Universitetet i Oslo
Institutt for informatikk

Utvikling av TCP og RED-moduler i en Javabasert Nettverkssimulator

Bjørnar Libæk

Hovedoppgave

2. mai 2003

Forord

Denne rapporten er resultatet av mitt hovedfagsstudium ved Institutt for Informatikk ved Universitetet i Oslo i samarbeid med Simula Research Laboratory, utført i perioden 2001-2003.

Jeg vil takke Stein Gjessing for god veiledning og for hans motiverende engasjement.

En stor takk rettes også til Simula for gratis lunsj, og simula-elitefotball for inspirasjonsrike fotballkamper.

Bjørnar Libæk
Oslo, 1. mai 2003

Innhold

1 Innledning

1.1 Bakgrunn

1.1.1 RPR (Recilient Packet Ring)

Den raske veksten i båndbredde for LAN (Local Area Network) og WAN (Wide Area Network) teknologi har i det siste gjort at MAN (Metropolitan Area Network) har blitt hengende etter når det gjelder pakkesvitsjede nettverk. Dette har ført til at MAN-nettet, som fungerer som "ryggraden" som kobler sammen flere mindre nett, blir en flaskehals.

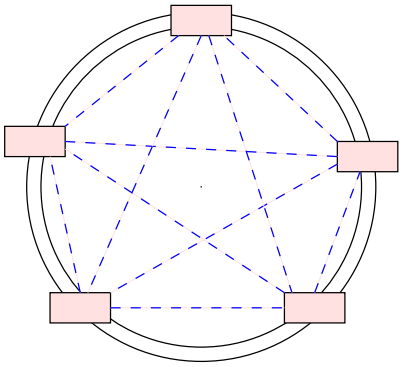
I dag brukes teknologier som Packet over SONET (PoS) og ATM over SONET til transport av IP (Internet Protocol) pakker i MAN. SONET (Synchronous Optical NETwork) er en forbindelsesorintert TDM (Time Division Multipleksing) protokoll som operer på det fysiske laget i OSI-modellen, og ble designet og optimalisert for transport av talesignaler. SONET benyttes ofte over ringer av optisk fiber, og utnytter ringstrukturen slik at beskyttelse ("protection") kan skje innen 50ms.

Etterhvert som mengden av IP-trafikk øker, får teknologiene som er basert på SONET problemer med å tilpasse seg. Pakke-basert transport ende til ende blir sett på som en løsning på dette skaleringsproblemet. Dette vil nå bli nærmere forklart.

1.1.1.1 Problemer med SONET. Som nevnt blir SONET vanligvis benyttet over ringer av optisk fiber. Det blir satt opp logiske kretser mellom nodene, der hver krets får tildelt en fast båndbredde (se figur 1). Hvis en krets ikke benytter seg av all båndbredden sin, kan den ikke brukes av de andre kretsene. Dette er en ulempe for data-trafikk, hvor pakker ofte kommer i klynger ("bursts"). Man kunne tenke seg at en krets fikk lov til å "låne" båndbredde fra de andre kretsene når en klynge med pakker skal sendes over denne kretsen, og de andre står ubrukt. Dette er altså ikke mulig i SONET. I tillegg er det kostbart å sette opp og administrere alle de logiske kretsene.

Et annet problem med SONET er manglende støtte for multicast. Hvis en pakke skal sendes til flere noder på ringen, må den kopieres og sendes ut på alle de aktuelle kretsene.

1.1.1.2 Ethernet i MAN. Siden de fleste lokalnett i dag bruker Ethernet-teknologi, er det naturlig å vurdere muligheten for å benytte Ethernet-svitsjing i MAN-nettverk. Ethernet er i dag standardisert med hastigheter opp til 10 Gigabit/s. På den måten kan Ethernet-rammer transporteres hele veien fra sender til mottaker, og man slipper unna unødig bruk av ressurser tilknyttet innramming av data. Ethernet-standarden er velkjent og velbrukt gjennom mange år, og er enkel og billig. Den har heller



Figur 1: *Logiske kretser i SONET - De stiplete linjene markerer logiske kretser, og de heltrukne linjene markere fiber*

ingen vesentlige problemer med å utnytte den tilgjengelige båndbredden, slik som SONET. Problemet er desverre at den ikke fungerer særlig godt i ringstrukturer, noe som er ønskelig siden det allerede er lagt ut fiber-ringer i mange byer som i dag brukes av SONET. Spesielt har ikke Ethernet funksjonalitet til å gjøre beskyttelse så raskt som SONET. Ethernet-svitsjing benytter spenntre-algoritmen for å unngå løkker. Denne kan også benyttes for å oppdage kabelbrudd, men bruker mye lenger enn 50ms på dette.

1.1.1.3 RPR kommer inn i bildet. Pga de ovennevnte problemene ble det identifisert et behov for en ny nettverksarkitektur, som både utnytter en ring-struktur, og som behandler pakkebasert trafikk på en effektiv måte. Våren 2000 startet arbeidet med å utvikle en standard som skulle forsøke å tilfredstille dette behovet. Standarden er fortsatt under utvikling, og heter RPR (Resilient Packet Ring) IEEE 802.17. I tillegg til de to egenskapene som allerede er nevnt, skal RPR også tilby fairness (QoS) og metningskontroll. Det er også et mål at det skal være lite ressurskrevende å styre systemet. Protokollen skal kunne benyttes både i LAN, MAN og WAN, og har som mål å støtte ringer med en omkrets på opptil 2000km med maks 255 noder. Seksjon 3.4 i denne rapporten bekriver RPR.

1.1.2 TCP (Transmition Control Protocol)

I motsetning til RPR som er helt ny teknologi, har transportprotokollen TCP blitt brukt i nærmere 30 år. Den ble først utviklet og prøvet ut i ARPANet (The Advanced Research Projects Agency Network), som er forløperen til dagens Internett. Det var i hovedsak det amerikanske forsvaret som stod bak ARPANet. Etterhvert ble det et behov for et

kommunikasjonsnettverk som kunne benyttes selv om enkelte maskiner eller linker ble skadet i en krigssituasjon. TCP/IP ble utviklet for dette formålet, og ble tatt i bruk på 70-tallet. TCP sørget for pålitelig kommunikasjon ende til ende, mens IP sørget for ruting av pakker gjennom det "upålitelige" nettet.

Etterhvert som flere institusjoner koblet seg til, har Internett vokst i et høyt tempo, og har i dag flere hundre millioner brukere på verdensbasis. TCP brukes som transportprotokoll for et betydelig antall applikasjoner, som f.eks FTP (File Transfer Protocol), HTTP (Hyper Text Transfer Protocol) og E-post. En betydelig andel av trafikken på Internett er derfor TCP-trafikk. (Målinger[1] utført i 1999 viste at av den totale trafikken på Internett, hadde TCP-trafikk en gjennomsnittlig andel på ca 93%.) TCP har hele tiden vært under kontinuerlig utvikling og kommet i ulike versjoner. De viktigste endringene opp gjennom årene, er tillegg av mekanismer som sørger for metningskontroll. Metningskontrollen gjør at TCP tilpasser senderaten sin til det underliggende nettets tilstand, slik at ikke metning oppstår. Disse mekanismene er sentrale i resten av denne rapporten, og vil bli beskrevet i detalj i seksjon 3.3.

1.2 Problemstilling

Når en ny nettverksarkitektur som RPR utvikles, er det absolutt nødvendig å gjennomføre simuleringer for å få en pekepinn på hvordan den oppfører seg i ulike situasjoner. Av den grunn er det i dag laget nettverkssimulatorer som simulerer RPR-ringer. Simuleringsresultatene vurderes og tolkes, slik at mulige problemer med standarden avdekkes før den endelige versjonen slippes.

Siden en stor del av dagens Internett-trafikk er TCP-trafikk, er det nødvendig å gjennomføre simuleringer der TCPs trafikkmønster etterliknes og sendes over en RPR-ring. Dette trafikkmønsteret er vanskelig å gjenskape ved hjelp av statistiske/matematiske funksjoner (syntetisk drevet simulering), på grunn av at TCPs trafikkmønster er komplisert og avhenger av hvordan det underliggende nettet responderer. Isteden er det mulig å gjøre såkalte eksekveringsdrevne simuleringer. Det vil si at man benytter en TCP-implementasjon som er utviklet spesielt for simulatoren. Denne "modulen" vil på bakgrunn av tilbakemeldinger fra systemet, gjenskape et realistisk trafikkmønster. Nettverkssimulatorer som NS-2[3] og OPNET[2] inneholder slike TCP-moduler.

Denne rapporten beskriver arbeidet med å utvikle en slik "lett" TCP-implementasjon som skal brukes til å simulere TCP-trafikk i en nettverkssimulaor skrevet i programmeringsspråket Java. Denne simulatoren er en såkalt "Discrete Event Simulator", og inneholdt fra før en RPR-modul som gjorde det mulig å simulere en RPR-ring. Oppgaven var dermed å utvikle en TCP-modul som kunne generere TCP-trafikk over en

slik ring. Denne TCP-implementasjonen måtte være i stand til å skape et trafikkmønster som er en god imitasjon av virkeligheten, og derfor inneholde de metningskontrol og flytkontrollmekanismene som dagens TCP-implementasjoner benytter.

I tillegg bestod oppgaven i å utvikle en RED (Random Early Detection)-modul til simulatoren. RED er en teknikk som benyttes i portnere ("gateways") i Internett for å raskere få TCP til å oppdage at metning er i ferd med å oppstå. Det var derfor også ønskelig å gjennomføre simuleringer der TCP-trafikken passerer RED-portnere på vei inn og ut av RPR-ringen.

1.3 Rapportens struktur

Denne rapporten beskriver som nevnt arbeidet med å utvikle disse to modulene. Kapittel 2 gjøres det rede for måten arbeidet er utført på, og hva slags arbeidsmetoder som er benyttet. Deretter gis en innføring i de aktuelle teknologiene . Dette gjøres i kapittel 3, som først beskriver OSI-modellen, et rammerverk mye brukt i datakommunikasjon, og deretter tar for seg TCP, RPR og RED. I kapittel 4, beskrives de ulike modulene i simulatoren i detalj, og det gjøres rede for hvilke valg som er gjort underveis. Kapittel 5 presenterer resultater oppnådd i simuleringer som hadde til hensikt å teste de forskjellige mekanismene som ble implementert. Til slutt i rapporten konkluderes det med at modulene oppfører seg tilsynelatende som forventet, men at de ikke har blitt testet godt nok til at man kan slå fast at de genererer et trafikkmønster som er tilstrekkelig realistisk.

1.4 Kildekode og elektronisk versjon av rapporten

Kildekoden til simulatoren inkludert de to modulene, og en elektronisk utgave av denne rapporten er tilgjengelig på web:

`http://www.simula.no/download/nd/rpr/TCP-RPR-libaek/`

Nedlastbar versjon:

`http://www.simula.no/download/nd/rpr/TCP-RPR-libaek.zip`

2 Metode

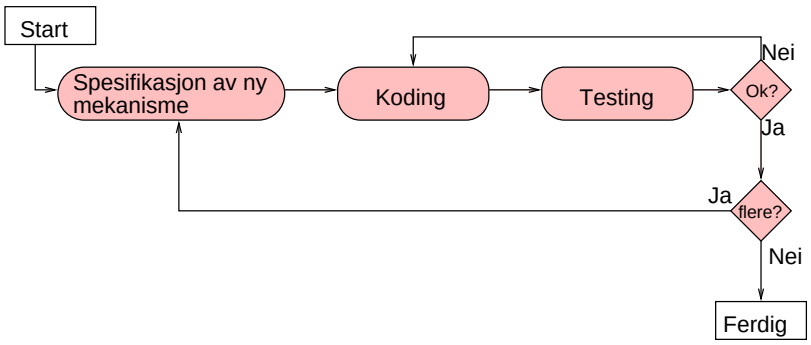
Dette kapittelet beskriver hva slags fremgangsmåte som er brukt i dette arbeidet.

Som nevnt innledningsvis, gikk oppgaven ut på å utvikle en TCP-modul, og en RED-modul til en nettverkssimulator. De eneste kravenene som ble stilt, var at TCP-modulen skulle skape et realistisk trafikkmønster, og at RED-modulen skulle oppføre seg i henhold til spesifikasjonen til RED-algoritmen. Denne algoritmen er detaljert beskrevet i én enkelt artikkel[4], som også inneholder kodeeksempler på hvordan RED kan implementeres. Dette gjorde at kodingen kunne starte tidlig uten store mengder forarbeid og litteraturstudier.

Når det gjelder TCP derimot, var situasjonen annerledes. Protokollspesifikasjonen til TCP er spredt rundt i et mangfold av RFC'er (Request for Comments). Årsaken er at nye versjoner, mekanismer og utvidelser har blitt lagt til opp gjennom årene. Mange av disse tilleggene og endringene har påvirket trafikkmønsteret TCP genererer. Hovedsakelig gjelder dette metningskontrollmekanismene. Dette førte til at en utfordring lå i å finne ut hva et "realistisk trafikkmønster" er for en TCP-strøm. Det første som måtte gjøres var derfor å sette seg inn i "TCP-jungelen", for å få en oversikt over alle de ulike versjonene, mekanismene og utvidelsene. Beskrivelsene av utvidelser og mekanismer var det relativt enkelt å finne frem til, siden de fleste spesifiseres detaljert i RFC'er. Der spesifikasjonene imidlertid var uklare, ble også kildekoden fra ordinære TCP-implementasjoner fra operativsystemer med åpen kildekode (Linux, BSD) studert. Når det gjaldt å få en oversikt over ulike versjoner (implementasjoner), var ikke det fullt så enkelt, men etter å ha lest diverse "Internetts historie" sider på nettet, sammenliknet disse, og vurdert de opp mot lærebøker og enkelte artikler, ble en oversikt etterhvert satt sammen. Grunnen til at dette var viktig, var for å finne ut hvilke TCP-mekanismer de mest brukte operativsystemene i dag implementerer. Andre nettverkssimulatorer ble også studert, for å se hvilke mekanismer og versjoner av TCP de har tatt med.

Når denne oversikten var klar, var det mulig å se for seg hvordan man kunne utvikle en TCP-modul som skaper et realistisk trafikkmønster. Arbeidet bestod da i å velge ut hvilke mekanismer og utvidelser som skulle være med. Disse valgene, og argumentasjonen bak dem, blir lagt frem i seksjon 4.4.

Når det var gjort, gjenstod å få en god nok forståelse av de ulike mekanismene slik at de kunne implementeres i simulatoren. Som sagt er de fleste av dem beskrevet relativt detaljert i spesifikasjonene, så utfordringen lå i å tilpasse dem til simulatorkjernen og den allerede eksisterende RPR-modulen. Dette arbeidet beskrives i seksjon 4.3 og 4.5.



Figur 2: *Fremgangsmåte for utvikling av TCP*

2.1 Utviklingsmetode

I større utviklingsprosjekter er det vanlig å velge en bestemt systemutviklingsmodell før arbeidet starter, avhengig av hva slags oppgave som skal løses. Siden dette prosjektet har vært forholdsvis lite, er det ikke gjort noe bevisst valg av en slik modell. Det er allikevel i ettertid mulig å beskrive en form for modell som ble fulgt. Denne gjelder bare for arbeidet med å utvikle TCP-modulen. RED-modulen er så liten at det er meningsløst å plassere arbeidet med den i en utviklingsmodell.

Oppgaven ble i grove trekk løst på denne måten (se figur 2):

- Utviklet aller først kjernen i TCP (Glidende vindu med ende til ende flytkontroll).
- Testet kjernen og rettet opp feil.
- La til nye mekanismer en etter en. For hver av disse:
  - Kodet mekanismen
  - Kjørte simuleringer og sjekket om resultatet ble som forventet.
  - Hvis ikke ble det gjennomført debugging og nye simuleringer ble kjørt helt til resultatet var ok.
- Kjørte simuleringer som testet at mekanismene fungerte i fellesskap. F.eks at Slow Start tar over etter en timeout.

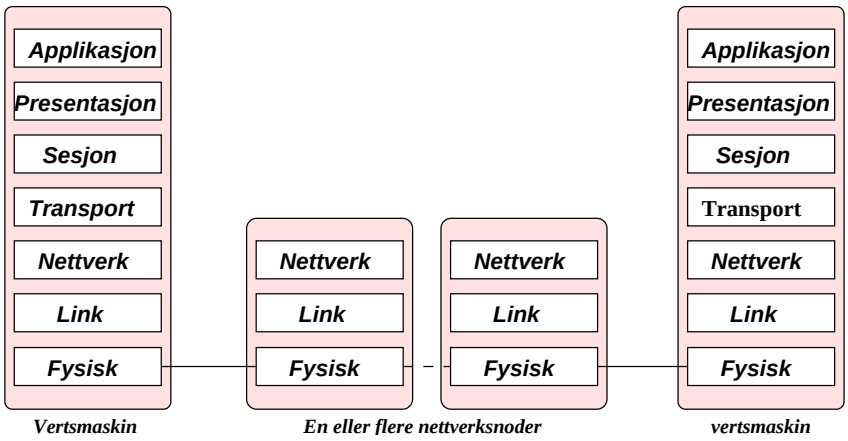
Flere av disse testene presenteres i kapittel 5.

De ulike mekanismene ble lagt til i denne rekkefølgen:

- Treveis håndtrykk
- Grensesnitt mot applikasjoner

- Zero window probing
- Timeoutmekanismen
- Slow start / Congestion Avoidance
- Fast Retransmit
- Fast Recovery

Det vil si at først ble mekanismene som gjør det mulig å opprette en forbindelse og overføre data pålitelig implementert, deretter ble metningskontrollen lagt til.



Figur 3: *OSI-modellen*

3 Teori

Dette kapittelet forklarer den viktigste teorien som denne oppgaven bygger på. Først beskrives OSI-modellen som brukes som et generelt rammeverk innen datakommunikasjon. Deretter beskrives TCP og RPR, og tilslutt gis en innføring i hvordan RED-mekanismen fungerer.

3.1 OSI-modellen

Når man tar for seg forskjellige deler av et kommunikasjonssystem, er det vanlig å plassere i dem i et rammeverk basert på lagdeling. Den mest brukte referansemodellen som gjør nettopp dette, kalles “Open Systems Interconnection (OSI)-modellen"[5] og ble definert av ISO (International Standards Organization). Poenget med en slik modell er å abstrahere bort detaljer fra de underliggende lagene, og å tilføre ny funksjonalitet på hvert lag. Bagrunnen for utviklingen av en referansemodell, var behovet for datakommunikasjon mellom datamaskiner fra forskjellige leverandører. Da disse f.eks benyttet ulike dataformater og protokoller ble programmeringen problematisk. Denne oppgaven vil flere steder referere til denne modellen. Her følger en kort beskrivelse av de forskjellige lagene, som også er illustrert i figur 3:

- *Fysisk lag*
  Tar seg av transmisjon og koding av en bit-strøm på det fysiske mediet.
- *Linklaget*
  Gjenkjenner en strøm av bit og setter disse sammen i en “ramme”. Funksjoner som feilkontroll, flytkontroll og synkronisering

er funksjoner som kan opptre på linklaget. Hvis det underliggende fysiske laget er et delt medium, sørger MAC(Medium Access Control)-laget for en rettferdig fordeling av tilgangen til dette mediet.

- *Nettverkslaget*
  Dette laget er ansvarlig for ruting av pakker mellom noder i nettet. Dataenheten blir her kalt “pakker”. Nettverkslaget tilbyr et grensesnitt til laget over som skjuler detaljer fra de underliggende lagene. En nettverksprotokoll kan enten være forbindelsesløs eller forbindelsesorientert. I det første tilfellet rutes pakker enkeltvis, enten ved at rutere tar egne rutingavgjørelser basert på pakkens destinasjonsadresse, eller ved at avsenderen bestemmer ruten (kilderuting). Hvis protokollen derimot er forbindelsesorientert, settes det først opp en logisk forbindelse i form av tilstand i alle svitsjene på veien. Denne forbindelsen settes opp ved hjelp av signalering før dataoverføringen starter, og tas ned når den er over.

- *Transportlaget*
  Tilbyr en ende til ende kanal mellom to prosesser, i motsetning til nettverkslaget som bare tilbyr ruting av pakker frem til endemaskinen. Dette gjøres ved å benytte et såkalt "portnummer", som brukes til å identifisere prosesser. Denne funksjonaliteten kalles "multipleksing". Dataenheten på dette laget kalles “meldinger”. Transportprotokoller kan også være forbindelsesløse eller forbindelsesorienterte. En forbindelsesløs transportprotokoll tilfører kun mulighetene til å rute meldingene helt frem til riktig prosess. En forbindelsesorientert protokoll derimot, kan tilføre funksjonalitet som sørger for pålitelighet, flytkontroll og metningskontroll. En forbindelse på transportlaget må også settes opp på forhånd, men i motsetning til på nettverkslaget er det kun endemaskinene som tar vare på tilstandsinformasjonen for en transportlagsforbindelse.

- *Sesjonslaget*
  Har ansvaret for å opprette, vedlikeholde og lukke sesjoner mellom applikasjoner. En sesjon kan bestå av flere forbindelser på transportlaget, og må derfor holde orden på disse.

- *Presentasjonslaget*
  Sørger for at data som sendes fra en applikasjon på ett system er leselig for applikasjonen på et annet system. Dette innebærer at dataene muligens må oversettes til ett felles kjent format. komprimering og kryptering er funksjoner som som kan opptre på presentasjonslaget.

- *Applikasjonslaget*
  Tilbyr et grensesnitt til applikasjonsprosessene for å sette opp forbindelser og for å sende og motta data. Dette grensesnittet blir ofte kalt et API (Application Programming Interface).

De syv lagene blir ofte nummerert og omtalt som “lag 1”, “lag 2” osv, der Fysisk lag er lag 1. De to viktigste lagene i denne oppgaven vil være linklaget, nettverkslaget og transportlaget der henholdsvis RPR, RED og TCP befinner seg.

3.2 Dagens Internett - “Best effort”

I dagens Internett brukes IP (Internet Protocol) som nettverksprotokoll. IP befinner seg altså på nettverkslaget, og er en forbindelsesløs protokoll. Det vil si at rutere ikke har noe per-flyt tilstand, men tar rutingavgjørelser på bakgrunn av destinasjonsadressen til hver enkelt pakke. Dagens Internett er designet rundt et grunnleggende prinsipp som kalles “best effort”. Det vil si at IP vil gjøre så godt den kan når det gjelder å levere data fra sender til mottaker, men det gis ingen garantier om at overføringen blir vellykket. Forskjellige feilsituasjoner kan oppstå, og IP overlater oppretting av disse feilene til overliggende lag. Hensikten med dette er at rutere i Internett skal bli så enkle som mulig, som igjen fører til lavere kostnader og bedre skalerbarhet. Det er imidlertid ingen selvfølge at dette er den riktige måten å fordele funksjonaliteten på. Det har i lengre tid foregått en diskusjon der det bl.a. argumenteres for at ruterne i Internett bør utstyres med mer funksjonalitet slik at det kan gis flere garantier. Det finnes teknologier som bygger på denne tankegangen. Et eksempel er ATM (Asynchronous Transfer Mode) som er en forbindelsesorientert nettverksteknologi. Her settes det opp "virtuelle kretser" i form av tilstandsinformasjon i alle nodene mellom avsender og mottaker. Følgene av dette blir en tilnærmet pålitelig overføring, men også kompliserte nettverksnoder.

Det er utenfor temaet til denne oppgaven å ta stilling til hva som er den “beste” måten å designe en nettverksarkitektur på, men det er viktig å se at begge tilnærmelser har både fordeler og ulemper, og vil hver for seg være bra til ulike formål.

3.3 TCP (Transmition Control Protocol)

Denne seksjonen beskriver sentrale deler av TCP. Pålitelighet, flyt og metnignskontroll og etablering av forbindelser er emner som blir berørt.

TCP er en transportprotokoll og befinner seg som navnet tilsier på transportlaget. Den er designet for å fungere over IP, som et suplement

til IP's manglende garantier. Disse to protokollene blir ofte omtalt som én protokoll med notasjonen TCP/IP.

Som nevnt er IP en forbindelsesløs protokoll som tilbyr en "best effort" tjeneste. Det innebærer at når pakker blir sendt med IP, har man ingen garantier for at

- de ikke blir borte,
- de ikke blir duplisert på veien,
- de ikke bytter rekkefølge med hverandre
- eller at de ikke inneholder bit-feil når de kommer frem til mottakeren.

Disse manglende garantiene gjør at vi sier at IP er "upålitelig". For et stort antall applikasjoner er denne tjenesten ikke tilstrekkelig. Lesing av e-post, surfing på Internett, nedlasting av filer og chatting på irc er eksempler på applikasjoner som krever en pålitelig overføring. Når man laster ned en fil, vil man at kopien skal være nøyaktig lik originalen.

Det er derfor behov for å tilføre funksjonalitet, slik at disse applikasjonene tilbys en pålitelig tjeneste. For å få til dette kreves at pakkene bærer mer informasjon, og at det settes opp en logisk forbindelse ende til ende. Det vil si at både senderen og mottakeren tar vare på ekstra tilstandsinformasjon tilknyttet en pakkestrøm. For å løse de fire problemene nevnt over, og dermed tilby pålitelighet, gjør TCP følgende:

- *Senderen retransmitterer tapte segmenter.* Pakkene nummereres med sekvensnummer, og mottakeren forteller senderen hvilke pakker som har kommet frem ved å sende kvitteringer. Det blir dermed opp til senderen å følge med på hvilke pakker som er kommet frem, og hvilke som eventuelt skal sendes på nytt. Dette krever at senderen bufrer pakker som er sendt, men som ikke er kvittert.
- *Mottakeren forkaster dupliserte pakker.* Ved å huske hvilke sekvensnummer som allerede har ankommet, kan mottakeren ignorere pakker som ankommer på et senere tidspunkt med det samme sekvensnummeret.
- *Mottakeren sorterer pakker som har byttet rekkefølge.* Sekvensnummerene benyttes også til dette. Hvis vi antar at pakkene nummereres i stigende rekkefølge, sørger mottakeren for at en pakke med sekvensnummer X ikke leveres til applikasjonen før alle pakker med sekvensnummer lavere enn X er levert. Dette krever at mottakeren bufrer pakker som ankommer ute av sekvens.

- *Mottakeren forkaster pakker som inneholder bit-feil.* Til dette benyttes en sjekksum-algoritme. Før senderen sender en pakke, beregner den en sjekksum ved hjelp av en spesiell algoritme. Denne summen sendes som en del av pakken. Når mottakeren får pakken, benyttes den samme algoritmen, og svaret sammenliknes med det som ble sendt med pakken. Hvis de er ulike, har pakken blitt skadet på veien, og blir forkastet.

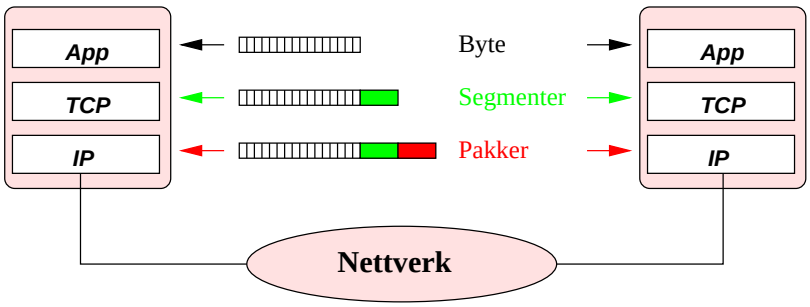
Disse mekanismene blir nærmere beskrevet i de neste avsnittene.

I andre tilfeller stiller ikke applikasjonen like strenge krav til pålitelighet, og da er det heller ikke noe poeng å benytte en pålitelig transportprotokoll. Et eksempel på en slik applikasjon er "live"-overføring ("streaming") av en videostrøm, der det ikke er kritisk at enkelte pakker blir borte. Her er det mer viktig at pakkene, eller bilderammene, ankommer innen en tidsfrist. Hvis TCP skulle blitt brukt i en slik sammenheng, ville eventuelle retransmisjoner kunne ankommet såpass sent at de allikevel måtte kastes. Av denne grunn er det ønskelig med en transportprotokoll som ikke bruker unødvendige ressurser på retransmisjoner. Det eneste transportprotokollen må sørge for i dette tilfellet, er multipleksingfunksjonalitet mellom applikasjonsprosessene på maskinen. UDP er et eksempel på en slik transportprotokoll. Hvis det er behov, kan applikasjonen selv implementere f.eks feilsjekking.

I tillegg til pålitelighet stilles også krav til ytelse hos transportprotokoller. Det gjelder å få de individuelle forbindelsene til å utnytte den tilgjengelige båndbredden så bra som mulig, samtidig som alle forbindelsene tar et kollektivt ansvar for at nettets ressurser ikke brukes opp og går i "metning". TCP har en innebygget metningskontroll som sørger for dette.

En tredje funksjonalitet som er ønskelig i tillegg til metningskontroll og pålitelighet, er flytkontroll. Dette er en mekanisme som forhindrer at sendereren sender data med en høyere rate enn det mottakeren klarer og håndtere. Mottakeren har begrensede ressurser i form av prosesseringskraft og minnestørrelse, så den må la senderen få vite hvilken rate den kan sende data med. Flytkontroll må altså ikke forveksles med metningskontroll, som er en mekanisme for å unngå å sende data med en høyere rate enn den det underliggende nettverket kan håndtere.

Denne seksjonen vil først forklare TCPs segmentformat, og vil deretter fortsette med å vise i detalj hvordan TCP kan tilby flytkontroll og en pålitelig tjeneste ved hjelp av "glidende vindu"-algoritmen. Deretter følger en beskrivelse av TCPs metningskontroll, etterfulgt av et avsnitt om ulike utvidelser. Til slutt i denne seksjonen gis en oversikt over ulike versjoner av TCP.



Figur 4: *Kommunikasjon mellom lagene*

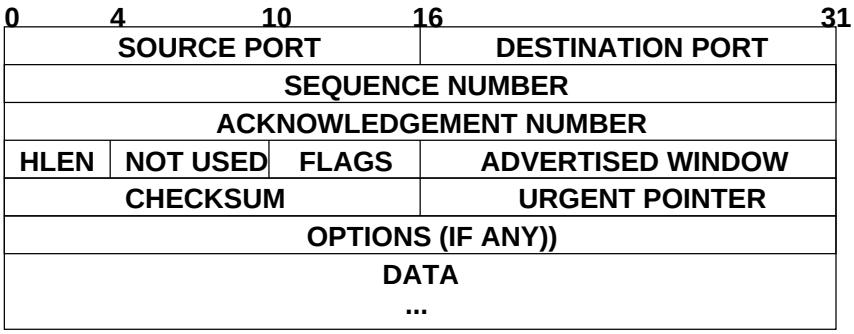
3.3.1 TCP segmentformat

TCP er en "bytestrøm" protokoll. Dvs at den på sendersiden mottar en strøm med byte fra programmet på nivået over, og på mottakersiden gir fra seg den samme strømmen med byte. Hvis vi tenker "lagvis", kan vi si at de forskjellige lagene kommuniserer på denne måten, som også er illustrert i figur 4:

- Applikasjonene sender data til hverandre. Hvis kommunikasjonen går over TCP, består disse dataene av byte.
- TCP sender "segmenter". Et segment består av et "hode" (figur 5) og en datadel. Hodet inneholder informasjon TCP trenger for å tilføre den nødvendige funksjonaliteten som nettlaget ikke har. Datadelen består av byte fra applikasjonsprosessen.
- På nettlaget overføres IP(Internet Protocol)-pakker. Segmentene fra transportlaget "pakkes" inn i IP-pakker, ved at det blir lagt til et IP-hode, før de sendes ut på nettet.

TCP-hodet har formatet som er vist i figur 5. Vi skal senere i teksten se mer detaljert hva de forskjellige feltene brukes til. Her er en kort forklaring:

- *Sourceport* og *Destinationport* brukes til multipleksing mellom flere prosesser på en maskin.
- *Sequencenumber* brukes til å nummerere hvert enkelt segment, slik at det er mulig å identifisere dem. På denne måten kan TCP se hvilke pakker som er forsvunnet i nettverket og hvilke som er kommet frem. Sekvensnummeret brukes også til å holde orden på rekkefølgen på segmentene.
- *Acknowledgementnumber* brukes av mottakeren til å fortelle senderen hvilke segmenter som er ankommet.



Figur 5: *TCP pakkehode*

- *Hlen* er lengden på TCP-hodet inkludert evt opsjoner.
- *Flags* inneholder flaggene SYN, ACK, FIN, RESET, PUSH og URG.
- *Advertised Window* inneholder annonsert vindusstørrelse.
- *Checksum* brukes av en algoritme for å sjekke om segmentet (og en del av IP-hodet) har blitt skadet under overføringen.
- *Urgent Pointer* peker dit hvor den delen av dataene som ikke er "urgent" begynner. En pakke som inneholder viktige data (viktige for applikasjonen), setter URG-flagget, og legger de viktige dataene først i datafeltet. Urgent pekeren markerer dermed slutten på disse dataene.
- *Options* feltet brukes til forskjellige valgfrie opsjoner (utvidelser).

For å gjøre overføringen effektiv, samler TCP et antall byte, og sender disse ut på nettet i ett segment, i motsetning til å sende ett og ett byte i hvert segment. Dette antallet kan ikke overstige variabelen MSS (Maximum Segment Size), som vanligvis blir satt lik MTU'en (Maximum transmission Unit) på nettet maskinen er tilkoblet, minus det antallet byte som IP og TCP -pakkehodet trenger. TCP sender av gårde et segment ved tre anledninger:

- TCP har mottatt MSS byte fra senderprosessen.
- Senderprosessen har eksplisitt sagt fra om at segmentet skal sendes med én gang. (F.eks hvis senderprosessen er telnet, som sender ett og ett tegn av gangen).
- TCP har ikke mottatt data fra senderprosessen på en stund, og en timer går av.

3.3.2 Pålitelighet og flytkontroll

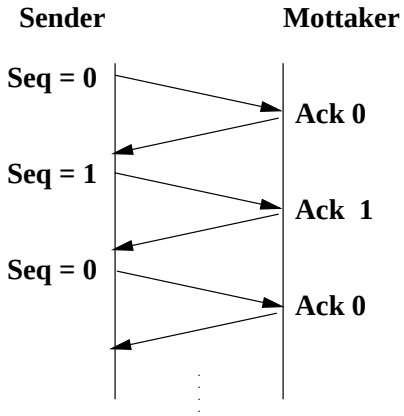
Som nevnt kan flere feilsituasjoner oppstå i nettet. Pakker kan forsvinne, bytte rekkefølge, dupliseres, skades og forsinkes. Det blir derfor opp til transportprotokollen og sørge for en pålitelig overføring, hvis dette er ønskelig av applikasjonen. De neste avsnittene forklarer hvordan TCP sørger for pålitelighet og flytkontroll ved hjelp av en algoritme som heter "glidende vindu", og som inneholder en timeout-mekanisme.

3.3.2.1 Glidende vindu (Sliding Window) For å sikre flytkontroll og en pålitelig overføring av data, er TCP bygget opp rundt en “glidende vindu”-algoritme. Under forklaringen av denne algoritmen, gjør vi den forenklingen at den ene parten kun er sender, og den andre parten kun er mottaker. I realiteten kan begge parter både være mottaker og sender, siden en TCP-forbindelse er duplex.

Glidende vindu er en utvidelse av en algoritme som kalles “Stop and Wait”. Denne går ut på at senderen nummererer segmentene med enten 0 eller 1, og at mottakeren kvitterer hvert enkelt av disse med en kvittering som inneholder sekvensnummeret til segmentet den kvitterer for. Når senderen mottar en slik kvittering, vet den at det forrige segmentet er levert, og kan sende et nytt. Hvis en kvittering på et segment ikke er ankommet innen en viss tid, sendes segmentet på nytt. Figur 6 viser en tidslinje for en Stop and Wait forbindelse, og kan oppsummeres slik:

- Senderen sender et segment med data. Segmentet har sekvensnummer 0.
- Mottakeren kvitterer for dette segmentet ved å sende en kvittering for segment nr 0.
- Senderen tolker denne kvitteringen som at segment 0 er mottatt og at mottakeren er klar til å motta segment nr 1. Senderen sender derfor de påfølgende data i et segment med sekvensnummer 1.
- Dette gjentas så lenge det er data og sende og det ikke oppstår pakketap.

Denne algoritmen oppfyller for så vidt krav til pålitelighet og flytkontroll, men den er lite effektiv. Siden senderen hele tiden må vente på kvitteringen fra mottakeren, står en stor del av båndbredden ubrukt. Glidende vindu har derfor til hensikt å utnytte denne ubrukte båndbredden. Hvis vi tenker på linkene mellom partene som et rør, gjelder det for senderen å dytte så mange pakker inn i dette røret som mulig. Tanken er å “holde røret fullt”. Når mottakeren tar en pakke ut av røret, vil senderen dytte inn en ny (figur 7). På denne måten vil senderen hele

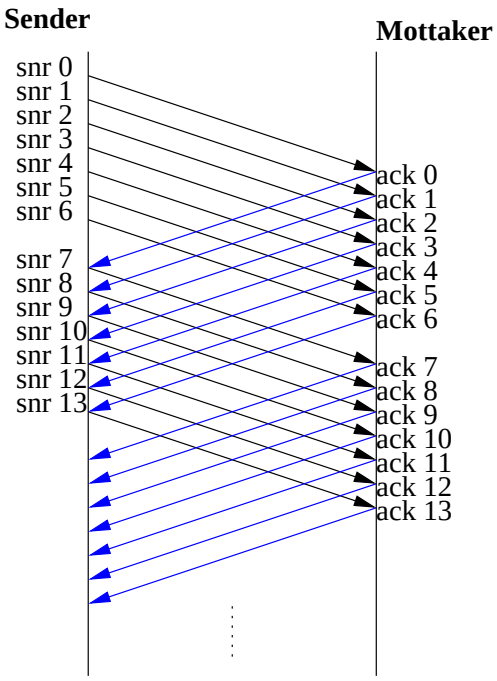


Figur 6: *Tidslinje for Stop And Wait*

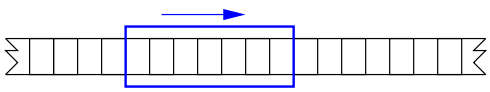
tiden ha flere segmenter ute som ennå ikke er kvittert for. Når en kvittering kommer, forteller det senderen at mottakeren har tatt en pakke ut av “røret”, og at det er plass til en ny. Senderen må forholde seg til en variabel som bestemmer hvor mange ukvitterte pakker den kan ha ute, eller hvor mange pakker det er plass til i røret. Denne variabelen kalles ‘"det nåværende vinduet" ("current window" eller "effective window"), og er oppad begrenset av to faktorer:

- *Mottakerens kapasitet.* Mottakeren holder senderen oppdatert om hvor mye ledig plass den har i mottaksbufferet ved å bruke feltet **ADVERTISED WINDOW** i kvitteringene. Dette kalles det "annonserte vinduet".
- *Nettets kapasitet.* Variabelen cnwnd ("concestion window" eller "metningsvinduet") er et mål på hvor stort vinduet kan være uten at nettet blir mettet. Metningskontrollen i TCP består hovedsakelig av mekanismer som styrer størrelsen på cnwd. Disse mekanismene forklares i detalj senere.

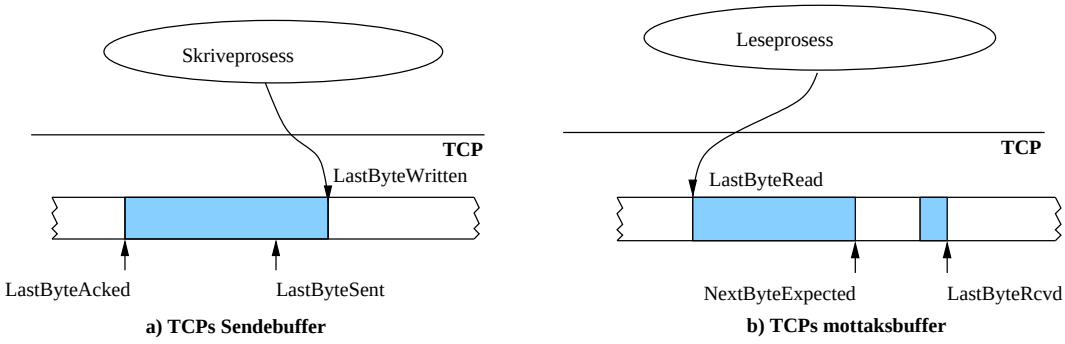
TCPs vindu vil altså være lik den minste av mottakerens annonserte vindu og metningsvinduet. Senderen er begrenset til å ikke sende mer enn en “vindusstørrelse” byte per RTT(Round Trip Time). Vi tenker oss at et vindu rammer inn de dataene i bufferet som senderen har lov til å sende (se figur 8). Når et antall byte blir kvittert for, flytter vinduet seg til høyre med det samme antallet byte. Etterhvert som kvitteringer ankommer, “glir” vinduet langs bufferet. Når det annonserte vinduet eller metningskontrollvinduet øker eller minker slik at TCP må tilpasse senderaten, sier vi at vinduet åpner og lukker seg.



Figur 7: *Tidslinje for Glidende vindu med vindusstørrelse 7.*



Figur 8: *Glidende vindu. Vinduet er et mål på mengden ukvittert data senderen får lov å ha. Vinduet glir mot høyre etterhvert som kvitteringer kommer inn.*



Figur 9: *TCP buffere*

De dataene som ligger til venstre for vinduet er data som allerede er sendt og kvittert. De dataene som ligger til høyre for vinduet er data som ennå ikke er sendt. Inne i vinduet ligger de dataene som er sendt, men som ikke er kvitter for (hvis vi antar at TCP sender alle data inne i vinduet umiddelbart etter at det har flyttet seg).

Senderens buffer. Algoritmen forklares nå mer i detalj, og vi begynner med senderen. Senderen har som tidligere nevnt et sendebuffer (figur 9a). Dette blir fylt opp med bytes av applikasjonsprosessen, og tømmes i den andre enden av TCP etter hvert som utsendte data blir kvittert for. I tillegg til det annonserte vinduet (AdvertisedWindow) og metningskontrollvinduet (cnwnd), har senderen tre viktige variable å ta vare på: LastByteSent, LastByteAcked og LastByteWritten. Alle disse tre er pekere inn i sendebufferet.

Vi tenker oss at LastByteWritten starter på byte 0 til venstre i bufferet, og flytter seg mot høyre etter hvert som applikasjonsprosessen fyller opp bufferet. LastByteAcked vil alltid være mindre eller lik LastByteSent, fordi et segment ikke kan kvitteres uten at det er sendt først. LastByteSent må være mindre eller lik LastByteWritten, fordi senderen ikke kan sende data som ennå ikke er skrevet av applikasjonsprosessen:

$$LastByteAcked \leq LastByteSent \leq LastByteWritten$$

For å tilpasse seg flytkontroll og metningskontroll, må senderen passe på at følgende invariant er gyldig:

$$LastByteSent - LastByteAcked \leq min(AdvertisedWindow, cnwnd)$$

Det vi si at antallet utestående byte (byte som er sendt men ikke kvittert) må være mindre eller lik det nåværende vinduet, som altså er lik det minste av mottakerens annonserte vindu og metningsvinduet.

Mottakerens buffer. Mottakeren har også tre pekere inn i bufferet sitt (figur 9b): NextByteExpected, LastByteRecvd og LastByteRead. Vi tenker oss at bufferet blir fylt opp fra venstre mot høyre etter hvert som data kommer inn fra nettet. NextByteExpected peker på plassen til høre for det siste bytet som er ankommet hvis alle tidligere byte også er ankommet. Dette er nødvendig siden segmenter kan komme inn i feil rekkefølge, og TCP vil ikke la leser-prosessen lese byte som er ute av sekvens. Derfor gjelder

$$LastByteRead < NextByteExpected.$$

NextByteExpected peker altså på det første "hullet" i bufferet (figur 9).

Hvis lese-prosessen er treg til å lese data i forhold til senderens senderate, og LastByteRead flytter seg saktere mot høyre enn LastByteRecvd, fylles bufferet opp. Flytkontroll er derfor nødvendig. Det er her TCP på mottakersiden må si fra til TCP på sendersiden, at den må sende mindre data. Dette gjør den som sagt med å annonsere en vindusstørrelse til senderen. Verdien til AdvertisedWindow er ikke noe annet en et mål på hvor mye ledig plass det er i mottaks-bufferet:

$$AdvertisedWindow = MaxRcvBuffer - (LastByteRecvd - LastByteRead)$$

MaxRcvBuffer er den totale plassen i bufferet. Flytkontroll håndteres altså ved at senderens vindu til enhver tid er mindre eller lik AdvertisedWindow. Hvis mottakerens buffer er fullt, vil LastByteRecvd minus LastByteRead være lik MaxRcvBuffer, og følgelig vil AdvertisedWindow bli lik null. Når senderen ser dette, må den stoppe å sende, og skrive-prosessen vil da kunne fylle opp sendebufferet. Hvis dette skjer vil skrive-prosessen bli blokkert helt til mottakeren øker AdvertisedWindow igjen, og plass i sendebufferet blir frigitt. Denne økningen kommer etterhvert som lese-prosessen leser data fra bufferet, LastByteRead øker, og plass frigis.

Når senderen venter på at vinduet skal åpne seg etter at det har vært helt lukket, er det fare for at forbindelsen går i en gjensidig ventetilstand. Senderen venter på en kvittering fra mottakeren som inneholder en vindusoppdatering, samtidig som mottakeren venter på et segment med data slik at den kan sende en kvittering som inneholder en vindusoppdatering. For å unngå denne tilstanden bruker TCP en mekanisme som kalles "Zero Window Probing". Den går ut på at senderen, mens den venter på en vindusoppdatering, sender ut "prober" når en spesiell timer utløper. Når mottakeren får denne meldingen, vil den sende en kvittering som enten åpner vinduet eller fortsatt lar det være lukket. Hvis senderen får beskjed om at det fortsatt er lukket, vil den doble timeoutverdien (eksponensiell backoff).

3.3.2.2 Timeout-mekanismen i TCP. Som nevnt benytter TCP en timer for å detektere pakketap. Denne fungerer slik at senderen vil få en timeout hvis det ikke kommer inn noen kvitteringer innen RTO (Retransmission Timeout) sekunder. Variabelen RTO kalkuleres på bakgrunn av RTT (Roundtrip Time)-målinger, og variasjoner i disse målingene. RTT-målinger skal i følge RFC1122[7] gjøres med Karn's algoritme[6]. Denne går ut på at senderen setter et tidsstempel i pakkehodet, som mottakeren returnerer. TCP benytter en variabel SRTT (smoothed RTT), som beregnes med et lavpassfilter, slik at endringer i RTT gis en minder vekt enn den gamle verdien av SRTT. Algoritmen er spesifisert rfc 2988 [8]. Hvor ofte TCP gjør en RTT-måling er forskjellig mellom ulike implementasjoner, men generelt anbefales det å ta målinger så ofte som mulig, og minst en gang per RTT er et krav i følge [8].

Når senderen får en timeout, må det tidligst sendte segmentet som ennå ikke er kvittert for re-sendes. Deretter må timeren startes på nytt, men nå med en doblet verdi for RTO. RTO settes altså lik 2*RTO, og fører til "eksponensiell backoff". Denne doblingen av RTO gjennomføres hver gang senderen får en timeout, helt til en kvittering som kvitterer nye data ankommer. Da går TCP tilbake til den "vanlige" utregningen av RTO, basert på RTT-målinger.

Siden senderen må være forberedt på å sende et segment på nytt, er det viktig at segmentet ikke fjernes fra sendbufferet før det er kvittert. Derfor må senderen ha pekeren *LastByteAcked* som vi så i forrige seksjon.

3.3.2.3 Delayed Acks er en mekanisme som gjør at kvitteringer blir holdt tilbake et bestemt tidsinterval før de sendes. Grunnen er at TCP vil forsøke å sende kvitteringene i samme pakke som data ("piggybacking"). Hvis en kvittering blir holdt tilbake, og ingen data skal sendes i løpet av tidsintervallet, vil *delayed ack timeren* gå av. RFC 813[9] sier at en "fornuftig" verdi på tidsintervallet er 200ms, men at en sofistikert TCP implementasjon kan ha en adaptiv mekanisme for beregne denne verdien.

3.3.3 Forbindelser i TCP

TCP er en forbindelsesorientert protokoll. Det betyr at før dataoverføringen kan starte, må det settes opp en forbindelse mellom de to endepunktene. Dette gjøres ved at partene lagrer flere variabler som beskriver tilstanden til forbindelsen. Siden TCP er en full-duplex protokoll, må begge parter i en forbindelse ha tilstand både som sender og mottaker. For hver forbindelse opprettes det en "kontrollblokk" i minnet som inneholder all tilstanden til forbindelsen, inkludert pekere til bufferne.

Det er operativsystemavhengig nøyaktig hvordan dette gjøres. Uttrykket "kontrollblokk" er hentet fra operativsystemet UNIX.

Pga muligheten for å ha flere TCP-forbindelser ut fra samme maskin, er det behov for multipleksingfunksjonaltet. Dette innebærer at TCP må vite hvilken forbindelse et innkommende segment tilhører. Til dette brukes et "portnummer", som er et unikt nummer hver forbindelse får tildelt på hver maskin. En forbindelse har derfor et portnummer i hver ende. Når et segment ankommer, inspiserer TCP **DESTINATION PORT** feltet i hodet, som beskriver hvilken port denne pakken skal til. På bakgrunn av dette hentes riktig kontrollblokk frem, og tilstanden til forbindelsen oppdateres. Deretter skrives dataene til mottaksbufferet for denne forbindelsen. Når applikasjonen skal lese dataene, oppgir den hvilken port den vil lese fra til operativsystemet, som leter frem mottaksbufferet og returnerer dataene. Det er igjen viktig å presisere at hvordan dette gjøres er avhengig av operativsystemet som brukes. I UNIX brukes Socketgrensesnittet som er en abstraksjon over portgrensesnittet. På denne måten blir det å lese fra en TCP-forbindelse det samme som å lese fra en fil for en applikasjon.

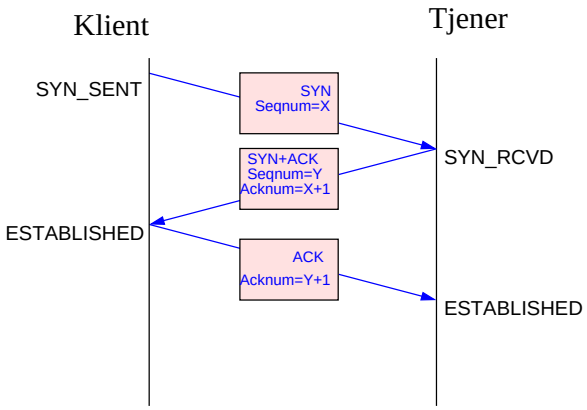
3.3.4 Oppkobling og nedkobling i TCP

Som nevnt må en forbindelse settes opp før dataoverføringen kan starte. Dette innebærer å sette av plass til tilstandsvariable i minnet, og å initiere disse. Når dataoverføringen er ferdig, må denne plassen frigjøres. Vi sier derfor at livet til en forbindelse består av tre faser: Oppkoblingsfasen, dataoverføringsfasen og nedkoblingsfasen.

Figur 11 viser et tilstandsdiagram for en TCP-forbindelse. Den øvre halvdelen er oppkoblingsfasen, mens den nedre er nedkoblingsfasen. I midten befinner tilstanden "ESTABLISHED" seg, som er dataoverføringsfasen. Denne kan også deles opp i flere undertilstander, men poenget med figuren er å vise opp- og nedkobling.

3.3.4.1 Oppkobling Poenget med en oppkoblingsfase er å utveksle overføringsparametere som er felles for begge parter, før selve dataoverføringen kan starte. Oppkoblingsfasen blir dermed en måte å synkronisere de to partene på. Hvilket sekvensnummer hver av de to skal starte med, er viktige parameterene i denne sammenhengen. Siden en TCP-forbindelse er duplex, er det i praksis to datastrømmer som sendes. Pakkene i hver av disse strømmene sekvensnummereres uavhengig av pakkene i den andre strømmen. Siden det initielle sekvensnummeret for en strøm skal velges tilfeldig[1], må de to partene fortelle hverandre

[1] Dette gjøres for å unngå at to inkarnasjoner av samme forbindelse benytter samme sekvensnummer innenfor en kort tidsperiode



Figur 10: *Treveis håndtrykk*

hvilket de skal starte med. Andre ting som kan avtales under oppkoblingsfasen, er hvilke utvidelser som skal benyttes under dataoverføringen (Ulike typer utvidelser beskrives i seksjon 3.3.7).

For at partene skal være sikre på at den andre har mottatt overføringsparametrene som har blitt utvekslet, kreves det at oppkoblingen er pålitelig. Ikke før den andre parten har bekreftet å ha mottatt informasjonen, kan TCP begynne å sende data. I TCP er dette løst med en såkalt "treveis håndtrykk"-protokoll. Den går i hovedsak ut på at den aktive parten ("klienten"), initierer oppkoblingen ved å sende sine parametere til den passive "tjeneren". Når tjeneren mottar denne pakken, svarer den med å sende en kvittering tilbake. Pakken med denne kvitteringen inneholder i tillegg tjenerens parametere. Ved mottak av denne pakken, får klienten en bekreftelse på at tjeneren har mottatt klientens parametere, og kan nå starte dataoverføringen. Siden denne pakken også inneholdt tjenerens parametere, må klienten sende en kvittering på disse. Når denne kvitteringen ankommer tjeneren, er protokollen fullført, og begge kan sende data. Figur 10 illustrerer kommunikasjonen. Når en pakke inneholder parametere, er SYN-flagget i pakkehodet satt. Når pakken inneholder en kvittering, er ACK-flaget satt.

Oppkoblingsfasen deles igjen inn i flere tilstander. En forbindelse befinner seg dermed i en spesiell tilstand avhengig av hvor langt treveis håndtrykket har kommet. Tilstandsdiagrammet i figur 11 viser alle tilstandene en forbindelse kan være i under dens levetid. Den øvre halvdelen (LISTEN, SYN_SENT og SYN_RCVD) tilhører oppkoblingsfasen, tilstanden ESTABLISHED representerer dataoverføringsfasen, og tilstandene på den nedre halvdelen tilhører nedkoblingsfasen. Pilene viser hvilke hendelser som gjør at en forbindelse forandrer tilstand.

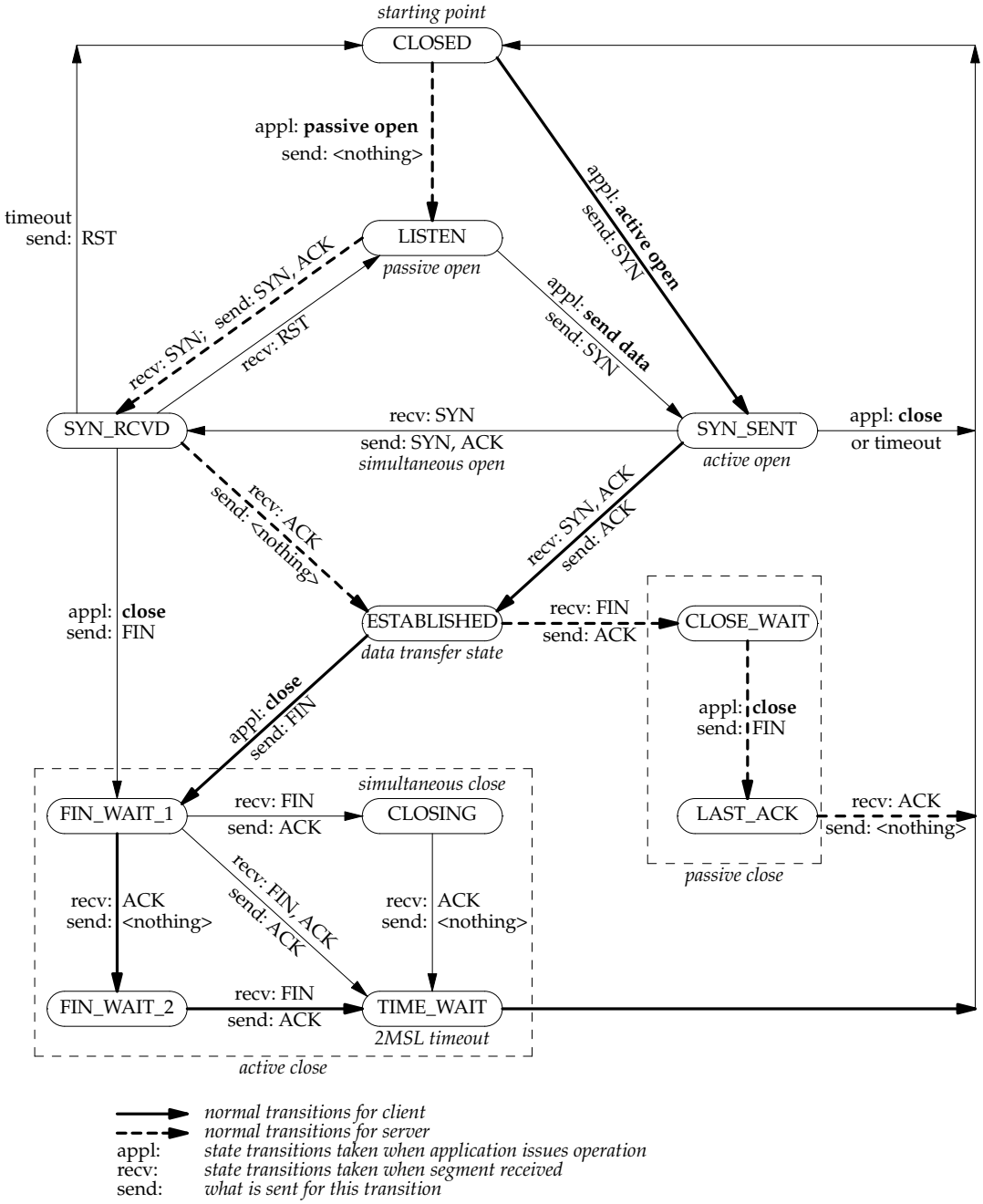
Her følger en oppsummering over treveis håndtrykk-protokollen, hvil-

ke felt i pakkehodet som benyttes, og hvilke tilstander forbindelsene befinner seg i:

- Tjeneren gjør en “passive open”. Det vil si at den legger seg til å “lytte” på en spesiell port. Dette portnummeret er kjent for klienten. Tjeneren er nå i tilstanden “LISTEN”.
- Klienten gjør en “active open” mot tjeneren. Det vil si at den sender et segment til tjeneren, hvor SYN-flagget er satt, og feltet *Sequencenumber* inneholder sekvensnummeret klienten ønsker å starte på. Nå er klienten i tilstanden “SYN_SENT”.
- Tjeneren svarer med å sende et segment tilbake, som både fungerer som en kvittering for klientens sekvensnummer, og som annonsering av sitt eget initielle sekvensnummer. Her er flaggene SYN og ACK satt. *Sequencenumber* inneholder som nevnt sekvensnummeret tjeneren ønsker å starte på. *Acknowledgementnumber* inneholder klientens sekvensnummer+1. Tjeneren er nå over i tilstanden “SYN_RCVD”.
- Klienten svarer til slutt med å kvittere for tjenerens sekvensnummer. Det gjør den ved å sette ACK-flagget i tillegg til å legge tjenerens sekvensnummer+1 i *Acknowledgementnumber*. Klienten er nå i tilstanden “ESTABLISHED”, og kan begynne å sende data.
- Når tjeneren mottar kvitteringen går også denne over i tilstanden “ESTABLISHED”, og forbindelsen er oppe.

Legg merke til at når et sekvensnummer x kvitteres, sendes verdien x+1 tilbake i kvitteringsfeltet. Dette står i motsetning til det som ble beskrevet under forklaringen på "stop and wait" og "glidende vindu" tidligere. Der ble sekvensnummeret kvittert ved å sende det tilbake i kvitteringen uten å forandre det. Poenget er at begge metodene kan benyttes, så lenge begge partene er enige om betydningen. I TCP sitt tilfell betyr altså dette feltet at mottakeren er klar til å ta i mot sekvensnummeret den setter i kvitteringsfeltet.

3.3.4.2 Nedkobling Når det gjelder terminering av forbindelsen, vil det ikke gis noen detaljert beskrivelse her. Det som er viktig er at begge partene må gjøre seg ferdig med dataoverføringen før forbindelsen termineres. Når den ene er ferdig med å sende sine data, setter den FIN-flagget, men kan ikke terminere før motparten er ferdig med å sende sine data. Når motparten mottar dette FIN-flagget, registrerer den at den andre er ferdig med å sende, men fortsetter allikevel å sende sine data. Når begge er ferdig, begge har altså mottatt et FIN-flagg og kvittert for



Figur 11: *Tilstandsdiagram for en TCP-forbindelse. De runde boksene representerer tilstander, og pilene mellom dem representerer hendelser som får en forbindelse til å forandre tilstand - hentet fra [10]*

dette, kan forbindelsen termineres. Det er imidlertid elementer som ikke er tatt med her, som kompliserer nedkoblingen.

3.3.5 Metningskontroll

Metningskontroll er en viktig del av TCP, og vil nå bli gjennomgått i detalj. Når to pakker ankommer en ruter omtrent samtidig på hver sin link, og begge skal ut på samme link, må en av pakkene vente. I denne ventetiden må pakken køes i ruteren. Hvis mye trafikk skal ut på samme link, vil køen vokse, og etterhvert fylles opp. Resultatet av en full kø er at pakker må kastes, og når slike pakketap forekommer over lengre tid sier vi at det har oppstått *metning*. Buffere kan også fylles opp som følge av at det ankommer en klynge ("burst") med pakker, men denne klyngen vil ankomme innenfor et en kortere tidsperiode, og bufferet vil få tid til å tømmes igjen. Dette regnes altså ikke som metning.

Metning er en uønsket situasjon, og det er derfor utviklet forskjellige metoder for å forhindre og for å komme ut av den. Både ruterne og endemaskinene (TCP) kan iverksette tiltak mot metning. I seksjon 3.5 forklares en algoritme med navn RED (Random Early Detection), som kan implementeres i portnere for å forsøke å unngå at en metningssituasjon oppstår. TCPs metningskontroll består av flere mekanismer som tilsammen gjør at TCP har muligheten til å komme seg ut av en metningssituasjon. Hovedsakelig er dette de fire mekanismene “Slow Start”, “Congestion Avoidance”, “Fast Retransmit”, og “Fast Recovery”. I løpet av de følgende avsnittene vil alle disse bli forklart.

3.3.5.1 To måter å øke metningsvinduet på. Målet til TCP, er å sende dataene så fort som mulig. For å finne en fornuftig verdi for metningsvinduet, er senderen nødt til å prøve seg frem ved å forsiktig øke senderaten. Det er to forskjellige måter TCP gjør dette på, der den ene er mer forsiktig enn den andre. I hvilke sammenhenger disse blir brukt, forklares i de neste avsnittene. Nå forklares kun hvordan de to fungerer. Den ene kalles “Congestion Avoidance” eller “additive increase”, og er den mest forsiktige av de to. Når TCP er i denne “modusen”, økes metningsvinduet med MSS/cnwnd bytes for hver kvittering som kommer inn. Dette fører til en lineær vekst av metningsvinduet. Hvis vi antar at hver kvittering kvitterer for MSS bytes, tilsvarer dette å øke metningsvinduet med MSS byte pr RTT.

Den andre “modusen” senderen kan være i, kalles “Slow start”. Da økes metningsvinduet med MSS bytes for hver kvittering som kommer inn. Når TCP går inn i Slow Start-modus, settes alltid metningsvinduet til ett segment, og senderen kan til å begynne med kun sende dette ene segmentet. Når kvitteringen for dette segmentet kommer, økes altså

metningsvinduet med MSS bytes, slik at den nye verdien blir 2*MSS = to segmenter. Når senderen har sendt disse to, og mottatt kvittering for begge, er metningsvinduet økt til fire segmenter osv. Dette fører til en eksponensiell vekst av vinduet. Slow Start brukes alltid helt i starten av en overføring. Metningsvinduet vokser da eksponensielt inntil en av følgende hendelser oppstår:

- *cnwnd passerer ssthresh*
  ssthresh (Slow start threshold) er en variabel TCP bruker for å bestemme nettopp hvor lenge slow start skal holde på. Initielt settes denne lik motpartens annonserte vindu. Følgen av dette er at senderaten vil øke eksponensielt helt til flytkontrollen tar over (forklart i seksjon 3.3.2).

- *Pakketap*
  Som vi skal se senere er dette et tegn på metning, og vil føre til at metningskontrolltiltak blir iverksatt.

Metningsvinduets eksponensielle vekst stemmer ikke helt overens med navnet "Slow start". Grunnen er at Slow start ikke sammenliknes med lineær vekst, men med å sende hele det annonserte vinduet på en gang, uten å vente på kvitteringer. Denne metoden ble brukt i tidligere versjoner av TCP.

3.3.5.2 Hvordan TCP oppdager metning IP har ingen funksjoner for å informere transportlaget om metning, så TCP må bruke annen informasjon den får fra nettlaget. Det er hovedsakelig pakketap det dreier seg om. Ved å implementere mekanismer for å detektere pakketap, og å kontinuerlig monitorere disse, kan TCP danne seg en oppfatning om nettets tilstand. Hvis ingen pakker forsvinner, er det et tegn på at nettets ressurser ikke blir utnyttet fullt ut, og det er derfor forsvarlig å øke senderaten. Hvis det derimot forekommer pakketap, er det tegn på metning, og senderaten må senkes. Imidlertid er det ikke alltid nødvendig å senke senderaten like mye. Hvis kun én pakke forsvinner, men de påfølgende pakkene kommer frem, er dette en indikasjon på at metning ikke er et faktum, men er i ferd med å oppstå. En annen mulig årsak til et enkelt pakketap kan være at pakken har blitt skadet under overføring og kastet som følge av en feildeteksjon. I Internett forekommer dette svært sjelden (mye mindre enn 1% [11]) i forhold til pakketap som følge av at buffrene fylles opp i en ruter. Det er derfor forsvarlig å anta at en overbelastet ruter er årsaken. Hvis derimot mange påfølgende pakker forsvinner, kan TCP med stor sannsynlighet slå fast at nettet er mettet. Da er det behov for en mer drastisk redusering av senderaten.

Spørsmålet nå er hvordan TCP oppdager pakketap. I 3.3.2 ble det nevnt at en timeoutmekanisme gjør nettopp dette. Fra nå av omtales

denne som “RTO(Retransmission Timeout)-timeren”. Når et segment med data sendes, startes RTO-timeren hvis den ikke allerede er i gang. RTO inneholder som beskrevet i 3.3.2 det antall sekunder RTO-timeren skal løpe. Når en kvittering som kvitterer ny data ankommer, vil RTO-timeren startes på nytt. Dette medfører at en timeout oppstår kun hvis det ikke ankommer noen kvitteringer i løpet av RTO sekunder. Hvis en enkelt pakke forsvinner i nettet, vil mottakeren allikevel sende kvitteringer når de påfølgende segmentene ankommer. Disse vil returnere til senderen før RTO-timeren slår til, og timeren vil bli startet på nytt.

Av dette ser vi at RTO-timeren kun kan brukes til å detektere tap av mange pakker etter hverandre, ikke ett enkelt pakketap. TCP trenger derfor ytterliger en mekanisme som kan gjøre nettopp dette. Denne mekanismen kalles “Fast retransmit”. Når én pakke forvinner, og de påfølgende kommer frem, vil mottakeren få et hull i bufferet sitt (se figur 9b), og vil fortsette å sende kvitteringer der **ACKNOWLEDGEMENT NUMBER** inneholder sekvensnummeret på det tapte segmentet. Senderen oppfatter dette som “dupliserte” kvitteringer, som kan tolkes på to måter. Enten er en pakke forsvunnet, eller så har det forekommet en omstokking av segmenter. Siden senderen ikke vet hvilke av disse to tilfellene som er årsak til den dupliserte kvitteringen, venter den til det er ankommet tre slike før den tolker det som at en pakke er mistet.

3.3.5.3 Hvordan TCP reagerer ved metning Vi har nå sett at det TCP har to forskjellige mekanismer for å detektere metning. Én som oppdager ett enkelt pakketap, og én som oppdager tap av mange pakker. Innledningsvis i denne seksjonen så vi at det ikke er nødvendig å redusere senderaten like mye når kun ett segment er forsvunnet. TCP reagerer derfor forskjellig på pakketap, avhengig av hvilken mekanisme som oppdaget det. Hvis en timeout oppstod, setter senderen ssthresh til halvparten av "flightsize" og går deretter inn i “Slow start”, som medfører den mest drastiske reduksjonen av senderaten. Flightsize er lastByteSent minus lastAckReceived, og er et mål på antall utestående pakker. Denne vil ligge i nærheten av det "nåværende vinduet". Hvis det derimot ble oppdaget tre dupliserte kvitteringer, vil senderen retransmittere det tapte segmentet (“Fast retransmit”) og utføre “Fast recovery”. Det betyr at isteden for å gå inn i Slow start etter pakketapet, vil cnwnd halveres (“multiplicative decrease”) og Congestion avoidance bli iverksatt. Fast Recovery gjøres forskjellig i ulike versjoner av TCP. Dette beskrives nærmere i neste seksjon.

3.3.6 TCP-versjoner

TCP har vært under kontinuerlig utvikling fra 70 tallet og frem til i dag. Gjennom denne perioden har TCP vært offer for både store og små forandringer og utvidelser. Dette avsnittet vil gi en oversikt over de viktigste av disse forandringene, og spesielt fokuseres det på TCP's metningskontroll.

Etterhvert som forskningsmiljøer har foreslått endringer i TCP-spesifikasjonen, har utviklerne av operativsystemer tatt inn disse endringene i TCP-implementasjonene sine. I henholdsvis 1988 og 1990 slapp operativsystemet BSD Unix ut to TCP-implementasjoner som offisielt fikk navnet "BSD Network Release 1.0 (BNR1)" (også kalt "Net/1") og "BSD Network Release 2.0 (BNR2)" (også kalt "Net/2"). Begge disse ble brukt i 4.3BSD-versjonen av operativsystemet. De to versjonene fikk kallenavnene "Tahoe" og "Reno" (som begge er er byer i delstaten Nevada). Noe senere, i 1995, ble det spesifisert en versjon som fikk navnet "Vegas"[12] (Las Vegas er også er en by i Nevada-ørkenen), men denne har blitt lite brukt. I 1996 kom "NewReno"[13], med små, men allikevel nyttige endringer til Reno-versjonen. De viktigste forskjellene mellom disse versjonene, er metningskontroll-mekanismene, og som sagt er det de det fokuseres på i denne oversikten. De fleste av disse ble detaljert beskrevet tidligere i dette kapittelet, og vil i liten grad bli gjentatt her.

3.3.6.1 TCP Tahoe Denne versjonen ble utviklet av Van Jacobson[14], som innførte følgende mekanismer:

- Additive increase/multiplicative decrease
- Slow start
- Fast Retransmit

Tidligere versjoner av TCP gjorde lite for å unngå metning, så innføringen av disse mekanismene var revolusjonerende på dette tidspunktet. Tahoe hadde altså ikke Fast Recovery-mekanisme, som ble innført i Reno. Fraværet av denne betyr at senderen går inn i Slow Start-modus etter at tre dupliserte kvitteringer har ankommet.

3.3.6.2 TCP Reno Den viktigste endringen mellom Tahoe og Reno, er innføringen av Fast Recovery. Det betydde altså at isteden for å "slow-starte" etter en Fast Retransmit, halveres metningsvinduet, og for hver dupliserte kvittering som ankommer etter dette, sendes en ny pakke ut. Reno innførte også "Delayed Acks"[9] og "Header prediction"[15]. Header prediction er en implementasjonsteknikk som gjør at koden som håndterer innkommende segmenter er optimalisert for at segmenter ankommer i riktig rekkefølge. Delayed Acks er beskrevet i 3.3.2.3.

3.3.6.3 NewReno-utvidelene Reno Fast Recovery fikk problemer hvis flere enn en pakke blir borte innenfor ett vindu med data. Grunnen er at Reno avslutter Fast Recovery modus med en gang en kvittering ankommer som kvitterer for ny data. Hvis flere pakker har blitt tapt i samme vindu, kan det hende denne ACK'en kun kvitterte data opp til neste "hull" i mottakerens vindu. Den tapte pakken som representerer dette hullet vil dermed ikke bli retransmittert, og en timeout vil etterhvert oppstå. I NewReno identifiseres en slik kvittering som en "delvis kvittering" ("partial acknowledgement"). Mottaket av en slik vil altså ikke avslutte Fast Recovery, men fortsette å benytte dupliserte kvitteringer som klokke for utgående data. Resultatet blir at senderen kan retransmittere en tapt pakke en gang per RTT (Round Trip Time), og dermed unngå en "katastrofal" timeout. SACK-utvidelsen (se3.3.7.3) gjør mer for å hindre dette problemet.

3.3.6.4 TCP Vegas Vegas[12] benytter Fast Retransmit og Fast Recovery som i Reno, men skiller seg ut på tre områder:

- *Beregner tilgjengelig båndbredde (metningsvinduet) annerledes.* Der Reno basserer beregning av metningsvinduet på mottak av kvitteringer, bruker Vegas en mer sofistikert metode. Den går ut på å regne ut hvor mye data som er køet i svitsjer og rutere, ved å se på variasjon i forsinkelse.
- *Oppdager pakketap på en annen måte.* I steden for å vente på tre dupliserte kvitteringer for å retransmittere en pakke, kan Vegas gjøre dette etter å ha mottatt en duplisert kvittering. Dette er mulig fordi senderen tar var på tidspunktet hvert segment sendes. Når en duplisert kvittering ankommer, sjekker senderen tidsstempelet på den første ukvitterte pakken, og re-sender dette hvis denne ble sendt ut for mer enn RTO sekunder siden. I motseting til Reno's RTO beregning som i praksis runder av verdien til nærmeste 500ms, er Vegas' RTO finkornet.
- *Annerledes oppførsel i Slow Start.* Den viktigste forandringen her er måten senderen avgjør at Slow Start skal avsluttes for første gang, og Congestion Avoidance skal starte. Reno benytter en fast initiell verdi på ssthresh til dette, som lett kan føre til pakketap. Vegas benytter også her en algoritme der variasjonen i forsinkelse overvåkes.

3.3.7 Utvidelser

Det finnes en rekke utvidelser til TCP som kan forbedre ytelsen i spesielle situasjoner.

3.3.7.1 Big windows

TCPs senderate er oppad begrenset av metningsvinduet. Hvis metningsvinduet har en størrelse på cnwnd byte, er senderaten begrenset til cnwnd/RTT Byte/s. Forsinkelse*båndbredde produktet er et mål på hvor mye ukvittert data senderen kan ha ute. Så lenge metningskontrollen tillater det, er det et poeng å ha et så stort metningsvindu som mulig, slik at senderaten blir høy. Siden feltet i TCP-hodet er på 16 bit, kan ikke metningsvinduet bli større enn 65535 byte. Dette kan medføre at tcp ikke klarer å utnytte all den tilgjengelige båndbredden. Tilfeller hvor dette kan være aktuelt er ved forbindelser med høy forsinkelse, f.eks over satellittkanaler, eller i nettverk med høy båndbredde, som f.eks RPR-nettverk. For å løse dette problemet finnes en utvidelse av protokollen i form av en opsjon som kalles “big windows” [16]. Denne øker antallet bit som brukes til annonsert vindu fra 16 til 30, slik at vinduet kan bli over 1 gigabyte stort. For at denne opsjonen skal kunne benyttes, må begge endene være enige om å bruke den. Dette forhandles under oppkoblingen. Om Big Windows-utvidelsen er nødvendig når det kommuniseres over RPR diskuteres i 4.4.2.

3.3.7.2 PAWS (Protection Against Wrapped Sequence numbers)

Et spesielt problem kan oppstå hvis Big Windows-utvidelsen benyttes. Når båndbredden økes, blir tiden det tar for en forbindelse å bruke opp hele sekvensnummer-rommet mindre. Vi sier da at sekvensnummeret "wrapper" (starter på nytt). Dette medfører at samme sekvensnummer kan brukes to ganger innenfor 2*MSL (Maximum Segment Lifetime), som er 120 sekunder. Dette er ikke ønskelig, fordi et segment kan i teorien køes så lenge at et nytt segment med samme sekvensnummer sendes ut. Mottakeren har da ingen mulighet til å skille disse to segmentene fra hverandre. RFC 1110[17] identifiserte dette problemet, og konkluderte derfor at Big Windows-utvidelsen ikke kunne i brukes i nettverk der pakker kunne bytte rekkefølge på veien. Internett har ikke denne garantien, og derfor ble det i RFC 1323[18] foreslått en løsning som fikk navnet PAWS. Denne går ut på at senderen setter et tidsstempel i pakkehodet før transmisjon, og at mottakeren benytter dette til å skille gamle segmenter fra nye. PAWS kan benytte det samme tidsstempelet som brukes til RTT-målinger (se 3.3.2.2).

3.3.7.3 SACK (Selective Acknowledgements)

TCP (Reno og delvis NewReno) oppnår dårlig ytelse hvis flere pakker går tapt innenfor ett vindu. I verste fall vil det oppstå en timeout. Siden TCP benytter kumulative kvitteringer, får senderen bare vite om maksimum ett tapt segment pr RTT. Dette er tilfelle fordi mottakeren vil fortsette å sende dupliserte kvitteringer helt til det retransmitterte segmentet ankommer. Ikke før dette er mottatt kan mottakeren eventuelt sende en ny duplisert kvitter-

ing for neste tapte segment. Når senderen mottar en duplisert kvittering, vet den ikke om det er ett eller flere segmenter som er forsvunnet. Den kan derfor velge mellom å kun re-sende det ene segmentet og vente en RTT for å finne ut om det var tilstrekkelig, eller den kan re-sende flere segmenter etter det ene som ble etterlyst av mottakeren. Den første løsningen lar båndbredden stå ubrukt, mens den andre kan risikere å sløse bort båndbredde ved å re-sende segmenter som mottakeren allerede har mottatt. SACK ([19]) er en utvidelse av TCP som løser dette problemet. Tanken er at kvitteringene fra mottakeren til senderen inneholder mer detaljert informasjon om hvilke segmenter som er ankommet. Dermed kan senderen re-sende alle segmentene som ikke er ankommet, men unngå å unødvendig re-sende segmenter som er kommet frem. Simuleringer [20] har vist at TCP med SACK implementert oppnår en bedre ytelse enn implementasjoner som ikke benytter SACK-opsjonen i situasjoner der flere en ett segment forsvinner innenfor ett enkelt vindu.

3.4 RPR (Resilient Packet Ring)

Denne seksjonen vil først gi et overblikk over RPR-standarden som er under utvikling. Seksjon 3.4.2 beskriver MAC-laget i RPR sitt grensesnitt mot laget over, og seksjon 3.4.3 beskriver RPR-arkitekturen i korthet.

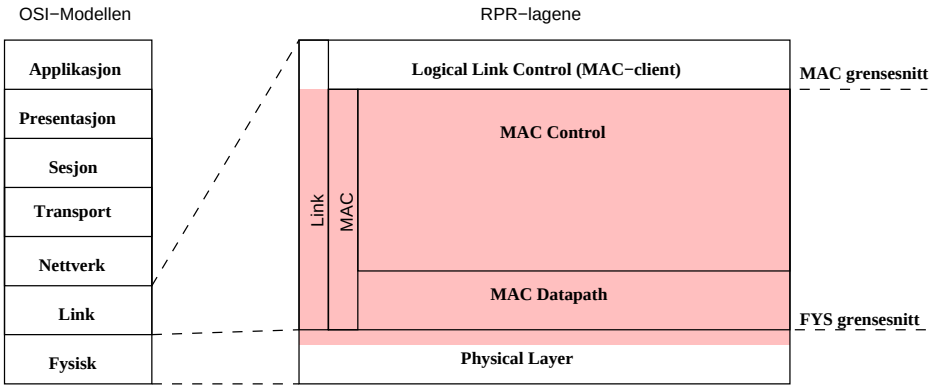
RPR-standarden spesifiseres i et dokument som utarbeides av IEEE 802.17-arbeidsgruppen. Dette er et såkalt "draft", og altså ikke en endelig versjon. Dokumentet er ikke offentlig tilgjengelig, men dispensasjon er gitt i forbindelsen med utviklingen av simulatoren. Stort sett all informasjonen om RPR i denne rapporten er hentet fra versjon 2.2 av dette dokumentet. Når det flere steder i teksten refereres til "standarden", er det dette dokumentet det vises til.

3.4.1 Overblikk over RPR

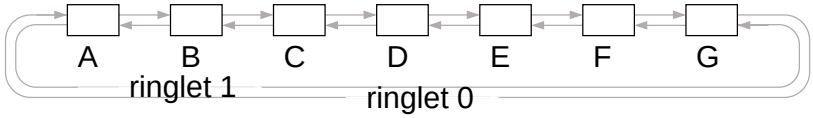
De neste avsnittene vil gi et overblikk over RPR. Detaljer om hvordan RPR-arkitekturen er bygget opp presenteres i seksjon 3.4.3.

3.4.1.1 RPR - lagdeling. For å få best mulig kontroll over ressursene på ringen, ble det valgt å la RPR være en protokoll på MAC-laget[2]. Grunnen er at det er på MAC laget man mest effektivt styrer tilgangen til et delt medium, noe en RPR-ring klassifiseres som. Standarden spesifiserer ikke hvilken teknologi som skal benyttes på det fysiske laget, men åpner derimot for at dette er valgfritt. Det vil si at både SONET og Ethernet kan brukes som underliggende teknologi.

[2]MAC (Media Access Control)-laget er et sublag av linklaget. Dette er den delen av linklaget som sørger for å fordele tilgang til et felles medium som deles mellom flere



Figur 12: *RPR-lagene i forhold til OSI-modellen. RPR omfatter den skyggelagte delen.*

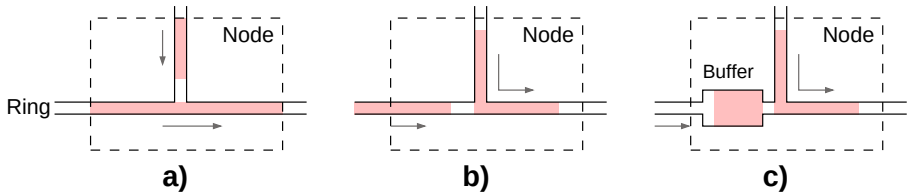


Figur 13: *RPR - ringstruktur*

Figur 12 viser de forskjellige delene av RPR sett i forhold til OSI-modellen, de skyggelagte delen blir definert i RPR-standarden. Vi kan se at MAC-laget igjen er delt inn i to sublag: *MAC-Control* og *MAC-Datapath*. I tillegg til innholdet i disse, spesifiseres også grensesnittet til "MAC-client"-laget (kalles "Logical Link Control" i 802-verdenen), og grensesnittet mot det fysiske laget. Siden et grensesnitt vanligvis defineres av laget på undersiden, i dette tilfellet det fysiske laget, trengs det et "tilpasningslag" mellom MAC-laget og det fysiske laget. Standarden beskriver også hvordan dette laget må se ut for de mest brukte fysiske teknologiene.

3.4.1.2 Ringstrukturen. En RPR-ring består av to ringer, én der rammer sendes med klokka, og én der rammer sendes mot klokka. Disse blir henholdsvis kalt "enkelt-ring 0" (eng: "ringlet") og "enkelt-ring 1" (se figur 13). Det kreves derfor at de fysiske linkene mellom nodene er full dupleks, slik at data kan sendes begge veier samtidig. Nodene på ringen blir kalt RPR-stasjoner eller RPR-noder, og er tilkoblet begge enkelt-ringene.

maskiner

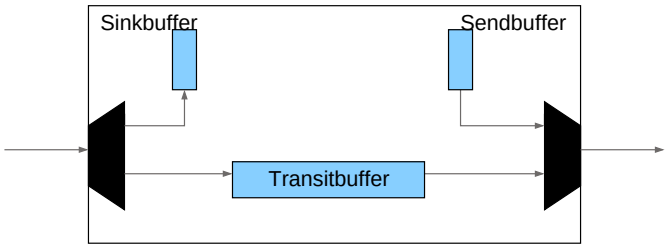


Figur 14: *a) Pakker som sendes ut på ringen kan kollidere med pakker som allerede er på vei forbi stasjonen. b) Pakker i transitt kolliderer med en pakke som er i gang med å sendes ut på ringen. c) "Buffer insertion": Pakken i transitt bufres så lenge noden sender en pakke ut på ringen.*

3.4.1.3 Buffer Insertion Ring. Et problem som må løses i alle typer ring-nettverk, er hvordan man kan sende en pakke ut på ringen uten at den kolliderer med en pakke som allerede er i transitt. Figur 14 viser to situasjoner der slike kollisjoner oppstår. I 14a er pakken i transitt allerede i gang med å passere noden som skal sende ut en pakke på ringen. I det denne noden starter å sende, vil de to pakken kollidere (og ødelegges). Dette problemet kan løses på samme måte som i Ethernet (CSMA/CD-protokollen), der node lytter på mediet og ikke starter å sende før det er ledig. Dette krever at pakken som skal sendes bufres. Situasjonen i figur 14b er det verre med. Der er noden i gang med å sende en pakke når en transittpakke ankommer. Her vil det oppstå en kollisjon. Andre typer ringer løser dette ved hjelp forhåndsallokerte tidsluker ("slotted ring"), eller ved hjelp av et "token" som gjør at kun en node kan sende om gangen ("Token Ring", IEEE802.5). Begge disse forhindrer kollisjoner, men utnytter ikke kapasiteten på ringen fullt ut. Dette er fordi mediet ofte vil stå ubrukt mens en stasjon venter på å få lov til å sende (enten fordi den venter på sin tidsluke, eller fordi den venter på et token).

RPR har imidlertid løst dette på en annen måte, ved at et "insertion buffer" (heretter kalt transittbuffer) innføres. Poenget med dette er å ha muligheten til å bufre pakker i transitt hvis en node er i gang med å sende en pakke ut på ringen. Dette illustreres i figur 14c. Avsnitt 3.4.1.5 forklarer hvordan RPR kan gjøre det som kalles "spatial reuse", siden det ikke er nødvendig å vente på en tidsluke eller et token for å sende data ut på ringen.

3.4.1.4 Tapsfri ring. En RPR ring skal være tapsfri. Det vil si at pakker i transitt ikke skal kastes av stasjonene (med mindre en pakke inneholder bitfeil). Dermed må transittpakker prioriteres foran pakker som skal sendes ut på ringen, slik at ikke transittbufferet fylles opp. Figur 15 viser hvordan bufferne i en enkelt-node på en enkelt-ring organiseres. En



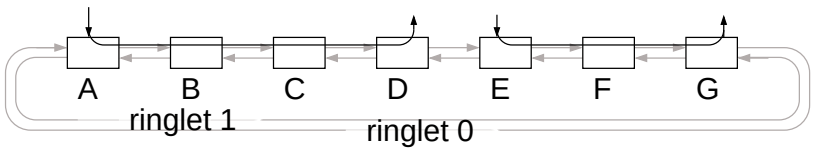
Figur 15: *Buffere i en enkelt-node på en enkelt-ring*

RPR-stasjon består altså av to slike, en for hver enkelt-ring. Her ser vi at det finnes to buffere for utgående pakker. Det er transittbufferet, og sendbufferet. Sendbufferet brukes til å bufre pakkene som skal sendes ut fra denne stasjonen.

For at ringen skal være tapsfri, må transittbufferet minst være stort nok til å bufre én pakke, slik at den pakken som er på vei ut fra denne stasjonen kan sendes ut i sin helhet. Når dette er gjort, må den bufrede transittpakken sendes med en gang. Det er her transittpakken eventuelt prioriteres foran en pakke i sendbufferet. Man kunne tenke seg en annen algoritme, der man sende pakker fra transittbufferet og sendbufferet annenhver gang, men det kan føre til at transittbufferet fylles opp, og pakker må kastes. Figuren viser også sinkbufferet. Innkommende pakker som er addresser til denne stasjonen "strippes" av ringen (forklares i neste avsnitt) og legges her.

3.4.1.5 Sending av data - Unicast. Når stasjon A skal sende data til stasjon D, må trafikken passere andre stasjoner. Når MAC-laget på stasjon A mottar pakken fra laget over, må det først avgjøres hvilken enkelt-ring pakken skal sendes ut på. Denne avgjørelsen vil sannsynligvis basseres på at det skal være færrest mulig hopp mellom sender og mottaker[3]. Deretter rammes pakken inn i et RPR-rammeformat, og legges i sendbufferet på den riktige enkelt-ringen. Omsider, når det er klart å sende denne rammen (når det ikke er noen transittrafikk), sendes den videre ned til fysisk lag som transmitterer rammen til neste stasjon på denne enkelt-ringen. I dette tilfellet vil dette være stasjon B. Den sjekker først om pakken er adressert til denne stasjonen, noe den ikke er. Dermed sendes rammen videre, med mindre den må bufres i transittbufferet fordi en pakke er på vei ut på ringen fra denne stasjonen. Til slutt ender rammen opp i stasjon D, som ser at den er adressert til seg. Dermed

[3]Dette er ikke bestemt i standarden. Det kan tenkes at man vil basere valget på f.eks minst avstand i meter, eller andre QoS-parametere. I resten av denne rapporten benyttes imidlertid "færrest hopp"-algoritmen.



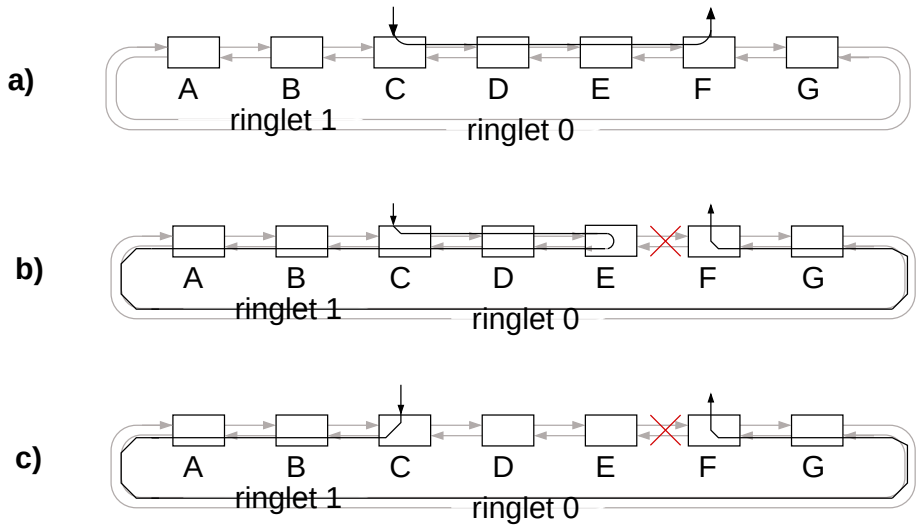
Figur 16: *RPR - Spatial Reuse*

"strippes" den av ringen (det vil si at den tas av ringen, som igjen betyr at den ikke videresendes), og dataene sendes opp til laget over. Siden pakken strippes av ringen hos mottakeren, er det mulig å benytte flere segmenter av ringen samtidig, slik at båndbredden ikke sløses bort. Dette kalles "spatial reuse", og er illustrert i figur 16. Her sendes rammer fra stasjon A til D, samtidig som stasjon stasjon E sender til G på samme enkelt-ring. Det er også mulig å sende data på den andre enkelt-ringen samtidig, men det vises ikke på figuren.

3.4.1.6 Sending av data - Broadcast og Multicast. RPR har også støtte for broadcast og multicast. Broadcast benyttes hvis en ramme er adressert til en ukjent mottaker (f.eks hvis mottakerstasjonen befinner seg på en annen ring tilknyttet via en RPR-bro[4]). Det vil si at denne mottakeren ikke finnes i topologibildet (forklares i neste avsnitt) til avsenderstasjonen. En broadcast/multicast pakke kan enten sendes ut på kun den ene enkelt-ringen, og strippes når den har gått helt rundt, eller den kan sendes ut på begge enkelt-ringene, og strippes når den har gått halvveis. En stasjon avgjør om en pakke skal strippes av ringen på basis av TTL (time to live)-feltet i RPR-hodet som settes av avsenderen og dekrementeres av alle stasjonene rammen er innom.

3.4.1.7 Topologi. Et viktig prinsipp for RPRs funksjonalitet er topologidatabasen. Alle stasjoner på ringen må ha et oppdatert bilde av topologien. Funksjonalitet som sending av rammer, beskyttelse og fairness benytter alle seg av denne databasen. Den inneholder hovedsakelig informasjon om hvilke stasjoner (MAC-adresse) som til enhver tid er tilknyttet ringen. For hver stasjon lagres også variable som antall hopp til denne stasjonen på begge enkelt-ringene. For at stasjonene skal kunne skape seg et slikt bilde, er det spesifisert en "Topology Discovery"-protokoll. Denne sørger for distribusjon av oppdatert topologi-informasjon.

[4]Flere RPR-ringer kan knyttes sammen vha RPR-broer. Standarden definerer også disse

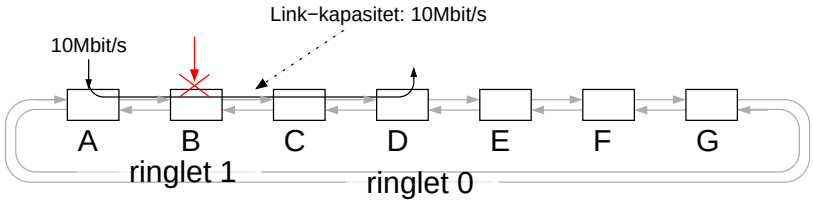


Figur 17: *a) Stasjon C sender data til stasjon F. b) Linken mellom stasjon E og F brytes, og stasjon E foretar "wrapping". Avsenderstasjonen C merker ikke dette. c) Linken mellom stasjon E og F brytes, og når avsenderstasjonen C oppdager denne topologiendringen foretar den "steering" og sender dataene ut på den andre enkelt-ringen i steden.*

3.4.1.8 Beskyttelse (Protection). Hvis en feilsituasjon oppstår, det kan være at en stasjon går ned eller brudd på en kabel, skal RPR automatisk sørge for at pakkene blir sendt den andre veien (på den andre enkelt-ringen) frem til mottakeren. Standarden har definert to forskjellige måter å gjøre dette på, "Steering" og "Wrapping" (se figur 17). Steering går ut på at avsenderstasjonen sender pakkene ut på den andre enkelt-ringen når den får informasjon om at en link har gått ned. Wrapping betyr at den noden som befinner seg nærmest bruddet sender alle pakkene den mottar tilbake på den andre enkelt-ringen. Wrapping gjøres altså kun lokalt på en stasjon og er transparent for andre stasjoner. Steering krever at det distribueres informasjon om feil til alle stasjonene på ringen, slik at de eventuelt kan velge å sende pakker den andre veien. Begge metodene utnytter ringstrukturen og kan i følge standarden utføre beskyttelse på under 50ms.

3.4.1.9 Fairness. For at ikke enkelte stasjoner skal kunne bruke "urettferdig" mye båndbredde på ringen, inneholder RPR en distribuert fairness-mekanisme. Ved fraværet av en slik mekanisme, er det mulig at enkelte stasjoner "sultes" (se figur 18).

Fairness-mekanismen i RPR baseres på distribusjon av fairness kontrollrammer. I all hovedsak går dette ut på at hver stasjon overvåker sine



Figur 18: *Stasjon B ønsker å sende til C på enkelt-ring 0, men sultes fordi A bruker opp all den tilgjengelige båndbredden. En fairness-mekanisme er derfor nødvendig*

innkommende linker (2 stk), og sender tilstandsmeldinger tilbake på de utgående linkene. Eksempelvis, kan stasjon B i figur 18 sende en fairnessmelding tilbake til A på enkelt-ring 1, som forteller at B har data å sende. A kan dermed senke senderaten slik at B slippes til. Denne informasjonen begrenses ikke bare til nabostasjoner, men sendes videre til andre eventuelle aggressive sendere.

3.4.2 MAC grensesnittet.

Avsnitt 3.4.1.1 beskrev lagdelingen i RPR, og forklarte at standarden definerer MAC-lagets grensesnitt mot MAC-klientlaget. Figur 12 illustrerer dette. Dette avsnittet vil gå nærmere inn på dette grensesnittet. Først gis en forklaring på hvilken funksjonalitet MAC-klienten har, deretter beskrives kontroll og dataprimitivene dette grensesnittet består av. I denne sammenhengen introduseres begrepet om tjenesteklasser. Disse definerer hvilke typer tjenester RPR tilbyr når det gjelder prioritering av ulike former for data.

3.4.2.1 MAC-klentlaget. Dette laget blir vanligvis kalt LLC (Logical Link Control)-laget for de andre IEEE 802-standardene. LLC er spesifisert som IEEE 802.2, og har hovedsakelig tre oppgaver:

- Tilby et felles grensesnitt for alle 802-protokollene mot nettverkslaget. MAC-lagene i disse protokollene (inkludert RPR) tilbyr ulike grensesnitt til laget over. Det vil altså si at nettverkslaget (IP) ikke trenger vite om det benyttes RPR eller vanlig Ethernet på MAC-laget.
- Demultipleksing slik at pakker blir levert til riktig nettverksprotokoll (IP/NetWare/NetBIOS osv).
- Pålitelighet på linklaget hvis det er ønskelig. Aktuelt hvis det fysiske mediet har høy feilrate, som f.eks trådløse nettverk (802.11).

For MAC-klientlaget over RPR er de to første nødvendige, men ikke den siste, siden transporten hovedsakelig vil gå over linker med lav feilrate. Påliteligheten blir dermed opp til lagene over. For RPR er det imidlertid andre oppgaver som er aktuelle å legge til dette laget. Dette er typisk funksjonalitet som er utelatt i MAC-laget for å gjøre det så enkelt og raskt som mulig. Et eksempel er "Virtual Destination Queueing" (VDQ) som er en teknikk som kan benyttes for å unngå "Head of line blocking"[21]. Dette er et problem som ofte oppstår når man har en felles ut-kø for alle destinasjoner. Hvis pakken som ligger først i køen ikke kan sendes fordi destinasjonen ligger bak et metningspunkt (bestemmes av fairness-mekanismen i RPR), sperrer den for andre pakker som ikke skal forbi metningspunktet. Disse pakkene har man mulighet til å sende dersom man har en kø for hver destinasjon (VDQ).

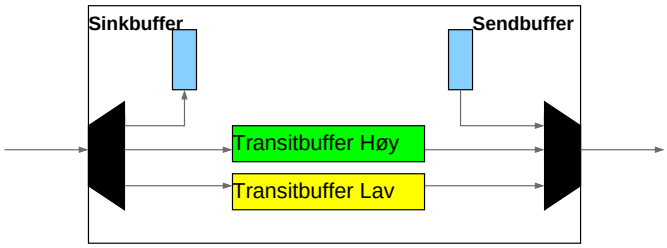
De to neste avsnittene beskriver henholdsvis hvilke tjenesteklasser og hvilke primitiver som er tilgjengelige for MAC-klientlaget

3.4.2.2 Tjenesteklasser. RPR tilbyr tre tjenesteklasser til laget over. Det vil si at MAC-laget diskriminerer rammer på bakgrunn av hvilken klasse/prioritet de tilhører, ved at de gis ulike garantier for båndbredde og forsinkelse. Nærmere bestemt er følgende tjenesteklasser definert:

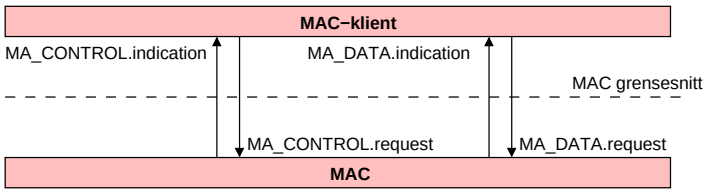
- *Klasse A.* MAC laget garanterer lav forsinkelse for all trafikk av denne typen som ikke overstiger den allokerte båndbredden for denne klassen. Den mengden av trafikk som ikke holder seg innenfor båndbredde-allokeringen kastes.
- *Klasse B.* MAC laget garanterer en begrenset forsinkelse for all trafikk av denne typen som ikke overstiger den allokerte båndbredden for denne klassen. Den mengden av trafikk som ikke holder seg innenfor båndbredde-allokeringen behandles som klasse C trafikk.
- *Klasse C.* MAC-laget gir ingen garantier for denne trafikken, annet enn at pakker på ringen ikke skal kastes (forklart i 3.4.1.4).

Ved bruk av klasse A og B må det altså allokeres båndbredde på forhånd. Hvis den allokerte båndbredden ikke er i bruk, kan den benyttes til trafikk av lavere klasser.

For å implementere en RPR-stasjon som skiller mellom høy og lavprioritetstrafikk, er det mulig å ha to transittbuffere på hver enkelt-ringnode. Det ene kan brukes til høyprioritetspakker (klasse A), mens det andre brukes til lavprioritetspakker (klasse B og C). På denne måten er det mulig å holde igjen lavprioritetspakker i transitt til fordel for pakker som skal sendes ut på ringen fra denne stasjonen.



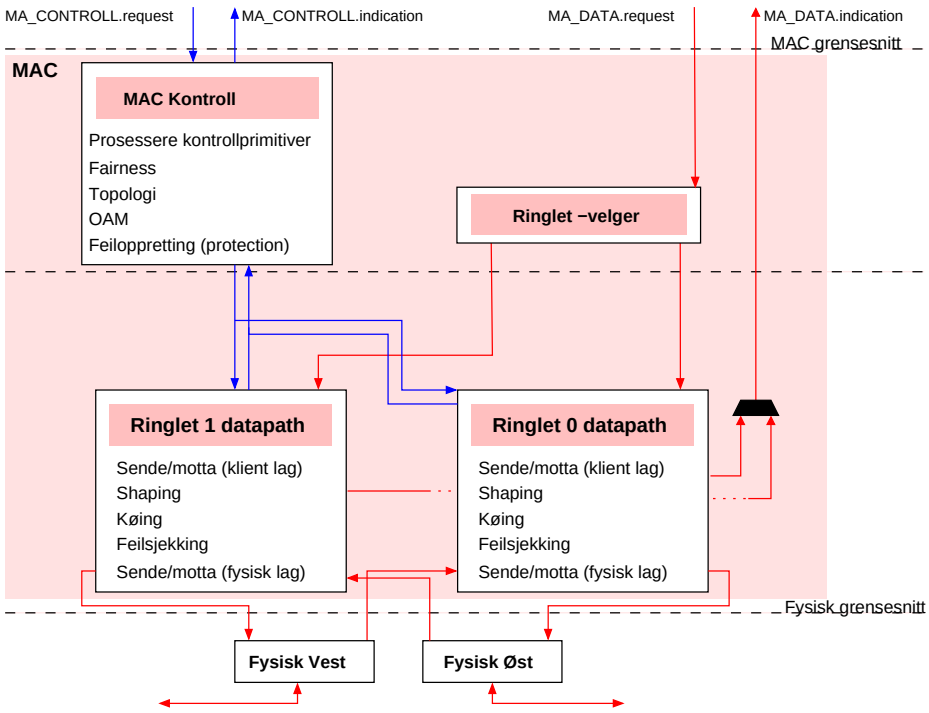
Figur 19: *Mulighet for å ha to transittbuffere for å skille mellom høy og lavprioritets pakker i transitt*



Figur 20: *MAC-primitiver*

3.4.2.3 MAC primitiver. Figur 20 illustrerer MAC-primitivene. De tar alle et definert sett med parametere, men disse forklares ikke i detalj.

- *MA_CONTROL.request* Benyttes for å sette konfigurasjonsparametere i MAC-kontrollaget, eller for å hente statusvariable.
- *MA_CONTROL.indication* Benyttes av MAC-kontrollaget for å sende statusinformasjon til MAC-klientlaget. Det er opp til klienten om den ønsker å benytte denne informasjonen. Dette kan være informasjon om topologiendringer, om det er oppstått metning på ringen, eller om det er tillatt å sende data. Det siste er spesielt viktig. Kontrollaget vil overvåke data som sendes, og i henhold til båndbredden som er allokert for de ulike tjenesteklassen ta vare på tilstand som sier om det er tillat å sende pakker eller ikke. Kontrollaget er pliktig å sende en indikasjon til klienten når denne tilstanden endrer seg i form av de såkalte "sendA", "sendB", og "sendC"-indikasjonene.
- *MA_DATA.request* Benyttes av MAC-klientlaget for å sende data til MAC-laget. En viktig parameter for dette primitivet, er tjenesteklassen (A, B eller C) som disse dataene tilhører.
- *MA_DATA.indication* Benyttes av MAC-laget for å levere data som har ankommet på en av enkelt-ringene, og som er adressert til denne maskinen, opp til MAC-klientlaget.



Figur 21: *RPR MAC-arkitektur*

3.4.3 RPR - arkitekturen

Denne seksjonen vil i korthet beskrive hvordan en RPR-stasjon er bygget opp arkitekturmessig. For en fyldigere beskrivelse henvises til standarden[22], men det som forklares i de følgende avsnittene er nok til å lese resten av denne rapporten.

Som forklart i 3.4.1.1, er MAC-laget delt inn i to sublag, *MAC Datapath* og *MAC Control*. Kontrollaget har ansvaret for å vedlikeholde fairness, beskyttelse (protection) og topologi - informasjon, mens datalaget sørger for mottak, trafikkforming ("shaping")[5] og videresending av pakker i henhold til parametere bestemt av kontrollaget. Dette gjelder både data som kommer fra MAC-klientlaget, og data som kommer inn på enkelt-ringene. De to lagene beskrives nå hver for seg:

3.4.3.1 MAC Datapath. Dette laget er organisert som vist på nederste del av figur 21. Det meste av funksjonaliteten er assosiert med en spesiell enkelt-ring, og derfor er det en instans av datapath-logikken for hver av de to enkelt-ringene. I tillegg finnes en del som er uavhengig av

[5] Shaping (trafikkforming) betyr at man forsinker pakker i en pakkestrøm, slik at strømmen ikke benytter for mye båndbredde

enkelt-ring, som er kalt "enkelt-ring-velger" i figuren. Hovedsakelig sørger denne for at data som kommer fra klienten sendes videre til riktig enkelt-ring. Det er imidlertid i datapath-logikken det meste av funksjonaliteten ligger. Hovedoppgavene til dette laget er altså å behandle pakker som mottas eller som skal sendes. Det er tre ulike måter datalaget kan motta en pakke på, og disse vil nå bli gjennomgått:

- *En ramme kommer inn fra fysisk lag.* Først gjøres feilsjekking, og rammen vil eventuelt kastes. Deretter sjekkes det om rammen er adressert til denne maskinen. Hvis den er det, sendes den enten til klienten eller til kontrollaget avhengig om det er en data eller kontrollramme. Hvis rammen ikke var adressert til denne maskinen, skal den videresendes ut på det andre fysiske grensesnittet, eller bufres i transittbufferet (figur 15).
- *En ramme kommer fra klienten.* Først velges det hvilken enkelt-ring rammen skal sendes ut på. Som tidligere nevnt kan dette gjøres på basis av antall hopp. Det er derfor nødvendig at enkelt-ring-velgeren har tilgang til topologidatabasen. Deretter sendes rammen til enkelt-ring-logikken for den riktige enkelt-ringen. Her gjøres trafikkforming, slik at trafikken fra denne stasjonen holder seg innenfor de båndbreddebegrensningene som er satt av for de forskjellige tjenesteklassene. Det er fairness-mekanismen som styrer trafikkformingsparameterne. Etter trafikkformingen køes pakkene og er klare for å sendes ut på ringen. Denne køen er sendbufferet, som også kalles "stagebuffer". Det har til hensikt å frakople timingmekanismene hos det fysiske grensesnittet fra timingen hos klientgrensesnittet (Det fysiske laget er implementert i maskinvare, mens klientlaget vil gjerne være implementert i programvare).
- *En ramme kommer fra kontrollaget* Bortsett fra at enkelt-ring-velgeren ikke benyttes her, er operasjonen nærmest identisk som for forklart i forrige punkt, og vil ikke bli beskrevet i nærmere detaljer her. Hva slags meldinger som sendes fra kontrollaget forklares i neste avsnitt.

Igjen poengteres at mange detaljer er utelatt fra denne forklaringen. I hovedsak dreier det seg om hvordan pakker ulik prioritet behandles forskjellig.

3.4.3.2 MAC Control. Hovedfunksjonene til kontrollaget vises også i figur 21. Det meste av funksjonaliteten som styres herfra er allerede gjennomgått, så dette blir en oppsummering:

- *Prosessere kontrollprimitiver.* Kontrollaget håndterer kontrollprimitivene mot klientlaget. Disse ble forklart i 3.4.2.3.

- *Fairness.* Som beskrevet i 3.4.1.9, benytter RPR en distribuert fairness mekanisme. Kontrollaget sørger for å utveksle fairnessmeldinger med de andre stasjonene på ringen, og å konfigurere "trafikkformerne" ("shapers") på de to enkelt-ringene.

- *Topologi.* Topologidatabasen bygges også opp ved hjelp av en distribuert algoritme, som forklart i 3.4.1.7

- *OAM (Operations, Admininistrations.& Management.* Dette punktet har ikke vært nevnt tidligere. Denne delen av kontrollaget benyttes til styring og administrasjon av systemet. Tilgang til denne modulen gis via MA_CONTROL.request-primitivet.

- *Beskyttelse.* Som nevnt i 3.4.1.8 finnes to typer beskyttelse, "wrapping" og "steering". Hvis steering brukes, må kontrollaget følge med på endringer i topologien, og eventuelt sørge for å konfigurere enkelt-ring-velgeren. Hvis wrapping benyttes, må kontrollaget overvåke linkene ut fra denne stasjonen. Hvis en av de brytes, f.eks utgående link på enkelt-ring 0, må all transitt-trafikk som kommer inn på enkelt-ring 0 sendes tilbake på enkelt-ring 1.

Flere av disse mekanismene kommuniserer altså med andre stasjoner, og til dette benyttes spesielle kontrollrammer. Kontrollaget benytter datalagets tjenester for å sende disse meldingene.

3.5 RED (Random Early Detection)

I 3.3.5 ble metningskontrollen i TCP forklart. Sentralt i denne diskusjonen var TCPs evne til å observere at nettet er i ferd meg å gå i metning, ved å monitorere pakketap. Videre så vi hvordan TCP reduserer senderaten ved en slik observasjon. Denne seksjonen tar for seg ruterne i nettet og hva de kan gjøre for å unngå at metning oppstår.

3.5.1 Bakgrunn

At en ruter må kaste en pakke kommer vanligvis av at et buffer er fullt, med mindre spesielle mekanismer som f.eks RED (Random Early Detection) er implemtentert. Når bufferet er fullt, er allerede metning oppstått, og det tar tid før TCP oppdager og reagerer på pakketapet. I løpet av denne tiden vil TCP-flytene gjennom denne ruteren fortsette å sende med samme rate, og vil derfor miste mange pakker. Det er derfor ønskelig at TCP oppdager metningen en viss tid før bufferet i ruteren er fullt, slik at reduksjonen i senderaten forhindrer at metningen i det hele tatt oppstår. Siden ruteren selv er den første som ser at bufferet er i ferd med å fylles opp, er det naturlig at denne implementerer en mekanisme

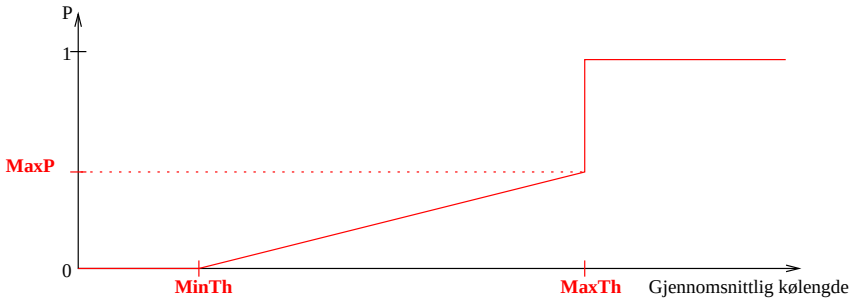
som kan hjelpe transportprotokollen med å oppdage metning tidligere. RED er en mekanisme som gjør nettopp dette. Ved å overvåke bufferet med en spesiell algoritme, oppdager ruteren at bufferet er i ferd med å fylles opp, og kan begynne å kaste pakker før bufferet er helt fullt. På denne måten sendes en implisitt melding til TCP om at den skal redusere senderaten. Selve algoritmen for å bestemme hvilke pakker som skal kastes forklares i neste avsnitt. Merk at dette er en metode som er spesielt designet for TCP, men RED kan også brukes i samarbeid med andre transportkontroller. F.eks kan ruteren sette et metnings-flagg i pakkehodet, som forteller transportprotokollen at den skal redusere senderaten. Eksempler på dette er ECN (Explicit Congestion Notification) [24] og DEC-bit [23].

3.5.2 RED-algoritmen

RED [4] er en algoritme som avgjør hvilke innkommende pakker som skal merkes. I tilfeller der RED skal fungere sammen med TCP består denne merkingen av å kaste pakken. Først gis et overblikk over algoritmen, deretter forklares noen av elementene i detalj. Målet er som nevnt å merke pakker før bufferet fylles opp. Dette gjøres ved at for hver innkommende pakke regnes det ut en gjennomsnittlig kølengde. Denne sammenliknes med en øvre (MaxTh) og nedre (MinTh) grense som er bestemt på forhånd. Hvis den gjennomsnittlige kølengden er lavere en den nedre grensen, blir ikke pakken merket. Hvis den er høyere enn den øvre grensen, blir pakken merket. Hvis den imidlertid er et sted mellom den øvre og nedre grensen, blir pakken merket med en sannsynlighet P. Denne sannsynligheten er en funksjon av den gjennomsnittlige kølengden, slik at jo nærmere den er den øvre grensen, jo større er sannsynligheten for at denne pakken blir merket (figur 22). Funksjonen som regner ut P, har som mål at pakker skal merkes med jevne mellomrom, og å merke nok pakker til å kontrollere den gjennomsnittlige kølengden. Årsaken til at det er den gjennomsnittlige kølengden som brukes, og ikke den virkelige, er at trafikken ofte er ujevn. Det er først når trafikken er relativt høy over tid at metning oppstår. Den virkelige kølengden kan variere mye over kort tid, og det er derfor ikke riktig å merke pakker kun fordi denne er høy.

Det er to algoritmer som bør forklares i denne sammenhengen. Den første er den som regner ut den gjennomsnittlige kølengden, og den andre er den som regner ut sannsynligheten P. Begge er hentet fra [4].

3.5.2.1 Beregning av gjennomsnittlig kølengde. Den gjennomsnittlige kølengden beregnes ved hjelp av et lavpassfilter. Meningen er at variasjoner i kølengden innenfor korte tidsintervaller ikke skal påvirke



Figur 22: *Random Early Detection (RED)*

den gjennomsnittlige kølengden. På denne måten skilles klynger (bursts) med pakker fra metning. Lavpassfilteret er et såkalt EWMA (Exponential Weighted Moving Average), der den gjennomsnittlige kølengden "gjsn" tilordnes en mye tyngre vekt enn kølengden "k":

$$gjsn \leftarrow (1 - vekt) * gjsn + vekt * k \ , \ vekt = 0.002$$

Vekten er satt til 0.002 som foreslått i [25], men kan varieres noe avhengig av hvor store klynger man ønsker å tillate. Dette gjennomsnittet beregnes hver gang en pakke legges inn i køen.

Et spesialtilfelle oppstår når en pakke ankommer en tom kø. Da beregnes m som det antallet små pakker som kunne blitt sendt ut av bufferet i den perioden køen var tom. Deretter beregnes gjennomsnittet som om m pakker skulle blitt lagt inn køen med en kølengde lik null. Hvis s er transmisjonstiden for en liten pakke, og tom_tid inneholder tidspunktet der køen ble tom, gjøres denne utregningen slik:

$$m \leftarrow \frac{tid - tom\_tid}{s}$$

$$gjsn \leftarrow (1 - vekt)^m * gjsn$$

3.5.2.2 Beregning av sannsynligheten. Når en pakke ankommer køen, og den gjennomsnittlige kølengden er mellom de to grensene MinTh og MaxTh, må sannsynligheten P for at denne pakken skal kastes beregnes. Som beskrevet over, er P en funksjon av den gjennomsnittlige kølengden gjsn som vist på figur 22. I tillegg øker P hvis det er lenge siden sist en pakke ble kastet. Bakgrunnen for denne algoritmen er forklart i [4].

Først beregnes den "initielle sannsynligheten" P_i, ved hjelp av likningen

$$P_i \leftarrow maxP * \frac{gjsn - MinTh}{MaxTh - MinTh}$$

På denne måten vil P være null når gjsn er lik MinTh, og lik maxP når gjsn er lik MaxTh, akkurat som vist på figur 22 (brøken vil variere mellom null og én når gjsn varierer mellom MinTh og MaxTh). Deretter justeres P_i, slik at den endelige sannsynligheten P øker med antall pakker som har ankommet siden sist en pakke ble merket (count):

$$P \leftarrow \frac{P_i}{1 - count * P_i}$$

Med denne metoden sikres det i følge [4] at pakkene blir kastet ofte nok til å kontrollere den gjennomsnittlige kølengden, men ikke for ofte slik at global synkronisering oppstår. Global synkronisering kan inntreffe hvis pakker fra mange strømmer kastes omtrent samtidig, slik at alle strømmene senker senderaten på likt

3.6 Oppsummering

Dette kapittelet har gitt en innføring i de tre teknologiene denne rapporten berører. Førts ble OSI-modellen gjennomgått, et rammeverk der de tre teknologiene kan plasseres i. Deretter, i seksjon 3.3 ble transportprotokollen TCP gjennomgått, med vekt på flyt og metningskontrollmekansimer. Seksjon 3.4 ga en oversikt over RPR-standarden som er under utvikling. Til slutt ble RED beskrevet, som er en mekanisme som implementeres i portnere for å hjelpe TCP med å tidligere oppdage metning. Det neste kapittelet vil gå gjennom de forskjellige modulene i simulatoren hver for seg.

4 Diskusjon og beskrivelse av simulatoren

Som nevnt i innledningen, var målet med dette arbeidet å utvide en simuleringsmodell skrevet i Java, med en TCP-modul og en RED-modul. Den eksisterende modellen bestod hovedsakelig av en RPR-modul, bygget opp rundt en simulatorkjerne. Denne seksjonen vil først ta for seg kjernen, og den allerede eksisterende RPR-modulen. Deretter beskrives RED og TCP-modulene. Underveis forklares metoden som er brukt, hvilke problemer som har oppstått og tilhørende løsninger .

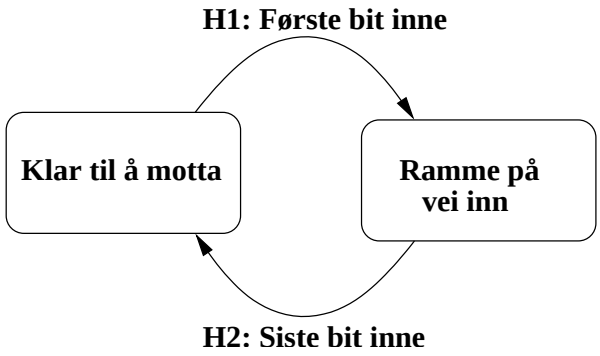
4.1 Simulatorkjernen

Denne seksjonen vil først forklare litt om hva en "discrete event simulator" er, og deretter beskrives hvordan dette er implementert i denne simulatoren

4.1.1 Discrete event simulator

Simulatoren er en “discrete event simulator”. Det vil si at atomiske hendelser legges i en hendelseskø, og eksekveres i den rekkefølgen de ligger i. At hendelsene er diskrete, betyr at mellom eksekveringene av dem skjer det ingenting. Alternativet ville vært å simulere med en kontinuerlig tidsskala. At en hendelse er atomisk, betyr at den starter og slutter på samme tidspunkt, slik at ingen andre hendelser kan oppstå i mellomtiden (dvs det finnes ingen mellomtid).

Når man utvikler en simuleringsmodell basert på en slik simulator, er det nødvendig å definere hvilke hendelser som skal kunne oppstå. En kontinuerlig hendelse fra den virkelige verden modelleres ved at den "oversettes" til et sett med diskrete hendelser. Et eksempel på en slik kontinuerlig hendelse, kan være at en ramme ankommer et buffer. Denne kan deles opp, slik at ankomsten av hvert bit i rammen tolkes som en atomisk hendelse. Hvis modellen ble laget på denne måten, ville den blitt nøyaktig, i den forstand at den etterlikner virkeligheten med høy grad av "finkornethet". Problemet med denne løsningen er at den medfører et stort antall hendelser som skal eksekveres, og simulatoren blir dermed lite effektiv. Hvis rammen var på 1Kbyte, vil altså 8000 hendelser måtte eksekveres. En langt mer effektiv løsning kan oppnås hvis man har muligheten til å si at mange av disse hendelsen er uinteressante, og at de dermed kan utelates. Poenget er å se bort fra de hendelsene som ikke forandrer tilstanden i noen simuleringsobjekter. I dette eksempelet vil bufferet være et simuleringsobjekt. Hvis en av tilstandsvariablene til bufferet er en teller som teller antall byte som til enhver tid ligger i bufferet, er det ikke noe poeng i å telle alle bit’ene. Det hadde da vært nok å ta med de hendelsene som representerer det første (og kanskje siste)



Figur 23: *Tilstandsdiagram for buffer-input*

bit'et i hvert byte, og antall hendelser ville blitt mindre. Hvis vi derimot kun er opptatt av at bufferet bare kan få inn én ramme om gangen, og ikke antall bit eller byte, vil bufferets tilstand være enten "klar til å motta", eller "ramme på vei inn" (Figur 23. Dette gjør at det er kun to av de 8000 hendelsene som er interessante:

- **H1:** Det første bitet i rammen ankommer bufferet
- **H2:** Det siste bitet i rammen ankommer bufferet

Ved å ignorere de hendelsene som ikke forandrer tilstanden til bufferet, har vi altså gjort simulatoren betydelig mer effektiv. Prisen for dette er mindre finkornethet, og dermed mindre nøyaktighet. Det kommer derfor an på hva man skal simulere, og hvilke tilstandsendringer som er viktige å få med seg, som avgjør hvilke hendelser som må modelleres.

4.1.2 Implementasjon av kjernen

Som nevnt blir hendelsene lagt i en hendelseskø. For å holde orden på denne køen, og dermed rekkefølgen hendelsene skal oppstå, benyttes en simulatorkjerne. Kjernen har en teller som representerer den globale tiden. I tillegg har den oversikt over en liste av hendelser som skal eksekveres på bestemte tidspunkt. Kjernens hovedoppgave blir dermed å inkrementere tiden, og starte opp hendelser på riktig tidspunkt.

Javasimulatoren som er brukt i dette tilfellet, har implementert denne funksjonaliteten på følgende måte: Alle simuleringsobjekter må være subklasse av klassen **Unit**. Denne klassen inneholder bl.a. annet en variabel "nextSchedule", som forteller når dette objektet skal vekkes opp, og en metode "act()", som skal inneholde koden som representer hendelsen. Kjernen implementeres nå med en prioritetskø av objekter av

klassen **Unit**, der det objektet som har den laveste verdien i nextSchedule til enhver tid ligger foran i køen. Dermed kan kjernen gå i en løkke og gjøre følgende:

- Hente objektet som ligger først ut av køen
- Sett den globale tiden lik objektet sin nextSchedule
- Kall objektet sin act()-metode
- Når act() returnerer, gå til begynnelsen av løkken

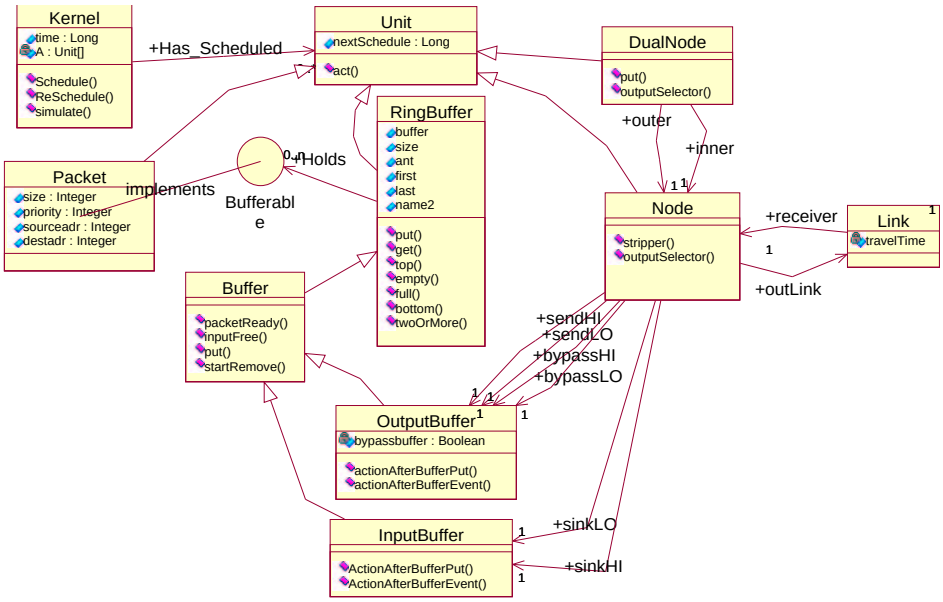
Det blir dermed objektene selv som innholder koden som skal representere en hendelse, kjernen holder kun orden på rekkefølgen de skal oppstå. Objektene må selv "melde fra" til kjernen om at de vil bli vekket opp på et senere tidspunkt. Dette gjøres ved å kalle en metode i kjernen "reschedule(Unit u, long tid)", som sørger for å legge objektet u inn på riktig plass i prioritetskøen ved å se på tiden som blir gitt som parameter. Tiden det tar å sette et objekt inn i en prioritetskø øker med O(logN), der N er antall elementer i køen. Dette er en av grunnene til at simulatoren blir tregere når finkornetheten er høy.

Hvis et objekt ønsker å implementere flere typer hendelser, må det selv holde orden på hvilke tidspunkt disse skal oppstå, og melde fra til kjernen om den tidligste av disse. Årsaken er at et objekt kan kun ligge på én plass i prioritetskøen av gangen. Et eksempel på dette er når et buffer både har en ramme på vei inn, og en annen ramme på vei ut av bufferet. Da vil det gjerne bli vekket opp både på det tidspunktet der den førstnevnte er helt inne, og der den sistnevnte er helt ute. Dette kan gjøre at at koden blir mer komplisert og det er større fare for feil. En mulig løsning på dette er å dele opp objektet i to, ett for bufferinngangen, og ett for bufferutgangen.

Konklusjonen herfra blir dermed at selv om finkornethet fører til en bedre imitasjon av virkeligheten og en "riktigere" simulering, lønner det seg å holde mengden definerte hendelser på et moderat nivå. Dette fordi simulatoren blir mer effektiv, og koden blir mer oversiktlig.

4.2 RPR-modellen

Dette avsnittet skal beskrive den opprinnelige simulatoren. Dette blir hovedsakelig en gjennomgang av hvilke klasser som inngår og hvilken funksjon de har. De to neste avsnittene vil henholdsvis vise hvordan RED og TCP er implementert som en utvidelse av den opprinnelige modellen. I tilleg til å implementere de to modulene, var det også nødvendig å gjøre noen endringer på den opprinnelige modellen. Detaljene rundt dette forklares i de neste avsnittene.



Figur 24: *UML klassediagram - Simulatoren før*

4.2.1 Klasser

Figur 24 viser en oversikt over de viktigste klassene som var implementert i simulatormodellen før TCP og RED ble lagt til. Denne kunne brukes til å sette opp en RPR-ring. Egne applikasjonsklasser ble brukt til å generere og sende data ut på ringen fra hver stasjon. Senere versjoner av denne modellen inneholder også en fairness-mekanisme (se seksjon 3.4.1.9). Under følger en forklaring på hvilken funksjonalitet de forskjellige klassene implementerer.

- **Kernel**
  Implementerer simulatorkjernen, som beskrevet i 4.1. Inneholder også en metode med navn “simulate”, som starter opp selve simuleringen.

- **Unit**
  Alle “aktive” klasser må være subklasse av denne, som beskrevet i 4.1.

- **Ringbuffer**
  Denne klassen implementerer et FIFO ringbuffer, som har et grensesnitt hvor det er mulig å legge inn pakker bakerst i bufferet og hente de ut foran.

- **Buffer**
  Subklasse av **Ringbuffer**, og legger til funksjonalitet som håndterer at pakker bruker tid på å flyttes inn og ut av bufferet. Bufferets act()-metode implenteres her.

- **OutputBuffer & InputBuffer**
  Transittbufferne og nodenes sendbuffere (se figur 25, er instanser av klassen **OutputBuffer**. Nodenes sinkbuffere (tar i mot pakker som er adressert til denne noden) er instanser av klassen **InputBuffer**. Buffere må utføre forskjellige handlinger når pakker legges inn eller taes ut, avhengig av hvor dette bufferet er plassert. Derfor har **OutputBuffer** og **InputBuffer** forskjellige implementasjoner av metodene actionAfterBufferPut(), og actionAfterBufferEvent().

- **Packet**
  Denne klassen implementerer grensesnittet (Java: interface) **BufferAble**, og skal modellere pakker. De har attributter som avsender og mottaker adresse (RPR-stasjons-id), størrelse og prioritet. Prioriteten kan være enten "høy" eller "lav".

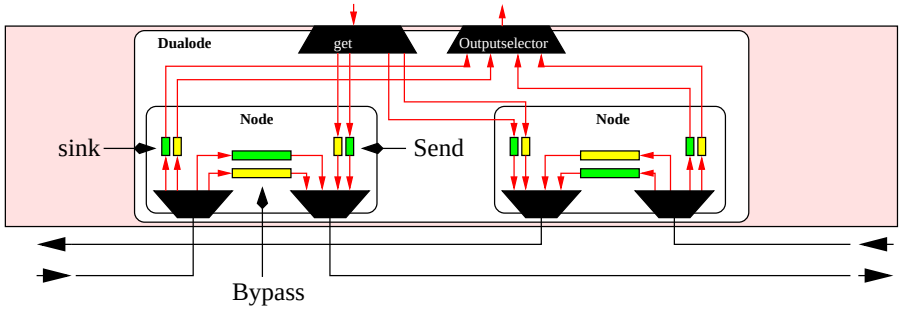
- **Node**
  Denne klassen modellerer en RPR-node. En RPR-stasjon har to noder, en på hver enkelt-ring (se 3.4). En node inneholder ett sinkbuffer, ett sendbuffer og ett transittbuffer for hver prioritet. I denne modellen er det kun to prioriteter, og dermed seks buffer i hver node (se figur 25). I tillegg inneholder noden en kontrollmekanisme som sørger for at riktig pakke blir sendt på riktig tid, i metoden outputSelector() (denne fairnessalgoritmen blir beskrevet straks).

- **DualNode**
  En "dualnode" modellerer en RPR-stasjon. Den inneholder altså to RPR-noder, og en mekanisme som bestemmer hvilken enkelt-ring en pakke fra denne stasjonen skal sendes ut på, basert på mottakeradressen i pakken.

- **Link**
  En link modellerer den fysiske forbindelsen mellom to RPR-noder, og har en forsinkelse som settes før simuleringen starter.

4.2.2 Forenklet fairness

RPR-Fairnessalgoritmen som er implementert i denne modellen er en forenklet utgave i forhold til RPR-standarden (fairnessalgoritmen bestemmer hvilket buffer som skal få lov til å sende den neste pakken ut på linken. I denne modellen gjøres det på denne måten: Det bufferet



Figur 25: *Skjematisk oversikt dataflyten over en RPR stasjon. Hver node har to transittbuffere (ett for hver prioritet), to sinkbuffere (ett for hver prioritet) og to sendbuffere (ett for hver prioritet)*

med den høyeste prioriteten som har en pakke å sende, får lov å sende. Bufferne er prioritert slik:

1 Høyprioritet i transitt.

2 Høyprioritet fra denne stasjonen

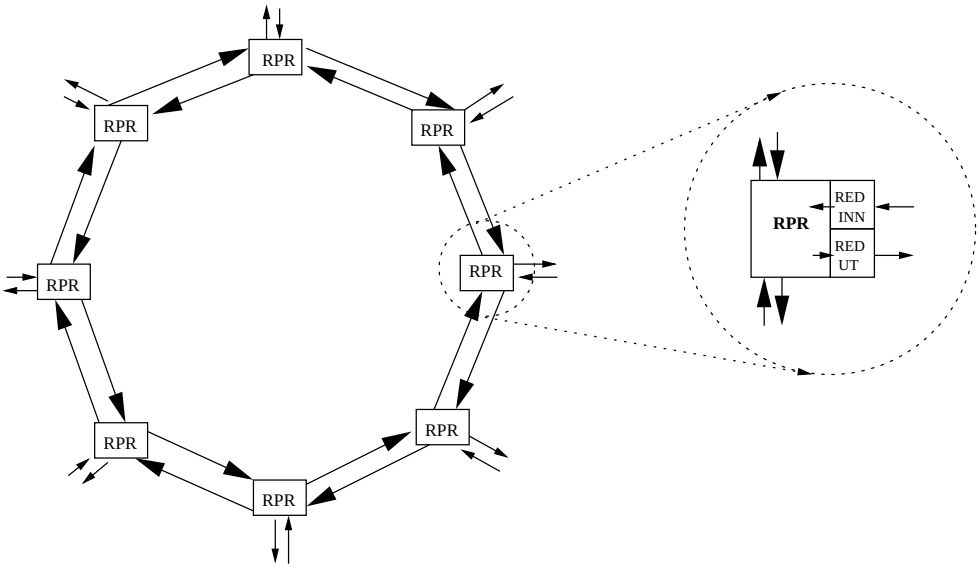
3 Lavprioritet i transitt

4 Lavprioritet fra denne stasjonen

Dvs at hvis det f.eks kommer en stor flyt høyprioritetspakker fra en stasjon oppstrøms, som skal videre til neste stasjon, vil alle de andre bufferne måtte vente til denne flyten er ferdig.

4.2.3 Bufferhastighet/bitrater

En pakke må gjennom en lang rekke buffere fra den genereres av en applikasjonsprosess til den mottas av en mottakerprosess. Dette er buffere som er lokalisert forskjellige steder, f.eks i maskinens internminne (RAM), eller på maskinvare spesielt utviklet for RPR. Dette fører naturligvis til at klokkehastigheten på bufferne vil være forskjellig, noe som fører til at pakkene bruker ulik tid på å kopieres inn og ut av de forskjellige minnebrikkene. Denne tiden spiller en rolle for den totale forsinkelsen av pakken, i tillegg til propagasjonsforsinkelsen på linken. Den opprinnelige simulatoren opererte med en felles klokkehastighet for alle bufferne. Dette kan forsvares med at alle bufferne var lokalisert i RPR-maskinvaren, og derfor var av samme type. (Dette stemmer imidlertid ikke i henhold til RPR-standarden, som åpner for at logikken i “MAC



Figur 26: *En RPR-ring med to RED-enheter på hver stasjon, en for innkommende trafikk, og en for utgående (Må ikke forveksles med RIO (RED with In Out))*

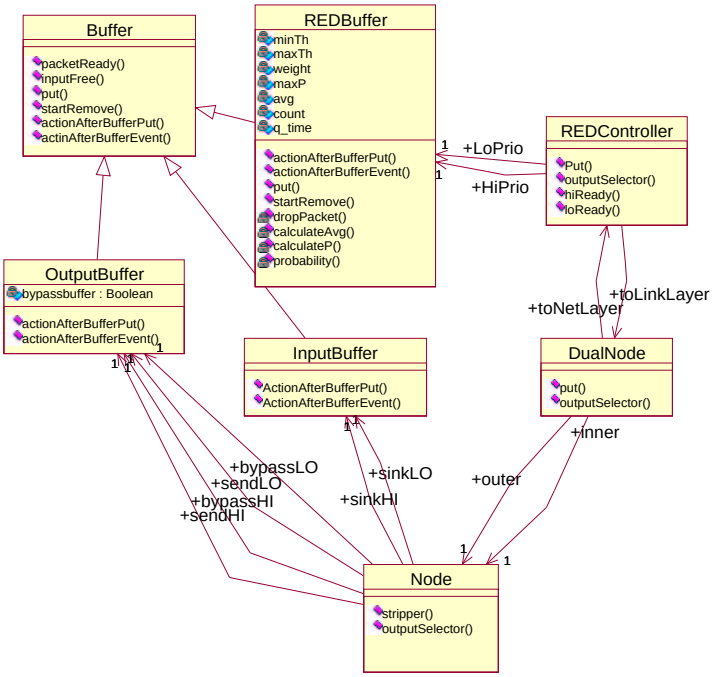
client"-laget kan være tregere enn logikken som sender data ut på linken. Et "stagebuffer" benyttes for å mellomlagre pakker slik at de kan sendes ut på linken med riktig hastighet).

Siden RPR og TCP implementeres på ulike steder i et system, vil bufringen foregå i forskjellige typer minnebrikker. I simuleringer der denne forskjellen er avgjørende for resultatet, bør bufferne modelleres med ulik klokkehastighet. Dette kan enkelt løses ved at klassen **Buffer** har en variabel som bestemmer bufferhastigheten, og som sendes med som en parameter til konstruktøren for denne klassen. Den opprinnelige modellen hadde en felles konstantverdi for alle buffere, definert i klassen Const.java (ikke beskrevet over).

4.3 RED-modellen

For å kunne kontrollere TCP-trafikken på ringen, var det ønskelig å bruke RED både for pakker som kommer inn, og for pakker som forlater ringen. Derfor var det nødvendig å ha to RED-enheter på hver RPR-stasjon. En som tar seg av innkommende trafikk, og en som tar seg av utgående trafikk (figur 26).

Som nevnt vil dette avsnittet forklare hvordan en RED-modul ble integrert i den opprinnelige simulatoren. I tillegg til å utvikle selve RED-funksjonaliteten, bestod dette arbeidet også i å tilpasse grensesnittet



Figur 27: *UML klassediagram - med RED utvidelsen*

mellom RPR-stasjonen på linklaget, og nettlaget der RED er implementert. Figur 27 viser hvordan de to nye klassene, **REDBuffer** og **REDController** er relatert til de opprinnelige klassene. De neste avsnittene gjennomgår disse klassenes funksjon og innhold.

4.3.1 RED-buffer

Teorien bak RED ble forklart i 3.5 og vil ikke bli gjentatt her. Hovedsakelig består en RED-implementasjon av tre deler:

- Beregning av gjennomsnittlig kølengde.
- Beregning av sannsynligheten p for merking, basert på den gjennomsnittlige kølengden.
- Avgjøre om en pakke skal merkes på bakgrunn av p.

Disse tre oppgavene skal utføres i det en pakke ankommer ruteren, og de gjøres i den nevnte rekkefølgen. I tillegg må tidspunktet ved start av en "idle"-periode (den perioden der bufferet er tomt) tas var på for å beregne den gjennomsnittlige kølengden ved slutten av perioden. Dette må gjøres i det den siste pakken forlater bufferet. Det var en triviell

```
public void put(Packet pk) {
    if (dropPacket()) {
        // dropping (ignoring) packet
        <Rapportering og statistikk>
    }
    else {
        // Do the putting
        super.put(pk);
    }
}
```

Figur 28: *Metoden put() i klassen REDBuffer - pseudokode*

oppgave å inkorporere disse funksjonene i modellen ved å opprette en subklasse av **Buffer** med navn **REDBuffer**, for så og redefinere [6] metodene som legger inn og tar ut en pakke av bufferet, henholdsvis "put()" og "startRemove(). Med trivielt menes at det kun var nødvendig å gjøre små endringer i koden arvet fra **Buffer**, i tillegg til å innføre funksjoner som gjør de nødvendige beregningene. Ingen nye hendelser ble innført. Metoden put() (figur 28)trengte kun en if-test på om pakken skal kastes eller ikke før den evt kaller på superklassen **Buffer** sin put()-metode. Utrykket if-testen evaluerer er en boolsk funksjon som sørger for at de tre oppgavene nevnt over utføres, og returnerer "sann" hvis pakken skal kastes. Metoden startRemove trengte kun å sjekke om den pakken som nå forlater buffer er den siste, og i så fall ta vare på tidspunktet i en global variabel (q_time).

RED algoritmen ble som nevnt implementert i funksjonen dropPacket() med tilhørende hjelpefunksjoner, og denne jobben var ikke "triviell". Avsnitt 4.3.3 beskriver disse funksjonene, men først gis et overblikk over grensesnittet mellom RED (nettverkslaget) og RPR (Linklaget), og hvilke nye klasser som ble innført.

4.3.2 Grensesnitt mellom RED og RPR

I seksjon 3.4.2 ble grensesnittet mellom det såkalte MAC-klientlaget og MAC-laget beskrevet. Der ble det forklart at MAC-klientlaget bl.a har som

[6] Å "redefinere" en metode (eng: "Override"), betyr i denne sammenhengen at hvis en klasse A er subklasse av klassen B, og A definerer en metode m(),med samme navn som en metode i B (også m()), vil As implementasjon eksekveres ved et kall på a.m(), der a er en instans av klassen A. Hvis man vil gjøre et kall på metoden deklarert i B, kan man i A gjøre et kall på super.m(). Dette gjøres i figur 28.

oppgave å tilpasse MAC-lagets grensesnitt til nettverkslaget. I denne simulatoren er det valgt å legge RED på nettverkslaget fordi det vanligvis er der det hører hjemme. Siden MAC-klientlaget ikke har noen funksjon i denne simulatoren, har det blitt utelatt, og nettverkslaget aksesserer dermed MAC-laget direkte. I praksis blir dermed nettverkslaget et klientlag for RPR. Figur 29 viser hvor RED kommer inn i modellen.

Grensesnittet mellom nettverkslaget og linklaget omfatter mekanismer som muliggjør sending av data og kontrollmeldinger begge veier mellom lagene. En slik mekanisme (også kalt "et primitiv") kan enten være et funksjonskall, eller et maskinvareavbrudd. Hvordan dette implementeres generelt, avhenger stort sett av hvor i protokollstakken vi befinner oss, og hvilket operativsystem som benyttes. Ved sending av pakker mellom de høyere lagene, det vil si fra nettverkslaget og oppover, brukes vanligvis funksjonskall . Implementasjonen av linklaget er som oftest lokalisert både i nettverkskortet (maskinvare) og i operativsystemet i form av en nettverksdriver (programvare). Grensesnittet mellom nettverkslaget og linklaget realiseres dermed som funksjonskall i driveren og i nettverksprotokollen(e).

Derfor vil all kommunikasjon mellom lagene i denne simulatoren foregå via funksjonskall. Så til det sentrale spørsmålet: Hvordan skal dette grensesnittet se ut? For å implementere dette må det avgjøres

- hvilke funksjonskall grensesnittet skal bestå av,
- hva de skal gjøre og
- når de skal kalles.

Siden RPR-standarden var på et tidlig stadium da disse valgene skulle gjøres, var det uklart hvordan grensesnittet skulle se ut i simulatoren. Målet har dermed vært å gjøre grensesnittet så enkelt som mulig uten at viktig funksjonalitet blir tilsidesatt. Det var også et mål at det skal være enkelt å benytte denne RED-Implementasjonen i andre simulatormodeller, uten å måtte forandre altfor mye på dem. I dette tilfellet var funksjonaliteten enkel.

Det skulle være mulig å sende høyprioritetspakker og lavprioritetspakker både “opp og ned” (figur 29), noe som skulle tilsi at fire primitiver var nødvendig, to på linklaget og to på nettlaget. For å gjøre grensesnittet så enkelt som mulig, ble det imidlertid valgt å kun ha ett primitiv på nettlaget, og ett på linklaget. Dette er mulig ved at begge lagene gjør demultipleksing på bakgrunn av pakkeprioriteten når de mottar en pakke. [7]

[7] Dette ville ikke uten videre la seg gjøre i virkeligheten, siden RPR-hodet (og dermed feltet som beskriver pakkens prioritet) strippes av før pakken sendes opp til nettlaget. Da måtte man enten benyttes et felt i IP-hodet, eller man kunne ha et primitiv for hver prioritet.

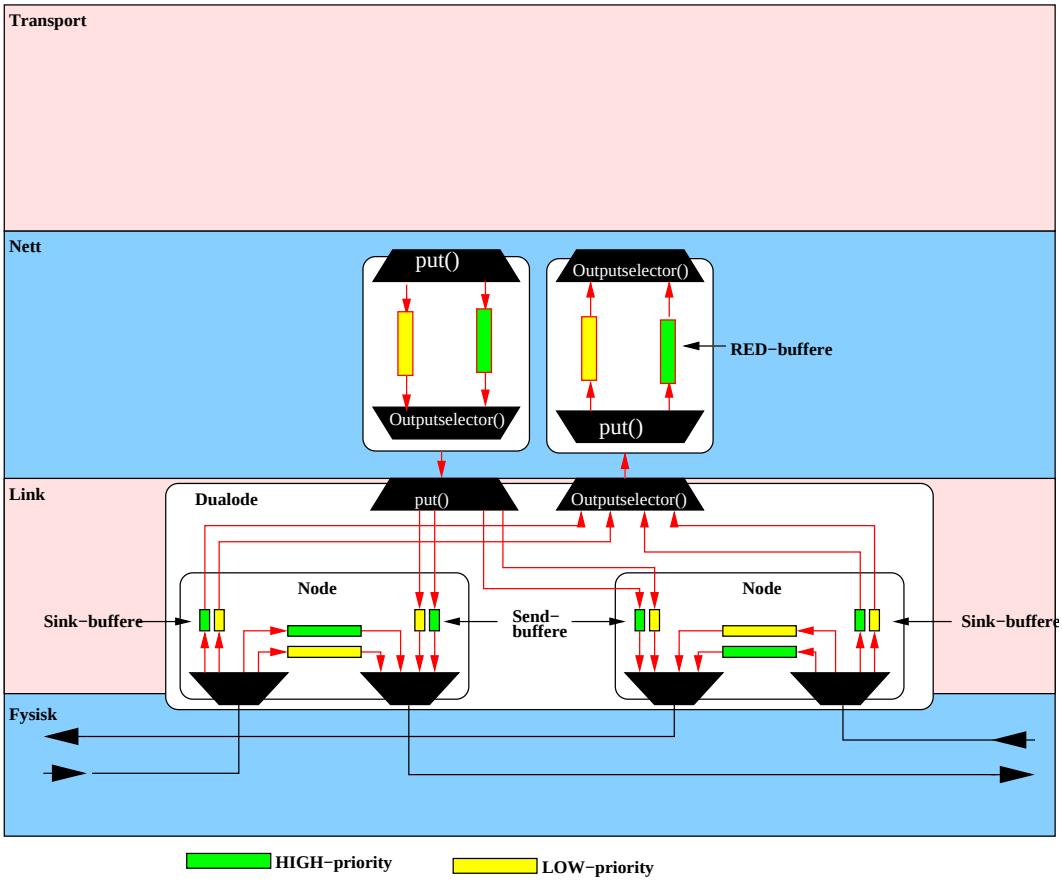
Som beskrevet i 3.4.2, er grensesnittet nå blitt spesifisert i standarden i form av to dataprimitiver og to kontrollprimitiver. Implementasjonen i simulatoren blir noe enklere bl.a siden fairnessmekanismen ikke er med. Dette gjør at klientlaget ikke trenger å bry seg om om det er "lov" å sende data, men kan sende så fort linken er ledig (det vil si at sendbufferne på figur 29 er ledig).

Nettlaget ble implementert med to identiske "RED-kontrollere" representert ved klassen **REDController**, der den ene kontrollerer innkommende trafikk, og den andre kontrollerer utgående. De inneholder to REDbuffere hver, en for høy og en for lavprioritetspakker (se figur 29). Det er primært bare lavprioritetspakker som skal merkes av RED-algoritmen, men det kan tenkes at man også vil merke høyprioritetspakker hvis de opptar for mye båndbredde. Da er det mulig å justere parameterne i de to bufferne forskjellig avhengig av hvor ofte man ønsker å merke pakkene (se seksjon 3.5).

De to neste avsnittene vil nå henholdsvis ta for seg implementasjonen av grensesnittet for sending av data fra RED til RPR, og fra RPR til RED.

4.3.2.1 Fra RED til RPR Primitivet for å sende en pakke fra nettlaget til linklaget er implementert som metoden put(Packet pk) i klassen **Dualnode**. For å sende fra linklaget til nettlaget benyttes metoden put(Packet pk) i klassen **REDController**. Den første var allerede implementert i den opprinnelige modellen, men trengte en liten forandring. Siden ut-bufferne på linklaget er relativt små (plass til kun to pakker), er sannsynligheten for at de er fulle når nettlaget vil sende en pakke ganske stor. Put()-metoden i **Dualnode** returnere derfor en boolsk verdi som forteller om pakken ble kopiert eller ikke. Hvis det siste er tilfellet, må kopieringen skje på et senere tidspunkt (siden det ikke er aktuelt å kaste pakken), når bufferet har en ledig plass. Dette løses ved at det kalles på en funksjon signalL3() nå en pakke har forlatt et av disse små utbufferne. Dermed trigges mekanismen som forsøker å sende en pakke fra nettlaget til linklaget (outputselector() i klassen **REDController**. En alternativ måte å gjøre dette på, som gjøres i moderne operativsystemer, er å la put-metoden i **Dualnode** være en blokkerende funksjon. Dvs at prosessen som gjør kallet (brukerprosessen) blir lagt i en ventetilstand av operativsystemet hvis bufferet er fullt. Put() vil dermed ikke returnere før det ble ledig plass i bufferet og kopieringen var utført. Dette ville innført unødvendig kompleksitet i modellen, som ville hatt lite å si for simuleringsresultatet.

4.3.2.2 Fra RPR til RED Når det gjelder mekanismen som muliggjør sending av pakker fra linklaget til nettverkslaget, dvs inn i en RED-



Figur 29: *Skjematisk oversikt over dataflyten i en RPR stasjon med RED-buffere på nettlaget*

kontroller, består den av to deler. Den ene er metoden i klassen **DualNode** outputSelector(), som bestemmer hvilket sink-buffer som skal få sende neste pakke. Den andre delen er to boolske funksjoner i klassen **REDController** som angir om det henholdsvis er klart å sende høy eller lavprioritetspakker inn i RED-bufferne. Disse er nødvendige fordi det kan ankomme to pakker på hver sin enkelt-ring tilnærmet samtidig, som begge skal inn i samme RED-buffer. Metoden outputselector() må som sagt velge hvilket buffer som skal få sende neste pakke opp til nettlaget. Dette gjøres med følgende algoritme:

- Ta vare på informasjon om hvilket høyprioritets og lavprioritets sinkbuffer som sist fikk sende en pakke.
- Hvis flere buffere har en pakke å sende på samme tidspunkt, prioriter i denne rekkefølgen:
  - Høyprioritet som ikke fikk sende sist.
  - Høyprioritet som fikk sende sist.
  - Lavprioritet som ikke fikk sende sist.
  - Lavprioritet som fikk sende sist.

På denne måten sikres det at høyprioritet alltid kommer foran lavprioritet, samtidig som det ikke diskrimineres mellom trafikk fra forskjellig enkelt-ring.
Et valg som er gjort i denne sammenhengen er å kun kopiere én pakke om gangen. Det kunne imidlertid være mulig å kopiere to pakker parallelt, så lenge de ikke har samme kildebuffer eller destinasjonsbuffer. Det vil si én høy og én lavprioritetspakke kan sendes samtidig. Effekten av dette vil i denne modellen være minimal, i og med at det uansett gis full prioritet til høyprioritetspakkene på vei ut av RED-kontrolleren igjen.

4.3.3 RED-funksjoner

Funksjonene som utgjør RED-implementasjonen beskrives nå i detalj, og noen problemer i forbindelse med dem diskuteres. Disse ligger, som nevnt i seksjon 4.3.1, i klassen **REDBuffer**, og er ikke synlige utenfor denne. Funksjonen dropPacket() implementer den overordnede mekanismen som tar bruk de øvrige funksjonene. Algoritmen er forklart i kapittel 3.5, og vil ikke bli gjentatt her. Implementasjonen er i stor grad basert på rådene gitt i [4].

4.3.3.1 dropPacket() Denne funksjonen kalles før en pakke skal kopieres inn i RED-bufferet, og returnerer 'sann' hvis denne pakken skal

kastes. Den sørger først for at den gjennomsnittlige kølengden beregnes, og bruker så denne til å avgjøre hvilken skjebne pakken skal få.

```
private boolean dropPacket() {
  calculateAvg();
  if<gjennomsnittlig kølengde ligger mellom minTh og maxTh>{
       P = calculateP();
       if (probability(P)) {
            <kast pakke>
       }
  }else{
       if<gjennomsnittlig kølengde er større enn maxTh> {
            <kast pakke>
       }else{
            <behold pakke>
       }
  }

}
```

Figur 30: *Metoden dropPacket() i klassen REDBuffer - pseudokode*

4.3.3.2 calculateAvg() Denne funksjonen beregner ny gjennomsnittlig kølengde "avg" basert på nåværende kølengde "curr_size" og den gamle verdien av avg. De to verdiene gis hver sin vekt, slik at spontane forandringer i curr_size har en begrenset effekt på avg (seksjon 3.5.2 forklarer hvordan dette "lavpassfilteret" fungerer). Java-koden som gjør denne beregningen ser slik ut:

```
avg = (1-weight)*avg + weight*curr_size;          (1)
```

Som forklart i 3.5.2, må avg beregnes noe annerledes hvis køen er tom når en pakke ankommer. Det gjøres da approksimasjon av hvor mange små pakker som kunne blitt behandlet av RED-portneren i den tiden køen har vært tom. Hvis vi kaller dette antallet "m", vil følgende javakode beregne det nye gjennomsnittet:

```
avg = power(1 - weight,m) * avg;                  (2)
```

Der funksjonen java.lang.Math.power(a,b) returnerer a^b. I denne sammenhengen var det nødvendig å gjøre en avskjæring for å effektivisere koden. Power-funksjonen bruker relativt lang tid på å beregne dette

produktet, spesielt for store eksponenter. m risikerer og bli stor (f.eks 31250 med en idle-periode på 10ms, en bitrate på 1Gbit/s og en pakkestørrelse på 40 byte), og for tilstrekkelig store verdier vil avg bli så nær null at det er unødvendig å gjøre beregningen. Det er derfor satt en maksimalverdi for m i simulatoren, slik at hvis m er større enn dette, blir avg satt til null uten å gjøre beregningen. Maksimalverdien som ble valgt var 1024. Bakgrunnen for dette er at weight er 0.002 (anbefalt i [25]), og ved å sette inn disse verdiene i (2), blir avg beregnet til 0.00026788. Å ha mindre avg enn dette gir ingen betydelige forandringer i simuleringsresultatene.

I virkelige RED-implementasjoner kan denne beregningen ,i følge [4], gjøres enda mer effektiv ved å ha en tabell hvor verdiene er regnet ut på forhånd.

4.3.3.3 calculateP() Returverdien fra denne funksjonen er sannsynligheten for at en pakke skal kastes. Den er avhengig av avg, maxP, minTh, maxTh og count (Se seksjon 3.5 for en forklaring på beregning av sannsynligheten). Først beregnes en sannsynlighet P1 slik:

```
P1 = (C1*avg) - C2;                                    (3)
```

Der C1 og C2 er konstanter (utledet av konstantene maxP, minTh og maxTh). Videre beregnes den endelige sannsynligheten P2 avhengig i verdien på count (antall pakker som har ankommet siden sist en pakke ble kastet):

```
P2 = P1/(1-count*P1);                                  (4)
```

Et problem i denne sammenhengen, er at (4) kan returnere negative og positive verdier utenfor intervallet [0,1], som er ulovlige verdier for sannsynligheten. For å forhindre at funksjonen returnerer en ulovlig verdi, ble returverdien satt til 1 hvis P2 lå utenfor intervallet [0,1]. At P2 settes til 1 dersom verdien var større enn 1, er åpenbart greit, men det er ikke like enkelt å se at det samme skal gjøres dersom verdien var mindre enn null. Siden P1 ikke kan være negativ, må årsaken til en negativ P2 være at count*P1 > 1. Videre betyr dette at count > 10, siden P1 < maxP, som er lik 0,1 [25]. Siden P1 nå er positiv og count er stor, blir det riktig å sette returverdien til 1.

4.3.3.4 probability() Dette er en enkel funksjon som returnerer den boolske verdien "sann" med en sannsynlighet P angitt som parameter. For å oppnå dette benyttes funksjonen java.util.Random.nextDouble() til å produsere et tilfeldig flyttall x uniform-fordelt over intervallet [0,1], og videre ved å returnere sann hvis x er mindre enn P.

4.4 Valg av TCP-mekanismer

Til nå er to av de tre delene av simulatoren beskrevet, RPR og RED-modellene, og det som nå gjenstår er TCP-modellen. I forhold til arbeidet med å lage RED, har TCP-modellen helt klart vært mest omfattende. Som nevnt i innledningen av oppgaven, var hovedmålet med dette arbeidet å utvikle en TCP-modul til RPR-simulatoren. Formålet er å benytte den til å observere hvordan RPR-ringen oppfører når "naturlig" TCP-trafikk sendes over den. I motsetning til RED, som er rimelig entydig beskrevet i faglitteraturen, består TCP av mangfoldige mekanismer, versjoner, implementasjoner og ikke minst spesifikasjoner. Kapittel 2 utdypet dette. En god del av tiden har derfor gått med til sette seg relativt godt inn i TCP-landskapet, for så å gjøre en del valg før selve programmeringen kunne starte. (Referanselisten bakerst i denne rapporten gir et godt bilde av forskjellen i arbeidsmengde mellom TCP og RED. Kun en av artiklene beskriver RED, mens ca 20 omhandler TCP).

Denne seksjonen beskriver først hvilke TCP-mekanismer som er implementert i simulatoren, og hva som er bakgrunnen for at nettopp disse er tatt med. Deretter kommer et avsnitt som lister opp de viktigste mekanismene som ikke er tatt med i modellen. Også her begrunnes valgene som er gjort, og konsekvensene som følger av å utelate dem analyseres. Den neste seksjonen (4.5) beskriver hvordan simulatoren er implementert, hvilke klasser som er lagt til osv.

4.4.1 Hvilke er implementert?

Her følger en liste over mekanismer som er implementert i simulatoren, og en beskrivelse av hvorfor de er tatt med. Det er viktig å poengtere at målet her ikke er å utvikle en TCP-versjon som er så effektiv som mulig, men en som kan generere TCP-trafikk som er mest lik den vi finner i dagens Internett.

- *Treveis håndtrykk (Three way handshake)*
  Denne algoritmen er implementert som beskrevet i [26], med unntak av håndtering av “simultaneous open”. Dette er situasjonen som oppstår når en forbindelse er i tilstanden SYN_SENT og mottar en SYN-melding fra motparten. Oppkoblingsmekanismen ble tatt med i simulatoren for å gjøre arbeidet med å sette opp forbindelser enklere. I den virkelige verden er dette helt nødvendig for å sette opp tilstanden hos de to kommuniserende partene. Parameterne det er snakk om her er TCP-portnummer, initielt sekvensnummer og annonsert vindusstørrelse (beskrevet i 3.3.4.1). I simulatoren hadde det imidlertid vært en mulighet å sette verdiene manuelt, ved å f.eks sende dem som parametere til konstruktøren for

objektene som representerer TCP-forbindelser (implementasjonen beskrives senere). Det ble allikevel valgt å ta med mekanismen, både siden det var en relativt enkel jobb, og fordi det gjør det enklere å sette opp simuleringer med TCP.

- *Glidende Vindu*
  Dette er grunnlaget for TCPs kommunikasjon, og det er derfor ikke noe spørsmål om denne skulle være med eller ikke. Her inngår også ende til ende flytkontroll. Glidende Vindu-algoritmen er implementert med sekvensnummerering av pakker, i motsetning til byte. Fordelen med dette, og årsaken til dette valget, er at koden blir vesentlig enklere. Prisen for dette er muligens en litt annen oppførsel i de tilfeller TCP retransmitterer pakker. I TCP, der bytenummerering benyttes, behøver ikke nødvendigvis retransmisjonene å inneholde nøyaktig de samme dataene som de originale segmentene. Når pakkenummerering brukes, vil derimot dette være tilfellet. I visse tilfeller kan dette gjøre retransmisjonene mindre effektive. Hvis f.eks to etterfølgende segmenter går tapt, kan det være mulig å retransmittere dem i samme pakke dersom bytenummerering benyttes. Trolig er det ikke mye å vinne på dette, i og med at MSS (Maximum Segment Size) i de fleste tilfeller vil tvinge senderen til å retransmittere dataene i to segmenter. Det er allikevel verdt å vurdere dette i simuleringer der dette kan ha innvirkning på resultatet (f.eks hvis mange små pakker sendes), og evt reimplementere Glidende Vindu-mekanismen slik at bytenummerering benyttes.
  En annen effekt av å benytte pakkenummerering er at sekvensnummerrommet ikke blir brukt opp like raskt. Dette vil eventuelt gi økt pålitelighet, men ikke få noen innvirkning på trafikkmønsteret.

- *Delayed ACKS*
  Denne mekanismen er implementert som spesifisert i [9]. Som i virkelige TCP-implementasjoner er det mulig å slå av denne mekanismen om ønskelig. Den fungerer slik at mottakeren kun sender ACK i følgende tilfeller:

  - Mottakeren har mulighet til å "piggyback'e" [8]en ACK på utgående datasegmenter.
  - Mottakeren har allerede en "ventende ACK". Effekten av dette er at kun mottak av annenhver pakke i en pakkestrøm genererer en ACK (denne ACK'en kvitterer dermed to segmenter).
  - ACK-timeren går av (satt til 200ms, se avsnitt 3.3.2.3).

[8]Å "piggyback'e" betyr å sende ACK-informasjonen i samme pakke som utgående data

- *Zero window probing*
  Denne mekanismen er implementert som beskrevet i [7], og ble tatt med for å unngå at forbindelser går i vranglås hvis mottakeren har fylt opp mottakerbufferet sitt (Se 3.3.2.1). Siden RFC 1122 [7] krever at TCP-implementasjoner benytter denne mekanismen, og at det ikke behøves kompliserte endringer i koden for å ta den med, var dette valget rimelig enkelt.

- *RTO (Retransmission Timeout)-beregning*
  Siden RFC 1122[7] sier: "*A host TCP MUST implement Karn's algorithm and Jacobson's algorithm for computing the retransmission timeout*", er dette gjort. Jacobson's algoritme [8] spesifiserer hvordan RTO skal beregnes på bakgrunn av innkommende RTT(Round Trip Time)-målinger samlet inn med Karn's algoritme[27]. Siden [8] i detalj tar for seg hvordan algoritmen kan implementeres, ble denne fulgt nærmest slavisk. Implementasjonen ble allikevel testet for å være sikker på at RTO ble beregnet fornuftig (se 5.1.3).

- *Slow start og Congestion Avoidance*
  Disse er forklart i 3.3.5 og er implementert som beskrevet i RFC 2581[28], som er en oppdatering av RFC 2001[11]. Alle TCP-implementasjonen må i følge RFC 2001 ha med disse.

- *Fast Retransmit og Fast Recovery*
  Som forklart i 3.3.6, finnes det tre versjoner av disse mekanismene: "Tahoe", "Reno" og "NewReno". De fleste TCP-implementasjoner som brukes i dag (det vil si Windows 98/ME, Windows NT SP5, Windows 2000/XP[29], Linux 2.2+ og 4.3BSD Reno Net/2+) benytter Reno eller NewReno-versjonen av Fast Recovery-mekanismen. I tillegg benytter de nyeste seg også av SACK(Selective Acknowledgements). Implementasjonen i denne simulatoren er av typen Reno. Det vil si at det ikke er gjort noen optimalisering for å unngå timeout ved tap av flere pakker fra samme vindu. Både NewReno og SACK-utvidelsen representerer løsninger på dette problemet. Grunnen til at Reno ble valgt fremfor Tahoe, er pga den store utbredelsen av Reno-implementasjoner. For å avgrense oppgaven ble ikke NewReno eller SACK tatt med, selv om det var ønskelig. Dette diskuteres mer i neste avsnitt.

4.4.2 Hvilke er ikke implementert?

Pga tidsbegrensninger er ikke alle mekanismer tatt med i simulatoren, selv om det hadde vært ønskelig å ha de med. Disse er hovedsakelig utvidelser som gir TCP ulike ytelsesforbedringer, men som ikke er blant

standardmekanismene. Spesielt fokuseres det i denne seksjonen på utvidelsene som øker ytelsen til TCP-forbindelser som kommuniserer over linker med et stort forsinkelse*båndbredde-produkt (heretter forkortet FBP). Big Windows, PAWS, SACK og NewReno, er alle utvidelse som er aktuelle i denne sammenhengen. Først argumenteres det for at hvis de to kommuniserende partene i en TCP forbindelse er direkte knyttet til en RPR-ring, vil FBP kunne bli "stort", og at Big Windows-utvidelsen er aktuell. Videre diskuteres behovet for de øvrige utvidelsene, utifra at FBP er stort. Det taes både stilling til om utvidelsene bør benyttes i virkeligheten, og om de bør implementeres i simulatoren. For å avgjøre det sistnevnte, analyseres bruken av mekanismene i nåtidens mest brukte TCP-implementasjoner. Målet med simulatoren er altså ikke å lage utvikle en implementasjon med best mulig ytelse over RPR, men en som kan generere TCP-trafikk som likner mest mulig på den vi finner i dagens Internett.

4.4.2.1 Big Windows-utvidelsen. Som nevnt i 3.3, er denne utvidelsen aktuell å bruke når FBP blir stort. Dette produktet er et mål på hvor mye data det er plass til i "røret" mellom sender og mottaker, og følgelig hvor stor bufferplass som kreves for å ta vare på ukvittert data. Siden **ADVERTISED WINDOW**-feltet i TCP-hodet har en maksverdi på 65K, og fungerer begrensende på senderens mengde av utestående data, får TCP problemer med utnytte båndbredden når FBP er av en størrelsesorden 10^6 og oppover (65Kbyte= $5.2*10^5$ bit). Dette er tilfellet når den underliggende forbindelsen mellom de to partene enten har en betydelig høy forsinkelse (f.eks satellittlinker), eller en høy båndbredde. RPR-nettverk er av den siste typen. Nærmere bestemt er det planlagt at RPR skal støtte båndbredder på 655Mb/s, 2.5Gb/s, 10Gb/s og 40Gb/s. Derfor er det naturlig å stille spørsmål om det er nødvendig for TCP-forbindelser å benytte Big Window-utvidelsen når de kommuniserer over RPR. Ved den laveste båndbredden, 655Mbit/s, overstiger FBP 65K dersom forsinkelsen kommer opp i 0.8ms. Dvs at hvis to kommuniserende parter har en forsinkelse mellom seg på mindre enn 0.8ms og båndbredden er 655Mbit/s, er det ikke behov for å benytte utvidelsen. Tabell 1 viser den maksimale forsinkelsen for de aktuelle båndbreddene, slik at produktet mellom de er tilnærmet lik 65Kbyte. For at forsinkelsen skal være så lav som dette, skal ikke pakkene passere mange rutere på veien. Er de imidlertid begge koblet direkte til RPR-ringen, er muligheten tilstede. Forsinkelsen er da avhengig av disse faktorene:

- Avstand mellom endepunktene.
- Det fysiske mediets propagasjonshastighet (ca $2*10^8$m/s i optisk fiber).

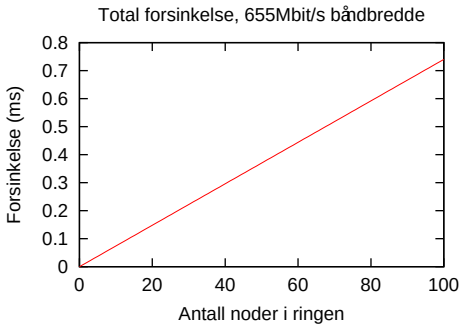
Båndbredde (Mbit/s)	Maks forsinkelse (ms)
655	0.8
2500	0.21
10000	0.05
40000	0.01

Tabell 1: *Maksimale forsinkelser for gitte båndbredder slik at produktet er tilnærmet lik 65Kbyte*

- Antall noder mellom endepunktene.
- Trafikkmengde.
- Pakkeprioritet (QoS).
- Fairnessalgoritme.

Det er derfor vanskelig å si noe sikkert om hvor stor forsinkelsen vil være i denne sammenhengen, uten å gjøre simuleringer. Desverre var det pga tidsbegrensninger ikke mulig å gjennomføre simuleringer for å anslå dette. Det er imidlertid mulig å beregne en nedre grense for forsinkelsen, gitt avstanden mellom nodene, propagasjonshastigheten, antall noder på ringen og båndbredden. Figur 31 viser en slik beregning der båndbredden er satt til 655Mbit/s. Denne båndbredden er valgt fordi den er mest aktuell når RPR benyttes i lokalnettsammenheng, altså der endesystemene er tilknyttet samme RPR-ring. I beregningen forutsettes at det ikke går noen bakgrunnstrafikk på ringen, slik at pakkene videresendes med en gang de er ankommet i sin helhet ("store $ forward"). Avstanden mellom nodene har liten betydning i og med at propagasjonshastigheten er så høy som $2*10^8$m/s. Pakkestørrelsen er satt til 1200 byte. Grafen forteller at hvis antall noder på ringen overstiger 100, vil den totale forsinkelsen være nedad begrenset til 0.8ms, som også var den maksimale forsinkelsen i figur 1. Dette viser spesielt at hvis antall noder nærmer seg 100, vil FBP bli så stort at Big Windows må benyttes for å kunne utnytte den tilgjengelige båndbredden. Generelt kan vi si at siden køing i nodene vil øke forsinkelsen betraktelig, Vil det alltid være aktuelt å benytte utvidelsen når RPR benyttes i lokalnett med en båndbredde på 655Mbit/s. De samme beregningene ble gjort for de andre båndbreddene, og de viste den samme tendensen. For disse blir naturlig nok den totale forsinkelsen lavere, men siden maksforsinkelsen i tabell 1 var lavere, vil det også her være nødvendig å bruke Big Window-utvidelsen.

Til nå har vi kun sett på endesystemer som er direkte knyttet til samme RPR-ring, eventuelt to forskjellige ringer med bro i mellom. Hva



Figur 31: *Nedre grense for forsinkelse i en RPR ring. Pakkene vil ikke passere mer enn halvparten av antall noder i ringen. Grafen representerer likningen f(n)=n/2*(k+p), der k=14,7µs er køforsinkelsen i hver node når pakkestørrelsen er 1200 byte (store&forward), og p=0.1µs er propagasjonsforsinkelsen mellom to noder.*

så med systemer som er tilkoblet via linker med lavere båndbredde? Vil de behøve Big Window-utvidelsen? Dette blir et generelt spørsmål som ikke er spesielt for RPR. Vurderinger må her gjøres som for alle andre TCP-forbindelser i Internett.

Konklusjonen fra denne diskusjonen er altså at Big Windows bør benyttes hvis to TCP-endesystemer er direkte tilkoblet en RPR-ring. Neste spørsmål er om denne utvidelsen burde vært implementert i simulatoren for å kunne produsere realistiske resultater? Svaret er nei, og grunnen er at den begrensningen TCP har med hensyn på at **ADVERTISED WINDOW**-feltet må være på 16 bit, ikke eksisterer i simulatoren. Derimot brukes en vanlig 32-bits integer til å representere dette feltet i pakkene. Dermed kan vi egentlig si at utvidelsen er implisitt implementert. Det man imidlertid må passe på, er at sender og mottaker har store nok TCP-buffere. Disse bestemmes "manuelt" når scenarioet settes opp, dvs før kompilering.

4.4.2.2 PAWS (Protection Against Wrapped Sequence numbers). Denne opsjonen må benyttes når Big Windows brukes, for å skille gamle segmenter fra nye med samme sekvensnummer (se 3.3.7). Siden simulatoren tillater store vinduer (forklart i forrige punkt), burde denne opsjonen vært implementert, men måtte dessverre prioriteres bort pga tidsbegrensninger. Det er imidlertid kun små endringer som må gjøres på mottakersiden for å innføre denne mekanismen. Seksjon 4.2.1 i RFC 1323[18] beskriver disse endringene. De innebærer hovedsakelig at det innføres en ny variabel, "TS.Recent", i tilstandsblokken for forbindelsene, som lagrer det sist mottatte tidstempelet. Hver gang et segment ankom-

mer, sjekkes tidsstempelet i pakkehodet mot TS.Recent, og forkastes hvis pakken har et lavere tidsstempel enn dette. Tidsstempel-utvidelsen er allerede implementert, og brukes til beregning av RTO (Retransmission Timeout) (seksjon 3.3.2.2). En skulle tro at fraværet av PAWS gjør TCP-implementasjonen upålitelig, siden sekvensnummer-rommet vil kunne gjenbrukes med korte tidsintervaller. Dette er i og for seg riktig, men "heldigvis" mangler Glidende Vindu implementasjonen støtte for at sekvensnumre kan starte på null. Denne mangelen beskrives i 4.6.

4.4.2.3 SACK-utvidelsen[19]. Som nevnt i 3.3, er dette en mekanisme som forbedrer TCPs ytelse under fast recovery ved at senderen har bedre oversikt over hvilke segmenter som er kommet frem til mottakeren og kan dermed re-sende færre segmenter. Videre ble det forklart at SACK er spesielt nyttig for forbindelser med et stort FBP (forsinkelse*båndbreddeprodukt), der det er større sannsynlighet for at flere enn en pakke går tapt innenfor ett vindu med data. Tidligere i dette avsnittet ble det konkludert med at hvis begge parter i en TCP-forbindelse er tilkoblet samme RPR-ring, kan FBP bli stort, og følgelig vil SACK-utvidelsen være aktuell. Problemet med Reno TCP (som er implementert her), er at den yter dårlig når flere pakker tapes fra ett vindu. Dette kom også frem i simuleringene som blir presentert senere i denne rapporten, ved at senderen ofte får en timeout (se tabell 3). Siden de aller fleste moderne TCP-implementasjoner inkluderer SACK, burde simulatoren absolutt ha inneholdt denne utvidelsen. Også her var det tiden som begrenset arbeidet. SACK krever at både sender og mottakersiden modifiseres, og innebærer vesentlige forandringer på sendersiden når det gjelder retransmisjoner. Fraværet av PAWS vil ikke ha noen innvirkning på sendemønsteret.

4.4.2.4 The NewReno extensions. Som nevnt i forrige avsnitt, er TCP-versjonen i denne simulatoren av typen "Reno". NewReno[13] tilfører to optimaliseringer til Reno, som beskrevet i 3.3.6:

- Øker slow starts initielle verdi på metningsvinduet til 2*MSS.
- Forbedrer Fast Recovery mekansismen, slik at ytelsen økes ved tap av flere pakker fra samme vindu. Utnytter "delvise kvitteringer" for å få til dette.

Et spørsmål som kommer frem i denne sammenhengen er om det er nødvendig å både implementere SACK og NewReno, siden de begge løser problemet med tap av flere pakker fra samme vindu. Simuleringene i [20] viser at NewReno klarer seg bra når opp til tre-fire pakker blir borte, men dersom dette antallet øker ytterligere, er SACK en klar vinner. Dermed antydes det at det ikke er noe poeng implementere NewReno hvis

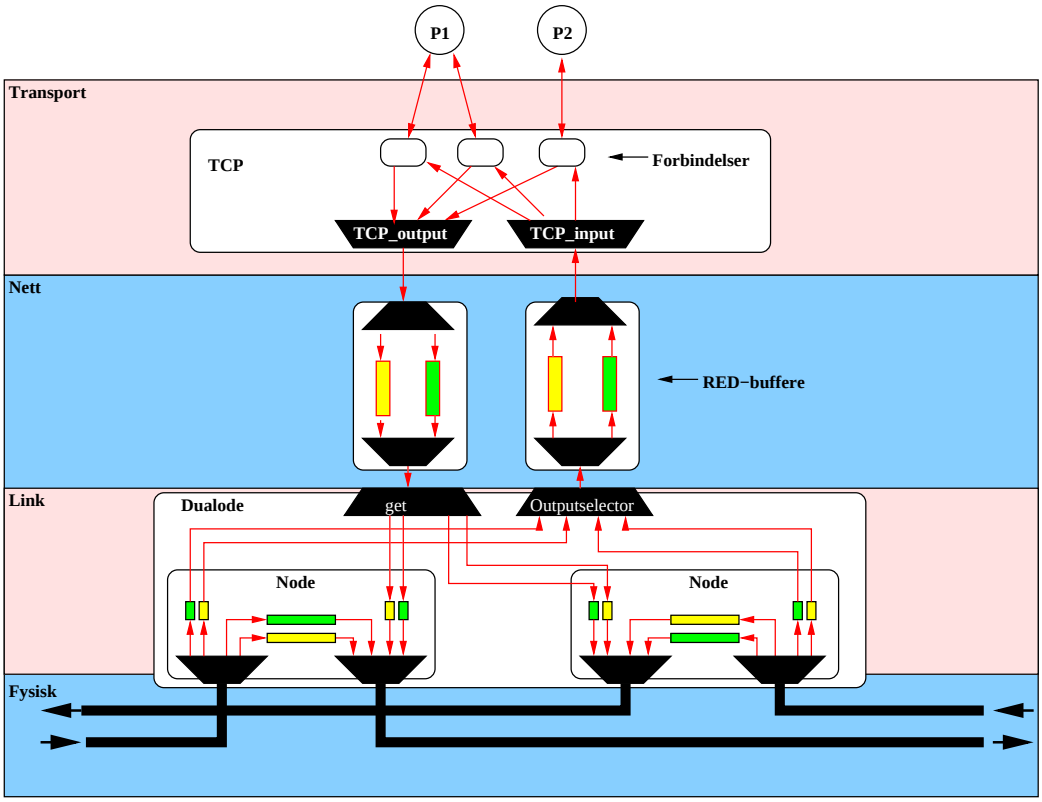
SACK brukes. Det kan f.eks være mulig å implementere SACK, og tillegg øke den initielle verdien på cnwnd i Slow Start til 2*MSS. Dette er den optimale løsningen dersom man vil ha en versjon som er mest lik de som benyttes i virkeligheten. [29] sier at Windows 2000/XP, Windows 98, og til og med nyere versjoner av Windows 95 har gjort akkurat dette. I følge [30] har SACK og NewReno vært implementert i Linux-kjernen siden versjon 2.2. En annen mulighet, er å kun implementere NewReno, siden dette medfører adskillig mindre endringer i koden enn innføringen av SACK.

4.4.2.5 TCP Vegas. Som nevnt i 3.3.6, er Vegas benyttet i langt mindre grad enn Reno. Det er grunnen til at denne versjonen ikke ble valgt.

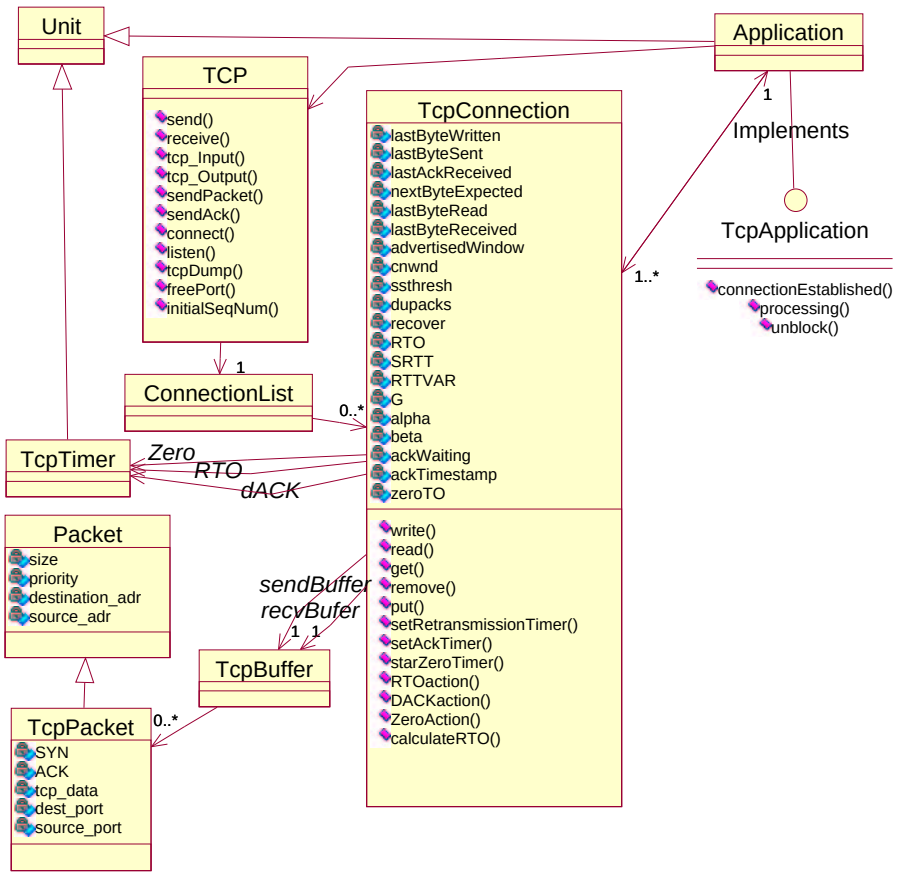
4.4.2.6 Lukking av forbindelse. Funksjonalitet for å lukke en forbindelse er ikke implementert. For å gjøre dette måtte seks nye tilstander blitt innført: FIN_WAIT_1, FIN_WAIT_2, CLOSING, TIME_WAIT, CLOSE_WAIT og LAST_ACK. I tillegg måtte FIN-flagget innføres i TCP-pakken. Fraværet av denne mekanismen gjør at en forbindelse som opprettes vil leve helt til simuleringens slutt. Dette betyr ikke at en forbindelse sender pakker hele tiden, men at det holdes tilstand i endenoden under hele simuleringen. Det er opp til applikasjonslaget hvor mye data som skal sendes. Følgen av dette er at mye minne blir unødvendig brukt opp, særlig hvis det simuleres med et stort antall TCP-forbindelser. Nå skal det sies at simulatoren generelt sett bruker lite minne, så dette vil antagelig ikke være noe problem.

4.5 Implementasjon av TCP

Etter å ha sett hvilke mekanismer TCP skal inneholde, er det nå på tide å forklare hvordan de er implementert, og hvordan TCP-modulen passer inn i simulatoren. Implementasjonen av denne modulen innebærer ikke bare kodingen av TCP, men også utviklingen av et enkelt grensesnitt mot applikasjoner som skal benytte TCP. Figur 32 beskriver hvordan de forskjellige modulene i simulatoren forholder seg til hverandre. Pilene representerer dataflyten. På hver node er det ett TCP-objekt, og ett objekt som representerer hver TCP-forbindelse. Forbindelses-objektet inneholder all tilstand tilknyttet en forbindelse og identifiseres med et portnummer. Disse objektene representer det som kalles "kontrollblokker" i ordinære TCP-implementasjoner. TCP-objektets hovedfunksjon er å sende og motta TCP-pakker til og fra nettverkslaget. I tillegg finnes det noen kontrollprimitiver som ikke vises på figuren, men som blir forklart i avsnittene som beskriver grensesnittene.



Figur 32: *Skjematisk oversikt over dataflyten i en RPR stasjon med RED-buffere på nettlaget, og TCP på transportlaget*



Figur 33: *UML klassediagram - med TCP utvidelsen*

Først beskrives altså de nye klassene som TCP-modulen består av og hvilke oppgaver de har. Deretter, i seksjon 4.5.2, ser vi på grensesnittet mellom RED og TCP. Seksjon 4.5.3 beskriver så grensesnittet mellom TCP og applikasjonene, og til slutt følger en detaljert forklaring over de viktigste funksjonene i TCP. Det er her vi vil se hvordan de forskjellige mekanismene er implementert. Underveis presenteres problemstillinger som har dukket opp under implementasjonen, og hvilke valg som er blitt gjort.

4.5.1 Klasser

Forrige avsnitt (og figur 32) forklarte at det på hver node er ett TCP-objekt, og ett objekt for hver forbindelse. Vi skal nå se nærmere på hvilke klasser de realiserer, og hvilke oppgaver de har. I tillegg til disse to klassene, består TCP-modulen også av andre klasser. Figur 33 viser et

UML klassediagram for TCP-modulen. De to viktigste klassene er **TCP** og **TcpConnection**. Her ser vi at sistnevnte inneholder all tilstanden til en forbindelse, men også noen metoder. Dette skiller seg fra virkelige implementasjoner, der kontrollblokken kun inneholder variable. Årsaken til dette er at denne simulatoren er programmert i Java, mens ordinære implementasjoner som oftest er utviklet i programmeringsspråket C. Siden Java er et objektorientert språk, og simulatoren ikke stiller like strenge krav til effektivitet som vanlige implementasjoner, er det valgt å legge en del metoder inn i forbindelses-klassen. Dette er metoder som endrer og returnerer tilstanden til en forbindelse. I **TCP** klassen derimot, ligger metoder som har med sending og mottak av pakker fra nettverkslaget, i tillegg til funksjoner som er uavhengig av forbindelsene. De viktigste av disse metodene beskrives i avsnitt 4.5.4. Her følger derimot en oversikt over alle klassene:

- **TCP.** Dette er "TCP-prosessen" som kjører på hver node. Dens hovedoppgave er å sende og motta pakker fra nettverkslaget. Ved mottak av en pakke, demultiplekses denne med hensyn på portnummeret i pakkehodet (variabelen dest_port i klassen **TcpPacket**), slik at pakken blir levert til riktig forbindelsesobjekt. Ved sending av data sørges det for at regler for flytkontroll og metningskontroll følges. Denne klassen inneholder også metoder som realisere applikasjonenes grensesnitt for å opprette forbindelser. Disse metodene er listen() og connect(), som henholdsvis implementerer en "passive open" og en "active open"[9].

- **TcpConnection.** Som nevnt inneholder denne klassen variable og metoder knyttet til en TCP-forbindelses tilstand. I tillegg har hver forbindelse ett mottaks-buffer og ett send-buffer. Disse bufferne er representert med klassen **TCPBuffer**. Forbindelsene er også assosiert med hver sin instans av tre timer-objekter (klassen **TcpTimer**). Disse tre er:

  - *RTO-timeren.* Sørger for at en forbindelse timer ut når kvitteringer uteblir
  - *dACK (delayed ACK)-timeren.* Sørger for at en kvittering ikke blir forsinket mer en 200ms hvis "delayed ACKs" er i bruk.
  - *Zero-timeren.* Sørger for at senderen sender "probe-meldinger" når mottakeren har annonsert en vindusstørrelse på null. ("Zero Window Probing", avsnitt 3.3.2.1).

[9]En passive open (tjenersiden) betyr at forbindelsen legger seg til å lytte på en bestemt port. En active open (klientsiden) betyr at en forbindelse sender en SYN melding til en port på en maskin der en applikasjon allerede har gjort en passive open.

I avsnitt 3.3.4 ble det forklart at en TCP-forbindelse til enhver tid befinner seg i en av de 11 tilstandene beskrevet i figur 11 på side 24. Siden nedkobling ikke er implementert, kan en TCP-forbindelse i denne modellen kun befinne seg i en av de 4 tilstandene:

  - LISTEN
  - SYN_RCVD
  - SYN_SENT
  - ESTABLISHED

- **ConnectionList.** Dette er en klasse som inneholder en lenket liste med TCP-forbindelser, og et sett metoder for å søke etter, legge inn og ta ut slike objekter fra listen. Hvert objekt av klassen **TCP** har en slik liste over alle TCP-forbindelser på en node.

- **TcpTimer.** Dette er som nevnt en timer, og må arve fra klassen Unit i simulatoren (se avsnitt 4.2) siden det er et "aktivt" objekt. Et objekt av denne klassen kan enten være av en de tre typene nevnt over. Denne typen bestemmer hvilken metode som skal kalles i forbindelsesobjektet når timeren går av. Årsaken til at det ble valgt å ha egne objekter for å representere timere, var for å forenkle **TcpConnection**-klassen. Som diskutert i 4.1, er et "aktivt objekt" nødt til å selv holde orden på sine egne hendelser. I steden for at forbindelses-objektet fikk tre hendelser å ta vare på, har timer-objekter kun én.

- **TcpPacket.** Denne klassen er en enkel utvidelse av klassen Packet, ved at den tilfører en TCP-header til pakken. Feltene i dette hodet er:

  - SYN-flagg.
  - ACK-flagg.
  - PUSH-flagg. (Brukes for å si til mottakeren at en kvittering ikke skal forsinkes (delayed acks) i Slow Start.
  - Avsenders portnummer.
  - Mottakers portnummer.
  - Data-flagg (Dette flagget er satt hvis pakken inneholder data).

- **TcpBuffer.** Forbindelsenes send og mottaks-buffere representeres ved denne klassen. Det er et buffer som innholder objekter av typen **TcpPacket**. Det spesielle med dette bufferet er at det indekseres med TCPs sekvensnumre. Operasjonene for å søke etter og hente ut pakker krever dermed et sekvensnummer. På denne måten

blir forbindelsesobjektet skjermet fra mappingen mellom sekvensnummer og den "fysiske" plassen i bufferet. En intern funksjon i **TcpBuffer** tar seg av denne oversettingen, og er bassert på modulo-operatoren.

- **TcpApplication.** Dette er ikke en klasse, men et "Java Interface" (grensesnitt). Den definerer et sett med funksjoner en applikasjonsklasse må inneholde for å kunne benytte seg av TCP. Dette er nødvendig fordi TCP er nødt til å kalle metoder i applikasjonsobjektet. Dette forklares nærmere i avsnitt 4.5.3.

De to neste avsnittene vil å forklare hvordan TCP-modulen kommuniserer med de to andre modulene.

4.5.2 Grensesnitt mellom RED og TCP

Dette avsnittet tar for seg hvordan pakker sendes mellom RED og TCP. I avsnitt 4.3.2 ble grensesnittet mellom RED og RPR beskrevet, og grensesnittet mellom RED og TCP har mye til felles med dette. Grunnen er at nettverkslaget består av to identiske RED-kontrollere som tilbyr det samme grensesnittet for å sende pakker inn den (figur 32). For å sende en pakke til nettverkslaget må derfor TCP-objektet ha en referanse til den riktige RED-kontrolleren, og kalle denne sin put()-metode. Hvis RED-bufferet er fullt, blir pakken kastet, og vil etter hvert bli retransmittert av TCP. Da dette skulle implementeres, oppstod det et problem. Grunnen var at det tok tid å sende en pakke inn i et buffer, og dette blir komplisert fordi de forskjellige TCP-forbindelsene konkurrerer om det samme RED-bufferet. Hvis to eller flere forbindelser forsøker å sende en pakke tilnærmet samtidig, må alle utenom én vente. To mulige løsninger ble vurdert i denne sammenhengen, en enkel, og en mer komplisert. Den enkle ble valgt. Den gikk ut på å se på det at en pakke sendes fra TCP til RED-bufferet var en atomær hendelse. Selv om dette ikke gjenspeiler virkeligheten hundre prosent, er det ikke langt fra sannheten. Denne operasjonen dreier seg stort sett ikke om annet enn å sende en peker ned til nettverkslaget, noe som normalt gjøres over kun få klokkesykler. Den andre løsningen, ville vært å legge mer funksjonalitet i TCP, slik at pakker som skulle sendes ned til nettverkslaget blir lagt i en egen kø. Denne køen måtte være felles for alle forbindelser på maskinen. Denne løsningen og følgene av dette valget diskuteres nærmere i avsnitt 4.6.

Når det gjelder sending av data den andre veien, foregår dette på samme måte. RED-kontrolleren må nødvendigvis ha en referanse til TCP-objektet.

4.5.3 Grensesnitt mellom TCP og Applikasjoner

Applikasjonene er tegnet som prosess P1 og P2 i figur 32. Dette grensesnittet er noe mer omfattende enn det som ble beskrevet i forrige avsnitt. Det består av to hoveddeler: Sending av data, og oppkobling av forbindelse. Grunnen til at det er nedlagt forholdsvis mye arbeid i utviklingen av dette grensesnittet, var for å gjøre arbeidet med å lage nye applikasjonsklasser enklere for kommende brukere av simulatoren.

4.5.3.1 Sending av data mellom TCP og applikasjonen foregår slik at det er applikasjonen som i begge tilfeller er den aktive. Applikasjonsobjektet har en referanse til et (eller flere) forbindelsesobjekt(er), og kaller på metodene read() og write() i dette objektet for å lese og skrive data. Dette ser i og for seg greit ut, men det er to ting som kompliserer dette:

- En applikasjon prøver å lese fra en tom kø.
- En applikasjon prøver å skrive til en full kø.

Moderne operativsystemer løser dette ved å blokkere applikasjonsprosessen, og ikke starte opp eksekveringene igjen før blokkeringsbetingelsen er tilfredsstilt (køen er ikke lenger tom eller full). Det er dette som gjør at flytkontrollen fungerer mellom prosessene i begge ender av forbindelsen. For å få til noe liknende i simulatoren, er følgende gjort:

- Metodene read() og write() i klassen **TcpConnection** gir en returverdi til applikasjonen som angir om operasjonen var vellykket eller ikke. Det blir dermed opp til applikasjonen å blokkere seg selv hvis bufferet enten var tomt eller fullt. At applikasjonen blokkerer betyr i praksis at den ikke ber simulatoren om å vekke den opp på et senere tidspunkt.
- Hver gang en pakke tas ut av et TCP-buffer, eller legges inn, kaller TCP på en metode i applikasjonsobjektet. Dette er metoden "unblock()" i **TcpApplication.**-grensesnittet. Denne tar en parameter som angir om dette gjelder en "leseblokkering" eller en "skriveblokkering". Applikasjonsklassen må derfor implementere denne metoden, som da skal få applikasjon til å enten begynne å lese eller skrive data igjen.

Dette blir dermed en form for synkroniseringsløsning for et produsent-konsument problem.

4.5.3.2 Oppkobling av forbindelser er den andre delen av grensesnittet mellom TCP og applikasjonen. Denne implementasjonen kan sees

på som en meget forenklet versjon av Socket-grensesnittet som brukes i Unix. Tjener-siden av forbindelsen gjør først en passive open ved å kalle på metoden listen() i klassen **TCP**. Denne tar et portnummer som paramerer, og resulterer i at det opprettes et forbindelsesobjekt med tilstanden LISTEN, som "lytter" på den angitte porten. Klient-siden gjør tilsvarende en active open ved å kalle på connect(). connect() tar mottakeradresse og portnummer som parameter, og sørger for at det sendes en SYN-melding til riktig port på riktig maskin. Deretter opprettes et forbindelsesobjekt med tilstanden SYN_SENT. Når denne SYN-meldingen ankommer mottakeren, sjekkes det om det finnes noen forbindelser som lytter på den riktige porten. Hvis det er tilfellet, opprettes nok et nytt forbindelsesobjekt på tjener-siden. Dette har imidlertid tilstanden SYN_RECV, og tildeles et nytt ledig portnummer. Objektet som lå og lyttet til å begynne med fortsetter med dette etterpå, slik at det er mulig for en tjenerapplikasjon å være tilknyttet flere klienter samtidig.

Siden listen() og connect() kallene ikke er blokkerende, trengs en mekanisme for å signalisere til applikasjonen at forbindelsen er oppe. Etterhvert som treveishåndtrykk-protokollen propagerer, flytter forbindelsesobjektene seg over i tilstanden ESTABLISHED. I det dette skjer, kaller TCP på en annen funksjon definert i **TcpApplication.**-grensesnittet. Denne funksjonen heter connectionEstablished(), og tar som parameter en referanse til forbindelsesobjektet som skal benyttes for å kommunisere over.

4.5.4 TCP-funksjoner

Til nå har det blitt forklart hvilke klasser TCP-modellen består av, og hvordan grensesnittene ser ut, men det gjenstår fortsatt å se hvordan de ulike TCP-mekanismene er implementert. Dette avsnittet vil derfor beskrive dette, med fokus på flyt og metningskontroll. Det er i all hovedsak seks metoder som tar seg av dette:

- tcp_Output(), i klassen **TCP**.
- tcp_Input(), i klassen **TCP**
- calculateRTO, i klassen **TcpConnection**
- RTOaction(), i klassen **TcpConnection**
- DACKaction(), i klassen **TcpConnection**
- Zeroaction(), i klassen **TcpConnection**

Hovedoppgavene til disse seks funksjonene, og noen detaljer om deres implementasjon vil nå bli beskrevet:

4.5.4.1 tcp_Output() har ansvaret for sending av datapakker fra TCP til nettlaget. Det er to hendelser som fører til TCP kaller denne metoden. Den ene er hvis applikasjonen har skrevet en pakke, og det andre er hvis det har kommet inn en kvittering (som forklart i 3.3, fungerer innkommende kvitteringer som klokke for utgående data). tcp_Output() vil sende så mange pakker som er tillatt av flyt og metningskontroll, ved å beregne det tilgjengelige vinduet ("usable window"). Når en pakke sendes, startes RTO-timeren hvis den ikke allerede er i gang [11]. Det er altså kun én timer som løper, ikke én for hver pakke som er en vanlig misforståelse. Dersom en "delayed ack" venter, "piggy-backes" denne med datapakken.

4.5.4.2 tcp_Input() tar i mot pakker fra nettverkslaget, og leter først opp riktig forbindelsesobjekt på bakrunn av mottaker-portnummeret i pakkehodet. Deretter handler metoden forskjellig avhengig av hvilken tilstand forbindelsen befinner seg i. Hva som skjer i LISTEN, SYN_SENT og SYN_RECV er allerede forklart tidligere, og uinteressant når vi fokuserer på flyt og metningskotroll. Tilstanden ESTABLISHED er derimot viktig i denne sammenhengen. Hvis pakken inneholder data, blir de skrevet til mottaksbufferet, og Glidende Vindu-variablene oppdateres. Enten blir en kvittering sendt på dette tidspunktet, eller så startes DACK-timeren. Hvis pakken inneholdt en ACK, beregnes først ny RTO-verdi ved hjelp av funksjonen calculateRTO(). Deretter kopieres verdien i *Advertised Window*-feltet i pakkehodet til tilstandsblokken for forbindelsen. Hvis ACK'en kvitterer for ny data, oppdateres Glidende Vindu-variablene, og plass frigjøres i send-bufferet. Hvis alle utsendte data nå er kvittert stanses RTO-timeren, hvis ikke restartes den. Til slutt oppdateres metningsvinduet "cnwnd". Som forklart i 3.3, øker cnwnd eksponensielt hvis forbindelsen er i slow start, og lineært hvis congestion avoidance er i gang. Forbindelsen er per definisjon i slow start dersom cnwnd er lavere enn ssthresh.

```
if(c.cnwnd < c.ssthresh){
    c.cnwnd += 1;           // slow start
}else{
    c.cnwnd += 1/c.cnwnd;   // congestion avoidance
}                       }
```

Figur 34: *Java kode som viser hvordan metningsvinduet cnwnd økes når ny data kvitteres. 'c' er en referanse til et forbindelsesobjekt*

Hvis det derimot var en duplisert kvittering, inkrementeres telleren

"dupacks", og når denne blir 3, re-sendes et segment (Fast retransmit). Da starter også Fast Recovery. Fast Recovery avsluttes når kvittering for ny data ankommer.

4.5.4.3 calculateRTO() er metoden som regner ut den nye RTO-verdien basert på en RTT-måling som kommer inn med ACK-pakker. Den benytter Jacobson's algoritme[8] som beskrevet i 3.3.

4.5.4.4 RTOaction() er metoden som blir kalt når RTO-timeren går av. Den gjør ikke annet enn å re-sende et segment, doble verdien verdien til RTO (eksponensiell backoff), og sette metningsvinduet til 1 (Slow start).

4.5.4.5 DACKaction() er metoden som kalles når DACK-timeren går av. Det betyr at det har gått 200ms siden sist pakke med data ankom, og ingen ACK har blitt sendt etter dette. Metoden sender derfor en ACK.

4.5.4.6 Zeroaction() er metoden som kalles når Zero-timeren går av. Selv om navnet tilsier at dette er metode som ikke gjør noenting, er ikke dette tilfellet. Den sender en probe-melding, og gjør eksponensiell backoff på Zero-timeren.

4.6 Feil og mangler i TCP og RED-modulene

Både underveis i utviklingsprosessen, og i ettetid, er det oppdaget feil og mangler i simulatormodellen. På grunn av tidsbegrensningene har det ikke vært anledning til å rette opp alle disse. De fleste av disse er allerede blitt nevnt i de foregående seksjonene, men vil beskrives nærmere i dette avsnittet. Det innebærer en detaljert beskrivelse av den uønskede oppførselen simulatoren får som følge av disse manglene, og et forslag til mulig løsning. Siden det er RED og TCP som er utviklet i dette arbeidet, vil ikke simulatorkjernen og RPR-modellen tas med i denne sammenhengen. Det kan allikevel nevnes at RPR-modellen er en tidlig versjon, som mangler noe av funksjonaliteten spesifisert i forslaget til 802.17-standarden. I de påfølgende avsnittene diskuteres de forskjellige feilene hver for seg:

4.6.1 Grensesnitt mellom TCP og RED

Avsnitt 4.5.2 presenterte problemet med at flere TCP-forbindelser konkurrerer om ett RED-buffer, og dette var løst ved å gjøre sending av pakker fra TCP til nettverkslaget til en atomær hendelse. Dette avsnittet skal forklare hvorfor denne løsningen vil påvirke TCP-trafikken, og hvordan det burde vært løst. Problemet ligger i tcp_Output-funksjonen. Når den

blir kalt, sender den alle pakkene som er tillatt av det tilgjengelige vinduet, som er gitt ved:

```
min(cnwnd,AdvertisedWindow) - (lastByteSent - lastAckReceived)
```

Det vil altså si sendevinduet minus mengden ukvitterte data, som i noen tilfeller kan bli et høyt tall hvis vinduet er stort. Når det å sende en pakke er en atomær hendelse, vil tcp_Output sende alle pakkene på samme tidspunkt i simulatoren. Hvis vi tenker oss en situasjon der TCP's tilgjengelige vindu er stort, og tcp_Output blir kalt for en spesiell forbindelse, vil det i virkeligheten ta noe tid å sende alle disse pakkene. I mens dette pågår, kan vi videre tenke oss at det kommer en pakke inn fra den andre parten. Hvis denne inneholder data, skal en kvittering sendes. Denne kvitteringen kunne vært "piggybacket" på en de pakkene som ennå ikke er sendt, men siden simulatoren sender hele vinduet på en gang, mister vi denne muligheten. Følgen blir altså at kvitteringen i verste fall forsinkes i 200ms hvis "delayed Acks" benyttes. Dette er ikke et problem som påvirker sendemønsteret til TCP i betydelig grad.

Måten dette kunne vært løst på, er å la TCP-objektet legge alle utgående segmenter i en "transmisjonskø", som altså er felles for alle forbindelsene. Videre ville det vært behov for en mekanisme som sørger for at pakkene i denne køen sendes til nettverkslaget én etter én. Nye utfordringer vil dukke opp. Hva skal f.eks gjøres dersom denne køen er full?

4.6.2 Wrapping av sekvensnummer

I avsnitt 4.4.2 ble det nevnt at Glidende Vindu-implementasjonen ikke håndterer at sekvensnumrene wrapper. Årsaken til dette er at koden som behandler sekvensnumre forutsetter at de nyeste pakkene alltid har høyere sekvensnummer enn de som er eldre. Innledningsvis i utviklingen ble dette gjort for å forenkle algoritmene, men intensjonen var å erstatte dette med kode som var "korrekt". Det viste seg i midlertid at det ikke var noe stort behov for å rette opp denne mangelen, i og med at sekvensnumrene nummerer pakker isteden for byte. Sekvensnummeret er representert med en 32-bits integer, som tillater en maksimal verdi på ca 4 mrd. Det vil si at hvis en TCP-forbindelse sender segmenter på MSS byte (ca 1400 byte), vil forbindelsen kunne sende ca 6 Terrabyte med data ($6*10^{12}$). Dette burde holde for de aller fleste applikasjoner. Hvis det allikevel er ønskelig med et større sekvensnummerrom, er det ikke noe i veien for å benytte datatypen "long" (64 bit) isteden.

4.6.3 RTO-timeren er for "finkornet"

I ordinære TCP Reno-implementasjoner, er det vanlig og "vekke opp" en forbindelse hvert 500ms for å sjekke om det har oppstått en timeout. Det vil i praksis si at en timeout i verste fall ikke oppstår før 499ms etter den egentlig skulle oppstått. Dette er et kjent problem med hensyn på ytelsen i timeout-situasjoner, men er gjort for å øke den generelle ytelsen på maskinen. I denne simulatoren er ikke timermekanismen implementert på denne måten. Forbindelsen får derimot en timeout med en nøyaktighet på nanosekundet(!). Dette vil naturligvis føre til en annen oppførsel enn ordinære Reno-implementasjoner, noe som ikke er ønskelig. Følgen blir at timeren vil gå av relativt lenge før det som er vanlig. Den vil imidlertid ikke gå av så tidlig at unødige retransmisjoner sendes. Effekten av dette er at trafikkmønsteret blir mindre klyngete (bursty) enn det som er normalt.

Denne feilen ble desverre ikke oppdaget før etter at simuleringene i neste kapittel ble gjennomført, og det var ikke anledning til å gjøre de på nytt.

Det er ikke mye endringer som skal til for å rette opp denne feilen. RTO-Timeren (og muligens de to andre timerene) kan endres slik at de runder av tidspunktet de skal "vekkes" på opp til nærmeste 500ms. Ingen andre endringer vil være nødvendig.

4.6.4 TCP-mekanismer som burde vært implementert

Dette ble diskutert i 4.4.2, så dette er bare ment som en kort oppsummering. SACK (Selective Acknowledgements) burde vært implementert, siden de fleste operativsystemer i dag benytter denne utvidelsen. Et viktig spørsmål er om fraværet av SACK har mye å si for trafikkmønsteret. Siden SACK gjør at tapte pakker kan retransmitteres raskere, og dermed unngå en timeout, vil trafikkmønsteret se annerledes ut. Uten SACK (og NewReno) vil forbindelsen oppleve flere timeouts, og dermed flere Slow Start-perioder. Når TCP brukes sammen med RED, vil imidlertid sannsynligheten for at flere pakker kastes fra samme vindu bli mindre, siden RED har som mål å være rettferdig.

I tillegg burde mekanismen for pålitelig nedkobling av forbindelser vært implementert, slik at det er mulig å gjennomføre simuleringer der et høyt antall forbindelser oppstår og dør slik som i virkeligheten, uten at simulatoren benytter unødvendig mye minne.

4.7 Oppsumering

Dette kapittelet har beskrevet simulatoren og de to modulene som ble utviklet. Først ble det gitt en enkel innføring i hva en "discrete event si-

mulator er", deretter ble simulatorkjernen og den allerede eksisterende RPR-modulen beskrevet. Deretter, i seksjon 4.3, ble RED-modulen gjennomgått .Seksjon 4.4 gjorde rede for hvilke mekanismer som er tatt med i simulatoren, og hvilke som ble utelatt, etterfulgt av en seksjon som beskrev selve implementasjonen av TCP. Til slutt ble feil og mangler ved modulene belyst.

5 Tester og simuleringer

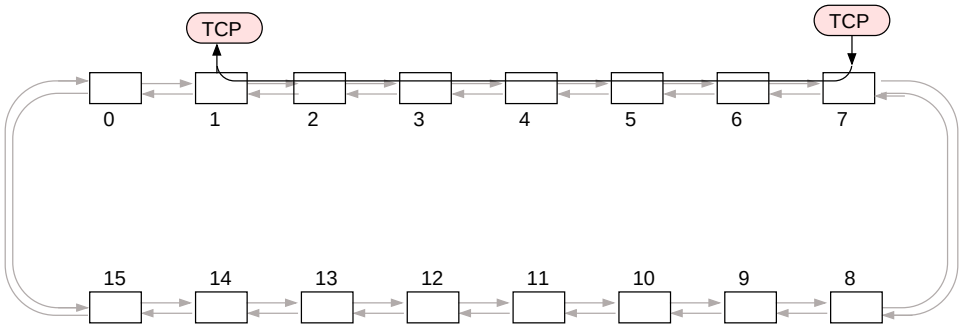
Til nå har rapporten beskrevet hvordan simulatoren er bygget opp, men ikke hvordan den fungerer. Dette kapittelet vil derfor presentere i alt seks testsimuleringer, som viser om de ulike mekanismene i TCP og RED-modulene svarer til forventningene eller ikke. Først beskrives fire tester av ulike mekanismer i TCP, og deretter en test av RED. Den siste simuleringen viser hvordan RED og TCP fungerer kollektivt. For hver simulering tolkes resultatene, og det tas stilling til om forventningene ble nådd eller ikke. Til slutt i dette kapittelet vurderes testsimuleringenes relevans i forhold til det å kunne fastslå at modulene er korrekt implementert.

5.1 Test av funksjoner i TCP

Målet med disse simuleringene er å vise hvordan noen utvalgte mekanismer i TCP-simulatoren fungerer. Her er det lagt vekt på metningskontrollmekanismene. Disse simuleringene er gjennomført slik at det kun er én TCP-forbindelse som sender data på ringen. For å simulere pakketap og forsinkelse vil én av nodene på ringen være programmert til å kaste eller forsinke bestemte pakker. (Selv om RPR ikke skal kaste pakker, kan pakker kastes på nettverkslaget i ingress eller egress-noden. For TCP sin del spiller det ingen rolle om det er i eller utenfor ringen dette skjer). Dette forklares nærmere under hver enkelt simulering. En alternativ måte å gjøre dette på ville være å belaste ringen med relativt mye trafikk slik at pakketap og forsinkelse ville oppstå “naturlig”, f.eks ved å benytte RED på ingress-noden. Den første metoden ble valgt fordi det blir enklere å observere TCP’s reaksjoner når man vet nøyaktig når et pakketap oppstår. RED-modulen er altså ikke i bruk i disse simuleringene. Scenariet er vist i figur 35. En TCP-forbindelse opprettes mellom node 1 og 7. Det er ingen annen trafikk på ringen. Applikasjonene på node 1 og 7 er henholdsvis en passiv mottaker og en aktiv sender. Dvs at applikasjonen på node 1 kun mottar data og sender kvitteringer på disse. Applikasjonen på node 7 skriver data med en rate som er såpass høy at TCP alltid har data å sende. Klokkehastigheten (bitraten) på linkene er 1Gbit/s, og avstanden mellom nodene er på 50km. Denne avstanden fører til en propagasjonsforsinkelse på 250μs mellom nodene, og en total forsinkelse ende til ende på 1500μs.

5.1.1 Fast Retransmit/Fast Recovery og Slow Start/Congestion Avoidance

Den første simuleringen ble gjennomført for å teste hvordan mekanismene Slow Start, Congestion Avoidance, Fast Retransmit og Fast Recovery" fungerer. Simuleringen skulle teste følgende:



Figur 35: *Scenario under TCP-simuleringer*

- Om metningsvinduet økes og minskes korrekt i de tre modiene (Slow Start, Congestion Avoidance og Fast Recovery)
- Om Slow Start avsluttes på riktig tidspunkt og Congestion Avoidance tar over.
- Om Fast Retransmit og Fast Recovery iverksettes etter mottak av tre dupliserte kvitteringer, og at mottakeren sender dupliserte kvitteringer så lenge det er et "hull" i mottakerbufferet.
- Om Fast Recovery avsluttes på riktig tidspunkt, og Congestion Avoidance tar over.

5.1.1.1 Scenario. For å teste Slow Start og Congestion Avoidance, er det ikke nødvendig å gjøre noe annet enn å sette opp en forbindelse og starte overføring på vanlig måte. For å teste Fast Retransmit og Fast Recovery derimot, må et pakketap simuleres. Dette ble gjort ved at node 1 (figur 35) overvåker innkommende pakker og kaster den første pakken med sekvensnummer 350 (retransmisjonen blir altså ikke kastet).

5.1.1.2 Forventet resultat. Når senderen ser den tredje dupliserte kvitteringen, vil den først re-sende det tapte segmentet, og deretter gå inni fast recovery modus. Oppsummert består dette av å sette ssthresh til halparten av antall utestående pakker, og å sette cnwnd til ssthresh+3. For hver dupliserte kvittering som ankommer etter dette vil metningsvinduet øke med 1 pakke (MSS byte), helt til kvittering for ny data ankommer. Da slutter fast recovery fasen, metningsvinduet settes tilbake til ssthresh, og “congestion avoidance” tar over med lineær økning av metningsvinduet. Simuleringen vil også vise at forbindelsen begynner i Slow start modus.

En graf som viser metningsvinduet,ssthresh i forhold til tiden, illustrerer hvordan denne mekanismen virker. Simulatoren vil produsere et punkt på cnwnd-grafen (Y=cnwnd,X=tid) for hver gang senderen mottar en kvittering. Det forventes at metningsvinduet vokser eksponensielt med tiden (slow start) helt til verdien ssthresh nåes, for deretter å vokse lineært (congestion avoidance). Det er også å forvente at metningsvinduet halveres på det tidspunktet senderen oppdager pakketapet. Dette vil være ca en RTT etter enn tidspunktet pakken ble sendt. Deretter skal vi se at metningsvinduet øker lineært. For å bedre illustrere hva som skjer, er det plottet flere verdier som funksjon av tiden:

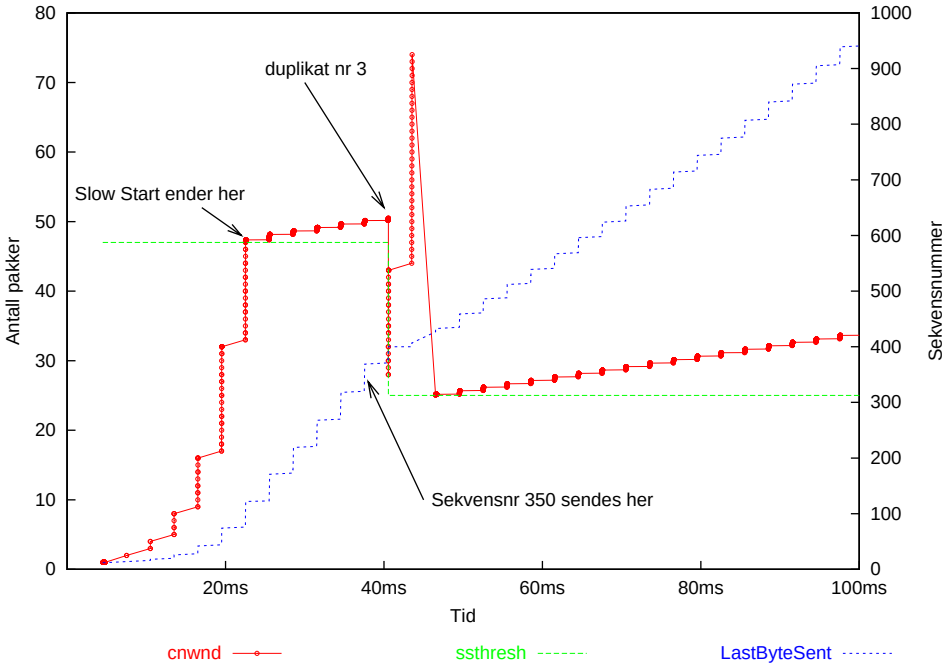
- *ssthresh*
  Denne verdien er grensene som sier når slowstart skal avsluttes og congestion avoidance skal starte. Denne grafen benytter samme skala på y-aksen som cnwnd.

- *lastByteSent*
  Ved å vise lastByteSent som funksjon av tiden, illustreres progresjonen i sekvensnummer. Av dette kan vi se hvor ofte senderen sender ut ny data, og dermed danne et bilde av gjennomsnittsraten. Som beskrevet i 3.3 inneholder lastByteSent det høyeste sekvensnummeret som er sendt. Denne grafen benytter den høyre y-aksen i plottet.

5.1.1.3 Resultat og konklusjon. Figur 36 viser resultatet av simuleringen. På x-aksen vises tiden i millisekunder. Den venstre y-aksen måler antall pakker, og benyttes av cnwnd og ssthresh. Den høyre y-aksen måler sekvensnummer, og benyttes av lastByteSent. Vi ser at cnwnd stiger eksponensielt til å begynne med. Deretter, i det cnwnd er like stor som ssthresh, avsluttes Slow start, og Congestion Avoidance overtar (ved tiden 22 ms). Dette viser at algoritmen ssthresh benyttes på riktig måte.

Siden propagasjonsforsinkelsen er 1500μs, vil RTT være ca dobbelt av dette, altså ca 3ms (siden det ikke er noen annen trafikk på ringen vil ikke pakkene køes i RPR-nodene). Under Slow Start forventes det derfor at kvitteringer vil ankomme i klynger hver RTT. Dette stemmer overens med cnwnd-grafen som viser at metningsvinduet dobles ca hvert tredje millisekund. Grafen viser også at metningsvinduet øker lineært når congestion avoidance er i gang. Konklusjonen av dette er at metningsvinduet justeres korrekt under Slow Start og congestion avoidance i dette scenariet.

Figuren viser også at TCP sender pakker med en rate som er tilpasset metningsvinduet. Grafen lastByteSent stiger hver gang en pakke med nye data sendes, og også her antydes det en eksponensiell vekst under slow start, og lineær vekst under congestion avoidance. Ved ca 37,5 ms sen-



Figur 36: *Illustrerer TCP Renos "Slow Start / "Congestion Avoidance" og "Fast Retransmit" / "Fast Recovery". Congestion Avoidance tar over etter Slow Start når metningsvinduet (cnwnd) passerer ssthresh. Pakken med sekvensnummer 350 forsvinner, og Fast Retransmit/Fast Recovery starter når dupliserte kvittering nr 3 ankommer. Fast Recovery "blåser opp" vinduet ved mottak av påfølgende duplikater, og ender når ny data kvitteres.*

des pakken med sekvensnummer 350, og vil som nevnt forsvinne et sted på veien. Etter ca 40,5 ms (RTT sekunder etter at pakke 350 ble sendt) får senderen den tredje dupliserte kvitteringen, og går inn fast recovery modus. I figuren vises dette ved at ssthresh settes til 25, halvparten av flightsize, og at cnwnd-grafen brått faller fra ca 50 til 28 (ssthresh + 3). Deretter stiger cnwnd like brått opp til ca 43. Dette er fast recovery som "blåser opp" vinduet ved mottak av flere dupliserte kvitteringer. Etter ytterligere RTT sekunder kommer flere dupliserte kvitteringer, og cnwnd blåses opp til over 70. Denne oppblåsingen fører til at senderen får lov til å sende flere pakker med nye data, selv om lastByteAcked er uforandret. Dermed opprettholdes "ack-klokkingen" som beskrevet i 3.3. LastByteSent-grafen viser at ny data blir sendt under Fast Recovery. Når ny data kvitteres (ved ca 43,5ms), blir metningsvinduet satt ned til ssthresh, og Fast Recovery perioden avsluttes. Dette beviser at det tapte segmentet ble retransmittert da den tredje dupliserte kvitteringen ankom (Fast Retransmit). Tilslutt ser vi at congestion avoidance iverksettes og varer ut simuleringstiden, ved at cnwnd vokser lineært.

Konklusjonen av denne simuleringen er gjennomgående positiv. Oppsumert viser grafene følgende ved denne simuleringen:

- Metningsvinduet økes korrekt i Slow Start modus.
- Metningsvinduet økes korrekt i Congestion Avoidance modus.
- Overgangen fra Slow Start til Congestion Avoidance inntreffer på riktig tidspunkt.
- Fast Retransmit utføres ved mottak av tre dupliserte kvitteringer.
- Fast Recovery modus overtar etter Fast Retransmit.
- ssthresh halveres korrekt ved inngangen til Fast Recovery.
- Metningsvinduet settes til ssthresh ved inngangen til Fast Recovery, og blåses opp ved mottak av dupliserte kvitteringer helt til ny data kvitteres.
- Fast Recovery avsluttes på riktig tidspunkt, og Congestion Avoidance tar over.

Disse grafene viser at Fast Recovery opererer som forventet ved tap av én pakke. Hva som skjer ved tap av flere pakker innenfor samme vindu av data vises imidlertid ikke her. Forventningsvis vil da en timeout oppstå, og Slow Start ta over. Siden dette er Reno TCP uten SACK-utvidelsen, (se avsnitt 3.3.6) gjøres det ikke noe spesielt for å unngå denne timeouten. De to neste avsnittende vil beskrive simuleringene som ble gjennomført for å teste henholdsvis timeout-mekanismen og beregningen av RTO (Retransmission Timeout)-verdien.

5.1.2 Timeout-mekanismen

Den neste simuleringen skal vise at timeout-mekanismen fungerer. Dvs at hvis det ikke ankommer en kvittering hos senderen i løpet av RTO sekunder etter at timeren ble startet, skal RTO-timeren gå av. Da skal den pakken som ble sendt tidligst, men som ikke har blitt kvittert, sendes på nytt. Deretter skal senderen gå inn i slow start. Denne simuleringen vil ikke vise om RTO beregnes riktig, kun hva som skjer når RTO-timeren går av. Den neste simuleringen derimot, vil teste hvordan RTO varierer med en flukturerende RTT.

5.1.2.1 Scenario. For å unngå at det kommer inn kvitteringer i løpet av en periode på RTO sekunder, er node 1 (figur 35) programmert til å kaste alle innkommende pakker i et tidsinterval som er stort nok til at alle pakkene i et vindu blir kastet, men mindre enn at retransmisjonen ikke blir kastet. Det største mulige tidsintervallet er valgt, som er på ett sekund. Grunnen er at RTO aldri vil bli mindre enn 1 sekund (se 3.3). "Nedetiden" på node 1 er satt til å vare fra 100 ms til 1100 ms.

5.1.2.2 Forventet resultat. I likhet med forrige simulering, er det variablene cnwnd, ssthresh og lastByteSent som er interessante. Her forventes det at forbindelsen startes i Slow Start, og går over i Congestion Avoidance på samme måte som i forrige simulering. Ved tiden 100ms vil node 1 starte nedetiden, og mottakersiden vil dermed slutte å sende kvitteringer. Relativt kort tid etter dette (ca 1/2 RTT) vil senderen oppleve at kvitteringer uteblir, og får dermed ikke mulighet til å sende nye data (det er mottak av kvitteringer som trigger utsending av ny data). Deretter vil det gå lengre tid, minst 1 sekund, før timeren går av. Da vil senderen retransmittere det tidligst sendte segmentet som ikke er kvittert, sette ssthresh lik flightsize/2 (forklart i seksjon 3.3.5.3) og gå inn i Slow Start modus.

5.1.2.3 Resultat og konklusjon. Figur 37 viser resultatet av denne simuleringen. Siden denne simuleringen varer lengre enn den forrige, ser grafene litt annerledes ut. Den forrige simuleringen varte i 100ms, mens denne varer i 1400ms for å få med hele timeout-perioden. Derfor ser det som om cnwnd og LastByteSent vokser raskere, men det er altså ikke tilfellet.

Selv om det ikke er lett å se, antyder cnwnd-grafen at senderen opererer i slow start helt til ssthresh cnwnd passerer ssthresh ved ca 25ms. Deretter overtar congestion avoidance. Ved ca 100ms ser vi som forventet at lastByteSent og cnwnd slutter å vokse. Deretter går det ca 100ms før den siste kvitteringen ankommer. Grunnen til at denne kommer så

sent i forhold til de andre er at den ble holdt igjen av "delayed acks"-mekanismen hos mottakeren[10]. Som nevnt i avsnitt 3.3.2.3 gjør denne mekanismen at mottakeren kun sender kvitteringer på annenhvert segment, med mindre delayed-ack timeren går av. Etter mottaket av denne kvitteringen går det lengre tid før noe skjer. Det betyr at senderen har sluttet å motta kvitteringer, og følgelig fortsetter RTO-timeren å løpe. Ca ett sekund etter dette settes ssthresh til 30 som er halvparten av flightsize på det tidspunktet siste kvittering ankom. (Vi ser at cnwnd også lå på 60 på dette tidspunktet, noe som viser at senderen hadde sendt ut det maksimale antallet pakker tillatt av metningsvinduet). I tillegg til senkingen av ssthresh settes metningsvinduet til ett segment. Dette betyr at timeren har gått av, og Slow Start er i gang. Her kan vi også se på lastByteSent-grafen at flere pakker sendes på nytt ved at den først synker et lite stykke før den begynner å øke igjen. Dette er fordi senderen må re-sende alle pakkene som ikke er kvittert, og det vil i dette tilfellet være et fullt vindu. Siden ssthresh er senket tar congestion avoidance over på et tidligere tidspunkt enn initielt.

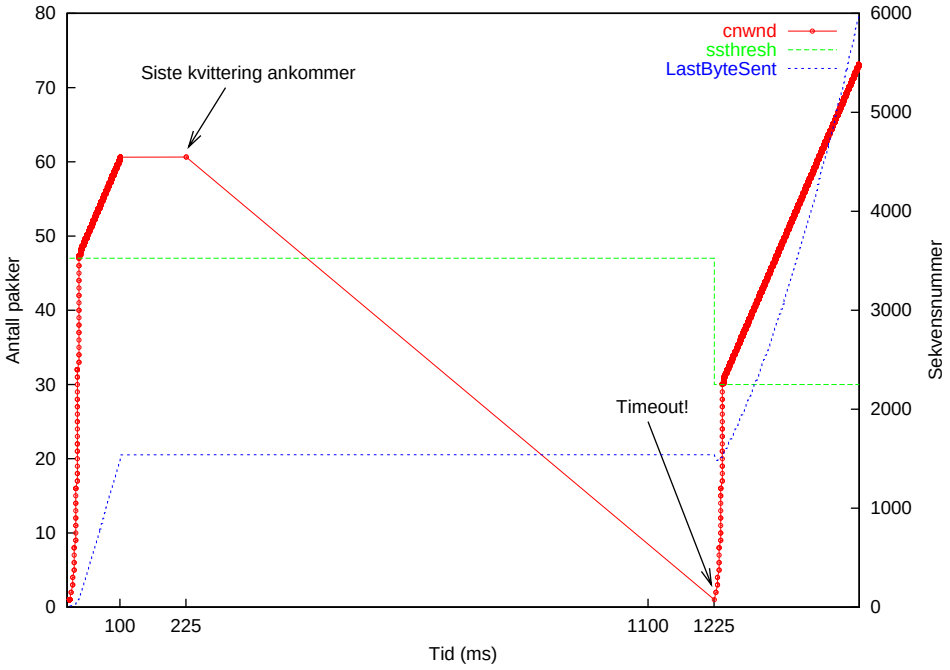
En annen ting som kan observeres av dette plottet, er at senderaten øker når metningsvinduet økes. Dette kom ikke så godt frem i forrige simulering, siden simuleringstiden var kort. Her er den imidlertid lang nok til å se at lastByteSent-grafen krummer seg litt oppover når cnwnd øker. Dette synes best etter at timeouten har gått av. Denne krumningen betyr at stigningstallet til grafen øker, og dette stigningstallet er målet på senderaten.

Denne simuleringen antyder dermed at timeout-mekanismen fungerer.

- Timeren går av RTO sekunder dersom ingen kvitteringer ankommer før dette.
- ssthresh settes til halvparten av flightsize (lastByteSent minus lastAckReceived).
- Metningsvinduet settes ned til ett segment.
- Det tidligst sendte segmentet som ikke er kvittert re-sendes.
- Slow Start tar over og re-sender resten av de ukvitterte pakkene.

Selv om det ikke blir presentert her, er det også testet og verifisert at RTO-verdien dobles hvis det retransmitterte segmentet ikke blir kvittert (eksponensiell backoff). Det neste avsnittet beskriver simuleringen som testet om RTO beregnes riktig.

[10]Delayed ack-timeren var satt til 100ms i denne simuleringen uten noen spesiell grunn.



Figur 37: *Viser at RTO-timeren går av (ved ca 1225ms) på riktig tidspunkt (RTO = 1 sekund), og at ssthresh og cnwnd justeres korrekt. Deretter retransmitteres ett segment og Slow Start tar over.*

5.1.3 RTO (Retransmission timeout) beregning

Målet med denne simuleringen er å teste RTO beregningen på sendesiden TCP. Som beskrevet i seksjon 3.3, benytter TCP-modulen Jacobson's algoritme[14][8] og Karn's algoritme[27] for å gjøre dette. Der ble det også forklart at TCP bruker to variable for å beregne RTO. Dette er SRTT (Smoothed Round Trip Time) og RTTVAR (RTT Variation). Mer detaljert vil RTO beregnes slik for hver gang en RTT-måling (sample) kommer inn:

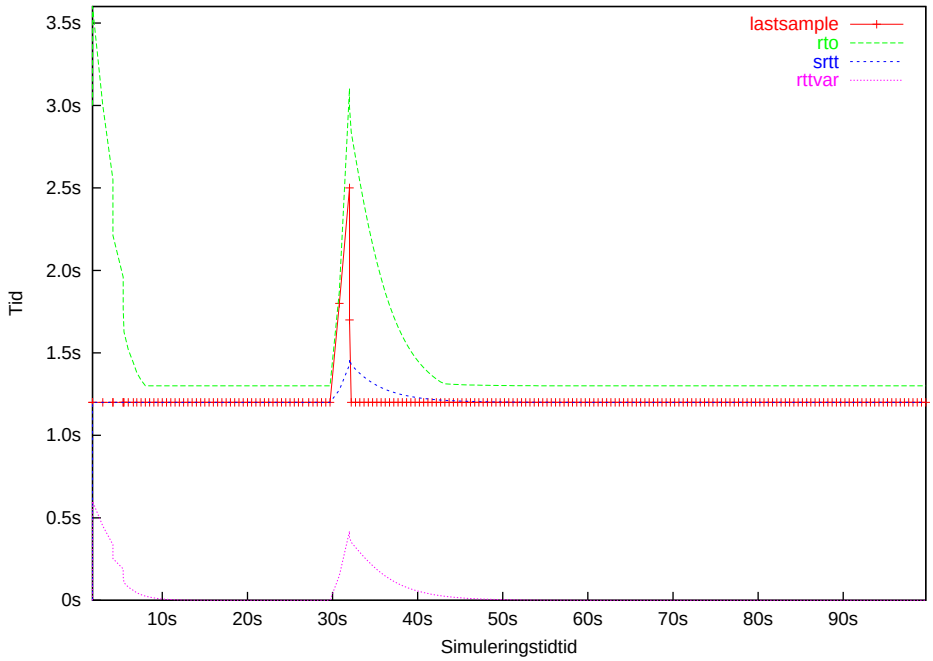
$$RTTVAR = (1 - \beta) * RTTVAR + \beta * |SRTT - sample$$
$$SRTT = (1 - \alpha) * SRTT + \alpha * sample$$
$$RTO = SRTT + max(G, 4 * RTTVAR)$$

Konstantene α og β er satt til 1/8 og 1/4 som spesifisert i [8]. RTTVAR er altså et mål på variasjonen i målinger (gjennomsnittsavviket), mens SRTT er et vektet gjennomsnitt av RTT-målingene.

Dette fører til at RTO alltid vil ligge minst G tidsenheter over SRTT. G er klokkegranulariteten til timereren. I denne simulatoren er denne egentlig null, men for at RTO alltid skal ligge litt over SRTT er den satt til 100ms her. Om dette er nødvendig er ikke fastslått, men det fører til en mer konservativ oppførsel. Det vi vil teste i denne sammenhengen, er om RTO tilpasser seg SRTT og RTTVAR, og dermed RTT-målingene, på riktig måte.

5.1.3.1 Scenario. Simuleringsscenariet er det samme som i de forrige simuleringene, men her kastes ingen pakker i løpet av simuleringstiden. Isteden forsinkes noen av pakkene på veien slik at de får en høyere RTT en det som er normalt. De forsinkes imidlertid ikke så lenge at senderen får en timeout. Mer detaljert er node 1 (figur 35) programmert til å forsinke pakker som ankommer i tidsrommet 30s til 30.5s. Avstanden mellom nodene er satt til 20000km, slik at propagasjonsforsinkelsen blir 100ms. Dette er en usannsynlig stor avstand (RPR har som mål å støtte ringer med en omkrets opp til 2000km), men er gjort for å få RTT over 1 sekund. Grunnen er at RTO har en minimumsverdi på 1 sekund[8], og hvis ikke rtt-målingene er høyere enn dette, vil vi ikke se noen variasjon i RTO.

5.1.3.2 Forventet resultat. Vi forventer at når RTT-målingene holder seg konstant, vil SRTT, RTTVAR og følgelig RTO også holde seg konstant. Når RTT-målinger for de forsinkede pakkene kommer inn, vil SRTT og RTTVAR vokse. Det fører igjen til at RTO øker. Så lenge det er små forandringer i RTT-målingene, vil etterhvert også RTO jevne seg ut.



Figur 38: *Viser hvordan RTO varierer med RTT-målingene. Strekene vinkelrett på lastsample-grafen markerer RTT-målinger gjort av senderen ved ankomst av kvitteringer. Tre pakker blir forsinket i nettet ved ca 30s, slik at RTT for disse pakkene øker. Avstanden mellom nodene er satt til å være stor for at RTT skal bli over ett sekund.*

5.1.3.3 Resultat og konklusjon. Figur 38 viser resultatet av denne simuleringen. RTO synker gradvis fra sin initielle verdi (3.5s) ned til en minimumsverdi (SRTT + G). Så lenge RTT-målingen er jevne er RTO stabil som forventet.

Etter ca 30s kommer det inn tre RTT-målinger som avviker sterkt fra det normale. Dermed vokser RTTVAR og SRTT, som gjør at også RTO øker. Når de "normale" målingene begynner å komme inn igjen, synker RTO gradvis tilbake til det "normale" nivået. Dette viser at de tre variablene endres i riktig retning i forhold til hverandre, og siden det er konstantene α og β som styrer utjevningsgraden på RTTVAR og SRTT, kan vi med relativt stor sikkerhet anslå at algoritmen er riktig implementert.

Som beskrevet i avsnitt 4.6.3, er det allikevel et avvik som gjør at RTO-verdiene i denne TCP-modulen fraviker noe fra ordinære TCP-implementasjoner. Det ble tidligere i dette avsnittet nevnt at konstanten G egentlig er lik null. Det vil altså si at RTT-målingene har nanosekund nøyaktighet (siden simulatoren operer med nanosekunder som tidsenhet). I ordinære implementasjoner er G 500ms, som gjør at RTO "hopper" mellom nivåer som ligger 50ms fra hverandre. Dette er en svakhet ved TCP-modulen og bør rettes opp.

5.1.4 Zero window probing

TCP-mottakeren annonserer en vindustørrelse på null, når mottaksbufferet er fullt. Denne situasjoner er forklart i 3.3.2. Dette skjer hvis applikasjonsprosessen leser data med lavere rate enn det som kommer inn fra senderen. Når senderen ser dette “null-vinduet” venter den med å sende mer data. Dette fører til at ack-klokken stopper opp. Mottakeren får aldri sendt en vindusoppdatering siden den ikke får noen data å kvittere. Zero Window Probing løser dette problemet ved at senderen sender ut “probe-meldinger”. Hensikten med disse er å få mottakeren til å generere vindusoppdateringer for å finne ut om vinduet har åpnet seg. Denne simuleringen skal teste både at senderen sender ut probemeldinger på de riktige tidspunktene, og at mottakerens annonserte vindu gir et riktig blide av mottaksbufferets tilstand.

5.1.4.1 Scenario. Scenariet i denne simuleringen er likt med forrige simulering når det gjelder avstand mellom nodene, men ingen pakker forsinkes eller kastes. Det er ingen spesiell grunn til at det er valgt å ha denne store avstanden, annet en at simuleringene ble gjennomført i den rekkefølgen de presenteres her, og at det ikke var noen grunn til å minke avstanden. For at TCP på mottakersiden skal fylle opp mottaksbufferet sitt, brukes en klient-applikasjon som avbryter lesingen fra mottaksbufferet i perioden mellom 30s og 60s.

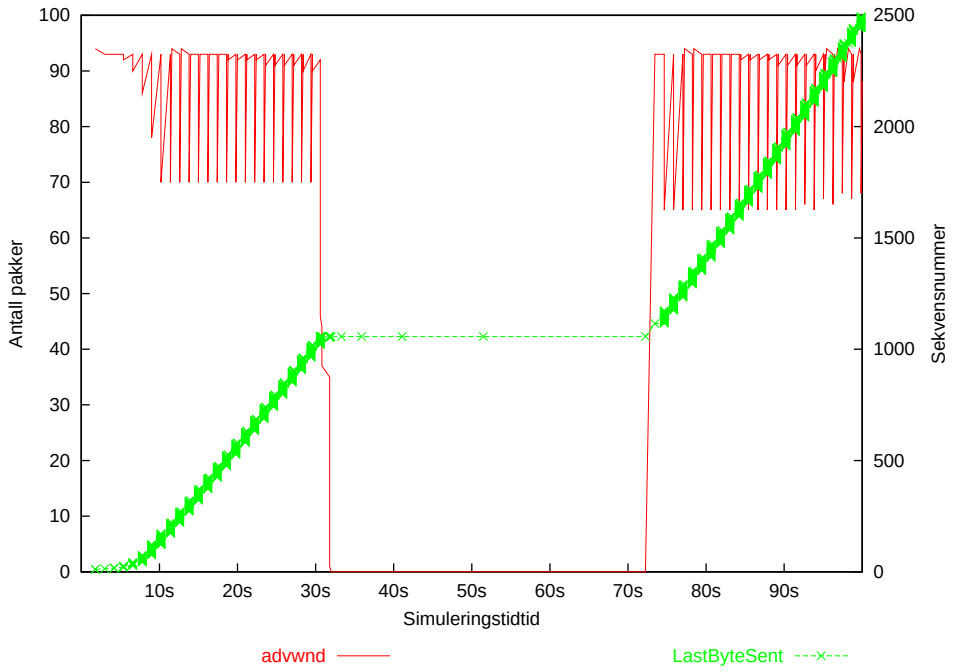
5.1.4.2 Forventet resultat. Det forventes at mottaksbufferet fylles opp når klienten slutter å lese etter 30s av simuleringstiden. TCP på mottakersiden da vil annonsere et minkende vindu, som tilslutt kommer helt ned i null. Når senderen ser dette, må den slutte å sende data, borsett fra probe-meldingene. Så lenge vinduet er null vil disse probe-meldingene sendes ut med intervaller som øker eksponensielt (RTO*2^n). Etter 60s skal applikasjonen begynne tømmingen av bufferet, og dermed vil det annonserte vinduet åpne seg når neste probemelding blir kvittert

5.1.4.3 Resultat og konklusjon Figur 39 viser senderens variable advwnd (mottakerens annonserte vindu) og LastByteSent. At advwnd varierer tilsynelatende mye fortjener en kommentar selv om det ikke er sentralt i denne sammenhengen. Grunnen er at senderen sender data i klynger (bursts). Mottakeren prosesserer disse dataene saktere enn de ankommer, slik at bufferet fylles opp. Når hele klyngen har ankommet, fortsetter applikasjonen og lese data og vil dermed tømme bufferet igjen. Etter ca 30s ser vi at advwnd synker raskt ned til null. Dette kommer av at mottakerprosessen har stoppet å lese data fra bufferet. Deretter ser vi at lastByteSent flater helt ut. Det betyr at senderen ikke sender noen data. Punktente på lastByteSent-grafen representerer sending av en pakke, og vi ser at noen pakker sendes med økende mellomrom i denne perioden. Dette er probe-meldingene som sendes etter at "zerowindow-timeren" går av. Den siste av disse sendes like etter 70s, og kvitteringen for denne inneholder en positiv vindusoppdatering (siden mottakeren har begynt å prosessere data igjen). Dermed fortsetter sendingen av data som normalt. Grafen viser at senderaten er omtrent lik før og etter denne venteperioden. Senderen har altså ikke gjort noen forandring på metningsvinduet. Dette er fornuftig, siden det ikke var metning som var årsaken til at senderen måtte begrense senderaten. Figur 39 viser også hvordan zerowindow-timeren gjør en "eksponensiell backoff" for hver gang et null-vindu ankommer, ved at avstanden (i tid) mellom probemeldingene er eksponensielt økende.

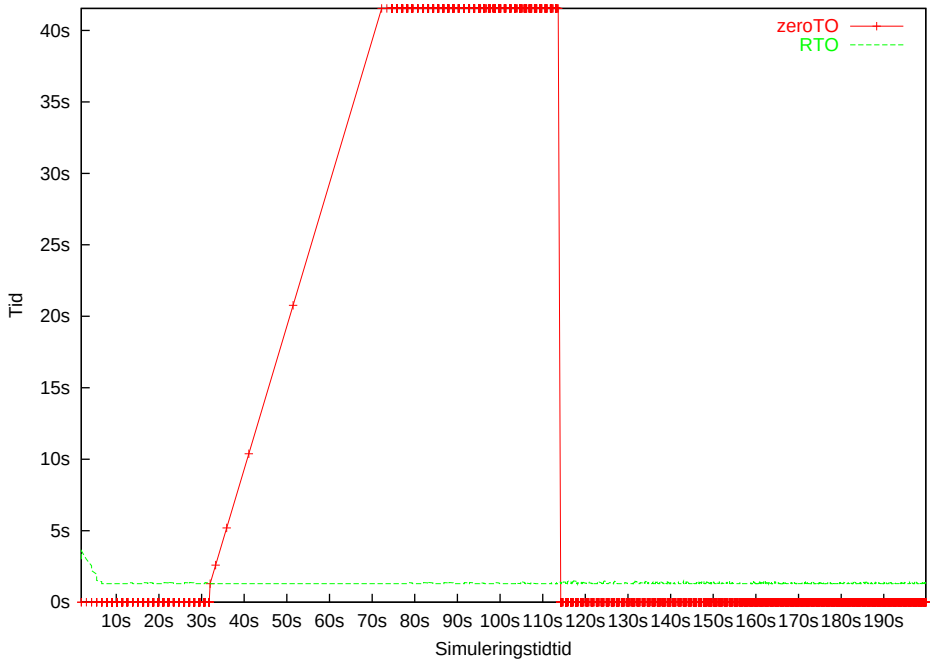
Konklusjonen her blir dermed at i dette tilfellet fungerte både mottakersiden og sendersiden som de skulle.

5.2 Test av RED

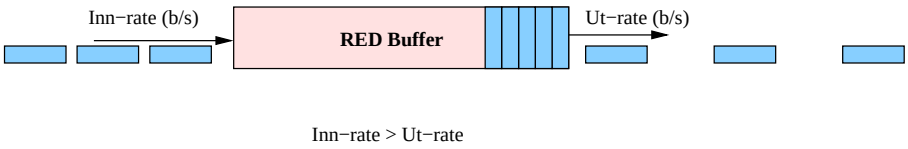
Den neste simuleringen skal vise om RED (Random Early Detection)-algoritmen fungerer, gitt et sett statiske parametere. Disse parameterne er ikke valgt etter nøyaktige beregninger av hvordan trafikkmengde RED-bufferet skal slippe igjennom. Det er heller ikke poenget her, men at RED oppfører seg riktig i henhold til parameterne som er gitt. Algoritmen er forklart i 3.5. Parameterne det er snakk om er:



Figur 39: *Zero window probing. Viser at senderen slutter å sende data når mottakeren annonserer et null-vindu, og at senderen sender probemeldinger med eksponensielt økende tidsintervaller.*



Figur 40: *Zero window probing. Viser at Zero-timeren øker eksponensielt når probemeldingene fortsatt genererer nullvinduer hos mottakeren. (Zero-timeren nullstilles ikke før timeren som ble satt for den sist sendte proben går av, men som vist i figur 39 åpner vinduet før dette)*



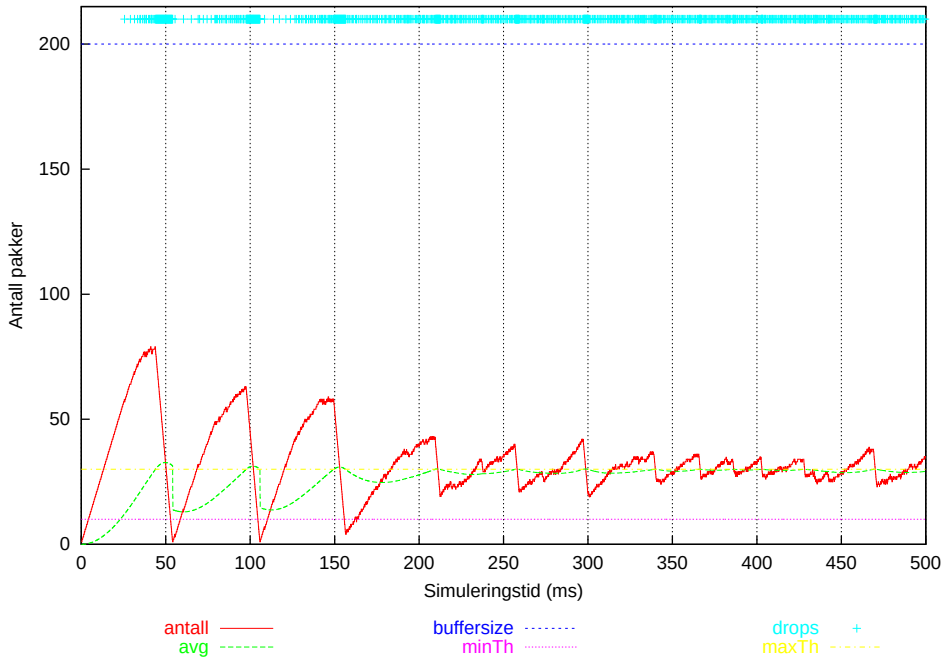
Figur 41: *RED - Buffer som fylles opp*

- **minTh** (minimum threshold)
- **maxTh** (maximum threshold)
- **weight** (Brukes til å beregne gjennomsnittlig kølengde)
- **maxP** (Brukes til å beregne sannsynligheten for å merke en pakke. Se figur 22 på side 44)
- **REDsize** (Størrelsen på RED-bufferet).

Alle disse parameterne påvirker RED-algoritmen i hvor ofte den skal merke (kaste) pakker. De må settes på forhånd og forandrer seg ikke under simuleringen.

5.2.1 Scenario.

For at kølengden (og den gjennomsnittlige kølengden) skal øke, må pakkeraten være større inn i bufferet enn ut av bufferet. Vanligvis skjer dette naturlig ved at det oppstår kollisjoner (contention) i ruteren, slik at pakken først i bufferet må vente, samtidig som det kommer inn pakker "bakfra". For å simulere dette kan man enten sende en strøm mellom to andre noder, sånn at slike kollisjoner oppstår, eller man kan gå inn å manipulere pakkeraten som sendes ut av RED-bufferet. Den siste metoden ble valgt i denne simuleringen. Hadde den første blitt valgt, ville forskjellen mellom inn-raten og ut-raten (se figur 41) i bufferet vært udefinert, og dermed ville det blitt vanskeligere å tolke resultatene av simuleringen. Isteden ble ut-raten eksplisitt satt til å være ca 1.3 ganger så stor som innraten, slik at antall pakker i bufferet vokser med konstant hastighet, som om det skulle gått en jevn transitt-strøm forbi denne stasjonen. Nærmere bestemt sendes pakker inn i bufferet med en rate på ca 15.75 Mbit/s inn i bufferet, mens ut-raten er ca 12.16 Mbit/s. Disse ratene er konstante under hele simuleringen, og verdiene ble valgt etter eksperimentering med forskjellige verdier for å produsere grafer som er lette å tolke. Kildene som sender data benytter altså ikke TCP, så de vil ikke tilpasse seg nettets tilstand.



Figur 42: *RED - Konstant trafikkpåtrykk*

Simulatoren rapporterer bufferets nåværende kølengde og den gjennomsnittlige kølengden for hver gang en pakke kommer inn eller går ut av bufferet. I tilleg rapporteres tidspunktene der pakker blir merket.

5.2.2 Forventet resultat.

Siden datakildene ikke tilpasser senderaten etter pakketap, forventes det at den gjennomsnittlige kølengden skal stabilisere seg rundt maxTh (som beskrevet i [4]). Kølengden vil øke konstant helt til den gjennomsnittlige kølengden passerer den nedre grensen (minTh). Herfra vil vi kunne observere pakketap, noe som igjen fører til at kølengden vokser noe tregere. Den gjennomsnittlige kølengden vil forsette å øke, og vil nærme seg den øvre grensen maxTh. På et tidspunkt, enten litt før eller litt etter at den øvre grensen nås, vil kølengden nå et lokalt maksimum. Den vil deretter synke, og ta med seg den gjennomsnittlige kølengden. Dette vil føre til at færre pakker kastes, og at kølengden når et lokalt minimum, for så å begynner å øke igjen. Dette mønsteret vil gjenta seg under hele simuleringen, men oscilleringen vil altså avta, slik at gjennomsnittet stabiliserer seg om den øvre grensen. Det forventes også at pakken vil kastes tettere når den gjennomsnittlige kølengden er høy.

5.2.3 Resultater og konklusjon

Figur 42 viser resultatet av simuleringen over en periode på 500ms. De to nederste vannrette linjene markerer minTh og MaxTh, som er på henholdsvis 10 og 30 pakker. Den vannrette linjen ved 200 pakker markerer størrelsen på bufferet, og punktene (angitt med +'er) markerer at RED kaster én pakke.

Plottet forteller at forventningene oppfylles, ved at den nåværende kølengden oscillerer om maxTh, og at de lokale maksima og minima for denne grafen befinner seg på riktig sted i forholde til den gjennomsnittlige kølengden. I tillegg kan man observere at den gjennomsnittlige kølengden etterhvert stabiliserer seg tett opptil den øvre grensen, noe som også var forventet. Den nåværende kølengden stabiliserer seg også om denne grensen, men i noe mindre grad. Det er naturlig, siden den styres av den gjennomsnittlige kølengden. Det ble eksperimentert med ulike parametere (forhold mellom inn og ut-rate, weight og maxP), og disse simuleringene viste også denne stabiliseringen. Disse simuleringene blir imidlertid ikke presentert her. Avslutningsvis, er det også mulig å observere at kastetettheten vokser med den gjennomsnittlige kølengden.

I denne simuleringen oppførte altså RED seg som forventet, noe som tyder på at beregning av den gjennomsnittlige kølengden er riktig implementert.

5.3 Interaksjon mellom RED og TCP

Hittil har simuleringene dreid seg om hvordan de forskjellige mekanismene i TCP og RED oppfører seg hver for seg. Denne seksjonen tar for seg hvordan de fungerer sammen.

5.3.1 Motivasjon

De foregående simuleringen har antydet at TCP's metningskontroll tilpasser seg pakketap på riktig måte, og at RED tilsynelatende kaster pakker når den skal. Denne simuleringen ble gjennomført for å bekrefte eller avkrefte begge disse resultatene, i tillegg til å avgjøre om RED er rettferdig. Dvs at den tilgjengelige båndbredden fordeles relativt likt mellom flere TCP-flyter. Det er RED-algoritmens evne til å fordele merking av pakker jevnt utover som avgjør dette (den har ingen "per flyt" informasjon). Dette oppnås ved å merke tilfeldige pakker, slik at sannsynligheten for at en TCP-flyt blir valgt, omtrent er proporsjonal med denne flytens størrelse (gjennomstrømning). Det ble derfor gjennomført flere simuleringer med ulike antall TCP-forbindelser.

I tillegg vil valget av parameteren MinTh være en problemstilling i denne sammenhengen. MinTh er avgjørende for hvor store klynger med

Parameter	Verdi
Bufferstr	200
Weight	0,002
MaxP	0,1
MinTh	x
MaxTh	3*MinTh

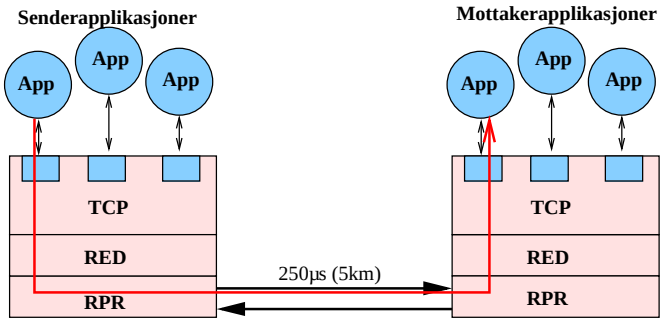
Tabell 2: *RED - Simuleringsparametere*

pakker algoritmen godtar før merkingen starter. I tillegg styrer den størrelsen på den gjennomsnittlige kølengden. Det vil derfor være en "tradeoff" mellom lav forsinkelse, og optimal utnyttelse av den tilgjengelige båndbredden. MaxTh virker hovedsakelig som en øvre grense for flyter som ikke samarbeider med RED, f.eks UDP (User Datagram Protocol)-flyter (vi så virkningen av dette i forrige simlering, ved at den gjennomsnittlige kølengden stabiliserte seg ved maxTh). Hvilke verdier disse parameterne bør ha, er avhengig av hva slags og hvor mye trafikk som skal passere RED-ruteren og hastigheten på linken(e). I følge Sally Floyd (en av grunnleggerene av RED), bør maxTh være tre ganger så stor som minTh [25]. Spørsmålet er da hvor man bør sette minTh. Som nevnt, vil en høyere minTh føre til økt forsinkelse. Dermed spørs det hvor stor propagasjonsforsinkelse og transmisjonsforsinkelse det er på linken. Hvis køforsinkelsen er mye mindre enn disse, kan minTh økes uten at det har stor betydning for den totale forsinkelsen. Jo høyere minTh er, jo større klynger av pakker kan passere "umerket", og jo bedre utnyttelse av linken oppnås. Siden denne simulatoren (og RPR) er bassert på høyhastighetslinker (>655Mbit/s) som gjerne strekkes over lange distanser, vil hastigheten pakker klokkes inn og ut av bufferne være liten i forhold til propagasjonsforsinkelsen på linken. Dette taler for en høyere minTh. For å se hvordan valg av minTh påvirker den totale gjennomstrømningen, ble det gjort flere simuleringer med ulike minTh. Valg av størrelsen på selve bufferet er avhengig av hva man tillater som maksimal forsinkelse. I disse simuleringene er denne satt til 200 pakker, noe som kan være forholdsvis mye (den siste pakken i bufferet vil få en forsinkelse på minst 2.4ms forutsatt en bitrate på 1Gbit/s og en pakkestørrelse på 1500 byte, mer hvis det går transitt-trafikk på ringen).

Når det gjelder weight, som brukes til å beregne den gjennomsnittlige kølengden, er denne satt til 0.002. MaxP er satt til 0.1. Begrunnelsen for disse kan leses i [25].

Oppsumert vil denne simuleringen teste følgende:

- Bekrefte/avkrefte at TCPs metningskontroll tilpasser seg pakketap i nettet.



Figur 43: *Scenario*

- Bekrefte/avkrefte at RED-algoritmen kaster pakker når den skal.
- Om RED er rettferdig.
- Om den gjennomsnittlige kølengden blir kontrollert slik at den stabiliserer seg om minTh.

5.3.2 Scenario

I alt ble det gjennomført ni simuleringer, der tre forskjellige verdier av minTh (10/15/25 ant pakker) ble kombinert med tre forskjellige antall forbindelser (1/2/10). Tabell 2 oppsummerer parameterne som ble brukt. Alle simuleringene ble kjørt i 100s ($100*10^9$ simulator klokketikk). Applikasjonen som kjører over TCP består av en sender som sender data med en rate på 15.75Mbit/s, og en passiv mottaker som ikke gjør annet enn å prosessere de dataene som kommer inn. TCPs bufferstørrelse, som bestemmer vindusstørrelsen, er satt stor nok til at senderaten ikke begrenses av flytkontroll. I de simuleringene der det er flere enn én forbindelse, er alle senderne lokalisert på samme stasjon, og alle mottakerene lokalisert på en annen stasjon. På denne måten får alle forbindelsene like stor propagasjonsforsinkelse (se figur 43). Som i forrige simulering, er ut-raten til RED-bufferet begrenset til 12.16Mbit/s, slik at det bygges opp en kø i bufferet. Det forventes derfor at alle TCP-forbindelsene skal dele denne båndbredden mellom seg. RED-bufferet er lokalisert på inngangen til ringen, det vil si ingress-noden.

5.3.3 Resultater og konklusjon

Resultatene fra de ni simuleringene er oppsummert i tabell 3. Den viser hvordan den totale gjennomstrømningen, det totale antall retransmisjoner og det totale antall timeout'er for alle forbindelsene, ved forskjellige

Simnr	minTh	# TCP Forb.	Gjennom strømning	Gj.sn. # retrans	Gj.sn. # timeout	# kastet (RED)
1	10	1	9,85Mb/s	400	19	617
2	15	1	11,07Mb/s	260	8	387
3	25	1	11,66Mb/s	118	4	181
4	10	2	10,35Mb/s	449	35	1262
5	15	2	10,44Mb/s	316	23	985
6	25	2	11,18Mb/s	170	12	544
7	10	10	9,33Mb/s	156	70	3507
8	15	10	10,71Mb/s	205	59	3511
9	25	10	11,15Mb/s	205	42	3473

Tabell 3: *Simuleringsresultater. Alle TCP-forbindelser sender gjennom ett RED-buffer med maksimal gjennomstrømning på 12.16 Mbit/s.*

Forb.nr	Gj.strømning	# Retrans	# Timeout
1	5,12Mb/s	449	36
2	5,23Mb/s	448	34

Tabell 4: *Per forbindelse-informasjon, simnr 4 i tabell 3. Begge forbindelsene sender gjennom ett RED-buffer med maksimal gjennomstrømning på 12.16Mbit/s*

kombinasjoner av minTh og antall TCP-forbindelser. I tillegg viser tabellen hvor mange pakker som ble kastet av RED.

Her kan man se at gitt et antall forbindelser, så vil den totale gjennomstrømningen ligge tettere oppunder den maksimale gjennomstrømningen (12,16Mbit/s) når den nedre grensen (minTh) heves. Dette bekrefter påstanden i 5.3.1 om at høyere ytelse oppnås med en høyere minTh. Vi kan også se at det er en klar sammenheng mellom den totale ytelsen og antall retransmisjoner, men det kommer ikke som noen overraskelse. Tabellen viser også at antall retransmisjoner er lavere enn det antallet RED kaster. Grunner er at en del segmenter retransmitteres i Slow Start, og disse er ikke tatt med her.

Når det gjelder den rettferdige fordelingen av båndbredde mellom flere TCP-forbindelser, viser tabell 4 og 5 henholdsvis resultatene fra en av simuleringene med to forbindelser, og med 10 forbindelser. I begge disse simuleringene var minTh satt til 10 pakker. I tilfellet med to forbindelser, hadde den ene en gjennomstrømning på 5,12Mbit/s, mens den andre hadde en gjennomstrømning på 5,23Mbit/s, som utgjør en forskjell på ca 2%.. Med tanke på usikkerheten som ligger i at pakker er i omløp når simuleringen avbrytes, i tillegg til at simuleringstiden

Forb.nr	Gj.strømning	# Retrans	# Timeout
1	1.04Mbit/s	161	68
2	1.06Mbit/s	167	69
3	0.72Mbit/s	118	72
4	1.00Mbit/s	161	70
5	1.19Mbit/s	170	69
6	0.83Mbit/s	138	72
7	0.82Mbit/s	178	70
8	0.85Mbit/s	153	70
9	0.77Mbit/s	142	72
10	0.99Mbit/s	171	70
Sum: Gj.sn: Std.avvik:	9.33Mbit/s 0.93Mbit/s 0.15Mbit/s		

Tabell 5: *Per forbindelse-informasjon, simnr 7 i tabell 3. Alle forbindelsene sender gjennom ett RED-buffer med maksimal gjennomstrømning på 12.16Mbit/s*

(100s) er relativt lang, kan denne forskjellen regnes som liten. Når antall forbindelser økes til 10, blir forskjellen mellom den som får mest (1,19Mbit/s) og den som får minst (0,72Mbit/s) båndbredde større. Distribusjonen av gjennomstrømninger har et standardavvik på 0.15Mbit/s, som utgjør ca 16%. Dette tallet er høyt. I praksis betyr det at i det mest ekstreme tilfellet, får forbindelse nr 3 en båndbredde som er ca 60% av den båndbredden forbindelse nr 5 får. Også her ligger det en usikkerhet i hvilken tilstand de to forbindelsene har når sumuleringen avbrytes, men den er ikke på langt nær så stor at det forklarer den store forskjellen. Hvis vi ser på grafene til tilstanden i RED-bufferet i figur 44, kan vi se at den gjennomsnittlige kølengden aldri slår seg til ro, men oscillerer kraftig hele veien. Til sammenlikning blir den gjennomsnittlige kølengden mer stabil når minTh økes. Figur 45 viser RED-bufferet i simulering nr 9, der minTh ble satt til 25 pakker. Standardavviket til båndbredden for de forskjellige forbindelsene ble noe lavere i denne simuleringen, noe som antyder at det er en indirekte sammenheng mellom minTh og RED-algoritmens rettferdighet. Man må også huske at disse simuleringene representerer ekstreme tilfeller der forbindelsene sender maksimal mengde data over lang tid. I tillegg har alle de 10 forbindelsene nøyaktig det samme sendemønsteret, som kan resultere i global synkronisering. Dermed er det tilfeldigheter (skeduleringsalgoritmen i simulatorkjernen) som avgjør hvilke forbindelser som får sende først. Jevnere resultater

kunne kanskje blitt oppnådd hvis denne simuleringen ble gjennomført noe annerledes, f.eks ha forskjellig avstand mellom sendere og mottakere, og innføre et mer tilfeldig sendemønster. Problemet da blir å sammenlikne de ulike forbindelsene, siden de da ikke skal ha den samme gjennomstrømningen.

Figurene 44 og 45, spesielt den siste, viser at det er en antydning til stabilisering av den gjennomsnittlige kølengden i nærheten av minTh. Det antyder at "samarbeidet" mellom RED og TCP fungerer som det skal. Begge simuleringene har også vist at pakker kastes av RED, og ikke fordi bufferet er fullt (med unntak av starten av noen simuleringene, der alle forbindelsene er i Slow Start samtidig (figur 45)). At det er store variasjoner i kølengden er naturlig, siden TCP sender pakker i klynger.

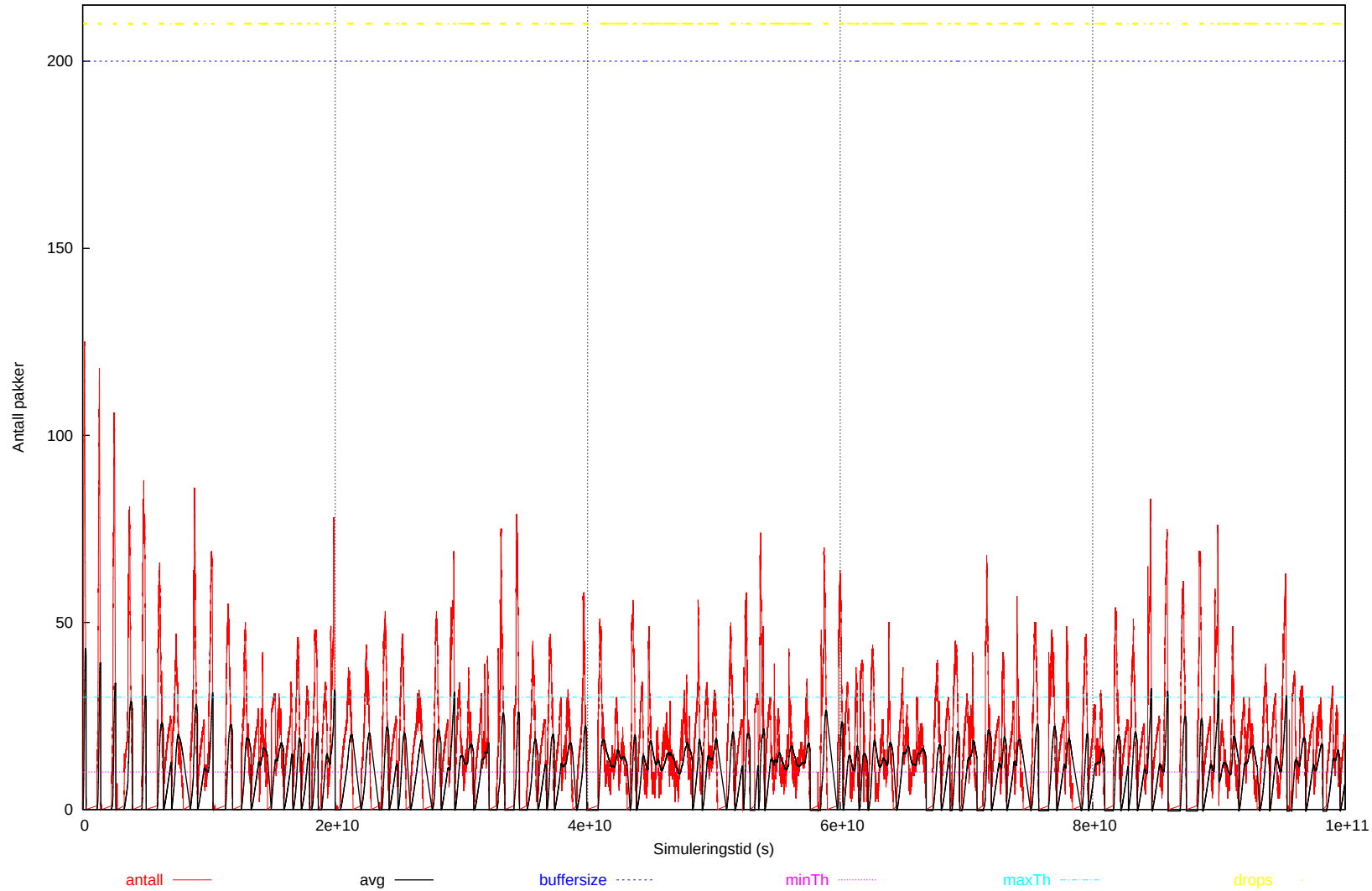
5.3.4 Usikkerhet

Det har allerede vært nevnt at det er knyttet usikkerhet til simuleringene. En årsak til usikkerhet er TCP-forbindelsenes tilstand når simuleringen avbrytes. En forbindelse kan ha et fullt vindu med data utestående, mens en annen kan ligge å vente på en timeout. Hvis en forbindelse har en total gjennomstrømning på f.eks 5Mbit/s og en RTT på ca 500μs, vil et fullt vindu inneholde ca 2,5Kbit med data. Jo lengre simuleringstid som benyttes, jo mindre vil dette ha å si for resultatet. Ved 100s simuleringstid tilsvarer dette 0,0005%, som er ubetydelig.

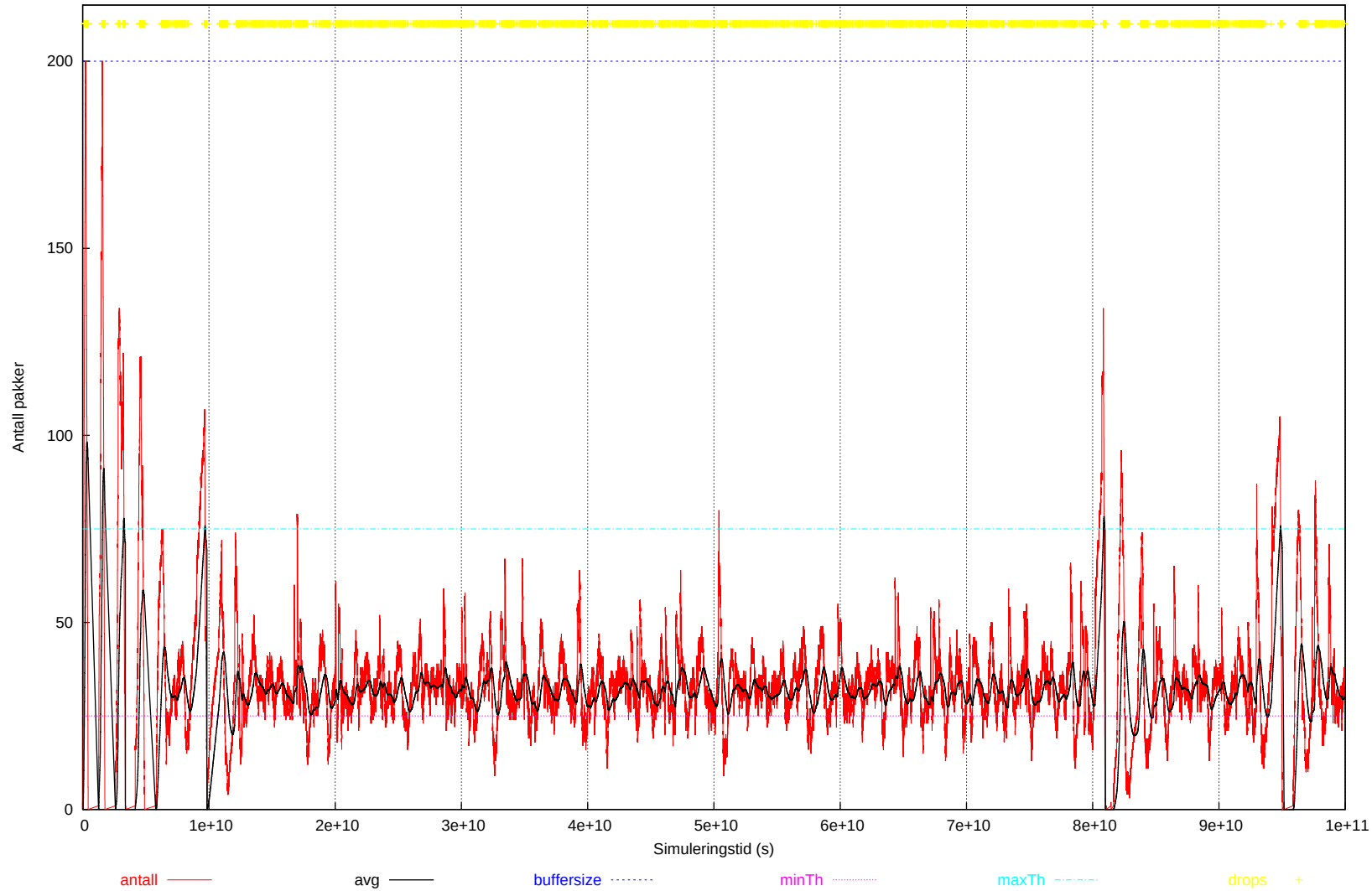
En annen årsak kan være selve simulatorkjernens skeduleringsalgoritme. Den fungerer slik at hvis to enheter (objekter av klassen Unit, se kapittel 4) ønsker å bli vekket opp på samme tidspunkt, vil den som “ytret ønske” om dette først, bli eksekvert først. I simuleringen der det var flere TCP-forbindelser, kan effekten av dette ha blitt at de forskjellige forbindelsene blir vekket opp i samme rekkefølge hver gang, siden alle startet å sende data helt likt (ved simuleringens start). Dette vil imidlertid bli jevnet ut etterhvert, siden RED velger å kaste “tilfeldige” pakker. Dermed vil en lang nok simuleringstid også her motvirke usikkerheten. Ved å la forbindelsene begynne å sende data på forskjellige tidspunkt, og å ha forskjellig propagasjonsforsinkelse mellom sendere og mottakere, vil denne usikkerhetsfaktoren forsvinne.

5.4 Om testing av simulatoren

Testene som er gjennomført og presentert i de foregående seksjonene, har antydet at de implementerte mekanismene oppfører seg som de skal. For å imidlertid kunne fastslå dette med tilstrekkelig grad av sikkerhet, må mer omhyggelig testing gjennomføres. Dette vil innebære å sørge for at alle grener av koden er eksekvert, og at alle aktuelle kombinasjoner av tilstander i de forskjellige simuleringsobjektene er prøvet



Figur 44: *RED-buffer - Simulering nr 7. 10 TCP-forbindelser, minTh=10*



Figur 45: *RED-buffer - Simulering nr 9. 10 TCP-forbindelser - minTh=25*

ut. Desverre ble det ikke anledning til slike grundige tester på grunn av tidsbegrensningene denne oppgaven var underlagt

Det er også mulige å gjennomføre "formell verifisering" av en implementasjon av en kommunikasjonsprotokoll, ved å komme frem til matematiske konklusjoner som bekrefter eller avkrefter implementasjonens korrekthet. Slike bevis er omfattende og kompliserte, og krever mye arbeid. Av den grunn er det heller ikke vanlig å benytte slike metoder på ordinære kommunikasjonsprotokoller. Mer vanlig er det å prøve ut implementasjonene ved å sette de ut i live, og se hvordan de fungerer i Internett, den største simulatoren av de alle.

Mer aktuelt ville det vært å sammenlikne simuleringer fra denne simulatoren med simuleringer fra andre nettverkssimulatorer som allerede har en TCP-modul. Ved å sette opp enkle identiske scenarier i begge simulatorene, ville det være enkelt å sammenlikne resultatene og vurdere eventuelle forskjeller. Desverre ble det ikke anledning til gjøre slike sammenlikninger i denne forbindelsen, men det er noe som bør gjennomføres før modulen benyttes til simuleringer.

5.5 Oppsummering

Dette kapittelet har presentert seks testsimuleringer som ble gjennomført for å se hvordan de ulike mekanismene i TCP og RED fungerer. Først ble metningskontroll, flytkontroll og timeoutmekanismen i TCP testet. Resultatene antydet at modulen oppførte seg som forventet i de scenariene som ble gitt. Deretter, i seksjon 5.2, ble RED-modulen testet. Resultatene her var også positive. Seksjon 5.3 presenterte en test som viste samarbeidet mellom RED og TCP. Denne testen bekreftet resultatene fra de foregående simuleringene. Til slutt i dette kapittelet ble det konkludert med at selv om testresultatene var positive, er mer grundig testing nødvendig for å kunne slå fast at implementasjon av de to modulene er korrekt.

6 Konklusjon og videre arbeid

Denne rapporten har beskrevet arbeidet med å utvide en Javabasert nettverkssimulator med en TCP-modul og en RED-modul. Nettverkssimulatoren skal primært benyttes til å analysere ulike aspekter ved et RPR-nettverk. For å kunne observere RPRs oppførsel når TCP-trafikk sendes over ringen, var det ønskelig å utføre eksekveringsdrevne simuleringer. Av den grunn var behovet for en TCP-modul og en RED-modul tilstede.

Målene med oppgaven var derfor å utvikle og evaluere disse to modulene, som tilsammen skulle gjøre det mulig å sende datastrømmer på RPR-ringen med et trafikkmønster som svarer til dagens TCP-trafikk.

Rapporten har gitt en innføring i TCP, RPR og RED. I beskrivelsen av TCP ble det lagt vekt på flyt og metningskontrollmekanismene, som er grunnlaget for TCP's sendemønster. Videre ble de forskjellige delene av simulatoren beskrevet, både de som allerede var implementert, og de to nye modulene. Forskjellige valg som ble gjort underveis er redegjort for, og feil og mangler er belyst. Avslutningsvis ble testsimuleringer presentert og diskutert, og det har blitt konkludert med at ytterligere testing er nødvendig før modulene tas i bruk.

6.1 Resultater og måloppnåelse

For å kunne avgjøre om hovedmålet med oppgaven ble nådd, er det først nødvendig å vurdere de forskjellige delmålene som ble satt underveis.

- *TCP - Fast Retransmit / Fast Recovery*
  Disse to mekanismene ble implementert, og simuleringsresultatet i seksjon 5.1.1 viste at de oppførte seg som forventet i det scenariet som ble gitt.
- *TCP - Slow Start & Congestion Avoidance*
  Disse to mekanismene ble implementert, og simuleringsresultatet i seksjon 5.1.1 viste at de oppførte seg som forventet i det scenariet som ble gitt.
- *TCP - Timeout-mekanisme*
  Denne mekanismen ble implementert, og testet i seksjon 5.1.2. Resultatet viste at den oppførte seg som forventet i det scenariet som ble gitt.
- *TCP - Timeout-beregning*
  Denne mekanismen ble implementert, og testet i seksjon 5.1.3. Resultat viste at RTO (Retransimition Timeout) ble beregnet riktig i scenariet som ble gitt. Det er imidlertid et problem at mekanismen operer med en klokke som er mer finkornet enn det som er vanlig i TCP, og sendemønsteret vil derfor bli forandret i visse situasjoner.

- TCP - SACK / NewReno
  Disse mekanismene ble ikke implementert med den begrunnelsen at oppgaven måtte avgrenses. Begge forbedrer TCPs ytelse ved tap av flere pakker innenfor samme vindu, ved at timeout-perioder med påfølgende Slow Start forhindres. Ved bruk av RED, vil fraværet av disse likevel ikke påvirke TCP sendemønster i altfor stor grad, siden sannsynligheten da minskes for at flere pakker fra samme vindu kastes.

- *RED - Beregning av gjennomsnittlig kølengde*
  Denne funksjonen ble testet i seksjon 5.2, og resultatet viste at den oppførte seg som forventet i det scenariet som ble gitt.

- *RED - Beregning av kaste-sannsynligheten*
  Denne funksjonen ble testet i seksjon 5.2, og resultatet viste at den oppførte seg som forventet i det scenariet som ble gitt.

- *Integrasjon av modulene i simulatoren*
  Denne jobben ble gjort. Til tross for enkelte problemer ble det utviklet enkle grensesnitt mellom de ulike modulene. Disse grensesnittene har blitt testet bl.a gjennom samtlige av de simuleringene som er presentert i denne rapporten.

6.2 Konklusjon

Det er nevnt at alle mekanismene bør testes grundigere, og sammenliknes med andre nettverkssimulatorer før det kan slås fast at de er korrekt implementert. Dette til tross for at simuleringsresultatene i denne rapporten har vist positive resultater. Disse resultatene antyder at flyt og metningskontrollen er i orden, og siden det er disse som hovedsakelig styrer sendemønsteret i TCP, er det lite som tyder på at trafikken denne modulen genererer vil skille seg merkbart fra ordinær TCP-trafikk.

Det er allikevel ett av delresultatene som bryter direkte med hovedmålet, og det er den finkornete timeoutklokken. Konklusjonen blir derfor at modulen ikke er optimal med hensyn på å generere "naturlig" TCP-trafikk, men at det sannsynligvis ikke er store forandringer som skal til for å si at hovedmålet er nådd. Hva som gjenstår oppsumeres i neste seksjon.

API'et (Application Programming Interface) TCP-modulen tilbyr applikasjonsmodulene er utviklet med tanke på at det skal være lett for kommende brukere av simulatoren å sette opp TCP forbindelser over RPR-ringen, f.eks for å kjøre simuleringer med FTP-trafikk. Dette grensesnittet har blitt testet ut i forbindelse med alle testsimuleringene som er gjennomført, og har vist seg å være enkelt å bruke. Har man erfaring

med noe socketprogrammering, vil man kjenne igjen semantikken som er benyttet her.

Arbeidet med å utvikle disse to modulene har vært lærerikt på flere områder. Åpenbart har det ført til detaljkunnskap om TCP og RED, og noe mer generell kunnskap om RPR. Man lærer også å hvor man skal lete etter dokumentasjon og spesifikasjoner, og hvordan man plukker ut relevant informasjon fra disse. I tillegg har det gitt innsikt i hvordan kommunikasjonen foregår i fagmiljøet, og ikke minst hvordan en ny protokollstandard utvikles.

Når det gjelder fordelingen av utviklingsarbeidet, har søking etter litteratur, og det å sette seg inn i spesifikasjonene absolutt vært en dominerende aktivitet. Anslagsvis har 50-60% av tiden gått med til dette, mens resterende andel er fordelt mellom koding, testing og debugging. Totalt består de to modulene av ca 1500 linjer med kode.

6.3 Videre arbeid

For at de to modulene skal kunne benyttes til simuleringer, er det som nevnt i avsnitt 5.4 anbefalt å utføre grundigere testing og sammenlikninger med andre nettverkssimulatorer. Dette vil være et omfattende arbeid som krever god innsikt i TCP og RED. I tillegg er det for TCP-modulens vedkommende to deler som anbefales utbedret.

- *Timeout-klokken*
  TCP sjekker vanligvis om det har oppstått en timeout hvert 500ms (implementasjonsavhengig), mens TCP-modulen i denne simulatoren i praksis sjekker dette hvert nanosekund. Det er kun enkle endringer i koden som skal for å rette opp dette (f.eks alltid runde RTO opp til nærmeste 500ms). Dette er beskrevet i avsnitt 4.6.3.

- *Selective Acknowledgements (SACK)/ NewReno-utvidelsene*
  De aller fleste TCP-implementasjoner inneholder en av disse utvidelsene, som gjør at TCP kan forhindre å vente i en lang periode på en timeout, med etterfølgende Slow Start når flere pakker mistes innenfor samme vindu. Isteden retransmitteres de tapte segmenten under Fast Recovery-perioden. På grunn av tidsbegrensninger måtte disse velges bort, selv om det var ønskelig å ha med en av dem. Siden NewReno krever langt færre endringer i koden enn SACK, anbefales det å velge denne. Dette er beskrevet i avsnitt 4.4.2 og 4.6.4.

Når det gjelder RED-modulen, har det ikke blitt avdekket noen mangler, men også her bør grundigere testing utføres. Utfordringen med denne modulen er imidlertid å bestemme verdiene på RED-parameterne (MaxTh, MinTh, MaxP, weight og bufferstørrelse), slik at kastepolitikken tilpasses

RPR-ringen og trafikkmengden. I denne rapporten er det ikke gjort noen grundige vurderinger på dette området, men det er noe som bør gjøres før simuleringer med RED gjennomføres.

Dersom kommende brukere av simulatoren ønsker å legge til utvidelser eller gjøre endringer i TCP-modulen, bør ikke det by på store problemer. Koden er nøye kommentert, og modulen er utviklet med stor vekt på at koden skal være enkel å forstå fremfor at den skal være effektiv å kjøre. Bruken av et objektorientert språk er langt på vei en faktor som har gjort dette mulig.

Referanser

[1] K.C. Claffy, "Internett measurements: State of DeUnion", Url: (30.04.2003)

http://www.caida.org/outreach/presentations/Soa9911/mgp00020.html

[2] Url: (30.04.2003)

http://www.opnet.com/

[3] Url: (30.04.2003)

http://www.isi.edu/nsnam/ns/

[4] Floyd, S., and V. Jacobson, "Random Early Detection gateways for Congestion Avoidance", IEEE/ACM Transactions on Networking, Vol.1, No.4, 1993, pp.397-413.

[5] "ISO/IEC 7498-1 1994, Information technology - Open System Interconnection - Basic reference model: The BasicModel"

[6] "Round Trip Time Estimation," P. Karn & C. Partridge, ACM SIGCOMM-87, August 1987.

[7] "Requirements for Internet Hosts – Communication Layers," B. Braden, ed., RFC 1122, Oktober 1989.

[8] "Computing TCP's Retransmission Timer", Vern Paxson, Mark Allman, RFC 2988, November 2000.

[9] "Window and Acknowledgment Strategy in TCP," D. Clark, RFC-813, Juli 1982.

[10] "TCP/IP Illustrated Vol.1: The Protocols", W. R. Stevens, 1994.

[11] "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms", W. Stevens, Januar 1997.

[12] "TCP Vegas: End to End Congestion Avoidance on a Global Internet", Lawrence S. Brakmo, Larry L. Peterson, June 1995.

[13] "The NewReno Modification to TCP's Fast Recovery Algorithm", Floyd, S. and T. Henderson, RFC 2582, April 1999.

[14] “Jacobson, V., "Congestion Avoidance and Control", Computer Communication Review, vol. 18, no. 4, pp. 314-329, Aug. 1988.”

[15] "4BSD Header Prediction", Van Jacobson, ACM Computer Communication Review, April 1990.

[16] “TCP big window and NAK options”, R. Fox, RFC 1106, Juni 1989.

[17] “Problem with the TCP big window option”, A.M. McKenzie, RFC 1106, August 1989.

[18] "TCP Extensions for High Performance", V. Jacobson, R. Braden ,D. Borman, RFC 1323, Mai 1992.

[19] “TCP Selective Acknowledgement Options”, M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, RFC 2018, Oktober 1996.

[20] “Simulation-based Comparisons of Tahoe, Reno, and SACK TCP” Kevin Fall, Sally Floyd, Computer Communication Review, July 1996.

[21] "Avoiding Head-of-Line Blocking using an Enhanced Fairness Algorithm in a Resilient Packet Ring", Stein Gjessing, Fredrik Davik, ICT 2002, Juni 2002.

[22] IEEE Draft P802.17/D2.2, "Resilient Packet Ring Access Method & Physical Layer Specifications", versjon 2.2, April 2003.

[23] ”A Binary Feedback Scheme for Congestion Avoidance in Computer Networks with a Connectionless Network Layer”, K.K. Ramakrishnan, Raj Jain, Mai 1990.

[24] K. Ramakrishnan, S. Floyd, A Proposal to add Explicit Congestion Notification (ECN) to IP, RFC 2481, January 1999.

[25] Url: (30.04.2003)

`http://www.icir.org/floyd/REDparameters.txt`

[26] "Transmission Control Protocol," J. Postel, RFC-793, September 1981

[27] ”Karn, P. and C. Partridge, "Improving Round-Trip Time Estimates in Reliable Transport Protocols", SIGCOMM 87.”

[28] "TCP Congestion Control", Stevens, W., Allman, M. and V. Paxson, RFC 2581, April 1999.

[29] Url: (30.04.2003)

http://www.microsoft.com/windows2000/docs/tcpip2000.doc

[30] Url:

http://lxr.linux.no/source/net/ipv4/tcp.c?v=2.2.20

Ordliste og Forkortelser

Ack
Kvittering (Acknowledgement).

API
Application Programming Interface.

ATM
Asynchronous Transfer Mode.

Backoff
Tilbaketrekning. Venter en periode før man prøver på nytt.

Buffer Avgrenset minneområde hvor data kan lagres.

Beskyttelse
Eng: "Protection". Tiltak som gjøres for å rute trafikk utenom et kabelbrudd eller en ruter/svitsj/RPR-stasjon som har feilet.

Datapath
Elektriske ledninger (stier) som leder data-signalene.

Demultipleksing
Motsatt av multipleksing (se "multipleksing). Splitter ett signal inn i flere separate signaler.

Duplex
Brukes om en fysisk eller logisk forbindelse der det er mulig å sende data i begge retninger samtidig.

FBP
Forsinkelse * Båndbredde produktet.

Klynge
Eng: "burst". Utrykker at pakker sendes i klynger, periodevis.

Lavpassfilter
Uttrykk hentet fra elektronikken. Brukes for å fjerne støykomponenter fra et signal.

LAN
Local Area Network.

MAC
Medium Access Controll.

MAN
Metropolitan Area Network.

Multipleksing
Kombinere flere signaler inn på ett signal. Brukes både i fysisk og logisk sammenheng.

PAWS
Protection Against Wrapped Sequence numbers.

Portner
Eng: "Gateway". Kantruter. Ruter som står ved inngangen/utgangen til et domene.

PoS
Packet over SONET.

QoS
Quality of Service (Tjenestekvalitet).

RED
Random Early Detection.

RPR
Resilient Packet Ring.

RTT
Round Trip Time.

RTO
Retransmition Timeout.

SACK
Selective Acknowledgements.

SONET
Synchronous Optical NETwork.

SRTT
Smoothed RTT.

ssthresh
Slow Start Treshold.

Steering
Navn på en mekanisme i RPR som sørger for at beskyttelse (se "beskyttelse") utføres ved at avsenderen benytter den andre enkeltringen.

TCP
Transmition Control Protocol.

Timeout
Oppstår når en timer går av.

Timer
En vekkeklokke-mekanisme som settes til å gå av etter en bestemt tidsperiode.

Trafikkforming
Eng: "Shaping". En mekanismen som brukes for å "forme" dataflyter, slik at de ikke benytter mer båndbredde enn de har krav på.

Trigge
Utløse. Sette i gang.

UML
Unified Modelling Language.

WAN
Wide Area Network.

Wrappe
Engelsk uttrykk. Brukes om situasjonen der sekvensnummerrommet er brukt opp og starter på nytt igjen.

Wrapping
Navn på en mekanisme i RPR som sørger for at beskyttelse (se "beskyttelse") utføres lokalt i en stasjon ved at fra den ene enkelt-ringen (se "enkelt-ring") sendes tilbake på den andre enkelt-ringen.

Figurer

Tabeller