid, vil



Figur 2.7: Et nettverk av Ethernet-svitsjer før (i) og etter (ii) svitsjene har benyttet spenntre-algoritmen. C er valg til rot i treet. Stiplede kanter illustrerer linker som er lagt uvirksomme av algoritmen.

---

svitsjen slette ruten til A fra rutetabellen. Slik hindrer man at adresser lever evig i tabellen, noe som for eksempel er gunstig dersom en node flyttes innad i nettverket.

Å sende pakker som en flom i et Ethernet forutsetter at nettverket ikke inneholder løkker. Om det finnes en løkke risikerer vi at flere uheldige situasjoner kan oppstå. For eksempel vil en node kunne motta samme pakken flere ganger. Sett at vi benytter flom i nettverket i figur 2.7(i). H sender en pakke P til G med destinasjonsadresse A. Siden G ikke kjenner veien til A, videresendes P både til B, C og F. Både B, C og F har nylig vært i kontakt med pakker fra A, så P sendes fra B til A, fra C til A og fra F til A (via E). Med andre ord mottar A pakken P hele tre ganger; nettverket har duplisert pakken. Legg også merke til en annen uheldig egenskap ved denne flommen: F mottok en pakke fra G med avsenderadresse H. Med andre ord vil F nå tro at pakker som skal til H bør sendes ut på linken i retning G, til tross for at F fortsatt har en direkte link til H. Akkurat i dette eksempelet medfører ikke denne vranglæren en veldig stor omvei, men det illustrerer likevel et problem en flom kan føre med seg i et nettverk med løkker: En node kan i løpet av en flom via forskjellige linker motta pakker med samme avsenderadresse, noe som fører til en lite stabil rutetabell.

Det største problemet tilknyttet løkker i Ethernet oppstår i det en node ønsker å kringkaste en pakke i nettverket. I en slik situasjon vil enhver svitsj som mottar pakken sende den ut på alle andre linker enn den pakken kom inn på. I et nettverk med løkker vil en slik masseutsendelse raskt kunne føre til en massiv dublisering av pakker, noe som igjen i løpet av kort tid

vil kunne overbelaste nettverket og gjøre det ubrukelig. For å unngå løkker implementerer derfor Ethernet en egen spenntre-algoritme vi skal kikke på i neste delkapittel.

2.6.2 Spenntre-algoritmen

Ethernet har gjennom IEEE 802.1D[2, 3] fått standardisert hvordan svitsjer skal unngå løkker i et nettverk. IEEE 802.1D definerer en egen spenntre-algoritme som svitsjene i nettverket benytter[8]. Først blir alle svitsjene enige om en felles rot. Etter at roten er valgt skal svitsjene kun videresende pakker ut på de porter som ligger langs en vektet, korteste vei fra roten og til en gitt link. Vekten til en link er i utgangspunktet avhengig av en links hastighet, men kan overstyres. For vår del, hvor alle linker har samme hastighet, holdes vekten lik for alle linkene i et nettverk. Dersom flere svitsjer (eller én svitsj med flere porter) er koblet til samme link, og de begge representerer samme vektede avstand fra linken til roten, velges den svitsjen (eller porten) som har lavest *broidentifikator*[42] (eller *portidentifikator*[42]). Slik unngår man at nettverket inneholder løkker. Figur 2.7 illustrerer hvordan spenntre-algoritmen legger linker uvirksomme. Merk at den originale spenntre-algoritmen var beregnet på kringkastingsnettverk hvor hver link som regel utgjorde et eget nettverkssegment med mer enn to noder koblet til. Om vi relaterer dette til nettverket i figur 2.7, ville for eksempel linken mellom B og G ha et sett med endenoder koblet til seg. Det er da vesentlig at enten B eller G faktisk videresendte pakker ut på også denne linken, til stoss for at linken står i fare får å danne en løkke i nettverket. Så lenge kun én av de to svitsjene foretar videresending av pakker er vi likevel trygge. For et Ethernet uten kringkastingssegementer vil B eller G i teorien fortsatt videresende pakker ut på linken seg i mellom, men siden den andre svitsjen har lagt linken død, fremstår også linken i sin helhet som uvirksom.

2.6.3 Flytkontroll

Behovet for eksplisitt flytkontroll i Ethernet oppstod i overgangen fra kringkastingsnettverk til svitsjede nettverk på 90-tallet. Innføringen av dedikerte toveis-linker førte til at mottager av trafikk ikke lenger kunne bremse avsender ved å selv sende trafikk ut på det delte mediet. En "protestpakke" fra mottager førte tidligere til en kollisjon i nettverket, noe som igjen medførte at avsender ville ta en pause. Toveis-linker hvor man samtidig kan sende trafikk i begge retninger forhindrer kollisjoner fra å oppstå. For å gi mottager av trafikk tilbake muligheten til å bremse avsender fikk Ethernet definert flytkontroll gjennom IEEE 802.3x[5, 42].

[8] IEEE 802.1D omhandler *broer*. Broer og svitsjer har den samme grunnleggende funksjonalitet, men refererer gjerne til forskjellige generasjoner av noder med videresendingsfunksjonalitet[42].

Ethernets flytkontroll baserer seg på en egen kontrollpakke i form av en *pause-pakke*. Pause-pakken inneholder en tallverdi t som angir hvor lenge flytkontrollen skal være skrudd på. På/av-flytkontroll implementeres ved at mottager av trafikk ved behov sender ut en pause-pakke til avsender. Avsender mottar kontrollpakken og pauser trafikken for en periode avhengig av t. Siden tallverdien til t er begrenset, kan det være behov for lengre pauser enn hva man kan representere ved hjelp av én pause-pakke. En lengre pause oppnår man ved å sende nye pause-pakker etter hvert som tiden går. En t i en ny pause-pakke overskriver nemlig en gammel t hos avsender. Slik kan mottager av trafikk pause avsender så lenge det er ønskelig. Avsender vil først starte å sende trafikk når tiden enten går ut, eller dersom avsender mottar en pause-pakke med t lik null. En pause-pakke med t lik null fungerer som en beskjed om at man kan skru av flytkontrollen umiddelbart.

Bruk av pause-pakker i steden for av- og på-pakker er ment å øke robustheten til flytkontrollen. Om en av-pakke blir borte kan man risikere og pause linken unødvendig lenge, og i verste fall permanent. Ved bruk av pause-pakker vil tiden etter hvert gå ut, og flytkontrollen vil bli skrudd av igjen, selv ved pakke-tap.

2.6.4 Ethernet som tett koblet nettverk

For å kunne benytte Ethernet for tett koblede nettverk benytter vi oss av på/av-flytkontroll implementert ved help av pause-pakker. Slik søker vi å redusere antallet pakker som kastes i nettverket. Samtidig er vi avhengig av at alle linker i nettverket tas i bruk for å oppnå god ytelse. Det er dermed en nødvendighet at spenntre-algoritmen skrus av. Det er denne algoritmen som i Ethernet sørger for at linker bli liggende uvirksomme. De problemer som da oppstår i Ethernet med duplikatpakker, pakker som sendes evig rundt i nettverket og rutetabeller som ikke konvergerer, håndterer vi ved å ta kontroll over rutetabellene. Ethernet-svitsjene kan ikke lenger lære ruter ved behov og falle tilbake til flomutsendelse dersom en destinasjonsadresse er ukjent. Vi har sett at en flom i et nettverk med løkker kan få uheldige konsekvenser. Vi fyller derfor ut rutetabellene til svitsjene ved hjelp av up*/down*-ruting. Merk at det å fjerne behoved for flomutsendelse i seg selv vil ha en positiv effekt på ytelsen til nettverket siden man unngår å feilsende pakker i nettverket, pakker som ville lagt beslag på ressurser frem til de ble kastet.

2.7 Oppsummering

I dette kapittelet har vi utdypet og presisert en del begreper knyttet til våre tre problemstillinger. Vi har gått igjennom ulike egenskaper ved tett koblede nettverk; vi har sett på hva slags tjenestekvalitet vi ønsker for denne typen nettverk, og vi har nevnt fire vesentlige karaktertrekk: topologi, ruting,

flytkontroll og svitsjing. Videre har vi vært innom tre situasjoner vi søker å unngå i et nettverk: utsulting, kontinuerlig omruting og vranglås. Til slutt gikk vi gjennom tre sider ved Ethernet som er relevante for denne oppgaven: Vi har sett hvordan svitsjer i et Ethernet vedlikeholder rutetabellene sine, vi har sett hvordan spenntreet i et Ethernet bygges opp, og vi har helt til slutt sett hvordan Ethernet støtter flytkontroll ved hjelp av en egen pause-pakke.

Kapittel 3

Simulatoren

Dette kapittelet inneholder en grundig gjennomgang av den simulatoren vi har implementert for å besvare våre problemstillinger. Vi begrunner først valg av simuleringsmodell, før klasse-filer og designavgjørelser blir forklart. Simulatorens validitet blir diskutert mot slutten av kapittelet.

Som beskrevet i kapittel 1.5 ønskes problemstillingene i denne oppgaven besvart ved hjelp av simulering. Når simulering benyttes som metode for å besvare problemstillinger knyttet til nettverksteknologi, søker man å implementere et virtuelt nettverk i programvare på en slik måte at man med tilstrekkelig presisjon speiler virkeligheten, og ved dette kan bruke det virtuelle nettverket til å trekke konklusjoner om hvordan et tilsvarende nettverk i den virkelige verden vil oppføre seg. Programvare som brukes for å implementere slike virtuelle nettverk kalles gjerne en *nettverkssimulator*, eller bare *simulator* dersom konteksten er entydiggjørende. Er simulatoren godt implementert vil den kunne fremstå som en verktøykasse for å modellere forskjellige typer nettverk. Man vil kunne ha muligheten til å variere nettverksparametre som antall linker og noder i nettverket (både endenoder og svitsjer), hvor mye trafikk som genereres i nettverket (gjerne kalt *påtrykket*), hva slags svitsjing og rutealgoritme som benyttes, bufferstørrelser et cetera. Dette gjør en simulator svært fleksibel sammenliknet med nettverk i den virkelige verden. En simulator fristiller en fra hensyn som for eksempel hvor mye komponenter i et nettverk koster å kjøpe inn, og hvilke teknologier som faktisk er implementert i tilgjengelige produkter. Man vil også typisk ha mulighet til å la programvaren selv variere nettverksparametre. Slik kan simulatoren enkelt settes til å iterere over forskjellige nettverksoppsett hvor man for eksempel varierer nettverkstopologien mens andre parametre holdes fast.

Nå er det dessverre ikke kun fordeler ved å bruke simulering som metode.

Det å implementere en simulator tar tid, og det er rikelig med muligheter for at feil oppstår under veis. Samtidig vil det gjerne være behov for store mengder regnekraft for å kjøre de ønskede simuleringer. En vesentlig utfordring ligger i å sikre seg best mulig forståelse av det systemet man ønsker å simulere slik at den *modellen* man bygger opp gjenspeiler virkeligheten på en tilstrekkelig god måte. Det er viktig å her presisere kravet om at modellen er *tilstrekkelig* god siden man må påberegne å gjøre forenklinger i forhold til virkeligheten for at modellen skal være håndterbar og la seg implementere med de ressurser man har tilgjengelig. Her må man være påpasselig for å unngå logiske feil i modellen, samt passe på at man ikke forenkler bort deler av virkeligheten som kan ha vesentlige implikasjoner på det man skal studere. Etter hvert som modelleringen skrider frem, vil man starte på selve implementasjonsprosessen - realiseringen av modellen i form av en simulator. Dette medfører gjerne implementering av modellen i et generelt programmeringsspråk, eller eventuelt definering av modellen ved hjelp av et simuleringsverktøy[1]. Uansett må man forvente at denne overgangen kan introdusere nye feil og unøyaktigheter.

3.1 Simuleringsmodell

For å kunne bygge en god simulator er det viktig at man finner og benytter gode verktøy for både modellering og implementering. Man må ta stilling til om man ønsker å implementere simulatoren fra grunnen av, om man ønsker å videreutvikle en simulator andre har implementert tidligere, eller om det er hensiktsmessig å ta utgangspunkt i et skreddersydd simuleringsverktøy. Som en del av denne prosessen er det naturlig å vurdere hvilke verktøy og språk som egner seg til den typen simulering man skal gjennomføre, samt å velge *simuleringsmodell*. Det finnes en stor mengde simuleringsmodeller. Et par eksempler er endelig-tilstandsmaskiner og prosessemulering, men innen nettverkssimulering er nok varianter av en *diskret hendelsesmodell* mer vanlig[17, 41].

En simuleringmodell speiler komponenter fra virkeligheten samt interaksjon mellom disse komponentene. En diskret hendelsesmodell gjør dette ved at virkeligheten representeres ved et sett *tilstandsvariable*. Et subsett av disse variablene reflekterer gjerne én komponent fra virkeligheten. Verdiene til alle tilstandsvariablene definerer til en hver tid systemets tilstand og kan sees på som et øyeblikksbilde fra den verden man ønsker å simulere. Modellen beveger seg fra en tilstand til en annen ved at en eller flere av disse variablene forandrer verdi. Verdiforandring skjer som resultat av en veldefinert *hendelse*. En hendelse skjer imidlertid ikke ved et vilkårlig tidspunkt, siden tiden i simulatoren er *diskret*. Man snakker med andre ord ikke om et

[1] *Inside discrete-event simulation software: how it works and why it matters*[41] gir en kort introduksjon til noen simuleringsverktøy og deres logiske oppbygning.

kontinuerlig tidsperspektiv, men en tid brutt opp i små *klokketikk*. Et tikk er således den minste oppløsningen for tid i modellen. Det er ikke slik at ethvert tikk må romme en eller flere hendelser, men det er samtidig aldri slik at en hendelse strekker seg over flere tikk. En hendelse startes og fullføres innen samme tikk, og har dermed ingen utstrekning i tid innad i modellen. Alle hendelser innen samme tikk ansees å hende samtidig til tross for at de utføres i streng sekvensiell rekkefølge. En hendelse kan forårsake en eller flere andre hendelser - hendelser som ligger null eller flere hele tikk lengre frem i tid. Typiske eksempler på hendelser er starten og slutten på en *aktivitet*. En aktivitet har en bestemt utstrekning i tid, og kan i nettverkssammenheng for eksempel gjenspeile det å sende informasjon fra en node til en annen. Antall tikk aktiviteten tar skal i dette eksempelet reflektere hvor lang tid det tar å sende informasjon mellom de to nodene. Én hendelse vil i dette tilfelle definere starten på den aktuelle aktiviteten. Dette innbefatter blant annet generering av en ny hendelse et bestemt antall tikk lengre frem i tid, nemlig mottak av informasjonen hos mottagernoden. Hendelsen knyttet til mottaket av informasjonen avslutter en aktivitet, men vil samtidig kunne fungere som start på en annen aktivitet igjen, eller eventuelt en *køaktivitet*. En køaktivitet skiller seg fra en vanlig aktivitet ved at man ikke på forhånd vet hvor langt frem i tid aktivieten er ferdig. Som navnet tilsier skyldes dette gjerne at man ligger i kø for å vente på at en eller annen betingelse skal oppfylles i modellen.

For å holde orden på hendelsene og når de skal gjennomføres, organiseres de i en *hendelseskø*. Organisering av hendelser i hendelseskøen og styring av tiden utgjør selve kjernen i en diskret hendelsesmodell. Først i hendelseskøen ligger alltid den neste hendelsen som skal eksekveres. Tiden drives fremover av hendelseskøen ved at tiden alltid settes lik tiden for den hendelse som hentes ut av køen. Tiden er således *hendelsesdrevet*. Dette står i kontrast til en *tidsdrevet* modell, hvor man for hvert klokketikk sjekker alle komponenter i modellen for å se hvem som har noe de ønsker å utføre. Hva som er mest effektivt av en hendelsesdrevet og tidsdrevet modell avhenger i stor grad av forholdet mellom antall hendelser og tid. En tidsdrevet modell vil håndtere et stort antall hendelser per tid bedre enn en hendelsesdrevet modell. På den andre siden kan erfaring tyde på at en hendelsesdrevet modell er enklere å videreutvike og forbedre[24].

3.2 Simulatoren brukt i denne oppgaven

En simulator bygget rundt en diskret hendelsesmodell er valgt som utgangspunkt for å besvare problemstillingene i denne oppgaven. Olav Lysne har implementert et rammeverk for nettverkssimulering som han tidligere har brukt som grunnlag for flere simuleringsprosjekter både ved Institutt for informatikk ved Universitetet i Oslo og Simula Research Laboratory. Ram-

Simulatorkjernen	Beskrivelse
Kernel.java Unit.java RandomNumberGenerator.java	Kernel.java inneholder hendelseskøen og funksjonalitet for håndtering av denne, mens alle klasser hvis objekter skal ligge i hendelseskøen, direkte eller indirekte, må være en subklasse av Unit.

Tabell 3.1: Oversikt over klassefilene som til sammen danner kjernen i simulatoren.

meverket, heretter referert til som *NetSim*, mangler i utgangspunktet en god del funksjonalitet for direkte å kunne brukes til å belyse denne oppgavens problemstillinger. NetSim er imidlertid implementert i det objektorienterte programmeringsspråket Java, er godt dokumentert, og er således et bra utgangspunkt for videreutvikling. Objektorienterte språk viser seg også å fungere ypperlig som utgangspunkt for å implementere simulatorer[29].

En gunstig side ved det å basere seg på NetSim er at denne simulatoren har kjernefunksjonalitet som alt er grundig testet og utprøvd. Selv om det alltid vil ta tid å sette seg inn i en annen persons arbeid, og NetSim samtidig vil måtte utvides ganske betraktelig for å kunne brukes i denne oppgaven, er inntrykket likevel at NetSim vil bringe oss raskere til en simulator vi kan stole på at modellerer virkeligheten på en tilstrekkelig god måte, enn om vi hadde startet på bar bakke. Det ble derfor funnet hensiktsmessig å bruke dette rammeverket som utgangspunkt for å bygge opp en simulator tilpasset våre problemstillinger.

3.3 Klassefiler

Den ferdige simulatoren består av klasse-filene listet i tabellene 3.1 til 3.6. Alle filene er del av den samme Java-pakken, *Network*, men for å gjøre det mer oversiktlig er tabellene ordnet slik at filer innen samme tabell utgjør en naturlig komponent i simulatoren. Hver klasse-fil tilsvarer en Java-klasse. I tillegg finner vi lokale hjelpeklasser i filene merket med *(h)*. Vi skal nå gå igjennom de forskjellige komponentene og deres hovedfunksjonalitet, da med særskilt vekt på de utvidelser av NetSim som er implementert som en del av denne oppgaven. Til slutt i kapittelet vil vi gi en kort oppsummering av vårt implementasjonsarbeide.

Endenode	Beskrivelse
DrainNode.java SourceNode.java*(h)* TrafficConsentrator.java	Disse tre klassene utgjør til sammen en toveis endenode. SourceNode.java inneholder hjelpeklassen *Application*.
TrafficReporter.java	Denne klassen kan brukes for å rapportere trafikkgenerering hos en kildenode.
Distribution.java Domain.java Functions.java ParetoDistribution.java	Disse fire klassene fungerer som hjelpe-klasser for å generere selv-lik trafikk hos en kildenode.
Zipf.java	Denne klassen implementerer Zipf-distribusjon.

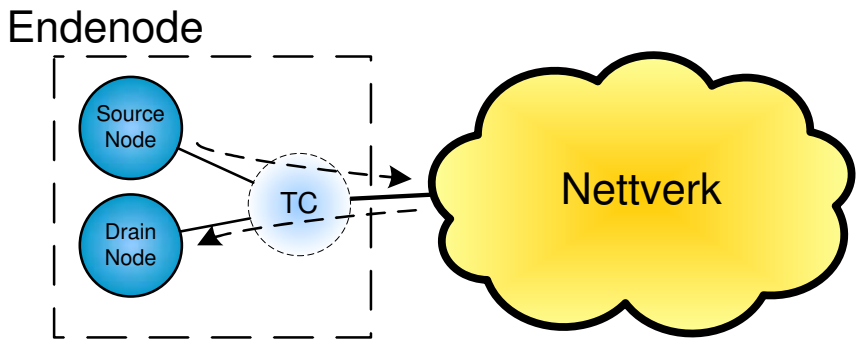
Tabell 3.2: Oversikt over klassefilene som til sammen danner en endenode i simulatoren.

3.3.1 Simulatorkjernen

De tre filene i tabell 3.1 utgjør selve kjernen i simulatoren. Som beskrevet i kapittel 3.1 består hovedoppgavene til kjernen i en diskret hendelsesmodell i å håndtere en hendelseskø, samt styre tiden. Denne funksjonaliteten er implementert i Kernel.java. Selve køen er implementert som en prioritetskø, hvor tiden for hver hendelse blir brukt som prioritet. Enhver hendelse som skal ligge i køen må direkte eller indirekte være en subklasse av klassen Unit. Unit.java definerer således et sett med funksjoner som alle hendelser må implementere. Spesielt kan nevnes metoden *act()* som gjenspeiler de operasjoner som vil bli utført i det en hendelse hentes ut av hendelseskøen.

Simulatorkjernen er gjennomgått relativt få forandringer fra NetSim. Den største forandringen ligger i at typen for tid har blitt forandret fra *int* til *long*. Dette viste seg å være nødvendig ved innføring av kildenoder som genererer selv-lik trafikk (kapittel 3.3.2). Her kan man oppleve behov for å legge inn hendelser som ligger svært langt frem i tid, noe som fører til at tallet som gjenspeiler tiden blir for stort til å kunne få plass i en variabel av typen integer.

Klassen RandomNumberGenerator er implementert av Sundar Dorai-Raj, Virginia Polytechnic Institute and State University, og gitt ut under GNU General Public License. Klassen og dens metoder brukes av klassen Kernel for å generere tilfeldige tall. RandomNumberGenerator.java er uforandret fra NetSim.



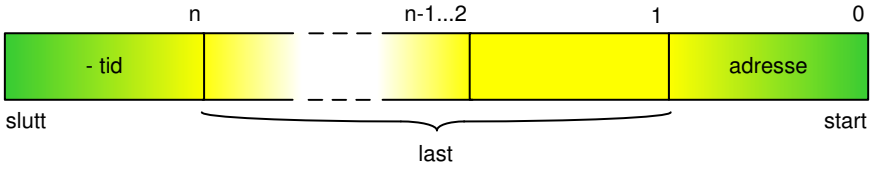
Figur 3.1: En endenode som både kan sende og motta pakker er satt sammen av en SourceNode, en DrainNode og en TrafficConsentrator (TC).

3.3.2 Endenodene

De tre klassene DrainNode, SourceNode og TrafficConsentrator utgjør til sammen hovedkomponentene til en endenode i nettverket (tabell 3.2). DrainNode implementerer funksjonalitet for å ta imot pakker fra nettverket, mens SourceNode implementerer funksjonalitet for å sende pakker ut i nettverket. Sammen utgjør de en toveis endenode – en endenode som både kan sende og motta trafikk. DrainNode og SourceNode er imidlertid uavhengige av hverandre i den forstand at man kan la en endenode kun bestå av en SourceNode eller en DrainNode. Da sitter man igjen med en endenode som henholdsvis kun kan sende trafikk ut i eller kun kan motta trafikk fra nettverket. Når vi snakker om en endenode i egenskap av å være en kildenode, er det i realiteten SourceNode-delen av endenoden vi snakker om. På samme måte er en destinasjonsnode DrainNode-delen av en endenode.

TrafficConsentrator (TC) innehar den funksjonalitet som trengs for at endenodene skal kunne delta ved automatisk konfigurering av nettverket. Konfigurering er kun nødvendig ved starten av hver simulering hvor topologiinformasjon utveksles og rutetabeller bygges opp[2]. Med andre ord innehar TC funksjonalitet som ikke er i bruk ved ordinær ende-til-ende-trafikk. Man kan sette opp nettverk og kjøre simuleringer uten bruk av TC, men man må da manuelt konfigurere nettverket og dets rutetabeller. Dette impliserer igjen at TC ved normal ende-til-ende-trafikk i størst mulig grad bør være

[2]Som også nevnes i kapittel 3.3.3 har vi i forbindelse med denne oppgaven ikke kjørt simuleringer hvor det er nødvendig med dynamisk rekonfigurering av nettverket. Alle simuleringer er stabile i den forstand at topologien ikke forandres mens simuleringen finner sted.



Figur 3.2: Pakkeformatet brukt av NetSim.

---

usynlig for de andre komponentene i nettverket. For å få til dette er TC implementert som en spesialsvitsj uten forsinkelse. TC har samme adresse som bakenforliggende SourceNode og DrainNode, og opptrer på vegne av disse to komponentene overfor resten av nettverket. Målet er at SourceNode og DrainNode skal oppleve tilnærmet de samme kvaliteter fra nettverket som om de var koblet direkte til første svitsj - uten å gå gjennom en TC. Figur 3.1 viser en toveis endenode bestående av TC, DrainNode og SourceNode.

SourceNode og trafikkgenerering

Kildenodene genererer informasjonspakker som sendes gjennom nettverket til destinasjonsnodene. I et reelt nettverk vil det være en eller flere applikasjoner som står bak denne trafikkgenereringen. I vår simulator forenkler vi virkeligheten ved å benytte syntetiske trafikkgeneratorer for å etterlikne disse applikasjonene. Det å konstruere en slik syntetisk trafikkgenerator innebærer da å ta stilling til følgende tre spørsmål: Hvor ofte skal man sende pakker ut i nettverket, hvor store skal pakkene være, og hvilke destinasjonsnoder skal man sende til? De designvalg man tar i det man besvarer disse tre spørsmålene vil ha direkte innvirkning på trafikken i nettverket, og vil således påvirke senere simuleringsresultat. Vi skal gjennomgå valg av algoritme for å styre hvor ofte vi genererer en ny pakke, før vi mot slutten av delkapittelet skal se litt på hvordan vi avgjør pakkenes fordeling blant destinasjonsnodene. Aller først skal vi imidlertid kommentere valg av pakkestørrelse.

Vår simulatoren har få begrensninger hva gjelder pakkestørrelse. Pakkene kan i teorien være så store man måtte ønske, men er nedad begrenset av at en pakke må inneholde destinasjonsadresse samt tidspunkt for når pakken ble laget hos kildenoden. Figur 3.2 illustrerer dette. Merk at dette pakkeformatet er svært enkelt sammenliknet med for eksempel det vi finner i Ethernet. Formatet er likevel tilstrekkelig til at vi for sendt med pakken all nødvendig informasjonen i vår simulator. Det er ingen ting i veien for at pakker som sendes i løpet av en simulering kan ha varierende størrelse. Når man benytter svitsjer som implementerer lagre-og-videresend, er det imidlertid naturlig å la flytkontrollen operere på pakkenivå. Da kan det være hensiktsmessig å benytte inn- og utbuffer som har plass til et bestemt antall hele pakker, noe

som igjen enklest lar seg implementere ved at alle pakker som sendes i løpet av en simulering har samme størrelse. Derfor er alle pakker som er generert innenfor samme simulering i denne oppgaven alltid av samme størrelse.

For å modellere hyppigheten av pakkegenerering hos kildene har NetSim støtte for bruk av to statistiske modeller, *uniform fordeling* og *Poisson-fordeling*. Støtte for en tredje modell er implementert som en del av arbeidet med denne oppgaven.

Ved bruk av uniform fordeling baserer man seg på at man for hver pakke som genereres, trekker et tall t. Tallet t representerer tiden som skal gå før neste pakke lages. Tidspunktet t blir trukket som et tilfeldig tall mellom en nedre og øvre grense, men på en slik måte at t over tid er jevnt fordelt i intervallet mellom grensene. Den nedre grensen kan man sette slik at man ved minste t gjenspeiler umiddelbar generering av ny pakke. Den øvre grensen kan man så bruke til å styre gjennomsnittlig pakkegenerering over tid, og dermed mengden pakker kilden sender inn i nettverket. Lar man den øvre grensen nærme seg den tiden det tar for kilden å sende en pakke, vil kilden øke påtrykket mot nettet og etterhvert generere en kontinuerlig pakkestrøm begrenset oppad av utlinkens kapasitet.

Implementasjon av trafikkgenerering med Poisson-fordeling som grunnmodell, har samme tilnærming som når man baserer seg på uniform fordeling. For hver nye pakke som lages trekker man en ny tid t, som er tiden til neste pakkegenerering. Tiden t blir trukket ut fra en Poisson-fordeling. Poisson-fordeling er en statistisk sannsynlighetsmodell med både gjennomsnitt og varians lik m [13]. Ved å justere m kan man variere påtrykket fra en kildenode mot nettverket.

Både uniform fordeling og Poisson-fordeling kan brukes som utgangspunkt for å bestemme hvor ofte en kilde genererer nye pakker. Begge metodene har imidlertid sine svakheter. Uniform fordeling gir for korte tidsintervaller variasjon i pakkegenereringen, men over lengre perioder oppnår man en svært stabil gjennomsnittelig generering av pakker. Dette står i kontrast til målinger gjort i reelle nettverk, hvor trafikken viser stor variasjon uavhengig av størrelsen på tidsintervallet [36, 38]. Poisson-fordeling har tilsvarende svakheter; trafikk man genererer gjenspeiler ikke på en god måte den aktivitet man finner i virkelige nettverk. Paxson og Floyd tar i artikkelen "Wide-Area Network: The Failure of Poisson Modeling"[38] for seg problemer med Poisson-fordeling i forbindelse med nettverksmodellering.

Målinger gjort av reell trafikk i forskjellige typer nettverk viser at trafikken er *statistisk selv-lik*[3] (eng. *self-similar*)[12, 36, 38]. Statistisk selv-likhet er en variant av matematisk selv-likhet, som igjen er et begrep innen fraktalteori [30]. Generelt kan man si at det er snakk om en type fenomener hvor

[3]Det diskuteres hvorvidt statistisk selv-lik distribusjon er en tilstrekkelig modell for å beskrive trafikk i nettverk [20, 45]. Denne diskusjonen ligger imidlertid på et detaljnivå som gjør den lite relevant for det vi ønsker å se nærmere på i denne oppgaven.

man finner igjen like (geometriske) mønstre når en studerer et konkret fenomen over forskjellige skalaer. Overført til trafikkgenerering i et nettverk gir det følgende: En kildenode produserer trafikk med perioder av høyt og lavt påtrykk, uten at slike perioder har noen naturlig lengde i tid. Sagt på en annen måte: Trafikken en kilde genererer består av perioder med lavt og høyt påtrykk, uavhengig av tidsskalaen vi bruker for registrering.

Selv-lik trafikkgenerering kan modelleres ved at hver kildenode settes sammen av flere *av/på-noder*. En av/på-node genererer nye pakker i på-periodene, mens den hviler i av-periodene. Av/på-nodene alternerer mellom disse to tilstandene. Lengden på av- og på-periodene hos forskjellige av/på-noder er uavhengig av hverandre, men må ta utgangspunkt i samme distribusjon. Dersom et sett av/på-noder jobber i parallell mot en kildenode, og varigheten av av- og på-periodene har uendelig varians, vil kildenoden de genererer pakker for fremstå som en selv-lik trafikkgenerator [46, 52].

For å skape av- og på- perioder med de ønskede egenskaper trenger vi en sannsynlighetsmodell som støtter uendelig (eller tilstrekkelig høy) varians. *Pareto-distribusjon* er en slik modell:

$$P(x) = \frac{\alpha\beta^{\alpha}}{x^{\alpha+1}} \qquad x \geq \beta \tag{3.1}$$

Grafen for funksjon (3.1) har lang *hale*. Denne halen gjenspeiler variansen til modellen. Parameteren α bestemmer tykkelsen på halen, mens β angir skalaen for x-aksen. En gunstig egenskap ved Pareto-distribusjon er det enkle forholdet mellom α og *Hurst*-parameteren $\mathcal{H}$. $\mathcal{H}$ er foreslått som mål på grad av selv-likhet [36]. Hurst-parameteren har en verdi fra 0.5 til 1.0, hvor 1.0 angir størst mulig grad av selv-likhet. Forholdet mellom α og $\mathcal{H}$ er gitt ved $\mathcal{H} = \frac{3-\alpha}{2}$. Forventet verdi for en Pareto-distribuert tilfeldig variabel er gitt ved $E(X) = \frac{\alpha\beta}{\alpha-1}$, $\alpha > 1$. Om vi nå velger α ut fra ønsket verdi for $\mathcal{H}$, kan vi sette β for av- og på-periodene ut fra hvor stort påtrykk vi ønsker fra en kildenode mot nettverket[4].

NetSim er utvidet med støtte for selv-lik trafikkgenerering etter modellen gitt ovenfor. Selve implementasjonen baserer seg delvis på arbeid utført av Amund Kvalbein i forbindelse med hans hovedfag ved SRL [35]. Amund Kvalbeins arbeide bygger igjen på en implementasjon av Pareto-distribusjon gjort av Kyle Siegrist og Dawn Duehring ved universitetet i Alabama, Huntsville. Deres arbeide er gjengitt i de fire filene Distribution.java, Domain.java, Functions.java og ParetoDistribution.java (tabell 3.2). TrafficReporter.java er i tillegg implementert for å kunne overvåke en kildenode og det trafikkmønster den genererer.

Ved målinger av nettverkstrafikk i Ethernet har man estimert $\mathcal{H}$ til å ligge rundt 0.9 [36]. Dette er brukt som utgangspunkt for å sette $\alpha = 1.2$. I vår simulator genererer en av/på-node under en på-periode pakker med hyppighet

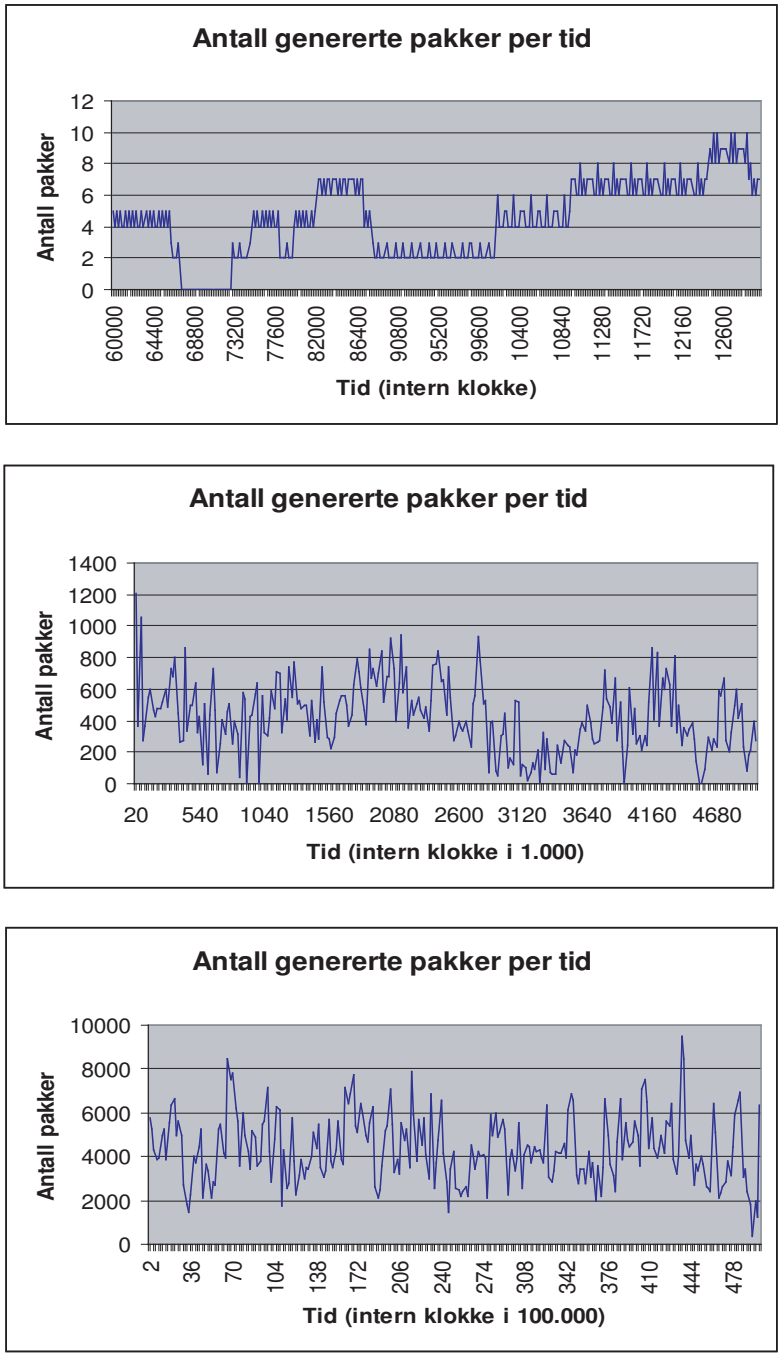
[4]Selv-likheten til en kildenode er robust med hensyn til valg av forventet lengde for av- og på-periodene [52].

tilsvarende 1/3 av kapasiteten til kildenodens utlink. En kildenode består av 10 av/på-noder. Figur 3.3 illustrerer trafikk generert av en kildenode brukt i vår simulator. β er for dette eksempelet satt slik at forventet bruk av kildenodens utlink er 2/3 av kapasiteten. Som vi ser av figurene genereres det over tid trafikk med forskjellige grader av påtrykk, uavhengig av tidsskala. Dette står i skarp kontrast til trafikk generert etter for eksempel uniform fordeling. Figur 3.4 viser trafikk fra to forskjellige kildenoder over et gitt tidsintervall. Den ene kilden genererer selv-lik trafikk, mens den andre genererer trafikk etter en uniform fordelingsmodell. Forventet påtrykk er det samme fra begge kildene. Vi ser at den uniforme kilden, som forventet, genererer en forholdsvis jevn strøm av pakker. Den selv-like kilden viser på sin side stor variasjon i pakkegenereringen.

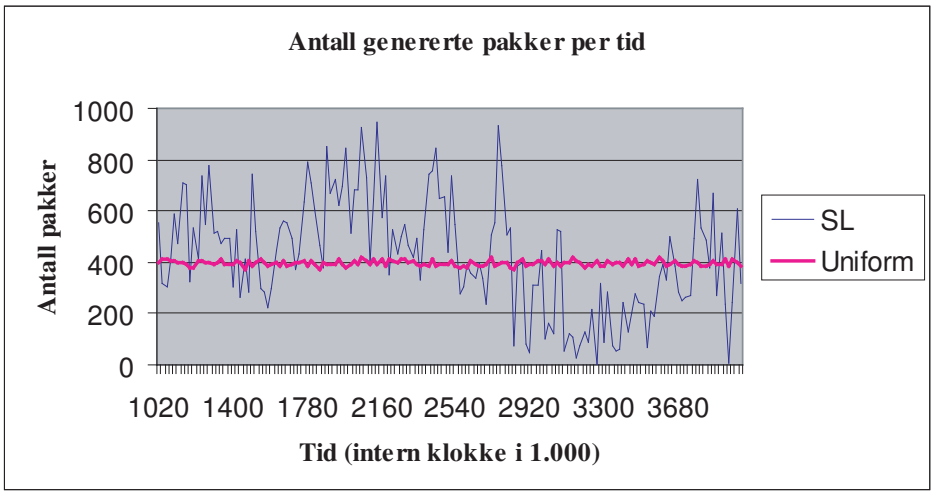
Til slutt kan det være verdt å merke seg to ting i forbindelse med genereringen av selv-lik trafikk for nettverkssimulering. For det første har vi til nå snakket om hyppigheten av pakkegenerering internt i en kildenode. Når en slik kilde sender pakkene ut i nettet, vil den være begrenset oppad av kapasiteten til utlinken. I perioder hvor kilden produserer mer trafikk enn utlinken kan bære, må pakker enten kastes eller lagres for senere sending. I vår simulator kaster ikke kilden pakker. Det forutsettes at kilden alltid har plass nok til å lagre de pakker som må vente. Slik lagring har implikasjon for trafikkmønsteret kilden genererer inn i nettverket. En periode med pakkegenerering over utlinkens kapasitet, vil på utlinken være representert ved en lengre periode med kontinuerlig, maksimal pakkeutsending. Dette er illustrert ved figur 3.5. Dersom det alltid genereres flere pakker enn det man kan sende, vil utlinken alltid sende en kontinuerlig pakkestrøm. I en slik situasjon spiller det ingen rolle hvilken statisktisk modell man har som utganspunkt for pakkegenereringen internt i kilden.

Det andre man bør merke seg er at svitsjene i et nettverk vil være med på å forme det generelle trafikkbildet i nettverket. Studier viser at svitsjene senker graden av selv-likhet internt i et nettverk hvor kildenodene genererer selv-lik trafikk. Samtidig øker svitsjene graden av selv-likhet for nettverk hvor kildene genererer trafikk som ikke er selv-lik [53]. Denne tilnærmingen av trafikkbildet er imidlertid ikke større enn at man fortsatt finner størst grad av selv-likhet internt i nettverk hvor kildene genererer selv-lik trafikk. På grunnlag av dette velger vi å bruke kildenoder som genererer selv-lik trafikk ved våre simuleringer. Brukes andre typer kildenoder vil dette bli oppgitt.

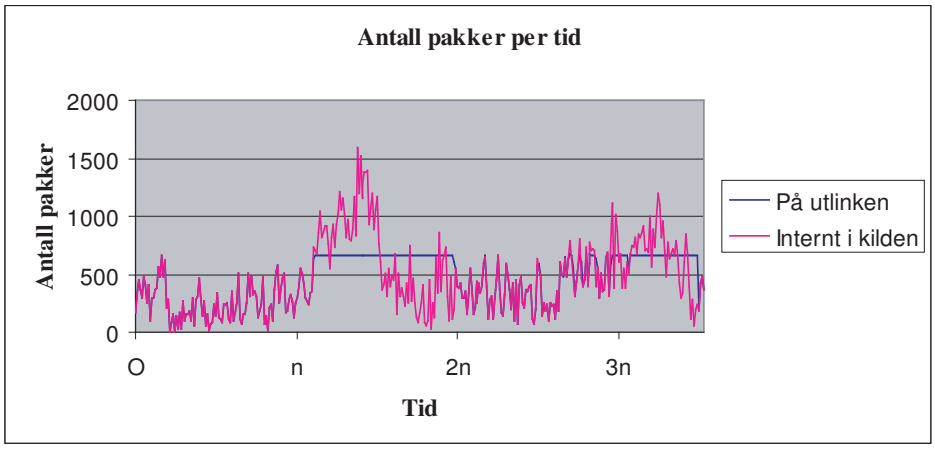
Når det gjelder destinasjonsfordeling, altså hvilke noder pakkene sendes til, har NetSim kun støtte for uniform fordeling. Dette er den mest brukte fordelingsmodellen for valg av destinasjonsnoder ved simulering av tett koblede nettverk[21]. Like fullt finnes det alternative modeller som også er mye brukt. Man kan for eksempel kjøre simuleringer med *alle-til-en-* eller *alle-til-noen*-kommunikasjon, man kan bruke varianter av *parvis* kommunikasjon



Figur 3.3: De tre grafene illustrerer hvordan en kildenode produserer trafikk med forskjellige grader av påtrykk, uavhengig av tisskala.



Figur 3.4: Grafene viser generering av pakker hos to kildenoder. Den ene kilden er selv-lik, den andre er uniform.



Figur 3.5: Grafene viser forholdet mellom intern pakkegenerering i en selv-lik node og det som sendes ut på linken.

eller man kan bruke en annen fordelingsmodell enn uniform fordeling som grunnlag for å velge destinasjonsnoder. Som et eksempel på det siste alternativet har *Zipf-distribusjon* blitt implementert som en del av arbeidet med denne oppgaven.

Uniform destinasjonsfordeling baserer seg naturlig nok på samme underliggende sannsynlighetsmodell som den vi snakket om i forbindelse med uniform fordeling ved hyppighet av pakkegenerering. I det man genererer en pakke hos kilden trekker man samtidig en adresse, A, blant alle lovlige destinasjonsadresser. A blir trukket på en slik måte at man over tid sender tilnærmet det samme antallet pakker til alle destinasjonsnoder. Sannsynligheten for at en node i sender en pakke til en node j er med andre ord lik for alle i og j i nettverket, $i \neq j$ [21, 40]. Vi krever $i \neq j$ siden vi kun er interessert i trafikk i nettverket, og ikke trafikk som en node eventuelt ønsker å "sende til seg selv". Man kan merke seg at dette kravet ikke vil ha innvirkning på fordelingsmodellen siden regelen er gjeldende for alle noder i nettverket.

Uniform fordeling gjenspeiler at alle noder i nettverket er like interessante når det gjelder informasjonsutveksling. Trafikken i nettverket fordeles jevnt blant alle noder, og er ved dette forholdsvis forutsigbar. En slik forutsigbarhet kan være nyttig ved simulering all den tid nettverkstrafikken i seg selv ikke er det man ønsker å studere. For eksempel kan et forutsigbart trafikkbilde være med å tydeliggjøre forskjeller i ulike rutealgoritmer eller topologier. Det vil fortsatt kunne være linker og svitsjer i nettverket som jevnt over er tyngre belastet enn andre linker og svitsjer, men dette skyldes da nettets topologi og valg av rutealgoritme mer enn benyttet destinasjonsfordelingen.

Det er imidlertid ikke vanskelig å tenke seg at noen noder kan sitte på mer interessant informasjon enn andre noder, eller at enkelte noder av forskjellige årsaker er gjenstand for hyppigere informasjonsutveksling enn den gjennomsnittlige node. Dette vil føre til skjevhet i trafikken i den forstand at områder i nettverket rundt mer trafikkerte noder vil være tyngre belastet enn resten av nettverket. Slike områder i nettverket er da gjerne mer utsatt for trafikkork, og det er økt sjanse for at nettverket som helhet først opplever metning rundt disse områdene. En slik vridning av trafikk over mot populære noder er observert ved analyse av trafikk i reelle nettverk[7, 51]. For å kunne simulere et tilsvarende trafikkbilde er *Zipfs lov* foreslått brukt som utgangspunkt[16, 22, 54].

Zipfs Lov har fått navn etter den amerikanske filolog og lingvist George Kingsley Zipf (1902-1950). Ved studier av naturlige språk oppdaget Zipf at det tilsynelatende er en sammenheng mellom frekvensen av hvert enkelt ord innen en større strukturert tekst og ordets rang om man sorterer ordene avtagende etter frekvens. La oss si at det ordet som forekommer oftest blir brukt x ganger i løpet av teksten. I følge Zipfs oppdagelser vil da det ordet som forekommer nest mest bli brukt omtrent $\frac{x}{2}$ ganger, mens det ordet som er brukt tredje mest vil forekomme cirka $\frac{x}{3}$ ganger. Slik fortsetter det med

ord nummer k brukt tilnærmet $\frac{x}{k}$ ganger.

La oss nå gjøre den forenkling å anta at Zipfs lov gjelder eksakt for den destinasjonsdistribusjon vi ønsker. Hver gang vi trekker en adresse A, må dette da gjøres slik at resultatet over tid følger Zipfs lov. La nå A_k betegne adressen som har rang k om vi sorterer destinasjonsnodenes adresser etter hvor ofte vi sender til dem. For at A_1 skal brukes k ganger så ofte som A_k er vi avhengig av at sannsynligheten for å trekke A_1, $P(A_1)$, er k ganger så stor som sannsynligheten for å trekke A_k, $P(A_k)$. Vi må altså ha

$$\begin{gathered} P(A_1) = k * P(A_k) \\ \Downarrow \\ P(A_k) = \frac{P(A_1)}{k}, k \neq 0. \end{gathered} \tag{3.2}$$

Samtidig vil vi alltid trekke nøyaktig én adresse blant alle mulige destinasjonsadresser. Summen av alle $P(A_k)$ må da være 1. Dermed har vi

$$\sum_{k=1}^{n} P(A_k) = 1, \tag{3.3}$$

hvor n er antall gyldige destinasjonsadresser i nettverket. Om vi nå kombinerer (3.2) og (3.3) gir det oss følgende:

$$\sum_{k=1}^{n} P(A_k) = 1 \Rightarrow \sum_{k=1}^{n} \frac{P(A_1)}{k} = 1 \Rightarrow P(A_1) \sum_{k=1}^{n} \frac{1}{k} = 1 \Rightarrow P(A_1) = \frac{1}{\sum_{k=1}^{n} \frac{1}{k}}$$

Som igjen gir oss:

$$P(A_k) = \frac{P(A_1)}{k} = \frac{\frac{1}{\sum_{k=1}^{n} \frac{1}{k}}}{k} = \frac{1}{k \sum_{k=1}^{n} \frac{1}{k}} = \frac{1}{kH_n} \tag{3.4}$$

for $k \neq 0$, $H_n = \sum_{k=1}^{n} \frac{1}{k}$. H_n kjenner vi igjen som de n første leddene i den harmoniske rekken, altså det *n'te harmoniske tall*.

Forholdet gitt ved (3.4) refereres gjerne til som den klassiske utgaven av Zipfs lov[5]. Det er denne utgaven av loven som er brukt som utgangspunkt for

[5]Matematisk uttrykkes Zipfs lov gjerne slik

$$f(k; s, N) = \frac{\frac{1}{k^s}}{\sum_{n=1}^{N} \frac{1}{n^s}}$$

hvor s typisk tar en verdi i nærheten av 1. Vår forenkling består i å velge s nøyaktig lik 1. En slik forenkling bringer oss til den klassiske utgaven av Zipfs lov:

$$f(k; 1, N) = \frac{1/k^1}{\sum_{n=1}^{N} 1/n^1} = \frac{1}{k \sum_{n=1}^{N} 1/n} = \frac{1}{kH_n}, \text{ hvor } H_n = \sum_{n=1}^{N} 1/n.$$

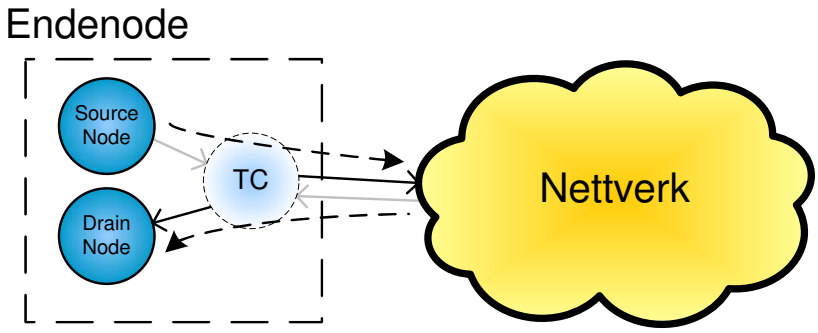
den Zipf-distribusjonen som er implementert i forbindelse med denne oppgaven. Klassen Zipf (Zipf.java) inneholder en metode som over tid vil returnere en Zipf-distribusjon av tallene fra og med 0 til en gitt øvre grense. Den øvre grensen vil for vår del være satt ut fra antall destinasjonsnoder i nettverket. Et trukket tall tilsvarer da adressen til en destinasjonsnode. Man kan velge om tallenes rang skal følge tallrekken i den forstand at tallet 0 returneres oftest, tallet 1 returneres nest oftest og så videre, eller om man ønsker at distribusjonen skal være tilfeldig. En tilfeldig Zipf-distribusjon oppnås ved at rangordningen til tallene, altså hvilke tall som skal returneres hvor ofte, permuteres ved hjelp av *Knuth's shuffle*[33] før distribusjonsmetoden tas i bruk. Det er uansett viktig å merke seg at alle kildenodene må benytte den samme Zipf-distribusjonen dersom trafikken i nettet totalt sett skal ha en destinasjonsfordeling som er Zipf-distribuert. Dette vil også sikre at det totale trafikkbildet blir Zipf-distribuert selv om hver node vil droppe ett tall fra distribusjonen – nemlig det tallet som ville tilsvare at noden sender en pakke til seg selv. Hva som benyttes av uniform fordeling og Zipf-distribusjon vil gå klart frem når vi senere skal gå igjennom våre konkrete simuleringer.

DrainNode og datainnsamling

En destinasjonsnodes oppgaver i nettverkssimulatoren består av kun to ting: Den må forsikre seg om at pakker den mottar er tiltenkt den selv, samtidig som den må holde oversikt over hvor mange pakker den mottar og hvor lang tid disse pakkene har brukt gjennom nettverket. Det første punktet er viktig siden et nettverk som sender pakker til feil mottagere ikke er til å stole på, og dermed kan tyde på feil ved en simulering. Samtidig er punkt to, datainnsamling, viktig da det blant annet er på disse verdiene vi vil baserer oss når vi ut fra simuleringsresultater vurderer to forskjellige nettverksoppsett opp mot hverandre. Destinasjonsnodene henter unna trafikk fra nettverket med samme hastighet som den linkene opererer med. Med andre ord forutsetter vi at destinasjonsnodene alltid har kapasitet til å fjerne pakker de får tilsendt fra nettverket. Destinasjonsnodene har kun gjennomgått små endringer fra NetSim.

TrafficConsentrator

Som allerede nevnt er TrafficConsentrator (TC) en hjelpeklasse konstruert for å gi endenodene mulighet til å delta under automatisk konfigurering av nettverket. Denne egenskapen besittes fra før av svitsjene i nettverket. TC er derfor implementert som en subklasse av klassen Switch (kap 3.3.3). Dette bringer imidlertid med seg en utfordring. En svitsj introduserer en naturlig forsinkelse for all trafikk som passerer svitsjen på sin vei gjennom nettverket. En TC bør på sin side introdusere minimalt med forsinkelse for den trafikken som tilhørende endenode sender eller mottar. Utfordringen kan brytes ned i



Figur 3.6: Kamuflerer man innlinkene til TrafficConsentrator (TC) vil resultatet bli at trafikk til og fra henholdsvis DrainNode og SourceNode tilsynelatende kun passerer én link mellon nett og endenode.

---

to deler ved å se på de elementene som en TC introduserer i nettverket: en ny svitsj i form av en TC, og en ny toveis-link for å plassere seg selv mellom endenoden og nettverket.

En diskret hendelsesdreven simulator har den fordelen at man kan utføre så mange operasjoner man måtte ønske innen samme hendelse. Som vi husker har heller ikke en hendelse noen utstrekning i tid sett i forhold til den interne simulatortiden. Denne egenskapen utnytter vi for å skape en TC med minimal forsinkelse. En TC vil kontrollere sine innlinker og kamuflere disse ved at den alltid, i den grad det ut fra bufferplass er mulig, umiddelbart og uten forsinkelse vil flytte pakker som skal over disse linkene. På samme måte vil pakker flyttes fra innport til utport hos TC så raskt det lar seg gjøre. Utlinkene fra TC er imidlertid beholdt som normale linker for å bevare den forsinkelse pakker til og fra endenoden må påberegne i det de passerer første eller siste link på sin vei gjennom nettverket. Dette er illustrert ved figur 3.6. Det er forøvrig et poeng at TC utstyres med minimale buffer. Dette er viktig for at en TC ikke skal kunne fungere som et ekstra pakkebuffer mellom endenode og nettverk. Et slikt buffer vil kunne tilføre økt forsinkelse for pakker på sin vei gjennom nettverket.

I utgangspunktet var TC tenkt holdt uforandret fra NetSim. Vi skal imidlertid se at både svitsjer og linker har gjennomgått omfattende utvidelser i forhold til NetSim, noe som igjen har ført til at også TC har måttet forandres en god del. Vi skal ikke i detalj gå inn på hvilke forandringer og tilpasninger som er gjort, men konstatere at virkemåten beskrevet ovenfor er forsøkt beholdt, og at TC slik sett fortsatt fungerer på tilsvarende måte som den man finner i NetSim. Simuleringer av nettverk med og uten TC viser

da også at hensikten er oppnåd: Det tillegg man opplever i forsinkelse ved bruk av TC er neglisjerbart sammenliknet med den totale forsinkelse som en pakke opplever på sin vei gjennom nettverket. Når man i tillegg tar med i betraktningen at alle endenoder i nettverket vil oppleve den samme økning i forsinkelse, og at dette gjelder for alle testoppsett, så er det rimelig å anta at dette ikke får vesentlige konsekvenser for våre resultater, og at bruken av TC således verken svekker simulatorens troverdighet eller resultatenes gyldighet.

3.3.3 Svitsjene

NetSim støtter dynamisk rekonfigurering av svitsjene i nettverket. Det vil si at svitsjene kan videresende trafikk mellom endenodene i nettverket parallelt med at svitisjene rekonfigureres dersom topologien i nettverket skulle forandre seg. Her er det naturlig nok en forutsetning at det faktisk finnes ruteinformasjon i rutetabellene, og at denne informasjonen kan brukes til å finne en fullstendig rute fra kildenode til destinasjonsnode gjennom nettverket. Dynamisk rekonfigurering introduserer utfordringer som at gamle ruter kan bli gjort ugyldige, pakker kan bli urutbare i det nye ruter blir gjort tilgjengelige, vranglås kan oppstå og liknende. De simuleringer som er gjennomført som en del av denne oppgaven har imidlertid ikke hatt behov for å utnytte denne funksjonaliteten, siden topologien er holdt uforandret gjennom hver enkelt simulering. Alle svitjsene har derfor kun gjennomgått en konfigurasjonsfase som første ledd av hver simuleringen. I løpet av denne fasen samler svitsjene inn all nødvendig topologi-informasjon og bygger opp rutetabellene sine. Først når denne prosessen er ferdig starter endenodene trafikkinjeksjon i nettverket. Med andre ord vil alle svitsjene først befinne seg i en konfigurasjonsfase, før de går over i en stabil tilstand som videresendingsnoder. Tabell 3.3 viser klassene som representerer alternative algoritmer for å håndtere konfigurasjonsfasen, mens tabell 3.4 viser de tre klassene som er i bruk mens en svitsj er i videresendingsfasen.

Konfigurasjonsfasen

For irregulære topologier har NetSim kun støtte for up*/down*-ruting. Dette er implementert ved klassen UpDownPlugIn, som er en subklasse av klassen PlugIn. PlugIn er en abstrakt superklasse for alle innstikksmoduler som ønskes knyttet til svitsjene[6] i nettverket.

I det en ny simulering starter startes også up*/down*-konfigureringen ved at en av svitsjene blir bedt om å sende ut en kontrollpakke. Denne pakken indikerer at topologien har forandret seg, og vil igangsette en topologiinnsamlingsfase hos alle svitsjene i nettverket. Svitsjene vil i løpet av denne

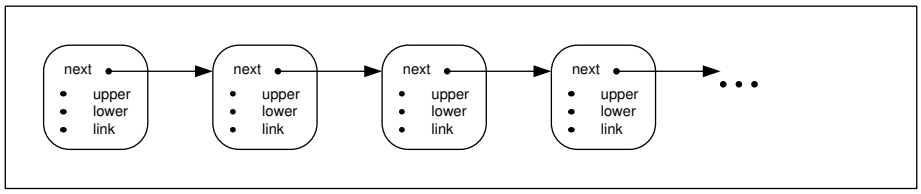
[6]Når vi snakker om nettverkets konfigurasjonsfase og konfigurering av svitsjene skal begrepet svitsj innbefatte instanser av både klassen Switch og av klassen TrafficConsentrator.

Rutealgoritme	**Beskrivelse**
PlugIn.java UpDownPlugIn.java*(h)*	PlugIn er en felles superklasse for alle innstikksmoduler for svitsjer, og da spesielt klassen UpDownPlugIn som er superklasse for alle rutealgoritmer implementert i forbindelse med denne oppgaven. UpDownPlugIn inneholder hjelpeklassen *Link*.
DetermUpDownPlugIn.java DetermShortUpDownPlugIn.java EthernetPlugIn.java*(h)* ShortestPathPlugIn.java*(h)* ShortestPathWeightedPlugIn.java*(h)*	Disse klassefilene implementerer forskjellige rutealgoritmer. De tre siste klassene inneholder hver sin variant av hjelpeklassen *ShortestPathNode*
PacketBuffer.java	Hjelpeklasse for rutealgoritmene.
DeadlockDetection.java*(h)*	Denne klassen håndterer vranglås i simulatoren. Klassen inneholder hjelpeklassen *DependentNode*.

Tabell 3.3: Oversikt over klassefilene som implementerer forskjellige rutealgoritmer, samt en egen klasse for håndtering av vranglås.

Svitsj	**Beskrivelse**
Switch.java RoutingTable.java*(h)* Arbiter.java	Disse tre klassene utgjør til sammen de aktive deler av en svitsj i videresendingsfasen. RoutingTable.java inneholder hjelpeklassen *OutPutChannelSet*.

Tabell 3.4: Oversikt over klassefilene som til sammen danner en svitsj i simulatoren.



Figur 3.7: En liste av link-informasjon bygget opp hos hver enkelt svitsj som en del av konfigurasjonsfasen.

---

fasen først sende ut informasjon som inneholder deres antatte plassering i up*/down*-treet. Denne utvekslingen pågår helt til en svitsj, direkte eller indirekte, av alle andre svitsjer har blitt godkjent som rot i det nye treet. Den nye roten vil så gi beskjed nedover i det nyopprettede up*/down*-treet om at et endelig tre er vedtatt. Samtidig sender roten informasjon om sine nærmeste svitsjer, noe som igjen igangsetter en utvekslingsfase hvor alle svitsjer rapporterer informasjon om sin endelige plassering og sine naboer i treet til roten. Roten vil igjen sørge for at all topologi-informasjon spres på nytt nedover i treet, og at alle svitsjer som et resultat av dette har en felles oppfatning av hele det nye up*/down*-treet. Svitsjene bruker så den topologi-informasjonen de nå besitter til å bygge opp friske rutetabeller. I det alle svitsjer har rutetabellen sin på plass er konfigurasjonsfasen over.

UpDownPlugIn.java har kun gjennomgått små forandringer fra NetSim. Retningen til linker mellom svitsjer på samme nivå i up*/down*-treet er forandret for at metoden skal ligge tettere opp til den originale definisjon av up*/down*[14]. I tillegg er det lagt til støtte for bedre kontroll av når svitsjene er ferdige med å bygge opp rutetabellene sine, slik at man kan starte trafikkinjeksjon umiddelbart etter at siste svitsj har rutetabellen sin klar. Begge disse forandringene er imidlertid av forholdsvis liten praktisk betydning. Det som er av større betydning er at up*/down*-ruting er adaptiv. Siden vi har valgt å holde oss til deterministiske rutealgoritmer har det vært behov for å modifisere NetSims up*/down*-implementasjon. De to rutealgoritmene *deterministisk up*/down** og *deterministisk korteste-vei-up*/down** fra kapittel 2.3.2 har derfor blitt implementert. I tillegg har vi implementert støtte for de to variantene av korteste-vei-ruting vi beskrev i kapittel 2.3.2, samt støtte for spenntre-metoden som benyttes av Ethernet.

Selv om vi ikke ønsker å benytte oss av NetSims UpDownPlugIn som rutealgoritme, inneholder klassen imidlertid metoder for å samle inn topologi-informasjon som det er hensiktsmessig å gjenbruke. Etter at topologi-innsamlingsfasen i UpDownPlugIn er unnagjort, men før rutetabellene bygges opp, sitter alle svitsjene i nettverket igjen med en datastruktur tilsvarende den vi ser i figur 3.7. Denne listen inneholder informasjon om

alle linker, hvilke to svitsjer hver enkelt link går mellom, *upper* og *lower*, og hvilken port linken er koblet til hos upper, *link*. Navnene upper og lower henspiller på at alle linker innen up*/down* har en retning, og at linken da har retning fra lower til upper[7]. Retningen til linkene i up*/down*-treet er imidlertid noe man kan se bort fra dersom man ønsker å bruke en annen rutealgoritme som for eksempel korteste-vei-ruting. Dermed inneholder faktisk denne listen tilstrekkelig informasjon om nettverkets topologi til at alle våre fem nye rutealgoritmer kan bygge opp korrekte rutetabeller. De fem klassene DetermUpDownPlugIn (DUD), DetermShortUpDownPlugIn (DSUD), ShortestPathPlugIn (SP), ShortestPathWeightedPlugIn (SPW) og EthernetPlugIn (ETH) er derfor alle implementert som subklasser av klassen UpDownPlugIn. Topologi-innsamlingsfasen er felles for alle sammen. Det som skiller dem er hvordan de utnytter informasjonen i listen til å populere rutetabellene.

DUD implementerer deterministisk up*/down*. Rutetabellen bygges da opp på følgende måte for svitsjen S: Først forsøker man å finne korteste vei fra svitsj S til destinasjsonsnode D ved kun å bruke linker i ned-retningen. Om en slik rute finnes, legges den inn i rutetabellen hos S. Finner man flere ruter som alle er like lange og kortest, legger man inn den første man finner. Dersom en rute som kun bruker linker i ned-retning ikke finnes, vil man i rutetabellen til S legge inn at neste steg på veien mot D skal følge en link i opp-retning. Om det finnes flere linker med opp-retning velger man en av dem som ligger langs en korteste vei mot roten. Som forklart i kapittel 2.3.2 er denne rutealgoritmen deterministisk. DUD er ikke veldig sofistikert når det gjelder å finne en smart vei videre, men er på den andre siden enkel å implementerer og krever ikke at man ved ruteoppslag tar hensyn til hvilken link en pakke kom inn til svitsjen via.

DSUD implementerer deterministisk korteste-vei-up*/down* og fremstår slik som en optimalisering av DUD. Rutetabellen hos svitsj S bygges opp slik: Først sjekker man om man kan nå destinasjonsnode D fra svitsj S ved å kun bruke linker i ned-retningen. Dersom man finner en eller flere slike ruter, legger man seg den korteste av dem på minne, R_{ned}. Finner man mer enn én rute som er kortest, velger man å ta vare på den første man finner. Så forsøker man for hver link med opp-retning fra S å se om man kan finne en rute til D ved å følge en eller flere linker med opp-retning, etterfulgt av null eller flere linker med ned-retning. Igjen tar man eventuelt vare på den korteste ruten, $R_{opp/ned}$, og igjen velger man den første man finner om det er flere å velge mellom. Til slutt sammenlikner man de to alternativene R_{ned} og $R_{opp/ned}$ og legger den korteste ruten inn i rutetabellen. Dersom vi finner

[7]Datatrær vokser gjerne opp ned. Det vil si at roten er øverst og bladene nederst. Det samme gjelder også for up*/down*-trær. Den noden som ligger lengst vekk fra roten er dermed lenger ned i treet enn en node som ligger nærmere roten. Siden up*/down*-linker, i den grad der er mulig, har retning mot roten, altså oppover i treet, har også linkene på samme måte retning fra en lav svitsj, *lower*, til en som ligger høyere, *upper*.

at R_{ned} og $R_{opp/ned}$ er like lange velges R_{ned}. Om $R_{opp/ned}$ er kortest må vi, som vi så i kapittel 2.3.2, også ta vare på en eventuell R_{ned} for pakker i retning D som kommer inn via en link i ned-retning. Dette må gjøres for at man ikke skal risikere å bryte up*/down*-invarianten. Brytes invarianten vil nettverket kunne gå i vranglås. DSUD er altså en optimalisering av DUD, hvor prisen man må betale er at man for hvert ruteoppslag må sjekke via hvilken link en pakke kom inn til svitsjen.

SP og SPW implementerer henholdsvis simpel og vektet korteste-vei-ruting slik de er definert i kapittel 2.3.2. En svitsj som benytter seg av SP eller SPW har derfor intet forhold til up*/down*-retningen til linker i listen gitt i figur 3.7. I stedet brukes listen til å bygge opp en graf $\mathcal{G}$ over alle svitjser og linker i nettverket. Ved konstruksjon av $\mathcal{G}$ representerer hjelpeklassen ShortestPathNode en svitsj samt informasjon om alle svitsjens linker til nabosvitsjer. $\mathcal{G}$ brukes så av hver enkelt svitsj som utgangspunkt for å beregne korteste vei fra seg selv til alle destinasjoner i nettverket. En svitsj som benytter seg av SP finner korteste rute til alle destinasjoner ved hjelp av et *korteste vei bredde først søk*[50] gjennom $\mathcal{G}$. På tilsvarende måte gjennomfører en svitsj som benytter SPW et *vektet korteste vei bredde først søk*[50]. Ved bruk av SPW vektes linken $l_{N_1N_2}$ mellom nodene N_1 og N_2 ved funksjon (2.1) fra kapittel 2.3.2:

$$v(l_{N_1N_2}) = g(N_1) + g(N_2) \tag{3.5}$$

$g(N)$ representerer graden til noden N.

Uavhengig av hvilken type søk man benytter vil hver enkelt svitsj S sette seg selv som startnode for søket. Resultatet av søket benyttes til å fylle ut rutetabellen til S. Dersom det fra S til destinasjonen D finnes flere ruter som alle kan sies å være kortest, vil søket gjennom $\mathcal{G}$ kun registrere den første ruten man finner. Slik sikrer man at rutetabellen kun inneholder én rute per kilde/destinasjons-par, og at rutealgoritmene implementert ved SP og SPW opptrer som deterministiske algoritmer.

ETH er den siste av rutealgoritmene simulatoren har fått støtte for. ETH er som navnet tilsier en implementasjon av Ethernet, eller mer presist en delvis implementasjon av IEEE 802.1D[2, 3] med automatisk utfylling av rutetabellen hos hver enkelt svitsj. Implementasjonen er delvis siden vi, for å dekke vårt behov, kun er ute etter å konstruere rutetabeller som tilsvarer dem man kunne hatt ved bruk av Ethernet. ETH implementerer derfor ikke fullt ut IEEE 802.1D, og har for eksempel ikke støtte for de forskjellige tilstandene til Ethernets spenntre-algoritme slik de ville utspilt seg i et reelt nettverk. Normalt vil rutetabellen i en Ethernet-svitsj gradvis bygges opp gjennom at svitsjen lærer av reell trafikk i nettverket[42]. Hos oss antar vi at svitsjene er ferdig utlært, eventuelt at rutetabellene er manuelt konstruert. Siden topologien ikke forandrer seg er det ikke behov for å vedlikeholde tabellen ved ny læring.

For å spare utviklingstid har ETH blitt basert på SP. Men mens en svitsj som benytter SP bruker hele grafen den har konstruert til å finne korteste vei, blir antall gyldige rute-alternativer kuttet ved bruk av ETH. Etter at ETH har konstruert en fullstendig graf over nettverkstopologien, reduseres nemlig antall linker i grafen ved at den konverteres til et *minimalt spenntre*[50]. Roten i det minimale spenntreet er den samme roten man ville hatt om man hadde benyttet seg av en av de to up*/down*-rutealgoritmene implementerte ved DUD eller DSUD. Etter at spenntreet er konstruert beregner ETH på vegne av svitsj S korteste vei fra S til alle andre noder ved hjelp av den samme algoritmen som SP benytter. I et spenntre finnes det imidlertid kun én vei til hver destinasjon. Dermed er den korteste vei mellom to noder det samme som den eneste vei mellom disse nodene[8]. Det neste steget for hver vei til korresponderende destinasjon legges inn i rutetabellen. Resultatet blir rutetabeller tilsvarende de man ville funnet om svitsjene benyttet seg av Ethernet, gitt at roten er den samme.

Det kan kanskje virke som om noen lite optimale designvalg er tatt i forbindelse med implementasjonen av de fem rutealgoritmene gitt ovenfor. Burde man ikke for eksempel for korteste-vei-ruting funnet en annen måte å samle inn topologi-informasjon på, enn å benytte seg av en metode som er implementert for up*/down*-ruting? Og hvorfor leter man etter korteste vei i et spenn-tre når det kun finnes én rute mellom to noder i et slikt tre? Er en slik implementasjon troverdig? Det er absolutt betimelig å stille disse og beslektede spørsmål. Det er imidlertid for denne oppgaven av liten interesse *hvordan* en svitsj har bygget opp rutetabellen sin. Selve konfigurasjonsfasen er uvesentlig all den tid rutetabellene har korrekt innhold og er klare til bruk i det vi starter trafikkinjeksjon i nettverket. Dette skyldes at all registrering av for oss interessante data forholder seg til trafikk i nettverket etter at konfigurasjonsfasen er over. På grunn av dette har følgende to faktorer fått prege implementasjonen av rutealgoritmene: Det har vært et poeng å holde programkoden enkel og oversiktlig for å minske muligheten for feil. Eventuelle feil i rutetabellene vil kunne være fatalt for gyldigheten til simuleringsresultatene. Samtidig har det vært et poeng å holde utvikingstiden nede, så fremt dette ikke har hatt vesentlige konsekvenser for eksekveringstiden til simuleringene. Her har gjenbruk spilt en viktig rolle. For eksempel er det å utveksle topologi-informasjon mellom svitsjene ved hjelp av reelle datapakker i nettverket en affære som potensielt bringer med seg både ekstra arbeid og nye mulige programfeil om man skulle skreddersy prosessen for hver enkelt rutealgoritme.

[8]Vi forutsetter da at man ikke bruker samme linken flere ganger, ei heller i motsatt retning av hva man tidligere har brukt. Om man kan bruke samme link flere ganger vil det jo kunne finnes uendelig mange veier mellom to noder.

Videresendingsfasen

Figur 3.4 viser de klasser som er i aktivt bruk mens svitsjene befinner seg i videresendingsfasen - altså etter at rutetabeller er bygget opp og nettverket kan sies å være i normal drift. Klassen Switch implementerer basisfunksjonene til en svitsj og styrer den rutinemessige håndteringen av pakker som kommer inn til svitsjen. Ruteoppslag gjøres ved hjelp av klassen RoutingTable. Siden pakker fra forskjellige innkanaler kan ønske å sende til samme utkanal samtidig, trengs en egen modul for å håndtere eventuelle konflikter. Klassen Arbiter implementerer den nødvendige funksjonalitet ved å styre tildeling av tilgang til en gitt utkanal dersom flere innkanaler ønsker den samme kanalen.

Svitsjene i NetSim benytter seg av kantbuffer. Hver virtuelle innkanal har et eget reservert innbuffer, mens hver virtuelle utkanal på tilsvarende måte har et eget reservert utbuffer. Alle buffer følger først-inn-først-ut-prinsippet. En svitsj kan kun gjøre ett ruteoppslag om gangen. Dersom flere pakker ankommer svitsjens forskjellige innbuffer samtidig, må de altså vente på at svitsjen sekvensielt behandler deres forespørsler om veien videre. Svitsjenes indre er imidlertid organisert slik at pakker fra forskjellige innbuffer kan forflyttes samtidig og uavhengig av hverandre gjennom svitsjen. Her er det forutsatt at ruteoppslaget for hver enkelt pakke er ferdig, og at det ikke er to pakker som skal til samme utbuffer.

Den orientering av svitsjenes indre struktur og virkemåte som er beskrevet i forrige avsnitt er valgt beholdt fra NetSim. Når det gjelder svitsjeteknikk har svitsjene i NetSim implementert kun støtte for wormhole-svitsjing. Denne funksjonalitet har vi utvidet med støtte for to varianter av lagre-og-videresend (LOV) og virtual cut-through. Som nevnt i kapittel 2, skal vi imiderltid kun forholde oss til LOV-svitsjing i denne oppgaven. Den første varianten av LOV tillater at man starter ruteoppslag så fort den del av en pakke som inneholder destinasjonsadressen har blitt mottatt hos svitsjen. Fordeling av utkanal og flytting av pakken gjennom svitjsens indre og over til rett utbuffer vil imidlertid ikke finne sted før hele pakken er mottatt. Den alternative LOV-metoden tillater ingen aktivitet, selv ikke ruteoppslag, før hele pakken er mottatt. Siden vi anser det som rimelig at en svitsj kan håndtere ruteoppslag parallelt med mottak av en pakke, er første alternativ den benyttede metode for alle simuleringer knyttet til denne oppgaven.

Arbiter står for tildeling av virtuelle utkanaler blant ventende virtuelle innkanaler. Dette betyr at Arbiter også må sørge for at ingen innkanaler utsultes. For å unngå utsulting tar Arbiter utgangspunkt i et sirkulært startpunkt når innkanaler skal gis tilgang til utkanaler. Om man sist startet med den virtuelle innkanalen v, vil man ved påfølgende tildelinger i tur og orden starte med alle andre virtuelle innkanaler, før man på nytt lar en tilordning starte med kanalen v igjen. Om svitsjen har n innkanaler vil altså hver n'te tilordning starte med innkanal v. Siden denne funksjonaliteten hos Arbiter

er beholdt fra NetSim, er det ikke gjort ytterligere resonnering rundt det å avgjøre om denne algoritmen er tilstrekkelig for å sikre at ingen innkanaler blir utsultet.

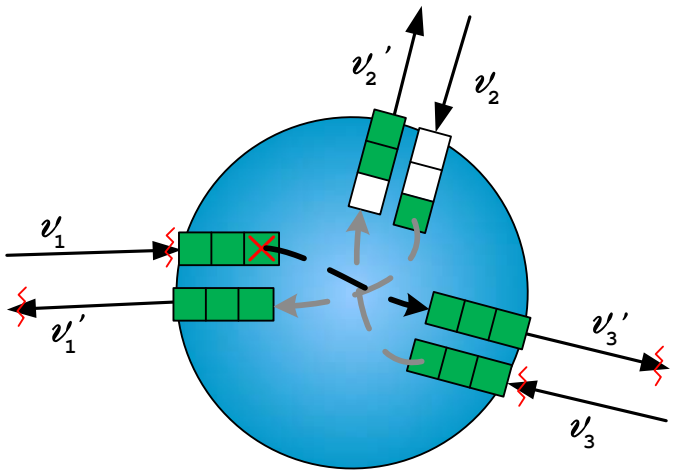
Vranglås

NetSim har ingen støtte for å avdekke eller oppløse vranglås i et nettverk. Dette skyldes naturlig nok at vranglås ikke kan oppstå ved bruk av de rutealgoritmer NetSim støtter. Up*/down*-ruting tillater ikke at en pakke som kommer inn til en svitsj via en link i ned-retning videresendes på en link i opp-retning. Nettopp denne begrensningen forhindrer mulige løkker i kanalavhengighetsgrafen, noe som igjen er ekvivalent med at up*/down*-ruting er en vranglåsfri rutealgoritme. Selv om vi har forandret litt på NetSims up*/down*-implementasjon for å gjøre den deterministisk har vi ikke forandret den grunnleggende begrensningen som hindrer overgang fra link i ned-retning til link i opp-retning. Vår utgave av up*/down*-ruting er derfor også vranglåsfri.

Vranglås kan heller ikke oppstå i Ethernet, men når det gjelder korteste-vei-ruting er saken en annen. Generelt vil et nettverk som benytter seg av korteste-vei-ruting kunne ha løkker i tilhørende kanalavhengighetsgraf. Det å avdekke og løse opp vranglås i et nettverk er derfor en nødvendig del av det å implementere korteste-vei-ruting. Her er det gitt at nettverket også benytter seg av flytkontroll. Dersom nettverket ikke benytter flytkontroll, vil pakker kastes i det buffer går fulle, og vranglås kan heller aldri oppstå.

I et reelt nettverk vil man typisk benytte seg av heuristiske metoder for å avgjøre om det er sannsynlig at et nettverk har gått i vranglås. Det er både vanskelig og ineffektivt å drive sentralisert innsamling av tilstandsinformasjon i et reelt nettverket for til en hver tid å kunne ta en korrekt avgjørelse om hvorvidt nettverket har gått i vranglås eller ikke[21]. For oss er det imidlertid ikke av interesse å studere vranglåsfenomenet i seg selv. Vi har derfor valgt den enkleste og mest presise løsningen for å avdekke og oppløse vranglås: Vi har implementert en sentralisert løsning hvor en egen modul i simulatoren holder oversikt over hele nettverkets tilstand og dermed til en hver tid kan kontrollere om deler av nettverket har gått i vranglås. Klassen DeadlockDetection implementerer denne funksjonaliteten. DeadlockDetection avdekker og oppløser vranglås på en optimal måte i den forstand at en vranglås vil avdekkes og løses umiddelbart. Siden en slik løsning er umulig å implementere i et reelt nettverk med dagens tekonologi, må DeadlockDetection sees på som en øvre grense for hvor raskt man kan oppdage og løse opp vranglås i et nettverk. Dette må man ta med i betraktning når man vurderer de forskjellige rutealgoritmer opp mot hverandre.

For å monitorere nettverket benytter DeadlockDetection seg av en *kanal-venter-på-graf* (KVG)[21, 48]. Nodene i en KVG tilsvarer de virtuelle kanaler som er reservert eller i bruk ved et gitt tidspunt i nettverket man monitorerer.



Figur 3.8: En svitsj med seks virtuelle kanaler. Figuren viser buffer knyttet til de virtuelle kanalene, og hvordan fulle buffer kombinert med flytkontroll kan føre til at en virtuell kanal må vente på en annen.

To noder har en rettet kant mellom seg dersom den virtuelle kanalen den ene noden representerer kan sies å *vente på* den virtuelle kanalen som den andre noden representerer. Kantens retning går fra den noden som tilsvarer den ventende kanal. Dersom man finner en *knute* i KVGen, er dette ensbetydende med at nettverket inneholder en vranglås[21, 48]. Vi skal komme tilbake til hva det vil si at en KVG inneholder en knute, men la oss først presisere hva som skal til for at en kanal kan sies å vente på en annen kanal.

Figur 3.8 viser en svitsj tilkoblet seks virtuelle kanaler. En kanal tar form av en heltrukket pil som viser kanalens retning. Hver kanal har et tilhørende buffer med plass til tre pakker. En grønn firkant illustrerer at en plass i et buffer er i bruk, mens en hvit firkant illustrerer en ledig plass. v_k og v'_k benevner henholdsvis inn- og utkaneler, hvor v_k og v'_k gjerne kan tenkes å til sammen danne en toveis-link. Alle buffer følger først-inn-først-ut-prinsippet, noe som medfører at dersom den pakken som har ligget lengst i et buffer ikke kan sendes videre, kan heller ingen andre pakker i samme buffer videresendes. En rød strek over hode til en pil markerer at den virtuelle kanelen har aktivert flytkontroll. En kanal v_i *venter på* kanal v_j dersom følgende tre krav er oppfylt[9]: 1) v_i har flytkontroll skrudd på og v_i's innbuffer er så fullt at

[9]For nettverk som benytter seg av wormhole-svitsjing kompliseres disse tre kravene av at en pakke alene kan legge beslag på flere virtuelle kanaler samtidig. Siden vi kun forholder oss til LOV-nettverk er imidlertid de tre kravene slik de er formulert her tilstrekkelig.

flytkontrollen ikke vil bli skrudd av før minst en pakke fjernes fra bufferet. 2) Første pakke i v_i's innbuffer har v_j som en potensiell rute videre. 3) v_j's utbuffer er fullt og v_j har flytkontroll aktivert. I figur 3.8 ser vi at første pakke i v_1's innbuffer, merket med et rødt kryss, ønsker å benytte v_3'. v_3''s utbuffer er imidlertid fullt. Siden både v_1 og v_3' har aktivert flytkontroll kan vi slå fast at v_1 venter på v_3'. Legg merke til at v_3 ikke kan sies å vente på v_2' siden v_2' har en ledig plass i sitt utbuffer. v_2 kan heller ikke sies å vente på v_1' til tross for at første pakke i v_2's innbuffer ikke kan flyttes over til v_1''s utbuffer. Dette skyldes at v_2 fortsatt har ledig plass i sitt eget innbuffer, og dermed ikke aktivert flytkontroll for kanalen.

En mengde noder, N, i en KVG utgjør en *knute* i grafen dersom det er slik at for alle noder $n \in N$ så er mengden av alle de noder man kan nå ved å fra n følge en eller flere rettede kanter i grafen nøyaktig lik mengden N selv. Generelt kompliseres det å finne en knute i en KVG av det faktum at en virtuell kanal potensielt kan vente på flere andre virtuelle kanaler. Dette skyldes at det kan være flere lovlige veier videre for en pakke som venter på å bli videresendt hos en svitsj. Vi arbeider imidlertid kun med deterministiske rutealgoritmer hvor det alltid bare finnes en lovlig vei videre. En virtuell kanal kan da maksimalt vente på én annen virtuell kanal, og det å finne en knute i en KVG blir redusert til det å finne en løkke i den samme grafen.

I tillegg er det mulig å forenkle KVGen ved å vurdere endenodenes rolle i nettverket. Vi opererer med destinasjonsnoder som med innlinkens hastighet henter ut trafikk fra nettverket. Dermed kan ikke en destinasjonsnode stoppe trafikken fra nettverket, og da ikke heller delta aktivt i å danne en vranglås. En kildenode kan på grunn av flytkontroll måtte vente med å sende pakker ut i nettverket, og kan slik indirekte bli påvirket av vranglås. Vi benytter oss imidlertid av LOV så en pakke som kilden er i ferd med å sende ut kan aldri ha kommet lenger enn til innbuffer hos første svitsj. Dermed kan denne pakken ikke heller forårsake at andre enn denne kilden selv må vente på den. Totalt tilsier dette at en endenode i et LOV-nettverk ikke kan delta aktivt i å danne en vranglås, og at det dermed er tilstrekkelig at DeadlockDetection forholder seg til de virtuelle kanaler som går mellom svitsjene i nettverket.

Når en vranglås først er oppdaget, har DeadlockDetection støtte for to alternative måter å løse opp vranglåsen på. Den første metoden plukker ut én av de virtuelle kanalene som inngår i vranglåsen og kaster første pakke i tilhørende innbuffer. Den alternative metoden kaster første pakke i innbuffer til hver eneste virtuelle kanal som direkte inngår i vranglåsen. Dette siste alternativet går altså mer drastisk til verks, og kaster potensielt langt flere pakker per vranglås. Idéen bak det å kaste mange pakker når nettet først er låst er å minske sjansen for at vranglås raskt gjenoppstår. Hvilke av de to metodene som benyttes ved en gitt simulering vil bli oppgitt.

Det kan være vanskelig å avgjøre om selve implementasjonen av DeadlockDetection er feilfri i den forstand at alle vranglåser oppdages og at man samtidig ikke oppdager falske positiver. Siden vår implementasjon opptrer

Link	Beskrivelse
VirtualChannel.java PhysicalLink.java PhysicalLinkForTC.java Wire.java*(h)* RingBuffer.java RoundRobin.java	Disse klassene implementerer forskjellige egenskaper ved en link i nettverket. Wire.java inneholder hjelpeklassen *FlitOnWire*.
FlowControlUnit.java*(h)*	Denne klassen implementerer flytkontroll. Klassen inneholder hjelpeklassen *FCMessage*.

Tabell 3.5: Oversikt over klassefilene som til sammen implementerer en link mellom to noder i simulatoren (inklusive buffer).

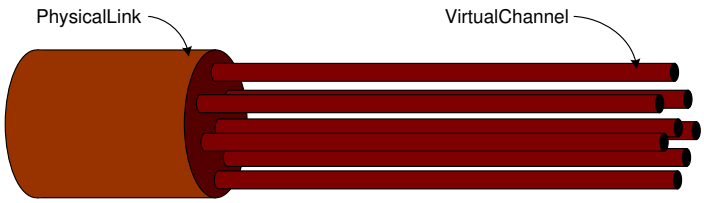
som en selvstendig modul i simulatoren uten å påvirke trafikken i nettverket med mindre vranglås faktisk oppstår, kan det være fristende å la DeadlockDetection kjøre også for de rutealgoritmer hvor vranglås ikke skal kunne finne sted. Dersom DeadlockDetection faktisk avdekker og løser opp vranglås i SP og SPW, men for DUD, DSUD og ETH ikke oppdager vranglås, styrker dette både troen på at de ulike rutealgoritmene er korrekt implementert og at DeadlockDetection avdekker vranglås, men ikke oppdager falske positiver. På grunn av dette har vi valgt å alltid benytte DeadlockDetection-modulen ved simulering, uavhengig av hvilken rutealgoritme som faktisk er i bruk. Så langt er det ingen ting som tyder på at DeadlockDetection avdekker falske positiver eller lar reelle vranglås-situasjoner gå upåaktet hen.

3.3.4 Linkene

Tabell 3.5 viser klassefilene som til sammen implementerer en link i simulatoren. En enveis-link mellom to noder består av en fysisk link med et tilhørende sett av virtuelle kanaler. Figur 3.9 illustrerer dette. Klassen PhysicalLink (PL) implementerer en fysisk link, mens klassen VirtualChannel (VC) implementerer en virtuell kanal. Ønsker man å benytte seg av toveis-linker, kan en slik link bygges opp av to enveis-linker, en i hver retning.

Kun én virtuell kanal kan til enhver tid benytte en gitt link. Vi tillater altså ikke at to forskjellige virtuelle kanaler sender informasjon parallelt over samme fysiske link. PL sin oppgave i simulatoren er derfor å styre hvilken av de tilhørende VCene som til en hver tid skal få benytte linken. PL har kun gjennomgått små forandringer fra NetSim.

Som forklart i kapittel 3.3.2 har vi av og til behov for å skjule enkelte linker i det vi kobler endenoder til resten av nettverket. Behovet melder seg for de testoppsett hvor vi benytter oss av klassen TrafficConsentrator. For å skjule en link lar vi tilgangen til linken håndteres av en spesialutgave av PL,



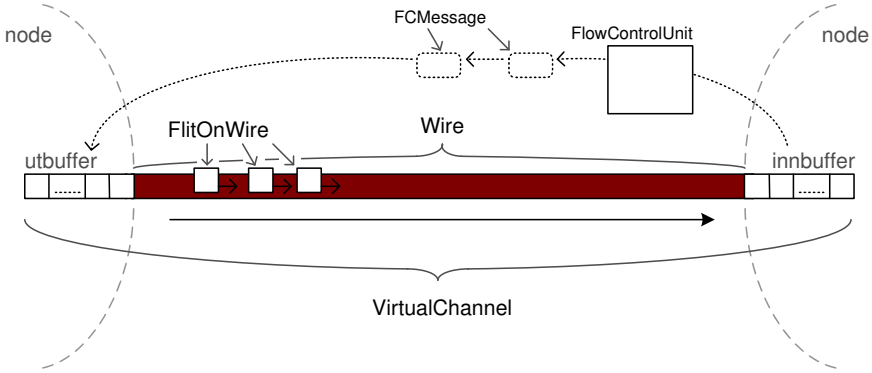
Figur 3.9: En fysisk link deles gjerne opp i et sett av virtuelle kanaler.

PhysicalLinkForTC. PhysicalLinkForTC vil flytte pakker over alle assosierte virtuelle kanaler med minimalt tillegg i forsinkelse, nettopp i et forsøk på å skjule linkens eksistens.

I et reelt nettverk vil alle buffer være knyttet til en node. En svitsj som benytter seg av kantbuffer har reserverte inn- og utbuffer assosiert med sine inn- og utlinker. På samme måte har en endenode inn- og utbuffer knyttet til sine respektive inn- og utlinker. Dersom linkene er delt opp i virtuelle kanaler vil hver kanal ha et buffer knyttet til seg i hver ende av den virtuelle kanalen. Siden alle buffer knyttes til en bestemt link eller en bestemt virtuell kanal, har NetSim implementert alle buffer som en del av klassen for virtuelle kanaler, VC. En slik løsning gir på en oversiktlig måte kontroll over hvordan man håndterer informasjon som forflytter seg fra utbuffer hos en node, over den virtuelle kanalen, og frem til innbuffer hos mottager. Integrasjonen av buffer i klassen for virtuelle kanaler er valgt beholdt fra NetSim. Bufferhåndteringen er utvidet med støtte for at VC kan operere med forskjellig størrelse på de to bufferne som er knyttet til kanalen. Dette er nødvendig for at man ved simulering skal kunne konstruere svitsjer som har forskjellig størrelse på inn- og utbuffer.

VirtualChannel har videre fått utvidet funksjonalitet gjennom to nye moduler, FlowControlUnit og Wire. Innføringen av disse to klassene med tilhørende hjelpeklasser, har ført til en kraftig modifikasjon av klassen VC sett i forhold til den man finner i NetSim. Figur 3.10 viser hvordan en virtuell kanal nå totalt består av klassene VirtualChannel, FlowControlUnit, FCMessage, Wire og FlitOnWire.

Klassen FlowControlUnit (FCU) håndterer flytkontroll for en virtuell kanal. To verdier knyttet til innbuffer hos mottagernoden styrer når flytkontroll skal skrus av og på. FCU bruker disse verdiene til å generere flytkontroll-beskjeder i form av FCMessage'er (FCM). Flytkontroll-beskjedene vil etter en justerbar forsinkelse avgjøre om flytkontroll skrus av eller på hos avsendernoden. Vi har gjort den forenkling å anta at FCMene er så små at de kan ignoreres i forhold til annen trafikk i nettverket, eventuelt at flytkontrollin-



Figur 3.10: Figuren illustrerer de forskjellige komponenter som til sammen utgjøre en virtuell kanal.

formasjonen kan haike med andre pakker som sendes i nettverket[10]. Merk imidlertid at en FCM beveger seg i motsatt retning av annen trafikk relatert til en gitt virtuell kanal. Med andre ord gir denne forenklingen kun mening dersom vi opererer med toveis-linker i nettverket. Da er det rimelig å kunne anta at en FCM benytter en tilsvarende motsatt rettet link. Siden utbuffer hos avsendernoden og innbuffer hos mottagernoden er styrt av den samme VCen, ender vi altså med en løsning hvor kun én virtuell kanal i simulatoren er involvert i forhold til de flytkontroll-beskjeder som sendes. Beskjedenes vandring er illustrert ved stiplede piler i figur 3.10.

FCU støtter flere grader av flytkontroll. Man kan benytte *hard* flytkontroll. Da vil den virtuelle kanalen slutte å sende pakker i perioder hvor flytkontrollen er skrudd på. Det er denne formen for flytkontroll vi har omhandlet tidligere i oppgaven. Alternativt kan man benytte en mykere variant av flytkontroll hvor den virtuelle kanalen bremser trafikken uten å stoppe den helt. I perioder hvor flytkontroll er skrudd på kan man da for eksempel velge å sende trafikk med 1/2, 1/10 eller 1/100 av kanalens kapasitet. Det er også støtte for å benytte kanaler uten flytkontroll. Man bør imidlertid merke seg at dersom man benytter myk eller ingen flytkontroll, vil pakker kastes hos mottagernoden dersom avsender sender raskere enn det mottager klarer å håndtere.

Når man sender en pakke fra en node til en annen, vil det mens pakken sendes være deler av pakken som verken befinner seg hos avsender eller mot-

[10]På engelsk bruker man gjerne betegnelsen *piggybacking* om det å la kontrollinformasjon haike med annen trafikk i nettverket.

Verktøy	Beskrivelse
Activator.java ResetCounters.java	Disse to klassene tar seg av aktivering av nettkverket og nullstilling av tellere etter at nettverkskonfigurasjonen er ferdig og nettverket kan sies å være i stabil drift.
ConfigurationTools.java	Verktøy for oppsett av nettverk og simulering.

Tabell 3.6: Oversikt over klassefilene for diverse hjelpeverktøy.

tager. Deler av pakken vil befinne seg på det fysiske mediet på vei fra den ene noden til den andre. Klassen Wire implementerer ved hjelpe av klassen FlitOnWire støtte for at deler av en pakke kan befinne seg på linken. De fysiske egenskaper til en link, dens lengde og båndbredde, vil avgjøre hvor stor del av en pakke som befinner seg på den aktuelle linken. Lengden på linken avjgør også hvor lang tid det tar for pakken å forflytte seg fra avsender til mottager. Ved hjelp av klassen Wire kan det fysiske transmisjonsmediet gjenspeiles i simulatoren.

3.3.5 Generelle hjelpeklasser

Tabell 3.6 inneholder tre generelle hjelpeklasser. Activator og ResetCounters er nye klasser, mens ConfigurationTools er en utvidet utgave av tilsvarende klasse hos NetSim.

Klassen Activator brukes for å klargjøre nettverket for ordinær trafikk etter at konfigurasjonsfasen er over. Dette innebærer blant annet at ønsket form for flytkontroll aktiveres[11], og at alle kildenoder får beskjed om at trafikk kan sendes ut i nettverket. Alle kildenoder er i utgangspunktet passive da vi ikke er interessert i å sende trafikk mellom endenoder før nettverket er ferdig konfigurert. De første pakkene som sendes vil oppleve en kunstig nettverkstilstand i form av at nettverket er mer eller mindre tomt for pakker. Disse tidlige pakkene vil da oppleve en tilgang på ressurser som ikke nødvendigvis er representativt for et nettverk i normal drift. Derfor nullstilles all bokføring av antall sendte og mottatte pakker, brukt forsinkelse gjennom nettverket et cetera, en gitt tid etter at Activator igangsatte trafikkinjeksjon hos endenodene. Nullstillingen håndteres av klassen ResetCounters.

Klassen ConfigurationTools inneholder blant annet en rekke funksjoner for å konstruere virtuelle kanaler, tilordne virtuelle kanaler til fysiske linker, koble sammen endenoder og svitsjer, og alt i alt en rekke funksjoner for å konstruere tilfeldige nettverkstopologier.

[11] Vi benytter alltid hard flytkontroll under konfigurasjonsfasen for ikke å miste pakker mens svitsjene bygger opp sine rutetabeller.

3.4 Simulatorens tidsperspektiv og oppløsning

Et reelt nettverk har et kontinuerlig tidsperspektiv i den forstand at en gitt begivenhet i nettverket ikke er låst til en diskret tidsskala. Riktignok vil en endenode eller en svitsj gjerne ha en krystall som faktisk styrer nodens klokke, og dermed legger føringer på når den aktuelle nodens handlinger kan inntreffe, men med mindre hele nettverket er synkronisert vil ikke nodene i nettverket være låst til den samme diskrete tidsskala. I en diskret hendelsesmodell opererer man imidlertid med en global klokke styrt av hendelseskøen. Dette medfører en utilsiktet global synkronisering av alle noders klokke i nettverket. Siden vi i utgangspunktet ikke ønsker å se på synkrone nettverk, er det viktig at innføringen av den globale klokken i minst mulig grad forårsaker at enheter i nettverket går i takt. For eksempel bør aktiviteter og deres varighet defineres relativt til den globale klokken og ikke benytte seg av egenskaper til den globale klokkens tallverdi. La oss for eksempel si at en bestemt hendelse h_a ønskes hvert hundrede klokketikk ved noden a. Dette bør defineres ved at hendelsen finner sted ved tiden $t_{h_a} + x * 100$, hvor $x = 0, 1, 2, 3, ...$ og t_{h_a} indikerer den globale tiden første gang hendelsen finner sted. Alternativt kunne man la h_a finne sted når verdien til den globale klokken heltallsdividert på 100 gir null i rest. Da vil h_a fortsatt finne sted hvert hundrede tikk hos noden a, men man øker sjansene for at forskjellige hendelser skal gå i takt. h_a vil for eksempel alltid gå i takt med en tilsvarende definert hendelse h_b hos noden b. Ved å bruke første alterntiv for å angi tiden for h_a, og tilsvarende h_b, vil h_a og h_b kun gå i takt dersom $t_{h_a} = t_{h_b} + 100 * y, y \in \mathbb{Z}$.

La oss nå øke tidens oppløsning til det tidobbelte ved å dele hvert klokketikk inn i ti nye tikk. Med en slik forfinet tidsskala vil vi kunne minske sjansene for at forskjellige hendelser går i takt. Det er imidlertid et poeng at oppløsningen ikke er kunstig høy. En for høy oppløsning vil kunne komplisere håndteringen av tid i simulatoren. Samtidig kan den gi inntrykk av et kunstig høyt presisjonsnivå sett i forhold til alle forenklinger som er gjort ved modellering og implementering av andre delere av simulatoren.

Valg av tidsoppløsning er for vår simulator dels basert på designvalg tatt ved implementasjon av NetSim, og dels sett i sammenheng med de hendelser som skal simuleres og hvor lang tid disse tar i forhold til hverandre. Oppløsningen er valgt tilstrekkelig høy til at de hendelser som spenner over kortest tid ikke blir tilordnet en kunstig lang tid kun fordi simulatorens klokketikk er for lange. Dette er også sett i sammenheng med simulatorens interne virkemåte. Simulatoren arbeider internt på flitt-nivå. Denne egenskapen er bevisst valgt beholdt fra NetSim som primært ble utviklet med tanke på wormhole-svitsjing. Selv om det i et LOV-nettverk er naturlig at for eksempel flytkontroll opererer på pakkenivå, er det andre deler av simulatoren som har behov for høyere oppløsning. For eksempel vil for korte linker kun en liten del av en pakke kunne befinne seg på linken. Det er derfor naturlig at

klassen Wire arbeider på flitt-nivå.

3.5 Tilfeldighet og simulering

Datamaskiner er deterministiske. Et dataprogram fungerer som en funksjon hvor man for et gitt sett med variabelverdier V alltid vil få samme resultat ut. Det er da en fortusetning at V også omhandler alle verdier programmet henter inn fra sine omgivelser. En datamaskins deterministiske natur er i de fleste tilfeller svært gunstig da den tillater oss å implementere forutsigbare programmer med et klart definert sett av funksjoner tilpasset konkrete bruksområder. Determinisme er således også en grunnleggende egenskap ved den simulatoren vi har implementert.

Av og til ønsker vi oss imidlertid tilfeldighet. For eksempel er vi avhengig av noe som tilsynelatende er tilfeldig for å kunne generere trafikk med utgangspunkt i ulike sannsynlighetsmodeller. Det samme gjelder ved oppsett av ulike topologier; det er en fordel om simulatoren selv kan konstruere et sett med tilfeldig valgte topologier for våre simuleringer. Vi ønsker imidelertid et visst inngrep med tilfeldigheten. Når vi for eksempel skal teste ulike rutealgoritmer mot hverandre, ønsker vi å simulere over et tilfeldig utvalg av topologier, men samtidig kunne styre tilfeldigheten slik at de genererte topologiene kan gjenskapes for flere ulike simuleringsoppsett. Slik kan vi teste ulike rutealgoritmer over det samme sett av tilfeldig valgte topologier. Det kan synes motstridene å skulle styre tilfeldighet, men den ønskede effekt oppnår vi ved bruk av det vi kaller *pseudo-tilfeldighet*. En generator av pseudo-tilfeldige tall produserer en tallrekke som tilsynelatende er tilfeldig, gjerne med utgangspunkt i en statistisk sannsynlighetsmodell, men som likevel alltid produserer den samme tallrekken dersom man gir generatoren det samme *frøet*. Frøet består typisk av et tall man gir generatoren som utgangspunkt for den tallrekken som skal genereres. Merk at frøet selv ikke spiller noen rolle innad i selve tallrekken som genereres. Frøets rolle er kun å gi selve tallrekke-generatoren et tilfeldig utgangspunkt. Ved å velge ut tilfeldige frø får man generert pseudo-tilfeldige tallrekker, som igjen kan brukes til å danne pseudo-tilfeldige topologier eller pseudo-tilfeldig trafikk med utgangspunkt i ulike sannsynlighetsmodeller. Ved våre simuleringer er tilfeldige frø for ulike simuleringer valg fra de tusen første primtall[12]. De tusen første primtall er gjengitt i tillegg B. Benyttede tall er markert i tillegget.

[12]I tillegg har et sett med simuleringer blitt kjørt med tallet 111 som frø. Tallet 111 ble valgt ved hjelp av en egen form for tilfeldighet; en trykkfeil i form av et ettall for mye i et skript for generering av simulatoroppsett. Dette frøet har siden blitt valgt benyttet på lik linje med andre frø.

3.6 Simulatorens validitet

For å kunne stole på de resultater som frembringes gjennom simulering er det vesentlig at simulatoren i størst mulig grad er feilfri. Som tidligere nevt kan feil oppstå på flere nivåer, både ved modellering og implementering av simulatoren og dens forskjellige moduler. Når det gjelder modelleringsarbeidet og tilhørende oversettelse av vår virkelighetsforståelse, er denne forsøkt sikret gjennom litteraturstudier samt diskusjon både med kolleger, medstudenter og ikke minst mannen bak NetSim, Olav Lysne. De forenklinger som har blitt gjort i forhold til virkeligheten er nevnt tidligere i dette kapittelet.

Når det gjelder selve programkoden er denne ikke forsøkt formelt verifisert[18]. En formell verifikasjon av simulatoren ble vurdert å være for omfattende til å være praktisk realiserbar innen denne oppgavens rammer. For likevel å sikre høy grad av feilfrihet har forskjellige deler av programkoden blitt utsatt for både dynamisk og statisk testing[43]. All kode har blitt utsatt for grundig kodelesing. Alle vesentlige deler av programkoden er håndsimulert med papir og blyant, inklusive testing av grenseverdier for ulike datastrukturer. I tillegg er topologi-informasjon og rutetabeller for enkelte små nettverk skrevet ut under simulering, for i etterkant å kunne kontrollere at rutetabellene bygges opp riktig ved bruk av de ulike rutealgoritmene. Som tidligere nevnt blir alle simuleringer testet for vranglås - selv de simuleringer som benytter seg av vranglåsfrie rutealgoritmer. Og ikke minst: Alle resultater blir vurdert med et kritisk blikk. Det er viktig ved analyse av resultatene å huske på at simulatoren ikke er en perfekt gjengivelse av virkeligheten. Som vi skal se i neste kapittel er alle resulater vi har fått intuitivt forklarbare. Dette styrker simulatorens troverdighet.

Det er selvfølgelig vanskelig å kunne hevde med absolutt sikkerhet at simulatoren ikke inneholder feil eller forenklinger som kan ha vesentlig innvirkning på resultatene. Det er imidlertid heller ingen ting som tyder på at vår simulator faktisk inneholder slike grove feil eller overforenklinger. Dette inntrykket forsterkes av at simulatoren gir resulateter som stemmer over ens med resultater gitt av andre simulatorer, og at alle resultater som nevnt er intuitivt forklarbare. I tillegg er deler av simulatorens kjernefunksjonalitet nedarvet fra NetSim. Til sammen har dette skapt den nødvendige tiltro til at våre resultater er gyldige under de forutsetninger som blir gitt i løpet av oppgaven.

3.7 Oppsummering

Vi har i løpet av dette kapittelet gått grundig igjennom ulike sider ved den simulatoren som er benyttet for å besvare våre problemstillinger. Vi har sett på simulatorens virkemåte, og forklart og begrunnet de designvalg som er tatt.

Det viste seg å bli et langt større implementasjonsarbeid å skulle tilpasse NetSim til våre problemstillinger enn først antatt. Faktisk har så og si alle klasser fra NetSim på en eller annen måte blitt modifisert. I tillegg har en rekke nye klasser blitt lagt til. For å presisere hvilket arbeide som er gjennomført som del av denne oppgaven skal vi gå igjennom de viktigste punktene i denne oppsummeringer.

Kildenodene i nettverket har fått sin syntetiske trafikkgenerator utvidet med to nye statistiske modeller. Støtte for selv-lik hyppighet av pakkegenerering av blitt implementert. Samtidig har destinasjonsdistribusjonen blitt utvidet med støtte for Zipf-fordeling.

Svitsjene i nettverket har fått kraftig utvidet funksjonalitet. Det har blitt implementert støtte for både lagre-og-videresend- og virtual cut-through-svitsjing. I tillegg har svitsjene fått implementert støtte for fem nye rutealgoritmer; to deterministiske utgaver av up*/down*-ruting, to deterministiske utgaver av korteste-vei-ruting og støtte for rutetabeller bygget opp tilsvarende det man vil finne i tradisjonelt Ethernet. Som en del av implementasjonen av de to korteste-vei-rutealgoritmene, har det i tillegg vært nødvendig å implementere støtte for å oppdage og oppløse vranglås i et simulert nettverk.

Linkene i simulatoren har først og fremst gjennom klassen VirtualChannel blitt totalt skrevet om. Fra å være en svært enkel klasse med begrenset funksjonalitet fremstår nå VirtualChannel, sammen med sine hjelpeklasser, som en voksen modul tilpasset våre problemstillinger. Først og fremst er linkene nå utstyrt med en fleksibel flytkontroll-modul hvor man kan variere både graden av benyttet flytkontroll, og flytkontrollens triggerverdier. I tillegg har det blitt implementert støtte for å modellere at pakker bruker tid på å bevege seg fra utbuffer hos en node og over til innbudder hos en annen. Det har med andre ord blitt implementert støtte for at deler av en pakker (eller mer) "ligger på linken". Som vi husker er bufferne i nettverket knyttet til VirtualChannel. Disse bufferne kan nå ha ulik størrelse for samme kanal, og ikke minst; man kaster pakker dersom et innbuffer er fullt.

All kildekode til simulatoren ligger tilgjengelig via WWW. Under følger URLer som både gir tilgang til kildekoden og til de jar-filer benyttet for simuleringer tilknyttet denne oppgaven. Man finner også en lite readme-fil som helt enkelt forklerer hvordan ulike parametre til simulatoren kan settes via kommendolinjen.

`http://heim.ifi.uio.no/~ernstgr/HF/` (readme-fil og jar-filer)
`http://heim.ifi.uio.no/~ernstgr/HF/Network/` (klasse-filene)

Kapittel 4

Simuleringer, resultater og analyse

I kapittel 1 introduserte vi oppgavens problemstillinger og begrunnet valg av simulering som metode. Bakgrunnen for oppgaven ble utdypet og presisert i kapittel 2 før vi i kapittel 3 gjennomførte en grundig gjennomgang av vår egenutviklede verktøykasse, selve simulatoren. I dette kapittelet skal vi gå igjennom de simuleringsoppsett som er benyttet for å besvare oppgavens problemstillinger. Vi skal presentere resultater fra simuleringene, analysere dem og vurdere dem i lys av våre problemstillinger.

4.1 Simuleringsoppsett

I det vi ønsker å gjennomføre en konkret simulering må vi ta stilling til en del spørsmål som omhandler eksakte størrelser ved de nettverk vi skal etterlikne. Vi må avgjøre hvor mange svitsjer, endenoder og linker våre simulerte nettverk skal bestå av, og vi må konkretisere de ulike komponentenes oppførsel og egenskaper. Dette innebærer blant annet å bestemme størrelser på alle inn- og utbuffer, velge pakkestørrelse, avgjøre linkenes egenskaper, samt presisere hva slags type syntetisk trafikkgenerator som er valgt benyttet. Disse avgjørelsene kommer i tillegg til de antagelser og valg vi har gjort på vår vei gjennom kapittel 1, 2 og 3. I det man treffer de konkrete avgjørelser reduseres et uendelig, mangedimensjonalt rom av mulige simuleringsoppsett til en håndterbar størrelse vi kan benytte til å belyse våre problemstillinger. Et konkret simuleringsoppsett som entydig definerer alle variable størrelser knyttet til det nettverket vi ønsker å simulere, sier vi gjerne at simulerer en konkret *nettverkskonfigurasjon*.

Før vi går nærmere inn på de konkrete simuleringer vi har gjennomført, skal vi gjennomgå de generelle egenskaper og antagelser som er felles for alle våre simuleringsoppsett. Vi vil samtidig repetere viktige avgjørelser fra tidligere kapitler, slik at det ikke levnes tvil om hvilke forutsetninger og

antagelser våre resultater hviler på.

Endenodene

Alle våre endenoder er i stand til å hente unna trafikk fra nettverket ved link-hastighet. Et innbuffer hos en endenode vil altså aldri fylles. Videre er alle våre endenoder feilfrie i den forstand at de sender og mottar trafikk uten at feil oppstår i de pakkene som behandles.

Endenodene genererer trafikk med en hyppighet som er statistisk selv-lik. Dette ble beskrevet i kapittel 3.3.2. Det nøyaktige påtrykket nodene belaster nettverket med vil varieres innen hvert enkelt simuleringsoppsett. Slik søker vi å studere hvordan forskjellige nettverkskonfigurasjoner håndterer ulik belastning. Pakkestørrelsen er satt til 1500 byte uavhengig av påtrykk.

Når det gjelder destinasjonsdistribusjon har vi implementert støtter for både uniform fordeling og Zipf-distribusjon. For å holde antallet simuleringer nede på et håndterbart nivå har vi imidlertid sett oss nødt til å begrense oss til å benytte kun én av disse to distribusjonsmodellene. Valget falt på uniform fordeling siden dette er den mest brukte fordelingsmodellen for destinasjonsdistribusjon ved simulering av tett koblede nettverk[21]. Merk at vi alltid unndrar en endenodes egen adresse fra destinasjonsfordelingen, slik at en node aldri genererer pakker som er adressert til noden selv.

Svitsjene

Alle våre svitsjer benytter seg av kantbuffer. Bufferne følger først-inn-først-ut-prinsippet, og har alltid en størrelse tilsvarende et multiplum av benyttet pakkestørrelse. Den nøyaktige størrelsen til inn- og utbuffer angis ved hvert enkelt simuleringsoppsett.

Lagre-og-videresend benyttes som svitsje-teknikk. Vi tillater at man starter selve ruteoppslaget etter at pakkehodet har ankommet svitsjen, men kun dersom hodet ligger først i det aktuelle innbufferet. En pakke vil imidlertid ikke få tildelt plass i et utbuffer, og dermed heller ikke kunne forflyttes gjennom svitsjen, før hele pakken har ankommet svitsjen.

Svitsjenes indre nettverk opererer med samme hastighet som linkene i det ytre nettverket. Det indre nettverket er samtidig av en slik karakter at en svitsj parallelt og med full hastighet er i stand til å håndtere trafikkstrømmer fra alle innbuffer til alle aktuelle utbuffer, uten at strømmene forstyrrer hverandre. Det er da gitt at to strømmer ikke konkurrerer om samme utbuffer. Svitsjene opererer feilfritt i den forstand at alle pakker videresendes i henhold til gjeldende rutealgoritme uten at feil oppstår i pakkene.

Vi har implementert støtte for fem forskjellige distribuerte, deterministiske rutealgoritmer for irregulære topologier. Hvilken av algoritmene som benyttes vil bli angitt for hvert enkelt simuleringsoppsett. For de rutealgoritmene som er utsatt for vranglås, altså de to varientene av korteste-vei-

ruting, oppdager og oppløser vi vranglås som forklart i kapittel 3.3.3. Det er vesentlig å merke seg at vår tilnærming til vranglås forutsetter en kunstig, allvitende komponent i nettverket, en komponent som til enhver tid kjenner hele nettverkets tilstand. Komponenten vil umiddelbart oppdage og løse opp enhver vranglås i nettverket. En slik presisjon og hurtighet er det umulig å oppnå i et reelt nettverk, noe det er viktig å ta hensyn til når man vurderer de ulike rutealgoritmene opp mot hverandre.

Når en vranglås først er oppdaget har vi implementert to forskjellige tilnærminger for å løse den opp. Den ene tilnærmingen løser opp en vranglås ved å kaste første pakke i kun ett av de innbufferne som direkte deltar i vranglåsen. Den andre tilnærminger går mer drastisk til verks og kaster første pakke i hvert innbuffer som direkte deltar i vranglåsen. Hvilken av de to metodene som benyttes vil bli oppgitt for de ulike simuleringsoppsettene.

Linkene

Alle linkene vi benytter har identiske egenskaper. De er toveis, feilfrie, har samme lengde og samme båndbredde. Tiden det tar å forflytte en pakke over en link er med andre ord den samme for alle linker.

Helt konkret benytter vi oss av ca 3 meter lange linker med en hastighet på 10Gb/s. Vi skal imidlertid ikke ha fokus på den nøyaktige hastigheten til våre linker (og svitsjer), men snarere konsentrere oss om hvordan de forskjellige nettverkskonfigurasjonene vi simulerer yter i forhold til hverandre. Internt i en diskret hendelsesdrevet simulator beskjeftiger man seg uansett ikke direkte med den reelle hastigheten til en link (eller en svitsj), men fokuserer snarere på hvor lang tid ulike hendelser tar i forhold til hverandre.

Når det gjelder flytkontroll over linkene vil vi, om ikke annet er oppgitt, benytte oss av på/av-flytkontroll ved alle våre simuleringer. Flytkontrollen arbeider på pakkenivå. Dette medfører at en avsender av trafikk alltid vil fullføre sending av en pakke selv om noden skulle få beskjed om at flytkontrollen skal aktiveres mens sendingen pågår. Vi har antatt at flytkontrollpakkene er så små at de kan ignoreres i forhold til resten av trafikken i nettverket, eventuelt at flytkontroll-informasjonen kan haike med annen trafikk i perioder hvor den motsatt rettede linken er utsatt for stor belastning.

For de simuleringer som er gjennomført for å besvare våre to første problemstillinger er triggerverdiene for flytkontroll satt slik at innbufferet til enhver tid har størst mulig fyllingsgrad. Med andre ord skrur man ikke på flytkontroll over en link før korresponderende innbuffer er i ferd med å gå fullt. Samtidig vil man skru flytkontrollen av igjen så fort det er ledig plass i innbufferet. For vår tredje problemstilling vil vi komme tilbake til hvilke triggerverdier som benyttes. Her skal vi jo nettopp se på hvilken effekt ulike triggerverdier har på trafikken i nettverket.

	Svitsjer	Porter	Linker	Endenoder
S8E1:	8	6	16	8
S16E1:	16	8	32	16
S16E4:	16	8	32	64
S32E1:	32	12	64	32
S32E4:	32	12	64	128

Tabell 4.1: Tabellen angir antall svitsjer, endenoder og linker i de topologier vi har benyttet for våre simuleringer. Antall porter er angitt per svitsj, og antall linker representerer kun de linker som kobler sammen svitsjer.

Topologi-størrelser

For våre simuleringsoppsett har vi valgt å ta utgangspunkt i topologier med fem ulike størrelser. Størrelsene er gjengitt i tabell 4.1. En rekke i tabellen definerer størrelsen til en topologi ved å angi antall svitsjer, antall porter per svitsj, antall linker mellom svitsjene og antall endenoder i nettverket. Selv om antall porter per svitsj ikke direkte sier noe om topologiens størrelse, vil antallet ha innvirkning på hvordan svitsjene kan kobles sammen og dermed også spille en rolle for topologiens utforming. Antallet endenoder er fordelt likt blant svitsjene i nettverket. I tillegg til antallet linker gitt i tabell 4.1 finner vi også en link mellom hver endenode og svitsjen den er koblet til. Portene hos svitsjene er toveis, men merk at en svitsj ikke nødvendigvis har alle portene sine i bruk. Det minste antallet porter i bruk hos en svitsj tilsvarer én port per tilkoblede endenode, samt minst én port med en link som knytter svitsjen til resten av nettverket. Nøyaktig hvilke topologi-størrelser som benyttes vil bli angitt for våre ulike simuleringsoppsett.

Vi har navngitt de fem topologi-størrelsene i tabell 4.1 for enkelt å kunne referere til dem når vi senere skal presentere og analysere våre resultater. Navnene er gitt på formen $SmEn$ hvor m og n angir henholdsvis antall svitsjer og antall endenoder per svitsj i nettverket. Merk at antall porter per svitsj og antall linker mellom svitsjene i nettverket ved tabell 4.1 er entydig gitt ut fra antall svitsjer.

4.2 Måling av ytelse

Før vi kan si noe om hvordan to nettverkskonfigurasjoner yter i forhold til hverandre må vi bestemme oss for nøyaktig hvordan vi velger å måle et nettverks ytelse. I et tett koblet nettverk ønsker vi at så mange pakker som mulig skal komme frem med minst mulig forsinkelse i løpet av en gitt tid, uten at pakker kastes i nettverket. Ved bruk av flytkontroll ved våre simuleringer vil pakker kun kunne kastes i nettverk som benytter seg av vranglås-utsatte

rutealgoritmer. Med dette som utgangspunkt har vi valgt følgende to størrelser som mål på ytelse: *gjennomstrømning* og *gjennomsnittsforsinkelse*.

Et nettverks gjennomstrømning definerer vi som summen av det totale antall pakker som mottas av alle endenoder i nettverket i løpet av en tid t. Gjennomsnittsforsinkelse beregner vi så ut fra gjennomsnittlig forsinkelse opplevd av alle de pakker som til sammen utgjør gjennomstrømningen. Forsinkelsen til en enkelt pakke settes lik tiden det tar fra pakken legges i en endenodes utbuffer til pakken mottas av den endenoden pakken er tiltenkt.

Både gjennomstrømning og gjennomsnittsforsinkelse vil altså avhenge av tiden t. t er det hensiktsmessig å styre ved hjelp av simulatorens interne klokke for å sikre at ulike nettverkskonfigurasjoner simuleres over det samme tidsintervall. Som tidligere nevnt er vi kun interessert i å se på et nettverks ytelse mens det er i stabil drift. I det man starter å sende trafikk ut i et nettverk vil de første pakkene som sendes ut oppleve et tilnærmet tomt nettverk med en tilgang på ressurser som ikke nødvendigvis er representativt for et nettverk i normal drift. Tidsintervallet t starter derfor først etter at vi ved en tid t_0 antar at nettverket har stabilisert seg.

Når vi skal studere og sammenlikne gjennomstrømning for ulike nettverkskonfigurasjoner, er det spesielt interessant å finne de forskjellige konfigurasjonenes *frafallspunkt*. Et nettverks frafallspunkt definerer vi som maksimalt samlet påtrykk fra endenodene som er slik at nettverket fortsatt, over tid, klarer å hente unna og levere til rett destinasjon alle de pakker som ønskes sendt. Pakker en endenode ønsker å sende vil med andre ord over tid ikke køes opp hos avsender på grunn av manglende kapasitet i nettverket før etter at påtrykket har passert frafallspunktet. Frem til vi når dette punktet kan man forvente at gjennomstrømningen stiger i takt med påtrykket siden man i løpet av en gitt tid vil sende tilnærmet like mange pakker inn i nettverket som man henter ut. Jo høyere frafallspunktet ligger, desto mer trafikk klarer nettverket med andre ord å håndtere. I det vi øker påtrykket over frafallspunktet vil nettverket etter hvert gå i metning. Dette har implikasjoner på gjennomstrømningen; gjennomstrømningen vil ikke lengre fortsetter å stige i takt med påtrykket. I verste fall vil faktisk et nettverk bli så tungt belastet etter hvert som påtrykket øker at gjennomstrømningen begynner å minke igjen. Vi skal se eksempler på dette i det vi går igjennom våre simuleringsresultater.

Frafallspunktet til en nettverkskonfigurasjon kan vi finne ved å gjenta simuleringer med stadig økt påtrykk helt til nettverket over tid ikke lengre klarer å håndtere det ønskede antall pakker fra endenodene. Om vi plotter gjennomstrømning fra de forskjellige simuleringene som en funksjon av benyttet samlet påtrykk fra alle endenoder, vil gjennomstrømningskurven ved frafallspunktet bøye av fra en tenkt rett linje som representerer det optimale tilfellet hvor nettverket alltid klarer å ta hånd om alle ønskede pakker. Dersom vi plotter gjennomstrømningskurver for ulike nettverkskonfigurasjoner i samme figur, vil vi kunne se hvilke nettverkskonfigurasjoner som tåler

størst påtrykk uten å gå i metning, nemlig de konfigurasjoner med høyest frafallspunkt.

Når det gjelder gjennomsnittsforsinkele ønsker vi oss at denne skal være så lav og stabil som mulig. Pakker som sendes igjennom et nettverk har imidlertid ulik lengde på sin reisevei, så det er vanskelig å på forhånd skulle avgjøre hvor lav gjennomsnittsforsinkelse man kan forvente. Ved å plotte gjennomsnittsforsinkelsen fra forskjellige nettverkskonfigurasjoner med varierende påtrykk i samme figur kan vi likevel si noe om hvordan de ulike konfigurasjonene yter i forhold til hverandre når det gjelder forventet forsinkelse.

4.2.1 Estimering av gjennomstrømning og gjennomsnittsforsinkelse

Topologi-størrelsene i tabell 4.1 representerer ulike sett med topologier. For å kunne si noe om gjennomstrømning og gjennomsnittsforsinkelse for nettverk innenfor de forskjellige settene har vi valgt å plukke ut 16 tilfeldig topologier fra hver topologi-størrelse. De 16 topologiene er tilfeldig generert av vår simulator, men alltid de samme for ulike nettverkskonfigurasjoner vi ønsker å sammenlikne. For å kunne gjennomføre en rettferdig sammenlikning av for eksempel ulike rutealgoritmer velger vi å teste de forskjellige algoritmene over identiske topologier.

Ved en simulering av en konkret nettverkskonfigurasjon over 16 tilfeldige topologier fra samme topologi-størrelse, beregnes først gjennomstrømning og gjennomsnittsforsinkelse for hver enkelt topologi. Med utgangspunkt i disse verdiene beregnes så gjennomsnittlig gjennomstrømning g og gjennomsnittlig gjennomsnittsforsinkelse f for de 16 tilfeldig valgte topologiene. g og f vil da fremstå som estimater av henholdsvis gjennomsnittlig gjennomstrømning og gjennomsnittlig gjennomsnittsforsinkelse for et sett tilfeldig valgte topologier innen samme topologi-størrelse.

Ved hjelp av g og f søker vi å danne oss et bilde av hvordan man kan forvente at ulike nettverkskonfigurasjoner yter. Siden g og f er estimerte størrelser bør vi da samtidig vurdere usikkerheten til disse to estimatene. For å angi feilmarginene til våre estimerte verdier skal vi benytte oss av konfidensintervaller.

4.2.2 Konfidensintervaller

Et X%-konfidensintervall[13] er et statistisk begrep som refererer til det å beregne et intervall, relatert til en estimert størrelse ϕ, hvor X% av intervallene som fremkommer ved gjentatte, uavhengige estimeringer av ϕ vil inneholde den sanne verdien man søker å estimere. Merk at den sanne verdien man søker å estimere gjerne har en eksakt verdi. Det gir derfor ikke mening å si at et konkret X%-konfidensintervall inneholder den sanne ver-

dien med $X\%$ sannsynlighet. Likevel gir størrelsen på et konfidensintervall et bilde av den feilmarginen som er relatert til en estimering, og ikke minst et inntrykk av variasjonen blant de individuelle verdiene vi benytter for å beregne g og f. Jo større varians blant de individuelle verdiene, desto lengre vil konfidensintervallene være.

For å beregne konfidensintervaller for våre estimerte verdier g og f må vi gjøre enkelte antagelser. Det er vanlig å anta at simuleringsresultater over nettverkskonfigurasjoner hvor kun topologien forandres (men størrelsen holdes konstant) er poisson-fordelt. En poisson-fordeling med høy forventing er tilnærmet normalfordelt[13]. Vi antar derfor at de simuleringsresultater vi benytter for å estimere g og f stammer fra en distribusjon som er tilnærmet normalfordelt. Da hver estimering vi gjennomfører tar utgangspunkt i kun 16 verdier kan vi imidlertid ikke benytte oss direkte av at distribusjonen er normalfordelt. Vi må gå veien om t-fordeling[13]. t-fordeling gir lengre konfidensintervaller enn hva man ville fått ved å bruke normalfordeling direkte, men er samtidig mer robust i forhold til at estimatene baserer seg på få observasjoner. For alle våre estimerte g- og f-verdier har vi beregnet 95%-konfidensintervaller basert på en t-fordeling med 15 frihetsgrader. Antall frihetsgrader ved bruk av t-fordeling tilsvarer (antall observasjoner) $-$ 1.

4.3 Sammenlikning av ulike rutealgoritmer

Vår første problemstilling lyder som følger:

> P1: *Hvilken ytelsesforbedring kan vi forvente fra Ethernet som tett koblet nettverk om vi bytter ut Ethernets tradisjonelle rutetabeller med rutetabeller bygget opp ved hjelp av up*/down*-ruting?*

For å besvare denne problemstillingen har vi gjennomført et forholdsvis omfattende sett av simuleringer. Vi har simulert over de fem topologistørrelsene gjengitt i tabell 4.1. Innen hver størrelse har vi benyttet 16 tilfeldig valgte topologier for å kunne estimere gjennomstrømning og gjennomsnittsforsinkelse som forklart i kapittel 4.2.1. For hver topologi har vi så simulert med rutetabeller bygget opp ved hjelp av alle de ulike rutealgoritmene gjengitt i tabell 4.2. Kolonnen merket "Vranglåsoppløsing" angir for vranglåsutsatte rutealgoritmer hvorvidt én eller flere pakker kastes for å løse opp en eventuell vranglås. Akronymene i tabellen er de samme som vi vil benytte når vi markerer forskjellige kurver i våre resultatplott.

Hver kombinasjon av rutealgoritme og topologi er gjentatt simulert med tre ulike organiseringer av kantbufferne hos svitsjene: Svitsjer som er innbufret, svitsjer som er utbufret og svitsjer som er like-bufret. En svitsj som er innbufret har hos oss plass til 20 pakker i hvert innbuffer og 4 pakker i hvert utbuffer. En svitsj som er utbufret har plass til 4 pakker i hvert innbuffer og 20 pakker i hvert utbuffer. I en like-bufret svitsj er inn- og utbufferne

Akronym	Rutealgoritme	Vranglåsoppløsing
ETH	Tradisjonelt Ethernet	-
DUD	Deterministisk up*/down*	-
DKVUD	Deterministisk korteste-vei-up*/down*	-
SKV	Simpel korteste-vei	singel
VKV	Vektet korteste-vei	singel
SKVm	Simpel korteste-vei	multippel
VKVm	Vektet korteste-vei	multippel

Tabell 4.2: Tabellen gir oversikt over de ulike rutealgoritmer benyttet ved simulering.

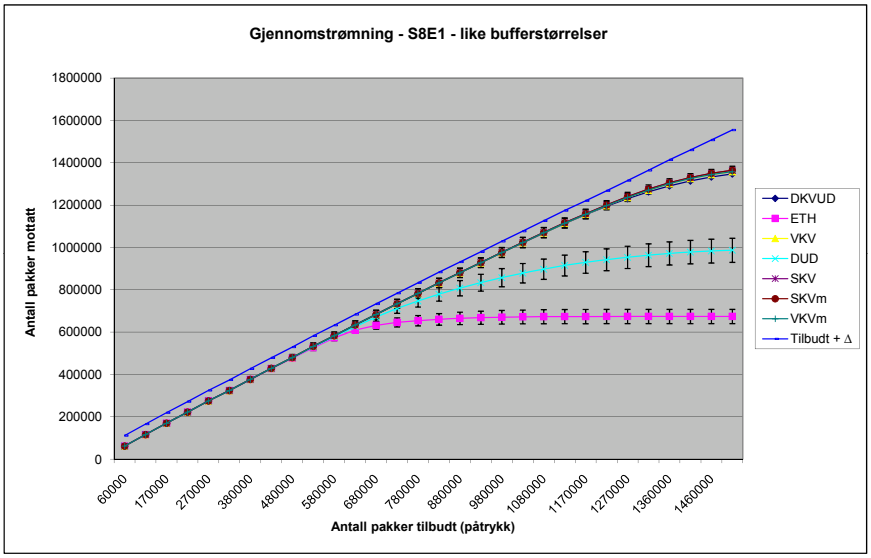
like store, med plass til 12 pakker i hvert buffer. Den totale bufferplassen er altså den samme for de ulike typene svitsjer.

Det er to årsaker til at vi velger å simulere over ulik organisering av buffer i svitsjene. For det første gir dette oss innsikt i om forholdet mellom ytelsene til de forskjellige rutealgoritmene kan se ut til å variere avhengig av svitsjenes bufferorganisering. Samtidig gir simuleringene oss resultater vi kan benytte for å besvare vår andre problemstilling, en problemstilling som nettopp stiller spørsmål ved hvorvidt ulike måter å organisere kantbuffer på i svitsjene vil ha innvirkning på ytelsen til et tett koblet nettverk. Problemstilling nummer to skal vi komme tilbake til i neste delkapittel.

Kombinasjonen av forskjellige topologier, rutealgoritmer og svitsjer med ulik bufferorganisering, samt et ønske om å simulere over ulike grader av påtrykk, har ført til av vi for å besvare vår første (og andre) problemstilling har gjennomført i overkant av 50.000 simuleringer. Selv om vi er ute etter gjennomsnittsverdier og vil presentere mange av resultatene i samme plott, er det verken plass eller særlig hensiktsmessig å gjengi alle resultatene i dette delkapittelet. Vi vil derfor i påfølgende avsnitt begrense oss til å presentere og analsyere et representativt utvalg av plottene fra de mange simuleringene. En mer fullstendig oversikt over plott av relevante resultater knyttet til våre to første problemstillinger finner man i tillegg A.1.

4.3.1 Gjennomstrømning

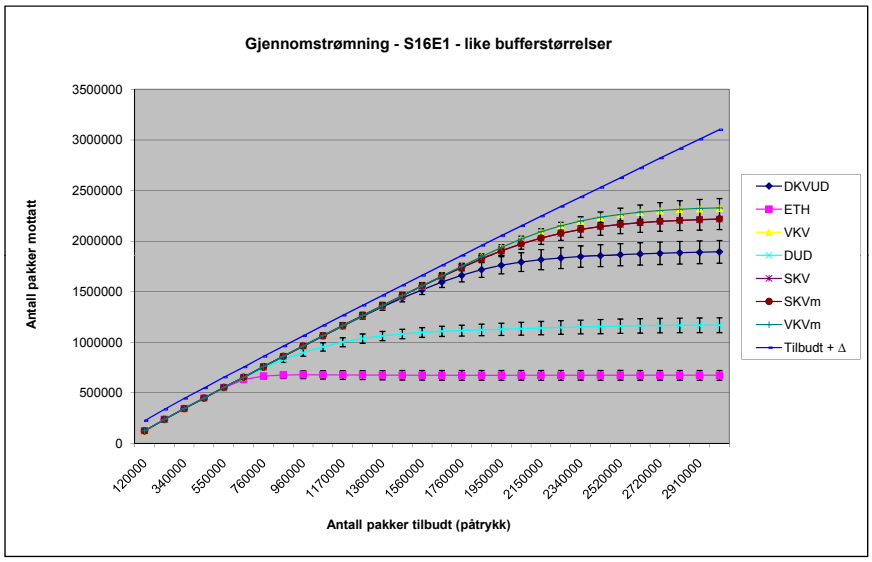
Figurene 4.1 til 4.6 viser plott av gjennomstrømning for våre rutealgoritmer over ulike topologistørrelser. Den horisontale aksen angir det totale antall pakker tilbudt av alle endenoder i nettverket over den samme tidsperiode som man har benyttet for å finne gjennomstrømning. Resultatene i figurene stammer alle fra simuleringer hvor svitsjene benytter inn- og utbuffer som er like store. For å tydeliggjøre hvor frafallspunktet ligger for de forskjellige rutealgoritmene har vi tegnet inn ideal-linjen "Tilbudt + Δ" i hver plott.



Figur 4.1: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologistørrelse S8E1. Svitsjene benytter inn- og utbuffer som er like store.

Linjen representerer drømme-situsjonen hvor nettverket alltid klarer å håndtere alle pakker ønsket sendt, uansett antall. Vi har lagt til en liten verdi Δ, slik at linjen ikke skal sammenfalle med de ulike rutealgoritmene før de når frafallspunktet. Konfidensintervaller er markert som vertikale stolper. Merk at enkelte konfidensintervaller er utelatt for å øke lesbarheten til plottene, og at to av plottene av samme årsak er gjengitt to ganger med forskjellige konfidensintervaller tegnet inn. Man vil finne flere konfidensintervaller i plott gitt i tillegg A.1.

For de relativt små topologiene innen GS8E1 (figur 4.1) ser vi at nesten alle rutealgoritmene gir tilnærmet samme frafallspunkt og gjennomstrømning. Konfidensintervallene er også små, noe som indikerer liten variasjon blant de simuleringer vi har basert oss på for å beregne gjennomsnittsverdiene presentert i figuren. De to rutealgortimene som skiller seg ut i negativ retning er deterministisk up*/down* (DUD) og tradisjonelt Ethernet (ETH). Det er rimelig å anta at ETHs lave frafallspunkt skyldes at en del linker i nettverket ikke benyttes. For DUD sin del husker vi at algoritmen ikke alltid er like flink til å finne en kort vei gjennom nettverket, men av og til angir ruter som benytter en lite optimal opp-link. Selv om DUD har et lavt frafallspunkt ser vi samtidig at denne rutealgoritemen fører til at nettverket går saktere i metning etter at frafallspunktet er passert enn hva som er tilfellet ved bruk av ETH. Deterministisk korteste-vei-up*/down* (DKVUD) har en gjennomstrømning på linje med de ulike korteste-vei-rutealgoritmene.

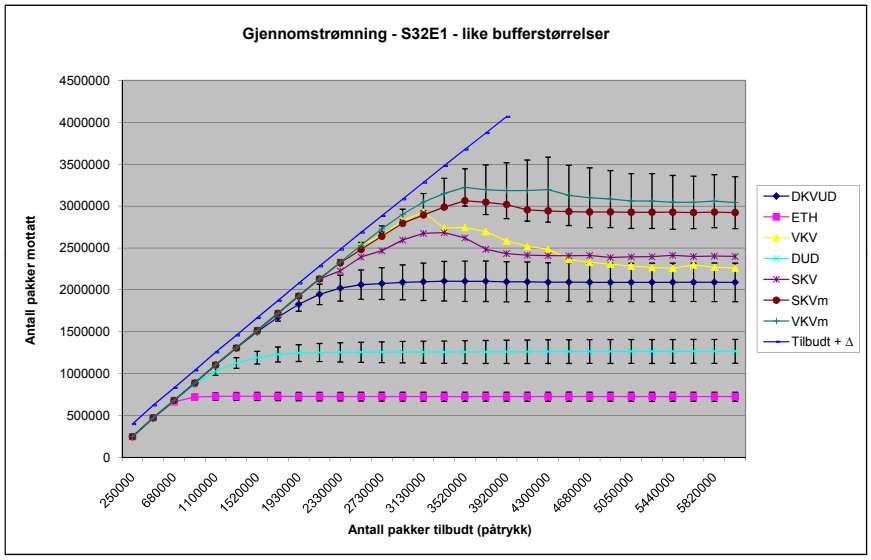


Figur 4.2: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi-størrelse S16E1. Svitsjene benytter inn- og utbuffer som er like store.

Om vi beveger oss over til topologi-størrelsen S16E1 (figur 4.2) ser vi at ETH og DUD fortsatt faller fra forholdsvis tidlig. Til forskjell fra størrelsen S8E1 får imidlertid også DKVUD nå problemer med å følge de ulike korteste-vei-rutealgoritmene i det påtrykket blir stort. Korteste-vei-algoritmene følger hverandre på sin side fortsatt tett. Om vi tar konfidensintervallene med i vurderingen, ser vi imidlertid at DKVUD ikke nødvendigvis stabiliserer seg så mye lavere enn de forskjellige korteste-vei-algoritmene[1].

At alle korteste-vei-rutealgoritmene våre yter så likt for S8E1 og S16E1 skyldes felles egenskaper hos disse forholdsvis små topologi-settene. For det første sammenfaller ofte vektede og ikke-vektede ruter. Samtidig ser det ut til at man får lite igjen for å vekte linkene de gangene vekten faktisk gir opphav til en alternativ korteste-vei-rute. I figur 4.2 ser vi hvordan de to vektede implementasjonene VKV og VKVm kun opplever en marginalt bedre gjennomstrømning enn sine to ikke-vektede alternativer SKV og SKVm. Tar man konfidensintervallene med i betraktningen er det vanskelig å argumentere for eksistensen av en vesentlig forskjell. De to tilnærmingene til å løse opp vranglås spiller ingen vesentlig rolle for topologier innen S8E1

[1]Ut fra plottet gjengitt i figur 4.2 alene skal man være forsiktig med å hevde at de ulike rutealgoritmene stabiliserer seg rundt de verdier som er vist for høyeste påtrykk i figuren. For de forholdsvis små topologi-størrelsene representert ved S8E1 og S16E1 har vi imidlertid ikke klart å spore noe tegn til at de ulike rutealgoritmene gir lavere gjennomstrømning i det påtrykket øker ytterligere, selv ikke ved simulering av maksimalt påtrykk: kontinuerlig trafikk fra alle endenoder.

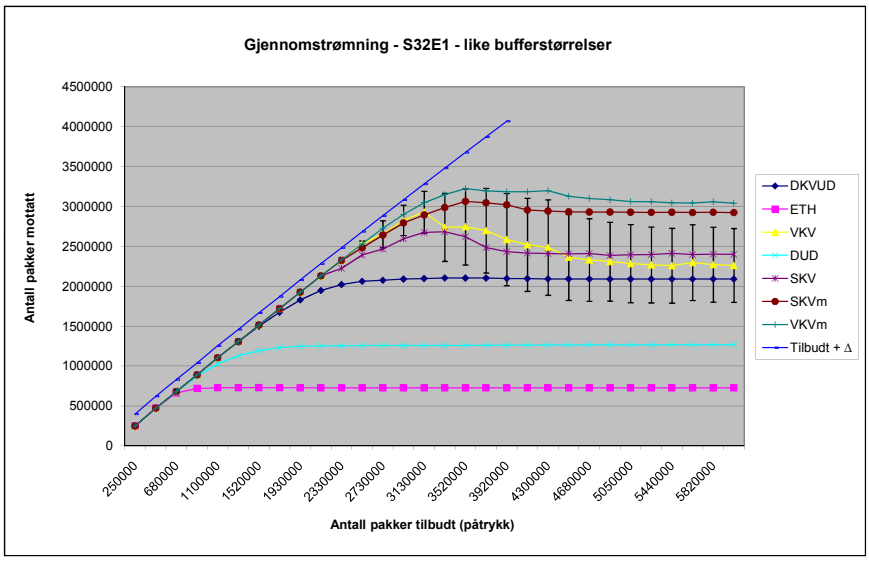


Figur 4.3: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi-størrelse S32E1. Svitsjene benytter inn- og utbuffer som er like store. Plott 1 av 2 for å vise konfidensintervaller for ulike rutealgoritmer.

---

og S16E1, siden disse topologi-størrelsene faktisk utgjør nettverk som er så små og enkle at vranglås sjelden oppstår. Vranglåshåndtering spiller først en større rolle når vi beveger oss opp til neste topologi-størrelse, S32E1.

Figurene 4.3 og 4.4 viser to plott fra det samme settet av simuleringer, denne gangen med topologi-størrelse satt til S32E1. Vi har valgt å benytte to plott for å øke lesbarheten til konfidensintervallene. I tillegg er konfidensintervallene for SKV og SKVm utelatt. Vi ser tydelig i figur 4.3 at trenden fra de mindre topologiene fortsetter for ETH og DUD; de faller begge fra og stabiliserer seg forholdsvis tidlig i forhold til de andre rutealgoritmene. DKVUD holder på sin side fortsatt stand ved vesentlig høyere påtrykk enn ETH og DUD, til tross for at vi nå for DKVUD kan spore en noe større variasjon gjennom lengre konfidensintervaller enn tidligere. Samtidig ser det ut til at DKVUDs frafallspunkt har blitt liggende ytterligere bak korteste-vei-rutealgoritmene. Videre ser vi nå for første gang antydning til hvordan forskjellige måter å håndtere vranglås på kan spille en rolle i nettverket. S32E1 representerer topologier hvor vranglås blir et problem i det påtrykket øker. SKVm og VKVm, som begge kaster flere pakker når en vranglås først oppstår, klarer å holde gjennomstrømningen oppe til tross for høyt påtrykk. SKV og VKV, som kun kaster én pakke per vranglås, ser ikke ut til å takle et høyt påtrykk like godt. Spesielt for VKV ser vi at gjennomstrømningen etter hvert avtar i det påtrykket blir stort. Alle de fire korteste-vei-rutealgoritmene



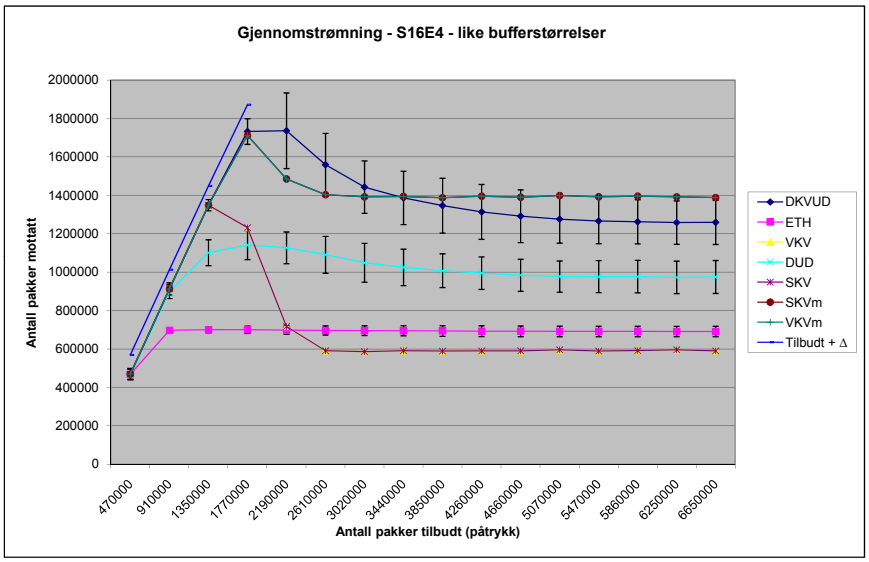
Figur 4.4: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologistørrelse S32E1. Svitsjene benytter inn- og utbuffer som er like store. Plott 2 av 2 for å vise konfidensintervaller for ulike rutealgoritmer.

---

har imidlertid konfidensintervaller som er forholdsvis lange. Dette gjelder ikke minst konfidensintervallene som er tegnet inn for VKV i figur 4.4 (Plott som viser konfidensintervaller for øvrige rutealgoritmer finnes i vedlegg A.1). Det er med andre ord tydelig at gjennomstrømning ved bruk av korteste-vei-ruting for nettverk med en størrelse som tilsvarer S32E1 vil avhenge av den konkrete topologien som benyttes.

De siste gjennomstrømnings-plottene vi skal vurdere for å sammenlikne våre ulike rutealgoritmer er gjengitt i figurene 4.5 og 4.6. Benyttet topologistørrelse er nå satt til S16E4. Vi har dermed gått ned på antallet svitsjer i forhold til S32E1, men knytter nå fire endenoder til hver svitsj i stedet for en. Dermed kan hver enkelt svitsj i nettverket, ved sine fire endenoder, ikke bare generere fire ganger så mye trafikk som tidligere, men også i parallell tilby fire nye pakker til nettverket. Vi skal se at dette får konsekvenser for gjennomstrømningen.

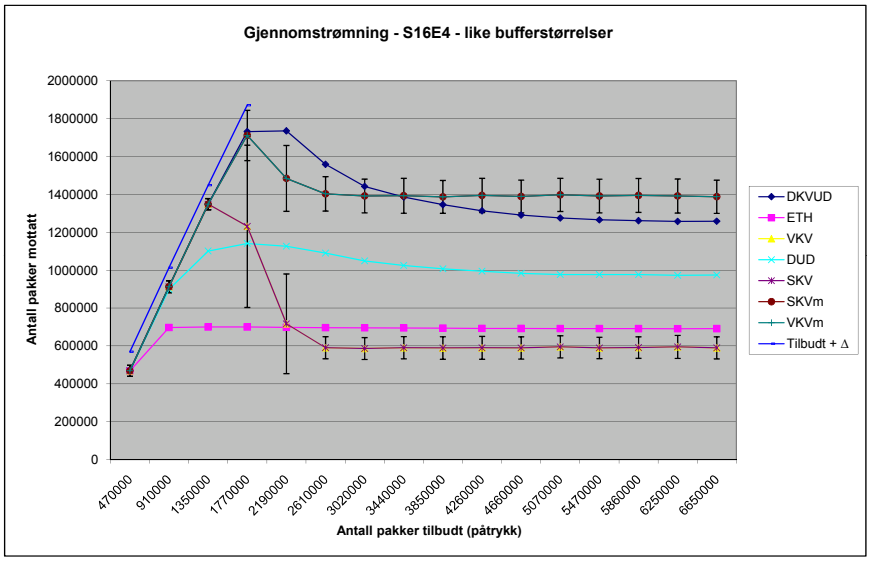
Forholdet mellom ETH, DUD og DKVUD viser i figur 4.5 de samme tendenser som tidligere, ETH faller dårligst ut, mens DKVUD takler størst påtrykk av de tre. Samtidig ser vi at kortest-vei-rutealgoritmene faller fra tidligere enn ved våre hittil omtalte simuleringer. Vi skal komme tilbake til hvorfor dette skjer, men først se på en ny generell tendens i dette plottet: For alle rutealgoritmene ser vi at gjennomstrømningen ser ut til å synke for en periode etter at man har passert frafallspunktet. Dette gjelder også ETH



Figur 4.5: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologistørrelse S16E4. Svitsjene benytter inn- og utbuffer som er like store. Plott 1 av 2 for å vise konfidensintervaller for ulike rutealgoritmer.

---

selv om nedgangen her er så liten i forhold til de andre rutealgoritmene at det ikke kommer tydelig frem i plottet. Riktig nok viser figur 4.6 svært store konfidensintervaller for korteste-vei-rutealgoritmene i områdene hvor kurvene synker, men merk at intervallene raskt blir kortere igjen i det kurvene stabiliserer seg ved et lavere nivå. Vi kunne kanskje ønske oss en høyere oppløsning for kurvene rundt de ulike frafallspunktene, men tendensen er uansett klar: Det er tydelig at det nye trafikkmønsteret i nettverket, representert ved fire endenoder per svitsj, fører til at et påtrykk over frafallspunktet i større grad enn tidligere har en negativ effekt på gjennomstrømningen. Helt konkret kan vi sammenlikne kurven for DKVUD i figur 4.2 med samme kurve i figur 4.5. De to kurvene representerer gjennomstrømning for det samme utvalg av topologier hvor eneste forskjell er antall endenoder per svitsj. Ved et totalt påtrykk på to millioner pakker indikerer de to plottene at man kan forvente en høyere gjennomstrømning for de nettverk hvor kun én endenode er koblet til hver svitsj. Den samme tendensen finner vi hos alle korteste-vei-rutealgoritmene også, og hos ETH og DUD om enn i noe mindre grad. Det er tydelig at overgangen til fire endenoder per svitsj fører til en annen og mere kritisk form for metning i nettverk av den størrelse vi her har sett på. Dette må skyldes at hver svitsj i nettverket nå parallelt kan tilbyr fire nye pakker til nettverket. Det kan se ut til at en slik form for påtrykk raskere overbelaster svitsjene, og dermed fører til en total nedgang i gjennomstrømningen. Øker

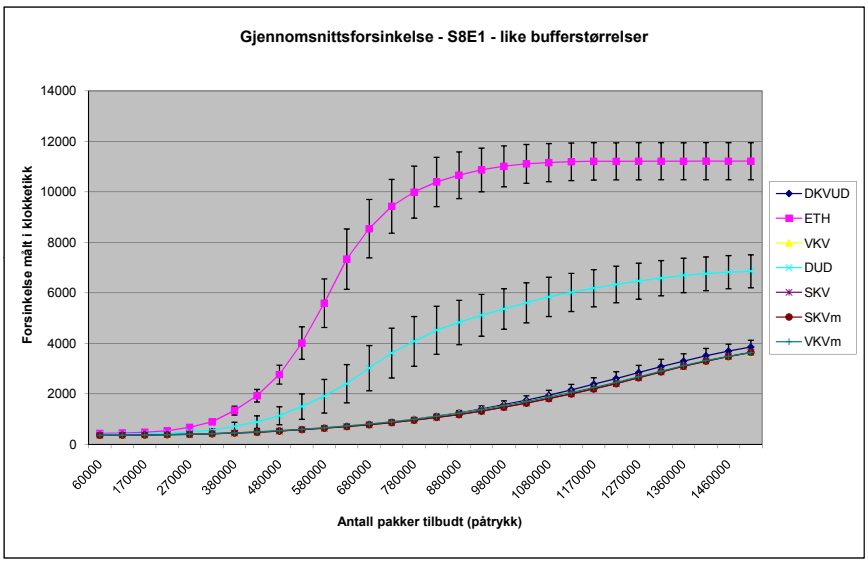


Figur 4.6: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologistørrelse S16E4. Svitsjene benytter inn- og utbuffer som er like store. Plott 2 av 2 for å vise konfidensintervaller for ulike rutealgoritmer.

---

vi størrelsen på topologien fra S16E4 til S32E4 vil vi se at denne nye tendensen tiltar i styrke: Nedgangen i gjennomstrømning er enda mer dramatisk i det man passerer frafallspunktet. Dette gjelder spesielt de rutealgoritmene som i utgangspunktet har høyest frafallspunkt. Plott for simuleringer over S32E4 finnes i tillegg A.1.

La oss nå vende tilbake til figur 4.6 og korteste-vei-rutealgoritmene. Det er tydelig at vi fortsatt får lite igjen for å vekte linkene i en topologi med kun 16 svitsjer. Kurvene til SKV og VKV er tilnærmet identiske. Det samme gjelder kurvene for SKVm og VKVm. Men hvorfor faller korteste-vei-algoritmene fra tidligere enn før? Svaret er knyttet til vranglås-problemet. I det vi øker antall endenoder per svitsj, øker vi også faren for vranglås betydelig, selv for en forholdsvis liten topologi som den vi ser på her. Dette går spesielt hardt ut over SKV og VKV som kun kaster én pakke i det vranglås oppstår. Vi ser at disse to implementasjonene etter hvert går fullstendig i stampe og stabiliserer seg på et nivå som faktisk ligger under ETH. Vranglås er også et problem for SKVm og VKVm, men det kan se ut til at det å kaste en pakke per svitsj for å løse opp vranglås er nok til at disse to implementasjonene fortsatt oppnår en god gjennomstrømning. Det er imidlertid viktig å huske på at vi her oppdager vranglås raskere enn hva man kan forvente i et reelt nettverk! Slik sett ligger kurvene til både SKVm, VKVm, SKV og VKV kunstig høyt.

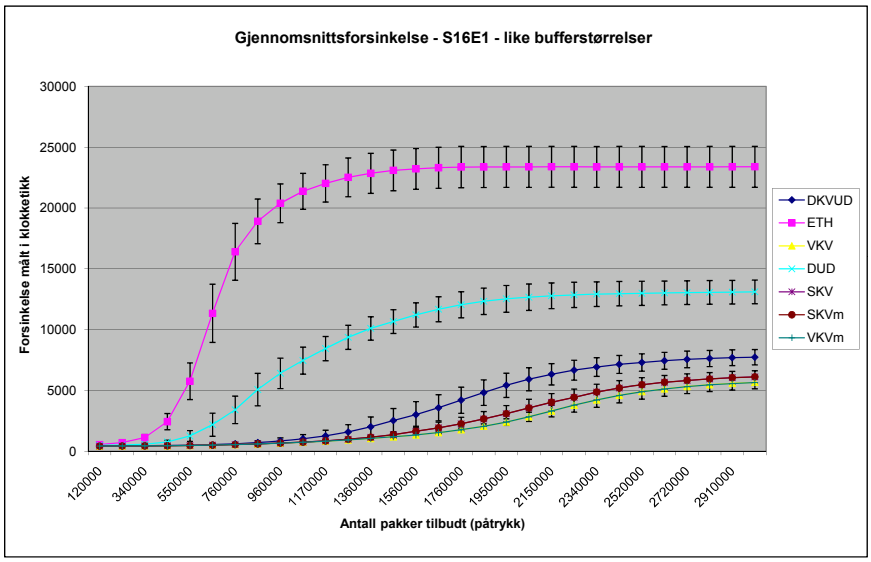


Figur 4.7: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi-størrelse S8E1. Svitsjene benytter inn- og utbuffer som er like store.

4.3.2 Gjennomsnittsforsinkelse

Figurene 4.7 til 4.10 viser plott av gjennomsnittsforsinkelse for de simuleringer vi benyttet i forrige avsnitt for å vurdere gjennomstrømning. Vi har også denne gangen utelatt enkelte konfidensintervaller for å øke lesbarheten til plottene. De intervaller som er utelatt vil man finne igjen blant de plott som er gjengitt i tillegg A.1.

For alle plottene, uavhengig av topologi-størrelse og rutealgoritme, ser vi den samme tendensen. Plottene starter med en lav gjennomsnittsforsinkelse for hver enkelt rutealgoritme. I det man nærmer seg de ulike frafallspunktene ser vi imidlertid at kurvene begynner å stige. Gjennomsnittetforsinkelsen fortsetter å stige selv etter at vi har passerer frafallspunktet, helt til den etter en tid stabiliserer seg ved et nytt og høyere nivå. Denne variasjonen i gjennomsnittsforsinkelse etter hvert som påtrykket øker kan vi forklare ved å se nærmere på forsinkelsen til en enkelt pakke. Den totale forsinkelsen F en pakke opplever på sin vei gjennom nettverket består av to komponenter, T_1 og T_2. T_1 gjenspeiler tiden det tar for en pakke å forflytte seg over linker og gjennom svitsjer fra kilde til destinasjon i nettverket med fravær av konkurranse fra andre pakker. T_1 er med andre ord en konstant størrelse for en gitt rute. T_2 gjenspiler på sin side venting i svitsjene som skyldes konkurranse fra andre pakker. En pakke P_1 må vente i en svitsj dersom en annen pakke P_2 holder den neste ressursen P_1 ønsker å benytte. I det man nærmer seg

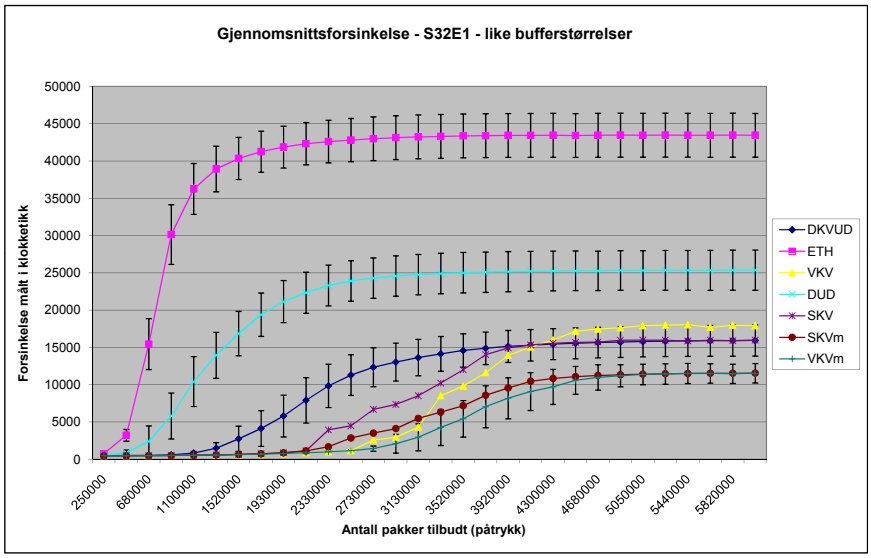


Figur 4.8: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi-størrelse S16E1. Svitsjene benytter inn- og utbuffer som er like store.

---

frafallspunktet begynner det å bli tett mellom pakkene i nettverket, og man kan forvente at ressurskonflikter mellom pakker oppstår oftere enn ved lavere påtrykk. Dermed vil T_2 vokse i takt med økt konkurranse mellom pakkene i nettverket. T_2 vil imidlertid ikke vokse ubegrenset. Langs enhver rute i nettverket er det et endelig antall ressurser. Om svitsjene forvalter ressursene rettferdig vil en pakke maksimalt vente en gitt tid for tilgang til hver enkelt ressurs den etterspør (forutsatt at nettverket ikke har gått i vranglås). T_2 vil dermed også være oppad begrenset for en gitt rute. Om vi nå vender tilbake til plottene vil den lave gjennomsnittsforsinkelsen hver kurve starter med reflektere et gjennomsnitt som er dominert av T_1-verdier og små T_2-verdier. I det påtrykket nærmer seg frafallspunktet tetner trafikken til og T_2-verdiene øker. Denne økningen fortsetter helt til de ulike T_2-verdiene er i ferd med å nå sine øvre grenser. I det grensene nås flater også kurven for gjennomsnittsforsinkelsen ut igjen.

Om vi nå tar for oss hvert enkelt av plottene i figurene 4.7 til 4.10 ser vi at det gjennomgående er slik at de rutealgoritmene som tidligere kunne vise til høyest frafallspunkt og størst gjennomstrømning også er de rutealgoritmene som gir lavest forsinkelse. Igjen kommer tradisjonelt Ethernet dårligst ut, mens deterministisk korteste-vei-up*/down*-ruting gjør det best av de to up*/down*-variantene. Korteste-vei-rutealgoritmene er som tidligere avhengig av at vranglås håndteres på en effektiv måte i det topologien og



Figur 4.9: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi-størrelse S32E1. Svitsjene benytter inn- og utbuffer som er like store.

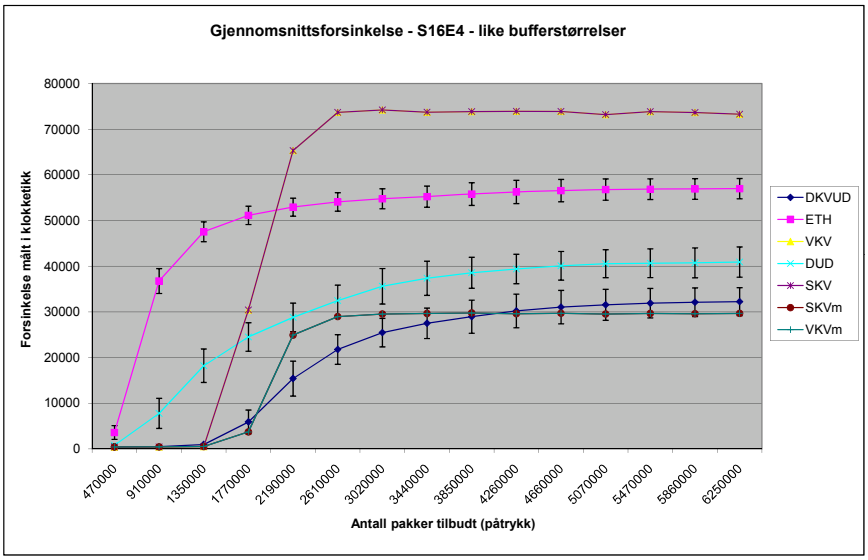
---

trafikkbildet blir mer komplisert.

4.3.3 Konklusjon

Alle de resultater vi har presentert tyder på at man kan oppnå en vesentlig ytelsesforbedring, både når det gjelder gjennomstrømning og gjennomsnittsforsinkelse, dersom man bytter ut rutetabellene i tradisjonelt Ethernet med rutetabeller bygget opp av en av våre varianter av up*/down*-ruting. Samtidig har vi sett at korteste-vei-ruting er helt avhengig av en god måte å håndtere vranglås på om man skal oppnå god ytelse i det topologi-kompleksiteten øker eller flere endenoder kobles til hver svitsj. I vår simulator har vi basert oss på en kunstig effektiv håndtering av vranglås. Siden det i et reelt nettverk er vanskelig å oppdage og oppløse vranglås på en presis og effektiv måte[21], er korteste-vei-ruting ikke et alternativ for oss. Av de rutealgoritmene vi har sett på fremstår deterministisk korteste-vei-up*/down*-ruting dermed totalt sett som det gunstigste alternativet om man ønsker å benytte Ethernet-teknologi for tett koblede nettverk.

Til slutt må vi huske at vi så langt kun har presentert resultater fra simuleringer hvor svitsjene har like store inn- og utbuffer. Om man studerer plottene i tillegg A.1 vil man imidlertid se at vi har fått tilsvarende resultater både for inn- og utbufrede svitsjer. Det ser altså ut til at vi kan forvente en vesentlig ytelsesforbedring ved å benytte oss av up*/down*-rutetabeller



Figur 4.10: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi-størrelse S16E4. Svitsjene benytter inn- og utbuffer som er like store.

---

i Ethernet, uavhengig av om svitsjene er innbufret, utbufret eller har like store inn- og utbuffer.

4.4 Sammenlikning av inn- og utbufrede svitsjer

Vår andre problemstilling lyder som følger:

> P2: *For én svitsj med kantbuffer er det, for å unngå køhodeblokkering, gunstig at kun en liten andel av svitsjens buffer benyttes som innbuffer. Kan vi si det samme om tilsvarende svitsjer i et tett koblet nettverk med flytkontroll, eller har ikke køhodeblokkering i et innbuffer her samme effekt i forhold til nettverkets totale ytelse?*

Som tidligere nevnt skal vi for å besvare denne problemstillingen benytte oss av de samme simuleringsresultater som dem vi benyttet for å belyse vår første problemstilling. For lettere å kunne vurdere en eventuell forskjell i ytelse i nettverk som benytter inn- og utbufrede svitsjer skal vi imidlertid endre litt på plottene. Den horisontale og den vertikale aksen skal fortsatt representere henholdsvis påtrykk og gjennomstrømning (eller gjennomsnittsforsinkelse), men hver enkelt figur vil nå inneholde simuleringsresultater for kun én kombinajon av rutealgoritme og topologi-størrelse. I samme figur plotter vi så

Type	Innbuffer	Utbuffer
Innbufret svitsj	20	4
Utbufret svitsj	4	20
Like-bufret svitsj	12	12

Tabell 4.3: Tabellen gir oversikt over konkrete bufferstørrelser for de inn-, ut- og like-bufrede svitsjer vi har benyttet ved simulering. Hver størrelse angir hvor mange hele pakker et buffer har plass til.

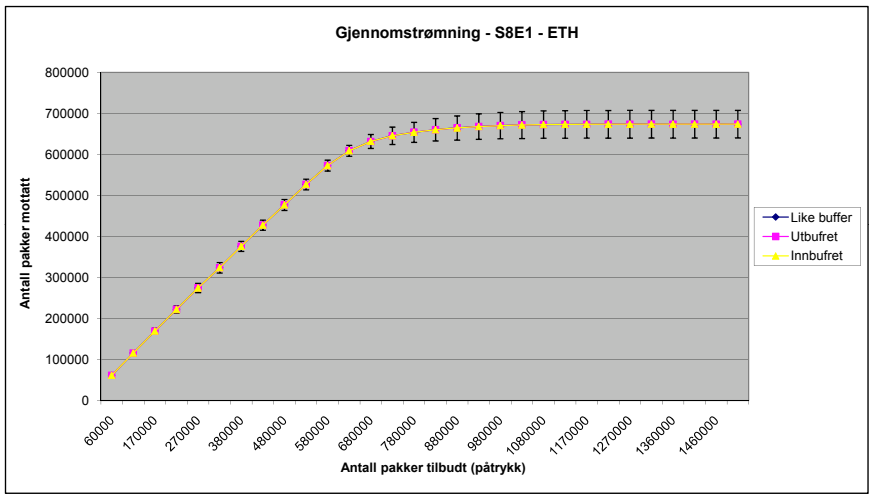
resultater fra to ulike simuleringssett som representerer nettverk hvor man har benyttet henholdsvis inn- og utbufrede svitsjer. I tillegg har vi valgt å plotte inn resultater fra simuleringer hvor vi har benyttet svitsjer med like store inn- og utbuffer. De tre kurvene bør, sammen med konfidensintervaller, kunne gi oss et bilde av om ytelsen kan sies å variere vesentlig avhengig av om man benytter inn- eller utbufrede svitsjer. De konkrete bufferstørrelsene vi har benyttet er gjengitt i tabell 4.3.

Produktet av antall rutealgoritmer og topologi-størrelser blir for stort til at vi kan presentere plott for alle kombinasjonene. Vi skal derfor begrense oss til et lite utvalg figurer som godt reflekterer de funn vi har gjort.

4.4.1 Gjennomstrømning

Figurene 4.11 til 4.14 viser gjennomstrømningsplott for inn-, ut- og like-bufrede svitsjer. Hver figur representerer en egen kombinasjon av rutealgoritme og topologi-størrelse. For alle plottene, med unntak av det i figur 4.11, har vi valgt å ikke ta med gjennomstrømning frem til man nærmer seg frafallspunktet. Dette valget er tatt for at vi skal kunne "forstørre" våre plott rundt de områder hvor påtrykket er relativt stort. Uten en slik forstørring vil det være vanskelig å se forskjell på kurver som representerer inn-, ut- og like-bufrede svitsjer, rett og slett fordi kurvene er nesten sammenfallende sett i forhold til hele det intervallet den vertikale aksen spenner over. Våre simuleringer viser samtidig at forskjellen mellom de ulike kurvene er ekstra liten frem til man nærmer seg frafallspunktet. Våre resultater tyder faktisk på at forskjellen i gjennomstrømning gjerne ligger på under en promille så lenge påtrykket i nettverket er lavt. I det man forstørrer plottene er det imidlertid viktig å huske på at aksene ikke lengre starter på null. Den relative forskjell mellom ulike kurver i plottene kan dermed virke større enn det den faktisk er.

Figur 4.11 viser tradisjonelt Ethernet benyttet i nettverk med topologier på størrelse med S8E1. Som vi ser er det i dette plottet umulig å spore særlig forskjell på de nettverk som har benyttet inn-, ut- og like-bufrede svitsjer. Konfidensintervallene er sammenfallende for alle tre kurvene, og indikerer at



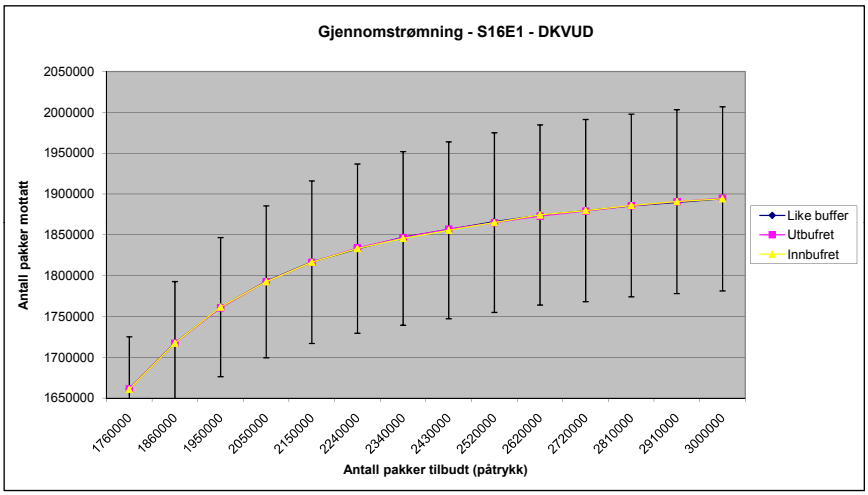
Figur 4.11: Gjennomstrømning ved bruk av inn-, ut- og like-bufrede svitsjer. Topologi-størrelse S8E1. Rutetabeller tilsvarer tradisjonelt Ethernet.

---

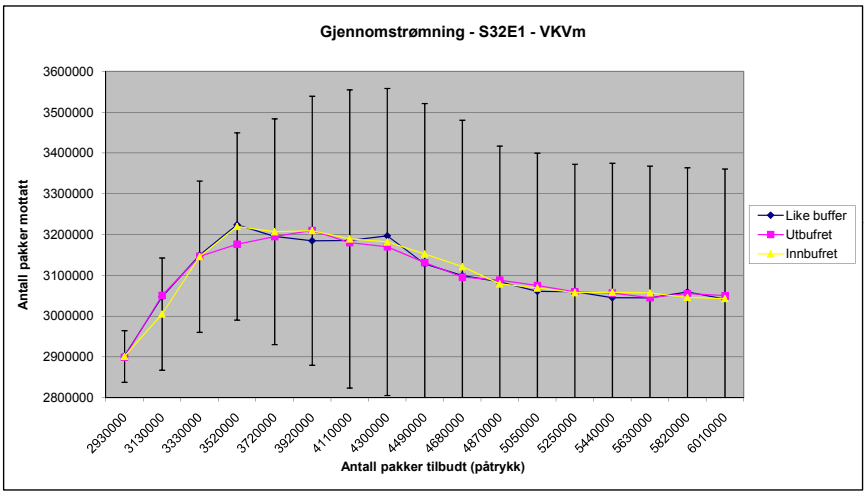
man antagelig vil finne minst like stor variasjon i gjennomstrømning på tvers av tilfeldig valgte topologier, som det man ser ut til å kunne finne i det man skifter mellom inn-, ut- og like-bufrede svitsjer for én konkret topologi.

I figur 4.12 har vi, som tidligere nevnt, valgt å kun vise et lite utsnitt av gjennomstrømningskurven. Denne gangen er deterministisk korteste-vei-up*/down*-ruting benyttet for topologier på størrelse med S16E1. Som vi ser er det fortsatt vanskelig å skille de tre kurvene. Konfidensintervallene er de samme for alle tre kurvene.

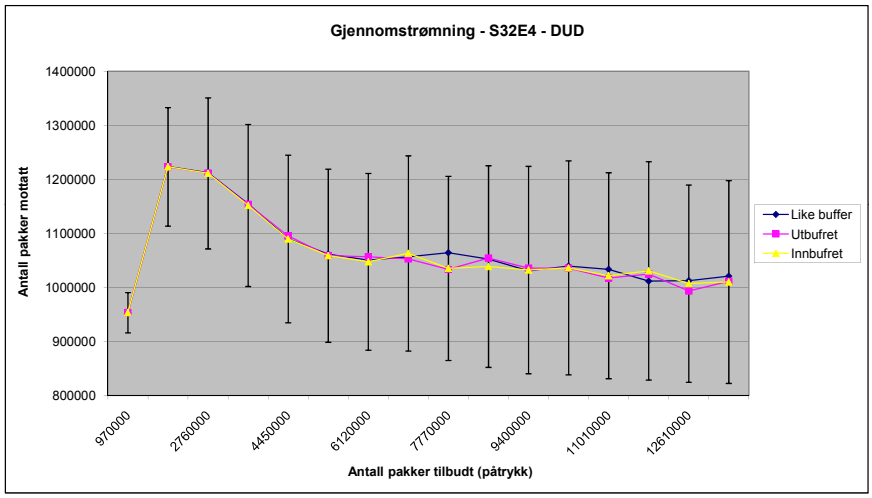
Den samme trenden fortsetter i det vi beveger oss over til topologier med 32 svitsjer. Figurene 4.13 og 4.14 viser plott av inn-, ut- og like-bufrede svitsjer for to nye kombinasjoner av topologi-størrelse og rutealgoritme: S32E1 kombinert med vektet korteste-vei-ruting med multippel kasting av pakker ved vranglås, og S32E4 kombinert med deterministisk up*/down*-ruting. Riktig nok er det nå litt enklere å se at de tre kurvene faktisk ikke er helt identiske, men variasjonene er fortsatt svært liten konfidensintervallene tatt i betraktning. Vi kan samtidig merke oss at det ikke er én av de tre kurvene som generelt ligger høyere enn de andre. Alle tre kurvene ser nærmest ut til å være tvunnet rundt hverandre, hvor en kurve som gir størst gjennomstrømning ett sted ved et litt annet påtrykk kommer dårligst ut av de tre.



Figur 4.12: Gjennomstrømning ved bruk av inn-, ut- og like-bufrede svitsjer. Topologi-størrelse S16E1. Svitsjene benytter seg av deterministisk korteste-vei-up*/down*-ruting.



Figur 4.13: Gjennomstrømning ved bruk av inn-, ut- og like-bufrede svitsjer. Topologi-størrelse S32E1. Vektet korteste-vei-ruting er benyttet, med multippel kasting av pakker ved vranglås.



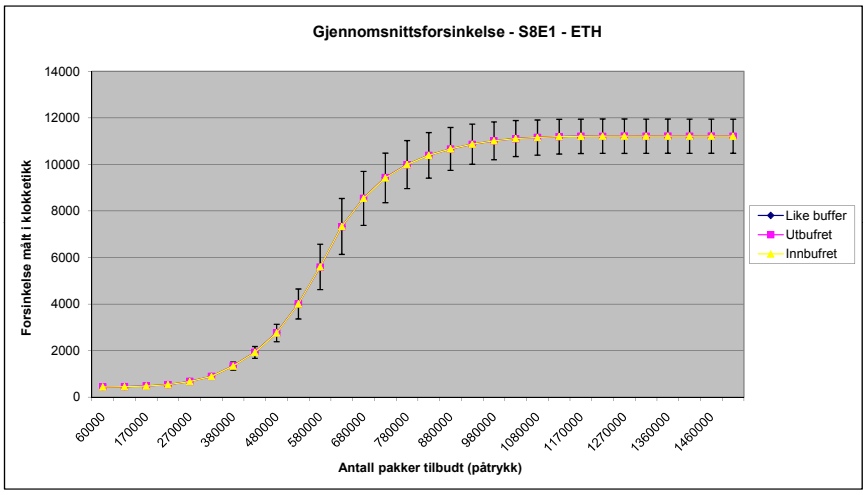
Figur 4.14: Gjennomstrømning ved bruk av inn-, ut- og like-bufrede svitsjer. Topologi-størrelse S32E4. Svitsjene benytter seg av deterministisk up*/down*-ruting.

4.4.2 Gjennomsnittsforsinkelse

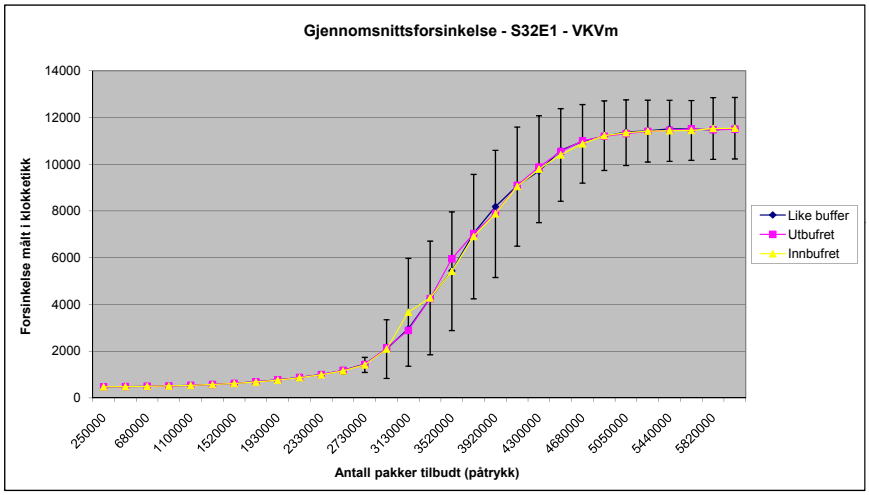
Figurene 4.15, 4.16 og 4.17 viser plott av gjennomsnittsforsinkelse for inn-, ut- og likebufrede svitsjer. De tre kombinasjonene av topologi-størrelse og rutealgoritme representert ved de tre plottene tilsvarer tre av de fire nettverkskonfigurasjonene vi benyttet da vi studerte gjennomstrømning i forrige avsnitt. Som vi ser er det også for gjennomsnittsforsinkelsen tilsynelatende svært liten forskjell på om man benytter inn-, ut- eller like-bufrede svitsjer. Noe variasjon er det mulig å spore, men som for gjennomstrømning er det ikke for noen av plottene slik at én av de tre kurvene alltid viser en lavere eller høyere forsinkelse enn de to andre. Lengden til konfidensintervallene ser ut til å være omtrent den samme uavhengig av om man har benyttet inn-, ut- eller like-bufrede svitjser. Med andre ord ser de tre forskjellige måtene å organsiere kantbuffer på ut til å ha liten innvirkning på den variasjon i gjennomsnittsforsinkelse man finner for ulike topologier innen samme topologi-størrelse.

4.4.3 Konklusjon

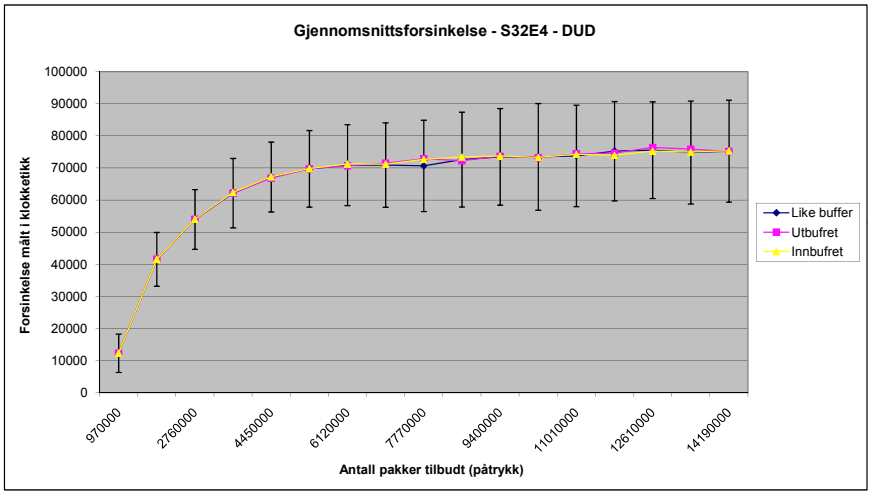
Både ved måling av gjennomstrømning og gjennomsnittsforsinkelse har det vist seg at simuleringer med inn-, ut- og like-bufrede svitsjer gir svært like resultater. Konfidensintervallenes lengder ser også ut til å være tilnærmet like. Med andre ord tyder våre simuleringer på at man ikke kan forvente høyere gjennomstrømning eller lavere forsinkelse ved å benytte seg av utbufrede



Figur 4.15: Gjennomsnittsforsinkelse ved bruk av inn-, ut- og like-bufrede svitsjer. Topologi-størrelse S8E1. Rutetabeller tilsvarer tradisjonelt Ethernet.



Figur 4.16: Gjennomsnittsforsinkelse ved bruk av inn-, ut- og like-bufrede svitsjer. Topologi-størrelse S32E1. Vektet korteste-vei-ruting er benyttet, med multippel kasting av pakker ved vranglås.



Figur 4.17: Gjennomsnittsforsinkelse ved bruk av inn-, ut- og like-bufrede svitsjer. Topologi-størrelse S32E4. Svitsjene benytter seg av deterministisk up*/down*-ruting.

---

svitsjer fremfor innbufrede. Dette er et resultat det kan være spennende å forfølge videre, siden man tradisjonelt har foretrukket utbufrede svitsjer for å unngå køhodeblokkering.

Før vi går videre kan vi raskt nevne at vi har gjentatt et utvalg av de simuleringer vi har benyttet i dette delkapittelet med den forskjell at alle nettverkene har hatt flytkontrollen permanent skrudd av. Resultater fra disse kjøringene bekrefter det inntrykket vi har fått fra simuleringer hvor flytkontroll har vært benyttet, men siden datagrunnlaget stammer fra et noe beskjedent antall simuleringer, og vi i et tett koblet nettverk ønsker å benytte flytkontroll, har vi valgt å holde disse resultatene utenfor denne oppgaven.

4.5 Sammenlikning av ulike triggerverdier for flytkontroll

Vår tredje og siste problemstilling lyder som følger:

> P3: *Når er det hensiktsmessig å skru på og av flytkontroll, og har kunnskap om dette implikasjoner for hvor store buffer svitsjene i et tett koblet nettverk bør ha?*

For å besvare denne problemstillingen er vi avhengig av å benytte oss av et annet sett simuleringer enn dem vi har benyttet tidligere i dette kapittelet. Som vi husker fra kapittel 2.3.3 skrur man på og av flytkontrollen ut fra

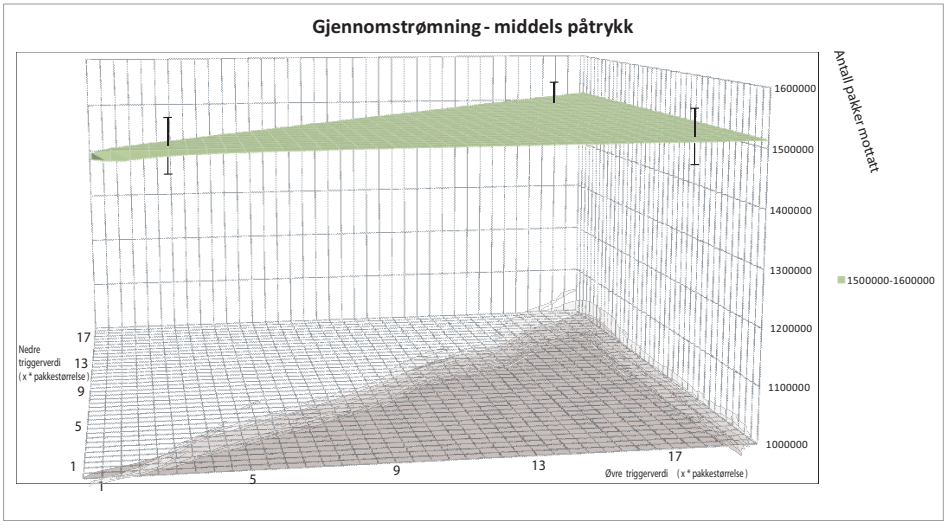
hvor fullt innbufferet er. Dersom innbufferet er fylt over en gitt grense $t_{på}$ skrur man flytkontrollen på, mens man skrur den av igjen så fort bufferet er tømt under en annen grense t_{av}. $t_{på}$ og t_{av} kaller vi gjerne henholdsvis øvre og nedre triggerverdi for flytkontrollen. De to grensene kan være sammenfallende, men det er viktig at $t_{på} \geq t_{av}$. Samtidig husker vi at $t_{på}$ ikke må settes så høyt at vi risikerer å kaste pakker på grunn av fullt innbuffer, og at t_{av} ikke bør settes så lavt at innbufferet tømmes unødvendig. For å besvare vår siste problemstilling skal vi benytte simuleringsoppsett som varierer de to triggerverdiene innenfor de grenser vi har gjengitt ovenfor, samtidig som nettverkskonfigurasjonen for øvrig holdes uforandret. Ved å måle gjennomstrømning og gjennomsnittsforsinkelse for de simulerte nettverk søker vi å danne oss et bilde av en eventuell variasjon i nettverkets ytelse som en funksjon av de to triggerverdiene.

De to triggerverdiene styrer ene og alene flytkontrollen ut fra hvor fullt innbufferet er. Vi skal derfor holde oss til innbufrede svitsjer. Et stort innbuffer gir større fleksibilitet når det gjelder det å variere de to triggerverdiene, enn om innbufferet er mindre. Samtidig har vi sett at innbufrede svitsjer i et tett koblet nettverk ser ut til å gi tilnærmet samme ytelse når det gjelder gjennomstrømning og gjennomsnittsforsinkelse, som et tilsvarende nettverk med ut- eller like-bufrede svitsjer. Våre innbufrede svitsjer har fortsatt plass til 20 pakker i sitt innbuffer og 4 pakker i sitt utbuffer.

Når det gjelder topologi-størrelse begrenser vi oss til S16E1. Vi skal imidlertid fortsatt benytte oss av 16 tilfeldig valgte topologier fra denne størrelsen for å beregne gjennomsnittsverdier for gjennomstrømning og gjennomsnittsforsinkelse. Konfidensintervaller vil bli beregnet på samme måte som tidligere, men siden våre plott denne gangen vil utgjøre tre-dimensjonale flater med svært mange målte verdier skal vi for lesbarhetens skyld kun angi et fåtall intervaller for hvert enkelt plott. De intervallene vi tegner inn er representative for de vi unnlater å ta med.

Deterministisk korteste-vei-up*/down* (DKVUD) er valgt som rutealgoritme. Dette er den av våre aktuelle rutealgoritmer for Ethernet som ser ut til å gi høyest gjennomstrømning og samtidig lavest gjennomsnittsforsinkelse. Vi har i tillegg kjørt initielle simuleringer med tradisjonelt Ethernet. Disse kjøringene har gitt resultater som tilsvarer dem vi har fått for DKVUD og blir av den grunn ikke presentert her.

Siden ulike kombinasjoner av $t_{på}$ og t_{av} potensielt gir svært mange simuleringsoppsett har vi sett oss nødt til å begrense variasjon i påtrykket en god del. Vi har valgt oss ut tre verdier som tilsvarer lavt, middels og høyt påtrykk. Ved lavt påtrykk er det så lite trafikk i nettverket at pakker skjelden konkurrerer om ressurser. Middels påtrykk tilsvarer et påtrykk som nærmer seg frafallspunktet. Nettverket klarer altså fortsatt over tid å hente unna all ønsket trafikk fra endenodene, men pakken vil ofte kunne oppleve ressurskonflikter på sin vei fra kilde til destinasjon. Ved høyt påtrykk er nettverket i ferd med å gå i metning. Frafallspunktet er passert, og pakker



Figur 4.18: Gjennomstrømning ved bruk av forskjellige triggerverdier for flytkontroll (middels påtrykk). Topologi-størrelse S16E1. Svitsjene benytter seg av deterministisk korteste-vei-up*/down*-ruting.

---

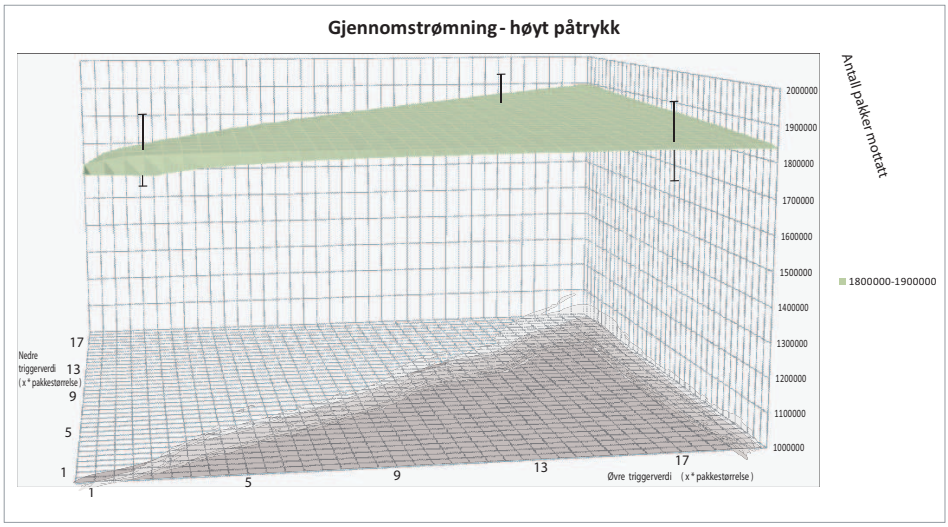
vil som regel måtte vente på tilgang til ressurser ved hver eneste svitsj de passerer i nettverket.

4.5.1 Gjennomstrømning

Ved lavt påtrykk var det tilnærmet umulig å spore variasjon i gjennomstrømning i det vi varierte våre to triggerverdier. Dette skyldes nok ganske enkelt at påtrykket er så lavt at innbufferne sjelden inneholder mer enn en til to pakker, og ofte gjerne bare deler av den pakke man er i ferd med å motta eller sende videre igjennom svitsjen.

Figur 4.18 viser gjennomstrømning som en funksjon av våre to triggerverdier ved middels påtrykk. Hver triggerverdi har blitt variert i steg tilsvarende en halv pakkestørrelse. Merk at det ikke er noe problem at triggerverdiene ikke er et multippel av pakkestørrelsen selv om vi benytter oss av lagre-og-videresend-svitsjing. Minimumsverdien for hver triggerverdi tilsvarer at det kun ligger en halv pakke i innbufferet. Maksimumsverdien er satt slik at det kun er plass til en halv pakke til i innbufferet. Dette er hos oss tilstrekkelig for at pakker ikke kastes på grunn av fulle innbuffer. Merk at nedre triggerverdi fortsatt må være mindre eller lik øvre triggerverdi. Flaten i figur 4.18 dekker derfor bare over den delen av rommet som er skyggelagt.

Som vi ser viser flaten i figur 4.18 svært liten variasjon i gjennomstrømning i det vi varierer våre triggerverdier (merk at den vertikale aksen ikke



Figur 4.19: Gjennomstrømning ved bruk av forskjellige triggerverdier for flytkontroll (høyt påtrykk). Topologi-størrelse S16E1. Svitsjene benytter seg av deterministisk korteste-vei-up*/down*-ruting.

---

starter på null). Variasjonen ligger stort sett innenfor et intervall på godt under én prosent. Det finnes to unntak: For det første finner vi en liten dupp i flaten når nedre triggerverdi er satt til halve pakkestørrelse. Forskjellen er fortsatt svært liten, men skyldes at vi faktisk har satt vår minimumsverdi litt for lavt. Riktig nok tømmes aldri innbufferet før første nye pakke ankommer etter at flytkontrollen har blitt skrudd av, men siden vi benytter lagre-og-videresend-svitsjing skrur vi likevel ikke flytkontrollen av tilstrekkelig tidlig. Vi må huske på at den nye pakken som ankommer svitsjen ikke kan videresendes før hele pakken er ankommet. For at vi kontinuerlig skal kunne sende pakker videre igjennom svitsjen må vi altså skru av flytkontrollen litt tidligere[2]. Det andre stedet hvor grafen dupper litt er i området hvor den øvre triggerverdien er lavere enn halvannen pakkestørrels. Forskjellen er imidlertid også her svært liten, godt under to prosent.

Figur 4.19 viser gjennomstrømning ved ulike triggerverdier for samme nettverkskonfigurasjon som den benyttet for figur 4.18, men denne gangen med høyt påtrykk. Vi ser fortsatt en dupp i flaten som skyldes at vår

[2] At vi ikke finner et like stort dupp i flaten når nedre triggerverdi er satt lik én ganger pakkestørrelsen skyldes at både siste pakke i innbufferet i det flytkontrollen skrus av, og første nye pakke som ankommer svitsjen, begge vil bli bremset litt av et ruteoppslag i svitsjen. Dette er nok til at duppen i grafen for dette konkrete simuleringsoppsette blir neglisjerbart. I et reelt nettverk må vi huske på at det ikke vil være lett å finne en presis og optimal nedre grense, siden en nedre grense blant annet vil avhenge av hvor langt tid flytkontroll-beskjeder bruker i nettverket, noe som igjen vil avhenge av nettverkstrafikken.

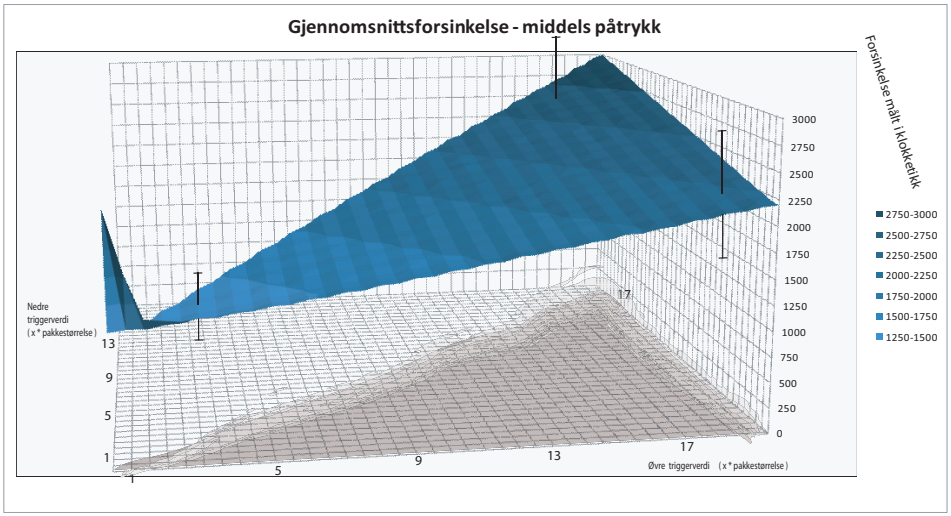
minimumsverdi for nedre triggerverdi er litt for lav. Samme duppen som tidligere er også til stede dersom øvre triggerverdi er under et par pakkestørrelser. Hele flaten ser nokså uforandret ut, men den heller faktisk i større grad en tidligere. Selv om det kan være litt vanskelig å se i figuren ligger flaten høyest i det området hvor de to triggerverdiene er tilnærmet like. Et toppunkt for flaten finner vi i området rundt der den øvre (og nedre) triggerverdien har sin maksimumsverdi. Forskjellene er fortsatt bare på noen få prosent, men det gir likevel et par svake hentydninger. For å oppnå størt mulig gjennomstrømning kan det se ut til at den nedre triggerverdien bør settes i nærheten av, eller sammenfalle med, den øvre triggerverdien. Samtidig kan det se ut til at man oppnår en økning i gjennomstrømning i det man øker den øvre triggerverdien. Om vi nå husker på at det å sette den øvre triggerverdi lavt tilsier at man kun benytter deler av en svitsjs innbuffer, kan vi relatere våre to funn til bufferbruk og bufferstørrelser hos svitsjer: For de bufferstørrelser vi har sett på er det for gjennomstrømningen sin del gunstig å søke en høy fyllningsgrad av bufferne (de to triggerverdiene bør ligge nær hverandre), og man kan forvente en svak økning i gjennomstrømning i det man øker bufferstørrelsen. Vi må imidlertid huske at vi ikke kan si noe om utviklingen i det vi passerer innbuffer på størrelse med 20 pakker, og at det først var ved høyt påtrykk man kunne spore en økning i gjennomstrømning på noen få prosent i det bufferstørrelsen økte.

4.5.2 Gjennomsnittsforsinkelse

Som ved måling av gjennomstrømning er det også for gjennomsnittsforsinkelse vanskelig å finne variasjon i det vi benyttet ulike triggerverdier for en nettverkskonfigurasjon med lavt påtrykk. Få pakker i nettverket medfører sjeldent venting. I det påtrykket ble økt til middels eller høyt ble imidlertid saken en ganske annen. Figurene 4.20 og 4.21 viser flater for målt gjennomsnittsforsinkelse for de samme simuleringssett som ble benyttet for å finne gjennomstrømning i forrige avsnitt. Som vi ser er de to flatene denne gangen langt fra parallelle med det horisontale planet.

Begge flatene har en pigg stikkende opp i området der begge triggerverdiene er satt til en verdi nær minimumsverdien vår. Denne piggen sammenfaller med et av de fallene vi tidligere så i gjennomstrømning i det øvre triggerverdi ble satt lavere enn halvannen pakkestørrelse. Når den øvre triggerverdien ligger så lavt, skrur man på flytkontroll så fort man har en pakke i innbufferet, samtidig som man, når pakken i bufferet videresendes, skrur av igjen flytkontrollen i seneste laget. Med andre ord holder flytkontrollen pakker unødvendig lenge igjen, noe som resulterer i økt gjennomsnittsforsinkelse.

Om vi ser på resten av de to flatene ser vi den samme tendensen vi så for gjennomstrømning: Flatene ligger høyest i området hvor de to triggerverdiene holdes like, og har et toppunkt i det området hvor begge verdiene ligger tett opp mot vår maksimumsverdi. Når det gjelder gjennomsnittsforsinkelse er vi



Figur 4.20: Gjennomsnittsforsinkelse ved bruk av forskjellige triggerverdier for flytkontroll (middels påtrykk). Topologi-størrelse S16E1. Svitsjene benytter seg av deterministisk korteste-vei-up*/down*-ruting.

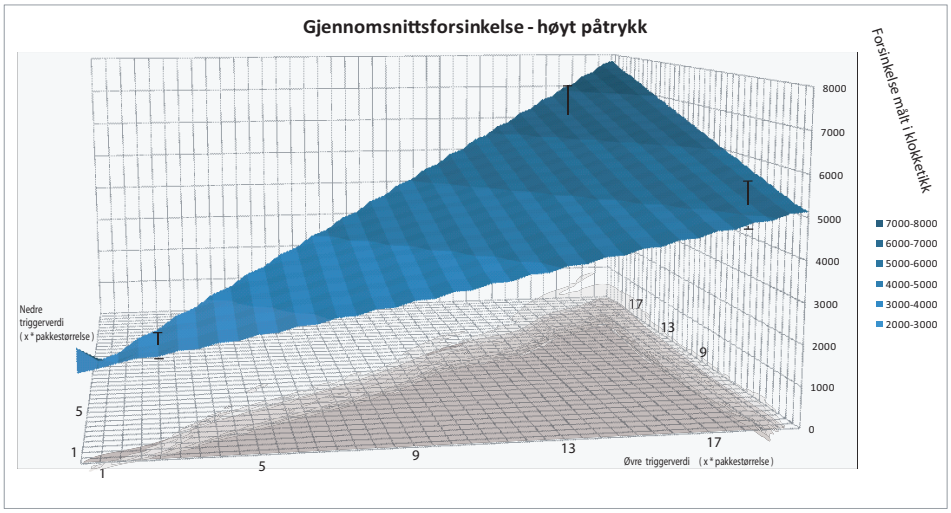
---

imidlertid interessert i lavest mulig verdi, med andre ord flatens bunnpunkt. Denne finner vi i området der de to triggerverdiene er små, men da slik at de holder seg over én pakkestørrelse.

Mens gjennomstrømningen kun opplevde små variasjoner i det triggerverdiene varierte, ser vi at situasjonen er en ganske annen når vi måler gjennomsnittsforsinkelsen. For middels påtrykk representerer flatens toppunkt en gjennomsnittsforsinkelse som er nesten tre ganger så stor som den vi finner i området der øvre triggerverdi er satt til å tilsvare rundt to, tre pakkestørrelser. For høyt påtrykk er forskjellen mellom flatens topp- og bunnpunkt enda større, rundt tre og en halv gang. At forsinkelsen øker så markant i det triggerverdiene øker skyldes at høyere triggerverdier, sammen med høyt påtrykk, både tilsvarer større buffer og høyer fyllningsgrad av bufferne. Dermed vil tiden hver enkelt pakke gjennomsnittlig tilbringer i buffer hos svitsjer i påvente av tilgang til ledige ressurser øke, noe som igjen medfører økt forsinkelse.

4.5.3 Konklusjon

Når det gjelder gjennomstrømning ser det ut til at man, i det påtrykket blir stort, kan oppnå en svak økning dersom man velger å legge de to triggerverdiene for flytkontroll nær hverandre, slik at fyllningsgraden til svitsjenes buffer holdes høyt. Videre kan et stort innbuffer se ut til å ha en positiv effekt på gjennomstrømningen. Økningen er imidlertid marginal sammenliknet med den forskjell man samtidig kan se ut til å ville oppleve for gjennomsnittsfor-



Figur 4.21: Gjennomsnittsforsinkelse ved bruk av forskjellige triggerverdier for flytkontroll (høyt påtrykk). Topologi-størrelse S16E1. Svitsjene benytter seg av deterministisk korteste-vei-up*/down*-ruting.

---

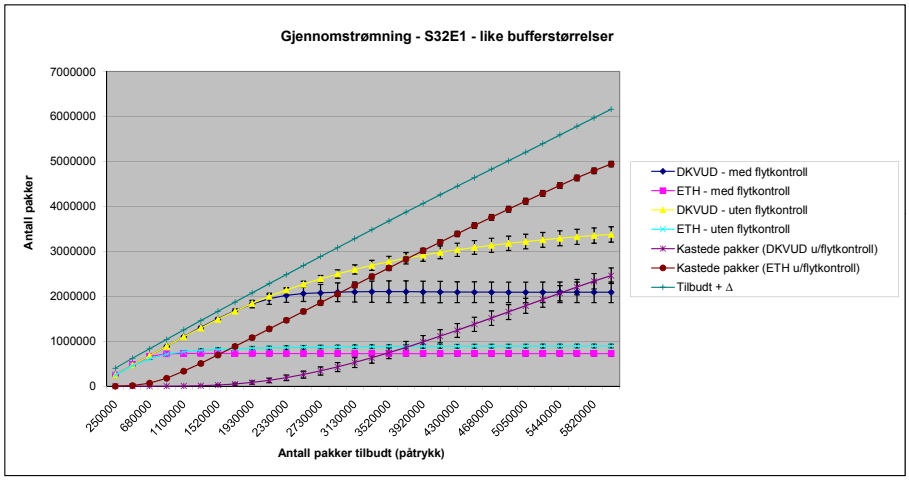
sinkelsen. Dessverre er situasjonen nå snudd på hodet: Det er små buffere og lav fyllningsgrad som gir lav gjennomsnittsforsinkelse, forutsatt at man holder de to triggerverdiene over en minimumsgrense slik at innbufferet ikke tømmes unødvendig. Minimumsgrensen vil avhenge av det konkrete nettverket man benytter. Med mindre forsinkelsen til pakkene i nettverket spiller liten rolle, ser det altså ut til at små buffer er å foretrekke. Små buffer gir en lav øvre grense for den forsinkelsen en pakke maksimalt kan oppleve på sin vei gjennom nettverket, forutsatt fravær av utsulting og vranglås, uten at gjennomstrømningen ser ut til å lide vesentlig. For tett koblede nettverk ser det da ut til at små buffer vil være å foretrekke, siden forsinkelse i denne typen nettverk spiller en viktig rolle.

4.6 Tradisjonelt Ethernet og deterministisk korteste-vei-up*/down*-ruting, med og uten flytkontroll

Tidlig i oppgaven hevdet vi at vi er avhengig av å benytte flytkontroll for at man ikke skal kaste pakker i et tett koblet nettverk. For at denne påstanden ikke skal stå ubegrunnet skal vi koste på oss en liten figur som illustrerer behovet for flytkontroll, før vi går videre til neste kapittel for å avrunde oppgaven.

Figur 4.22 viser gjennomstrømning for tradisjonelt Ethernet (ETH) og Ethernet med deterministisk korteste-vei-up*/down*-ruting (DKVUD). Begge rutealgoritmene er benyttet både med og uten flytkontroll. Som tidligere baserer plottet seg på 16 tilfeldig valgte topologier fra samme topologistørrelse, her S32E1. Svitsjene benytter inn- og utbuffer som er like store. I figuren har vi i tillegg til gjennomstrømning også plottet inn antall kastede pakker grunnet fravær av flytkontroll. Vi ser tydelig at man for begge rutealgoritmene starter å kaste et vesentlig antall pakker i det man passerer frafallspunktet dersom flytkontroll ikke benyttes. Riktig nok ser vi at DKVUD uten flytkontroll kan skilte med best gjennomstrømning i det nettverket går mot metning, men prisen er samtidig høy: Svært mange pakker kastes i nettverket.

Ytterligere resultatplott fra simulering av ulike nettverkskonfigurasjoner, med og uten flytkontroll, finnes i tillegg A.2.



Figur 4.22: Topologi-størrelse S32E1. Plottet viser gjennomstrømning ved bruk av tradisjonelt Ethernet og Ethernet med deterministisk korteste-vei-up*/down*-ruting, begge benyttet med og uten flytkontroll. Antall kastede pakker ved fravær av flytkontroll er også plottet inn.

Kapittel 5

Avrunding

5.1 Oppsummering

Vi har i løpet av denne oppgaven studert tre ulike sider ved det å benytte Ethernet-teknologi i tett koblede nettverk; vi har studert alternative rutealgoritmer for Ethernet, vi har sett på organisering av buffer i svitsjene og vi har studert hvordan ytelsen i nettverket påvirkes i det vi varierer triggerverdiene for flytkontrollen. Vi skal i påfølgende avsnitt oppsummere våre viktigste funn.

Ethernet og up*/down*-ruting

I vår første problemstilling spør vi om vi kan forvente økt ytelse fra Ethernet som tett koblede nettverk om vi bytter ut Ethernets tradisjonelle rutetabeller med rutetabeller bygget opp ved hjelp av up*/down*-ruting. Alle våre simuleringer tyder på at dette spørsmålet kan besvares med et klart "ja". I det vi tok kontroll over rutetabellene til Ethernet-svitsjene, og fylte dem ved hjelp av alternative rutealgoritmer, så vi at våre to ulike varianter av deterministisk up*/down*-ruting gav en vesentlig ytelsesøkning i forhold til tradisjonelt Ethernet. Best ut av de to up*/down*-variantene kom detetereministisk korteste-vei-up*/down*-ruting (DKVUD). DKVUD gav helt opp mot tre ganger så høy gjennomstrømning som tradisjonelt Ethernet, samtidig som forsinkelsen ble redusert til under det halve. Riktig nok så vi også at varianter av korteste-vei-ruting oppnådde et enda bedre resultat enn DKVUD, men siden disse resultatene baserer seg på en kunstig effektiv vranglåshåndtering kan man ikke forvente like god ytelse om man implementerer korteste-vei-ruting i et reelt nettverk.

Inn-, ut- og like-bufrede svitsjer

I nettversklitteraturen argumenteres det ofte for at utbufrede svitsjer på grunn av køhodeblokkering er å foretrekke fremfor innbufrede svitsjer[31,

39, 42, 44]. Siden argumentasjonen i de referanser vi har funnet kun tar utgangspunkt i hvordan én enkelt svitsj fungerer, spør vi i vår andre problemstilling om vi kan trekke en tilsvarende konklusjon for et tett koblet nettverk av svitsjer med kantbuffer; er det fortsatt slik at utbufrede svitjser er å foretrekke fremfor innbufrede?

Både inn-, ut- og like-bufrede svitsjer har oppnådd tilnærmet identiske resultater ved alle våre simuleringer. Riktig nok er det slik at vi har kunnet spore små variasjoner i resultatene i det påtrykket har blitt variert, men ikke for noen av våre nettverkskonfigurasjoner har én av de tre svitsje-typene kontinuerlig hatt en ytelse som har ligget over (eller under) de andre to. Vår konklusjon blir da at det i et tett koblet nettverk ser ut til å spille liten rolle hvor stor andel av et kantbuffer hos en svitsj som benyttes som innbuffer (eller utbuffer).

Triggerverdier for flytkontroll og bufferstørrelser

I vår siste problemstilling spør vi oss når det lønner seg å skru på og av flytkontrollen, og om kunnskap om dette kan si oss noe om hvor store buffer svitsjene i et tett koblet nettverk bør ha. Våre simuleringer har vist oss at gjennomstrømning og gjennomsnittsforsinkelse i et tett koblet nettverk påvirkes ulikt av hvordan triggerverdiene for flytkontroll settes; for lavt påtrykk ble trafikken i nettverket i svært liten grad påvirket av hvordan triggerverdiene ble satt, mens vi i det påtrykket økte kunne se at de triggerverdier som påvirket gjennomstrømningen i positiv retning, samtidig påvirket forsinkelsen på en negative måte. Generelt hadde en høy fyllningsgrad av innbufferne en positiv effekt på gjennomstrømningen, men samtidig en negativ effekt på gjennomsnittsforsinkelsen. Mens forskjellen i gjennomstrømning kun viste en variasjon på noen få prosent i det nettverket nesten hadde gått i metning, så vi imidlertid at gjennomsnittsforsinkelsen alt ved et middels påtrykk ble tredoblet dersom vi valgte "uheldige" triggerverdier. For gjennomsnittsforsinkelsen sin del var det helt klart lite gunstig å lagre flere pakker hos hver enkelt svitsj enn det som strengt tatt var nødvendig for å sikre en jevn trafikkflyt gjennom svitsjen. Både høy gjennomstrømning og lav gjennomsnittsforsinkelse er ønskede egenskaper ved et tett koblet nettverk, men siden gjennomsnittsforsinkelsen i vesentlig større grad enn gjennomstrømningen ser ut til å bli påvirket av innbuffernes fyllningsgrad lar vi gjennomsnittsforsinkelsen lede oss: Vi konkluderer med at små innbuffer er av det gode for svitsjer som skal benyttes i et tett koblet nettverk. Innbufferne trenger kun å ha plass til et lite antall pakker slik at man sikrer en jevn trafikkstrøm gjennom svitsjen. For våre simuleringsoppsett tilsvarte dette en bufferstørrelse på rundt to pakker.

Vi kan altså klare oss med et lite innbuffer, men hva med størrelsen på utbufferet? De simuleringer vi benyttet for å teste ulike triggerverdier benytter seg av et lite utbuffer, så hvordan kan vi vite at vi ikke oppnår bedre

ytelse dersom vi øker utbufferet igjen? Svaret på dette spørsmålet finner vi om vi vender tilbake til vår problemstilling nummer to. Der konkluderte vi med at inn- og utbufrede svitsjer vil gi tilnærmet samme ytelse i et tett koblet nettverk. Med andre ord vil vi kunne forvente at et nettverk hvor svitsjene har store utbuffer og små innbuffer yter svært likt et nettverk hvor svitsjene har små utbuffer og store innbuffer. Men i forbindelse med problemstilling nummer tre konkluderte vi jo nettopp med at et nettverk hvor svitsjene har små utbuffer og store innbuffer yter dårligere enn et nettverk hvor både inn- og utbufrene i svitsjene er små. Dermed kan vi ved hjelp av vårt svar på problemstilling nummer to også konkludere med at et nettverk med svitsjer hvor inn- og ut-bufferne er små vil gi en bedre ytelse enn hva man kan forvente fra et nettverk hvor svitsjene har store utbuffer og små innbuffer.

5.2 Videre arbeid

For å begrense antall simuleringsoppsett til et håndterbart nivå så vi oss nødt til å begrense oss til én statistisk modell for destinasjonsdistribusjon. Vår Zipf-implementasjon så vi oss denne gangen ikke i stand til å benytte. Det hadde vært spennende å gjenta våre simuleringer med en slik destinasjonsdistribusjon for å se om dette ville påvirket våre resultater nevneverdig.

Videre kunne det vært av interesse å simulere over større topologier enn de vi har begrenset oss til. I det topologien vokser vil trafikkbildet kunne kompliseres. Vi så for eksempel at vranglåsproblemet ikke gjorde seg særlig gjeldende før våre nettverk bestod av 32 svitsjer med én endenode per svitsj. Videre så vi at nettverket gikk mot en annen form for metning i det vi firedoblet antall endenoder per svitsj. Forandring i metningsbildet som en konsekvens av at man varierer antall endenoder koblet til hver svitsj, føler vi at vi kunne hatt nytte av å belyse ytterligere ved nye simuleringer.

Innledningsvis lot vi oss motivere av muligheten til å benytte rimelige Ethernet-svitsjer for å konstruere tett koblede nettverk. Vi har sett at det spiller liten rolle om svitsjene er inn- eller utbufret, og at små, og dermed billige, buffer faktisk kan gi oss den beste kombinasjonen av gjennomsnittsforsinkelse og gjennomstrømning. Videre har vi sett at up*/down*-ruting kan gi en vesentlig ytelsesøkning for tradisjonelt Ethernet. Samtidig må vi huske på at up*/down*-ruting tar utgangspunkt i et spenntre. Selv om alle linker i nettverket benyttes vil linker nær roten i et slikt tre kunne utgjør flaskehalser i nettverket. Up*/down*-ruting har for våre tilfeldig valgte topologier gitt en forholdsvis god ytelse sammenliknet med korteste-vei-ruting, men vi føler at det fortsatt kunne vært nyttig med simuleringer over andre topologier for å gi et bredere bildet av den ytelsen man kan forvente fra up*/down*. Hvordan vil for eksempel up*/down*-ruting yte for ulike regulære topologier? For forskjellige regulære topolgier finnes det godt kjente rutealgoritmer det kunne vært interessant å sammenliknet up*/down* med. Riktig nok kan man ikke

forvente at en generell rutealgoritme som up*/down* gir samme ytelsen som en rutealgoritme skreddersydd for en konkret regulær topologi, men et utvalg rutealgoritemer for regulære topologier sammenliknet med up*/down* vil likevel kunne gi økt innsikt i hvor godt up*/down*-algoritmen faktisk yter.

Bibliografi

[1] The ethernet: a local area network: data link layer and physical layer specifications. *SIGCOMM Comput. Commun. Rev.*, 11(3):20–66, 1981.

[2] IEEE Standard 802.1D. Media Access Control (MAC) Bridges. 1990.

[3] IEEE Standard 802.1D. Media Access Control (MAC) Bridges. 2004.

[4] IEEE Standard 802.3ae. Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications. Amendment 1: Media Access Control (MAC) Parameters, Physical Layers, and Management Parameters for 10 Gb/s Operation. 2002.

[5] IEEE Standard 802.3x. Specification for 802.3 Full Duplex Operation. 1997.

[6] IEEE Standard 802.6. Distributed Queue Dual Bus (DQDB) Subnetwork of a Metropolitan Area Network (MAN). 1990.

[7] Lada A. Adamic and Bernardo A. Huberman. Zipf's law and the internet. *Glottometrics*, 3:143–150, 2002.

[8] Jeff Andrews and Nick Baker. Xbox 360 system architecture. *IEEE Micro*, 26(2):25–37, 2006.

[9] ANSI. X3T9, Fibre Channel Physical and Signaling Interface (FC-PH), ANSI, New York, 1993.

[10] InfiniBand Trade Association. InfiniBand™ Architecture Specification Volume 2, Release 1.2.1. http://www.infinibandta.org.

[11] James Aweya, Michel Ouellette, and Delfin Y. Montuno. Interworking of switched ethernet and atm flow control mechanisms. *Int. J. Netw. Manag.*, 12(6):357–366, 2002.

[12] J. Beran, R. Sherman, M. S. Taqqu, and W. Willinger. Longrange dependence in variable bit-rate video traffic. *IEEE Transactions on Communications*, 43:1566–1579, 1995.

[13] Gouri K. Bhattacharyya and Richard A. Johnson. *Statistical Concepts and Methods*. John Wiley & Sons, 1977.

[14] Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Satterthwaite, Michael D. Schroeder, and Charles P. Thacker. Autonet: a high-speed self-configuring local area network using point-to-point links. *SRC Research Report 59*, April 1990.

[15] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[16] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. On the implications of zipf's law for web caching. Technical Report CS-TR-1998-1371, 1998.

[17] John S. Carson. Modeling and simulation worldviews. In *WSC '93: Proceedings of the 25th conference on Winter simulation*, pages 18–23, New York, NY, USA, 1993. ACM Press.

[18] Ole-Johan Dahl. *Verifiable Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1992.

[19] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[20] Trang Dinh Dang, Sándor Molnár, and István Maricza. Capturing the complete multifractal characteristics of network traffic. Dept. of Telecommunications & Telematics, Budapest University of Technology & Economics.

[21] Jose Duato, Sudhakar Yalamanchili, and Ni Lionel. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[22] Itamar Elhanany, Derek Chiou, Vahid Tabatabaee, Raffaele Noro, and Ali Poursepanj. The network processing forum switch fabric benchmark specifications: an overview. *IEEE Network*, 19(2):5–9, 2005.

[23] Mike Galles. Spider: A high-speed network interconnect — raising the bandwidth ceiling with a scalable, pipelined interconnect for distributed endpoint routing. *IEEE Micro*, 17(1):34–39, January/February 1997.

[24] Anthony P. Galluscio, John T. Douglass, Brian A. Malloy, and A. Joe Turner. A comparison of two methods for advancing time in parallel discrete event simulation. In *WSC '95: Proceedings of the 27th conference on Winter simulation*, pages 650–657, 1995.

[25] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification*. Addison-Wesley Professional, third edition, 2005.

[26] Ralph P. Grimaldi. *Discrete and Combinatorial Mathematics*. Addison-Wesley Publishing Company, third edition, 1994.

[27] IEEE 802.3 Backplane Ethernet Study Group. Offisiell hjemmeside: http://www.ieee802.org/3/ap/.

[28] Robert Whiting Horst. TNet: A reliable system area network. *IEEE Micro*, 15(1):37–45, February 1995.

[29] Jeffery A. Joines and Stephen D. Roberts. Fundamentals of object-oriented simulation. In *WSC '98: Proceedings of the 30th conference on Winter simulation*, pages 141–150, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[30] Hartmut Jürgens, Heinz-Otto Peitgen, and Dietmar Saupe. *Chaos and Fractals, New Frontiers of Sience*. Springer-Verlag, 1992.

[31] Mark J. Karol, Michael G. Hluchyj, and Samuel P. Morgan. Input versus output queueing on a space-division packet switch. *IEEE Transactions on Communications*, 35(12):1347–1356, 1987.

[32] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, 2006.

[33] Donald E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Professional, 1998.

[34] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[35] Amund Kvalbein. Bridging in RPR networks, Evaluation of an enhanced bridging algorithm. Simula Research Laboratory, Oslo, 2003.

[36] Will E. Leland, Murad S. Taqq, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of Ethernet traffic. In Deepinder P. Sidhu, editor, *ACM SIGCOMM*, pages 183–193, San Francisco, California, 1993.

[37] Lionel M. Ni and Philip K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2):62–76, 1993.

[38] Vern Paxson and Sally Floyd. Wide area traffic: the failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, 1995.

[39] Larry L. Peterson and Bruce S. Davie. *Computer Networks, A Systems Approach*. Morgan Kaufmann Publishers, Inc, 1996.

[40] Daniel A. Reed and Dirk C. Grunwald. The performance of multicomputer interconnection networks. *Computer*, 20(6):63–73, 1987.

[41] Thomas J. Schriber and Daniel T. Brunner. Inside simulation software: how it works and why it matters: inside discrete-event simulation software: how it works and why it matters. In *WSC '02: Proceedings of the 34th conference on Winter simulation*, pages 97–107. Winter Simulation Conference, 2002.

[42] Rich Seifert. *The Switch Book*. John Wiley & Sons, 2000.

[43] Ian Sommerville. *Software engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, fifth edition, 1996.

[44] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall PTR, third edition, 1996.

[45] Murad S. Taqqu, Vadim Teverosky, and Walter Willinger. Is network traffic self-similar or multifractal? *Fractals*, 5(1):63–73, 1997.

[46] Murad S. Taqqu, Walter Willinger, and Robert Sherman. Proof of a fundamental result in self-similar traffic modeling. *ACMCCR: Computer Communication Review*, 27, 1997.

[47] Boye Wangensteen. *Tanums store rettskrivningsordbok,* gjennomgått av Norsk språkråd. Kunnskapsforlaget, eight edition, 1996.

[48] Sugath Warnakulasuriya and Timothy Mark Pinkston. Implementation of deadlock detection in a simulated network environment. CENG Technical Report 97-01, 1997.

[49] J. Wechta, Armin Eberlein, and F. Halsall. The interaction of the TCP flow control procedure in end nodes on the proposed flow control mechanism for use in IEEE 802.3 switches. In *HPN*, pages 515–534, 1998.

[50] Mark Allen Weiss. *Data structures and algorithm analysis*. Benjamin-Cummings Publishing Co., Inc., 1992.

[51] Carey Williamson. Internet traffic measurement. *IEEE Internet Computing*, 5(6):70–74, 2001.

[52] 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.

[53] Yunkai Zhou and Harish Sethu. A simulation study of the impact of switching systems on self-similar properties of traffic. In *Proceedings of the IEEE Workshop on Statistical Signal and Array Processing*, Pennsylvania, USA, August 2000. Pocono Manor.

[54] George Kingsley Zipf. *The Psycho-biology of Language.* M.I.T. Press, Cambridge, 1965.

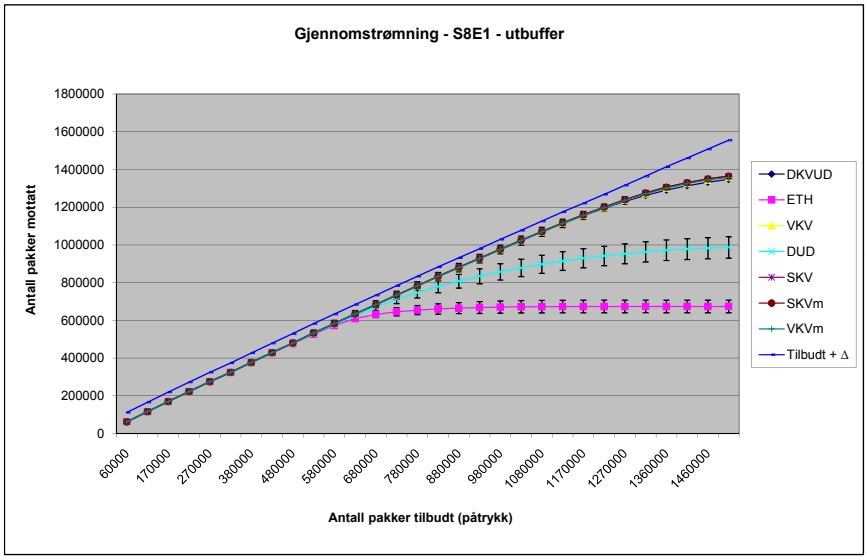
Tillegg A

Resultatplott

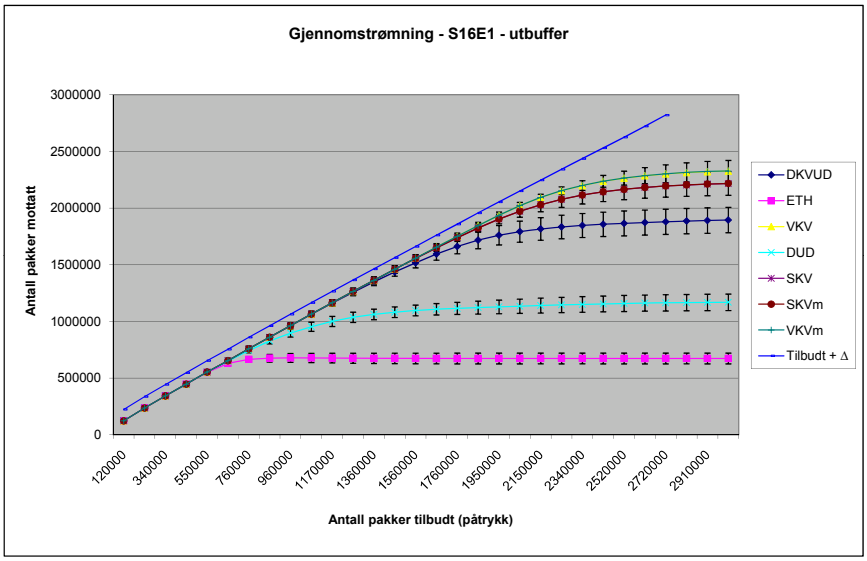
A.1 Simulering av ulike rutealgoritmer

Dette tillegget inneholder resultatplott fra simuleringer kjørt både for å teste ulike rutealgoritmer opp mot hverandre, samt for å teste tre typer svitsjer: svitsjer med størst innbuffer, svitsjer med størst utbuffer og svitsjer med like store inn- og utbuffer.

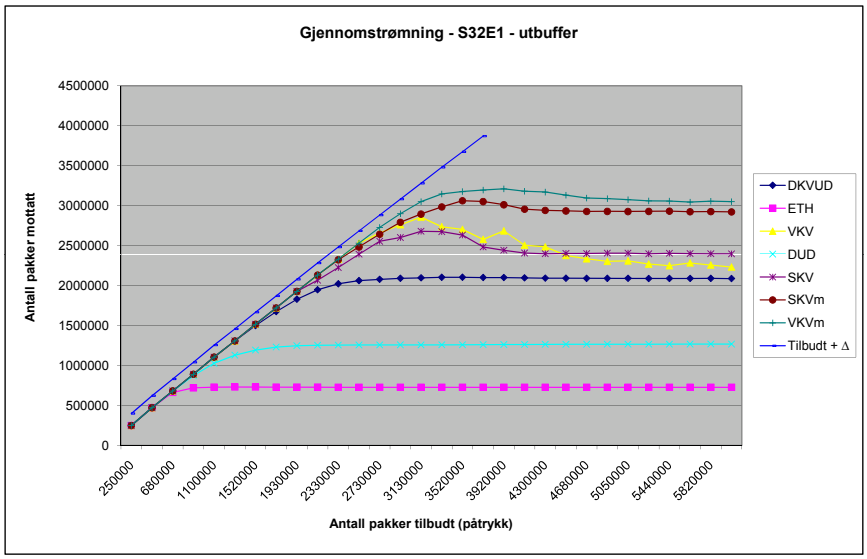
Gjennomstrømning



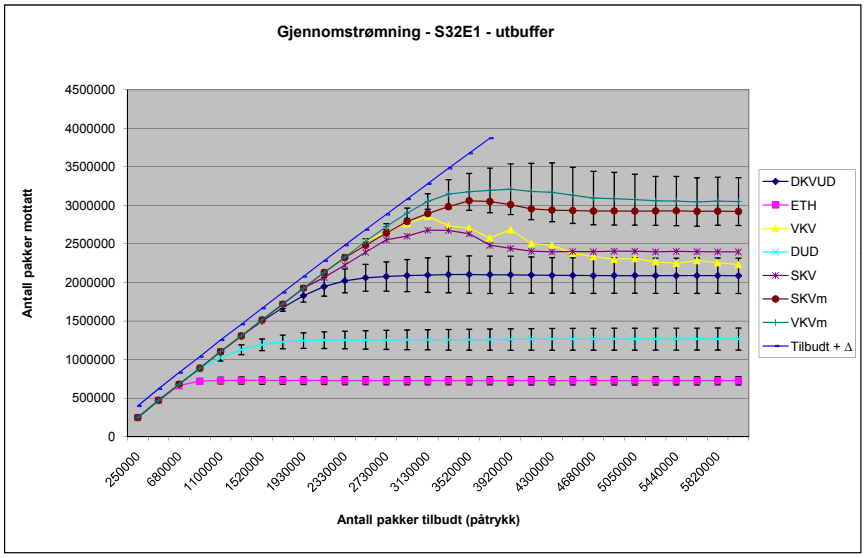
Figur A.1: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S8E1. Svitsjene har størst utbuffer.



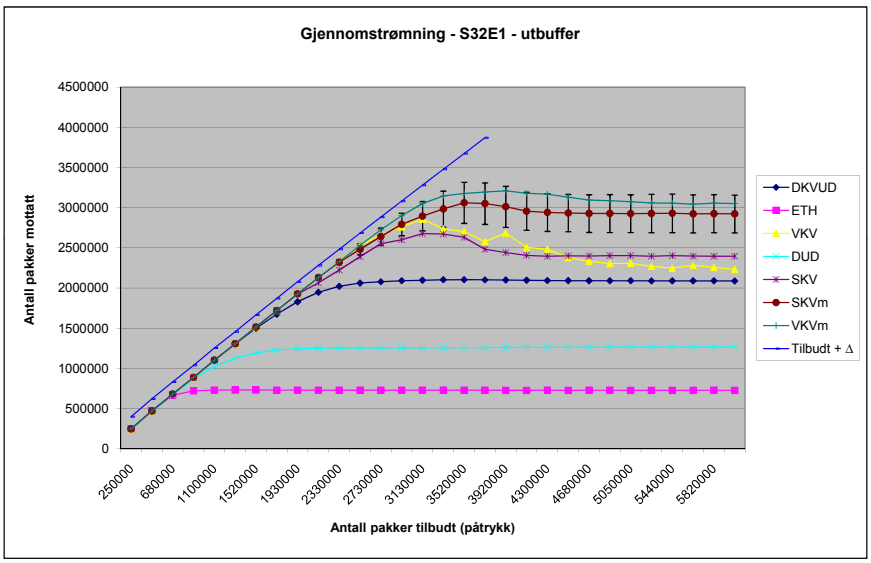
Figur A.2: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S16E1. Svitsjene har størst utbuffer.



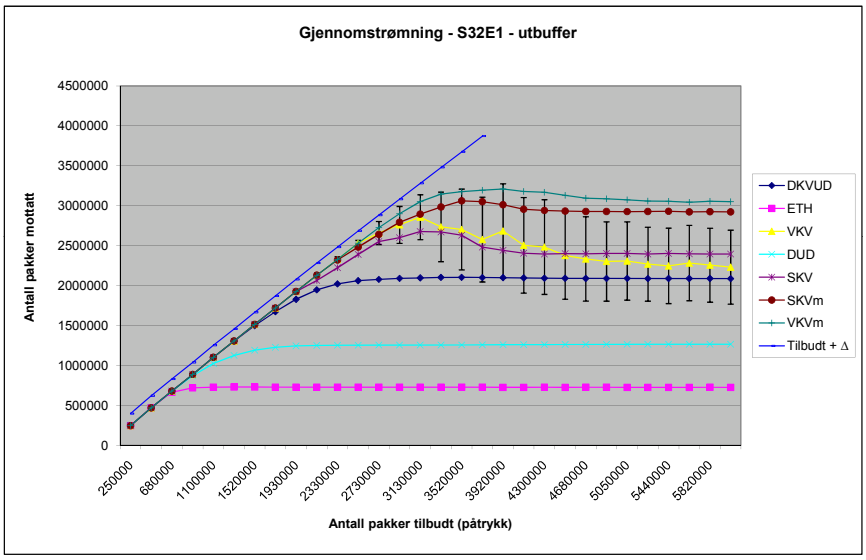
Figur A.3: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E1. Svitsjene har størst utbuffer. Plott uten konfidensintervaller.



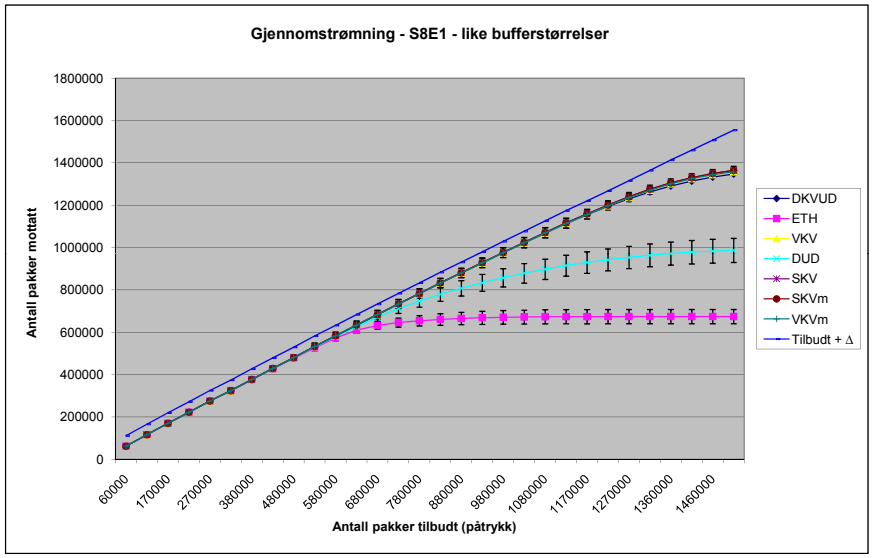
Figur A.4: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E1. Svitsjene har størst utbuffer. Plott 1 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.



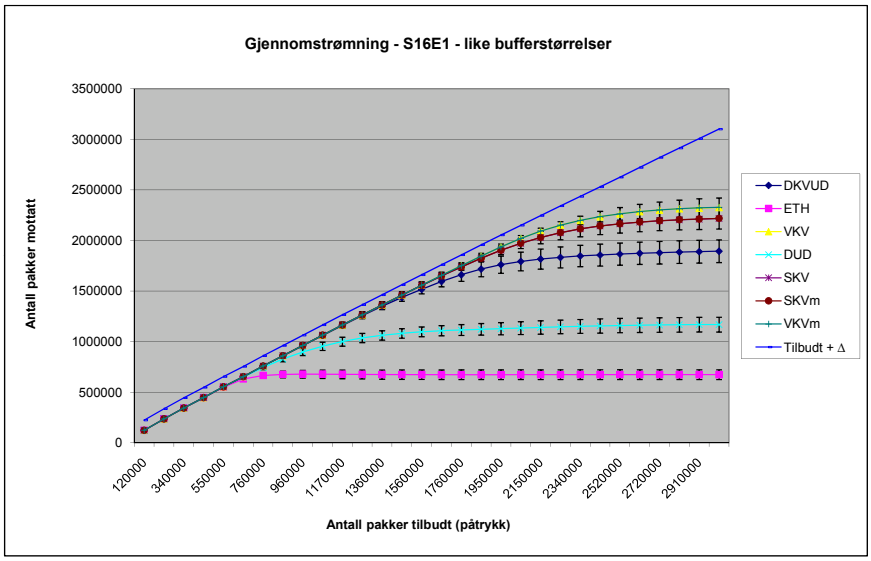
Figur A.5: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E1. Svitsjene har størst utbuffer. Plott 2 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.



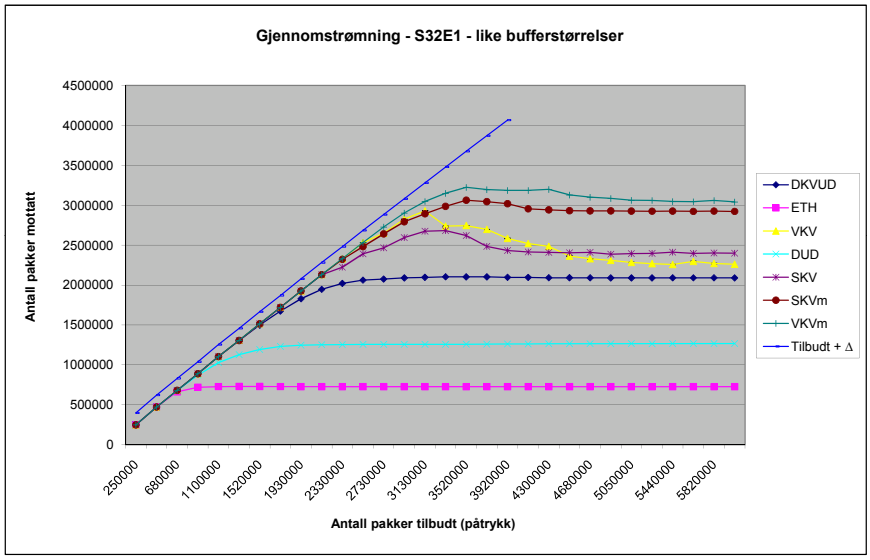
Figur A.6: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E1. Svitsjene har størst utbuffer. Plott 3 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.



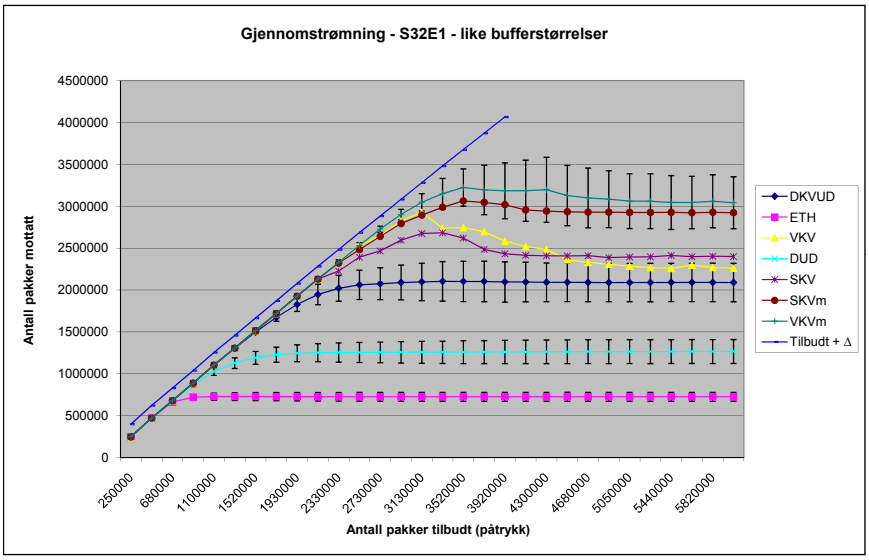
Figur A.7: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S8E1. Svitsjene benytter inn- og utbuffer som er like store.



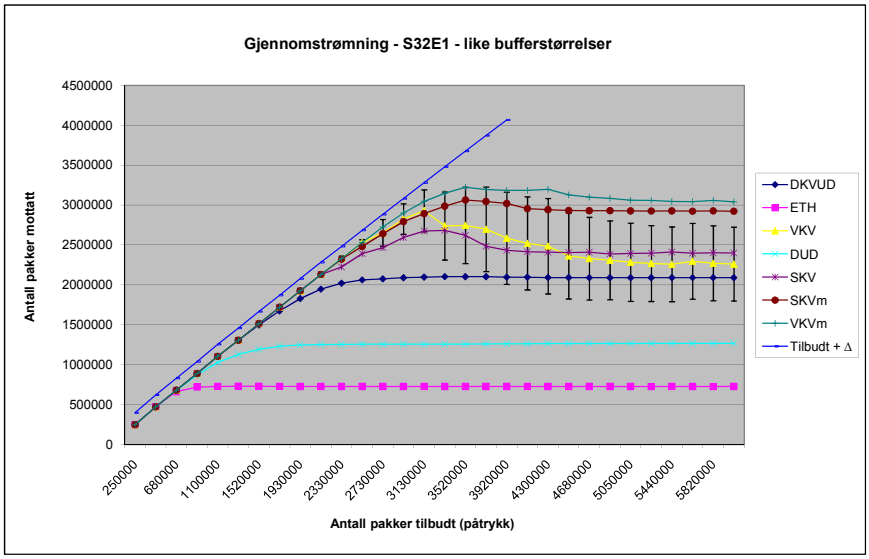
Figur A.8: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S16E1. Svitsjene benytter inn- og utbuffer som er like store.



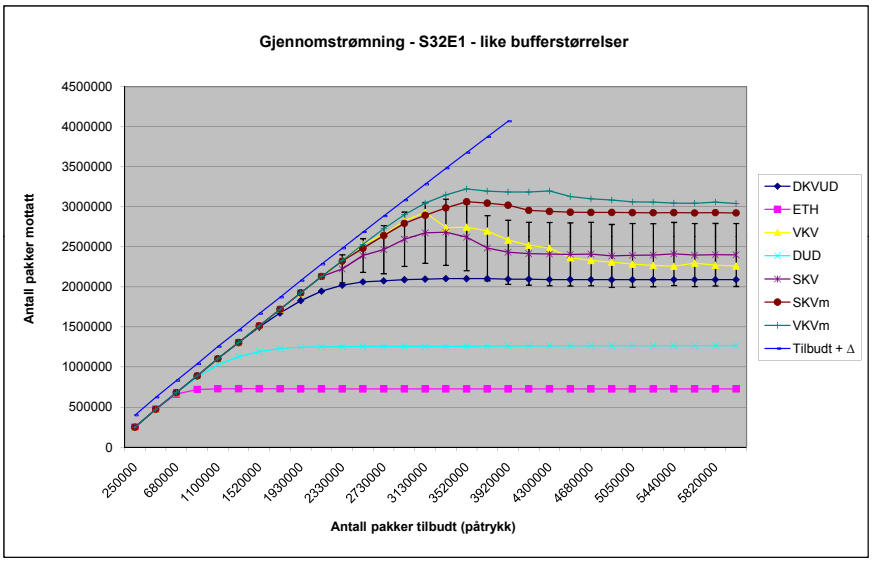
Figur A.9: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E1. Svitsjene benytter inn- og utbuffer som er like store. Plott uten konfidensintervaller.



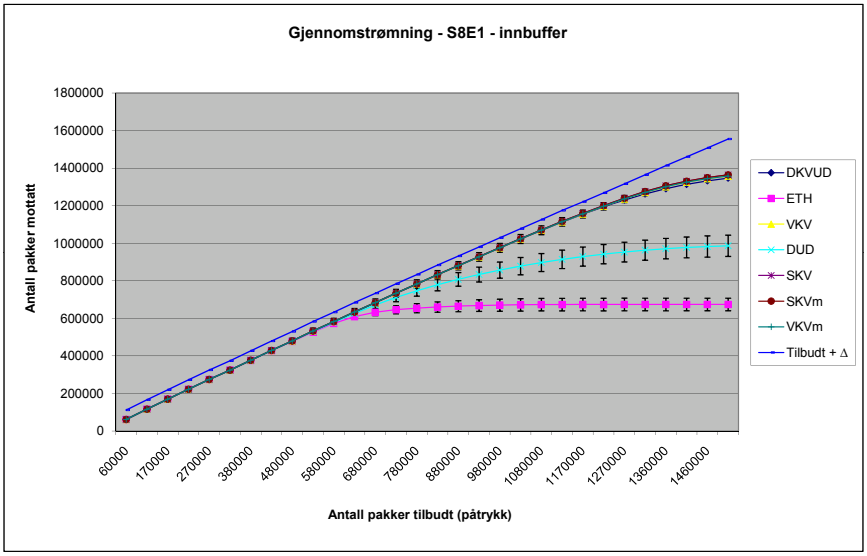
Figur A.10: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E1. Svitsjene benytter inn- og utbuffer som er like store. Plott 1 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.



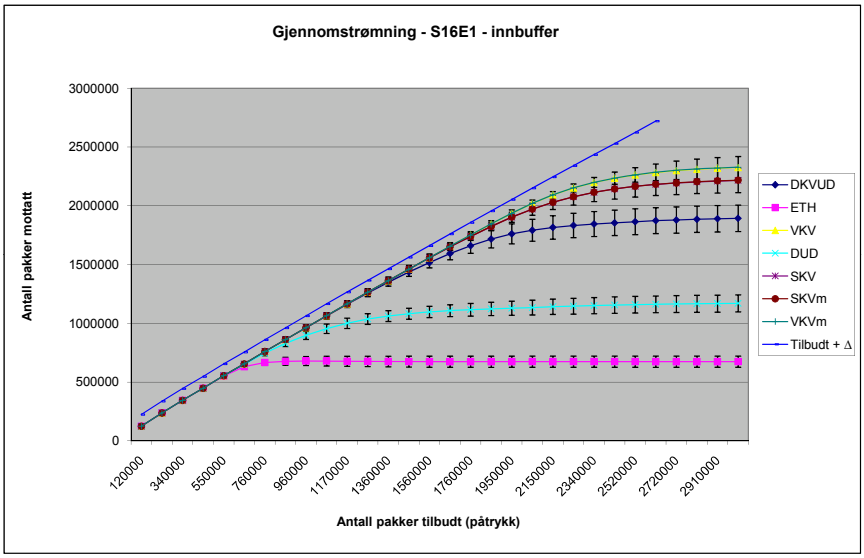
Figur A.11: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E1. Svitsjene benytter inn- og utbuffer som er like store. Plott 2 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.



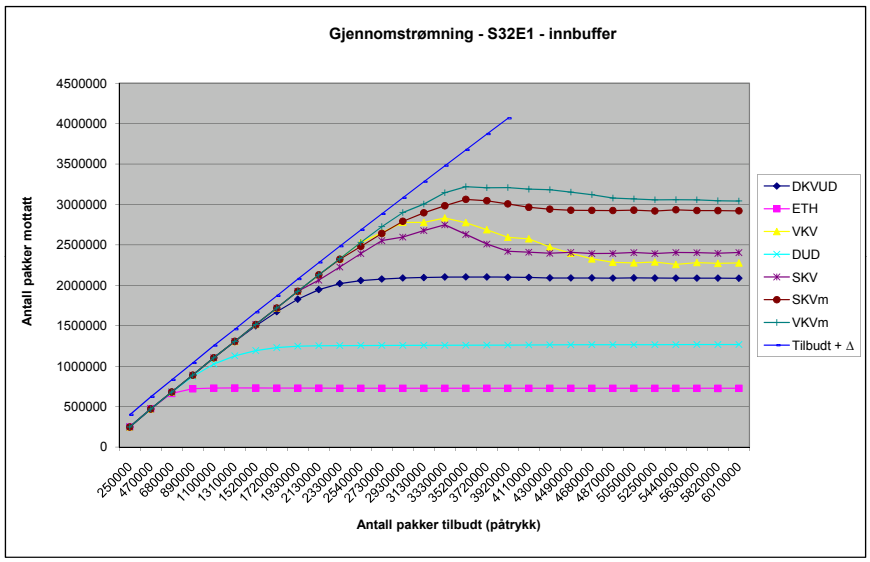
Figur A.12: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E1. Svitsjene benytter inn- og utbuffer som er like store. Plott 3 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.



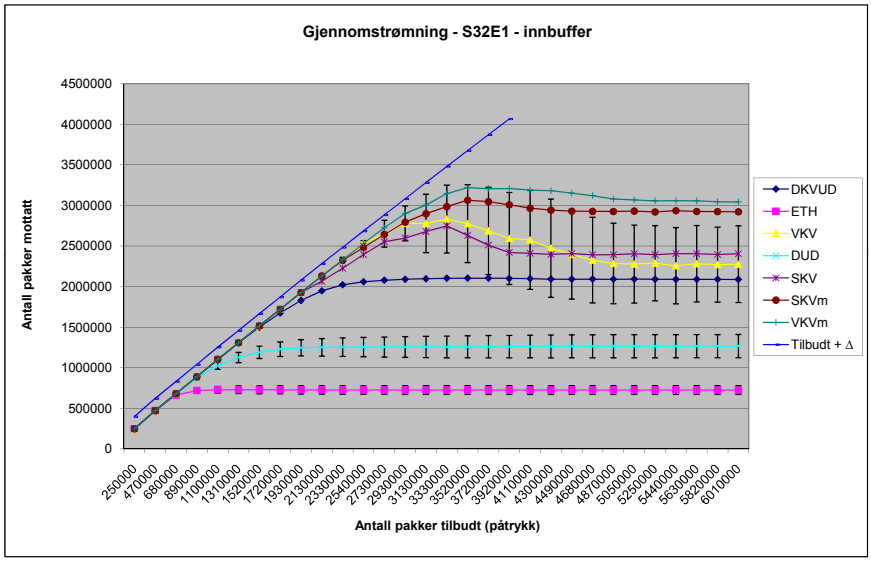
Figur A.13: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S8E1. Svitsjene har størst innbuffer.



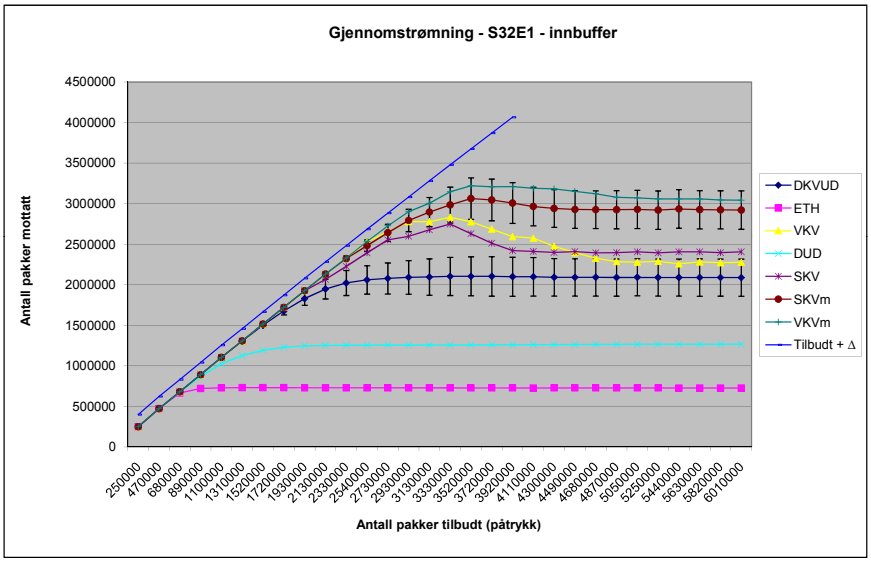
Figur A.14: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S16E1. Svitsjene har størst innbuffer.



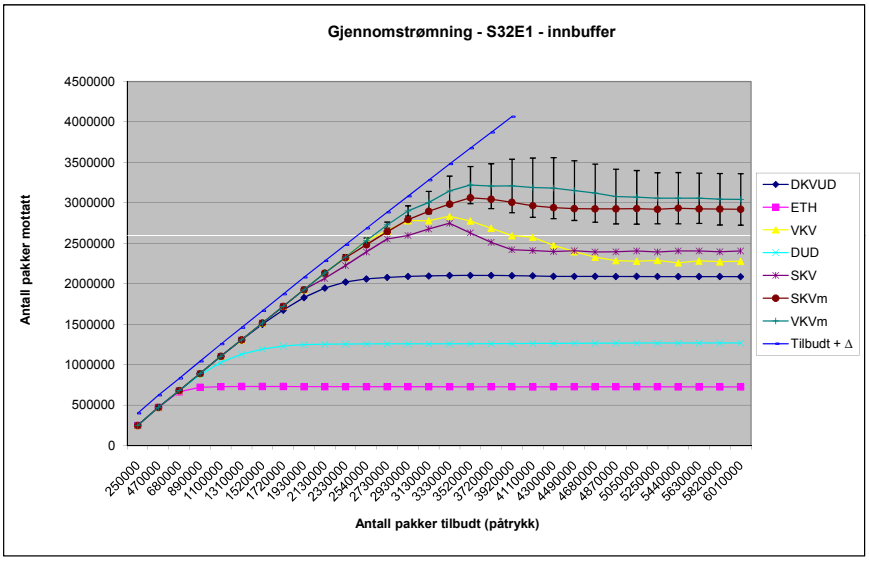
Figur A.15: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E1. Svitsjene har størst innbuffer. Plott uten konfidensintervaller.



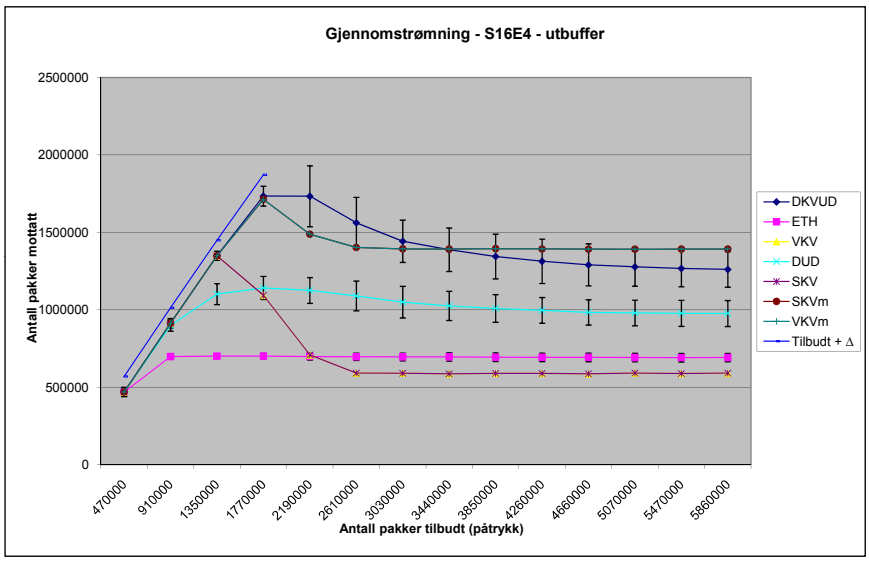
Figur A.16: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E1. Svitsjene har størst innbuffer. Plott 1 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.



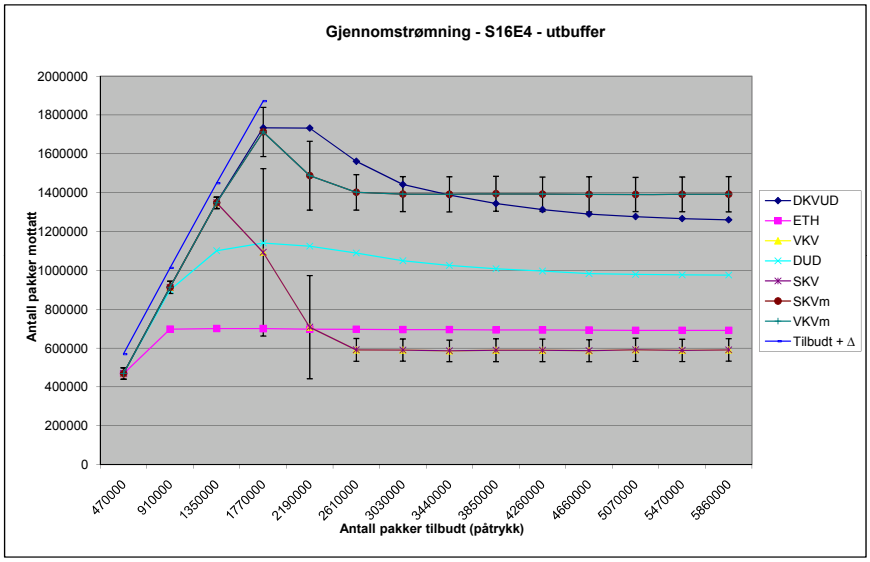
Figur A.17: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E1. Svitsjene har størst innbuffer. Plott 2 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.



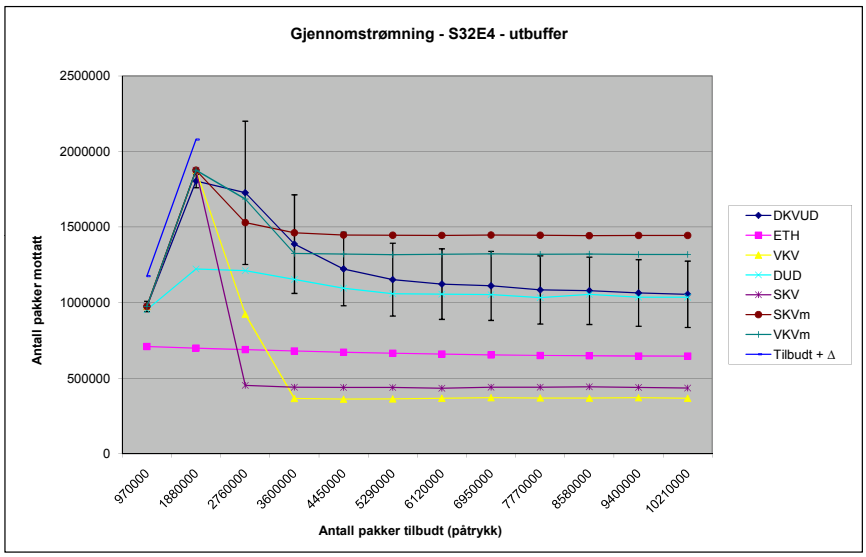
Figur A.18: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E1. Svitsjene har størst innbuffer. Plott 3 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.



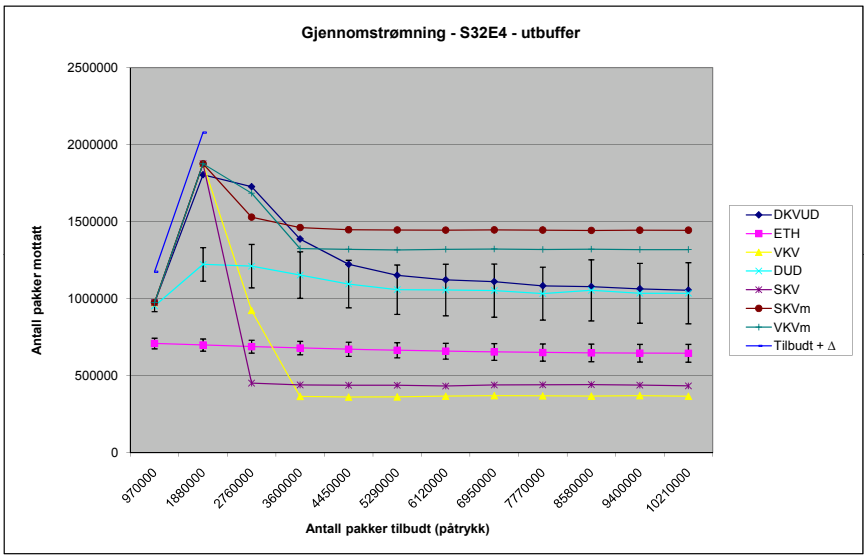
Figur A.19: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S16E4. Svitsjene har størst utbuffer. Plott 1 av 2 for å vise konfidensintervaller for ulike rutealgoritmer.



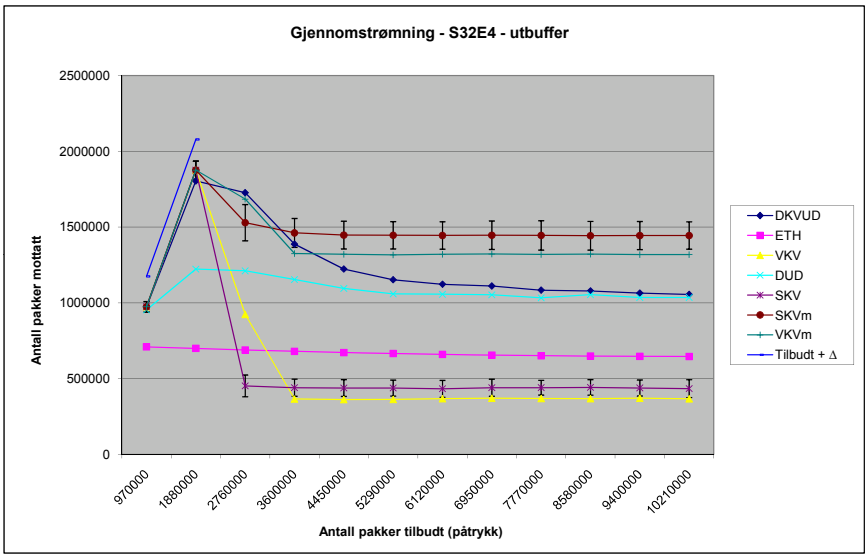
Figur A.20: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S16E4. Svitsjene har størst utbuffer. Plott 2 av 2 for å vise konfidensintervaller for ulike rutealgoritmer.



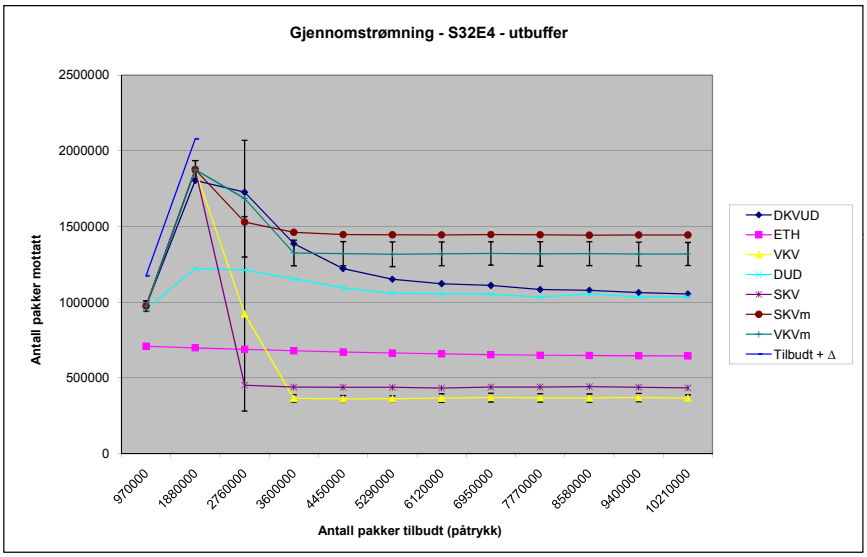
Figur A.21: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E4. Svitsjene har størst utbuffer. Plott 1 av 4 for å vise konfidensintervaller for ulike rutealgoritmer.



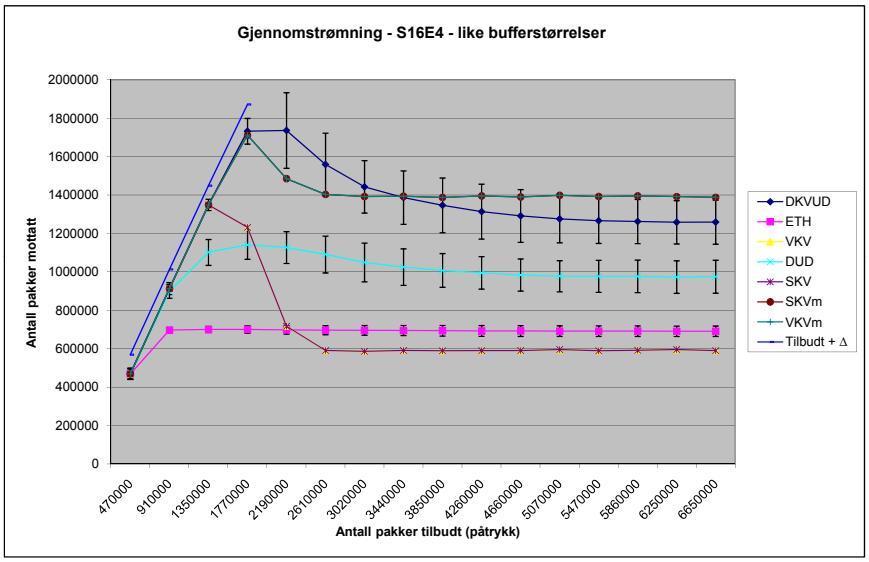
Figur A.22: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E4. Svitsjene har størst utbuffer. Plott 2 av 4 for å vise konfidensintervaller for ulike rutealgoritmer.



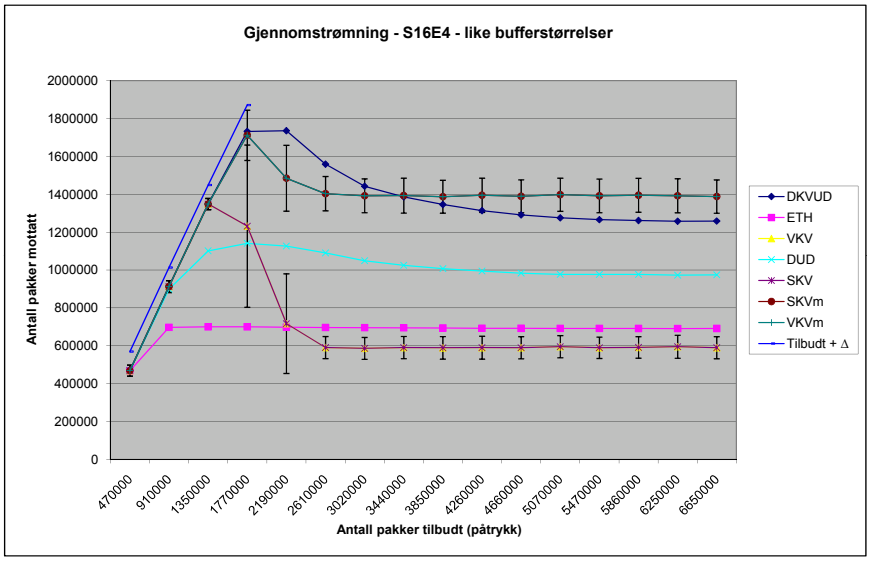
Figur A.23: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E4. Svitsjene har størst utbuffer. Plott 3 av 4 for å vise konfidensintervaller for ulike rutealgoritmer.



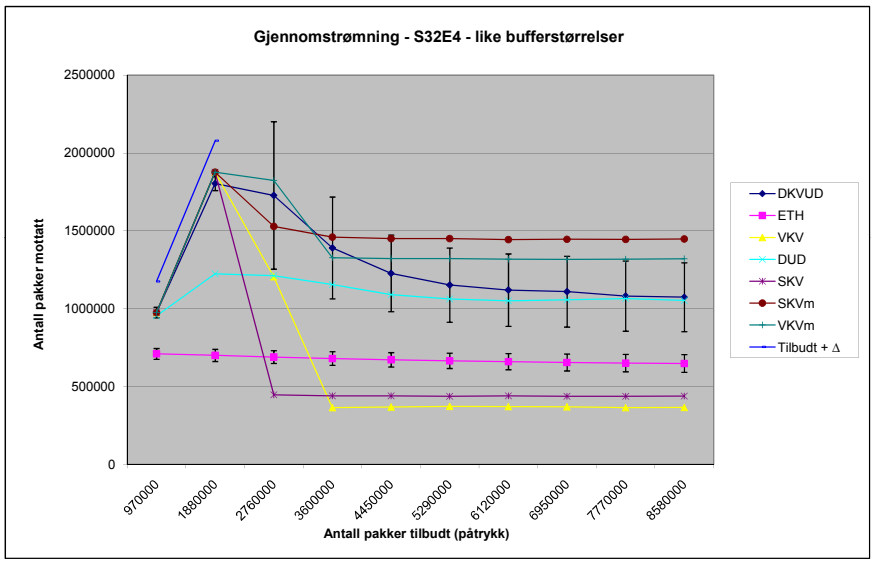
Figur A.24: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E4. Svitsjene har størst utbuffer. Plott 4 av 4 for å vise konfidensintervaller for ulike rutealgoritmer.



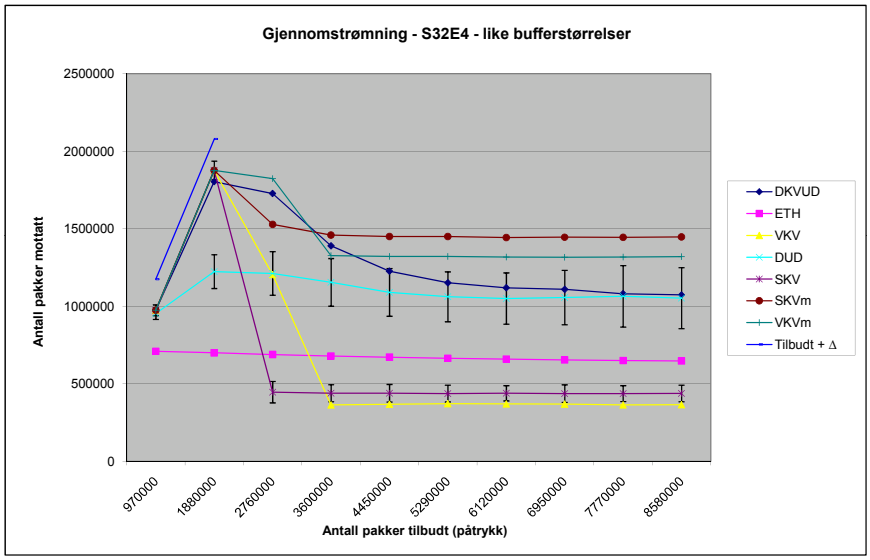
Figur A.25: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S16E4. Svitsjene benytter inn- og utbuffer som er like store. Plott 1 av 2 for å vise konfidensintervaller for ulike rutealgoritmer.



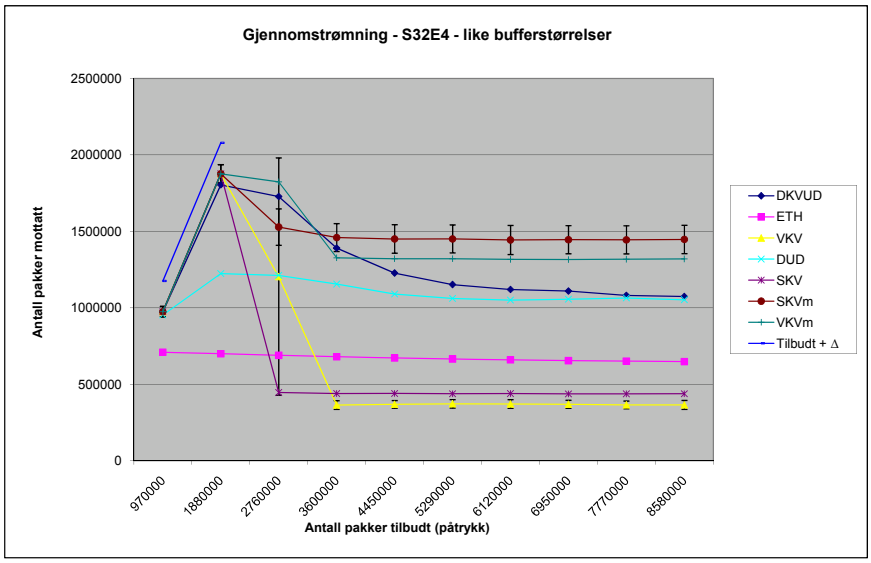
Figur A.26: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S16E4. Svitsjene benytter inn- og utbuffer som er like store. Plott 2 av 2 for å vise konfidensintervaller for ulike rutealgoritmer.



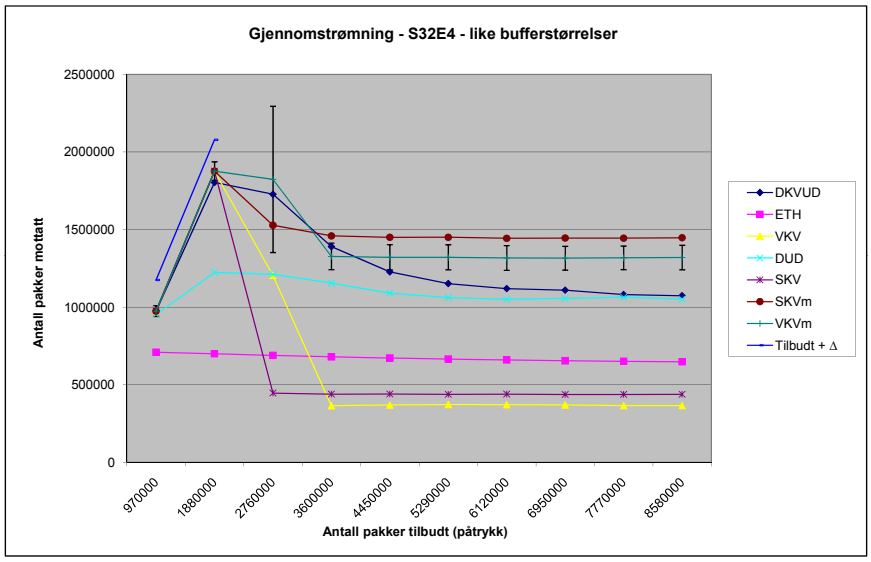
Figur A.27: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E4. Svitsjene benytter inn- og utbuffer som er like store. Plott 1 av 4 for å vise konfidensintervaller for ulike rutealgoritmer.



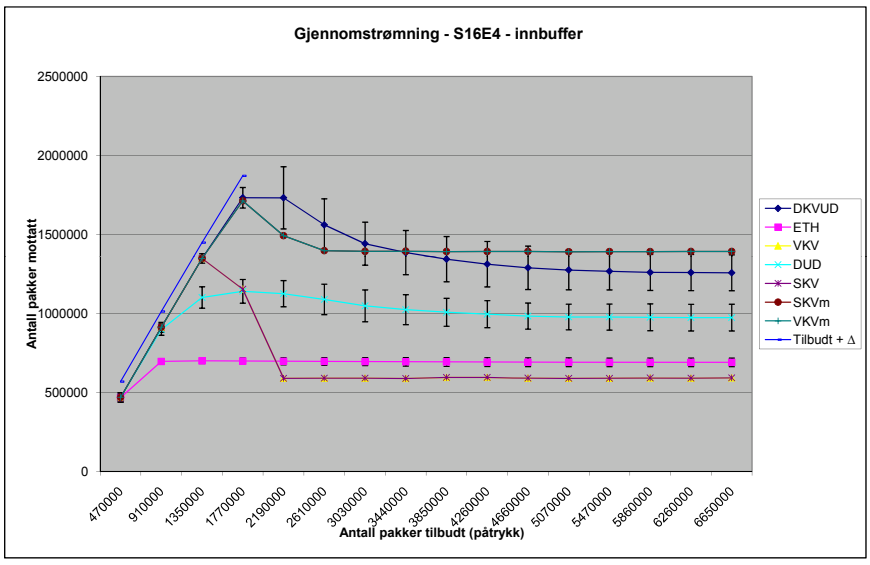
Figur A.28: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E4. Svitsjene benytter inn- og utbuffer som er like store. Plott 2 av 4 for å vise konfidensintervaller for ulike rutealgoritmer.



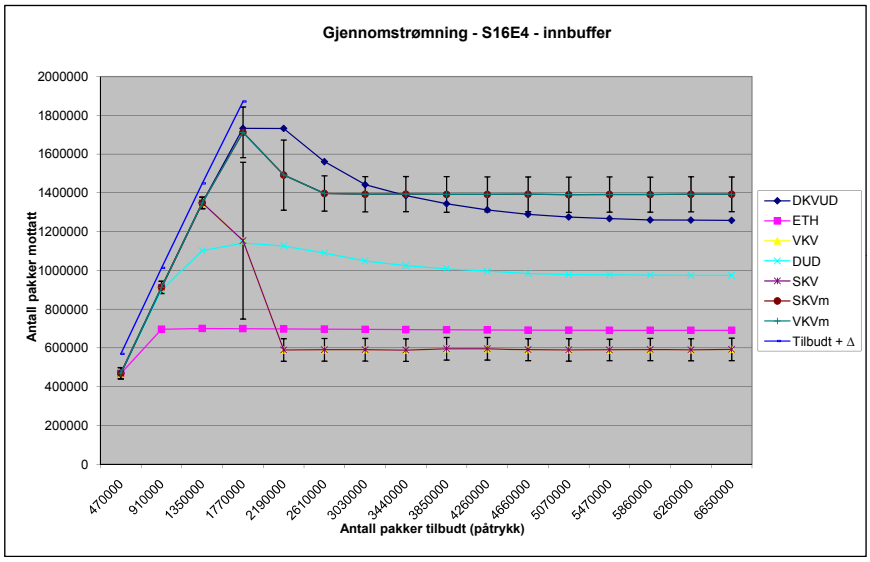
Figur A.29: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E4. Svitsjene benytter inn- og utbuffer som er like store. Plott 3 av 4 for å vise konfidensintervaller for ulike rutealgoritmer.



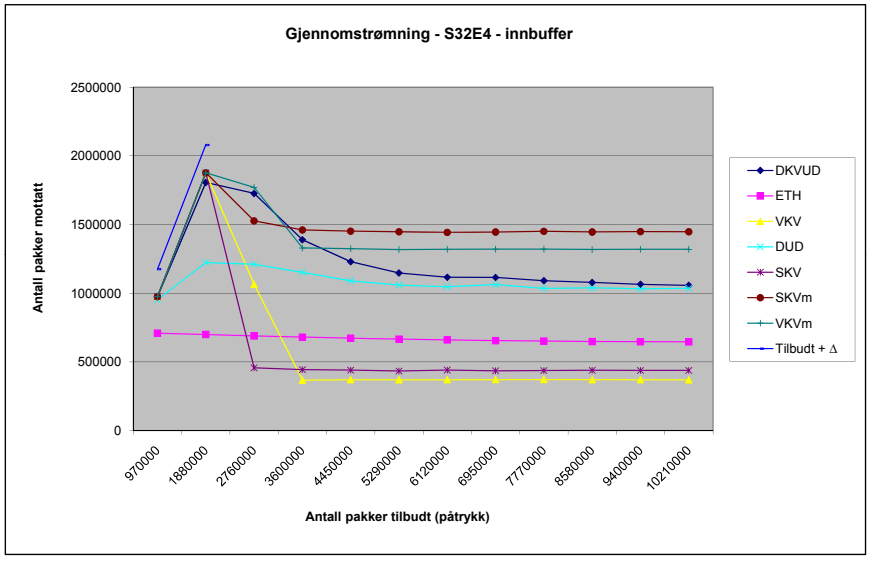
Figur A.30: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E4. Svitsjene benytter inn- og utbuffer som er like store. Plott 4 av 4 for å vise konfidensintervaller for ulike rutealgoritmer.



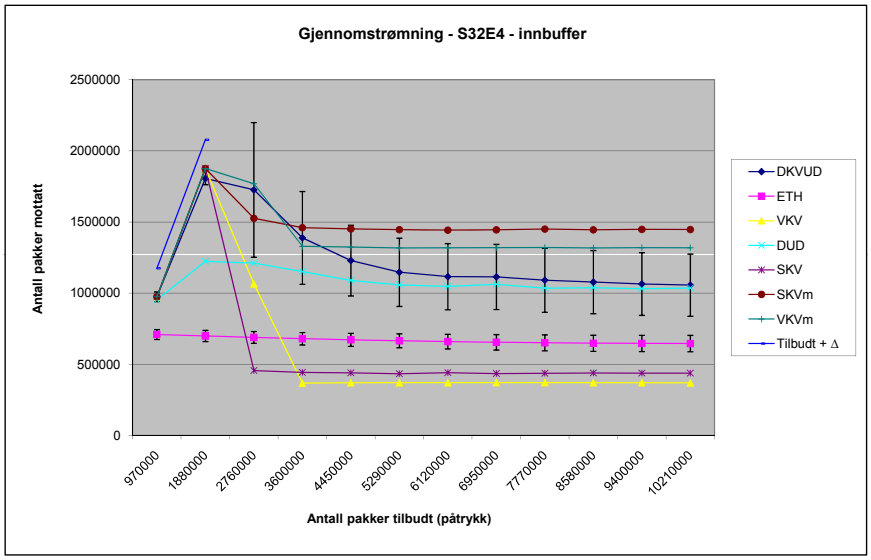
Figur A.31: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S16E4. Svitsjene har størst innbuffer. Plott 1 av 2 for å vise konfidensintervaller for ulike rutealgoritmer.



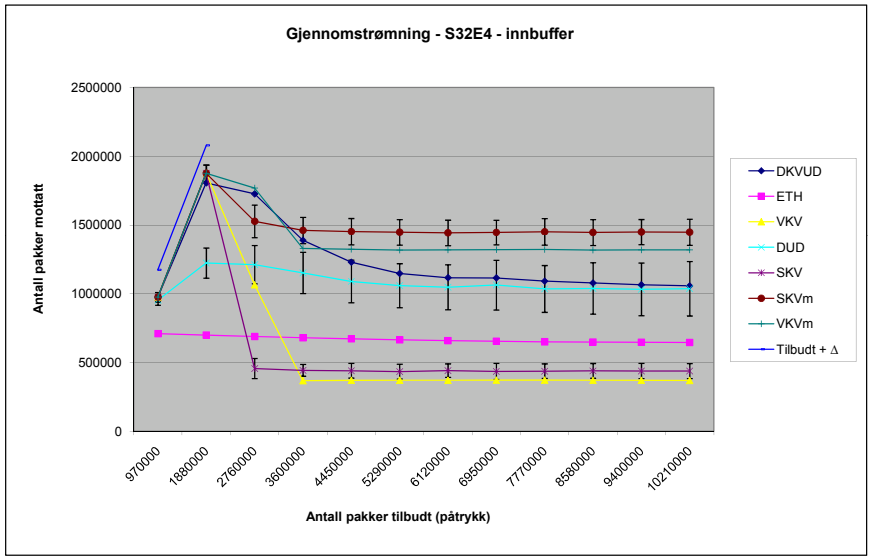
Figur A.32: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S16E4. Svitsjene har størst innbuffer. Plott 2 av 2 for å vise konfidensintervaller for ulike rutealgoritmer.



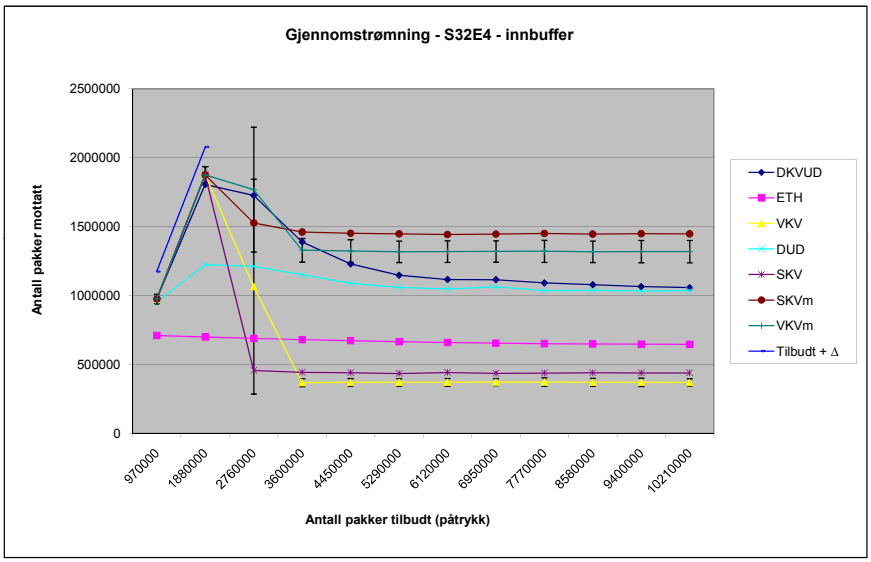
Figur A.33: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E4. Svitsjene har størst innbuffer. Plott uten konfidensintervaller.



Figur A.34: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E4. Svitsjene har størst innbuffer. Plott 1 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.

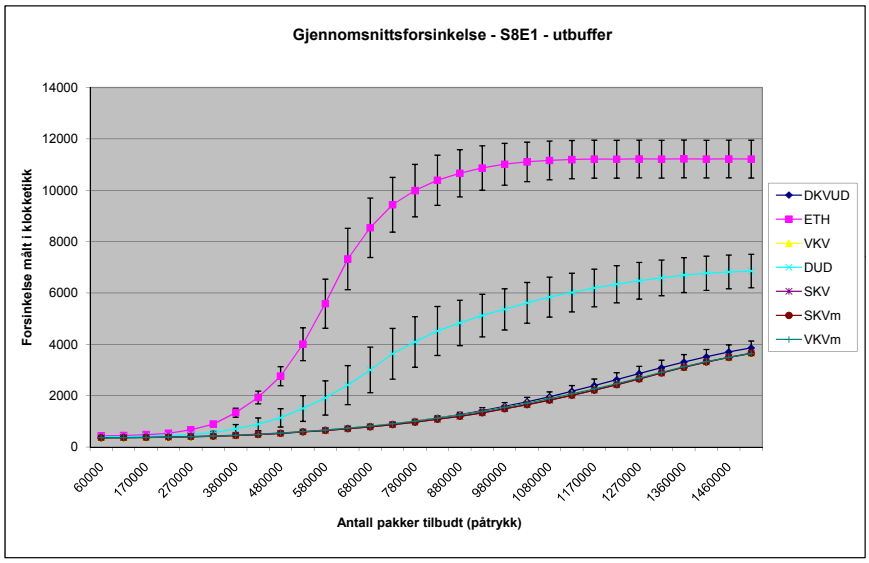


Figur A.35: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E4. Svitsjene har størst innbuffer. Plott 2 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.

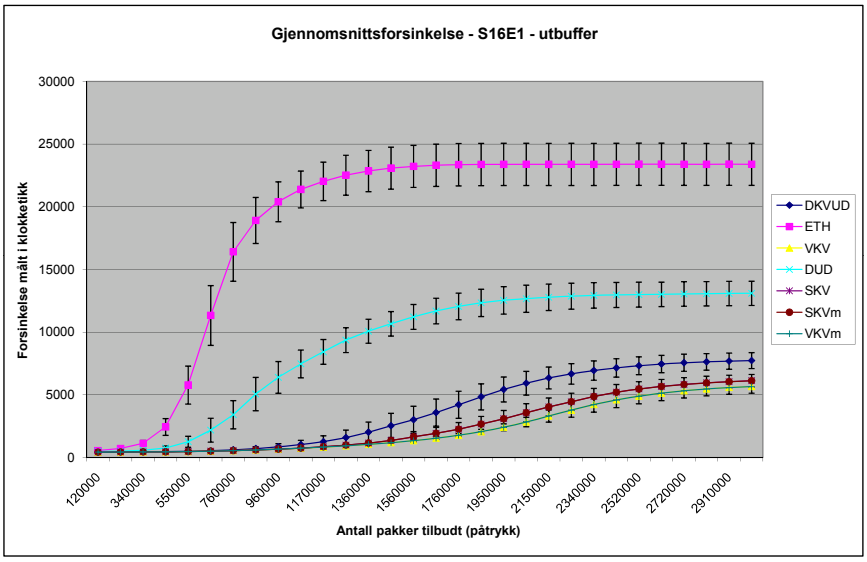


Figur A.36: Gjennomstrømning ved bruk av ulike rutealgoritmer. Topologi S32E4. Svitsjene har størst innbuffer. Plott 3 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.

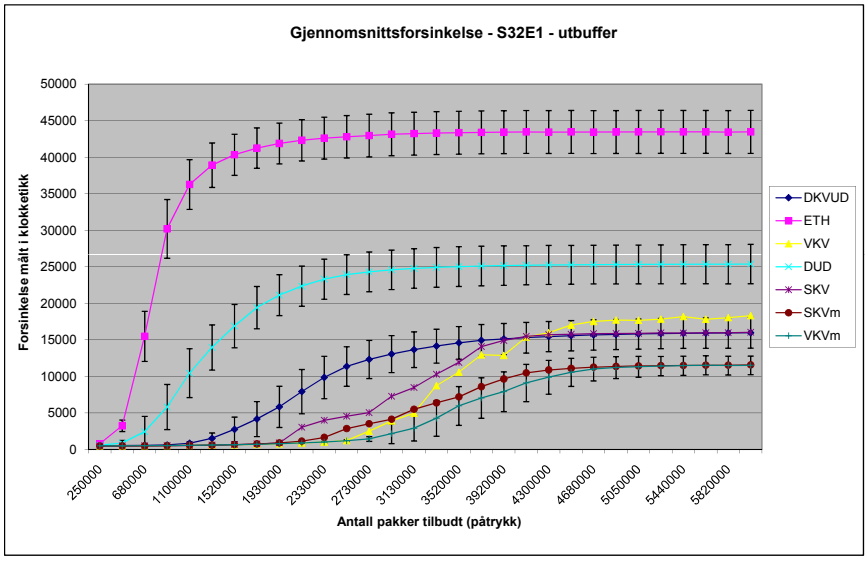
Gjennomsnittsforsinkelse



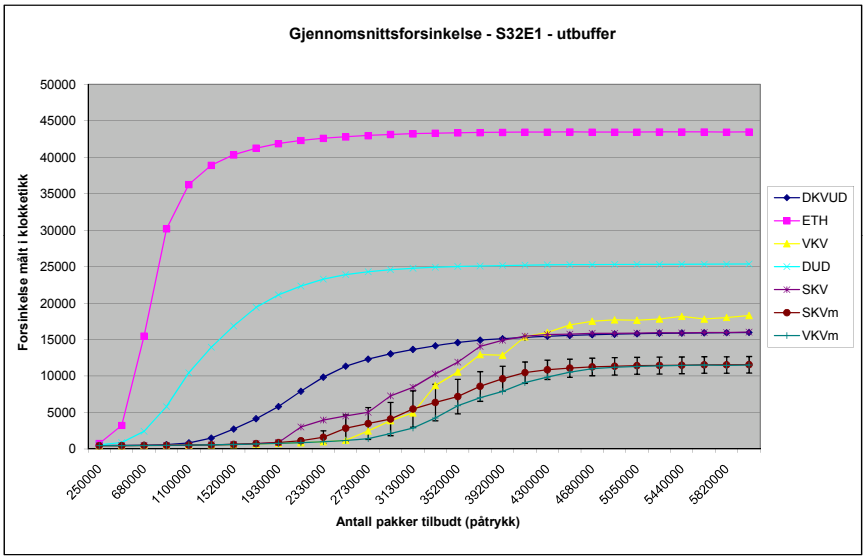
Figur A.37: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S8E1. Svitsjene har størst utbuffer.



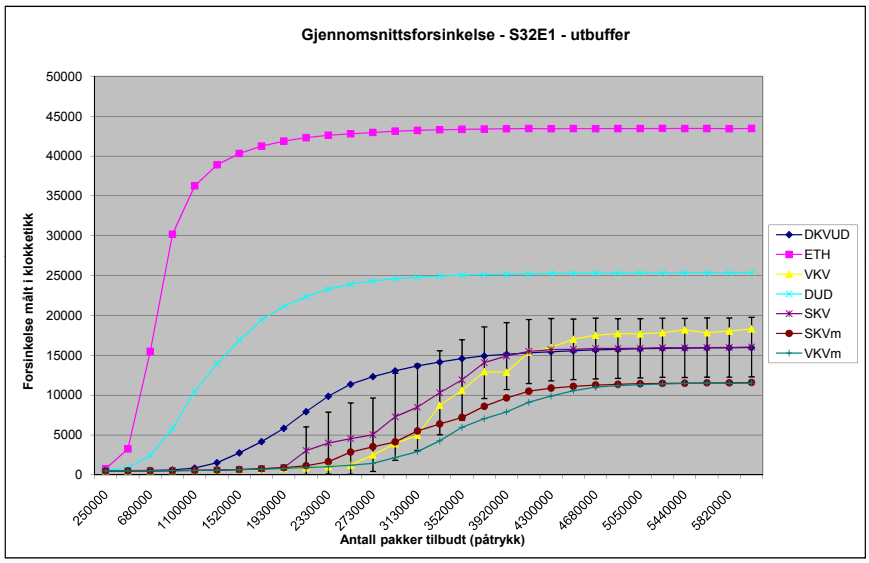
Figur A.38: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S16E1. Svitsjene har størst utbuffer.



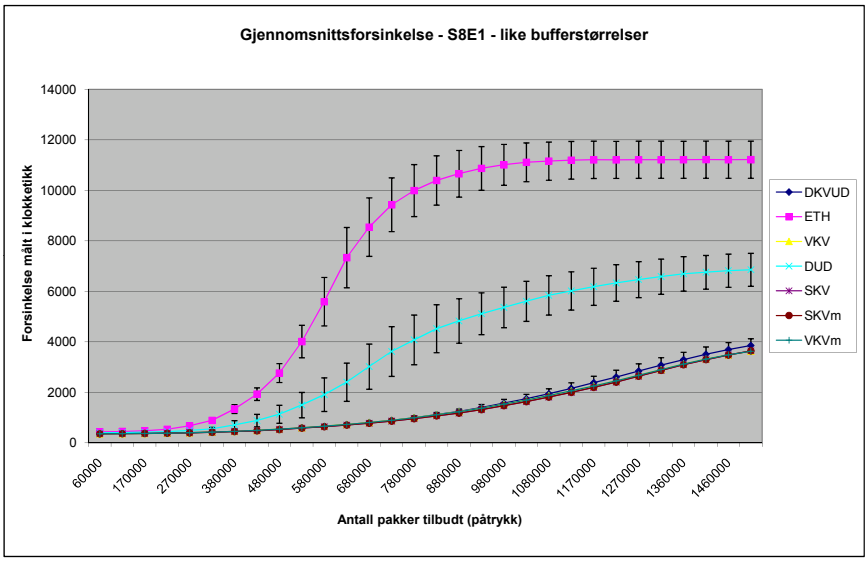
Figur A.39: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S32E1. Svitsjene har størst utbuffer. Plott 1 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.



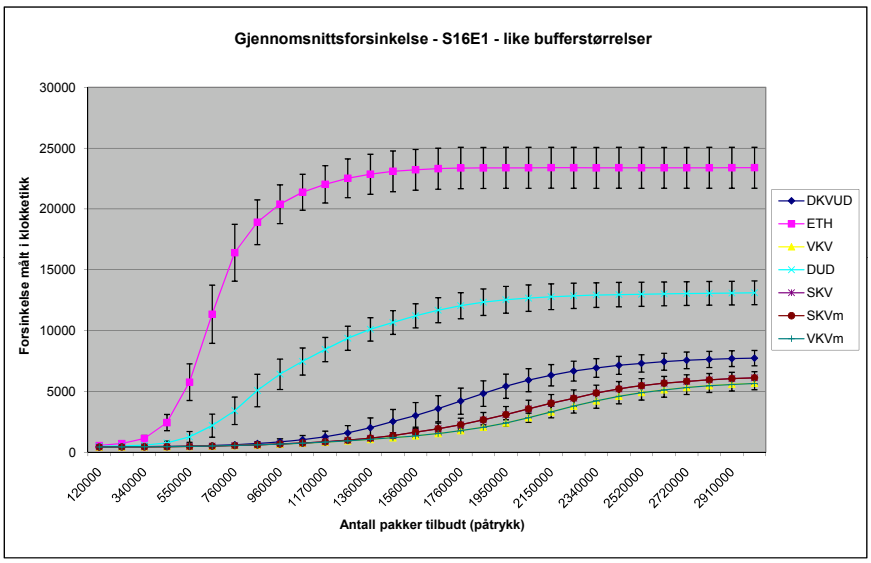
Figur A.40: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S32E1. Svitsjene har størst utbuffer. Plott 2 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.



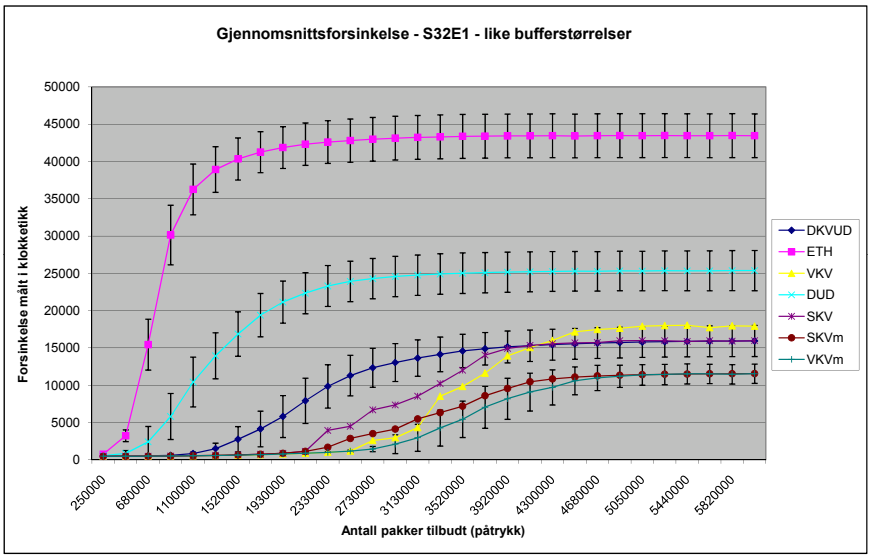
Figur A.41: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S32E1. Svitsjene har størst utbuffer. Plott 3 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.



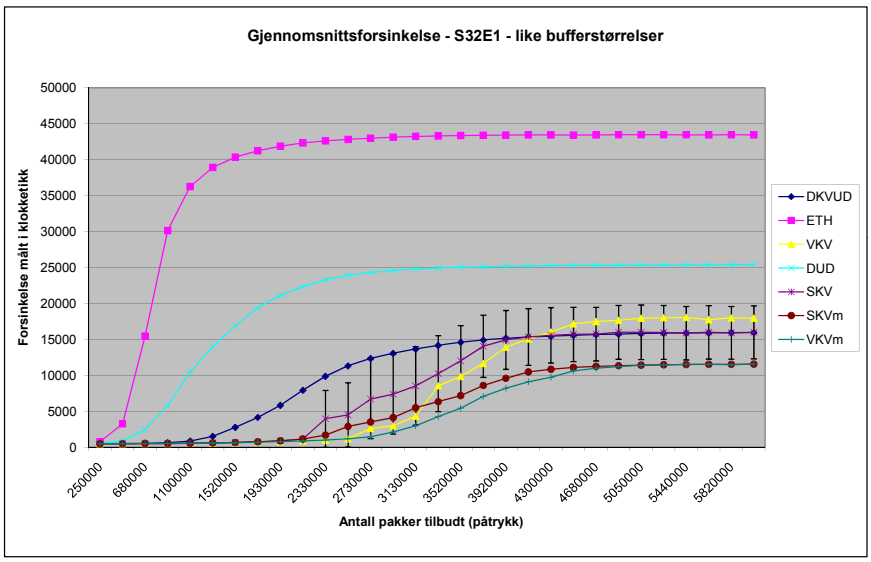
Figur A.42: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S8E1. Svitsjene benytter inn- og utbuffer som er like store.



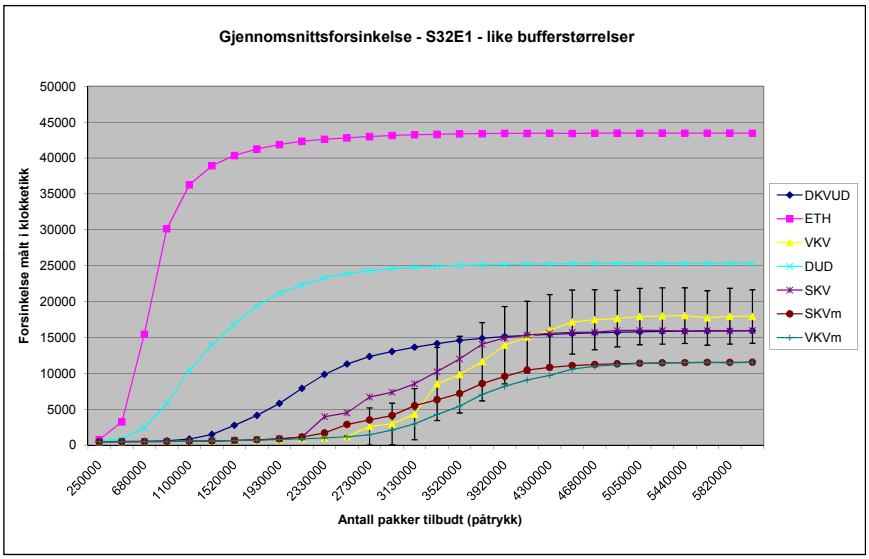
Figur A.43: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S16E1. Svitsjene benytter inn- og utbuffer som er like store.



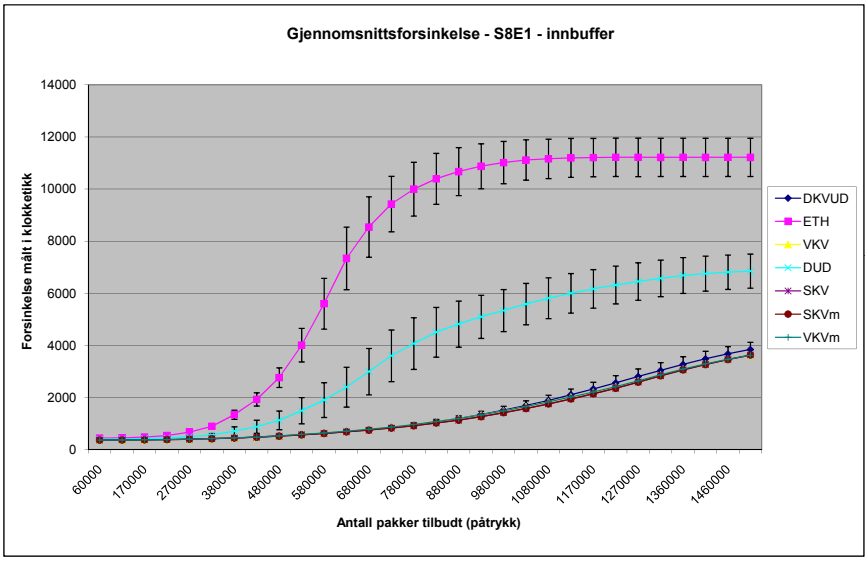
Figur A.44: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S32E1. Svitsjene benytter inn- og utbuffer som er like store. Plott 1 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.



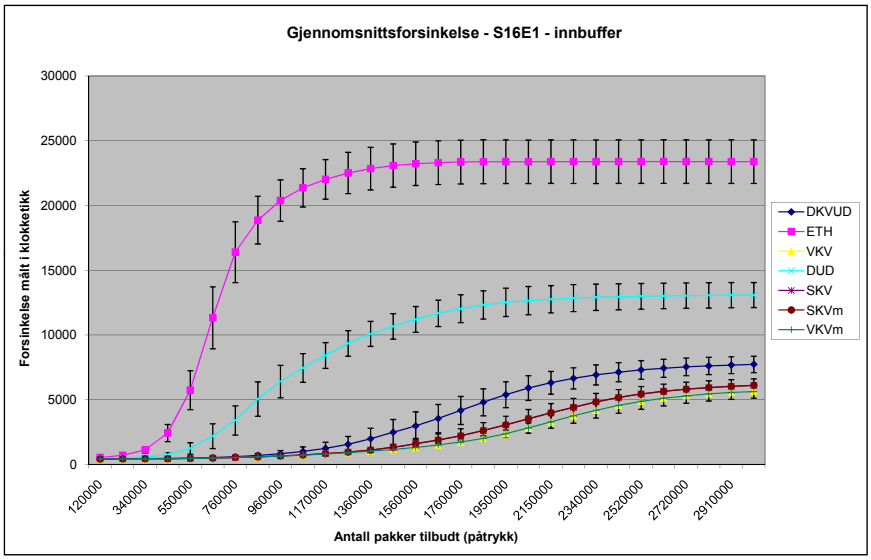
Figur A.45: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S32E1. Svitsjene benytter inn- og utbuffer som er like store. Plott 2 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.



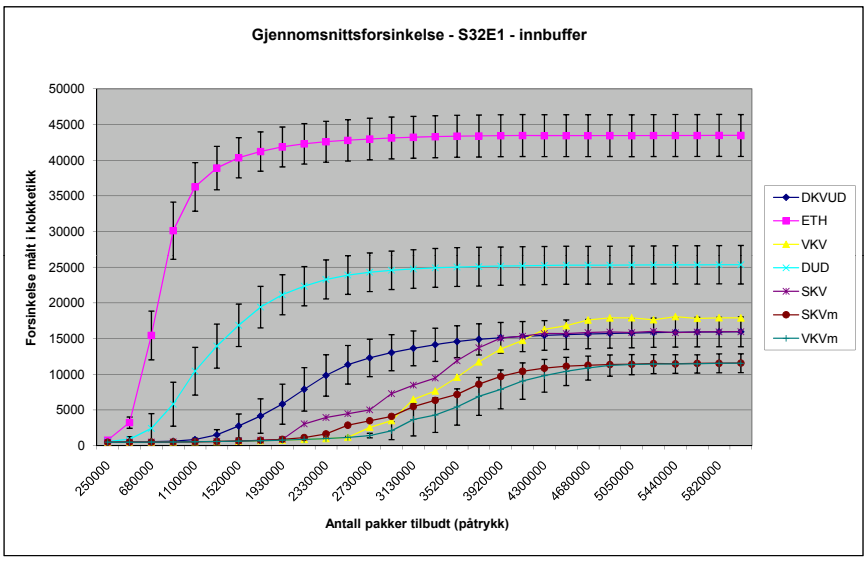
Figur A.46: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S32E1. Svitsjene benytter inn- og utbuffer som er like store. Plott 3 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.



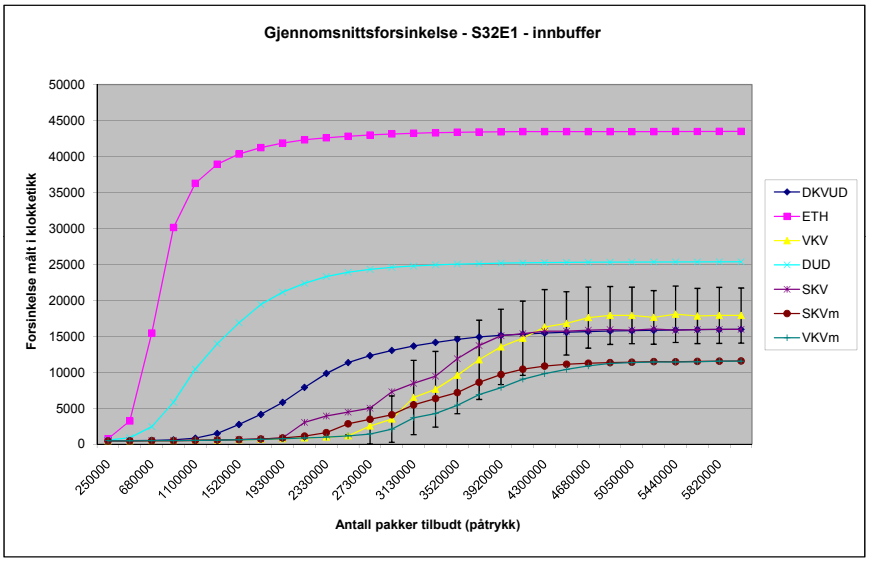
Figur A.47: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S8E1. Svitsjene har størst innbuffer.



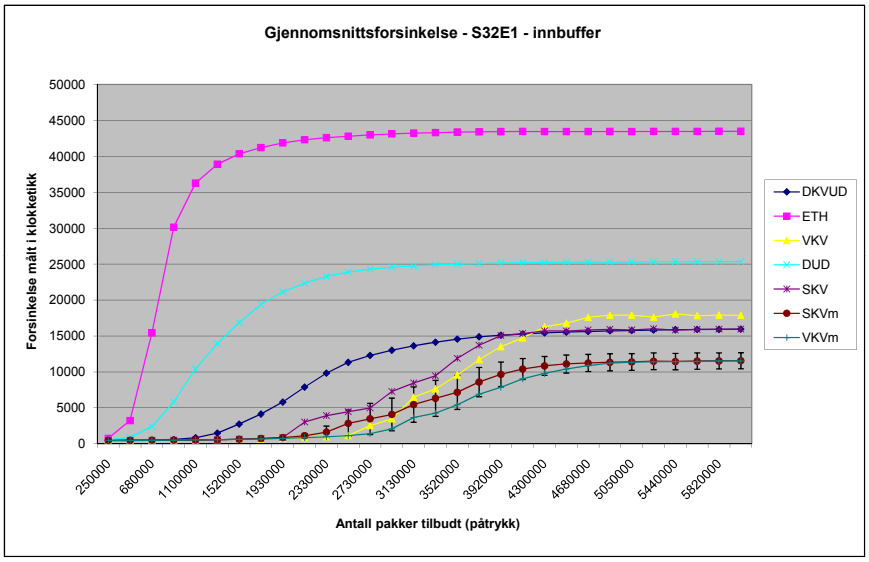
Figur A.48: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S16E1. Svitsjene har størst innbuffer.



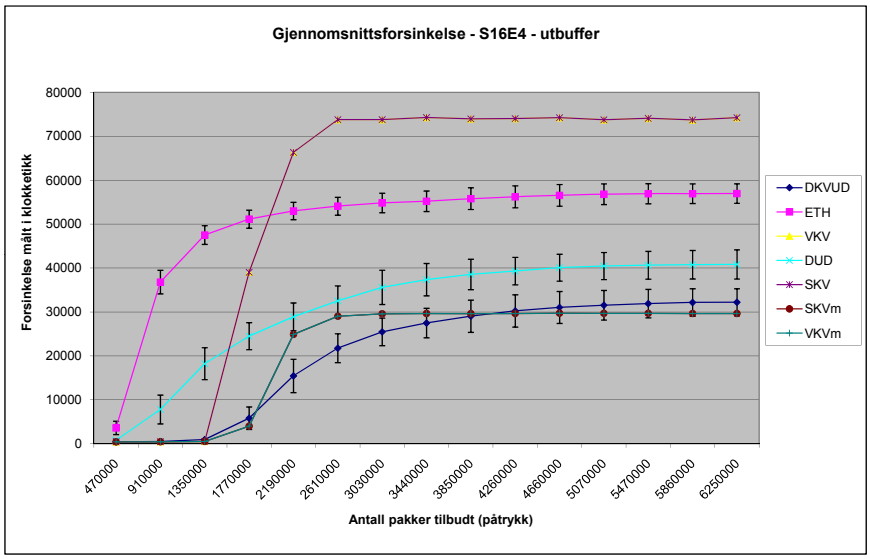
Figur A.49: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S32E1. Svitsjene har størst innbuffer. Plott 1 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.



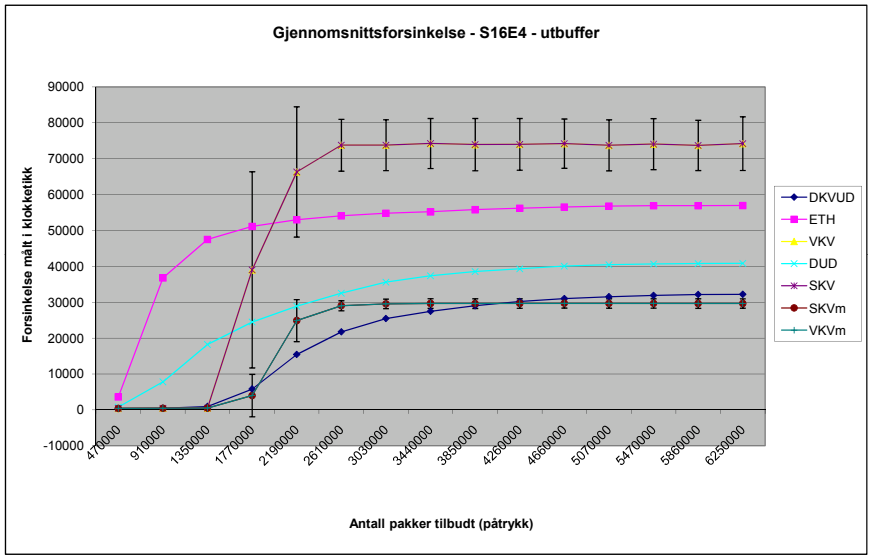
Figur A.50: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S32E1. Svitsjene har størst innbuffer. Plott 2 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.



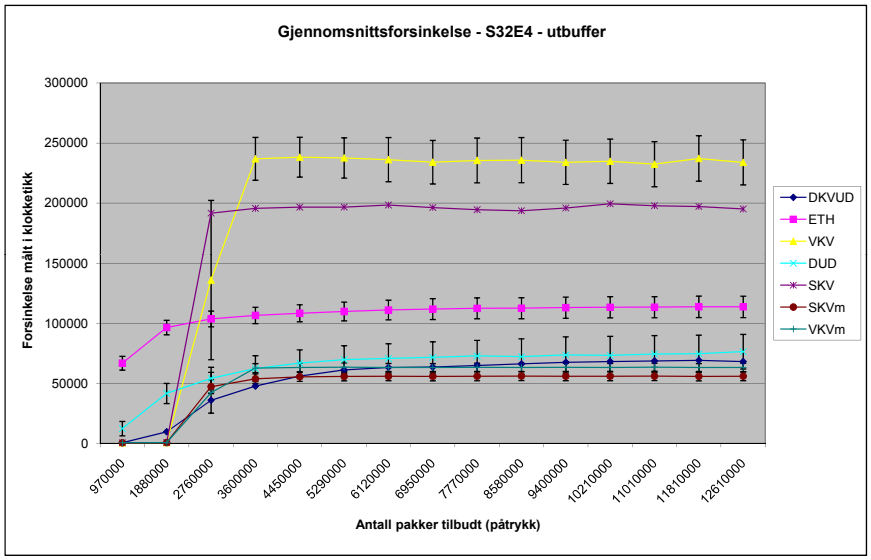
Figur A.51: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S32E1. Svitsjene har størst innbuffer. Plott 3 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.



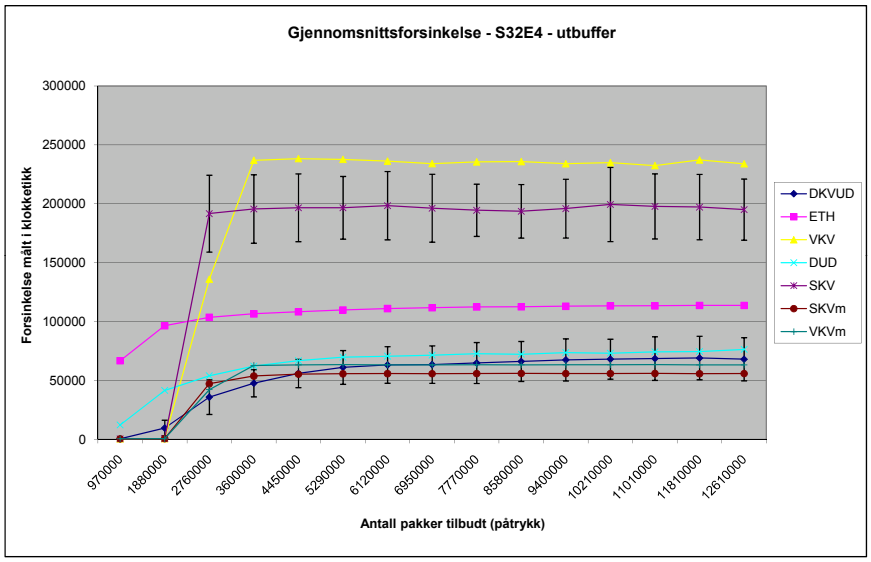
Figur A.52: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S16E4. Svitsjene har størst utbuffer. Plott 1 av 2 for å vise konfidensintervaller for ulike rutealgoritmer.



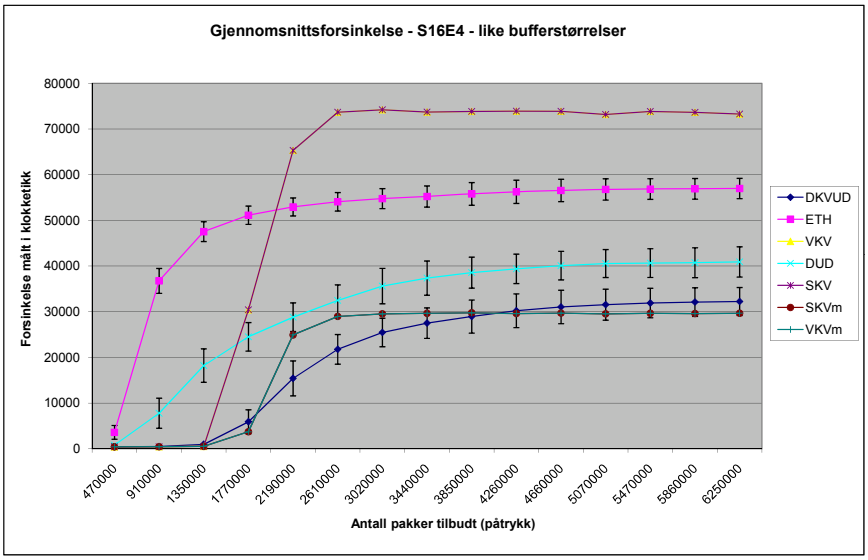
Figur A.53: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S16E4. Svitsjene har størst utbuffer. Plott 2 av 2 for å vise konfidensintervaller for ulike rutealgoritmer.



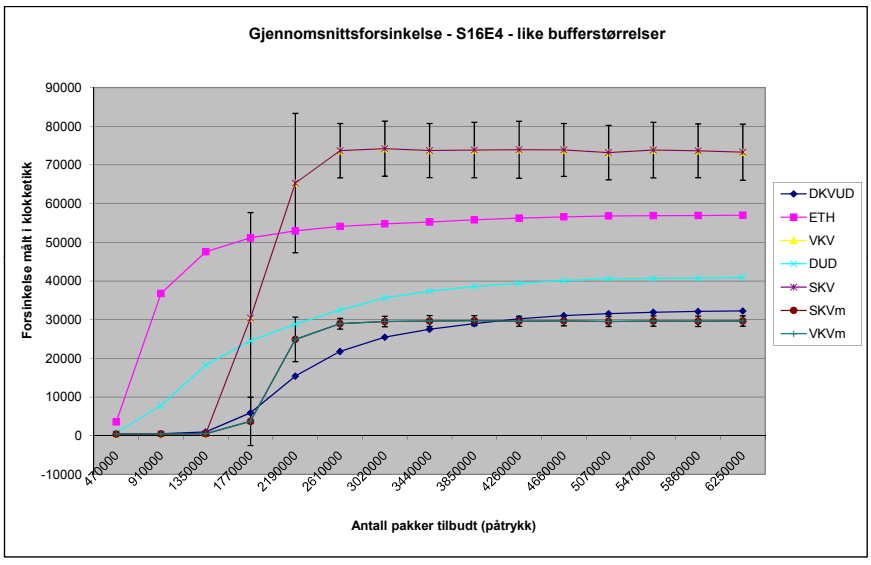
Figur A.54: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S32E4. Svitsjene har størst utbuffer. Plott 1 av 2 for å vise konfidensintervaller for ulike rutealgoritmer.



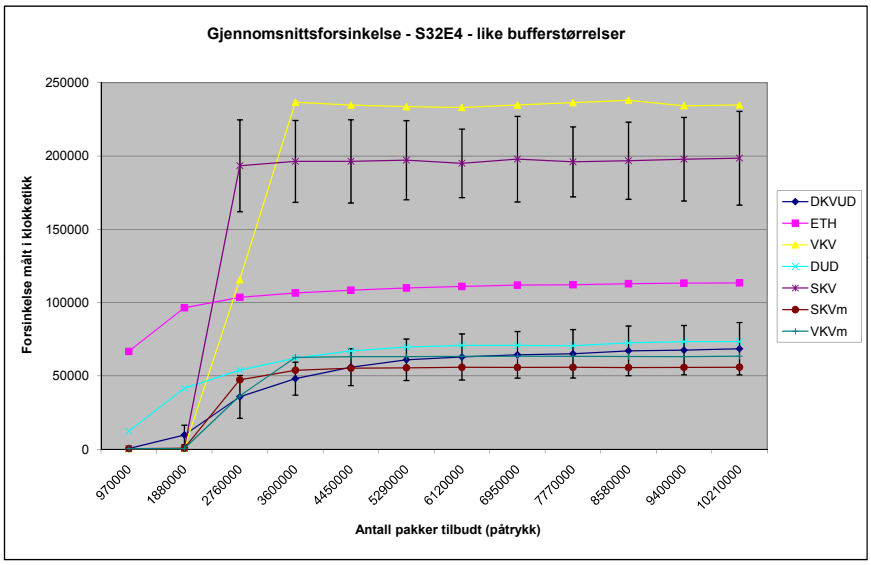
Figur A.55: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S32E4. Svitsjene har størst utbuffer. Plott 2 av 2 for å vise konfidensintervaller for ulike rutealgoritmer.



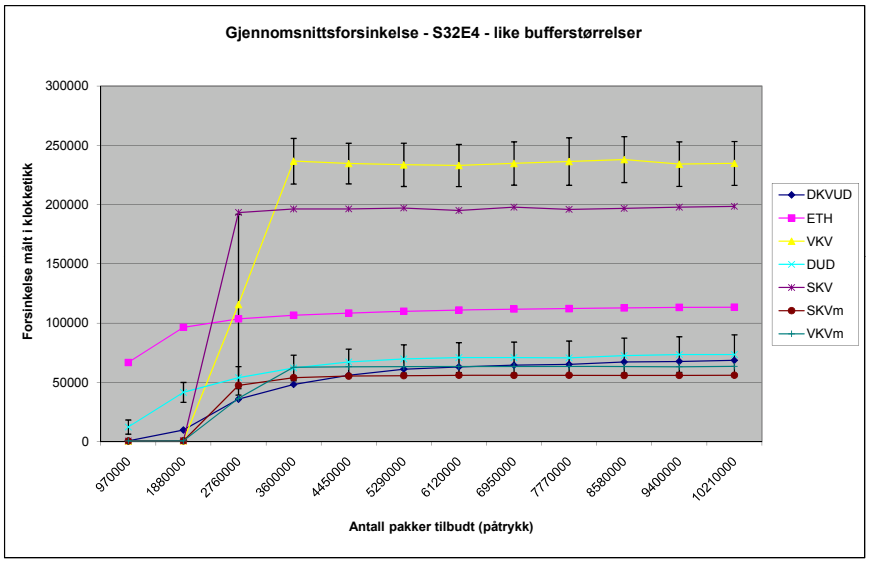
Figur A.56: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S16E4. Svitsjene benytter inn- og utbuffer som er like store. Plott 1 av 2 for å vise konfidensintervaller for ulike rutealgoritmer.



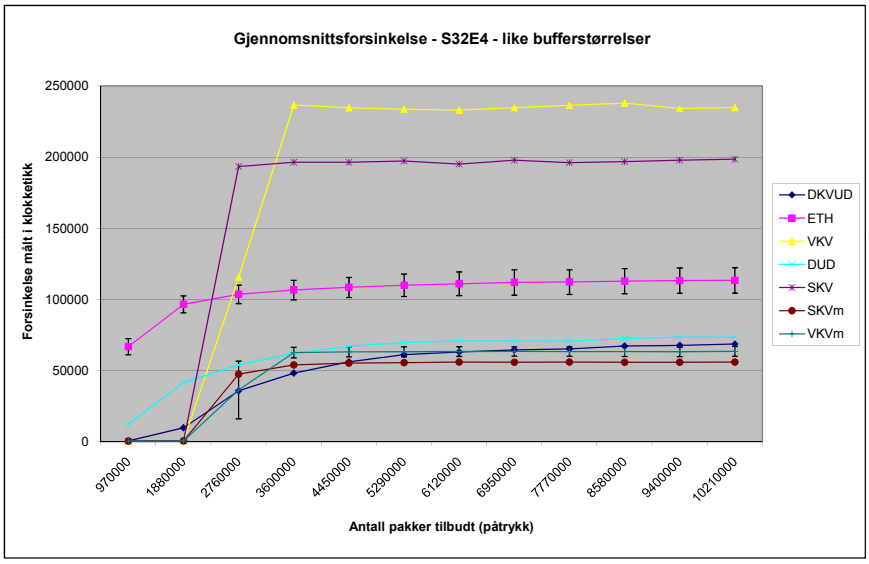
Figur A.57: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S16E4. Svitsjene benytter inn- og utbuffer som er like store. Plott 2 av 2 for å vise konfidensintervaller for ulike rutealgoritmer.



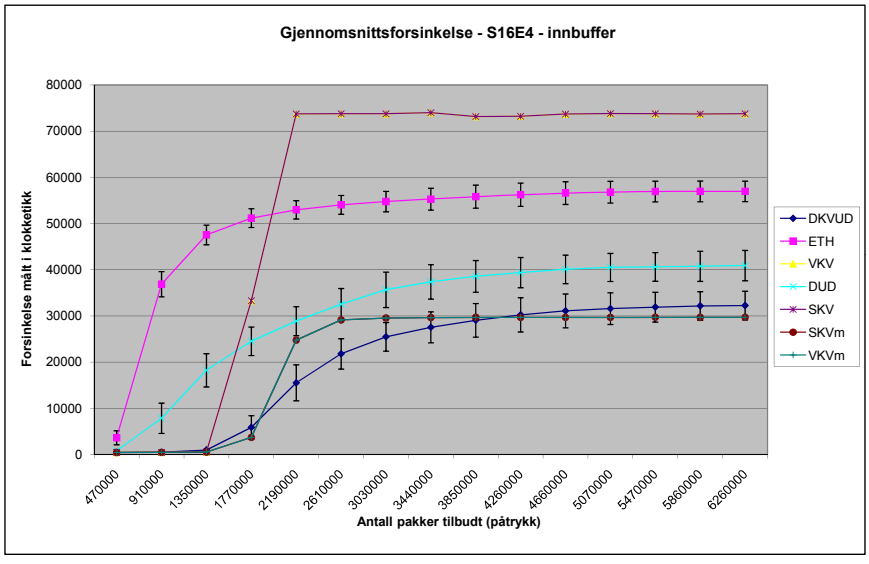
Figur A.58: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S32E4. Svitsjene benytter inn- og utbuffer som er like store. Plott 1 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.



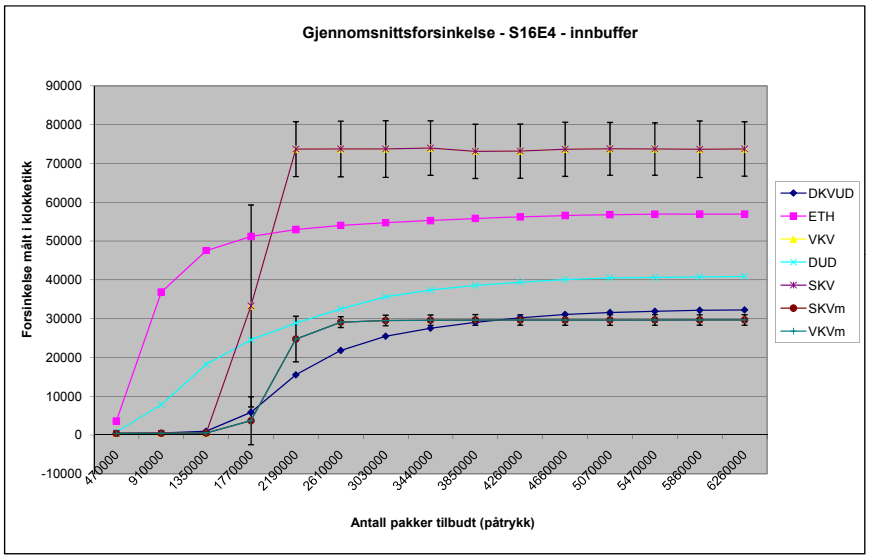
Figur A.59: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S32E4. Svitsjene benytter inn- og utbuffer som er like store. Plott 2 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.



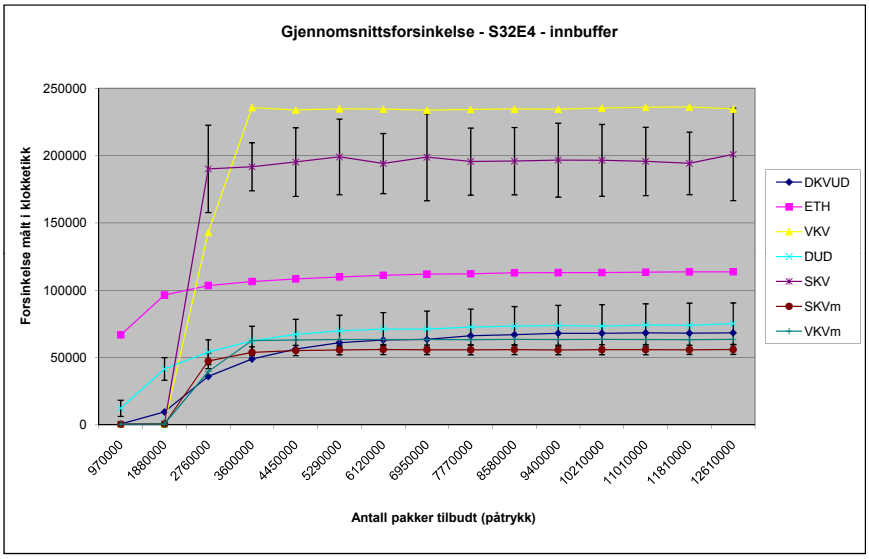
Figur A.60: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S32E4. Svitsjene benytter inn- og utbuffer som er like store. Plott 3 av 3 for å vise konfidensintervaller for ulike rutealgoritmer.



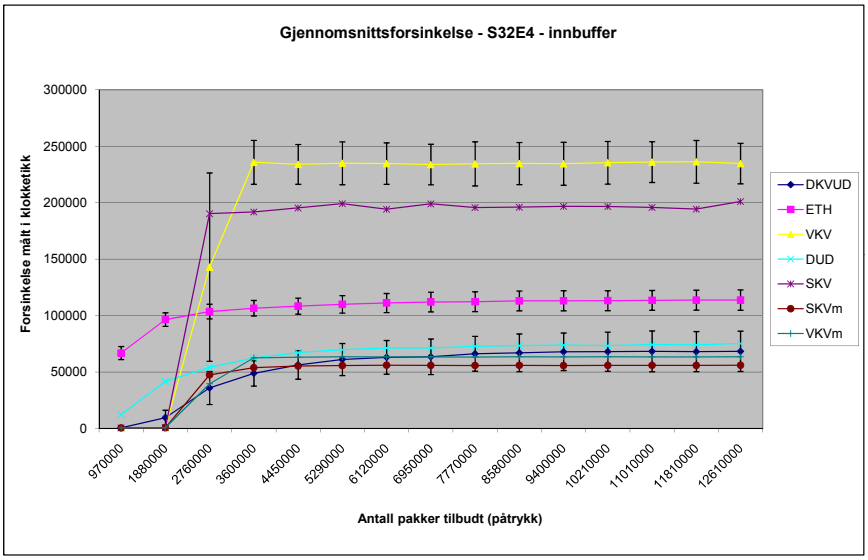
Figur A.61: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S16E4. Svitsjene benytter innbuffer. Plott 1 av 2 for å vise konfidensintervaller for ulike rutealgoritmer.



Figur A.62: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S16E4. Svitsjene benytter innbuffer. Plott 2 av 2 for å vise konfidensintervaller for ulike rutealgoritmer.



Figur A.63: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S32E4. Svitsjene benytter innbuffer. Plott 1 av 2 for å vise konfidensintervaller for ulike rutealgoritmer.

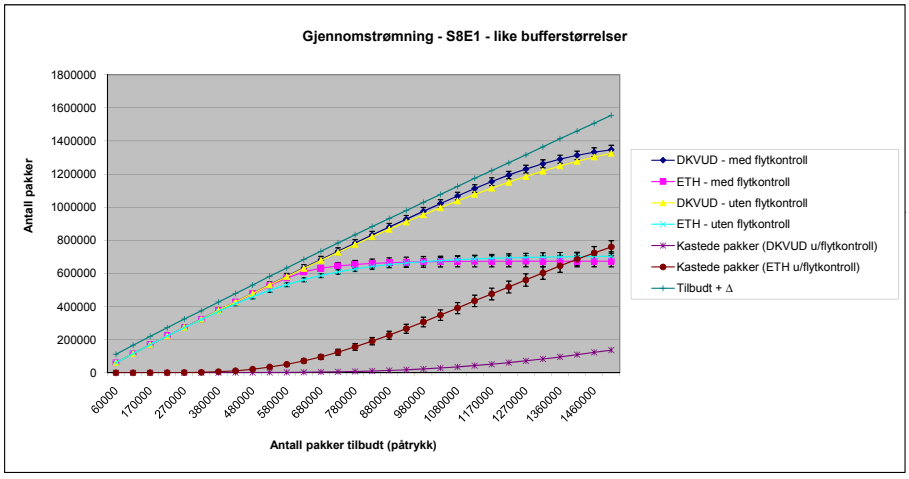


Figur A.64: Gjennomsnittsforsinkelse ved bruk av ulike rutealgoritmer. Topologi S32E4. Svitsjene benytter innbuffer. Plott 2 av 2 for å vise konfidensintervaller for ulike rutealgoritmer.

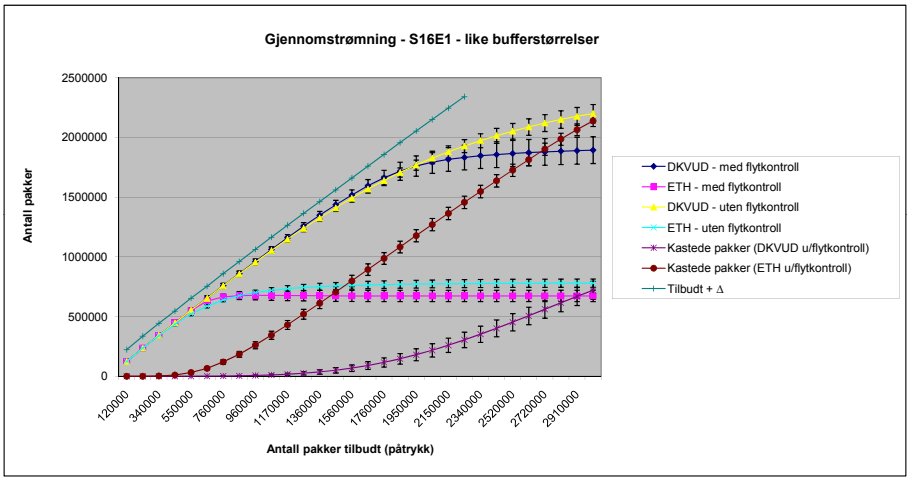
A.2 Simulering av tradisjonelt Ethernet og deterministisk korteste-vei-up*/down*-ruting, med og uten flytkontroll

Dette avsnittet inneholder resultatplott fra simuleringer kjørt for å sammenlikne trafikken i nettverk med og uten flytkontroll. Det er simulert med utgangspunkt i både tradisjonelt Ethernet og Eternet med deterministisk korteste-vei-up*/down*-ruting. Det er i tillegg simulert over tre typer svitsjer: svitsjer med størst innbuffer, svitsjer med størst utbuffer og svitsjer med like store inn- og utbuffer. Siden de tre svitsje-typene gir svært like resulatater har vi valgt å kun vise plott fra simuleringer med svitsjer hvor inn- og utbuffer er like store.

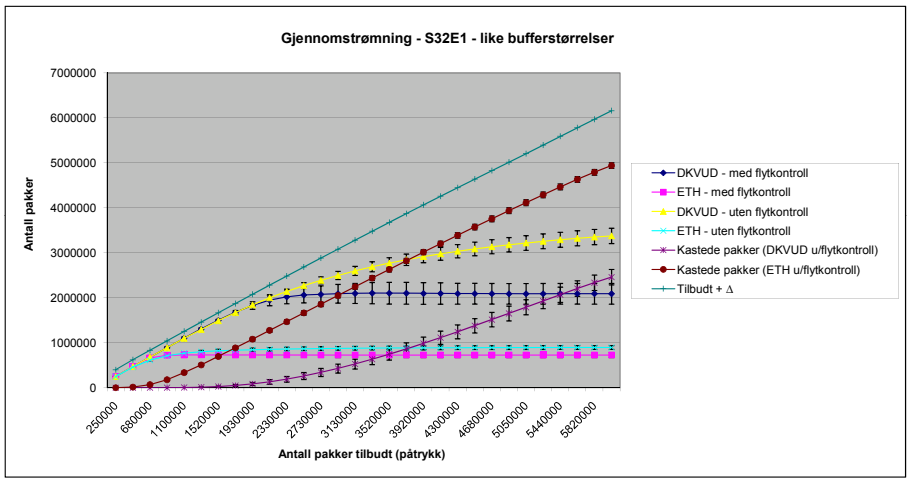
Gjennomstrømning



Figur A.65: Topologi S8E1. Plottet viser gjennomstrømning ved bruk av tradisjonelt Ethernet og Ethernet med deterministisk korteste-vei-up*/down*-ruting, begge benyttet med og uten flytkontroll. Antall kastede pakker ved fravær av flytkontroll er også plottet inn. Svitsjene benytter like store inn- og utbuffer.

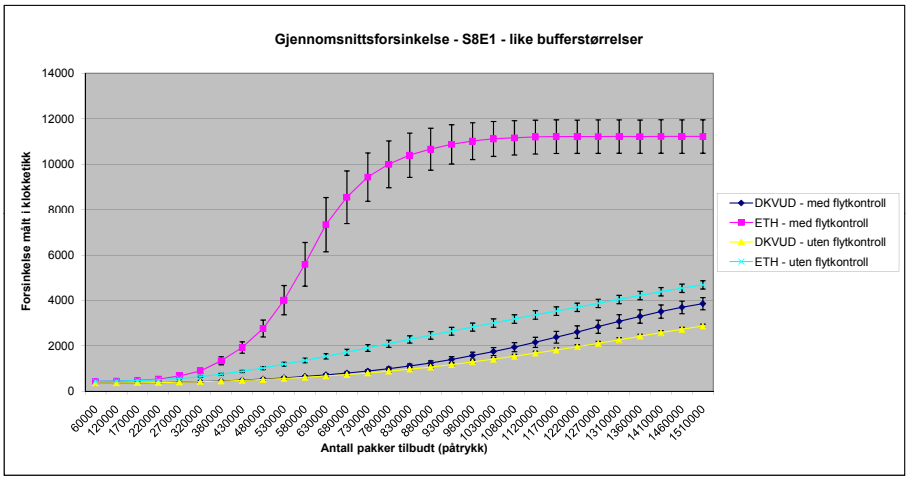


Figur A.66: Topologi S16E1. Plottet viser gjennomstrømning ved bruk av tradisjonelt Ethernet og Ethernet med deterministisk korteste-vei-up*/down*-ruting, begge benyttet med og uten flytkontroll. Antall kastede pakker ved fravær av flytkontroll er også plottet inn. Svitsjene benytter like store inn- og utbuffer.

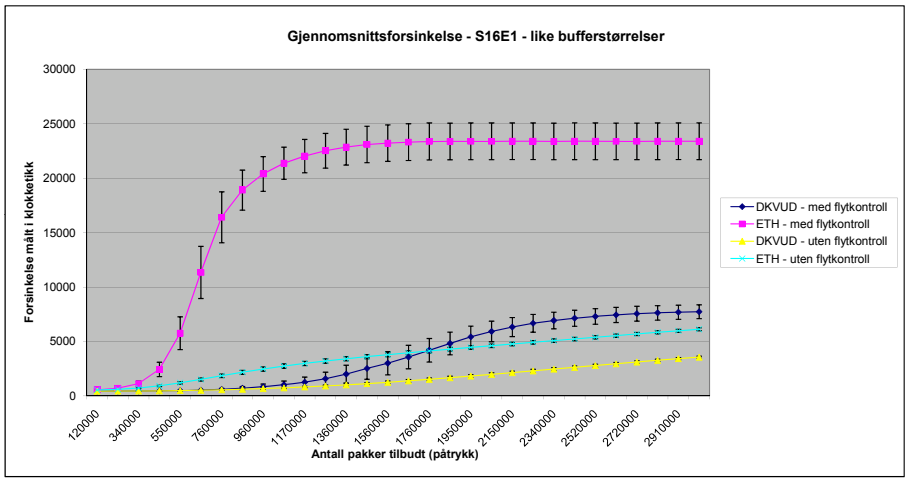


Figur A.67: Topologi S32E1. Plottet viser gjennomstrømning ved bruk av tradisjonelt Ethernet og Ethernet med deterministisk korteste-vei-up*/down*-ruting, begge benyttet med og uten flytkontroll. Antall kastede pakker ved fravær av flytkontroll er også plottet inn. Svitsjene benytter like store inn- og utbuffer.

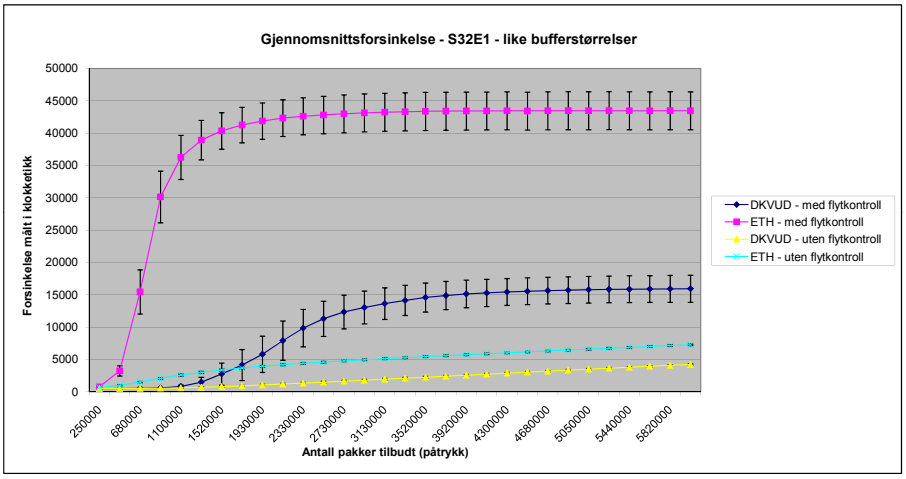
Gjennomsnittsforsinkelse



Figur A.68: Topologi S8E1. Plottet viser gjennomsnittsforsinkelse ved bruk av tradisjonelt Ethernet og Ethernet med deterministisk korteste-vei-up*/down*-ruting, begge benyttet med og uten flytkontroll. Svitsjene benytter like store inn- og utbuffer.



Figur A.69: Topologi S16E1. Plottet viser gjennomsnittsforsinkelse ved bruk av tradisjonelt Ethernet og Ethernet med deterministisk korteste-vei-up*/down*-ruting, begge benyttet med og uten flytkontroll. Svitsjene benytter like store inn- og utbuffer.



Figur A.70: Topologi S32E1. Plottet viser gjennomsnittsforsinkelse ved bruk av tradisjonelt Ethernet og Ethernet med deterministisk korteste-vei-up*/down*-ruting, begge benyttet med og uten flytkontroll. Svitsjene benytter like store inn- og utbuffer.

Tillegg B

De tusen første primtall

De primtall som er benytte ved simuleringer knyttet til denne oppgaven er markert med '*'.

2	3*	5	7	11	13*	17	19	23	29
31*	37	41	43	47	53	59	61	67	71
73	79	83	89	97	101	103	107	109	113
127	131	137	139	149	151	157	163	167	173
179	181	191	193	197	199	211	223*	227	229
233	239	241	251	257	263	269	271	277	281
283	293	307*	311	313	317	331	337	347	349
353	359	367	373	379	383	389	397	401	409
419	421	431	433	439	443	449	457	461	463
467	479	487	491	499	503	509	521	523	541
547	557	563	569	571	577	587	593	599	601
607	613	617	619	631	641	643	647	653	659
661	673	677	683	691	701	709	719	727	733
739	743	751	757	761	769	773	787	797	809
811	821	823	827	829	839*	853	857	859	863
877	881	883	887	907	911	919	929	937	941
947	953	967	971*	977	983	991	997	1009	1013
1019	1021	1031	1033	1039	1049	1051	1061	1063	1069
1087	1091	1093	1097	1103	1109	1117	1123	1129	1151
1153	1163	1171	1181	1187	1193	1201	1213	1217	1223
1229	1231*	1237	1249	1259	1277	1279	1283	1289	1291
1297	1301	1303	1307	1319	1321	1327	1361	1367	1373
1381	1399	1409	1423	1427	1429	1433	1439	1447*	1451
1453	1459	1471	1481	1483	1487	1489	1493	1499	1511
1523	1531	1543	1549	1553	1559	1567	1571	1579	1583
1597	1601	1607	1609	1613	1619	1621	1627	1637	1657
1663	1667	1669	1693	1697	1699	1709	1721	1723	1733
1741	1747	1753	1759	1777	1783	1787	1789	1801	1811

1823	1831	1847	1861	1867	1871	1873	1877	1879	1889
1901	1907	1913	1931	1933	1949	1951	1973	1979	1987
1993	1997	1999	2003	2011	2017	2027	2029	2039	2053
2063	2069	2081	2083*	2087	2089	2099	2111	2113	2129
2131	2137	2141	2143	2153	2161	2179	2203	2207	2213
2221	2237	2239	2243	2251	2267	2269	2273	2281	2287
2293	2297	2309	2311	2333	2339	2341	2347	2351	2357
2371	2377	2381	2383	2389	2393	2399	2411	2417	2423
2437	2441	2447	2459	2467	2473	2477	2503	2521	2531
2539	2543	2549	2551	2557	2579	2591	2593	2609	2617
2621	2633	2647	2657	2659	2663	2671	2677	2683	2687
2689	2693	2699	2707	2711	2713	2719	2729	2731	2741
2749	2753	2767	2777	2789	2791	2797	2801	2803	2819
2833	2837	2843	2851	2857	2861	2879	2887	2897	2903
2909	2917	2927	2939	2953	2957	2963	2969	2971	2999
3001*	3011	3019	3023	3037	3041	3049	3061	3067	3079
3083	3089	3109	3119	3121	3137	3163	3167	3169	3181
3187	3191	3203	3209	3217	3221	3229	3251	3253	3257
3259	3271	3299	3301	3307	3313*	3319	3323	3329	3331
3343	3347	3359	3361	3371	3373	3389	3391	3407	3413
3433	3449	3457	3461	3463	3467	3469	3491	3499	3511
3517	3527	3529	3533	3539	3541	3547	3557	3559	3571
3581	3583	3593	3607	3613	3617	3623	3631	3637	3643
3659	3671	3673	3677	3691	3697	3701	3709	3719	3727
3733	3739	3761	3767	3769	3779	3793	3797	3803	3821
3823	3833	3847	3851	3853	3863	3877	3881	3889	3907
3911	3917	3919	3923	3929	3931	3943	3947	3967	3989
4001*	4003	4007	4013	4019	4021	4027	4049	4051	4057
4073	4079	4091	4093	4099	4111	4127	4129	4133	4139
4153	4157	4159	4177	4201	4211	4217	4219	4229	4231
4241	4243	4253	4259	4261	4271	4273	4283	4289	4297
4327	4337	4339	4349	4357*	4363	4373	4391	4397	4409
4421	4423	4441	4447	4451	4457	4463	4481	4483	4493
4507	4513	4517	4519	4523	4547	4549	4561	4567	4583
4591	4597	4603	4621	4637	4639	4643	4649	4651	4657
4663	4673	4679	4691	4703	4721	4723	4729	4733	4751
4759	4783	4787	4789	4793	4799	4801	4813	4817	4831
4861	4871	4877	4889	4903	4909	4919	4931	4933	4937
4943	4951	4957	4967	4969	4973	4987	4993	4999	5003
5009	5011	5021	5023	5039	5051	5059	5077	5081	5087
5099	5101	5107	5113*	5119	5147	5153	5167	5171	5179
5189	5197	5209	5227	5231	5233	5237	5261	5273	5279
5281	5297	5303	5309	5323	5333	5347	5351	5381	5387
5393	5399	5407	5413	5417	5419	5431	5437	5441	5443

5449	5471	5477	5479	5483	5501	5503	5507	5519	5521
5527	5531	5557	5563	5569*	5573	5581	5591	5623	5639
5641	5647	5651	5653	5657	5659	5669	5683	5689	5693
5701	5711	5717	5737	5741	5743	5749	5779	5783	5791
5801	5807	5813	5821	5827	5839	5843	5849	5851*	5857
5861	5867	5869	5879	5881	5897	5903	5923	5927	5939
5953	5981	5987	6007	6011	6029	6037	6043	6047	6053
6067	6073	6079	6089	6091	6101	6113	6121	6131	6133
6143	6151	6163	6173	6197	6199	6203	6211	6217	6221
6229	6247	6257	6263	6269	6271	6277	6287	6299	6301
6311	6317	6323	6329	6337	6343	6353	6359	6361	6367
6373	6379*	6389	6397	6421	6427	6449	6451	6469	6473
6481	6491	6521	6529	6547	6551	6553	6563	6569	6571
6577	6581	6599	6607*	6619	6637	6653	6659	6661	6673
6679	6689	6691	6701	6703	6709	6719	6733	6737	6761
6763	6779	6781	6791	6793	6803	6823	6827	6829	6833
6841	6857	6863	6869	6871	6883	6899	6907	6911	6917
6947	6949	6959	6961	6967	6971	6977	6983	6991	6997
7001	7013	7019	7027	7039	7043	7057	7069	7079	7103
7109	7121	7127	7129	7151	7159	7177	7187	7193	7207
7211	7213	7219	7229	7237	7243	7247	7253	7283	7297
7307*	7309	7321	7331	7333	7349	7351	7369	7393	7411
7417	7433	7451	7457	7459	7477	7481	7487	7489	7499
7507	7517	7523	7529	7537	7541	7547	7549	7559	7561
7573	7577	7583	7589	7591	7603	7607	7621	7639	7643
7649	7669	7673	7681	7687	7691	7699	7703	7717	7723
7727	7741	7753	7757	7759	7789	7793	7817	7823	7829
7841	7853	7867	7873	7877	7879	7883	7901*	7907	7919