TCP/IP-stack extensies

Onderzoeksproject in opdracht van SURFnet

Pieter de Boer Martijn P. Rijkeboer

30 juni 2005

Samenvatting

De oorspronkelijke specificatie van de TCP/IP protocolfamilie is minder geschikt voor verbindingen met een hoog bandbreedte maal round-trip time product. Deze verbindingen komen echter steeds meer voor op het Internet. Het nadeel hiervan is, dat TCP op dergelijke verbindingen de maximaal haalbare doorvoersnelheid niet kan halen.

Het doel van dit project was het onderzoeken welke configuratie-opties en extensies de doorvoersnelheid van TCP op dergelijke verbindingen zouden kunnen verbeteren. Een aantal extensies en configuratie-opties worden uitgebreid besproken. Daarnaast zijn een aantal van deze configuratie-opties en extensies in een testomgeving getest, waarvan de resultaten in dit verslag te vinden zijn.

Tijdens deze tests bleek echter dat de testomgeving zich niet gedroeg zoals, op grond van de hardwarespecificaties, verwacht kon worden. De doorvoersnelheid bij verbindingen waarbij een grote latency werd gesimuleerd, bleef sterk achter bij de verwachte snelheid. Aangezien de betrouwbaarheid van de tests hierdoor in het geding kwam, is een onderzoek gestart naar de mogelijke oorzaken van deze afwijking. Dit onderzoek heeft een aantal interessante resultaten opgeleverd, die in dit verslag gepresenteerd worden.

Inhoudsopgave

Hoofdstuk 1

Inleiding

1.1 Probleemstelling

In de loop der jaren is de beschikbare bandbreedte op het Internet exponentieel toegenomen. De TCP/IP protocolfamilie, die de basis van het Internet vormt, is echter nagenoeg hetzelfde gebleven. Het gevolg van het onveranderd blijven van het protocol is, dat het betrouwbare TCP [16], dat gebruikt wordt voor het gros van het dataverkeer op het Internet, niet voldoende is afgestemd op de huidige beschikbare bandbreedte. Met name op verbindingen met een hoge bandbreedte en/of een hoge latency blijft de doorvoersnelheid van TCP achter bij de theoretisch haalbare doorvoersnelheid van de verbinding. Het beschikbaar komen van meer bandbreedte zal dit probleem in de toekomst alleen maar groter doen worden.

1.2 Doel

Het doel van het project is het onderzoeken welke mogelijkheden extensies en configuratie-opties bieden om de performance van TCP te verbeteren. Hierbij zal het eerste gedeelte van het onderzoek bestaan uit het onderzoeken welke extensies en configuratie-opties er zijn om dit te bereiken en wat de theoretisch verbeteringen hiervan zijn. Het tweede gedeelte van het onderzoek bestaat uit het testen van deze extensies en configuratie-opties in een testomgeving met als doel het achterhalen in hoeverre deze theoretische verbeteringen in de praktijk effect sorteren. De hierbij verkregen gegevens zullen onder andere worden gebruikt om de performance van de dienst “Kanon” zo optimaal mogelijk te kunnen maken. Deze dienst heeft tot doel het kunnen simuleren van (D)DoS aanvallen op machines en services en het kunnen meten van de prestaties van over het netwerk bereikbare machines en services.

1.3 Projectorganisatie

De opdrachtgever van dit project is dhr. J.J. Meijer, werkzaam bij de afdeling Middleware Services van SURFnet B.V. [18]. Dit bedrijf verzorgt het Nederlandse onderzoeksnetwerk hetgeen universiteiten, hogescholen, onderzoekscentra, academische ziekenhuizen en wetenschappelijke bibliotheken met elkaar en het Internet verbindt en is daarnaast onder meer verantwoordelijk voor de uitvoering van het Gigaport project.

De begeleiding vanuit de masteropleiding Systeem- en Netwerkbeheer wordt verzorgt door dhr. dr.ir. C.Th.A.M. de Laat en dhr. dr. C.P.J. Koymans. Beiden zijn verbonden aan de Universiteit van Amsterdam, Faculteit der Natuurwetenschappen, Wiskunde en Informatica.

1.4 Relevantie

Dit document is in eerste instantie gemaakt voor eigen gebruik door SURFnet alsook vanuit de voorlichtingstaak van SURFnet expliciet gericht op de bij SURFnet aangesloten instellingen.

Hoofdstuk 2

TCP/IP

2.1 Internet Protocol versie 4

2.1.1 Inleiding en featureset

Het Internet Protocol versie 4 (IPv4), zoals beschreven in RFC 760 [14] en RFC 791 [15], wordt gebruikt op het Internet als het transportprotocol. IPv4 draagt zorg voor het afleveren van datagrammen over een pakket geswitcht netwerk op basis van target adressen, 32 bits in lengte. Andere functies van IP zijn fragmentatie en samenvoegen van grote pakketten, quality of service (QoS) en controle van de integriteit van de packet header door middel van een checksum. Een time to live teller wordt gebruikt om pakketten die zich te lang in het netwerk bevinden te verwijderen. Het IP-protocol ondersteunt hiernaast nog de mogelijkheid extra opties toe te voegen aan de header, waarmee extra functionaliteit wordt geboden.

2.1.2 Ontbrekende features

Het IP-protocol kan gebruikt worden om op een efficiënte wijze gegevens tussen twee hosts uit te wisselen. Er kleven echter een aantal nadelen aan het protocol wanneer het gebruikt zou worden voor datatransmissie over het Internet:

- Alleen de header van een IP-pakket wordt gecontroleerd door middel van een checksum, de data in een IP-pakket kan beschadigd raken zonder dat het door de IP-laag opgemerkt kan worden.
- Er is geen *ordering* van IP-pakketten: IP-pakketten kunnen in een andere volgorde aankomen dan dat ze uitgestuurd zijn, hetgeen het samenvoegen van de ontvangen gegevens bemoeilijkt.
- Er is geen voorziening voor hertransmissie van verloren gegane pakketten in het protocol opgenomen.

- Het IP-protocol biedt geen mogelijkheden tot *flow-control* en *congestion control*, waarbij de zender ervoor zorgt dat er geen pakketten worden uitgestuurd die niet door het netwerk of door de ontvanger verwerkt kunnen worden.
- Het IP-protocol werkt met losstaande pakketten; er is dus geen notie van een *verbinding* tussen twee IP-sprekende hosts. Voor het multiplexen van verschillende gegevensstromen tussen twee hosts is dan ook geen voorziening opgenomen in het protocol.

Het IP-protocol is uitermate geschikt om pakketten van een host op het Internet naar een andere host te kunnen versturen. Het protocol is simpel en daardoor snel te verwerken, de adresruimte is vooralsnog voor de meeste netwerken en toepassingen groot genoeg gebleken[1] en het protocol kan prima gebruikt worden om andere protocollen in te encapsuleren.

Eén van de protocollen die in IP wordt geëncapsuleerd is het TCP-protocol. Dit protocol ondersteunt de hierboven genoemde features juist wel, waardoor het TCP-protocol, ingepakt in IP-pakketten, gebruikt kan worden voor datatransmissies over het Internet.

2.2 Internet Protocol versie 6

Het Internet Protocol versie 6 is de opvolger van versie 4. Het biedt een grotere adresruimte, versimpelde header, extra functionaliteit voor IPSec, automatische configuratie van IP-adressen, enz. IPv6 wordt gezien als het internet protocol van de toekomst en vindt steeds meer ingang bij grotere spelers op de markt, vooral in Azië. In Europa wordt IPv6 nog maar zeer beperkt door eindgebruikers toegepast en bieden ook nog maar weinig providers het aan hun klanten aan. SURFnet is één van de partijen die IPv6 wél aanbiedt. De hogescholen en universiteiten van Nederland kunnen *native* IPv6 van SURFnet afnemen en dat aanbieden aan de eindgebruikers. Daarmee is de relevantie van IPv6 voor ons project bepaald.

IPv6 is net als IPv4 nog steeds een connectieloos protocol met een beperkte feature set. De features als congestion control, packet ordering en multiplexen van verbindingen zijn niet in IPv6 zelf geïmplementeerd, maar worden aan de hogere lagen gelaten. IPv6 is daarin niet anders dan IPv4.

Er zijn voor het gebruik van TCP over IPv6 geen aanpassingen gedaan aan TCP. De hier beschreven en geteste configuratie-opties en extensies op TCP gelden dus voor zowel TCP over IPv4 als TCP over IPv6. Er kunnen echter wel verschillen bestaan tussen de maximale performance van een TCP-verbinding over IPv4 in vergelijking met de maximale performance van een TCP-verbinding over IPv6. Deze verschillen worden veroorzaakt door:

[1] Hoewel er sinds het ontstaan van IP andere addresseringsmethoden zoals VSML en CIDR nodig zijn gebleken, is de adresruimte van 32 bits nog steeds groot genoeg voor het overgrote gedeelte van de hosts op het Internet

- IPv6-adressen zijn vier maal langer dan IPv4-adressen. Het verwerken ervan kan daardoor langer duren.
- De IPv6-header bevat geen checksum, de IPv4-header wel. Checksums uitrekenen kost doorgaans veel tijd (hoewel dat voor de IPv4-header wel meevalt door de beperkte lengte).
- Een aantal netwerkkaarten kan zelf de IP en TCP-checksums uitrekenen, zodat het besturingssysteem die kostbare taak aan de netwerkkaart over kan laten. De netwerkkaart moet ook ondersteuning hebben voor IPv6 om TCP-checksums te kunnen maken voor TCP-segmenten die in IPv6-datagrammen zijn geëncapsuleerd en om ontvangen IPv6-pakketten met daarin TCP-segmenten te kunnen controleren.
- De IPv6-code van besturingssystemen en hardware is nog niet zo oud als de IPv4-code. In de loop der jaren is de afhandeling van IPv4-pakketten steeds verder geoptimaliseerd, terwijl dit bij IPv6 nog lang niet zoveel gebeurd is. Daardoor kan de performance van de verwerking van IPv6-pakketten achterblijven ten opzichte van IPv4.

Gezien de ons beschikbare tijd is IPv6 out of scope verklaard voor het project. De tests hebben echter alleen betrekking op TCP en kunnen dus (door anderen) ook voor TCP over IPv6 uitgevoerd worden. De gebruikte tool *Iperf* (beschreven in hoofdstuk 4.4.1) ondersteunt ook IPv6.

2.3 Transmission Control Protocol

2.3.1 Inleiding en featureset

Het Transmission Control Protocol (TCP) [16] wordt gebruikt om verbindingen op te zetten tussen twee hosts op een IP-netwerk om over die verbinding data te versturen. TCP zorgt voor de afhandeling van onderstaande punten:

- Dataintegriteit door middel van een checksum op het gehele TCP-segment plus een virtuele header (met daarin informatie uit de IP-header), die vóór het TCP-segment wordt geplaatst bij het uitrekenen van de checksum maar niet wordt verstuurd.
- TCP is een *betrouwbaar* protocol: segmentverlies wordt ondervangen door middel van acknowledgements. Als verloren beschouwde segmenten worden opnieuw gestuurd.
- Data uit verstuurde segmenten wordt aan de ontvangende kant op de juiste volgorde aan de hogere lagen doorgegeven. Er is dus sprake van packet ordering.

- TCP maakt gebruik van verbindingen: virtuele circuits tussen twee hosts, geïdentificeerd met een target en source IP-adres en TCP-poorten. Meerdere verbindingen tussen twee hosts zijn mogelijk door middel van de source en target TCP-poorten.
- TCP zorgt voor het afhandelen van flow-control en doet aan congestion control. Dit gebeurt door middel van een *sliding window* met variabele grootte. Dit principe wordt in paragraaf 2.3.3 verder uitgelegd.
- Naast IP doet ook TCP aan fragmentatie, zodat applicatiesoftware data in blokken van variabele grootte over een TCP-verbinding kan sturen. TCP zorgt voor het opdelen van de data in segmenten met de juiste grootte.

Al deze eigenschappen hebben het mogelijk gemaakt dat het TCP-protocol al ongeveer dertig jaar succesvol in gebruik is op het Internet. Anno 2005 voldoet het voor de meeste toepassingen nog steeds, hoewel er wel extensies zijn bedacht om het protocol te verbeteren. Toch kan gesteld worden dat het TCP-protocol zoals dat in de jaren '70 van de vorige eeuw bedacht is, nog steeds succesvol is en waarschijnlijk de komende jaren dat nog zal blijven.

2.3.2 Verbindingseigenschappen

De in de vorige paragraaf besproken eigenschappen kunnen alleen bewerkstelligd worden door middel van voorzieningen in het protocol zelf. In deze paragraaf worden de eigenschappen van het protocol die bepalen dat TCP *verbindingen* tussen twee hosts kan opzetten, onderhouden en afbreken besproken. Ook wordt het multiplexing-mechanisme besproken.

TCP gebruikt de zogenaamde *three way handshake* om een verbinding op te zetten tussen twee hosts. De three way handshake werkt als volgt:

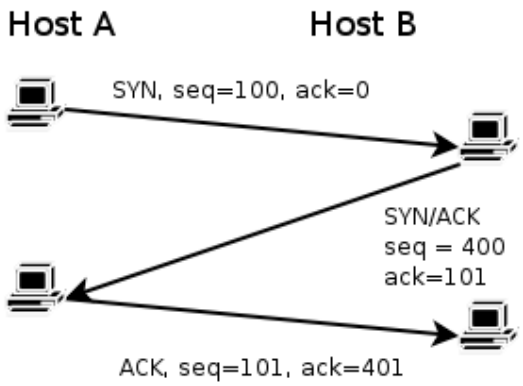
- Host A stuurt een TCP-segment naar host B met de SYN-vlag gezet.
- Host B stuurt een TCP-segment naar host A met zowel de SYN als ACK-vlag gezet.
- Host A stuurt een TCP-segment naar host B met de ACK-vlag gezet.

De SYN-vlag geeft aan dat een host een verbinding wil synchroniseren. De ACK-vlag wordt gebruikt om een ontvangen segment te acknowledgen. Een combinatie wil dus zeggen dat een host een inkomend SYN-segment acknowledged en zelf ook de verbinding wil synchroniseren. De laatste ACK geeft aan dat host A de synchronisatiepoging van host B accepteert en vanaf dat punt de verbinding als geopend beschouwt. Host B beschouwt de verbinding als geopend na het ontvangen van de ACK van host A.

Het synchroniseren van een verbinding houdt in dat twee hosts de te gebruiken *sequence numbers* en *window sizes* afspreken. Op het window en de window sizes wordt in paragraaf 2.3.3 teruggekomen. Sequence numbers zijn echter van

belang bij het opzetten, onderhouden en verbreken van een verbinding, daarom wordt dit hier besproken.

Bij het opzetten van de verbinding sturen beide hosts, in de segmenten met de SYN-vlag, elkaar hun *initial sequence number*. Deze sequence nummers worden gebruikt door de hosts om ontvangen data, per octet, te acknowledgen. Een segment met 100 octetten aan data zorgt dus voor een verhoging van het sequence nummer met 100. Figuur 2.1 is een voorbeeld van het opzetten van een TCP-verbinding, waarbij het gebruik van de sequence nummers duidelijk wordt gemaakt.



Figuur 2.1: Het opzetten van een TCP-verbinding

Tijdens de data-overdracht worden de sequence nummers gebruikt voor twee doeleinden:

- Het aangeven met welk octet in de datastroom het huidige segment begint. Dit wordt aangegeven in de TCP-header in het 'sequence number'-veld.
- Het aangeven welk octet wordt verwacht, waarmee ook wordt aangegeven welke octetten al zijn ontvangen. Dit wordt aangegeven in de TCP-header in het 'acknowledgement number'-veld. De ACK-vlag moet gezet zijn wanneer het segment een acknowledgement is.

De ontvangende kant kan door middel van het sequence nummer in de TCP-header TCP-fragmenten samenvoegen en doorsturen naar hogere lagen. Door middel van het acknowledgementnummer wordt de verzendende kant ervan op de hoogte gesteld dat alle octetten tot aan het acknowledgementnummer aan de ontvangende kant binnen zijn gekomen. Wanneer een acknowledgement niet binnen een bepaalde tijd wordt ontvangen, wordt er een recovery in gang gezet: de mogelijk verloren gegane segmenten worden opnieuw gestuurd. De zendsnelheid wordt hierbij verminderd, omdat het verloren gaan van segmenten een indicatie kan zijn van congestie. Door bij segmentverlies de zendsnelheid te

verlagen, kan congestie worden voorkomen en kan de bandbreedte van een verbinding zo efficiënt mogelijk gebruikt worden. In paragrafen 2.3.3 en 2.3.4 wordt hier dieper op in gegaan.

Het afsluiten van een verbinding gebeurt door middel van de FIN-vlag. Wanneer een segment ontvangen wordt met de FIN-vlag, beschouwt de ontvangende partij de verbinding gesloten vanuit de richting van de verzendende partij. Een half-gesloten verbinding is dus mogelijk. Ontvangen data wordt dan niet meer geaccepteerd, maar er kan nog wel data worden verstuurd. Wanneer beide hosts van een verbinding een segment met de FIN-vlag gestuurd en ontvangen hebben, en daar acknowledgements van binnen gekomen zijn, is de verbinding in principe gesloten: er kan geen data meer gestuurd worden en voor de hogere lagen lijkt de verbinding gesloten. Besturingssystemen ruimen echter pas na een vertraging de gegevens behorend bij de TCP-verbinding op. Dit wordt gedaan zodat de kans dat snel na elkaar opgezette verbindingen met elkaar zullen overlappen nihil wordt.

Multiplexing wordt in TCP geïmplementeerd met behulp van twee getallen: de source en target port. Samen met de source en target IP-adressen identificeren de source en target TCP-poorten de verbinding. Voor de poortnummers zijn 16 bits gereserveerd in de TCP-header, dat een aantal mogelijkheden van 65536 oplevert. Poortnummer "0" is echter niet toegestaan, wat dus de reeks $1 - 65535$ als geldige poortnummers oplevert. Een lijst van standaard poortnummers wordt bijgehouden door IANA [9]. Die standaard poorten zijn gekoppeld aan een dienst die aangeboden wordt "op" de desbetreffende poort. De bekende voorbeelden zijn poort 25 voor SMTP en poort 80 voor HTTP.

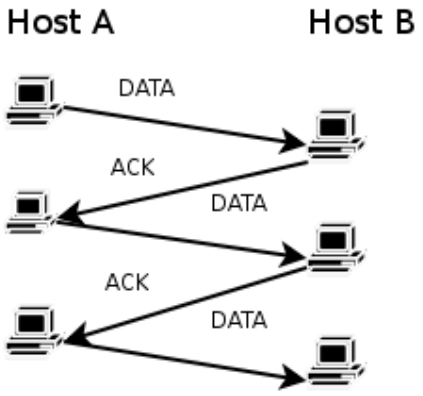
2.3.3 Sliding window

TCP maakt gebruik van het principe van een *sliding window*. Om dit principe uit te leggen, zal er eerst een wat simpeler protocol worden beschouwd. Dit protocol wordt het stop-and-wait protocol genoemd. Uit figuur 2.2 zal duidelijk worden waarom.

Host A begint met het sturen van een datasegment naar host B, die een acknowledgement stuurt. Pas na het ontvangen van de acknowledgement stuurt host A een nieuw datasegment. Wanneer dit protocol over een link met hoge latency wordt gebruikt, zal het zeer inefficiënt werken.

Om de efficiëntie te vergroten, kan TCP meer dan één segment tegelijkertijd versturen zonder per segment te hoeven wachten op een acknowledgement. Figuur 2.3 geeft weer hoe dataoverdracht in het geval van TCP plaats kan vinden.

Een host kan dus meerdere segmenten tegelijkertijd uitgestuurd hebben zonder dat voor het eerste segment een acknowledgement binnen moet zijn. Het sequence-number van de eerste octet waarvoor nog geen acknowledgement is ontvangen geldt als begin van het *window*. De bovenkant van het window wordt bepaald door een aantal factoren: het maximale receive window van de tegenpartij, eventuele congestion recovery en het principe van *slow start*. Slow start



Figuur 2.2: Dataoverdracht via het stop-and-wait protocol

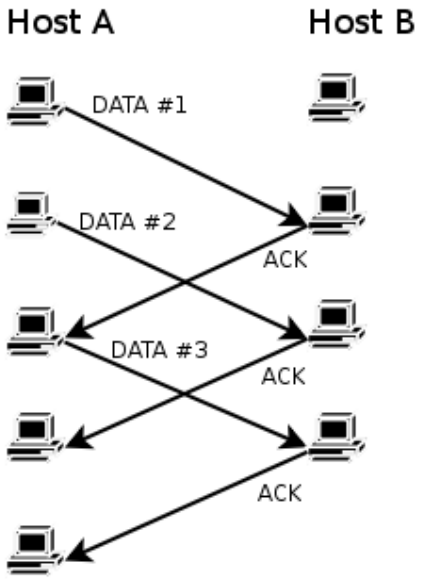
houdt in dat bij na het opzetten van een verbinding, of na het detecteren van congestion, het window van de versturende partij expres klein worden gehouden. Nadat er een acknowledgement van een verstuurd segment binnenkomt, wordt het send window verdubbeld, zodat de zendsnelheid toeneemt. Wanneer het window een bepaalde threshold *sshthresh* overschrijdt, wordt het congestion avoidance-algoritme ingezet. Hierbij wordt het window niet meer exponentieel vergroot maar lineair, namelijk met de segmentgrootte. Het window wordt vergroot totdat er congestie ontstaat (en dus segmentverlies) of totdat het gelijk is aan het maximale send window of de maximale receive window van de tegenpartij. De term "sliding window" komt voort uit het gedrag van de sequence nummers die de onder- en bovenkant van het window bepalen, ten opzichte van de totale sequence space van 32 bits. Het window "slide" per ontvangen acknowledgement een stukje verder door de sequence space door de oplopende sequence nummers.

2.3.4 Hertransmissie na gegevensverlies

Om TCP *betrouwbaar* te maken, dat wil zeggen: de hogere lagen de garantie te kunnen bieden dat verzonden data aankomt ondanks segmentverlies, moet er een vorm van hertransmissie in het protocol ingebouwd zijn. Om hertransmissie van verloren segmenten te kunnen doen, moet een host geïnformeerd worden over welke segmenten niet aan zijn gekomen. De oorspronkelijke, in de RFC beschreven, hertransmissiemethode was als volgt:

- Bereken de round trip time (RTT) aan de hand van het tijdstip dat een segment is verstuurd en de acknowledgement van dat segment is binnengekomen.
- Bereken een smoothed RTT (SRTT). Hiervoor wordt de RTT gecombineerd met de vorige RTT's door middel van onderstaande formule:

$$SRTT = (ALPHA * SRTT) + ((1 - ALPHA) * RTT)$$



Figuur 2.3: Dataoverdacht via een sliding window protocol

$ALPHA$ is hierbij de *smoothing factor*, die bepaalt welk percentage van de RTT wordt gebruikt en welk percentage van de SRTT wordt gebruikt om de nieuwe SRTT uit te rekenen.

- Bereken de retransmission timeout (RTO) als volgt:

$$RTO = min[UBOUND, max[LBOUND, (BETA * SRTT)]]$$

$UBOUND$ is de maximale waarde voor de RTO, bijvoorbeeld een minuut. $LBOUND$ is de minimale waarde, bijvoorbeeld een seconde. $BETA$ is de vertragingsfactor, bijvoorbeeld 1.3 tot 2.0. Als $BETA$ 2.0 is, wordt de RTO dus twee maal zo groot als de smoothed round trip time, binnen de door $LBOUND$ en $UBOUND$ gestelde limieten.

- Als voor een segment langer dan de waarde van RTO geen acknowledgement is ontvangen, stuur het segment dan opnieuw en laat een nieuwe timeout timer ingaan, met een verhoogde RTO.

Een verloren gegaan segment zorgt er bij gebruik van dit algoritme voor dat eerst de RTO moet aflopen voordat het segment opnieuw gestuurd wordt. Omdat TCP-segmenten op volgorde afgehandeld worden, betekent dit dat een verloren gegaan segment de aflevering van daarop volgende eventueel wél aangekomen segmenten tegenhoudt totdat de RTO-timer is afgelopen en het segment opnieuw gestuurd en ontvangen is. Hierdoor stort de doorvoersnelheid in bij segmentverlies.

Er dient een bepaalde timeout te zijn voor het opnieuw versturen van segmenten. Er zijn talloze situaties te bedenken waarvoor geldt dat een host een TCP-verbinding verbroken heeft, terwijl de andere kant daarvan niet op de hoogte

is. Wanneer er geen timeout zou zijn op de hertransmissies, zou een host niet ophouden met segmenten opnieuw te sturen, wat een ongewenste situatie zou zijn. Daarom is er in RFC 1122 [3] bepaald dat een host het volgende algoritme moet volgen wanneer er veel hertransmissies zijn:

- Er zijn twee timeouts: R1 en R2. R2 dient groter te zijn dan R1. Beide kunnen ofwel het aantal hertransmissies zijn ofwel een bepaalde tijdseenheid.
- Wanneer threshold R1 wordt overschreden, meldt dit aan de lagere laag.
- Wanneer threshold R2 wordt overschreden, verbreek de verbinding.
- R2 moet per connectie instelbaar zijn door de bijbehorende applicatie.
- De applicatie moet ingelicht worden over het feit dat R1 en R2 overschreden zijn.

De TCP schrijft voor welke waarden van R1 en R2 gekozen dienen te worden. R1 dient minimaal 3 hertransmissies te zijn en R2 dient minimaal 100 seconden te zijn. Een TCP-verbinding wordt dus pas na minimaal 100 seconden verbroken als segmenten niet afgeleverd konden worden of de acknowledgements ervan niet binnenkomen.

Tegenwoordig wordt voor het detecteren van segmentverlies een ander algoritme gebruikt dan oorspronkelijk in de RFC beschreven: *fast retransmit* en *fast recovery*. Dit algoritme verandert ook de werking van congestion avoidance na segmentverlies. Het algoritme gaat uit van het principe dat het ontvangen van meerdere duplicate acknowledgements een teken is van segmentverlies (namelijk het segment of de segmenten waarvoor geen acknowledgement is ontvangen). Daarnaast wil de ontvangst van acknowledgements zeggen dat er nog steeds een werkende verbinding is: er worden segmenten afgeleverd bij de tegenpartij en de acknowledgements ervan komen nog steeds binnen. Het algoritme werkt als volgt:

- Zodra er voor een segment voor de derde keer een acknowledgement binnen is gekomen, stel *sshthresh* in op de helft van het congestion window. Verstuur het missende segment opnieuw en verhoog het congestion window naar *sshthresh* plus drie maal de segmentgrootte.
- Verhoog voor elke volgende duplicate acknowledgement het congestion window met de segmentgrootte en verstuur een segment, mits toegestaan door het window.
- Wanneer een acknowledgement binnenkomt voor nieuwe data, maak het congestion window gelijk aan *sshthresh*.

Dit algoritme verhoogt de doorvoersnelheid bij segmentverlies aanzienlijk omdat het *slow start*-algoritme niet wordt uitgevoerd. Moderne TCP-implementaties gebruiken dit algoritme.

Hoofdstuk 3

Extensies

3.1 Inleiding

Het ontwerp van het TCP is erop gericht dat TCP betrouwbaar moet kunnen werken over bijna ieder willekeurig transportmedium, ongeacht de zendsnelheid, vertraging, corruptie, duplicatie of herordening van de segmenten. Met de komst van optische netwerken zijn er echter netwerken beschikbaar gekomen waarvan de zendsnelheid groter is dan waar TCP oorspronkelijk voor ontworpen is. Op deze netwerken is het dan ook mogelijk dat TCP niet de maximaal mogelijke doorvoersnelheid haalt.

Welke doorvoersnelheid wel gehaald kan worden is niet alleen afhankelijk van de zendsnelheid, maar ook van de round-trip-time. De round-trip-time is de som van de tijd die het kost om een pakket van host A naar host B te sturen, de verwerkingstijd op host B en de tijd om een acknowledgement terug naar host A te sturen. Het product van de zendsnelheid en de round-trip-time bepaalt hoeveel segmenten er ongeacknowledged verstuurd kunnen worden. Dit is van belang, aangezien het bepaalt hoeveel segmenten nodig zijn om de beschikbare bandbreedte te benutten.

3.2 Problemen met TCP

Zoals in de sectie 3.1 te lezen valt, is de doorvoersnelheid van TCP afhankelijk van het zendsnelheid maal round-trip-time product. Dit levert echter problemen op wanneer dit product zeer groot wordt, zoals gebeurt op verbindingen met hoge bandbreedte en/of grote latency. Een netwerk dat een dergelijke verbinding bevat wordt een “LFN” (Long Fat pipe Network) genoemd. Op deze netwerken treden een drietal performanceproblemen op die hieronder zullen worden behandeld.

- Window size limiet: In de TCP header wordt een 16 bits veld gebruikt om de receive window grootte door te geven aan de zender. Doordat de window size aangeeft hoeveel niet geacknowledgede octetten mogen

worden verzonden kunnen er maximaal $2^{16} = 65K$ octetten per round-trip-time worden verstuurd. Met deze window size is het niet mogelijk de maximale doorvoersnelheid van een LFN te halen.

- Herstellen na dataverlies: Wanneer er pakketverlies optreedt in een LFN heeft dit catastrofale gevolgen voor de doorvoersnelheid. In de originele specificaties werd beschreven dat segmenten opnieuw verstuurd werden na een timeout, waarna de slow start procedure werd gestart om de verbinding te herstellen. Later is er een aanpassing ontwikkeld in de vorm van de fast retransmit en fast recovery algoritmes. De combinatie van deze algoritmes maakt het mogelijk om te herstellen van het verlies van één pakket per window, zonder dat er op een timeout gewacht hoeft te worden. Wanneer er echter meerdere pakketten per window verloren gaan, zal dit resulteren in een retransmission timeout gevolgd door de slow start procedure. Met het groter worden van de window size, hetgeen nodig is om de LFN's goed te kunnen benutten, stijgt ook de kans dat er meer pakketten per window verloren zullen gaan. Dit heeft echter een vernietigende werking op de doorvoersnelheid.

- Round-trip time meten: TCP zorgt voor een betrouwbare data aflevering door segmenten opnieuw te sturen, wanneer er geen acknowledgement is gekomen binnen het retransmission timeout interval. Deze waarde moet correct en up-to-date zijn om in te kunnen spelen op veranderende eigenschappen van de verbinding. Wanneer de waarde te laag is zal dit zorgen voor onnodig veel retransmissions en wanneer de waarde te hoog is zal de doorvoersnelheid onregelmatig worden. De retransmission timeout interval wordt gebaseerd op het gemiddelde en de variantie van de gemeten round-trip time. In de standaard TCP-specificatie wordt er echter maar één meting per window gedaan, hetgeen bij grote windows, zoals die voorkomen op LFN's, een incorrecte round-trip time kan opleveren. Dit probleem wordt alleen maar groter wanneer er segmenten verloren gaan, aangezien er geen betrouwbare round-trip meting kan worden gedaan op basis van opnieuw gestuurde segmenten.

Naast deze performanceproblemen kunnen er ook betrouwbaarheidsproblemen optreden wanneer de verzendsnelheid hoog is. Dit komt doordat er dan situaties kunnen ontstaan waarbij de aannames met betrekking tot de TCP-mechanismes voor het detecteren van duplicaten en sequencing worden overtreden. Eén van de ernstigste fouten die hierbij kan optreden is het te snel hergebruiken van sequence numbers in segmenten. Dit zou er voor kunnen zorgen dat een nieuw segment eerder aankomt bij de bestemming dan een oud segment, doordat deze laatste opgehouden wordt in een buffer. Wanneer het sequence number van het nieuwe segment toevallig binnen het window valt, zal het segment worden geaccepteerd en raakt de data corrupt, zonder dat dit te detecteren is.

Daarnaast kunnen er problemen ontstaan wanneer er aan de zendende zijde een oude duplicate acknowledgement wordt ontvangen. Deze acknowledgement kan er voor zorgen dat de verbinding wordt geblokkeerd en dat er uiteindelijk

een reset moet worden gestuurd. Er zijn twee manieren waarop deze duplicate sequence nummers kunnen ontstaan:

- Het wrappen van sequence numbers binnen een bestaande verbinding: Wanneer de verzendsnelheid hoog genoeg is, is het mogelijk dat de 32 bits sequence numbers "wrappen" terwijl er nog oudere segmenten in buffers aanwezig kunnen zijn.
- Vorige instantie van een verbinding: Wanneer een verbinding wordt afgebroken, hetzij correct, hetzij door een crash en dezelfde verbinding wordt direct weer geopend, bestaat de kans dat een vertraagd segment van de vorige instantie wordt ontvangen. Wanneer het sequence number van het segment binnen het huidige window valt zal het worden geaccepteerd als correct.

In dit hoofdstuk zullen een aantal configuratie-opties en extensies worden besproken die de hierboven genoemde problemen kunnen verhelpen en/of de performance van TCP kunnen verbeteren

3.3 Send en receive space

De send- en receive space zijn configuratie-opties die bepalen hoeveel ruimte er beschikbaar is voor het bufferen van octetten op de hosts. De *send space* is de buffer aan de zendende kant waarin de verstuurde, maar nog niet geacknowledgede octetten zich bevinden. Wanneer er geen acknowledgement voor de octetten in deze buffer komt en de retransmit timer afloopt, zullen de octetten opnieuw worden verstuurd (retransmit). Wanneer er wel een acknowledgement komt, zal dat gedeelte van de octetten uit de buffer worden verwijderd en ontstaat er weer plaats voor nieuw te verzenden octetten.

Deze optie bepaalt de maximale hoeveelheid octetten die op ieder moment "in flight" kan zijn. De grootte van de send space moet derhalve minimaal even groot zijn als het zendsnelheid maal round-trip-time product, om de verbinding volledig te kunnen benutten. In het geval dat deze buffer kleiner is zal de maximale doorvoersnelheid niet gehaald kunnen worden.

De *receive space* is de buffer aan de ontvangende kant waarin de ontvangen octetten kunnen worden opgeslagen, voordat deze worden doorgegeven aan de hogere lagen. Het doel van deze buffer is het opslaan van incomplete data door het ontbreken van fragmenten. Zodra alle fragmenten zijn ontvangen wordt de data doorgegeven aan de hogere laag.

Deze optie bepaalt de maximale hoeveelheid octetten die kunnen worden ontvangen wanneer er een fragment ontbreekt. De grootte van de receive buffer moet derhalve minimaal even groot zijn als het zendsnelheid maal round-trip-time product, aangezien er anders segmenten kunnen moeten worden weggegooid op het moment dat er een fragment ontbreekt, hetgeen de doorvoersnelheid negatief zal beïnvloeden.

Wanneer de send- en/of de receive space buffers niet afdoende groot zijn, zal het niet mogelijk zijn de maximale doorvoersnelheid van de verbinding te behalen.

3.4 TCP extensions for high performance

In sectie 3.2 worden een aantal problemen met TCP op LFN's genoemd. In RFC 1323 [10] worden een drietal extensies voor TCP besproken die een aantal van deze problemen moeten verhelpen. In deze sectie zullen deze extensies achtereenvolgens aan bod komen.

3.4.1 TCP window scale option

De TCP window scale option maakt het mogelijk om grotere windows te gebruiken dan de oorspronkelijke 65K octetten. Deze extensie werkt door een extra optie toe te voegen aan het TCP-pakket waarin een schaalfactor wordt opgegeven. De waarde van deze schaalfactor wordt vervolgens gebruikt om de oorspronkelijke window grootte mee naar links te shiften. De maximale schaalfactor is 14 hetgeen een maximale window van 1 Gigabyte oplevert.

Deze optie wordt alleen verzonden in de SYN-segmenten, waardoor na het opzetten van de verbindingen de schaalfactor vast staat per richting. De uiteindelijke maximale grootte van het receive window en daarmee de schaalfactor wordt echter bepaald door de maximale receive bufferruimte.

Deze extensie op het TCP protocol maakt het mogelijk om zeer grote windows te gebruiken, waardoor er meer segmenten tergelijkertijd "in flight" kunnen zijn, hetgeen het mogelijk maakt om ook LFN's vol te krijgen.

3.4.2 Round-trip time measurement

De Round-trip time measurement extensie op het TCP protocol maakt het mogelijk dat er op een eenvoudige manier up-to-datete informatie over de round-trip time kan worden berekend. Deze extensie werkt door een extra optie toe te voegen aan het TCP-pakket met daarin een timestamp. De zendende kant zet een timestamp in het segment en de ontvangende kant stuurt deze weer terug (echoing) in de acknowledgement. Door nu de reply timestamp van de huidige tijd af te trekken is de round-trip time berekend.

TCP is echter een symmetrisch protocol, waardoor het mogelijk is dat er data verstuurd wordt in beide richtingen. Dit maakt het echter ook mogelijk dat de timestamps in beide richtingen worden gebruikt. Om de extensie eenvoudig te houden is er gespecificeerd dat het versturen en terug sturen van de timestamps in beide richtingen gebeurt. Dit wordt mogelijk gemaakt door de timestamp en de reply timestamp in één TCP timestamp option header onder te brengen. De zendende kant zet een timestamp in het timestamp value veld en de ontvangende kant stuurt deze timestamp terug in het timestamp echo reply veld, mits de acknowledgement vlag gezet is.

Deze extensie op het TCP protocol zorgt ervoor dat gebruikte round-trip times dichter bij de werkelijke round-trip time komen. Dit levert geen directe performance verbeteringen, maar zorgt er wel voor dat TCP zich beter kan aanpassen aan veranderingen, hetgeen de performance ten goede komt.

3.4.3 Protect against wrapped sequence numbers

De PAWS (Protect Against Wrapped Sequence numbers) extensie is ontwikkeld om de gevolgen van wrapped sequence numbers te voorkomen. PAWS werkt door per verbinding een oplopende waarde bij te houden. Deze waarde wordt gebaseerd op de timestamp die tevens wordt gebruikt voor de Round-Trip Time Measurement extensie. Doordat deze timestamps alleen maar groter worden, zal een segment met een lagere timestamp dan de timestamp dan de opgeslagen timestamp altijd een oud segment zijn, waarna het weggegooid moet worden.

Deze extensie is zodanig ontworpen dat het op een symmetrische manier werkt, hetgeen inhoudt dat de timestamps in zowel de data als de acknowledgement segmenten worden verstuurd. Doordat PAWS alle inkomende segmenten op dezelfde manier controleert, biedt het tevens bescherming tegen duplicate acknowledgements.

Deze extensie op TCP zorgt ervoor dat duplicate segmenten geen schade kunnen aanbrengen aan de verbinding, doordat ze kunnen worden herkend en vervolgens worden verwijderd. Dit is geen directe performance-extensie, maar zorgt er wel voor dat een aantal problemen met betrekking tot duplicate segmenten worden voorkomen. Wanneer deze problemen wél voorkomen leidt de performance hier zwaar onder.

3.5 TCP selective acknowledgement options

Wanneer er meerdere TCP-segmenten binnen een window verloren gaan, zal dit een vernietigende werking op de doorvoersnelheid hebben. Dit komt doordat TCP gebruik maakt van een cumulatieve acknowledgementmethode waarbij de ontvangen segmenten, die zich niet aan de linker kant van het receive window bevinden, niet worden geacknowledged. Dit heeft tot gevolg dat de zender een round-trip time moet wachten voor het ontdekken van iedere verloren segment of dat er onnodig segmenten herstuurd worden, ondanks dat deze wel correct zijn ontvangen.

Ter correctie van dit gedrag is de Selective Acknowledgement (SACK) extensie, beschreven in RFC 2018 [11], ontworpen. Deze extensie zorgt ervoor dat de ontvanger aan de zender kan doorgeven welke segmenten er correct zijn ontvangen, waarop de zender alleen de niet-ontvangen segmenten hoeft te hersturen.

Deze extensie maakt gebruik van twee TCP opties. De eerste optie is er één die aangeeft dat de SACK extensie gebruikt zou kunnen worden, zodra de verbinding is opgezet. Deze optie mag alleen voorkomen in SYN-segmenten. De

tweede optie is de SACK optie zelf. Deze optie mag alleen voorkomen op verbindingen die zijn opgezet en waar met behulp van de eerste optie toestemming is gegeven voor SACK.

De SACK optie wordt gebruikt op de verbinding van de ontvanger naar de zender. Deze verbinding is een simplex-verbinding. Een verbinding in de andere richting kan worden behandeld als een onafhankelijke verbinding.

Het gebruik van SACK is bedoeld om het aantal opnieuw te versturen segmenten tot het minimum te beperken, aangezien dit hersturen een negatieve invloed op de doorvoersnelheid heeft.

In RFC 2883 [6] wordt een extensie op SACK beschreven (D-SACK), die het mogelijk maakt om ook duplicate segmenten aan de zender door te geven. Deze optie maakt gebruik van de SACK optie en is compatibel met de standaard SACK. Dit zorgt ervoor dat wanneer de zendende kant de D-SACK extensie niet ondersteund het zal negeren.

3.6 Increasing TCP's initial window

In RFC 3390 [1] wordt een extensie beschreven welke de toegestane initiële window vergroot van 1 of 2 segmenten naar ongeveer 4 Kbytes. Deze vergroting heeft als doel het verhogen van de performance van verbindingen.

Er zijn echter ook een aantal nadelen aan het vergrootten van de initiële window. Ten eerste zijn er twee risico's die de kans op een implosie, op het moment dat er een congestie in het netwerk optreedt, vergroten. Het eerste risico is een scenario waarbij een groot deel van de segmenten op een verbinding met congestie bestaan uit duplicate segmenten die al ontvangen zijn door de ontvanger. Het tweede risico is dat een groot gedeelte van de segmenten op een verbinding met congestie bestaat uit segmenten die verderop in het netwerk zullen worden weggegooid, voordat de eindbestemming bereikt is. Verder bestaat de kans dat het verhogen van het initiële window het aantal weg te gooien segmenten verhoogt en daarmee een negatieve invloed heeft op ander verkeer.

In simulatie- en testomgevingen is echter duidelijk geworden dat de hierboven genoemde nadelen wel meevallen, aangezien het aantal weg te gooien segmenten maar zeer licht stijgt en er duidelijke performance winst kan worden behaald. Er treedt echter wel een duidelijke toename op in het aantal weggegooide segmenten en de hoeveelheid hertransmissies op netwerken die zeer grote congestie-problemen hebben.

Deze extensie kan met name performance verbeteringen opleveren wanneer het gaat om korte TCP sessies, zoals HTTP en SMTP sessies of om verbindingen met een grote round-trip time, zoals satellietverbindingen.

3.7 NewReno modification

In RFC 2582 [4] wordt een modificatie beschreven op het TCP fast recovery algoritme, zoals beschreven in RFC 2581 [2] en voor het eerst geïmplementeerd in de 1990 BSD Reno release. Deze modificatie heeft tot doel het verhogen van de performance van TCP wanneer er meerdere segmenten per window verloren zijn gegaan en er geen gebruik gemaakt kan worden van TCP selective acknowledgements (SACK).

Bij de Reno-versie van het TCP fast recovery algoritme wordt door de zender alleen een hertransmissie gedaan op het moment dat er een hertransmissie timeout optreedt of na het ontvangen van drie duplicate acknowledgements, waarop het fast retransmit algoritme wordt gestart. Een enkele hertransmissie timeout kan leiden tot het hersturen van verschillende datasegmenten, maar het starten van het Reno fast retransmit algoritme leidt tot het verzenden van slechts één data segment.

Dit kan echter problemen opleveren op het moment dat er meerdere segmenten van een window verloren zijn gegaan en de fast retransmit en fast recovery algoritmes zijn gestart. Wanneer dit gebeurt en er geen SACK beschikbaar is, is er weinig informatie beschikbaar voor de zender om te bepalen welke segmenten opnieuw moeten worden verstuurd. Aan de hand van de drie duplicate acknowledgements bemerkt de zender dat er een segment is verloren gegaan en zal deze het desbetreffende segment opnieuw versturen.

Op het moment dat er echter meerdere segmenten verloren zijn gegaan uit één window zal de eerstvolgende nieuwe informatie pas beschikbaar komen op het moment dat er een acknowledgement wordt ontvangen voor het herstuurde segment. Wanneer het om één verloren segment uit het window zou zijn gegaan, zou de acknowledgement voor dit segment tevens alle segmenten bevestigen die zijn verstuurd vóórdat fast retransmit mode gestart werd. Echter, wanneer er meer segmenten verloren zijn gegaan zal de acknowledgement enkele, maar niet alle segmenten bevestigen die zijn verstuurd voordat fast retransmit mode gestart werd. Een dergelijke acknowledgement wordt een *partial acknowledgement* genoemd.

In RFC 2582 wordt een aanpassing beschreven op het fast recovery algoritme dat gebruikt maakt van een extra variabele “recover” om beter te kunnen bepalen of een binnenkomende acknowledgement een partial acknowledgement is. Wanneer de acknowledgement geldt voor alle octetten tot aan “recover”, is het duidelijk dat de recovery-fase afgesloten kan worden. Zolang er nog geen acknowledgements voor alle octetten tot aan “recover” zijn ontvangen, blijft de host in recovery mode.

Deze extensie is niet de optimale verbetering voor dit probleem, maar testen hebben uitgewezen dat het een performanceverbetering oplevert in een groot aantal gevallen. Het gebruik ervan wordt aangeraden, zelfs wanneer de SACK-extensie beschikbaar is, aangezien SACK alleen mag worden gebruikt indien beide kanten van de verbinding het ondersteunen. Wanneer er echter wél SACK

toegestaan is op een verbinding heeft dit ten alle tijden de voorkeur, aangezien het beter werkt.

In RFC 3782 [5] wordt de hierboven beschreven extensie gepromoveerd van experimenteel naar standaard. Daarnaast wordt er in RFC 2582 geen aandacht besteed aan het voorkomen van meerdere fast retransmits na het optreden van een timeout, hetgeen in dit document wél wordt gedaan. Ten slotte beschrijft RFC 2582 twee versies van het algoritme, namelijk “Careful” en “Less Careful”. In RFC 3782 wordt de “Careful” versie gespecificeerd als de basisversie voor de NewReno extensie.

Hoofdstuk 4

Testmethodiek

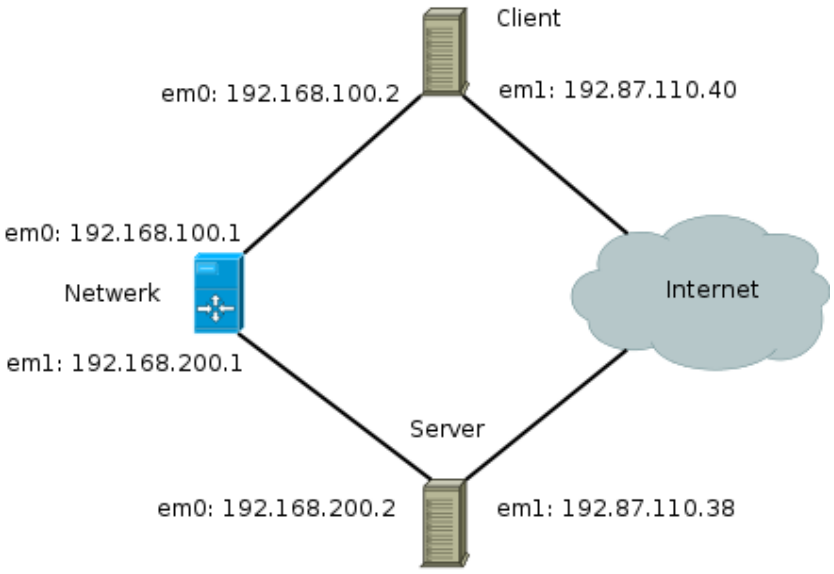
4.1 Hardware

Voor het uitvoeren van dit onderzoeksproject heeft SURFnet drie Dell PowerEdge 1850 machines beschikbaar gesteld. Twee van deze machines, de client en de server, hebben de volgende hardwarespecificaties:

- CPU: Dual 3.0GHz Xeon met Hyperthreading, 800MHz FSB en 1MB L2 cache
- Geheugen: 2 GB ECC DDR2-SDRAM (400MHz)
- RAID controller: PERC 4e/Si Ultra3320 met 256 MB cache
- Harddisk: Seagate Cheetah ST336607CC 10.000 RPM 36,7 GB SCSI
- Netwerk: twee 1Gbit Intel netwerkkaarten (Intel 82541), 64 bits PCI-X

De derde machine, die als netwerksimulator en router diende (netwerk), heeft de volgende hardwarespecificaties:

- CPU: Dual 2.8GHz Xeon met Hyperthreading, 800MHz FSB en 1MB L2 cache
- Geheugen: 1 GB ECC DDR2-SDRAM (400MHz)
- RAID controller: PERC 4e/Si Ultra3320 met 256 MB cache
- Harddisk: Seagate Cheetah ST336607CC 10.000 RPM 36,7 GB SCSI
- Netwerk: twee 1Gbit Intel netwerkkaarten (Intel 82541), 64 bits PCI-X



Figuur 4.1: De testopstelling

4.2 Opstelling

Voor het uitvoeren van de TCP performance tests werden er met behulp van Iperf pakketten van de client via de netwerksimulator naar de server verstuurd. De netwerksimulator werd gebruikt om latency en pakketverlies te simuleren. Een schema van de opstelling is te vinden in figuur 4.1.

Voor de netwerksimulatie is er in eerste instantie gekozen voor Dummynet, een stuk software waarmee de FreeBSD firewall "ipfw" de eigenschappen van netwerken kan simuleren, zoals latency en pakketverlies. In hoofdstuk 4.4.2 wordt Dummynet nader bekeken. Helaas leek het erop dat Dummynet niet erg geschikt is om met veel toegevoegde latency nog hoge doorvoersnelheden te testen. De doorvoersnelheid bij voorlopige tests bleek ver beneden de mogelijke snelheid te liggen, ondanks dat de client en server nog processortijd en netwerkbuffers beschikbaar hadden. Omdat de tests niet beïnvloed zouden mogen worden door de simulatiesoftware, is de overstap gemaakt naar Linux met de ingebouwde *Netem* simulatiesoftware. In hoofdstuk 4.4.3 wordt dit product besproken. De eerste keer dat er met Netem een latency van 100 ms werd gesimuleerd op de verbinding tussen de client en de server, crashte daarbij het systeem "netwerk" zonder kernel-panic of enige andere vorm van notificatie van problemen. We hebben daarom besloten niet nog meer tijd te steken in Linux, omdat we daar beiden minder ervaring mee hadden dan met FreeBSD en omdat de resultaten van Dummynet op FreeBSD ons meer vertrouwen gaven in het gebruik ervan ten opzichte van Netem op Linux.

4.3 Besturingssystemen

Er zouden in eerste instantie twee besturingssystemen getest worden, namelijk FreeBSD en Windows. De keuze voor FreeBSD is gemaakt om de volgende redenen.

- FreeBSD wordt door SURFnet gebruikt voor het aanbieden van allerlei netwerkdiensten, waaronder de dienst "Kanon".
- FreeBSD is jarenlang het besturingssysteem geweest waar *reference implementations* van nieuwe standaarden voor geschreven werden. De TCP/IP-stack van FreeBSD wordt nog steeds geroemd om de robuustheid en performance. Het testen van deze stack zou dus betrouwbare resultaten op moeten kunnen leveren.
- De auteurs hebben beiden veel ervaring met FreeBSD, waardoor het tunen van het besturingssysteem om zo de tests zo betrouwbaar mogelijk te laten verlopen goed mogelijk zou zijn.

In hoofdstuk 4.3.1 wordt verder ingegaan op de specifieke configuratie van de FreeBSD-systemen.

Om de volgende redenen is er voor Windows als testplatform gekozen.

- Veel van de bij SURFnet aangesloten instellingen maken zowel voor werkstations als ook voor servers veel gebruik maken van Windows.
- Het aanpassen van de TCP/IP-instellingen op Windows wordt door beheerders niet veel gedaan, hoewel het voor bepaalde situaties zeker nuttig kan zijn. De tests die zijn gedaan zouden voor beheerders van Windows servers nuttige informatie kunnen verschaffen met betrekking tot het *tunen* van de TCP/IP-stack.

Door het zoeken naar de oorzaak van de problemen met het simuleren van latency is er geen tijd overgebleven voor het uitvoeren van de tests op Windows.

4.3.1 FreeBSD

Voor de FreeBSD-machines is er gekozen voor FreeBSD 5.4-RELEASE. Dit is de huidige productie-release. Hoewel FreeBSD 4 nog steeds erg veel gebruikt wordt, is het aangemerkt als de *legacy* release. Aan FreeBSD 4 wordt niet meer ontwikkeld en zal op den duur dus gaan verdwijnen. Om het project relevant te laten zijn voor de toekomst, leek FreeBSD 5.4 daarom een betere keus.

De client en server waren in principe hetzelfde geconfigureerd. Er is een vrij standaard installatie gedaan en daarna is er nog enige configuratie uitgevoerd. De relevante aanpassingen die zijn gedaan zijn als volgt:

- Een SMP-kernel gebouwd. Standaard wordt gebruik gemaakt van een kernel die geen meerdere processoren ondersteund. Een SMP-kernel doet dat wel. Door gebruik te maken van meerdere processoren kan ervoor gezorgd worden dat de test minder beïnvloed wordt door processen die draaien naast Iperf. Ook kunnen een aantal kernel-taken parallel worden uitgevoerd, wat de performance ten goede komt.

- Polling uitgeschakeld. Normaal gesproken handelt de kernel het ontvangen van ethernet frames af elke keer wanneer er een interrupt wordt gegenereerd door de netwerkkaart. Door gebruik te maken van polling handelt de kernel inkomende frames af door regelmatig te controleren of er frames binnengekomen zijn. Hierdoor kan bij een zware netwerkbelasting voorkomen worden dat het systeem vrijwel alleen bezig is met het afhandelen van interrupts waardoor de processen geen processortijd meer krijgen (het principe van *live lock*). Hoewel we polling-support hadden ingebouwd in de kernel, is er op de client en de server uiteindelijk geen gebruik van gemaakt. Omdat op zowel de client als server een Iperf-proces draait die een bepaalde processorbelasting zou veroorzaken en de configuratie van polling de beheerder de keus geeft interrupts af te handelen ten koste van processen of andersom, zou daardoor het vinden van de best werkende configuratie erg veel tijd kosten. In onderstaande paragraaf, waarin de configuratie van het simulatiesysteem wordt besproken, wordt iets dieper in gegaan op polling. Er is vanwege de complexiteit van polling gekozen dit niet aan te zetten op de client en de server. De belasting van zowel de client als server leek geen bottleneck te zijn tijdens de tests waarbij de hoogste doorvoersnelheden werden gehaald, zodat het inschakelen van polling de performance waarschijnlijk niet substantieel had kunnen verhogen.

- De geheugenbuffers voor netwerkpakketten verhoogd in "/boot/loader.conf" (A.1, A.2). Hierdoor wordt voorkomen dat het systeem te weinig netwerkbuffers overhoudt voor het versturen en ontvangen van data, wat een grote terugval van performance zou veroorzaken.

- Door middel van de file "/boot/loader.conf" de waarde van de kernelvariabele "HZ" aangepast. Een aantal kernelactiviteiten vinden plaats bij elke kloktik gedefinieerd door de waarde van "HZ". Standaard vinden die activiteiten 100 keer per seconde plaats. Wanneer gebruik wordt gemaakt van polling is het verstandig deze waarde te verhogen, zodat de granulariteit van het *pollen* verhoogd wordt. Doordat polling is getest, is deze waarde verhoogd, maar omdat uiteindelijk geen gebruik is gemaakt van polling is dit waarschijnlijk overbodig geworden.

- Onnodige daemons uitgeschakeld vanuit rc.conf (A.7, A.5). Deze daemons waren "usbd", de USB-daemon, "cron", waarmee op specifieke tijden jobs gestart kunnen worden die de tests kunnen beïnvloeden en "sendmail", de MTA.

- De IP input queue lengte verhoogd, door de waarde van "intr_queue_maxlen" te verhogen in "/etc/sysctl.conf" (A.9). De standaardgrootte van deze queue is 50, wat te weinig bleek voor sommige tests. In eerste instantie is deze waarde verhoogd naar 500, maar dit bleek op de client nog niet hoog genoeg te zijn. Daarom is die later nog verder verhoogd.

De machine "netwerk" had op sommige punten een andere configuratie. De verschillen worden hieronder uitgelegd.

- Er werd gebruik gemaakt van een GENERIC kernel, zonder multi-processor support. Omdat in het voortraject van de tests duidelijk werd dat er enige performanceproblemen ontstonden bij het introduceren van latency, zijn allerlei configuraties van het simulatiesysteem onderzocht. Door gebruik te maken van een kernel zonder multi-processor support kan eenvoudiger gezien worden of de belasting van het systeem een bottleneck is of niet. Bij multi-processor systemen zou de belasting laag kunnen zijn terwijl het systeem het aangeboden werk eigenlijk niet aan kan. Dit kan voorkomen wanneer de processoren op elkaar wachten vanwege slecht te parallelliseren taken. Om het testen van het simulatiesysteem doorzichtig en overzichtelijk te houden is er gekozen om een uniprocessor kernel te maken.

- Op het simulatiesysteem is er wél gebruik gemaakt van polling. Omdat dit systeem slechts pakketten hoefde te routeren tussen twee interfaces en eventueel latency of packet loss hoefde te simuleren, was het mogelijk de kernel veel tijd te laten besteden aan het afhandelen van interrupts. Het voordeel van het gebruik van polling op dit systeem was dat daardoor er nog processortijd overbleef voor processen, zodat het mogelijk was om instellingen te wijzigen en de effecten daarvan te controleren, te controleren of bepaalde buffers niet volstroomden, enz.

- De optie *idle_poll* aangezet (A.8). Hiermee wordt de kernel geïnstrueerd te pollen voor inkomende ethernet frames als het systeem *idle* is, dat wil zeggen: wanneer de scheduler geen processen meer kan laten draaien. Hierdoor werd de performance verder verhoogd. Een andere manier om de kernel vaker te laten pollen is het gebruik van de *user_frac*-optie. Hiermee kan aangegeven worden wat de verhouding tussen het CPU-gebruik voor polling en voor processen moet zijn. Aangezien deze afweging moeilijk te maken en te testen is, hebben we ervoor gekozen deze waarde op de standaard waarde "50" te laten staan.

- Fastforwarding aangezet in "/etc/sysctl.conf" (A.8). Als deze optie aangezet wordt, kunnen inkomende pakketten zoveel mogelijk zonder gebruikmaking van de CPU weer uitgestuurd worden. De pakketten hoeven niet in een queue gezet of anderszins gekopieerd te worden, waardoor de performance sterk verbetert.

4.4 Software

In dit hoofdstuk wordt de gebruikte software anders dan de besturingssystemen besproken. Daarnaast zullen de gebruikte scripts worden beschreven. Met de informatie in dit hoofdstuk en de gebruikte software, moet het voor de lezer van dit document mogelijk zijn de uitgevoerde tests te herhalen.

4.4.1 Iperf

Voor het testen van de TCP bandbreedte is er gebruik gemaakt van de netwerk performance benchmarkapplicatie Iperf [19]. Er is gekozen om gebruik te maken van versie 2.0.1 en niet de laatste versie 2.0.2, aangezien deze eerste standaard beschikbaar is voor FreeBSD en er tevens via SURFnet een binary van deze versie voor Windows beschikbaar is.

Tijdens het testen zijn de volgende commando's op de client en server uitgevoerd:

```
client# iperf -c server -N -i 5 -t 60 -l4096
server# iperf -s -N
```

In de hierboven genoemde commando's wordt de server de ontvangende kant (-s) en de client de zendende kant. Verder wordt er de TCP NODELAY optie gezet (-N), wordt het rapportage-interval op 5 seconden gezet (-i 5), wordt de testduur op 60 seconden gezet (-t 60) en wordt de write-size op 4096 gezet (-l 4096). Tests wezen uit dat het gebruik van 4096 bytes de hoogste doorvoersnelheid opleverde bij de toen gekozen window-grootte van 256KB en socket buffergrootte van 512KB. Een mogelijke verklaring hiervoor is dat 4096 precies 128 keer in 512KB en dus 64 keer in 256KB past (waardoor het efficiënt in de buffers geplaatst kan worden), bepaalde buffers meervouden van 4096 bytes kunnen zijn en dat de page-size van het platform 4096 bytes is, waardoor eventuele kostbare kopieeracties vanwege *alignment* niet nodig zijn. Dit zou nog verder onderzocht kunnen worden.

4.4.2 Dummynet

Voor het simuleren van latency en pakketverlies is er gebruik gemaakt van Dummynet [17]. Deze applicatie maakt het mogelijk om latency, pakketverlies en bandbreedte te simuleren. In Dummynet worden verbindingen (pipes) gedefinieerd waaraan de gewenste eigenschappen kunnen worden toegekend, bijvoorbeeld:

```
ipfw pipe 1 config delay 10ms plr 0.01
ipfw pipe 2 config bw 50Kbit/s queue 10
```

De eerste regel maakt een verbinding met een latency van 10 ms en een pakketverlies van 1 procent. De tweede regel maakt een verbinding met een maximale bandbreedte van 50 Kbits per seconde.

Vervolgens kan er in de FreeBSD firewall ipfw verkeer aan een bepaalde verbinding worden toegekend, bijvoorbeeld:

```
ipfw add 10 pipe 1 ip from 10.0.0.0/24 to any
ipfw add 20 pipe 2 ip from any to 10.1.2.0/24
```

De eerste regel zorgt ervoor dat verkeer van het 10.0.0.0/24 netwerk naar ieder ander netwerk aan de eerste verbinding (pipe 1) wordt toegekend. Het zal derhalve minimaal een latency van 10 ms en een pakketverlies van 1 procent krijgen. De tweede regel zorgt ervoor dat al het verkeer naar het netwerk 10.1.2.0/24 een maximale bandbreedte van 50 Kbits per seconde zal hebben.

Tijdens dit project is er gebruik gemaakt van de Dummynet-versie die standaard in FreeBSD 5.4-RELEASE aanwezig is. In bijlage A.4 wordt de gebruikte ipfw configuratie vermeld. Voor het configureren van de latencywaardes en pakketverlieswaardes is gebruikt gemaakt van een script (B.1) op de netwerk machine.

4.4.3 Netem

Voor het simuleren van latency en pakketverlies is er tevens geprobeerd gebruik te maken van Netem [8]. Netem is een optie in de Linux kernel die het mogelijk maakt om variabele delay, pakketverlies, pakketduplicatie en reordering te simuleren. Netem is gebaseerd op NISTNet [13], maar biedt minder functionaliteit, aangezien deze functionaliteit al via andere mechanismes in de Linux kernel aanwezig zijn.

De Netem configuratie wordt gedaan met behulp van het commando **tc** dat onderdeel is van het iproute2 pakket. In het voorbeeld hieronder wordt een latency van 100 ms op de eerste netwerkinterface gedefiniëerd.

```
netwerk# tc qdisc add dev eth0 root netem delay 100ms
```

De versie die tijdens dit project is geprobeerd, is de versie die in de standaard Linux 2.6.11.9 kernel aanwezig is. Wegens de al eerder gemelde problemen met deze software is het verder niet gebruikt voor het project.

4.4.4 Testscript test-fbsd.pl

Alle tests zijn uitgevoerd met behulp van het script *test-fbsd.pl*. Dit script is gebouwd om op een snelle en eenvoudige manier honderden tests achter elkaar te laten draaien. Het script zorgt ervoor dat alle tests op een gestructureerde manier gelogd worden naar bestanden. De TCP-instellingen kunnen volledig automatisch door het script aangepast worden, zodat de tests betrouwbaar en snel uitgevoerd kunnen worden.

Een belangrijk component van het script is de definitie van tests, de aan te passen instellingen en de waarden van die instellingen. Door middel van simpele Perl-constructies zoals arrays en hashtables kunnen er meerdere tests worden

gedefiniëerd. Zo is het mogelijk een test te definiëren waarbij de waarden van de TCP-buffers worden afgewisseld, de latency op het simulatiesysteem wordt afgewisseld, SACK aan en uit wordt gezet, enz. Elke test bevat een lijstje met welke opties er gezamenlijk getest worden. Voor alle andere opties worden de standaard instellingen gebruikt die ook gedefiniëerd kunnen worden.

Het uitvoeren van een test houdt in dat het script allereerst alle opties op de standaard waarden zet. Daarna worden voor alle opties waarop getest wordt alle mogelijke waarden geconfigureerd op de betrokken systemen. De instellingen op het simulatiesysteem worden aangepast door het client-systeem. Er kunnen via een TCP-verbinding commando's worden doorgegeven aan het simulatiesysteem, die dan daar op worden uitgevoerd. Zo kan het client-systeem dus de instellingen van Dummynet op het simulatiesysteem aanpassen. Omdat de Iperf-server niet afsluit nadat een client klaar is met testen, wordt op dezelfde wijze door de client op de server Iperf afgesloten. Op zowel het simulatiesysteem als op de server worden hiervoor shellscripts gebruikt, zodat wijzigingen in de wijze waarop de client instellingen kan wijzigen of Iperf af kan sluiten makkelijk doorgevoerd kunnen worden. Deze scripts worden aangeroepen via een shell die gestart wordt vanuit inetd.

De tests zelf houden in dat Iperf vanaf de client naar de server een verbinding maakt en voor een bepaalde periode zoveel mogelijk gegevens stuurt naar de server. Iperf geeft op *stdout* de snelheid van de afgelopen aantal seconden weer. Het interval hiervan is instelbaar, evenals de totale duur van de test. Wij hebben gekozen voor een interval van 5 seconden en een testduur van 60 seconden. Door een interval van 5 seconden te gebruiken is het mogelijk goed te zien of er ook een opbouw of juist neergang in de doorvoersnelheid is. Wanneer een grotere interval gebruikt zou worden, zou dat niet zo duidelijk zijn. Een kleinere interval biedt slechts meer te verwerken gegevens zonder veel waarde toe te voegen. Het Perl-script zorgt er voor dat de uitvoer van Iperf per test in een apart logbestand wordt opgeslagen. Elke keer dat Iperf gedraaid wordt, wordt de output ervan dus in een specifiek bestand opgeslagen. De bestandsnaam bevat de naam van de test (die bestaat uit de opties die getest worden), de huidige waarden van die opties en een datum- en tijdsaanduiding. Zo kan aan de bestandsnaam direct gezien worden van welke test het logbestand de uitvoer bevat.

4.4.5 Rapportagescript averages.pl

Het eerste script dat gedraaid kan worden na het uitvoeren van een testrun is *averages.pl*. Dit script rekent voor een testrun per test uit wat de gemiddelde doorvoersnelheid voor die test is geweest. De testnamen en gemiddelde doorvoersnelheden worden in een bestand "averages.log" in de hoofddirectory van de testrun opgeslagen. Dit bestand kan daarna verder verwerkt worden om bijvoorbeeld grafieken te maken van de doorvoorsnelheden of de gemiddelden van meerdere testruns te vergelijken.

In bijlage C.1 is de broncode van dit script te vinden.

4.4.6 Rapportagescript merge-avg.pl

Om de betrouwbaarheid van testresultaten te verhogen, is het nuttig tests meermalen uit te voeren. Het testscript *test-fbsd.pl* slaat de resultaten van elke afzondere testrun op in aparte directories. Het script *averages.pl* maakt een bestand *averages.log* aan per testrun met daarin de gemiddelden voor elke test. Om nu de doorvoersnelheden van verschillende testruns samen te voegen tot één gemiddelde, kan het script *merge-avg.pl* gebruikt worden. Bijvoorbeeld als volgt:

```
client# perl merge-avg.pl 1 2 3
```

Het script zal in dit geval de gemiddelden van runs “1”, “2” en “3” samenvoegen tot één gemiddelde per uitgevoerde test en de resultaten hiervan op *stdout* weergeven. Deze uitvoer bevat de naam van elke test, zonder de timestamp, plus de gemiddelde doorvoersnelheid voor die test over alle testruns.

De broncode van dit script is te vinden in bijlage C.2.

Hoofdstuk 5

Testresultaten

In dit hoofdstuk worden de resultaten van de volgende tests besproken:

- Doorvoersnelheid bij wisselende buffergroottes en wisselende latencies met RFC 1323-extensies uitgeschakeld.
- Doorvoersnelheid bij wisselende buffergroottes en wisselende latencies met RFC 1323-extensies ingeschakeld.
- Doorvoersnelheid bij wisselend pakketverlies standaard TCP.
- Doorvoersnelheid bij wisselend pakketverlies met NewReno-extensie ingeschakeld.
- Doorvoersnelheid bij wisselend pakketverlies met SACK-extensie ingeschakeld.

Per test worden de resultaten besproken aan de hand van een grafiek. Iedere test is 2 maal uitgevoerd met een duur van 60 seconden per test. Het testscript is zo geschreven, dat voor de opties die niet getest worden, standaard waarden worden geconfigureerd. Deze standaard waarden zijn in het script in te stellen. Op deze manier zijn tijdens de door ons uitgevoerde test de niet-relevante opties uitgeschakeld, om zo de betrouwbaarheid van de tests te verhogen.

De getoonde waarden in de grafieken zijn het gemiddelde over de twee uitgevoerde tests. Tijdens het uitvoeren van de tests bleken de resultaten niet overeen te komen met de verwachtingen. In de volgende sectie wordt ingegaan op de afwijkende testresultaten, parameters die aangepast zijn om de resultaten betrouwbaarder te maken en eventuele oorzaken van de afwijkingen.

5.1 Afwijkingen

5.1.1 Inleiding

Tijdens de testweek bleek al snel dat de tests met hogere latency en grotere buffers niet de gewenste en mogelijke doorvoersnelheid opleverden. Het bleek

dat bij hogere latencies en de daarbij verhoogde buffergroottes de doorvoersnelheid erg achterbleef: bij een latency van 100 ms en een buffergrootte van 16MB bleef de doorvoersnelheid steken op ongeveer 70Mbits/s. Een buffergrootte van 16MB is echter groot genoeg om bij een latency van 100 ms de volledige bandbreedte van de verbinding te kunnen benutten. Er is dus duidelijk sprake van een bottleneck ergens in de testopstelling.

Omdat we zeker wilden zijn van de betrouwbaarheid van de testopstelling, is onderzocht waardoor deze afname van doorvoersnelheid veroorzaakt werd. Hiertoe zijn een aantal instellingen gewijzigd om te bepalen welke instellingen een positief danwel negatief effect zouden hebben op de doorvoersnelheid.

5.1.2 Uitgevoerde tests en aangepaste parameters

Om te achterhalen waar de bottleneck in de testomgeving aanwezig was, zijn er een aantal parameters aangepast tijdens het testen. Daardoor kon snel worden gezien welke parameters een positieve invloed op de doorvoersnelheid hadden en welke de doorvoersnelheid juist negatief beïnvloedden. De volgende parameters zijn aangepast:

Polling Zoals beschreven in hoofdstuk 4.3.1, kunnen met polling inkomende ethernet frames efficiënter afgehandeld worden, door interrupts te voorkomen hetgeen *live lock* voorkomt. Op de client en server is polling uitgeschakeld, maar op het simulatiesysteem is het ingeschakeld. Door polling in te schakelen op het simulatiesysteem werd de doorvoersnelheid verhoogd en werd live lock voorkomen zodat het mogelijk bleef aanpassingen op het systeem door te voeren tijdens tests. Door de optie *idle_poll* aan te zetten werd de performance nog verder verhoogd, doordat de kernel door middel van die instelling geïnstrueerd wordt ook te pollen wanneer de draaiende processen geen processortijd nodig hebben.

sysctl net.inet.ip.intr_queue_maxlen Deze sysctl instelling bepaalt de maximale lengte van de IP input queue. De standaard maximum lengte van deze queue is 50. Op de client bleek dat deze queue erg vaak te klein was om inkomende IP-pakketten in op te slaan. Omdat we dit als mogelijke oorzaak zagen van een verminderde performance, is de maximum lengte verhoogd naar 500. Op de client bleek dit nog steeds niet voldoende te zijn tijdens verdere tests. Daarom is dit toen nogmaals verhoogd, naar 5000. Toen bleek dat daardoor de doorvoersnelheid juist *negatief* beïnvloed werd, zijn er extra experimenten uitgevoerd om te achterhalen welke invloed deze setting op de doorvoersnelheid zou hebben.

Door de input queue te verkleinen naar minder dan 50 werd de doorvoersnelheid verhoogd, ondanks dat er pakketten verloren gingen vanwege de te kleine queue. Hieruit kan worden afgeleid dat het algoritme dat gebruikt wordt om pakketten te plaatsen in de queue of uit te lezen uit de queue minder efficiënt wordt naarmate de queue groter wordt. De reden

dat het pakketverlies geen negatieve gevolgen had voor de doorvoersnelheid ligt in welke pakketten verloren gingen. Omdat het hier de *input* queue van de *zender* betreft, gaat het om acknowledgements die de ontvanger aan de zender stuurt. Aangezien er geen pakketverlies optrad in de richting van de ontvanger, kwamen de verstuurde gegevens nog steeds aan. De acknowledgements van TCP zijn cumulatief, zodat wanneer er een acknowledgement verloren gaat, de daarop volgende acknowledgement geldt voor alle tot dan toe ontvangen gegevens. Het verlies van de acknowledgement heeft daardoor geen gevolgen voor de doorvoersnelheid.

Door op de ontvanger de optie "net.inet.tcp.delayed_ack" aan te zetten, worden er door de ontvanger minder acknowledgements gestuurd, waardoor de input queue minder snel te klein zal zijn voor het afhandelen van de inkomende pakketten. Tijdens de tests stond deze optie echter uit, waardoor de input queue te klein bleek.

Dummynet queue-groottes Wanneer Dummynet gebruikt wordt om latency te introduceren op een verbinding, moet Dummynet pakketten tijdelijk opslaan in een queue. De grootte van deze queue is in te stellen op twee manieren: in Kbytes en in slots. Eén slot kan één pakket bevatten. Dummynet heeft standaard een maximum queue grootte van 1000 Kbytes of 100 slots. Omdat bij hogere latencies veel gegevens in deze queue opgeslagen zouden moeten worden (net zoals dat gebeurt in de send en receive buffers van TCP) en omdat 1000 Kbytes niet groot genoeg zou zijn voor bepaalde door ons getestte bandbreedte maal round-trip-time producten, leek het ons logisch deze queues groter te maken dan de standaard maximale grootte.

Hiertoe hebben we aanpassingen gedaan in de broncode van de tool "ipfw" en in de broncode van Dummynet zelf. Dit leverde echter niet het gewenste resultaat. Het blijkt dat de grootte van de Dummynet queue weinig invloed heeft op de doorvoersnelheid. Zelfs met een latency van 100 ms was er geen verschil in doorvoersnelheid te bemerken tussen een test waarbij een queue van 100 Kbytes werd gebruikt en een test waarbij de queue 4000 Kbytes groot werd gemaakt. Wél blijkt dat kleine queue groottes als 40 Kbytes de doorvoersnelheid zeer negatief beïnvloeden.

Netwerk buffers FreeBSD maakt voor de tijdelijk opslag van gegevens die ontvangen zijn of verstuurd worden gebruik van zogenaamde *mbufs*. Het aantal beschikbare mbufs bepaalt hoeveel gegevens in de kernel kunnen worden opgeslagen. Wanneer er geen mbufs meer beschikbaar zijn bij het verwerken van een inkomend of uitgaand pakket, zal dit verloren gaan. Het is dus zaak om voldoende mbufs te hebben voor de verwachte netwerkbelasting. Het aantal mbufs op de testsystemen is verhoogd tot 65536, dat ruim voldoende is voor de uitgevoerde tests.

Interface errors Om uit te kunnen sluiten dat de netwerkkaarten op het simulatiesysteem problemen veroorzaakten, is er gekeken naar de errors die de netwerkkaart aan het besturingssysteem heeft gemeld. Door mid-

del van het commando `netstat -li` kunnen statistieken per interface verkregen worden over de ontvangen en verstuurde pakketten, eventuele errors en collissions. Op de drie testsystemen was het aantal errors ten opzichte van het aantal ontvangen en verstuurde pakketten verwaarloosbaar. Interface errors leken derhalve niet de oorzaak van de tegenvallende doorvoersnelheid te zijn.

Netwerkkaart debugging Om extra zekerheid te krijgen over het aantal errors op de interfaces, is op het simulatiesysteem de driver van de netwerkkaarten opgedragen debugging informatie weer te geven. Wanneer debugging aan staat in de driver, worden er regelmatig statistieken gegenereerd en weergegeven in de kernel messages buffer (dmesg). In de debugging uitvoer kwamen geen errors naar voren. De netstat-uitvoer en de driver debugging uitvoer bleken derhalve in overeenstemming met elkaar te zijn.

TCP/IP statistieken Aangezien er geen errors ontstonden in de hardware of in de drivers van de netwerkkaarten, is er gekeken naar de statistieken van de TCP/IP-stack zelf. Deze statistieken kunnen op het FreeBSD-platform opgevraagd worden met het commando `netstat -s`. De statistieken op de drie testsystemen gaven geen indicaties van problemen op het TCP/IP-niveau.

Uitgebreidere tests Op aanraden van de schrijver van Dummynet, Luigi Rizzo, zijn een groot aantal extra tests uitgevoerd. Hierbij is de latency op het simulatiesysteem in kleine stapjes verhoogd. Ook de TCP buffers zijn in kleinere stappen getest, zodat er een duidelijker beeld zou ontstaan over wanneer de doorvoersnelheid niet overeenkwam met de gewenste en mogelijke doorvoersnelheid. Uit deze tests bleek dat naar mate de latency groter wordt en er dus ook grotere buffers gebruikt worden, de doorvoersnelheid steeds verder achterblijft bij de mogelijke doorvoersnelheid. Er is echter uit de resultaten niet te concluderen of de bottleneck ligt bij Dummynet of bij de zender en ontvanger.

Deze tests zijn uitgevoerd met verschillende waarden van "HZ" (zie hoofdstuk 4.3.1). Op de drie testsystemen was de initiële waarde van HZ 2000. Omdat Dummynet pakketten slechts per tick, dat wil zeggen, in dit geval 2000 keer per seconde, kan verzenden, zou het aantal pakketten dat in één keer verstuurd zou worden erg groot kunnen worden. Daardoor zou ofwel het simulatiesysteem de pakketten bij het uitsturen kunnen verliezen, of zou de ontvanger ze niet kunnen verwerken. Door op het simulatiesysteem de waarde van HZ te verhogen naar 20000 is er getest of dit wellicht de oorzaak kon zijn van de lage doorvoersnelheid. Er bleek echter geen verschil te bestaan in doorvoersnelheid bij deze twee waarden van HZ.

5.1.3 Mogelijke oorzaken

De meest voor de hand liggende bottleneck ligt bij Dummynet zelf. Uit de tests blijkt dat wanneer er extra latency wordt gesimuleerd door Dummynet,

de doorvoersnelheid drastisch daalt, ongeacht hoe groot de gebruikte send en receive buffers op de client en server zijn.

Daarnaast is het mogelijk dat de bottleneck bij de client of server zelf ligt. Uit de gedane tests is vooralsnog niet duidelijk waar de bottleneck precies ligt. Tijdens een test met 50 ms latency en 16MB grote send en receive buffers kon het volgende vastgesteld worden:

- De doorvoersnelheid van een enkele verbinding ligt lager dan de totale doorvoersnelheid van meerdere gelijktijdige verbindingen. Hieruit kan geconcludeerd worden dat de enkele TCP-verbinding niet de hoogst mogelijke doorvoersnelheid haalt. Dit kan liggen aan Dummynet, maar kan ook voortkomen uit de verwerkingssnelheid op het systeem zelf.

- Het clientsysteem had tijdens de test nog 40% processortijd over. Hieruit zou geconcludeerd kunnen worden dat de bottleneck niet ligt in de verwerking op de client. Het clientsysteem is echter een multi-processorsysteem, waardoor niet altijd beschikbare processortijd daadwerkelijk ingezet kan worden door het besturingssysteem. Daarom kunnen uit het beschikbaar zijn van CPU-tijd niet zonder verder onderzoek conclusies worden getrokken.

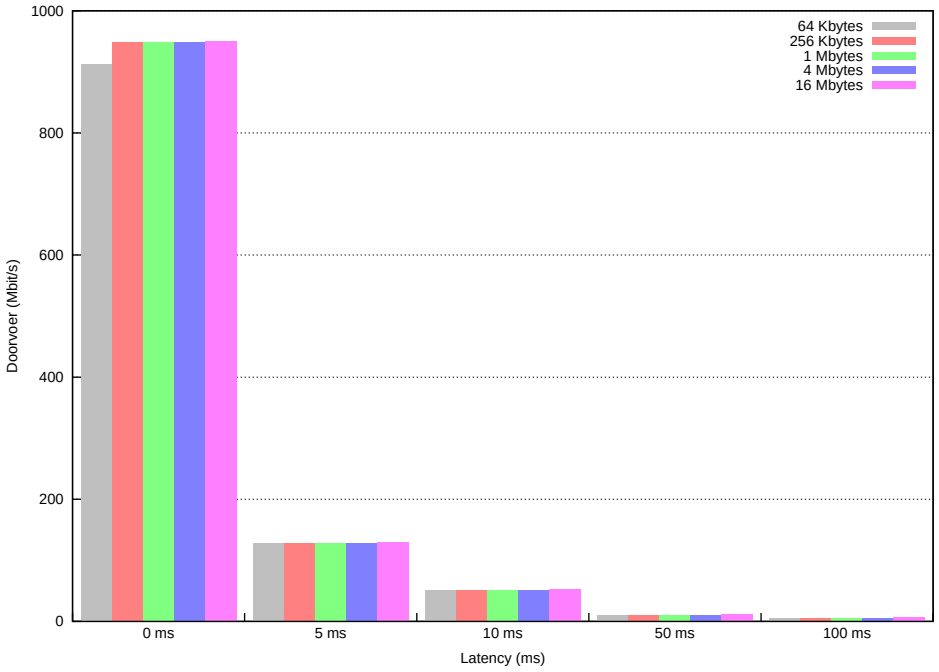
- De send buffer is tijdens de test voor elke TCP-verbinding vrijwel volledig gevuld. Afhankelijk van het in FreeBSD geïmplementeerde algoritme om segmenten uit een buffer te versturen, kan het zijn dat grote buffers het versturen van segmenten vertragen ten opzichte van kleine buffers. Hierdoor zou de doorvoersnelheid achter kunnen blijven.

- Wanneer de doorvoersnelheid enkele seconden op het maximum is aanbeland, loopt de latency toe van al het verkeer vanaf of naar de client, ook als dit verkeer over een andere interface uitgestuurd wordt dan waarover de test gedaan wordt. De latency liep op tot ongeveer 25 tot 30 ms RTT tussen het clientsysteem en “ping.xs4all.nl”. De standaard RTT tussen deze twee systemen is ongeveer 1,5 ms. Deze toename van latency wijst erop dat er een bepaalde bottleneck ligt in het client-systeem. Dit zou in een vervolgproject wellicht onderzocht kunnen worden.

Er lijkt dus een bottleneck te liggen bij de client. Vreemd is wel dat de performance zo ver achterblijft bij de theoretisch mogelijke performance. Bij Internet2 land speed record[20] pogingen zijn snelheden gehaald tot aan 7Gbits/s op redelijk vergelijkbare apparatuur met veel hogere latencies en veel grotere buffers. De resultaten van de door ons uitgevoerde tests zijn enkele ordes van grootte slechter, waarvoor vooralsnog geen duidelijke verklaring is. Er zou dus verder onderzocht kunnen worden waar de bottleneck nu precies ligt en of het probleem op te lossen is door bijvoorbeeld queueing-algoritmes te verbeteren of door wellicht de testopstelling te wijzigen.

5.2 Buffergrootte en latency

In deze tests wordt weergegeven wat de maximale gehaalde doorvoersnelheid is bij verschillende buffergroottes en latencies. Tijdens deze test stond de RFC 1323 extensie uit, zodat de maximale window size slechts 64 Kbytes zou zijn. De resultaten van deze tests zijn weergegeven in figuur 5.1.



Figuur 5.1: Buffergrootte en latency

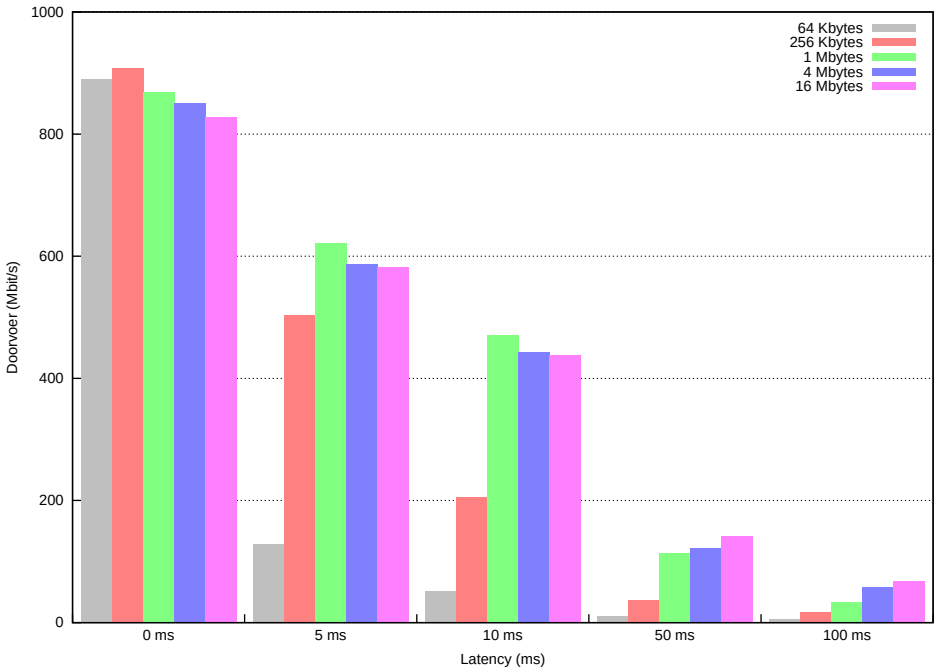
In deze figuur valt op dat de maximale doorvoersnelheid alleen redelijk wordt benaderd in het geval van 0 ms latency. Bij alle andere latencies wordt deze snelheid bij lange na niet gehaald. Dat deze snelheid niet wordt gehaald is echter correct, aangezien de maximale window grootte 64 Kbytes is ongeacht de send- en receivespace. Wanneer de window grootte 64 Kbytes is en de latency 5 ms bedraagt, is de maximale haalbare doorvoersnelheid:

$$\frac{1000 \text{ ms}}{5 \text{ ms}} \times 64 \text{ Kbytes} = 12800 \text{ Kbytes/s} = 100 \text{ Mbits/s}$$

Op het moment dat de latency nog verder omhoog gaat zal de maximaal haalbare doorvoersnelheid nog verder dalen, hetgeen ook goed te zien is in de figuur. Uit deze grafiek blijkt dus dat bij een hoog bandbreedte maal round-trip-time product een grotere window size nodig is dan de standaard 64 Kbytes.

5.3 Buffergrootte en latency met RFC 1323 ingeschakeld

In deze tests wordt er met dezelfde buffergroottes en latencies getest als in de sectie hierboven, alleen staat nu de RFC 1323 extensie wel aan. Deze extensie bevat onder andere een optie om de window grootte te kunnen schalen, zodat er window groottes van meer dan 64 Kbytes kunnen worden gebruikt. De resultaten van deze tests zijn weergegeven in figuur 5.2.



Figuur 5.2: Buffergrootte en latency met RFC 1323

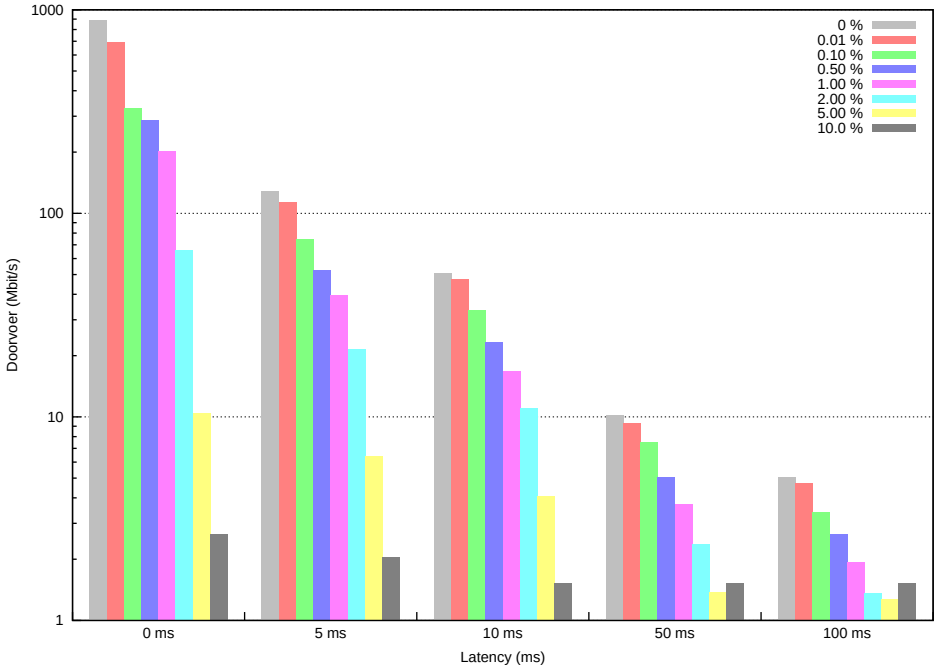
In deze figuur valt op dat de doorvoersnelheid door het aanzetten van de RFC 1323-extensie wel verbetert, maar bij hogere latencies achterblijft bij de theoretisch mogelijke doorvoersnelheid. Bij 50 ms en 100 ms latency is duidelijk het nut van grotere buffers te zien, ook al blijft de totale doorvoersnelheid ondermaats. Bij 100 ms latency en een buffer grootte van 16MB is de maximale doorvoersnelheid:

$$\frac{1000 \text{ ms}}{100 \text{ ms}} \times 16 \text{ Mbytes} = 160 \text{ Mbytes/s} = 1280 \text{ Mbits/s}$$

Aangezien de maximale doorvoersnelheid van de verbinding ongeveer 1000 Mbits/s is, zou het mogelijk moeten zijn de verbinding volledig te benutten. Hieraan is duidelijk te zien dat de testresultaten niet overeenkomen met het verwachtte resultaat. In hoofdstuk 5.1 is uiteengezet waar dit probleem zich zou kunnen bevinden.

5.4 Segmentverlies

Naast het testen van het effect van buffergroottes op de doorvoersnelheid, is ook getest wat het effect van segmentverlies is op de doorvoersnelheid. De eerste grafiek, figuur 5.3 geeft aan welke invloed segmentverlies heeft. Tijdens deze test is er gebruik gemaakt van een window van 64 Kbytes, waardoor bij het introduceren van latency de doorvoersnelheid zeer verminderd is. Uit de situatie waarbij 0 ms latency werd gesimuleerd is het effect van segmentverlies het duidelijkst af te leiden. Gesteld kan worden dat segmentverlies een drastische afname van doorvoersnelheid veroorzaakt. De figuur heeft daarom een logaritmische schaal; wanneer er een lineaire schaal gebruikt zou zijn, zou een groot gedeelte van de resultaten wegvallen ten opzichte van de situatie met 0 ms latency en 0% segmentverlies.

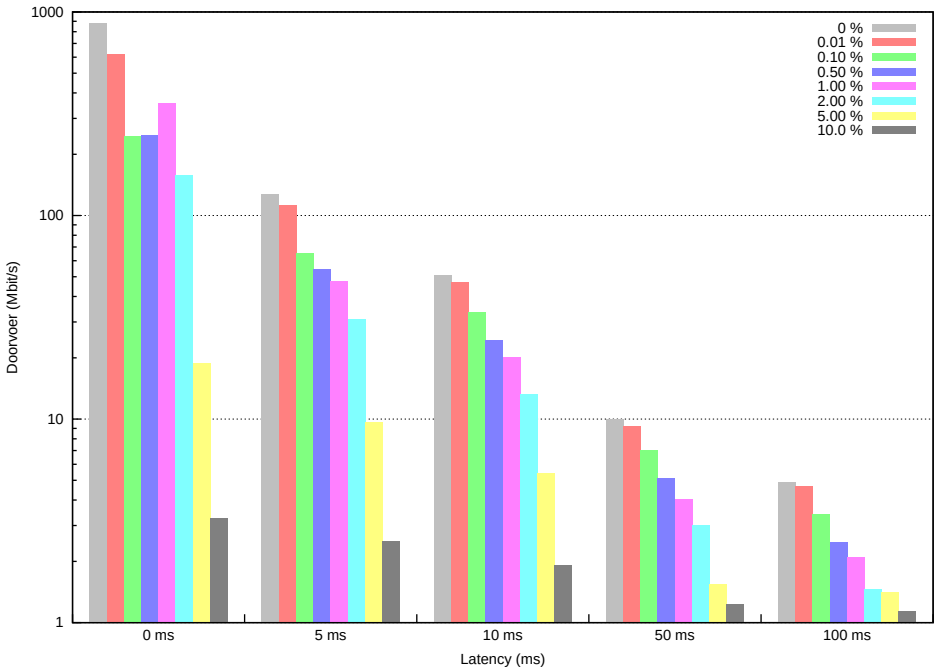


Figuur 5.3: Doorvoersnelheid met segmentverlies

5.5 TCP selective acknowledgement options

Een mogelijkheid om de negatieve effecten van segmentverlies zoveel mogelijk tegen te gaan is SACK. In figuur 5.4 zijn de resultaten van tests waarbij segmentverlies en latency werd gesimuleerd en SACK werd gebruikt op zowel de client als de server weergegeven. Een opvallende afwijking is dat de doorvoersnelheid bij 0 ms latency en 0,1% en 0,5% pakketverlies achterblijft bij 0 ms latency en 1% pakketverlies. Opvallend is ook dat de doorvoersnelheid nagenoeg gelijk is. Het is mogelijk dat deze afwijking zou verdwijnen wanneer de tests

vaker uitgevoerd zouden worden. Uit de figuur blijkt dat SACK een positief effect heeft op de doorvoersnelheid. In figuur 5.6 wordt dit duidelijker.



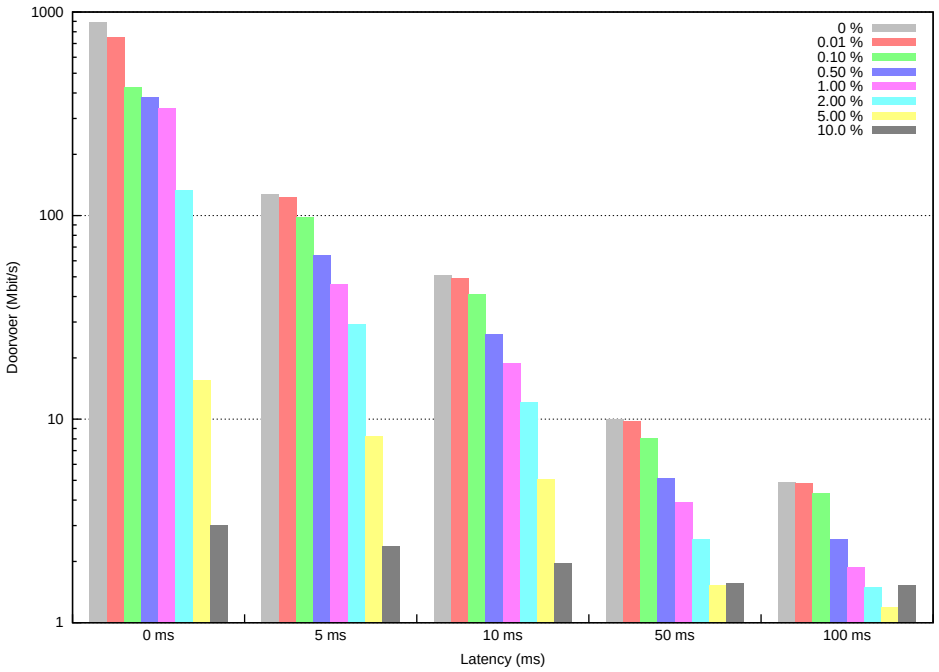
Figuur 5.4: Doorvoersnelheid met segmentverlies en SACK

5.6 NewReno modification

In figuur 5.5 zijn de resultaten van de test zonder SACK maar met NewReno weergegeven. De afwijkingen die zichtbaar waren in de vorige figuur treden hier niet meer op. Wel zijn er afwijkingen te zien bij een pakketverlies van 5% en 10% met een latency van 50 ms en 100 ms. Deze afwijkingen kunnen verklaard worden door de geringe doorvoersnelheid, waardoor pakketverlies grote schommelingen in de doorvoersnelheid kunnen veroorzaken. Doordat de hoeveelheid pakketten per seconde gering is, kan het detecteren van segmentverlies relatief lang duren, met schommelingen in de doorvoersnelheid tot gevolg.

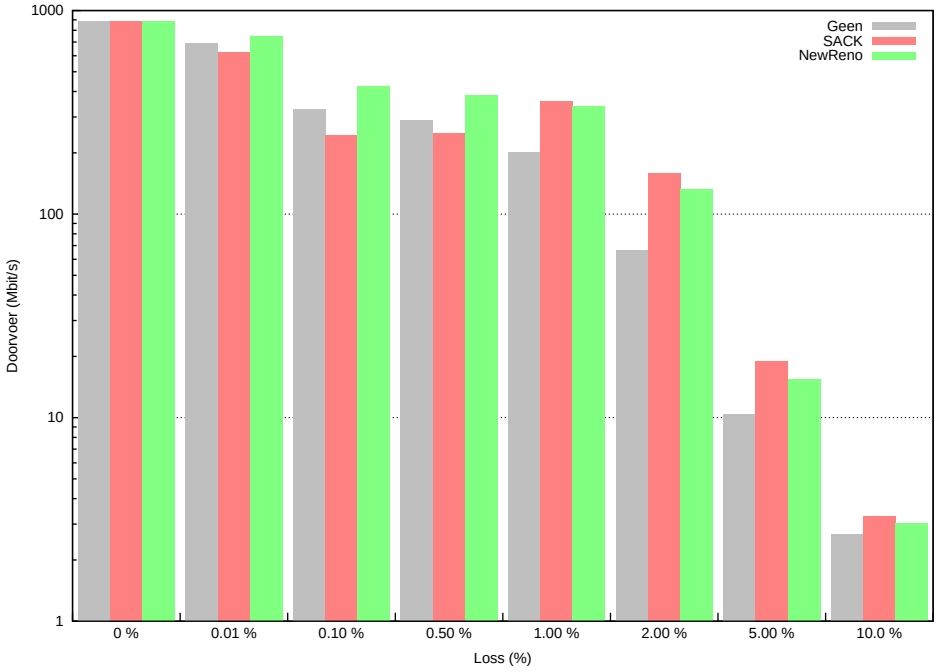
5.7 Vergelijking tussen SACK en NewReno

In figuur 5.6 is een overzicht gegeven van de doorvoersnelheid bij 0 ms latency en een buffergrootte van 64 Kbytes. Opvallend is dat in sommige gevallen de doorvoersnelheid zonder SACK groter is dan met SACK. Enige winst zou verklaard kunnen worden door de extra overhead van de SACK-optie in TCP-pakketten. Wanneer de SACK-opties worden meegestuurd, is er minder ruimte voor data, waardoor de doorvoersnelheid zal dalen. Ook kost de verwerking van



Figuur 5.5: Doorvoersnelheid met segmentverlies en NewReno

de SACK-opties extra tijd, ten nadele van de doorvoersnelheid. De verschillen in deze figuur zijn echter zo groot, dat ze niet verklaard kunnen worden met slechts de overhead van SACK. Mogelijkerwijs zouden meer tests de verschillen doen verkleinen. Ook de beperkte doorlooptijd van de tests (slechts 60 seconden) kan hierin een rol hebben gespeeld.



Figuur 5.6: Doorvoersnelheid met verschillende opties

Hoofdstuk 6

Conclusie

Het doel van het project bestond uit twee delen. Het eerste gedeelte besloeg uit het onderzoek naar de theoretische verbeteringen die bepaalde extensies moeten bieden. Dit onderdeel heeft een duidelijke verhandeling van deze extensies opgeleverd.

Het tweede gedeelte bestond uit het testen van een aantal van deze extensies in een testomgeving. Helaas hebben de problemen die zijn opgetreden met de testopstelling en de beperkte beschikbare tijd ervoor gezorgd dat de resultaten van dit gedeelte niet de betrouwbaarheid hebben behaald als oorspronkelijk de bedoeling was.

Dit project heeft echter wel een werkend testframework opgeleverd waarmee vervolgonderzoek eenvoudiger moet kunnen worden gedaan. Daarnaast zijn een aantal merkwaardige problemen in FreeBSD en Dummynet aan het licht gekomen, die verder onderzoek verdienen.

Ondanks deze problemen, is duidelijk te zien dat de geteste TCP-extensies zeker een bijdrage kunnen leveren aan het verhogen van de doorvoersnelheid op verbindingen met een hoog bandbreedte maal round-trip-time product en/of segmentverlies.

Bijlage A

FreeBSD configuratiebestanden

A.1 /boot/loader.conf “server” en “client”

```
kern.ipc.nmbclusters="65536"
kern.ipc.nmbufs="1024768"
kern.hz="2000"
```

A.2 /boot/loader.conf “netwerk”

```
ipfw_load="YES"
dummynet_load="YES"
kern.ipc.nmbclusters="65536"
kern.ipc.nmbufs="1024768"
kern.hz="2000"
```

A.3 /etc/inetd.conf “server” en “netwerk”

```
# Use port 1524 (ingreslock)
ingreslock        stream  tcp     nowait  root    /bin/sh      sh
```

A.4 /etc/ipfw.rules “netwerk”

```
add 10 pipe 1 ip from 192.168.100.2 to 192.168.200.2 xmit em1
add 20 pipe 2 ip from 192.168.200.2 to 192.168.100.2 xmit em0
add 50000 allow ip from any to any
```

A.5 /etc/rc.conf “client”

```
hostname="client.wind.surfnet.nl"
ifconfig_em1="192.87.110.40 netmask 255.255.255.128"
defaultrouter="192.87.110.1"
sshd_enable="YES"
usbd_enable="no"
```

```
cron_enable="no"
sendmail_enable="NONE"
ifconfig_em0="192.168.100.2 netmask 255.255.255.0"
static_routes="server"
route_server="192.168.200.0/24 192.168.100.1"
```

A.6 /etc/rc.conf “netwerk”

```
hostname="netwerk.wind.surfnet.nl"
ifconfig_em0="inet 192.168.100.1  netmask 255.255.255.0"
ifconfig_em1="inet 192.168.200.1  netmask 255.255.255.0"
inetd_enable="YES"
sshd_enable="YES"
usbd_enable="NO"
sendmail_enable="NONE"
cron_enable="no"
firewall_enable="yes"
firewall_type="/etc/ipfw.rules"
inetd_enable="yes"
```

A.7 /etc/rc.conf “server”

```
hostname="server.wind.surfnet.nl"
sshd_enable="YES"
usbd_enable="NO"
sendmail_enable="NONE"
cron_enable="no"
ifconfig_em1="192.87.110.38 netmask 255.255.255.128"
defaultrouter="192.87.110.1"
ifconfig_em0="192.168.200.2 netmask 255.255.255.0"
static_routes="client"
route_client="192.168.100.0/24 192.168.200.1"
inetd_enable="yes"
```

A.8 /etc/sysctl.conf “netwerk”

```
net.inet.ip.forwarding=1
net.inet6.ip6.forwarding=1
kern.polling.enable=1
kern.polling.idle_poll=1
net.inet.ip.intr_queue_maxlen=500
net.inet.ip.fastforwarding=1
```

A.9 /etc/sysctl.conf “server” en “client”

```
kern.polling.enable=0
net.inet.ip.intr_queue_maxlen=500
```

Bijlage B

FreeBSD scripts

B.1 /root/setpipe.sh “netwerk”

```
#!/bin/sh
ipfw pipe 1 config delay $1 plr $2
ipfw pipe 2 config delay $1 plr $2
```

B.2 /root/stoptest.sh “server”

```
#!/bin/sh
killall -9 iperf
```

B.3 test-fbsd.pl “server” en “client”

```
#!/usr/bin/perl

# This script is used to automate the tests which are done for this project.
# It is used on the client (client# perl test-fbsd.pl client <run>) and the
# server (server# perl test-fbsd.pl server <run>).

# This is an array of arrays defining the tests to be run. The inner arrays
# describe the options to change during the test run.
@tests = ( [ "bufspace", "rfc1323", "latency" ],
       [ "sack", "loss", "latency" ], [ "newreno", "loss", "latency" ] );

# This hashtable of arrays defines the values every option can have. All
# these values are looped through when running the tests with these
# options.
%values = ( "bufspace" => [ 65535, 262140, 1048560, 4194240, 16776960 ],
        "rfc1323" => [ 0, 1 ],
        "sack" => [ 0, 1 ],
        "newreno" => [ 0, 1 ],
        "loss" => [ 0, 0.0001, 0.001, 0.005, 0.01, 0.02, 0.05, 0.1 ],
        "latency" => [ 0, 5, 10, 50, 100 ]
      );

# These are the default values for the options. The defaults are restored
# before setting specific values. This is to ensure that relevant, but
# untested options don't interfer with the tests.
%defaults = ( "bufspace" => 65535,
```

```
          "rfc1323" => 1,
          "sack" => 0,
          "newreno" => 0,
          "loss" => 0,
          "latency" => 0,
          "rfc3390" => 0,
          "rfc3042" => 0,
          "delayed_ack" => 0,
          "inflight" => 0,
        );

# The rexec settings. $setpipecommand is the command run to set the pipe's
# parameters. It gets two arguments: latency and loss. $stoptestcommand is
# the command run on the server to stop the Iperf program.
$setpipecommand = "/root/setpipe.sh";
$stoptestcommand = "/root/stoptest.sh";
$rexec_port = 1524;
$network_host = "netwerk";
$server_host = "server";

# The test-commands
$client_testcommand = "iperf -c server -N -i 5 -t 60 -l4096";
$server_testcommand = "iperf -s -N";

my @command_permutations;
my $identity;
my $loss = 0;            # Default setting for loss
my $latency = 0;        # Default setting for latency
my $log_file;
my $log_directory;
my $run;

# Subroutines start here
sub set_bufspace($) {
    my $sockbuf = 2 * @_[0];

    system("sysctl kern.ipc.maxsockbuf=$sockbuf >/dev/null");
    system("sysctl net.inet.tcp.sendspace=@_[0] >/dev/null");
    system("sysctl net.inet.tcp.recvspace=@_[0] >/dev/null");
}

sub set_rfc1323($) {
    system("sysctl net.inet.tcp.rfc1323=@_[0] >/dev/null");
}

sub set_rfc3042($) {
    system("sysctl net.inet.tcp.rfc3042=@_[0] >/dev/null");
}

sub set_rfc3390($) {
    system("sysctl net.inet.tcp.rfc3390=@_[0] >/dev/null");
}

sub set_sack($) {
    system("sysctl net.inet.tcp.sack.enable=@_[0] >/dev/null");
}

sub set_newreno($) {
    system("sysctl net.inet.tcp.newreno=@_[0] >/dev/null");
}

sub set_delayed_ack($) {
    system("sysctl net.inet.tcp.delayed_ack=@_[0] >/dev/null");
}
```

```
sub set_inflight($) {
    system("sysctl net.inet.tcp.inflight.enable=@_[0] >/dev/null");
}

sub do_rexec($$) {
    system("echo @_[1] | nc @_[0] $rexec_port");
}

# Set the latency and loss on the remote simulation machine. Since we
# are using 2 pipes, divide with 2, otherwise the latency would be double
# that what we wanted.
sub do_latency_loss {
    $pipe_latency = $latency / 2;
    $pipe_loss = $loss / 2;
    do_rexec($network_host, "$setpipecommand $pipe_latency $pipe_loss");
}

sub set_latency($) {
    if ($identity eq "client") {
        $latency = @_[0];

        do_latency_loss;
    }
}

sub set_loss($) {
    if ($identity eq "client") {
        $loss = @_[0];

        do_latency_loss;
    }
}

# This subroutine is called to perform one client-side test. The client waits
# a few seconds to give the server time to setup. It then runs the testcommand.
# After the test has completed, it remotely executes the stop command on the
# server.
sub do_client_test() {

    # Create the log directory and enter it.
    mkdir $log_directory || die "Could not create test log directory '$log_directory'!";
    chdir $log_directory || die "Could not enter test log directory '$log_directory'!";

    # Wait for the server side to get things going
    sleep(3);

    # Run the test
    system($client_testcommand." >$log_file");

    # Go back to the run log directory
    chdir "..";

    # Let the server side go to the next test
    do_rexec($server_host, $stoptestcommand);
}

# This subroutine is called to perform one server-side test. It simply runs
# the test program. The test program is killed on the server by the client,
# so this function ends when the client has ended, too.
sub do_server_test() {
    system($server_testcommand);
}
```

```
# Do the test!
sub do_test() {
    if ($identity eq "client") {
        do_client_test
    }
    if ($identity eq "server") {
        do_server_test
    }
}

# Configure all options with the default value.
sub set_defaults() {
    foreach $option (keys %defaults) {
        eval "set_$option($defaults{$option})";
    }
}

# This is a recursive function. It takes two parameters:
# an array of options and the permutation of values at this point.
# For every option is tries all the values for that option and recurses into
# itself with argument 1 shifted one option and the second argument filled in
# with the values for the current option. This way all permutations of the
# option's values are found. They are then stored as arrays in the array
# @command_permutations.
sub permutate_options(@@) {
    my @options_list = @{$_[0]};
    my @values_list = @{$_[1]};

    if ($#values_list == -1) {
        @command_permutations = nil;
    }

    if ($#options_list == 0) {
        for $value (@{$values{@options_list[0]}}) {
            my @new_values_list = @values_list;
            push @new_values_list, $value;
            push @command_permutations, [@new_values_list];
        }
    } else {
        my @new_options_list = @options_list[1..$#options_list];

        for $value (@{$values{@options_list[0]}}) {
            my @new_values_list = @values_list;
            push @new_values_list, $value;
            permutate_options([@new_options_list], [@new_values_list]);
        }
    }
}


sub usage() {
    print "Usage: test-fbsd.pl <client|server> <run>\n";
    exit(0);
}


# Main routine starts here
if ($#ARGV < 1) {
    usage();
}

if ((@ARGV[0] ne "client") && (@ARGV[0] ne "server")) {
    usage();
}
```

```
$identity = @ARGV[0];
$run = @ARGV[1];

# If this is the client, create and enter 'run'-directory
if ($identity eq "client") {
    mkdir $run || die "Could not create run log directory '$run'!";
    chdir $run || die "Could not enter run log directory '$run'!";
}

# Go through all tests doing the following:
# - compile a list of all values to be set for all options
# - for every option-set (which means: for every test), first set the options
#   to their default values
# - then run the client/server side tests
for $test (@tests) {

    permutate_options([@{$test}], [ ]);

    foreach $permutation (@command_permutations) {
        $noperms = $#{@{$permutation}};

        if ($noperms == -1) {
            next;
        }

        # This is the per-test loop
        set_defaults;

        # Set the configuration options
        $log_file="";
        for $i (0..$noperms) {
            eval "set_@{$test}[$i](@{$permutation}[$i])\n";

            $log_file .= @{$permutation}[$i];
            if ($i < $noperms) {
                $log_file .= "-";
            }
        }

        # Create a per logfile date/time stamp and use it for the
        # logfile name.
        ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst)=localtime(time);
        $log_file = sprintf "%04d%02d%02d-%02d%02d:$log_file.log",
            $year+1900, $mon+1, $mday, $hour, $min;

        $log_directory="";
        for $i (0..$noperms) {
            $log_directory .= "@{$test}[$i]";
            if ($i < $noperms) {
                $log_directory .= "-";
            }
        }

        # Run test!
        do_test;
    }
}
```

Bijlage C

Parsing scripts

C.1 averages.pl

```
#!/usr/bin/perl

# This script runs through the iperf output log files for a given run and
# averages the throughput per log file. The filename of the iperf log file
# and the average are then written to a file 'averages.log' in the run's
# main directory.
#
# Example:
# ./averages.pl 1
#
# This will create a file 1/averages.log containing the averages for all
# tests in run 1.

# Variables
my $run;
my $AVGFILE;     # Filehandle for the averages file

sub usage() {
    print "Usage: parserun.pl <run>\n";
    exit(1);
}

# Rounds a given floating point number to the nearest integer
sub round($) {
    my($number) = shift;
    return int($number + .5 * ($number <=> 0));
}

# Add averages for the given filename to the averages file
sub add_average($) {
    my $fname = shift;

    open(FP, "<$fname") or die "Can't open logfile '$fname' for reading!\n";

    my $test_interval_duration = 0;
    my $test_mbits = 0;
    my $test_intervals = 0;

    while(<FP>) {
        next if /^------/;        # Skip 'useless' info
        next if /^Client connecting/;
        next if /^TCP window/;
```

```
        next if /connected with/;

        @columns = split;

        # Construct the interval used for this line of output
        $interval = @columns[2];
        if (!($interval =~ /^.+-.+$/)) {
            $interval .= @columns[3];
        }

        # Find out how long this interval was and save it
        @timing = split("-", $interval);
        $interval_duration = @timing[1] - @timing[0];
        if ($test_interval_duration == 0) {
            $test_interval_duration = $interval_duration;
        }

        $speed = @columns[$#columns - 1];
        $bitsyness = @columns[$#columns];

        # Recalculate speed to MBits/sec
        $speedmbits = $speed * 1024 if ($bitsyness eq "GBits/sec");
        $speedmbits = $speed if ($bitsyness eq "Mbits/sec");
        $speedmbits = $speed / 1024 if ($bitsyness eq "KBits/sec");
        $speedmbits = $speed / (1024 * 1024) if ($bitsyness eq "Bits/sec");
        $speedmbits = $speed * 1024 * 8 if ($bitsyness eq "GBytes/sec");
        $speedmbits = $speed * 8 if ($bitsyness eq "MBytes/sec");
        $speedmbits = $speed * 8 / 1024 if ($bitsyness eq "KBytes/sec");
        $speedmbits = $speed * 8 / 1024 / 1024 if ($bitsyness eq "Bytes/sec");

        # Only if this line wasn't the average iperf made itself,
        # add the number of mbits sent this interval to the total
        # number of mbits sent this test. Also raise the number of
        # intervals, because that number is used to calculate the
        # average bitrate for all intervals.
        if ($interval_duration == $test_interval_duration) {
            $test_mbits += $speedmbits;
            $test_intervals += 1;
        }
    }
    close(FP);

    # We now have all the mbits summed up and the number of intervals. So
    # it's time to calculate an average.
    $average_mbits = $test_mbits / $test_intervals;

    # Finally, write the log filename and it's average to file..
    print AVGFILE "$fname $average_mbits\n";
}

# Create averages for logfiles in the given directory
sub do_averages($) {
    opendir(DIR,@_[0]) or die "Can't open logfile directory $name!\n";
    @filenames = readdir(DIR) or die "Can't read logfile directory $name contents\n";
    closedir(DIR);

    foreach $filename (@filenames) {
        if ($filename =~ /\.log$/) {
            add_average("@_[0]/$filename");
        }
    }
}
```

```
# Main routine starts here

if ($#ARGV < 0) {
    usage();
}

$run = @ARGV[0];

chdir $run or die "Could not enter run logfile directory '$run'!\n";

# Create averages file
open(AVGFILE, ">averages.log") or die "Can't open 'averages.log' for writing!\n";

# Get directory listing of the run logfile main directory
opendir(DIR,".") or die "Can't open the run logfile directory!\n";
@dirnames = readdir(DIR) or die "Can't read run logfile directory's contents!\n";
closedir(DIR);

foreach $dirname (@dirnames) {
   next if ($dirname eq ".");   # skip the current directory entry
   next if ($dirname eq "..");  # skip the parent  directory entry

   if (-d $dirname){            # is this a directory?
    do_averages($dirname);
   }
}

close(AVGFILE);
```

C.2 merge-avg.pl

```
#!/usr/bin/perl


# This script accepts as arguments filenames of averages.log files from
# test runs. For every test in those log files (the logs should contain
# throughputs for the same tests, obviously), this script creates an
# average throughput for all runs. It outputs the test name (without
# timestamp) and the averaged throughput for that test on stdout.
#
# Example:
# ./merge-avg.pl 1/averages.log 2/averages.log

my %values;
my $numargs = $#ARGV + 1;        # Save number of input files
my @tests;

while(<>) {
    @columns = split;

    $testname = @columns[0];
    $result = @columns[1];

    $testname =~ s#/[0-9]+-[0-9]+:#-#;    # Filter out the timestamp

    # Save this test's name in an array to be able to retain the right
    # ordering. Only save the name once.
    if (! $values{$testname}) {
        push(@tests, $testname);
    }
    $values{$testname} += $result;
}
```

```
# Loop through the array of testnames and use the tests found in there
# to look up the totals in the hashtable. Then divide the totals by the
# number of log files parsed and output the found average.
foreach $key (@tests) {
    $value = $values{$key} / $numargs;    # Merge!
    print "$key $value\n";
}
```

Bibliografie

[1] Allman, M., S. Floyd en C. Partridge, *RFC 3390 - Increasing TCP's Initial Window*, oktober 2002.
http://www.faqs.org/rfcs/rfc3390.html

[2] Allman, M., V. Paxson en W. Stevens, *RFC 2581 - TCP Congestion Control*, april 1999.
http://www.faqs.org/rfcs/rfc2581.html

[3] Braden, R., *RFC 1122 - Requirements for Internet Hosts - Communication Layers*, oktober 1989.
http://www.faqs.org/rfcs/rfc1122.html

[4] Floyd, S., T. Henderson, *RFC 2582 - The NewReno Modification to TCP's Fast Recovery Algorithm*, april 1999.
http://www.faqs.org/rfcs/rfc2582.html

[5] Floyd, S., T. Henderson en A. Gurtov, *RFC 3782 - The NewReno Modification to TCP's Fast Recovery Algorithm*, april 2004.
http://www.faqs.org/rfcs/rfc3782.html

[6] Floyd, S., J. Mahdavi, M. Mathis, M. Podolsky, *RFC 2883 - An Extension to the Selective Acknowledgement (SACK) Option for TCP*, juli 2000.
http://www.faqs.org/rfcs/rfc2883.html

[7] FreeBSD Project, *FreeBSD website*, juni 2005.
http://www.freebsd.org/

[8] Hemminger, S., *Network Emulator website*, juni 2005.
http://developer.osdl.org/shemminger/netem/

[9] Internet Assigned Numbers Authority, *IANA website*, juni 2005.
http://www.iana.org/

[10] Jacobson, V., R. Braden en D. Borman, *RFC 1323 - TCP Extensions for High Performance*, mei 1992.
http://www.faqs.org/rfcs/rfc1323.html

[11] Mathis, M., J. Mahdavi, S. Floyd en A. Romanov, *RFC 2018 - TCP Selective Acknowledgement Options*, oktober 1996.
http://www.faqs.org/rfcs/rfc2018.html

[12] Microsoft Corporation, *Microsoft Windows Family website*, juni 2005.
http://www.microsoft.com/windows

[13] National Institute of Standards and Technology - Internetworking Technology Group (ITG), *NISTNet website*, juni 2005.
http://www-x.antd.nist.gov/nistnet/

[14] Postel, J., *RFC 760 - Internet Protocol*, januari 1980.
http://www.faqs.org/rfcs/rfc760.html

[15] Postel, J., *RFC 791 - Internet Protocol*, september 1981.
http://www.faqs.org/rfcs/rfc791.html

[16] Postel, J., *RFC 793 - Transmission Control Protocol*, september 1981.
http://www.faqs.org/rfcs/rfc793.html

[17] Rizzo, L., *Dummynet website*, juni 2005.
http://info.iet.unipi.it/~luigi/ip_dummynet/

[18] SURFnet B.V., *SURFnet website*, juni 2005.
http://www.surfnet.nl/info/home.jsp

[19] University of Illinois, *Iperf website*, juni 2005.
http://dast.nlanr.net/Projects/Iperf/

[20] Internet2 Land Speed Record, juni 2005.
http://lsr.internet2.edu/