



Faculteit Ingenieurswetenschappen
Vakgroep Informatietechnologie
Voorzitter: Prof. Dr. Ir. P. LAGASSE

Video Caching in het Access Netwerk

door

Hendrik BUYLE

Promotoren:

Prof. Dr. Ir. B. DHOEDT

Prof. Dr. Ir. F. DE TURCK

Scriptiebegeleiders:

Wim VAN DE MEERSSCHE

Tim WAUTERS

Scriptie ingediend tot het behalen van de academische graad
van licentiaat in de informatica

Academiejaar 2005–2006

Voorwoord

Ik wil heel even tijd nemen om iedereen te bedanken die mij gesteund hebben bij het schrijven van deze scriptie.

Als eerste wil ik mijn beide promotoren Prof. Bart Dhoedt en Prof. Filip De Turck bedanken, omdat jullie mij de kans gaven deze thesis te doen en steeds klaar stonden om mij te helpen.

Bovendien moet ik ook zeker Wim Van de Meerssche en Tim Wauters, mijn begeleiders, bedanken. Iedere vergadering opnieuw stuurden jullie mij in de goeie richting.

En - last but not least - mijn familie en vrienden. Omwille van de spreekwoordelijke trap onder mijn kont die ik van jullie kreeg iedere keer als ik niet aan mijn thesis werkte terwijl ik dat wel had moeten doen.

Hendrik Buyle, 2 juni 2006

Toelating tot bruikleen

“De auteur geeft de toelating deze scriptie voor consultatie beschikbaar te stellen en delen van de scriptie te kopiëren voor persoonlijk gebruik.

Elk ander gebruik valt onder de beperkingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit deze scriptie.”

Hendrik Buyle, 2 juni 2006

Video Caching in het Access Netwerk

door

Hendrik BUYLE

Scriptie ingediend tot het behalen van de academische graad
van licentiaat in de informatica

Academiejaar 2005–2006

Promotoren:

Prof. Dr. Ir. B. DHOEDT

Prof. Dr. Ir. F. DE TURCK

Scriptiebegeleiders:

W. VAN DE MEERSSCHE

T. WAUTERS

Faculteit Ingenieurswetenschappen
Universiteit Gent

Vakgroep Informatietechnologie
Voorzitter: Prof. Dr. Ir. P. LAGASSE

Samenvatting

Deze thesis handelt over het plaatsen van proxy servers op tussenliggende nodes in een netwerk dat gebruikt wordt om video over te streamen en welke effecten deze proxy servers zullen hebben op de belasting van de verschillende componenten van dat netwerk. We zullen achtereenvolgens onderzoeken of het al dan niet nuttig is om proxies te plaatsen, waar die proxies het best geplaatst kunnen worden en welke interne caching algoritmes die proxy servers het beste hanteren in welke omstandigheden. We zullen merken dat de aard van het verkeer over het netwerk makkelijk uitgebuit kan worden door bepaalde algoritmes om zo betere resultaten te bekomen. Daarna zullen we bekijken of - indien er zich meerdere proxy servers in het netwerk bevinden - er een zinvolle vorm van samenwerking tussen deze proxies bestaat. Het uiteindelijke doel van deze thesis is om een duidelijk overzicht te bieden van dergelijke verbeteringen aan het netwerk en hoe deze de belasting van de verschillende componenten en het netwerkverkeer zullen beïnvloeden.

Trefwoorden

Caching, Access Netwerk, Video Streaming, Time-shifted TV, Caching algoritme

Video Caching in the Access Network

Hendrik Buyle,

Supervisor(s): Bart Dhoedt, Filip De Turck

Abstract—In this document we talk about installing caches on the routers in the access network of a digital television provider. We talk about the effects those caches have on various components in the network. We research whether or not those proxy servers are at all useful, where in the network those servers should be placed and which caching algorithms should be used by those proxy servers in which circumstances. In a last part of this article we suggest a few means of cooperation between those proxy servers.

Keywords— cache, access network, video streaming, time-shifted TV, caching algorithm

I. INTRODUCTION

THE first part of this paper talks about a few basic concepts concerning video streaming, time-shifted television and RTSP Proxy servers in general.

Time-shifted television is a service which allows customers to pause, rewind and fast forward live television. It also allows the customer to start watching a particular show with a delay anywhere between a second and a few hours.

One of the most important aspects of Time-shifted Television is the demand curve. The demand curve describes how the popularity of a certain show evolves over time starting from the moment it is first broadcasted. The curve shows how many viewers will request to watch the show after it has gone live. It is important for the digital television provider to predict the demand curves as their shape will influence the results of proxy servers.

Caches inside the network can be used in roughly three different ways. They can be used in combination with a client (a settopbox with caching capabilities), they can be used inside the core network in order to lessen the load of the streamer. They can also be used on the intermediate nodes inside the access network.

The last part of this section gives an overview of how proxy servers work from a conceptual point of view. We describe the architecture of the proxy server as the sum of just three components: the cache itself, an algorithm called the cache algo and an algorithm called the topo algo. The cache stores the captured streams. The cache algo is the part of the proxy which decides on all actions to be taken by the cache, such as: should a certain piece a stream be cached or should another piece which already is inside the cache be removed from it? The topo algo is the algorithm which handles all decisions of the cache concerning cooperation with other proxy servers. After a request for a certain part of a show is received which cannot be served from the cache, that request must be forwarded to another proxy server or to a streamer. The algorithm that decides which component to forward the request to is the topo algo.

II. THE SIMULATOR

Because almost all results used in this document are generated by simulator, it is important to have a section on the working of

that simulator.

We need to have a simulator because it just would not be possible to try out the algorithms in a real test lab. The amount of machines needed to simulate an access network large enough to be of interest to us would just be too high.

The most important thing to know about the simulator is how the simulations differ from the real world. There are a total of six ways in which a simplification made by the simulator or a technical issue inside the simulator can influence the simulation results. I feel the reader must know about these simplifications if he is to interpret a simulation result himself or understand the conclusions i drew from my simulation results [1].

III. NON-COOPERATIVE ALGORITHMS

One of the features most described caching algorithms have in common is that they work with a sliding interval. An interval can be seen as a window in time, a succession of smaller segments belonging to the same television channel. Each of these algorithms caches the segments inside the interval and removes other segments who have fallen out of the interval. There is one interval for each television channel. Another feature of these algorithms is to divide the cache into two parts, an S and an L part. The S part will be used for temporary storage of segments in between cache recalculations. The L part will be used to store the segments as decided by the algorithm.

The algorithms described in this document are “Prefix Algorithm”, “Sliding interval with fixed interval length”, “Sliding interval with variable interval length” and “Survival of the fittest algorithm”.

- The prefix algorithm stores the first few minutes of every show with the intention to lessen the startup delay of a new RTSP stream.
- The Sliding Interval algorithm with fixed interval length uses one interval for each television channel and lets that interval slide, the size of the interval is fixed for each channel.
- The Sliding Interval algorithm with variable interval length also uses one interval for each channel. The main difference between both algorithms is that this one allows its intervals to grow and shrink to respond to a change in popularity.
- The Survival of the Fittest algorithm is like the sliding interval algorithm with the variable interval lengths but allows multiple intervals for each channel and considers intervals with just one segment in it to be intervals just as well.

IV. MEANS OF COOPERATION

In this section we talk about how Proxy server can cooperate and communicate with each other in order to improve results. Examples of information proxy servers wish to share are: the amount of free space in their caches, their level of performance, the current amount of streams passing through the links con-

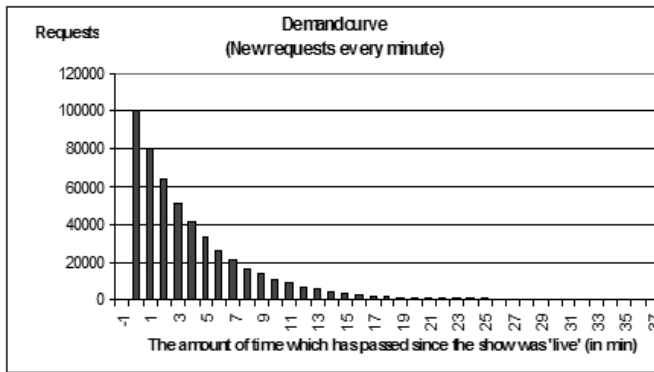


Fig. 1. Demandcurve

nected to the proxies, ... We suggest a centralized architecture is used to exchange this information with other proxies, one of the advantages of using a centralized server is that it can be used as a discovery service.

There are 4 different ways in which proxy servers can put shared information into good use:

1. In the calculation of the popularity of the segments
2. Enabling the promiscuous mode of the interfaces of the proxy server: this allows a proxy to capture and cache a stream which is passing through the router it is installed on.
3. Intelligent forwarding of requests: We suggest 4 different algorithms for the forwarding of requests:
 - (a) Forwarding to the central server.
 - (b) Normal Topo Algo: the proxy looks for another proxy which does have the segment to forward the request to
 - (c) Shared Topo Algo: the proxy looks for another proxy which does have the segment to forward the request to, if it can't find one, it will forward the request to the proxy which has the most free space in its cache. This makes that second cache request the segment from the central server, allowing it to cache the segment.
 - (d) Complex Topo Algo: the proxy looks for another proxy which does have the segment to forward the request to, if it can't find one, it will forward the request to the proxy which it has chosen based on a formula taking free space on that cache, performance of that proxy, weight of the path to that proxy, weight of the path from the streamer to that proxy, into account.
4. Intelligent caching: We suggest an algorithm which allows the provider to limit the number of copies of each segment on all proxy servers in the entire network to a specified amount.

V. SIMULATION RESULTS

Using the results from several simulations we can come to the following results:

The more flexible a caching algorithm is the better its results. Survival of the Fittest Algorithm gives us the best results, Sliding interval algorithm with fixed interval length gives us the worst.

When there is no cooperation between caches the used topology has no effect on the performance of the caches.

The more flexible a caching algorithm is, the more its performance will improve if the caching algorithm would use a larger cache.

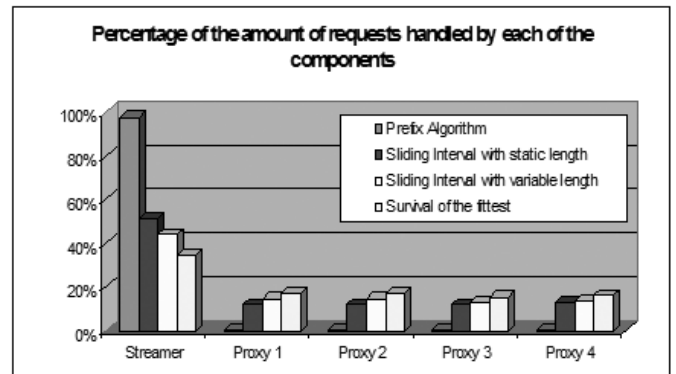


Fig. 2. Different caching algorithms

In most cases algorithms that need a value for ΔT will perform best with ΔT as small as possible.

Using promiscuous mode in combination with a simple forwarding strategy will result in a serious improvement of results.

The forwarding strategy with the best results is the Shared Topo Algo

If one wishes one could relieve the load on the streamer by using the Shared Cache Algo, the price to pay is a serious increase of the load of most links.

VI. CONCLUSIONS

In this paper several caching algorithms and several request forwarding strategies have been presented. We should remember that the Survival of the Fittest algorithm is the best Caching algorithm and that Shared Topo Algo is the forwarding strategy with the best results. Caches are very useful inside an access network and even inside the core network of a provider. Cooperation between those caches is also useful but it comes with a price. That price is an increase of the load of most of the links.

ACKNOWLEDGMENTS

The author would like to acknowledge, next to the (head)supervisors, the daily supervisors: Wim Van de Meerssche and Tim Wauters. My final paper would not have been possible without their help.

REFERENCES

- [1] Hendrik Buyle, *Video caching in het access netwerk* final paper University Ghent, 2006.

Inhoudsopgave

1	Inleiding	1
1.1	Situering van de scriptie	1
1.2	Doelstelling	3
1.3	Methodologie	4
2	Kenmerken van video streaming & caching in access netwerken	5
2.1	Demand	5
2.2	Opbouw van een access netwerk	8
2.3	De werking van een proxy server	11
2.3.1	Algemeen	11
2.3.2	Enkele ontwerpsbeslissingen	14
3	De TsTV Simulator	17
3.1	Waarom een simulator?	17
3.2	Werking van de simulator	18
3.2.1	Werking	18
3.3	Gemaakte vereenvoudigingen	21
4	Niet coöperatieve caching algoritmes	24
4.1	Algemene principes waarop de algoritmes gebaseerd zijn	24
4.1.1	Het interval	24
4.1.2	De S en L cache	28
4.1.3	Populariteit	29
4.2	Specifieke algoritmes	31
4.2.1	Het Prefix algoritme	31
4.2.2	Sliding Interval algoritme met vaste lengte	33

4.2.3	Sliding Interval algoritme met variabele lengte	34
4.2.4	Het survival of the fittest algoritme	36
4.3	Simulatie resultaten	37
4.3.1	Vergelijkende studie	37
4.3.2	De algoritmes specifiek	41
5	Samenwerkende proxy servers	47
5.1	Hoe?	47
5.2	Een overzicht van de samenwerkingsvormen	49
5.2.1	Berekenen van de populariteit van segmenten	49
5.2.2	Interfaces in promiscuous mode	50
5.2.3	Requests intelligent doorsturen	53
5.2.4	Intelligente caching beslissingen nemen	58
6	Conclusies	62
6.1	Conclusies	62
6.2	Toekomstig werk	63
A	De TsTV Simulator	65
A.1	Uitvoeren	65
A.2	Inputfiles maken	66
A.2.1	Het configuratiebestand	66
A.2.2	Het topologiebestand	68
A.2.3	Het demandbestand	70
A.3	Output files lezen	73
A.3.1	De clients	73
A.3.2	De proxies	74
A.3.3	De links	75
A.3.4	De streamers	76

Hoofdstuk 1

Inleiding

In dit stuk wordt besproken wat deze scriptie precies inhoudt en waarom het belangrijk is dit onderwerp te bestuderen. De gehanteerde methodologie zal uitgebreid overlopen worden en als laatste wordt in dit hoofdstuk kort uitgelegd wat we proberen te bereiken op gebied van resultaten.

1.1 Situering van de scriptie

Er is geen ontkennen aan, video streaming neemt een steeds meer belangrijk wordende plaats in ons leven in. Digitale televisie is hier en alles wijst erop dat ze hier zal blijven. En met de introductie van de digitale televisie kwamen een hele reeks van nieuwe mogelijkheden op de proppen, de belangrijkste hieronder zijn *Time-shifted Televisie* en *Video On Demand*.

Onder **Time-shifted Televisie (TsTV)** verstaan we de service aangeboden door de provider die het de klanten toelaat een televisieprogramma dat op dat moment “live” bezig is, te behandelen al was het een film op een video cassette. Een televisieprogramma enkele minuten pauzeren zonder iets te missen van het programma, beginnen kijken enkele minuten nadat de “live” uitzending begonnen is, zelfs terug en verder spoelen van het programma zijn services die we TsTV noemen.

Als we in de loop van deze scriptie het woord “live” gebruiken, dan bedoelen we dit niet in de context van een live opgenomen programma, maar in functie van de oorspronkelijke uitzendtijd van dat programma. Men kan live kijken naar de film van 20 uur als men om 20 uur stipt begint

te kijken, ook al is die film vijf jaar geleden opgenomen. Bij een film die beschikbaar wordt gesteld via de elektronische videotheek van de provider kan men het moment van “live-zijn” definiëren als het moment dat de film voor het eerst beschikbaar wordt gesteld aan het publiek. Dit klinkt misschien triviaal maar in het vervolg van deze scriptie zullen we nog dikwijls gebruik maken van termen zoals “een aanvraag om een live segment” en het is belangrijk dat hier geen misverstanden rond ontstaan.

Video on Demand (VoD) is dan weer de service die het beste kan omschreven worden als de online videotheek die al dan niet tegen betaling ter beschikking wordt gesteld aan de klanten. Een dikwijls grote groep programma’s die gedurende enkele dagen, weken of zelfs maanden beschikbaar zijn en eerder sporadisch zullen bekeken worden. Het onderscheid tussen VoD en TsTV wordt voornamelijk gemaakt in de periode die verstreken is sinds het “live zijn” van het programma en het moment dat de gemiddelde kijker er maar kijkt. Komen de meeste aanvragen binnen enkele minuten dan spreken we meestal van TsTV, zijn de aanvragen verspreid over een langere periode dan zal men het eerder over VoD hebben.

Reken dat er in België alleen al miljoenen potentiële abonnementen zijn en dat een één uur durende MPEG-1 film rond de 675 MiB in grootte is dan beseft men snel dat we het hier hebben over enorme hoeveelheden data die door een beperkt aantal streamers moeten geleverd worden. Als een provider 100,000 abonnees heeft en in elk van deze gezinnen gemiddeld een drietal uur televisie gekeken wordt, reken dan op een dagelijks verbruik van *193 TebiByte* bandbreedte, overhead en gewoon internetverkeer over dat accessnetwerk niet meegerekend.

Die bandbreedte hoeft niet per sé verbruikt te worden want ervaringen uit het dagelijkse leven leren ons dat de grote meerderheid van de mensen naar dezelfde 3 á 4 televisiezenders kijken. Als men er in slaagt om die relatief kleine hoeveelheid extreem populaire data dichterbij de klant te bewaren dan zou dat al een aanzienlijke verbetering zijn ten opzichte van de situatie zoals die nu is.

Er is zeker genoeg potentieel voor dergelijke technieken. We gaan verder met de fictieve digitale televisie provider die dagelijks 193 TiB te verwerken krijgt. We stellen dat die 50 televisie kanalen aanbiedt, dat elk van deze kanalen 24 uur per dag uitzendt en die provider slaagt er

in om die gegevens dicht bij de klant te cachen. Op deze manier kan *in theorie* de workload van de streamers verminderd worden tot 791 GiB. Dit is slechts 0.4 % van de workload die de streamers eerder te verduren kregen. Uiteraard zullen zo'n extreme resultaten nooit in de praktijk te brengen zijn maar dit voorbeeld illustreert het enorme potentieel van caches in een accessnetwerk.

1.2 Doelstelling

In het begin van deze scriptie zal uitleg verschaft worden bij de verschillende **aspecten** van video streaming die relevant zijn voor dit onderzoek. Er zal verteld worden over onderwerpen zoals de vorm van een demandcurve, de topologie van een accessnetwerk, de algemene werkwijze gevolgd door een proxy, en verschillende andere onderwerpen, in de hoop dat een lezer van deze scriptie meer vertrouwd wordt met het onderwerp ervan.

Een tweede doelstelling van deze thesis is het scheppen van een duidelijk **overzicht** van enkele caching algoritmen bruikbaar voor video streaming. We proberen de lezer al deze algoritmen goed te leren kennen en te laten begrijpen hoe ze werken, zodat hij weet welke hun voor- en nadelen zijn, in welke omstandigheden of topologiën ze het meest van nut zijn en hoe hun prestaties zich verhouden ten opzichte van elkaar.

Een andere doelstelling is om de lezer duidelijk te maken hoe de samenwerking tussen verschillende proxy servers kan verlopen. Er wordt opnieuw een overzicht gegeven van verschillende methodes die de proxies kunnen gebruiken om samen te werken en zo tot betere prestaties te komen. Er zal besproken worden op welke manier deze methodes van elkaar verschillen, zowel qua werking als qua prestatie in verschillende omstandigheden.

Als laatste hoop ik tot enkele duidelijke resultaten te komen. Zijn er topologieën, caching methodes of samenwerkingsmethodes die er in bepaalde situaties consequent bovenuit steken? Zo ja, welke zijn die? Gebruik makende van welke configuraties werken de algoritmes het beste? Deze scriptie zal ook advies proberen geven over de te gebruiken algoritmes in verschillende omstandigheden.

1.3 Methodologie

Op gebied van gehanteerde methodologie kan deze thesis opgedeeld worden in drie duidelijk gescheiden fases:

1. De literatuurstudie: De literatuurstudie omvat onderwerpen als RTSP, MPEG, de aard van Time-shifted television, de implementatie details van een bestaande RTSP proxy server en verschillende documenten omtrent de verschillende reeds bestaande caching algoritmes geschikt voor video.
2. Het bouwen van een Simulator: op het einde van de literatuurstudie was duidelijk geworden dat, als men waardevolle informatie wil over de werking van proxies in een netwerk, men verplicht is om een simulator te schrijven. Het bouwen van zo'n simulator leerde me bovendien ook veel van de details kennen omtrent de verschillende algoritmen. Eens voldoende algoritmes geïmplementeerd waren, kon het eigenlijke onderzoek beginnen.
3. Het onderzoek: Gebruik makende van de simulator konden verschillende algoritmen in verschillende topologiën gesimuleerd worden en iedere keer de resultaten geanalyseerd worden, de resultaten vergeleken worden en tot de juiste conclusies gekomen worden.

Hoofdstuk 2

Kenmerken van video streaming & caching in access netwerken

In dit hoofdstuk wordt dieper ingegaan op begrippen zoals *demandcurves*, de kenmerken en onderdelen van een access netwerk & de algemene werking van proxy servers. Een grondige kennis van deze onderwerpen is dan ook een vereiste om te kunnen beginnen met de studie van de algoritmen zoals dit zal gebeuren in hoofdstuk 4 en 5.

2.1 Demand

In het vorige hoofdstuk werd gezegd dat het belangrijkste verschil tussen TsTV en VoD erin bestaat dat de periode tussen het “live zijn” en het bekijken van het programma bij VoD veel groter kan worden dan bij TsTV programma's. Toch moeten we in dit hoofdstuk opmerken dat dit verschil voor deze thesis helemaal niet relevant is. Het is van geen belang op welke manier een programma aan de klant aangeboden wordt, ieder programma zal bepaald worden door een **demandcurve** en het is enkel de vorm van deze curve die belangrijk is voor ons. De demand-curve beschrijft het aantal aanvragen die gedaan worden voor een bepaald programma door de clients binnen het netwerk, in de loop van de tijd, beginnende met het moment van het “live-zijn”

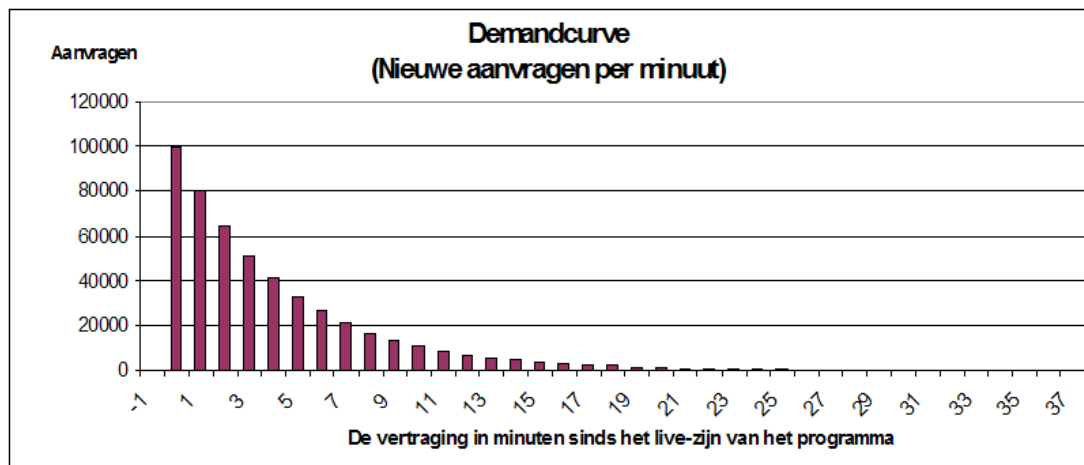
Het is helemaal niet belangrijk om te weten als onderdeel van welke service een programma gestreamd wordt, het is belangrijk dat men goed op de hoogte is welke demandcurve hoort bij dit programma. Het is natuurlijk wel juist dat verschillende services heel verschillende typische demandcurves zullen hebben. Het 7 uur journaal zal bijvoorbeeld heel dikwijls aangevraagd

worden gedurende eerste twee á drie uur na uitzending waarna het aantal nieuwe aanvragen plots zal uitsterven als gevolg van de uitzending van het laatavondjournaal. De demandcurve van het 7 uur journaal zou je dan kunnen beschrijven als een eerder bolle dalende curve die in een eerste fase eerder lineair daalt om dan in een tweede fase exponentieel te dalen. Een ander programma met een geheel andere curve zou een film uit de elektronische videotheek kunnen zijn. Hier zou een mogelijke curve kunnen omschreven worden als een lang uitgerokken, platte en heel langzaam dalende curve, met hier en daar zelfs een zwakke tijdelijke stijging op de tijdstippen wanneer er het meeste vraag is naar films. In beide gevallen is de algemene trend van de curve exponentieel dalend.

Het is natuurlijk onmogelijk om te voorspellen hoe de demandcurve van een programma eruit zal zien. Toch is het van belang voor een provider dat hij zo goed mogelijk kan inschatten hoe een typische demandcurve op zijn netwerk eruit ziet. De vorm van de curve zal namelijk in sterke mate een *invloed uitoefenen op de werking van de caching algoritmes*. Een provider die ernaar streeft om het nut van deze caches zo veel mogelijk te maximaliseren zal met andere woorden goed in staat moeten zijn om te voorspellen welke de populariteit van de programmas zal zijn en hoe die populariteit zal evolueren in de tijd. Het zal hem vooraf helpen inschatten in welke mate de caches efficiënt zullen zijn en dus opnieuw ingesteld moeten worden.

Toch is het mogelijk om een min of meer typische demandcurve te definiëren. Als in het vervolg van deze scriptie simulaties zullen worden gedaan dan zal meestal een dergelijke demandcurve gebruikt worden. Figuur 2.1 is een voorbeeld van een typische demandcurve van een programma van een half uur. Men kan zien dat de curve haar maximum bereikt in minuut 0 en daarna een exponentiële daling kent naar gelang er meer vertraging is ten opzichte van de live uitzending. Dit betekent: de meeste kijkers kijken live en hoe meer tijd er verstreken is sinds de aanvang van het programma hoe minder nieuwe kijkers er bij komen. Dit blijft duren tot op het moment dat er helemaal geen nieuwe kijkers meer bij komen.

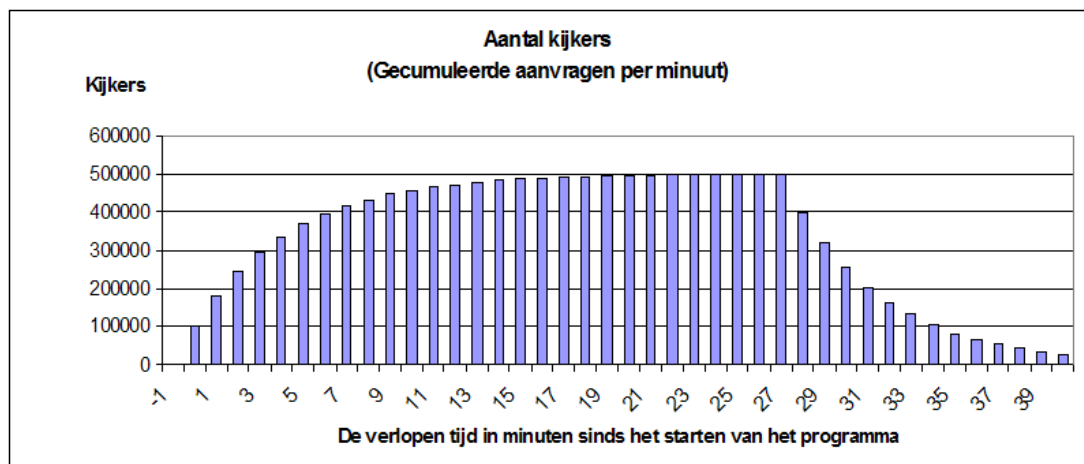
De reden dat die demandcurve van zo'n belang is, is omdat, als men de demandcurve van een programma kent, dan heeft men een idee hoeveel kijkers een bepaald programma zal hebben in de loop van de tijd. En dus ook, welk percentage van alle streams die over het netwerk lopen, over dat specifiek programma gaan. En het is dan weer de verdeling van deze streams



Figuur 2.1: Een typische demandcurve

(Bijvoorbeeld: 80% van alle lopende streams bevatten de beelden uit dezelfde 10 minuten van het programma) die bepaalt in welke mate een geïmplementeerd caching algoritme effectief zal zijn.

Een voorbeeld van een curve waarop men het aantal kijkers kan aflezen die hoort bij de curve uit het vorige voorbeeld, op voorwaarde dat iedereen die begint te kijken naar het programma blijft kijken tot het einde, kan gevonden worden op figuur 2.2. Er is een logaritmische stijging te zien bij het totaal aantal kijkers tijdens de duur van het programma en een exponentiële daling na afloop van het programma, van zodra de eerste kijkers het programma uitgekeken hebben.



Figuur 2.2: Kijkers curve afgeleid van de demandcurve

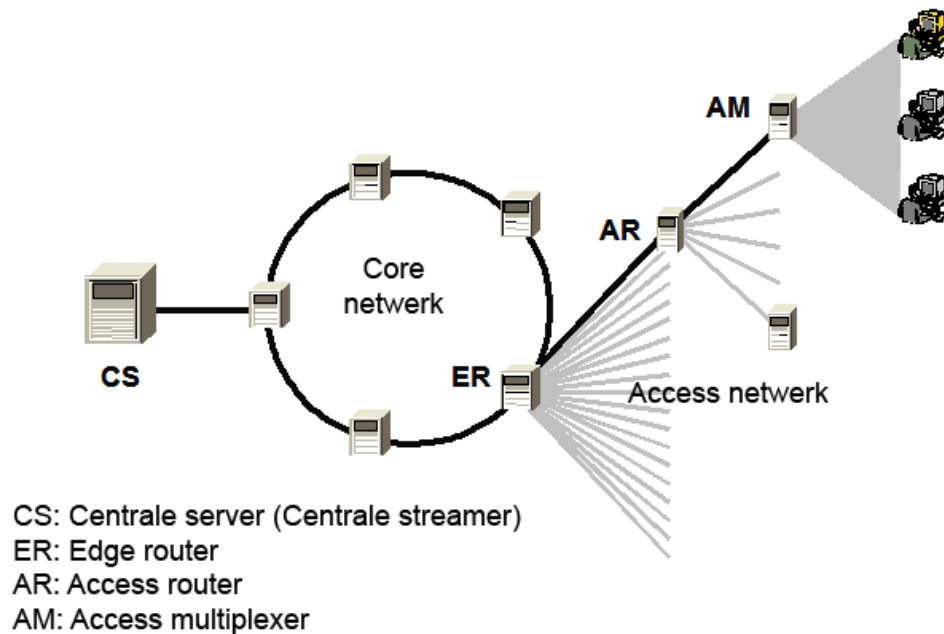
Samengevat: Als een provider op de meest efficiënte manier wil cachen dan heeft hij inzicht nodig in de kijkercurven van zijn toekomstige content. Een kijkercurve wordt bepaald door 3 variabelen. De lengte van het programma, het aantal kijkers bij aanvang van het programma en de vorm van de demandcurve. Het is duidelijk dat, voor aanvang van een programma, een provider enkel de eerste hiervan met zekerheid kan zeggen. De overige twee zijn afhankelijk van zoveel externe factoren dat geen enkele provider ze met voldoende zekerheid zal kunnen voorspellen. Ook hierdoor zal een demandcurve in de realiteit ook nooit een mooie exponentieel dalende curve zijn. Ze zal al dan niet in grote mate beïnvloed worden door tijdelijke externe factoren die de curve vervormen. De globale trend van de curve zal wel steeds exponentieel dalend zijn, daarom worden ook steeds curves van die vorm genomen bij de simulaties in het kader van deze scriptie.

2.2 Opbouw van een access netwerk

In de realiteit zal het toegangsnetwerk van een videostreaming provider zelden alleen gebruikt worden om video over te streamen. Meestal zal men de bestaande architectuur van een internet service provider gebruiken om over te streamen. Dit zorgt er dan ook voor dat een toegangsnetwerk maar weinig componenten zal bevatten die uitsluitend worden gebruikt om de streaming mogelijk te maken en dat het toegangsnetwerk voor het grootste deel uit de gewone standaard componenten bestaat waar het grootste deel van de rest van het internet is uit opgebouwd. Routers, switches, bridges, modems bij de clients thuis en hier en daar een firewall en een gewone proxy server. Natuurlijk zijn al deze componenten verbonden via links, dikwijls gebruikmakend van uiteenlopende technologieën en snelheden.

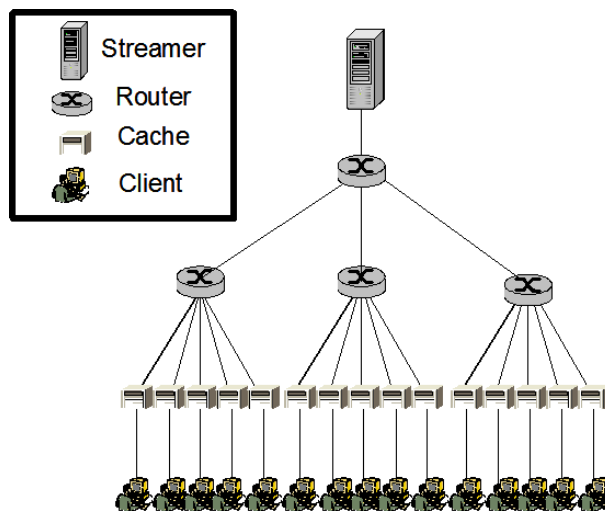
Een toegangsnetwerk voor videostreaming onderscheidt zich van een gewoon toegangsnetwerk voor internet door minimaal twee componenten. Ten eerste is er zeker één RTSP server, deze staat in voor de introductie en verspreiding van de video in het netwerk. Men moet beseffen dat een provider van redelijke grootte een zodanige bandbreedte te verduren krijgt dat in de praktijk één RTSP server op zichzelf nooit zal volstaan om alle aanvragen af te handelen. In de praktijk zal men een core netwerk bouwen, bestaande uit een mesh of een ring met daarin meerdere RTSP streamers, of één RTSP streamer en meerdere RTSP caches. Dit laatste voorbeeld van een core netwerk is door zijn aanwezigheid van caches, een interessante topologie om simulaties

over uit te voeren. Figuur 2.3 geeft een sterk vereenvoudigd voorbeeld van een toegangsnetwerk dat gebruik maakt van een dergelijke ring. Men zal de caches in de ring dikwijls de naam “Edge Server” of “Edge Router” geven.



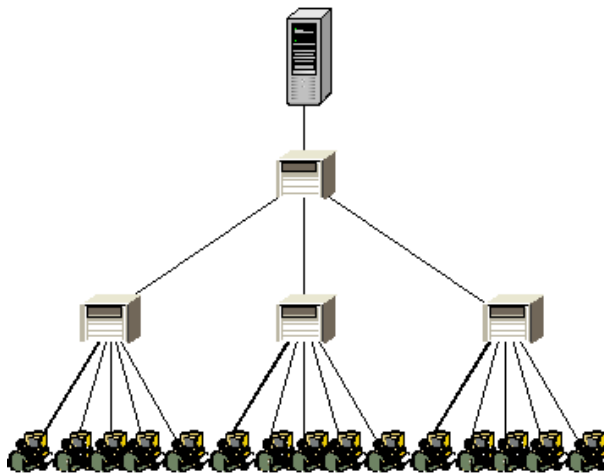
Figuur 2.3: Een transportnetwerk

Een tweede component die aanwezig moet zijn in het toegangsnetwerk is een client met de nodige functionaliteiten. Zo'n client kan meerdere vormen aannemen, zo kan een client een gewone desktop PC zijn met de nodige software, maar meestal zal de client bestaan in de vorm van een *Settopbox*. Een settopbox is een onderdeel van het toegangsnetwerk in de zin dat het de bron is van alle aanvragen die RTSP caches en RTSP servers zullen krijgen en het steeds de eindbestemming is van de streams over het netwerk. Het is niet de bedoeling dat deze thesis dieper in gaat op de architectuur of werking van zo'n settopbox maar er is één geval waarin de inhoud van zo'n box wel relevant is in het kader van deze scriptie, namelijk wanneer een settopbox zelf ook caching functionaliteiten kan aanbieden. Wanneer tijdens dit onderzoek simulaties worden gedaan om de prestaties van dergelijke caches te onderzoeken dan zullen we dit in de simulatie voorstellen als een proxy die tussen het toegangsnetwerk en de client staat en die verbonden is met de client via een link met oneindige capaciteit en kost nul. Dit wordt geïllustreerd door Figuur 2.4



Figuur 2.4: Settopboxes met caching capaciteiten

Tussen de streamer(s) en vele clients ligt dan het toegangsnetwerk, meestal een boom-vormig netwerk bestaande uit gewone internet componenten zoals routers. Wanneer men deze routers met caching functionaliteiten voorziet wordt dit opnieuw interessant in het kader van deze scriptie. Figuur 2.5 toont een boom-vormig toegangsnetwerk met caching functionaliteiten op alle tussenliggende routers.

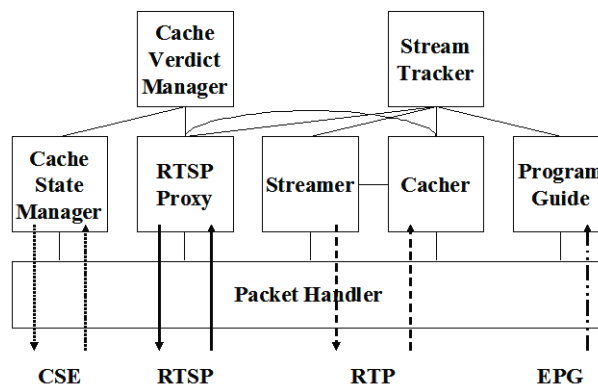


Figuur 2.5: Een boom met caches in tussenliggende nodes

2.3 De werking van een proxy server

2.3.1 Algemeen

De proxy server applicatie kan dus geïnstalleerd worden op zowel een router als op de settopbox. Natuurlijk zullen, door de verschillen in technologie en locatie binnenin het netwerk, er verschillen optreden bij de implementatiedetails en architectuur van beide applicaties. Toch zullen beide applicaties gelijk zijn op functioneel vlak. Zo zullen beide dezelfde caching algoritmes implementeren, beide dezelfde beslissingen moeten nemen op gebied van samenwerking. In dit deel wordt de structuur van dergelijke applicatie besproken.

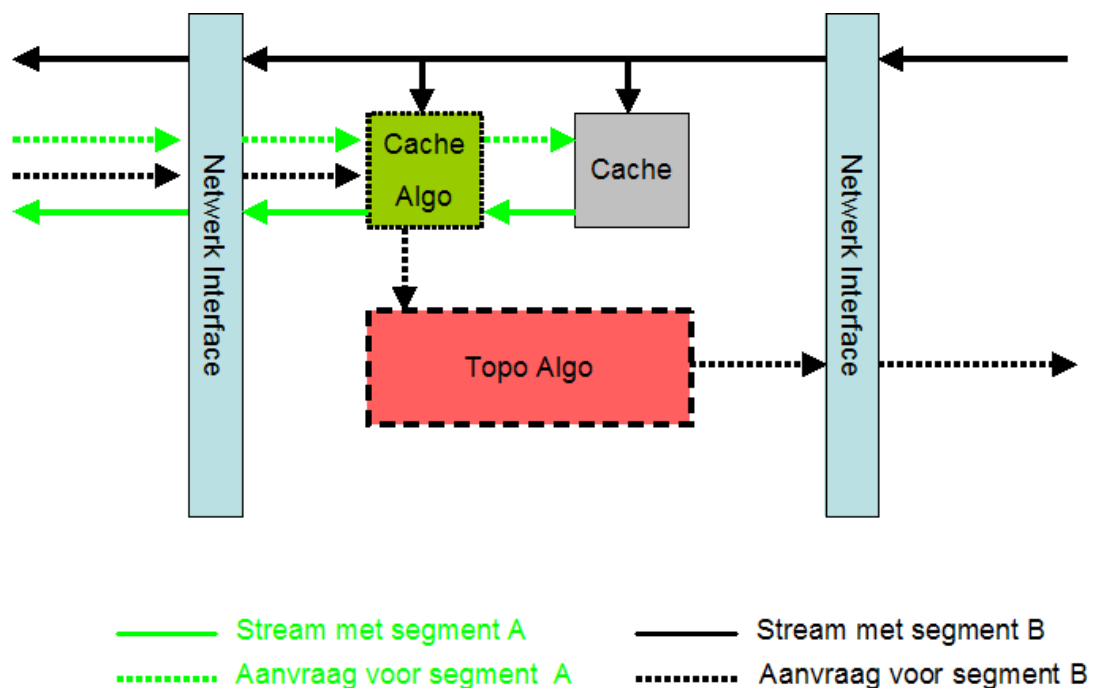


Figuur 2.6: Een mogelijke implementatie van een RTSP cache zoals die geïmplementeerd is binnen de onderzoeksgroep

Figuur 2.6 is een tekening van de architectuur van een RTSP Proxy server zoals die beschreven staat in [4]. Van alle onderdelen zijn “Cache Verdict Manager” en “Cacher” de componenten die ons het meest interesseren. De Cache Verdict Manager is verantwoordelijk voor de beslissing of een bepaalde stream gecached wordt. Met andere woorden, de cache verdict manager zal voor het grootste deel het caching algoritme implementeren en de cacher zal dan verantwoordelijk zijn van de details van het cachen zelf.

De andere componenten die te zien zijn op de tekening zijn verantwoordelijk voor de technische ondersteuning van de communicatie met de andere netwerk componenten, een studie van hun werking zou ons te ver leiden. Vanuit het standpunt van deze scriptie kan de proxy herleid worden tot 3 componenten met elk een duidelijk omschreven taak: één daarvan is de “Cache”, de andere twee kunnen gezien worden als onderdelen van de Cache Verdict Manager, namelijk het Cache Algo en het Topo Algo. Het **Cache Algo**, is verantwoordelijk voor de caching

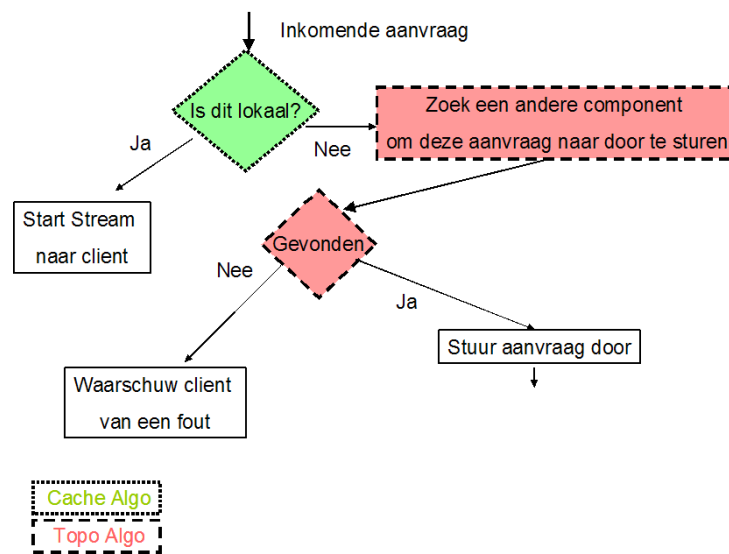
beslissingen. Het beantwoordt vragen zoals: *Wordt een bepaalde stream gecached? Wanneer moet een segment van een bepaalde stream uit de cache verwijderd worden?* Het **Topo Algo**, is verantwoordelijk voor het doorsturen van aanvragen. Afhankelijk van door welk algoritme de Topo Algo component precies geïmplementeerd is zal deze kiezen naar welke machine binnenin het netwerk een aanvraag die niet lokaal kon worden gevonden, zal worden doorgestuurd. Dit algoritme kan heel eenvoudig zijn, waar iedere aanvraag wordt doorgestuurd naar dezelfde streamer of dit algoritme kan bijzonder ingewikkeld zijn waarbij de Topo Algo's van verschillende Proxy Servers elkaar op de hoogte houden van aspecten zoals hun belasting, de belasting van de links en de verzadiging van hun Cache, en waar het Topo Algoritme beslissingen zal nemen rekening houdende met al deze aspecten.



Figuur 2.7: De logische architectuur van een RTSP Proxy Server

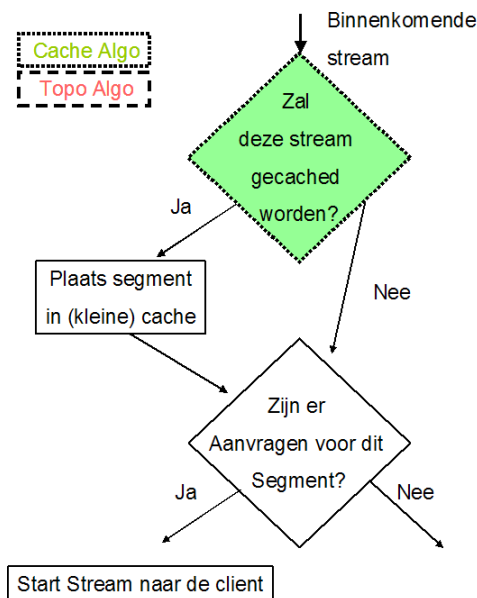
Op de tekening van de logische architectuur van de RTSP Proxy Server staan twee streams aangeduid. Segment A is er één die reeds lokaal in de cache zat. Segment B was nog niet lokaal bij moment van de aanvraag en de aanvraag moest dus doorgestuurd worden.

Om de werking van de RTSP Proxy server nog meer duidelijk te maken kan men de beslissingsboom ervan bekijken. Men onderscheidt 3 bomen, één voor elke externe stimulus die een proxy

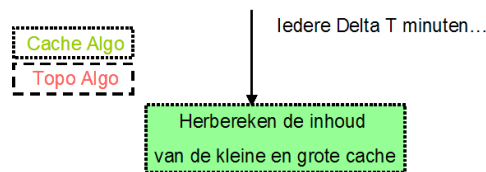


Figuur 2.8: Beslissingsboom 1 van een RTSP Proxy Server

server kan krijgen. Die stimuli zijn: “Een binnenkomende aanvraag”, “Een stream afkomstig van een andere proxy of een streamer wordt opgestart” en “De verplichte wachttijd (ΔT) is voorbij”. Deze laatste zal niet bij iedere implementatie van het caching algoritme een gevolg krijgen. In de bomen is een kleurencodering aangebracht als in de figuur met de logische architectuur.



Figuur 2.9: Beslissingsboom 2 van een RTSP Proxy Server



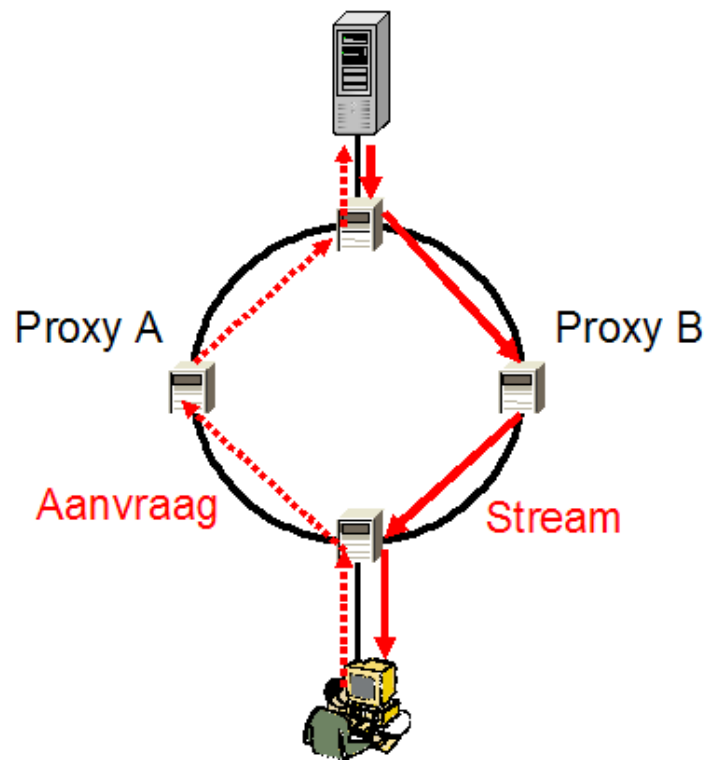
Figuur 2.10: Beslissingsboom 3 van een RTSP Proxy Server

2.3.2 Enkele ontwerpsbeslissingen

Betreft de werking van de RTSP proxy server en het toegangsnetwerk in het algemeen, zullen een drietal ontwerpsbeslissingen besproken worden omdat deze invloed zullen hebben op de prestaties van de proxy server.

Ten eerste, het binnenkomen of voorbij komen van een stream is niet noodzakelijk een gevolg van een binnenkomend (al dan niet onderschept en verder gestuurd) pakket. Het is duidelijk te zien in voorgaande beslissingsbomen dat we het binnenkomen van een aanvraag en het binnenkomen van een stream als twee volledig onafhankelijke stimuli zien. In de meeste eenvoudige topologieën zal het doorsturen van een aanvraag altijd een stream tot gevolg hebben, maar in een ring-topologie zal dit niet altijd zo zijn. Figuur 2.11 is een voorbeeld van dergelijke topologie. Het is mogelijk dat de aanvraag van de client onderschept en verder gestuurd wordt door proxy A, maar dat deze geen corresponderende stream ziet voorbij komen omdat deze via Proxy B komt. In deze situatie zal proxy B toch in staat zijn om de stream te captureren en te cachen op voorwaarde dat die proxy een feature ondersteunt, genaamd “Promiscuous mode”. Hier zullen de interfaces van de router waarop de proxy server is geïnstalleerd op zodanige manier geconfigureerd worden dat pakketten afkomstig van streams waarvan deze proxy noch de zender noch de ontvanger is, gecaptured kunnen worden.

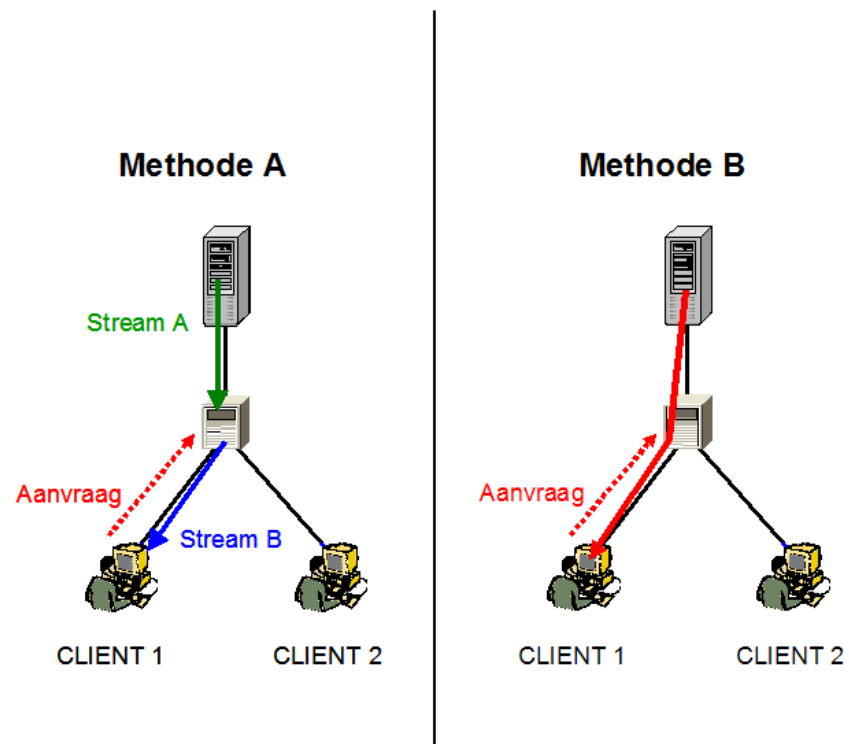
De proxy servers in het toegangsnetwerk zijn allemaal transparant voor de streamer en de clients. Dit wil zeggen, zowel de clients als de streamers hoeven niet van het bestaan van de proxy servers af te weten om te kunnen functioneren. Een client zal altijd al zijn vragen direct naar de streamer sturen. Als er proxy aanwezig is op het pad tussen de client en de streamer dan kan deze ervoor kiezen om die aanvraag te onderscheppen en opnieuw verder te sturen. Als de streamer een aanvraag krijgt dan zal die altijd veronderstellen dat die aanvraag afkomstig is van een client, zelfs al is die aanvraag in werkelijkheid afkomstig van een proxy. De streamer zal dan ook altijd een stream opstarten met als bestemming de oorsprong van de aanvraag, denkende



Figuur 2.11: De stream neemt een andere weg als de aanvraag

dat de oorsprong altijd een client is. Als bij de client een stream toekomt dan zal die altijd veronderstellen dat die stream van de streamer komt. Geen enkele component hoeft dus van het bestaan van de proxy servers af te weten.

Als een proxy server een bepaalde aanvraag onderschept en daarna opnieuw doorstuurt naar de streamer dan kan hij dat op twee verschillende manieren doen: Hij kan de streamer een aanvraag sturen met zichzelf als verzender (methode A), of hij kan de aanvraag sturen met de client als verzender (methode B). In beide gevallen zal de streamer een stream opstarten met die verzender als bestemming. Enkel bij methode A zal de proxy er absoluut zeker van kunnen zijn dat de stream voorbij hem passeert. Hij heeft dan wel nog zelf de verantwoordelijkheid om een nieuwe stream op te starten naar de client. Bij gebruik van methode B kan een situatie zoals in figuur 2.11 zich voordoen. Meer zelfs, bij een proxy die enkel gebruik maakt van methode B kan het ongewenste effect zich voordoen dat een bepaalde proxy grote hoeveelheden aanvragen krijgt voor een bepaald segment, al deze vragen doorstuurt omdat hij dat segment niet lokaal heeft maar nooit die stream zelf ziet passeren en dus ook nooit de kans zal krijgen om het te cachen. Dit is dan ook de reden waarom er gekozen is om methode A te implementeren in zowel



Figuur 2.12: Twee streams vs. Eén stream, één aanvraag

de bestaande RTSP proxy server als in de simulator.

Hoofdstuk 3

De TsTV Simulator

Nu we het probleem gesitueerd hebben en commentaar gegeven hebben in verband met enkele aspecten omtrent videocaching, staat er ons nog één ding te doen voor we beginnen met de eigenlijke caching algoritmes en hun werking. Dit is een korte bespreking maken omtrent de werking van de simulator. Het is nodig dat de lezer kennis bezit van de capaciteiten van de simulator en vooral waar de simulator niet tot in staat is.

3.1 Waarom een simulator?

In het vorige hoofdstuk is gezegd hoe een echt accessnetwerk eruit kan zien, een boom of een meer gecompliceerde structuur zoals een ring waar verschillende bomen rond zitten. Op elk van de tussenliggende nodes kan dan in theorie een proxy server geïnstalleerd worden. Het is niet nodig om zelf zo'n uitgebreid netwerk te bezitten om nuttige simulaties te kunnen doen. Maar het is eveneens niet voldoende om een simulatie uit te voeren met één client, één streamer en twee tussenliggende nodes met caching capaciteiten. Als men echt nuttige conclusies wil kunnen trekken uit een simulatie moet men een testopstelling hebben met meerdere clients en een tussennetwerk dat op zijn minst een vereenvoudigde voorstelling van een echt accessnetwerk is. Als men een boom gebruikt als testopstelling kan men pas goed zien of een bepaalde samenwerkingsvorm resultaat heeft als die boom minstens diepte 2 of 3 heeft.

Een tweede reden waarom geopteerd werd voor het werken met een simulator is omwille van het gemak van werken dat een simulator met zich mee brengt. Een simulator biedt het voordeel dat er bij het ontwikkelen van de algoritmes niet steeds geïnstalleerd moet worden op de nodes, dat

er op heel eenvoudige manier kan worden geswitched tussen de verschillende opstellingen en dat de simulatie gegevens heel makkelijk kunnen worden berekend en verzameld.

De derde en laatste reden waarom in dit geval geopteerd werd om een simulator te schrijven is de eenvoud qua implementatie van de verschillende algoritmes. Als men binnen een simulatoromgeving programmeert hoeft men zich niet steeds bezig te houden met de kleine details, iets wat het programmeren van een proxy server wel met zich meedraagt. Men kan zich meer bezig houden met de ontwikkeling en implementatie van de eigenlijke algoritmes. Het laat toe om op een eenvoudige wijze veranderingen toe te brengen aan de werking van de proxy server die indien men ze zou willen aanbrengen aan de echte proxy server dagen of weken programmeerwerk zouden vragen. Bijvoorbeeld: Het liet mij toe om de proxyservers te laten samenwerken en met elkaar te communiceren zonder mij zorgen te moeten maken over de protocollen die deze service zouden verzorgen.

3.2 Werking van de simulator

Het is niet de bedoeling om in dit punt te bespreken hoe men met de Simulator werkt, hoe hij gestart wordt of hoe de resultaten eruit zien die door de simulator geproduceerd worden. Dit wil niet zeggen dat deze informatie niet beschikbaar werd gesteld aan de lezer, maar hiervoor moet verwezen worden naar bijlage A. Het is dan weer wel de bedoeling om van dit hoofdstuk gebruik te maken om de algemene werking van de simulator voor te stellen met aandacht voor enkele specifieke kenmerken van de simulator die van belang zullen zijn tijdens het onderzoek in het vervolg van deze scriptie.

3.2.1 Werking

Net na het starten van de simulator is er een fase waarin de inputbestanden worden ingelezen, de informatie in die bestanden wordt gebruikt om in het geheugen van de simulator het beschreven netwerk te creëren. Volgens de simulator bestaat een netwerk steeds uit de volgende componenten: *Proxy, Server, Client, Node & Link*.

Onder *Proxy* verstaan we een component met caching capaciteiten, waar die zich ook mag be-

vinden binnen het netwerk. Het is eerder een gewone router binnenin het netwerk waarop de software van een RTSP proxy server is geïnstalleerd die de te simuleren algoritmes ondersteunt. Een *Server* is de gebruikte benaming van de streamer binnen het netwerk. Er is minstens één *Server* aanwezig binnen het te simuleren netwerk. Een streamer kan ook bekeken worden als een *Proxy component* die vanaf het begin van de simulatie steeds alles programma's lokaal heeft op een oneindig grote harde schijf. Een *Client*-component binnenin de simulator staat symbool voor één of meerdere clients uit een realistisch netwerk. Men kan een client-component het beste bekijken als een groepering van klanten. (Bijvoorbeeld een groep van abonnees van de provider die in dezelfde straat of buurt wonen en via hetzelfde toegangspunt op het netwerk aangesloten zijn.) Een *Node* is een component die twee of meerdere links van het netwerk verbindt maar zelf geen caching capaciteiten bezit. Een *Node* zou een router, een switch of een bridge kunnen zijn. Een *Link*-component is niet meer en niet minder als een link in een gewoon netwerk, een eenvoudige verbinding tussen componenten waarover data loopt. Van iedere link is de capaciteit en een gewicht bekend.

Ook in deze fase worden de kijkersaantallen berekend voor alle clients in het netwerk voor de hele simulatie. Meer specifiek, de simulator zal ieder programma opdelen in "*Segmenten*" van één minuut in lengte. Daarna zal de simulator, gebruik makende van het aantal beginnende kijkers van een programma en de demandcurve van dat programma, per minuut, berekenen hoeveel kijkers ieder segment heeft, voor de hele duur van de simulatie. (Hier wordt eigenlijk de kijkers curve zoals besproken in Hoofdstuk 2 berekend.) De vorm van de demandcurve wordt op een speciale manier gespecificeerd in de inputfiles. Dit gebeurt gebruik makende van één enkele variabele (C). Men berekent de demandcurve van een bepaald programma voor een bepaalde client als volgt: Eerst berekent men het product van die variabele en de lengte van dat programma, de betekenis van deze waarde het aantal minuten dat verlopen is sinds het starten van het programma op het moment dat er geen nieuwe kijkers meer bij komen. Met andere woorden, de x waarvoor de demandcurve een functiewaarde krijgt kleiner dan 1.

Men kan de formule die dergelijke exponentiele curve beschrijft als volgt afleiden:

Iedere exponentiële curve is van de volgende vorm:

$$E(x) = Ag^x$$

A is in dit geval het aantal kijkers dat het programma heeft in het begin. g is de waarde die we moeten berekenen. Als we de lengte van het programma definiëren als l dan zoeken wij een waarde voor g zodanig dat $E(x)$ gelijk wordt aan 1 bij een x-waarde van $l * C$:

$$1 = Ag^{lC} \Rightarrow \frac{1}{A} = g^{lC} \Rightarrow g = \sqrt[lC]{\frac{1}{A}}$$

Tijdens de simulaties zal meestal de waarde 1 gebruikt worden voor C.

Nadat de voorbereidingen getroffen zijn kan de tijd beginnen lopen en kan de eigenlijke simulatie beginnen, minuut per minuut. In het begin van iedere minuut zullen alle clients één voor één maar in willekeurige volgorde al hun *aanvragen doen*. Zo'n aanvraag gebeurt als volgt: Iedere client heeft een "bron" waar hij een aanvraag voor het nodige segment naar stuurt. Is die bron een streamer dan zal die streamer gewoon dat segment naar de client streamen, is die bron een proxy server dan is de zaak een stuk ingewikkelder.

Een Proxy heeft twee algoritmes die hij aanspreekt om de beslissingen die hij moet nemen voor hem te maken, namelijk een "CacheAlgo" en een "TopoAlgo". Een CacheAlgo staat in voor de beslissingen omtrent de interne caching beslissingen van de proxy. *Staat een bepaald segment lokaal? Zullen we dit segment cachen? Zullen we dit segment uit de cache verwijderen?* Elk van deze vragen wordt beantwoord door de instantie van CacheAlgo dat gebruikt wordt door de Proxy. Een TopoAlgo is verantwoordelijk voor de beslissingen die moeten genomen worden indien een bepaalde aanvraag niet lokaal kan geleverd worden. In dit geval moet de proxy beslissen waar hij dat segment zal zoeken. Ofwel kan hij een aanvraag doen aan een Streamer, ofwel kan hij de aanvraag doorsturen naar een andere Proxy waarvan hij weet dat die dit segment wel lokaal heeft. Ofwel kan hij een aanvraag doorsturen naar een andere Proxy die dit segment ook niet lokaal heeft met de vraag of die dat segment wil cachen.

Een bepaalde proxy server had net een aanvraag gekregen voor een bepaald segment. Op dit moment zal de proxy server zijn instantie van een CacheAlgo aanspreken om te vragen of hij dit segment lokaal heeft. Indien dit zo is antwoordt hij aan de aanvrager door dit segment te streamen. Indien dit niet zo is dan zal de proxy zijn instantie van een TopoAlgo aanspreken om te weten te komen naar wie hij deze aanvraag zal doorsturen? Hij zal dan de aanvraag doorsturen

naar die component. Als hij dan ook nog eens een positief antwoord krijgt van die component dan kan hij positief antwoorden aan zijn eigen aanvrager door dat segment te streamen naar de aanvrager. In dit geval komt bij deze Proxy zelf ook een stream toe, namelijk deze die gestuurd werd door de component aan wie hij de aanvraag had doorgestuurd. Bij het binnenkomen van deze stream wordt dit gemeld aan de instantie van CacheAlgo die op zijn beurt zal beslissen of dit segment lokaal zal gecached worden.

Elk van de componenten en elk van de links in het netwerk hebben logboeken. Elke component zal indien hij een actie onderneemt dit neerschrijven in zijn logboek indien hij dit interessant genoeg acht. Nadat de simulatie is afgelopen kunnen op deze manier de output bestanden geproduceerd worden door voor elk van deze componenten het logboek te lezen en te analyseren. Wat er precies in deze output bestanden geschreven staat kan gedetailleerd terug gevonden worden in Bijlage A.

3.3 Gemaakte vereenvoudigingen

1. **Kleinste tijdseenheid is 1 minuut:** Om de rekenlast van de simulatie te verminderen is gekozen om de kleinste tijdseenheid waarin dingen gebeuren en beslissingen worden gemaakt gelijk aan 1 minuut te maken. Dit houdt onder andere in dat programma's steeds een geheel aantal minuten lang zijn. Als een algoritme een programma in segmenten onderverdeelt, dan moeten deze altijd groter of gelijk aan 1 minuut video zijn. Dit houdt in dat een algoritme bijvoorbeeld enkel beslissingen kan nemen omtrent de grootte van zijn caching intervallen op het einde van iedere minuut. De kleinste data eenheid waar de simulator mee rekent is hierdoor de hoeveelheid video van één programma gedurende één minuut. Deze vereenvoudiging zal leiden tot een resultaat dat *in kleine mate afwijkt van de realiteit*. (Langs de ene kant worden de resultaten op een positieve manier beïnvloed, dit komt doordat kleinere fluctuaties in de demandcurve genegeerd zullen worden en het caching algoritme meer zal reageren volgens de algemene trend van de demandcurve. Langs de andere kant worden de resultaten ook op een negatieve manier beïnvloed door deze vereenvoudiging, namelijk doordat de beslissingen die genomen worden door een cache minder optimaal worden naargelang de kleinste tijdseenheid groter wordt.)

2. Opzetten van streams, delays en allerhande overhead wordt niet gesimuleerd:

De simulator zal veronderstellen dat deze zaken helemaal niet nodig zijn en dus niet plaats vinden. Dit houdt in dat er absoluut geen verkeer is op het gesimuleerde netwerk behalve dat van de streams, en dan ook enkel de eigenlijke streams: er zijn pakketten van het RTP type maar niet van het RTSP type. Als in de tweede fase van het onderzoek samenwerking tussen de caches gesimuleerd wordt, dan zal alle communicatie over het netwerk om die samenwerking mogelijk te maken gewoon genegeerd worden. Deze vereenvoudiging zal bijdragen tot een resultaat dat *een fractie te optimistisch is*.

3. Aanvragen, cachen & streamen gebeuren ogenblikkelijk:

Als een client een aanvraag doet voor een bepaald programma bij een proxy server dan zal die stream datzelfde ogenblik reeds opgezet zijn. Als een proxy opmerkt dat hij een fragment van een programma niet lokaal bezit en beslist om dat fragment zelf bij de streamer te halen dan gebeurt dit opnieuw onmiddellijk. Er zijn dus geen transmission delays, geen response delays & geen wachlijnen binnen de simulator.

4. Iedereen die begint te kijken naar een bepaald programma kijkt tot het einde:

Bij het starten van de simulatie op het moment dat - gebruik makende van de gegeven demandcurve - berekend wordt hoeveel kijkers een bepaald programma heeft en hoe die kijkersaantallen evolueren, wordt bij die berekening geen rekening gehouden met de mogelijkheid dat kijkers soms niet tot het einde van dat programma kijken, of dat soms kijkers halverwege een uitzending van een programma zullen beginnen kijken ook al is er een TsTV service beschikbaar. Het is moeilijk te zeggen of deze vereenvoudiging de resultaten nu op een positieve of op een negatieve manier beïnvloedt. Het is niet zo zeer aan de simulator om te weten hoeveel kijkers er weg vallen voor ieder programma, het is aan de gebruiker van de simulator om de demandcurves op zodanige manier te construeren dat ze eventuele afhakers in rekening neemt.

5. Er gaan nooit pakketten verloren en alle links hebben een eindige capaciteit:

In de realiteit zullen er op een druk gebruikt netwerk steeds pakketten verloren gaan. Links tussen componenten hebben steeds een maximale capaciteit, op het moment dat er te veel streams over zo'n link lopen zullen bij alle streams vele pakketten verloren gaan en een aanzienlijk kwaliteitsverlies optreden. In de simulator gebeurt dit niet.

6. Een aanvraag gebeurt steeds voor een segment en niet voor een heel program-

ma: Dit is eerder een implementatie detail en niet per sé iets waar de gebruiker van op de hoogte moet zijn. Toch heeft het gevolgen voor de simulatieresultaten. In de simulator zal een bepaalde client berekenen per minuut hoeveel aanvragen hij moet doen voor ieder segment van ieder programma. Bij het begin van iedere minuut zal hij elk van deze aanvragen doen. Dit heeft als gevolg dat een caching algoritme heel weinig besef heeft van het aantal gebruikers maar wel besef heeft van het aantal lopende streams. Deze vereenvoudiging heeft tot gevolg dat een caching algoritme meer vrij is in zijn keuzes dan hij zou zijn moest deze vereenvoudiging niet gemaakt zijn. Hij kan tussen twee minuten in, zonder probleem segmenten uit zijn cache verwijderen zonder zich zorgen te moeten maken of er een lopende stream bestaat die dit segment in de toekomst wel nodig zal hebben. Met andere woorden de proxy kan meer optimale beslissingen nemen. *De resultaten zullen dus iets beter zijn dan in de realiteit.*

Hoofdstuk 4

Niet coöperatieve caching algoritmes

In dit hoofdstuk zal voornamelijk de werking van het caching algoritme van een individuele proxy server uit de doeken worden gedaan. Eerst zullen enkele principes waarop al deze algoritmes gebaseerd zijn overlopen worden. Daarna volgt een overzicht van elk van de bestudeerde algoritmes. We besluiten met een reeks van simulatie resultaten voor deze algoritmes en onze bevindingen hieromtrent.

4.1 Algemene principes waarop de algoritmes gebaseerd zijn

4.1.1 Het interval

In dit onderdeel worden enkele principes vernoemd waarop een typisch TsTV algoritme gebaseerd zal zijn. Geen enkel algoritme zal alle genoemde specificaties implementeren, elk van hen zal op zijn eigen manier afwijken en zelfs dan nog heeft de programmeur genoeg vrijheid om een algoritme aan te passen en te configureren.

De algemene werkwijze is diegene waarbij het algoritme voor iedere televisiezender een “*Interval*” zal definiëren. Een interval is een reeks segmenten van enkele minuten die op dat moment in de cache zitten. Als we over intervallen spreken in het verband van TsTV caching algoritme kan men dat interval zien als een abstractie van het eigenlijke caching proces. Bijvoorbeeld: een cache bevat een interval van minuut 5 tot 10 van programma A en een andere interval van minuut 20 tot 30 van programma B. Sommige specifieke algoritmes zullen intervallen definiëren ten opzichte van het begin van programmas. Andere algoritmes zullen een interval laten glijden

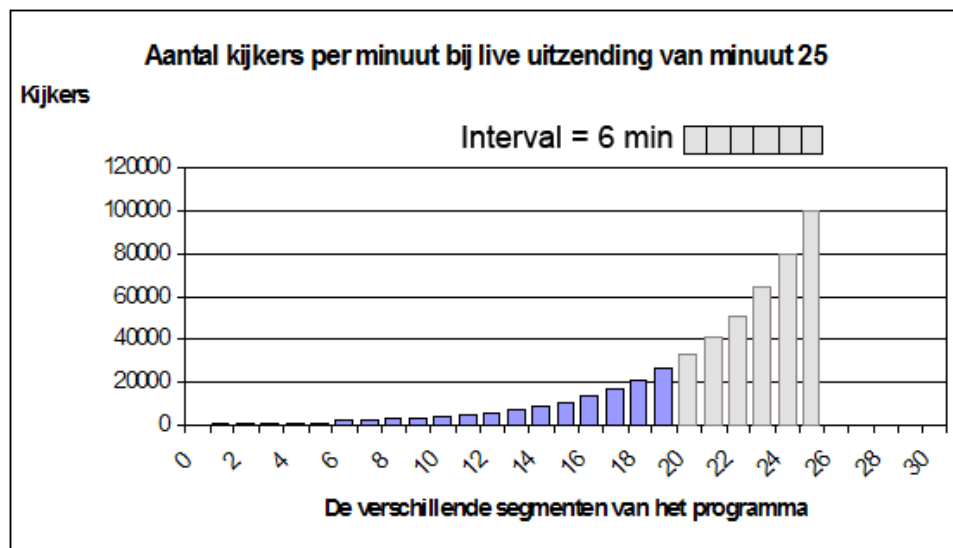
over verschillende programmas heen en maken dus abstractie van programmas. Ze denken enkel nog in termen van televisiezenders. Dergelijk algoritme moet zijn intervallen definiëren aan de hand van een vast begintijdstip.

Concreet zal een interval meestal data bevatten tussen een bepaald tijdstip in het verleden en nu, het moment van live-zijn. Maar opnieuw is er veel vrijheid op deze specificatie. Er zijn algoritmes waar een interval niet steeds op het moment van live-zijn moeten eindigen, maar ergens in het verleden kunnen eindigen. Nog andere algoritmes zullen meerdere niet aaneensluitende intervallen hebben voor dezelfde zender of zelfs voor hetzelfde programma.

Een interval kan uiteraard ook evolueren met de tijd. In deze context spreekt men van “*Glijden*”, “*Groeien*” en “*Krimpen*”. Op het moment dat een proxy pas gestart is en er dus nog voldoende plaats vrij is in zijn cache dan zullen al zijn intervallen groeien. In het begin is de grootte van het interval gelijk aan 0 minuten, nadat er een minuut is voorbij gegaan zal die data in de cache worden geplaatst, nog een minuut later zal ook deze data in de cache geplaatst worden. Het begintijdstip van het interval blijft gelijk en dus, hoe meer tijd er verstrijkt hoe groter het interval is. Op een bepaald moment zal de cache beslissen om dat interval niet meer te laten groeien. Deze beslissing kan een gevolg zijn van het feit dat de cache vol is of omdat het algoritme de voorkeur geeft aan een andere zender om zijn cacheruimte aan te spenderen. Op dit moment zal het interval beginnen te glijden. Dit wil zeggen, de nieuwste segmenten worden nog steeds in de cache geplaatst maar voor ieder gecached segment wordt het oudste verwijderd. Het blijft dus een aaneengesloten reeks van segmenten dat steeds de laatste N minuten van die zender bevat. Als laatste kan een algoritme er ook voor kiezen om een bepaalde interval te doen krimpen. Zo’n beslissing kan opnieuw het gevolg zijn van twee oorzaken. Als het een interval is over een programma maar niet over een zender en dat programma is afgelopen dan zal dit het krimpen van dat interval tot gevolg hebben. Een andere reden kan bijvoorbeeld zijn dat door dalende populariteit van de zender of door stijgende populariteit van de concurrerende zenders, het algoritme ervoor kiest om die zender met een bepaalde snelheid te doen krimpen. Bijvoorbeeld: Met het verstrijken van de tijd worden nog steeds de modernste segmenten in de cache geplaatst, maar voor iedere gecached nieuw segment worden er de twee oudste van dat interval verwijderd.

Er wordt gewerkt met intervallen en niet met een meer traditionele caching techniek zoals “least recently used” omwille van de relatieve eenvoud van werken met intervallen ten opzichte van de traditionele caching technieken. Deze laatste houden geen rekening met de glijdende structuur van een stream terwijl deze algoritmes daar juist gebruik van maken. In het bijzonder zijn er twee aspecten van TsTV die gebruikt worden door onze algoritmes om op een eenvoudige manier uitstekende resultaten te kunnen halen. Dit eerste aspect is de meest voor de hand liggende, de vorm van de demand curve.

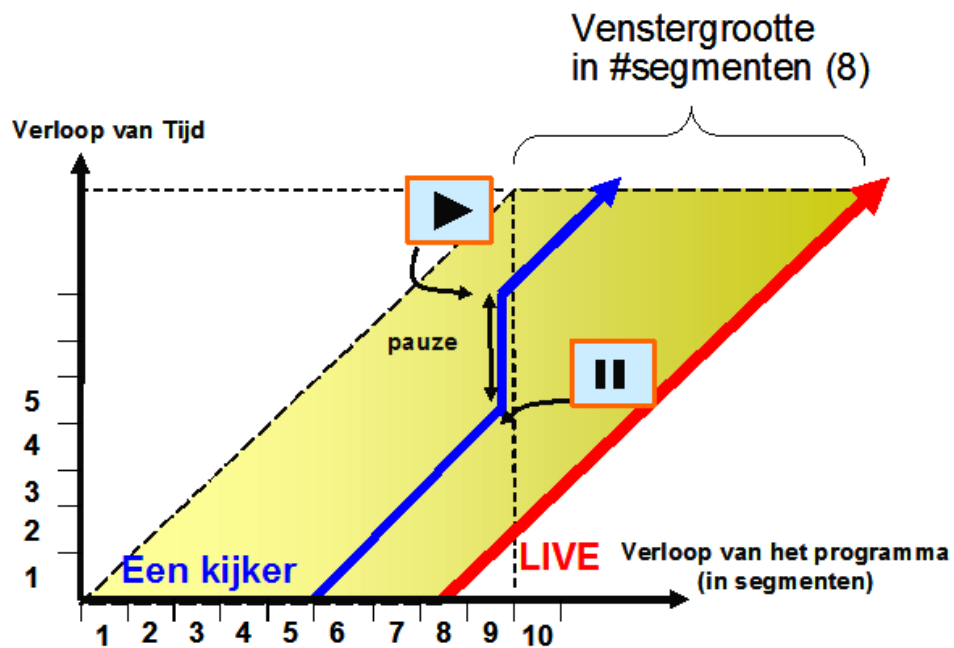
Het is al genoeg vermeld geweest dat de typische demandcurve een exponentieel dalende curve is. Deze vorm heeft als gevolg dat, als we voor een bepaald programma een dergelijke demand-curve optekenen, we direct zien dat het grootste volume ingesloten tussen de curve en de X-as zich net voor het moment van live-zijn bevindt. Door dus een venster te nemen dat schuift met de snelheid van de uitzending over die verdikking in de curve, kan men met gemak grote delen van het aantal aanvragen afhandelen door slechts een beperkt aantal segmenten in de cache te plaatsen.



Figuur 4.1: Demandcurve met interval

Figuur 4.1 is een gevolg van de demandcurve uit Hoofdstuk 2 (Figuur 2.1). Deze stelde de demand voor van een programma van 30 minuten waar 100,000 kijkers beginnen kijken bij aanvang van de uitzending. Deze figuur stelt voor hoeveel kijkers er naar de verschillende segmenten van het programma kijken op het moment dat men bezig is met live uitzending van de 25e minuut.

Stel nu dat een proxy wordt geplaatst die een glijdend interval met een constante grootte van 6 minuten hanteert. Op dat moment zullen alle lichtgrijs gekleurde aanvragen geleverd worden door die proxy en alle donker gekleurde door de streamer. Voor dit voorbeeld geeft dit een hit percentage voor de proxy van 74%. Terwijl op geen enkel moment meer dan 20% van het programma zich in de cache van die Proxy bevindt.



Figuur 4.2: Een gebruiker bevindt zich in het venster en pauzeert

Een andere reden waarom men grijpt naar algoritmes die intervallen hanteren is omwille van de extra functionaliteit die TsTV met zich mee brengt: het pauzeren, terug spoelen en verder spoelen van de programmas. Als algoritmes gebruikt worden die werken aan de hand van intervallen dan zullen voor bepaalde gebruikers deze acties gratis zijn, zolang deze acties binnen het huidige interval blijven. Een gebruiker die op dit moment naar een programma kijkt en zijn huidige aanvragen bevinden zich binnen het interval van de cache, dan zal hij afhankelijk van de grootte van dat interval in bepaalde mate kunnen pauzeren, verder en terug spoelen zonder uit dat interval te vallen. Op deze manier blijft zijn stroom door de proxy geleverd worden. Figuur 4.2 illustreert een gebruiker die zich binnenin het venster bevindt en pauzeert.

4.1.2 De S en L cache

In 4.1.1 werd de werking van het schuivende en groeiende interval beschreven. Ter vereenvoudiging werd verondersteld dat verschuiven en groeien steeds onmiddellijk gebeurde en dat alle intervallen steeds de optimale grootte hadden. In de praktijk is dit anders, omwille van voornamelijk technische redenen.

In de praktijk zullen de algoritmes hun cache onderverdelen in twee delen. Een deel **S** (Small) en een deel **L** (Large). S zal gebruikt worden om alle nieuwe (live) segmenten in op te slaan gedurende een bepaalde periode ΔT . Op het einde van iedere ΔT wordt de inhoud van L herberekend. L zal volledig opgevuld worden met een aantal intervallen bestaande uit segmenten die reeds in L zaten en nieuwe segmenten die nu in S zitten. Uiteindelijk wordt S leeg gemaakt, zodat die weer gevuld zou kunnen worden met nieuwe live segmenten. De manier waarop die herverdeling van L wordt gedaan is opnieuw afhankelijk van algoritme tot algoritme, de verschillen zullen onder andere gemaakt worden bij het toekennen van populariteit van segmenten en welke regels gehanteerd worden bij het herverdelen van de intervallen

S is het kleinste deel van beide. De grootte van S is vast en is makkelijk te berekenen, S is namelijk steeds de grootte van ΔT keer het aantal televisiezoekers in de simulatie, uitgedrukt in aantal minuten gecachte stream. De grootte van L zal bepaald worden door de specifieke hardware waarover de Proxy server beschikt, dus de totale grootte van de cache min S.

Het is duidelijk dat ΔT een belangrijke parameter zal worden van al deze algoritmen. Het bepaalt niet alleen de grootte van S maar het zal ook de parameter zijn die bepaalt hoe dikwijls de lengtes van de intervallen herberekend worden. Een kleine ΔT zal zorgen voor heel flexibele intervallen, maar ook veel rekenlast en intervallen die sterk beïnvloed worden door heel tijdelijke afwijkingen in de curves. Grote waarden voor ΔT zorgen voor beter afgewogen, maar tragere caching beslissingen. Een te grote ΔT zal zeer schadelijke gevolgen hebben voor de werking van de algoritmes naar mate deel L steeds kleiner wordt.

Zoals eerder vermeld groeien en glijden de intervallen niet geleidelijk met de uitzending mee. In de praktijk zullen de intervallen tijdens een ΔT periode altijd groeien (naar mate S dat oorspronkelijk leeg is, wordt opgevuld), op het einde van iedere ΔT zullen de intervallen sterk

groeien, of sterk dalen of gewoon gelijk blijven. De gemiddelde intervallengte zal wel steeds afnemen tussen iedere ΔT in, doordat S steeds volledig leeg gemaakt wordt. Hierna begint de cyclus weer opnieuw en groeien de intervallen opnieuw. Kiest men om ΔT gelijk te stellen aan 1, dan bekomt men wel de mooi schuivende vensters zoals eerst voorgesteld.

4.1.3 Populariteit

De exacte manier waarop beide delen van de cache herverdeeld worden en de lengte van de intervallen herberekend worden, wordt berekend in functie van een bepaalde populariteit die men de programmas of de segmenten waaruit dat programma bestaat moet toekennen. Bij het ontwerpen van een dergelijk algoritme moet men keuzes maken over hoe men deze populariteit zal definiëren, hoe men die zal verzamelen en op welke manier men die in verband brengt met de grootte van de intervallen.

De eerste van een reeks van keuzes die gemaakt moeten worden is of men de populariteit zal berekenen **per programma of per programma segment**. Beide methoden hebben hun voor- en nadelen, maar het belangrijkste argument is er één in het voordeel van per segment. Indien men per segment berekent dan heeft men meer uitgebreide informatie die eventueel zou kunnen leiden tot de beslissing om meerdere intervallen voor hetzelfde programma te nemen. (Zoals het Survival of the Fittest Algoritme doet.) In het andere geval zou dit niet lukken, daar zouden enkel intervallen mogelijk zijn die eindigen met het live segment. Een argument in het voordeel van berekenen per programma kan zijn dat het de implementatie van het algoritme zou vereenvoudigen.

Een volgende stap die men moet nemen is beslissen hoe men populariteit toekent aan de programmas of aan de segmenten. Er volgt kort een overzicht van enkele mogelijke werkwijzes. We beginnen met de meest eenvoudige en stoppen met de meest ingewikkelde methodes.

- Per aanvraag verhogen we de teller met 1
- Per aanvraag verhogen we de teller met het aantal hops naar de dichtstbijzijnde streamer
- Per aanvraag verhogen we de teller met de som van het aantal streams dat over alle links

loopt op weg naar de dichtstbijzijnde streamer

- Per aanvraag verhogen we de teller met het aantal hops naar de dichtstbijzijnde component die het segment lokaal heeft.
- Per aanvraag verhogen we de teller met de som van het aantal streams dat over alle links loopt op weg naar de dichtstbijzijnde component die het segment lokaal heeft.

De meest eenvoudige manier zou kunnen zijn om per programma of segment een teller bij te houden, en deze teller te **verhogen met 1** iedere keer er een aanvraag is. Deze methode is het makkelijkst te implementeren omdat ze geen extra informatie vereist in verband met die aanvraag, het feit dat er één is geweest is genoeg.

Het probleem van voorgaande methode is dat ze geen rekening houdt met de geografische ligging van de proxy ten opzichte van de dichtstbijzijnde streamer. Men zou bijvoorbeeld bij iedere aanvraag die teller kunnen verhogen met het aantal hops tussen de proxy en die streamer. Of men zou kunnen zoeken naar de dichtstbijzijnde component die dat segment wel heeft en het aantal hops daar naartoe kunnen berekenen. Deze manier van werken komt al redelijk dicht in het vaarwater van de samenwerkende proxies terecht.

Men kan op een nog meer gedetailleerde manier de populariteit berekenen. De vorige methode houdt wel rekening met de geografische ligging van de componenten, maar nog geen rekening met de belasting van de links tussen deze componenten. Stel nu dat onze proxy op ieder moment op de hoogte is van de belasting van iedere link in het netwerk. Alle proxies zijn op routers geïnstalleerd, dus een proxy moet in staat zijn om te schatten hoeveel streams er op dat moment over een bepaalde link lopen. Deze informatie zou verspreid kunnen worden tussen de proxy servers aan de hand van één of ander speciaal ontworpen protocol. Naar mate we ons in de loop van deze scriptie meer en meer zullen bezig houden met samenwerking tussen de proxy servers, zal de nood aan een dergelijk protocol steeds groter worden.

Eens men op de hoogte is van de belasting van alle links zou men voorgaande methode opnieuw kunnen uitbreiden. In plaats van de populariteit iedere keer te verhogen met het aantal hops, kan men de populariteit verhogen met de som van het aantal streams dat op dat moment over de links lopen op weg naar de dichtstbij zijnde streamer. (Of naar de dichtstbij zijnde proxy

die dat segment bevat, maar dat is dan weer samenwerking) Indien men niet in staat is om een aanneembare schatting te maken van het aantal streams die over een bepaalde link loopt kan men de som maken van andere factoren van de verschillende links. Een mogelijkheid is om de huidige doorvoer van die link (in kbit/s) te delen door de maximale capaciteit van die link, en dan al die coëfficiënten op te tellen. Er zijn nog vele andere mogelijkheden.

4.2 Specifieke algoritmes

4.2.1 Het Prefix algoritme

Het prefix algoritme is geen sliding interval algoritme, aangezien het interval niet verschuift of groeit. Het belangrijkste verschil tussen het prefix algoritme en de interval algoritmes zit in de situatie waarin men deze algoritmes gebruikt en het resultaat dat men probeert te bekomen. Terwijl de gewone interval algoritmes voornamelijk de bedoeling hebben om de belasting van het netwerk zoveel mogelijk te verminderen zal het prefix algoritme gebruikt worden in de hoop de setup tijden van de net opgezette streams te verminderen.

Het prefix algoritme werkt als volgt. Van ieder programma, of in ieder geval van enkele van de meest populaire programmas, wordt de eerste K minuten van dat programma gecached op een proxy server dichterbij de client. Er wordt enkel die K minuten per programma op de caches gezet. Als een bepaalde client een aanvraag doet om een bepaald programma te bekijken dan zullen enkel de eerste K minuten van dat programma zich in de lokale cache bevinden. Nadat die aanvraag gebeurt is zal de proxy zelf een aanvraag doen bij de streamer voor de overige minuten van dat programma. Vanaf het moment dat minuut K verstreken is en er zich dus niets meer van waarde voor die client in de cache bevindt dan zal de $K+1$ 'ste minuut verder gestreamed worden naar de client gebruik makende van de stream tussen de streamer en de proxy. Er zal dus wel een kleine buffer in het geheugen van de Proxy nodig zijn om heel tijdelijk enkele pakketten op te slaan om het tijdsverschil tussen beide streams te overbruggen. Deze pakketten worden echter niet persistent gemaakt door de cache.

De belangrijkste reden voor deze manier van werken is dat de setuptijd van een stream sterk verkort kan worden. Omdat de proxy dichterbij de clients staat is de "roundtriptime" tussen

de client en de proxy kleiner dan tussen de client en de streamer. Bij het opzetten van een stream via RTSP moet er redelijk wat heen en weer gestuurd worden tussen beide partijen. Een tweede reden waarom deze manier van werken de setuptijd vermindert is doordat een enkele Proxy steeds verantwoordelijk is voor een klein gedeelte van de totale populatie van alle clients. Hierdoor krijgt hij ook slechts een klein deel van alle aanvragen binnen en is zijn response tijd dus veel beter dan dat van de streamer.

Dit algoritme heeft natuurlijk ook een aantal nadelen. Het belangrijkste nadeel is dat het prefix algoritme amper iets doet om het verkeer op het netwerk en de belasting van de streamer te verminderen. Stel dat wij een netwerk hebben waar iedere client zijn aanvragen doet aan een proxy en iedere proxy server een prefix van 5% van ieder programma in zijn cache heeft. Als we veronderstellen dat alle kijkers naar onze programma's kijken van begin tot einde dan is de rekensom heel eenvoudig: 5% van alle verkeer over het netwerk zal geleverd worden door de verschillende Proxies. De overige 95% is afkomstig van de streamer. Het is duidelijk dat het algoritme op dit vlak heel zwak zal presteren.

Een Proxy die dit algoritme ondersteunt heeft bij voorkeur reeds alle prefixen van alle programma's in zijn cache. Als een client een aanvraag doet naar een programma waar op dat moment nog geen prefix van aanwezig is zal die aanvraag opnieuw een sterke delay kennen, die delay zal zelfs groter zijn dan als die client zijn aanvraag direct naar de streamer had gestuurd. (Want er moeten nu twee streams opgezet worden met twee keer vertraging tot gevolg) Het prefix algoritme doet in dat geval nog steeds zijn job voor de meeste aanvragen, maar zal enkele keren voor een onaanvaardbaar lange delay zorgen.

Dit algoritme is niet geschikt om te gebruiken binnen de context van het netwerk dat we in de vorige hoofdstukken hebben geschetst, een toegangsnetwerk van een Time-shifted television provider. Maar dat is natuurlijk niet de enige situatie waar content in de vorm van videostreams wordt aangeboden. Er zijn namelijk situaties waar men wel nut heeft aan dergelijk algoritme. Denk aan een situatie waar de content vast ligt, er een beperkt aantal programma's zijn, het aantal van die programma's verandert niet sterk over de tijd, waar het toegangsnetwerk beperkt is qua grootte en de opstartdelay van de streams relatief belangrijk is. Bijvoorbeeld het netwerk van een website die voornamelijk videostreams aanbiedt, een netwerk van een bibliotheek, een

museum, een hospitaal of een hotel.

In dergelijke gevallen biedt dit algoritme ons nog één voordeel dat we niet meteen hebben in een provider netwerk. Onderzoeken hebben namelijk uitgewezen dat bijna 90% van alle video-streams over het internet vroegtijdig beëindigd worden [3]. Als dergelijke stream dan vroeg genoeg beëindigd wordt en de prefixes zijn groot genoeg dan kan op deze manier toch nog een grote hoeveelheid van de belasting van de streamer vermeden worden.

4.2.2 Sliding Interval algoritme met vaste lengte

In tegenstelling tot het prefix algoritme is dit wel een interval algoritme. Iedere televisiezender heeft zo'n interval en dat interval blijft steeds gelijk qua grootte. Dit laatste is dan ook de enige manier waarop dit algoritme afwijkt ten opzichte van het voorgestelde standaard algoritme. Het interval blijft dus gelijk per televisiezender, de eerste minuten na het starten van een Proxy server zijn de enige minuten waar het interval van een televisiezender groeit, vanaf dan blijft het glijden, ongeacht het aantal aanvragen voor de programmas op die zender. De intervallen van verschillende zenders onderling moeten niet even groot zijn. In de praktijk zullen verschillende zenders verschillende populariteiten hebben, men bekomt de beste resultaten als men die populariteit laat weerspiegelen in de grootte van de intervallen.

Volgens deze manier van werken is er natuurlijk altijd het probleem van de variërende populariteit van de verschillende zenders. Stel nu dat er twee televisiezenders zijn die concurreren. De ene dag heeft zender 1 bijna alle kijkers, de andere dag is het zender 2. In deze situatie zouden er in minstens de helft van de tijd slechte resultaten geboekt worden. Natuurlijk kan de provider proberen om die populariteit te voorspellen en op regelmatige tijdstippen die venstergroottes opnieuw in te stellen maar dit is natuurlijk geen onderdeel van het algoritme en we kunnen het ook moeilijk een oplossing noemen. Een dergelijke situatie zal gelukkig niet zo veel voorkomen, toch niet in die mate. In de praktijk zullen de verschillende televisiezenders natuurlijk populariteiten hebben die sterk van elkaar afwijken maar fluctuaties in die populariteit zullen nooit van zodanige grootte zijn dat de populairste zender ineens de minst populaire wordt. De populariteiten zullen natuurlijk wel sterk wijzigen over langere periodes, maar niets houdt de provider tegen om, bijvoorbeeld iedere week de groottes van de intervallen aan te passen gebaseerd op de

populariteit van de zenders tijdens de afgelopen week.

Maar hiernaast biedt dit algoritme ook één heel groot voordeel, door deze manier van werken is er geen nood aan het onderscheid tussen het S en L deel, er is één grote cache. Algoritmes die wel met een S cache werken doen dat omdat er een soort buffer nodig is om alle data in op te slaan tijdens iedere periode ΔT , hierdoor is die buffer de helft van de tijd half leeg. Aan het begin van de periode is hij leeg, daarna wordt hij langzaam opgevuld tot hij volledig vol is. S is steeds de grootte van het aantal zenders maal ΔT . Bij providers die een groot aantal zenders aanbieden kan S dus een aanzienlijk deel innemen van de caches. Een gemiddelde Belgische provider biedt meer dan 60 zenders aan, combineer dit met een ΔT van 2 minuten en men krijgt een S cache van 120 minuten. Dit is 1,3 GiB die gemiddeld maar half gebruikt wordt.

Dit algoritme kan het gebruik van de S cache vermijden. Omdat de groottes van de intervals op voorhand gekend zijn kan de cache gewoon gevuld worden tot het window volledig gevuld is. Als op dat moment een nieuwe aanvraag binnen komt dan kan men gewoon het oudste fragment in dat interval zoeken, wissen en de inhoud van de binnenkomende stream daar schrijven. Op deze manier is na een tijdje de hele cache gevuld en blijft hij volledig gevuld. Een enorm voordeel dat in sommige situaties misschien het nadeel van dit algoritme kan overtreffen. *Een provider met veel content verdeeld over vele zenders wiens populariteit eerder voorspelbaar is kan dus voordeel halen uit dit algoritme.*

4.2.3 Sliding Interval algoritme met variabele lengte

Dit is het algoritme dat het minste afwijkt van de norm, het sliding interval algoritme met variabele intervallengte. Hier wordt net zoals het andere sliding interval algoritme in intervallen gecached maar met als belangrijkste verschil dat de intervallen hier kunnen groeien en krimpen. Dit is dus het eerste algoritme waar de populariteit van de verschillende segmenten van belang is.

Eerder in het hoofdstuk is reeds vermeld geworden dat er verschillende manieren zijn om de populariteit van programmas op te tekenen. Iemand die dit algoritme implementeert kan natuurlijk vrij kiezen op welke manier hij dit probleem oplost. Hier werd bij het implementeren van dit algoritme ervoor gekozen om de simulator te doen werken via de laatste besproken me-

thode: *Per aanvraag verhogen we de teller met de som van het aantal streams dat over alle links loopt op weg naar de dichtstbijzijnde component die het segment lokaal heeft.* Bovendien wordt populariteit bijgehouden per segment en niet per programma. Op deze manier heeft het algoritme de maximale hoeveelheid informatie om de juiste keuze te kunnen maken.

Gebruikmakende van de populariteit per segment worden de intervals op de volgende manier gebouwd: Iedere keer dat er een periode ΔT beëindigd is, zit cache S vol met segmenten en waarschijnlijk zitten in cache L ook redelijk wat segmenten. Voor elk van deze segmenten in beide caches is er een teller bijgehouden. Het algoritme om de intervals te bouwen werkt op de volgende manier:

Alle segmenten uit S en L worden verzameld in één verzameling, in iedere fase van het algoritme zal men alle groeiende segmenten zoeken. Een segment s noemen we een “groeiend segment” als:

- Indien de zender waartoe s behoort op dit moment nog geen interval heeft en s het live-segment was van de afgelopen minuut.
- Indien de zender waartoe s behoort op dit moment wel een interval heeft en het voorlopige vroegste segment van het interval van die zender het segment is dat vervolgt op s .

Het algoritme zal dus steeds alle groeiende segmenten zoeken, hetgene met de hoogste populariteit selecteren en een interval met dat segment uitbreiden. Deze stap zal herhaald worden totdat er geen segmenten meer zijn om uit te kiezen of omdat cache L vol zit.

Op deze manier wordt er dus voor iedere zender juist één interval geconstrueerd.

Dit algoritme zal dus tijdens de simulaties uitblinken in situaties waar flexibiliteit nodig is. In een situatie waar de verschillende zenders programma's hebben die achtereenvolgens heel populair en heel onpopulair zijn zal dit algoritme de beste resultaten boeken ten opzichte van de vorige algoritmes. Als op een bepaalde zender een onpopulair programma beëindigd is en een populair begint zal het kleine interval mooi over de zender heen schuiven en naar mate populariteit van de segmenten toe neemt zal dit algoritme beslissen om zijn interval voor deze zender te laten groeien. In dergelijke situaties is het dan ook het beste om de ΔT niet al te groot te nemen. Uiteraard moet ze groot genoeg zijn om de tijdelijke fluctuaties in de demandcurve te

overbruggen maar een te grote ΔT zal zorgen voor een te grote S cache en dat zorgt op zijn beurt voor een te kleine L cache om de intervallen in op te bouwen. Bovendien zal een te grote ΔT ervoor zorgen dat beslissingen omtrent de grootte van de intervallen te laat genomen worden.

4.2.4 Het survival of the fittest algoritme

Net zoals het prefix algoritme is dit geen interval algoritme. Toch is het helemaal niet zo verschillend van het Sliding Interval Algoritme met variabele interval lengtes. De enige manier waarop beiden verschillen is doordat Survival of the Fittest niet eist dat de segmenten in zijn cache opeenvolgend zijn. Hierbij mag elk segment in de cache zitten, onafhankelijk van welke andere segmenten van die zender ook aanwezig zijn in de cache. Natuurlijk is zo'n segment eigenlijk zelf al een mini-interval van 1 minuut in lengte en op die manier kan men het survival of the fittest algoritme *wel* als een interval algoritme zien, maar dan één dat meerdere intervallen per zender toe laat.

Bij het schrijven van de simulator was het onmogelijk om met pakketten te werken met een grootte van de orde van een KiB. Dit zou de simulator nodeloos ingewikkeld gemaakt hebben en het uitvoeren van een simulatie ontzettend lang. Er moest een vereenvoudiging gemaakt worden, de streams werden onderverdeeld in grotere blokken van elk 1 minuut data. We noemden deze blokken segmenten. Nadat deze vereenvoudiging werd gemaakt kwam een nieuwe vraag aan het licht: *Waarom kunnen caching algoritmes ook niet gewoon met segmenten van 1 minuut werken in plaats van met windows?*

Technisch zou dit haalbaar zijn. Bij het opstarten van een stream wordt telkens door de ontvangende proxy in zijn geheugen plaats voorzien van 1 minuut video. Tijdens het streamen wordt die buffer gevuld tot hij vol is. Daarna wordt de inhoud naar cache S gekopieerd en wordt een nieuwe buffer gevuld in het geheugen. Vanaf dan worden alle caching beslissingen genomen in de context van segmenten en niet meer in de context van programma's, zenders of intervallen. Dit zou bovendien heel wat problemen uit de wereld van de RTSP proxies herleiden tot problemen uit de wereld van de web proxies.

Concreet over de werking van het Survival of the fittest algoritme: De enige moment waarop

het gedrag van dit algoritme een verschil vertoont met dat van de Sliding Interval met variabele interval, is op het moment van herberekening van de L cache op het einde van iedere ΔT . Meer bepaald tijdens het in verband brengen van de populariteit van de verschillende segmenten met de inhoud van de caches.

De manier waarop dit gedaan wordt is hier eenvoudiger: Opnieuw worden alle segmenten uit S en L verzameld en deze keer worden ze gewoon gesorteerd volgens dalende populariteit. Men zal deze segmenten dan in die volgorde in L plaatsen totdat L vol is.

We vermoeden dat in de meeste gevallen de resultaten van dit algoritme gelijkwaardig of iets beter zullen zijn aan die van het Sliding Interval algoritme hierboven. Dit komt doordat in de meeste gevallen waar een gewone exponentieel dalende demandcurve wordt gebruikt en er geen samenwerking is tussen de proxies, het segment s1 op minuut K altijd een kleiner aantal aanvragen zal kennen dan het segment s2 op minuut K+1, met beide segmenten onderdeel van dezelfde zender. In dergelijke gevallen zal de inhoud van de cache van een SoF algoritme dan ook exact dezelfde zijn als die van het Sliding Interval algoritme. Maar het is juist in de situaties waar dit niet het geval is waar dit algoritme betere resultaten zal boeken.

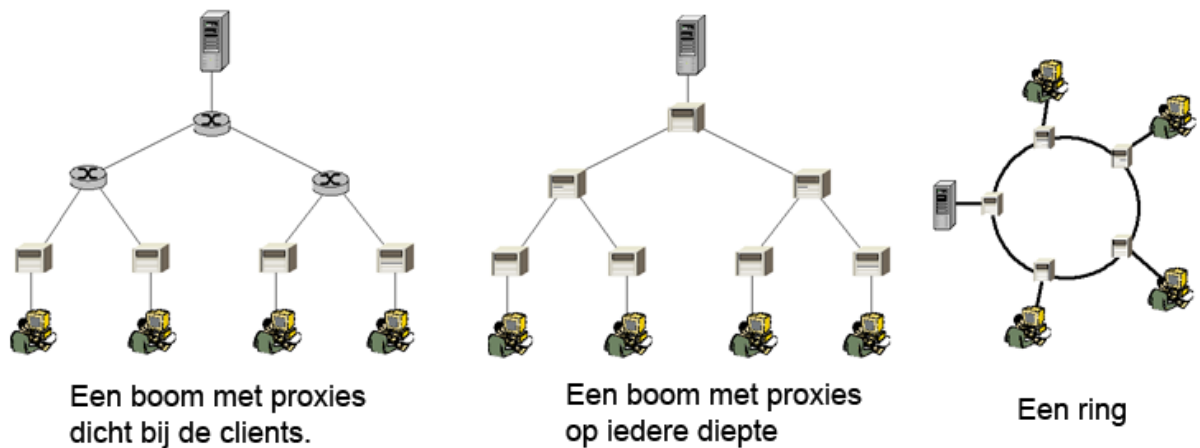
4.3 Simulatie resultaten

4.3.1 Vergelijkende studie

In dit deel zal getracht worden om de verschillende algoritmen te vergelijken ten opzichte van elkaar in verschillende topologieën. In totaal zullen 12 simulaties uitgevoerd worden, dit is elk van de 4 geziene algoritmes in 3 topologieën. Deze topologieën zijn: een boom met enkel proxies bij de clients, een boom met caches op elke diepte & een ring met caches.

Deze 12 simulaties delen de volgende eigenschappen:

- De demand bestaat uit 2 zenders die elk 3 programmas uitzenden, twee daarvan van een half uur in lengte, één daarvan een uur in lengte. De populariteit van die zenders is zodanig ingesteld dat op een bepaald moment altijd juist één van de twee zenders een populair programma heeft (met 1000 kijkers in totaal), en de andere een onpopulair programma heeft (met 500 kijkers in totaal).



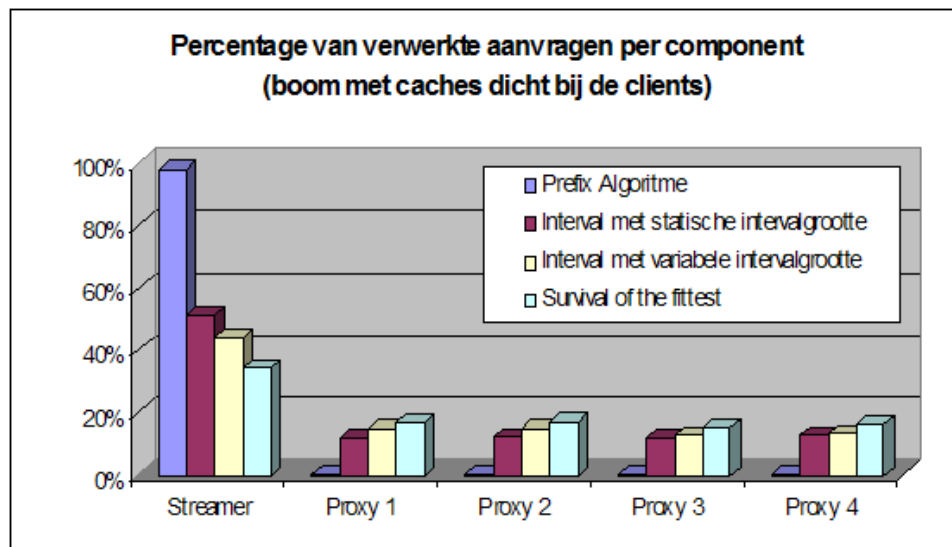
Figuur 4.3: De drie gesimuleerde topologieën

- De demandcurve is gedefinieerd door een constante C met waarde 1, dit wil zeggen: er komen geen nieuwe aanvragen meer naar het programma nadat de uitzending beëindigd is.
- Indien er proxy servers aanwezig zijn dan zullen die proxy servers een cache van 10 minuten hebben.
- Bij het simuleren van bomen zal downstream streamen dubbel zo duur zijn als upstream streamen.
- Er zullen altijd 4 clients aanwezig zijn, de verschillende aanvragen worden willekeurig verdeeld over deze clients.
- Er is geen samenwerking tussen de Proxies op vlak van caching beslissingen, de proxy servers hebben hun promiscuous mode afgezet.
- Bij algoritmes die een ΔT gebruiken zal deze gelijk gesteld worden aan 1.

Een boom met caches dicht bij de clients

We nummeren de proxies van 1 tot 4, van links naar rechts.

Zoals verwacht geeft het prefix algoritme een teleurstellend resultaat. Meer dan 95% van de aangevraagde segmenten moeten door de streamer geleverd worden. Dit is heel logisch als men weet dat het prefix algoritme er maar in slaagt om een prefix van 1 minuut per programma te bewaren in de cache. (2 minuten per programma met zes programmas zou niet meer in de caches passen) Veel betere resultaten krijgen we bij het glijdende interval algoritme met vaste



Figuur 4.4: Vergelijking belasting in een boom met proxies dicht bij de clients

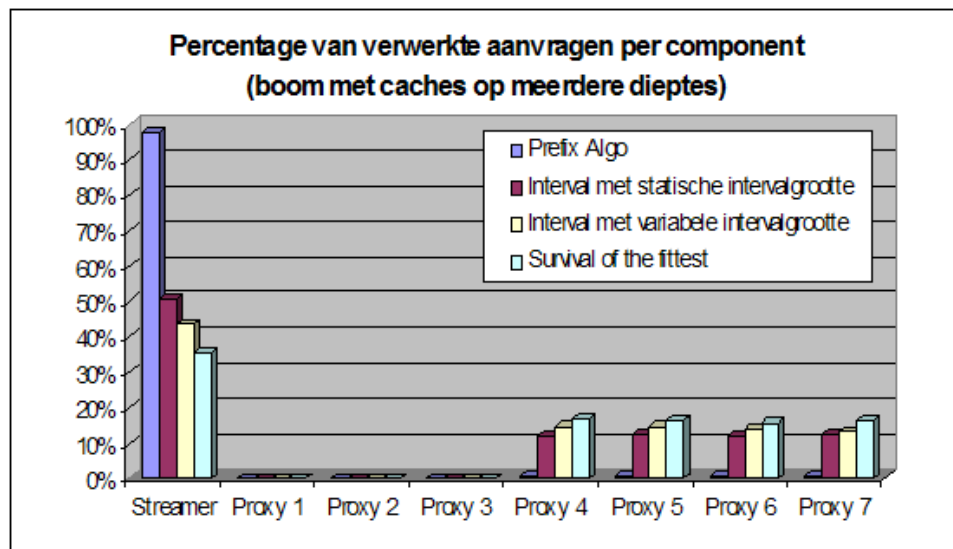
intervalgrootte. Het gelijkaardige algoritme met de variabele intervalgrootte slaagt erin om voor nog betere prestaties te zorgen doordat dat algoritme de afwisselende populariteit van de zenders beter kan volgen. Uiteindelijk doet het Survival of the Fittest algoritme nog beter omdat dit in staat is om nog meer flexibele cachingbeslissingen te maken.

Conclusie: In een omgeving waar niet samengewerkt wordt tussen de proxies kan men de prestaties van de geziene algoritmes als volgt sorteren van slechte hitrates naar goede hitrates: Sliding interval met constante intervals, sliding interval met variabele intervals, survival of the fittest. Het prefix algoritme is helemaal ongeschikt om bandbreedte te besparen.

Een boom met caches op verschillende dieptes

De proxies zijn als volgt genummerd: De proxy op diepte 1 krijgt 1, de twee proxies op diepte 2 krijgen 2 en 3 & de proxies het dichtst bij de client krijgen nummer 4, 5, 6 en 7.

Ten eerste moeten we opmerken dat alle algoritmes in deze topologie een heel gelijkaardig gedrag vertonen als in de vorige topologie. Verder is het belangrijk om op te merken dat proxies 1, 2, en 3 geen enkele aanvraag krijgen. Dit komt door het “domme” doorstuurgedrag van Proxies het dichtste bij de client op dit moment. Als een dergelijke Proxy een bepaald segment niet lokaal heeft dan zal hij die aanvraag doorsturen naar de streamer. Omdat de proxies op dit moment



Figuur 4.5: Vergelijking van de belasting van de componenten in een boom vol proxies

nog geen promiscuous mode gebruiken zullen hun caches dus ook leeg blijven.

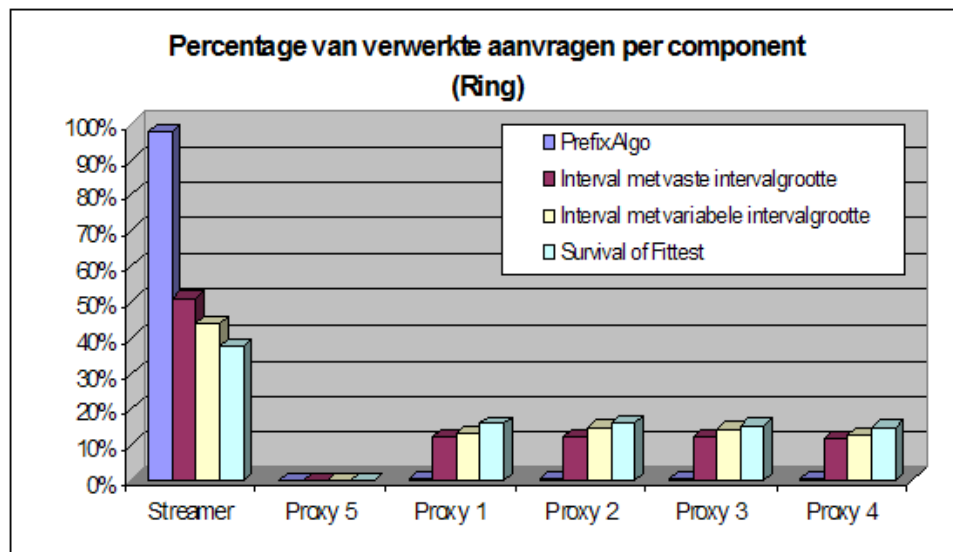
Conclusie: een intelligente vorm van het doorsturen van de aanvragen is strikt noodzakelijk opdat de proxies op kleinere diepte van een dergelijke boom nuttig zouden zijn.

Een ring met caches

De proxies zijn als volgt genummerd: Proxy 5 is degene die verbonden is met de streamer, Proxy 1 tot 4 zijn verbonden met de clients.

Ten eerste moeten we opnieuw opmerken dat de prestaties van de algoritmes hier opnieuw heel gelijkaardig zijn aan die in de vorige topologieën. We merken ook opnieuw op dat de proxy die niet verbonden is met een client opnieuw geen aanvragen krijgt als gevolg van dit domme doorstuur gedrag.

Conclusie: In een situatie waar er geen enkele vorm van samenwerking is tussen de Proxies zal de gebruikte topologie van niet veel belang zijn voor de belasting van de streamer en de gebruikte proxies.



Figuur 4.6: Vergelijking belasting in een ring

4.3.2 De algoritmes specifiek

In dit deel gaan we op zoek naar het effect van enkele factoren die de prestaties van de verschillende algoritmes kunnen beïnvloeden. Deze factoren zijn: de grootte van de caches en de gekozen waarde voor ΔT .

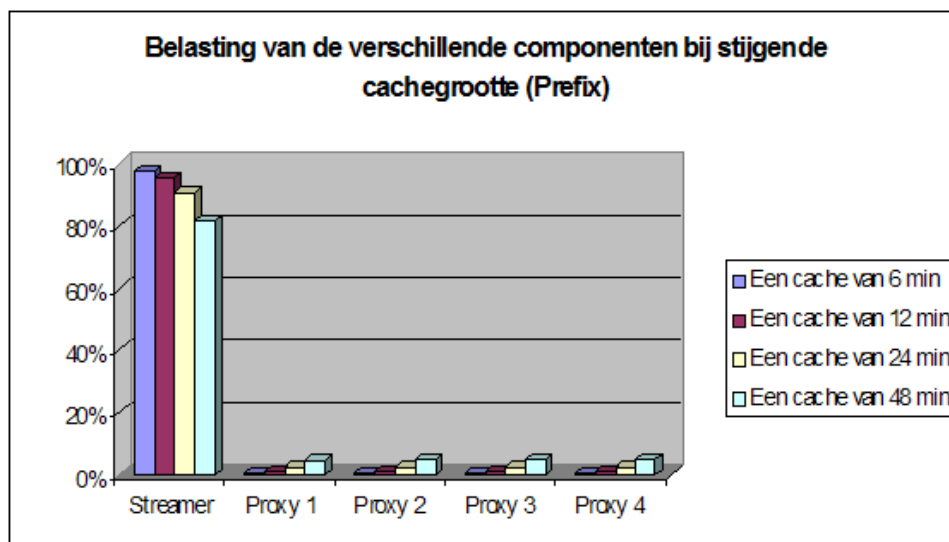
We gebruiken nog steeds dezelfde setup op gebied van zender en demandcurves als in het vorige deel. We doen al deze simulaties in een boom met enkel proxies dicht bij de clients.

Het prefix algoritme

Variërende cachegrootte

We doen 4 simulaties, de grootte van de caches varieert tussen 6, 12, 24 en 48. In onze simulatie zijn er 6 programmas dus dit zorgt voor een prefix van respectievelijk 1, 2, 4 en 8 minuten.

We merken op dat zelfs met grotere caches het prefix algoritme slecht blijft presteren op dit vlak. De reden is eenvoudig: door het groot aantal programmas moet de cachegrootte verdeeld worden in prefixen van slechts enkele minuten. Het is de verhouding van die prefixlengte met de totale lengte van een bepaald programma dat zal bepalen hoeveel percent van de aanvragen



Figuur 4.7: Prefix algo met variërende cachegroottes

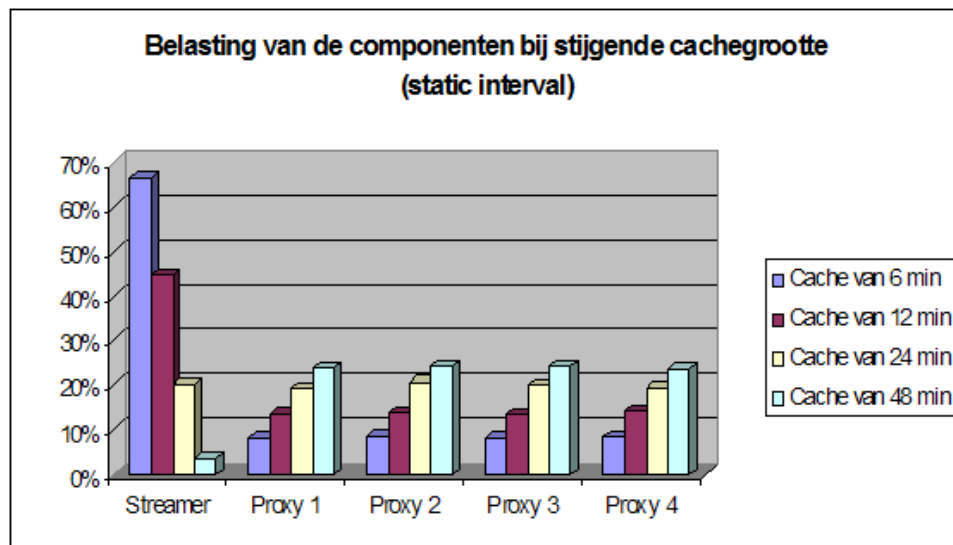
naar dat programma zullen kunnen geleverd worden. Er komen 60 minuten voor in de simulatie, om hiervoor een hitrate van 50% te bereiken (net zoals de concurrerende algoritmes) voor deze programma's heeft de Proxy een cache nodig van 30 minuten per programma. (Of 180 minuten in totaal)

Het sliding interval algoritme met constante interval grootte

Variërende cachegrootte

We doen 4 simulaties, één maal met in het netwerk enkel caches met grootte 6, 12, 24 en 48. In de simulatie zijn er 2 televisiezenders dus dit zorgt voor een constante interval grootte van 3, 6, 12 en 24 minuten

Het vergroten van de cachegrootte heeft een sterke vermindering van de belasting van de streamer tot gevolg. Toch heeft een verdubbeling van de cachegrootte geen halvering van de belasting van de streamer tot gevolg. Het effect van de grotere cache wordt afgeremd door de handicap van de constante interval grootte. Deze zorgt er soms voor dat, zelfs als de cache groot genoeg is om alle populaire segmenten in op te slaan, soms deze segmenten niet kunnen opgeslagen worden omdat een minder populair programma van een andere televisiezender moet gecached worden. Hopelijk zullen de volgende algoritmes dit beter doen.

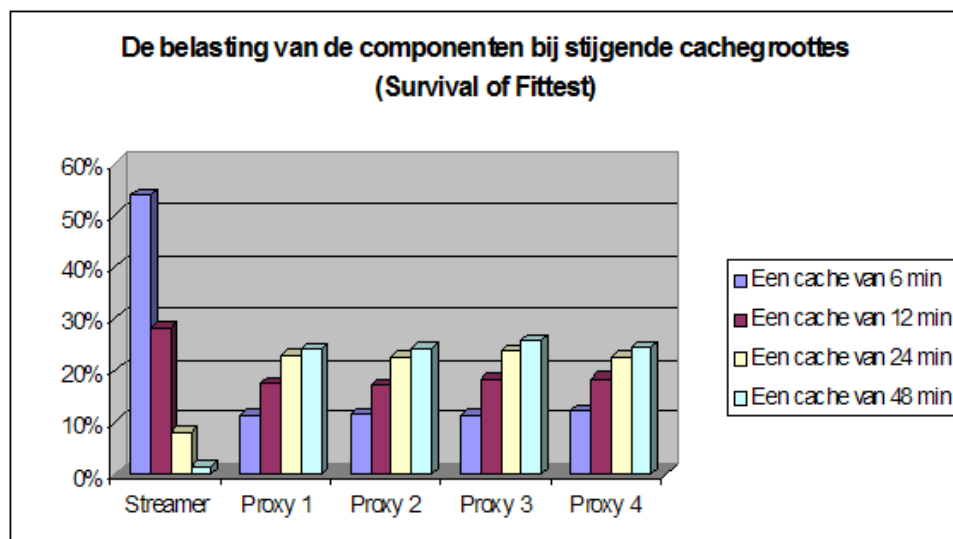


Figuur 4.8: Sliding interval met vaste intervalgrootte met variabele cachegroottes

Het sliding interval algoritme met variabele interval grootte

Variërende cachegrootte

We doen 4 simulaties, één maal met in het netwerk enkel caches met grootte 6, 12, 24 en 48.



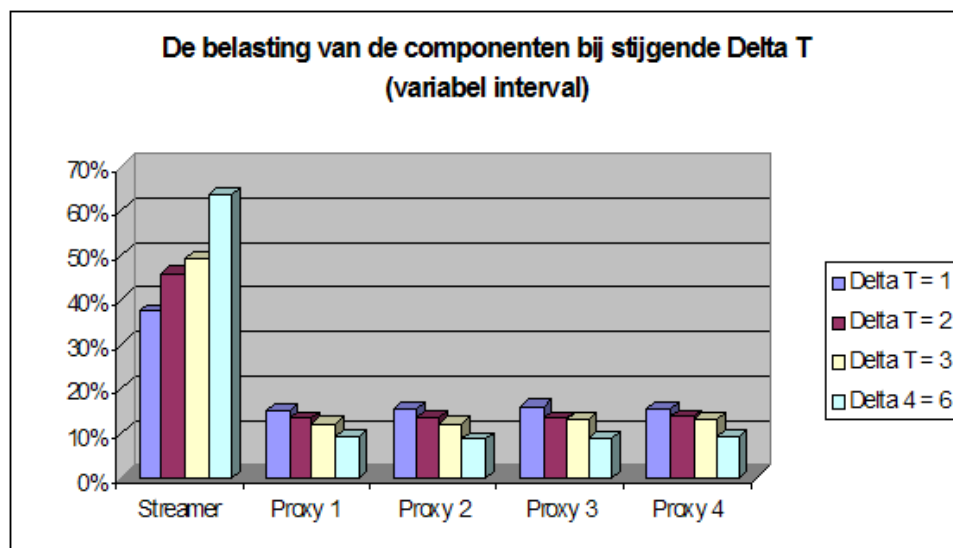
Figuur 4.9: Sliding interval met variabele intervalgrootte met variabele cachegroottes

Hier heeft een verdubbeling van de cachegrootte een iets beter effect. Dit algoritme heeft minder

last van de handicap van het vorige algoritme. De bijgekomen cacheruimte wordt meer optimaal gebruikt door ze te besteden aan de segmenten van de zender met de hoogste populariteit.

Variërende ΔT

Deze keer houden we de cachegrootte vast op 12 minuten en we laten de waarde van ΔT variëren over de waarden 1, 2, 3 en 6. Er zijn twee televisiezenders dus bij een waarde $\Delta T = 6$ zal de volledige cache ingenomen worden door S en zal L onbestaande zijn.



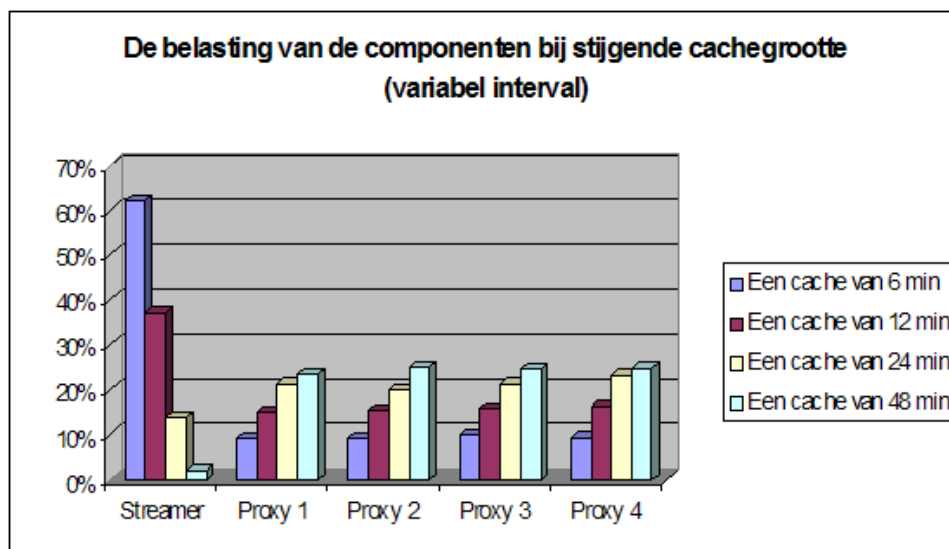
Figuur 4.10: Sliding interval met variabele intervalgrootte met variabele Delta T

Zoals verwacht zorgt een groter wordende ΔT voor steeds slechtere resultaten van de caches. Bij een kleinere cache, zoals hier, zal een grote ΔT ervoor zorgen dat het L gedeelte steeds kleiner wordt. Het is in het L gedeelte dat het algoritme vrijheid heeft om te kiezen welke segmenten hij zal cachen. Indien ΔT zodanig groot wordt dat S de volledige cache inneemt dan zal het algoritme helemaal niet meer kunnen werken, er blijft enkel nog S over.

Het survival of the fittest algoritme

Variërende cachegrootte

We doen 4 simulaties, één maal met in het netwerk enkel caches met grootte 6, 12, 24 en 48.



Figuur 4.11: Survival of the Fittest met variabele cachegroottes

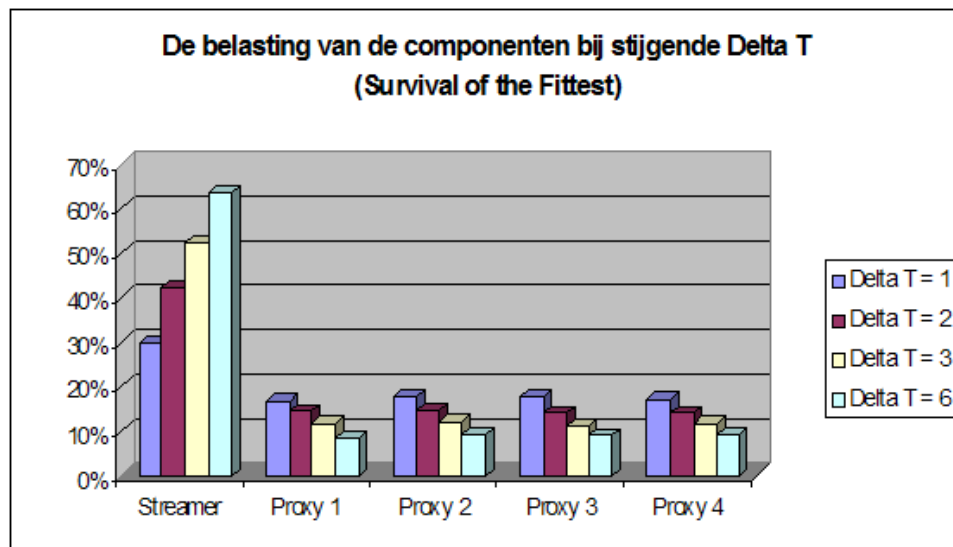
We zien hier dat bij Survival of the Fittest een verdubbeling van de cachegrootte een halvering van de belasting van de streamer tot gevolg heeft. De bijgekomen cacheruimte zal dus iedere keer optimaal gebruikt worden, dit door de flexibiliteit van dit algoritme

Variërende ΔT

Ook nu zetten we de cachegrootte vast op 12 minuten en we laten de waarde van ΔT variëren over de waarden 1, 2, 3 en 6. Er zijn twee televisiezenders dus bij een waarde $\Delta T = 6$ zal de volledige cache ingenomen worden door S en zal L onbestaande zijn.

Hier hebben we opnieuw de situatie waar een groter wordende ΔT voor slechtere caching resultaten zorgt. Opnieuw om dezelfde reden, hoe groter S wordt ten opzichte van L, hoe minder vrijheid het algoritme zal hebben in zijn beslissingen. Dit komt mooi tot uiting in het feit dat hoe groter ΔT wordt, hoe meer de resultaten van het sliding interval algoritme met variabele intervalgrootte en dit algoritme naar elkaar toe groeien. Bij $\Delta T = 6$ zijn de belastingen van de streamers zelfs aan elkaar gelijk. Dit is logisch want bij een $\Delta T = 6$ zijn beide eigenlijk hetzelfde algoritme.

Conclusie: In een situatie waar er geen samenwerking is tussen de Proxy servers en waar de demandcurve geen al te grote fluctuaties vertoont zal een grotere ΔT enkel tot slechtere presta-



Figuur 4.12: Survival of the Fittest algoritme met variabele Delta T

ties leiden.

Conclusie: Alle algoritmes zullen beter presteren bij een groter wordende cache. De relatieve prestatieverbetering is het sterkste bij het Survival of the Fittest algoritme, beide sliding interval zullen een prestatieverbetering ondervinden die relatief gezien minder groot is.

Hoofdstuk 5

Samenwerkende proxy servers

In het vorige hoofdstuk is besproken geweest hoe één enkele proxy server erin slaagt door de meest frequent aangevraagde segmenten te cachen de belasting van de streamer en de links daar naar toe te verminderen. In dit hoofdstuk vragen we ons af of we deze prestaties nog kunnen verbeteren door de proxy servers beter te laten samenwerken. Deze samenwerking kan onder ander inhouden dat men rekening houdt met de inhoud van de cache van naburige proxies bij de beslissing of een bepaald segment in de eigen cache wordt geplaatst/verwijderd. Verder bespreken we nog een drietal andere samenwerkingsvormen. Op het einde van dit hoofdstuk worden een tweetal speciale gevallen gesimuleerd en de resultaten geïnterpreteerd.

5.1 Hoe?

Samenwerking impliceert communicatie. Een vorm van communicatie die gewone proxy servers op dit moment nog niet ondersteunen. In een toegangsnetwerk waar nog niet samengewerkt wordt kunnen we grofweg een drietal vormen van communicatie (3 protocollen) onderscheiden die daar enkel zijn om de videostreams mogelijk te maken. Als eerste is er RTSP, dit wordt gebruikt om streams op te zetten, af te sluiten en verschillende andere aspecten van de streams te onderhandelen. De eigenlijke streams lopen over het RTP protocol. Een derde protocol dat nodig is om digitale televisie mogelijk te maken is er één dat de EPG (Elektronische Programma Gids) ondersteunt. Er moet immers een manier zijn om de gebruiker in te lichten over de content.

Om de samenwerking mogelijk te maken is er dus een vierde protocol nodig, één dat kan gebruikt

worden door de proxies om informatie over elkaar uit te wisselen. Die informatie kan zijn: de belasting van de proxy op dat moment, de hoeveelheid plaats die nog vrij is in de cache van de proxy, de belasting van de links rondom de proxy, het aanwezig zijn van een bepaald segment in de cache & eventueel ook het bijkomen/verwijderen van een proxy server in/uit het netwerk. Dit uitwisselen kan natuurlijk op vele manieren gebeuren, via een peer-to-peer structuur of via een gecentraliseerde server, via een push of via een pull architectuur. Het is uiteraard niet de bedoeling dat in deze scriptie deze vorm van communicatie of het protocol waar het over gebeurt uitgebreid wordt besproken (dit is voer voor een nieuwe scriptie), toch doen we hier kort een suggestie over hoe zo'n architectuur eruit zou kunnen zien.

De architectuur die ik voorstel maakt gebruik van één gecentraliseerde en gespecialiseerde server waar de proxy servers hun informatie zullen op pushen. Deze server zal op zijn beurt de informatie op de andere proxies in het netwerk pushen. Het is vanzelfsprekend dat hier niet kan worden gebruik gemaakt van informatie pullen, niet van de server en zeker niet van andere proxies. De verschillende algoritmes hebben elkaars informatie zodanig veel nodig dat ook maar de minste delay ongewenst is, informatie over de andere proxies moet als het ware reeds aanwezig zijn in het geheugen nog voor de proxy het nodig heeft gehad. Hierbij is het geen ramp dat die informatie enkele seconden oud is, een proxy herberekent immers niet iedere seconde de inhoud van zijn cache en een link die vijf seconden geleden zwaar belast was zal dit nu waarschijnlijk nog steeds zijn. Ik stel voor dat iedere Proxy in het netwerk alle informatie waarvan hij denkt dat andere proxies geïnteresseerd zouden kunnen zijn vergaart gedurende een periode van pakweg 10 seconden waarna hij die informatie doorstuurt naar de server, het is dan de verantwoordelijkheid van de server om deze informatie door te sturen naar alle andere proxies.

Het gebruiken van een centrale server heeft bovendien een ander voordeel, het kan de proxies min of meer plug 'n play maken. Tot hier toe gingen we er van uit dat iedere proxy server op zijn minst op de hoogte was van het bestaan en het IP adres van alle andere proxy servers en alle streamers in het netwerk. Maar men mag niet vergeten dat in een realistisch toegangsnetwerk er honderden of zelfs duizenden dergelijke proxy servers aanwezig kunnen zijn. Het handmatig moeten herconfigureren van al deze proxies door een adreswijziging van één van deze proxies is onaanvaardbaar. Daarom kan die centrale server dienst doen als een soort discovery service. Een nieuw geïnstalleerde proxy moet enkel het adres van de centrale server weten en alle nodige

informatie wordt hem opgestuurd: wat is het adres van de streamer(s), welke andere proxies zijn aanwezig, welke ΔT zal ik gebruiken, ... Het feit dat een nieuwe Proxy zich heeft aangemeld binnen het netwerk wordt dan op die manier ook naar de andere proxies gestuurd.

5.2 Een overzicht van de samenwerkingsvormen

In totaal onderscheiden we vier verschillende vormen van samenwerking tussen Proxy servers die gebruikt kunnen worden om de prestaties van die proxies te verbeteren:

5.2.1 Berekenen van de populariteit van segmenten

Wat?

De eerste methode is het berekenen van de populariteit van segmenten rekening houdende met de lokatie van identieke segmenten in andere componenten. Als een proxy server een aanvraag krijgt naar een bepaald segment dan zal, afhankelijk van het caching algoritme dat die proxy hanteert, al dan niet de populariteit van dat segment herberekend worden. Eén manier om dit te doen is door de bijkomende populariteit voor dat segment te berekenen als de som van gewichten van de route tussen de proxy zelf en de component die het dichtste bij is die dat segment wel heeft. Op deze manier zullen segmenten waar dichtbij ook een kopie van bewaard wordt een lagere populariteit krijgen als segmenten waar er slechts veraf een kopie van te vinden is. Op deze manier kan een algoritme proberen om de segmenten te bewaren die een te hoge kost zouden betekenen indien ze elders moesten gehaald worden.

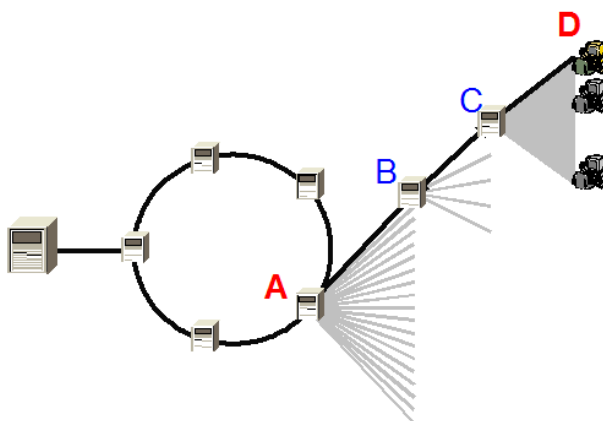
Omdat voor deze methode samenwerking tussen de proxy servers nodig is kunnen we niet anders dan het onder te verdelen bij de samenwerkingsvormen. Toch is er een subtiel verschil tussen deze techniek en andere samenwerkingsvormen zoals intelligent doorsturen. Hier wordt de verkregen informatie maar indirect gebruikt om de inhoud van de cache mee te bepalen en daardoor heeft deze techniek maar een beperkte invloed op de werking van de algoritmes. Daarom hebben we er reeds in het vorige hoofdstuk voor gekozen om deze techniek te gebruiken bij al onze caching algoritmes, ook degene die niet samen werken op een directe manier. Het is belangrijk dat de caching algoritmes over de juiste informatie beschikken en dus daardoor de werking en

de prestaties van die algoritmes meer tot uiting komen.

5.2.2 Interfaces in promiscuous mode

Wat?

Opnieuw is deze methode op het eerste zicht geen samenwerkingsvorm, er zijn geen afspraken tussen de proxy servers, maar toch moet deze techniek in dit hoofdstuk besproken worden. Deze techniek zorgt namelijk voor een prestatieverbetering bij een bepaalde proxy, enkel door gebruik te maken van iets wat een andere proxy doet.



Figuur 5.1: Tussenliggende proxies met promiscuous mode

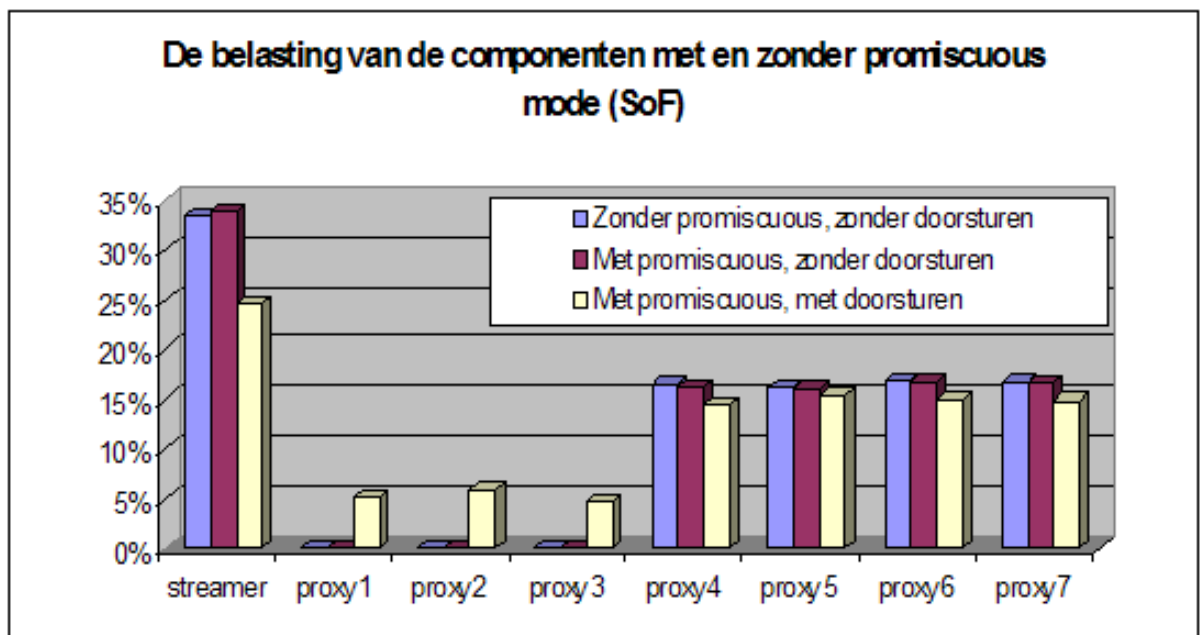
Onder promiscuous mode verstaan we de mode waarin een ethernet interface van een machine kan ingesteld worden die die machine kan toe laten om alle voorbijkomende pakketten te onderscheppen. Concreet voor deze situatie houdt dit in dat een proxy server waarvan de interfaces in promiscuous mode zijn ingesteld niet alleen de streams kan onderscheppen die bij hem toekomen maar ook die streams die enkel passeren. De proxy servers worden meestal op routers geïnstalleerd dus er passeert voorbij die routers heel wat data die niet voor die proxy server bedoeld is. Deze mode staat de proxy server toe deze streams te onderscheppen en eventueel te cachen. De tekening toont een situatie waar Proxy A een fragment streamt naar Client D, deze stream loopt voorbij Proxy B en Proxy C en als zij hun promiscuous mode geactiveerd hebben dan kunnen zij dit fragment cachen.

Vergelijkende simulaties

Voor de komende simulaties maakten we gebruik van een boom met proxies op meerdere dieptes en proxies met cachegrootte 10. We doen in totaal drie soorten simulaties: één waarbij die proxies geen promiscuous mode gebruiken, één waarbij de proxies wel promiscuous mode gebruiken maar nog steeds alle aanvragen doorsturen naar de streamer en één waar de proxies gebruik maken van promiscuous mode én aanvragen naar elkaar doorsturen. We voeren deze simulaties twee keer uit, eens met Sliding Interval met vaste lengte algoritmes en eens met Survival of the Fittest algoritmes op alle proxies:

We nummeren opnieuw de proxies als volgt: de proxy op diepte 1 heet proxy 1, die op diepte 2 zijn proxy 2 en proxy 3, die op diepte 3 zijn proxy 4 tot 7.

Met Survival of the Fittest:

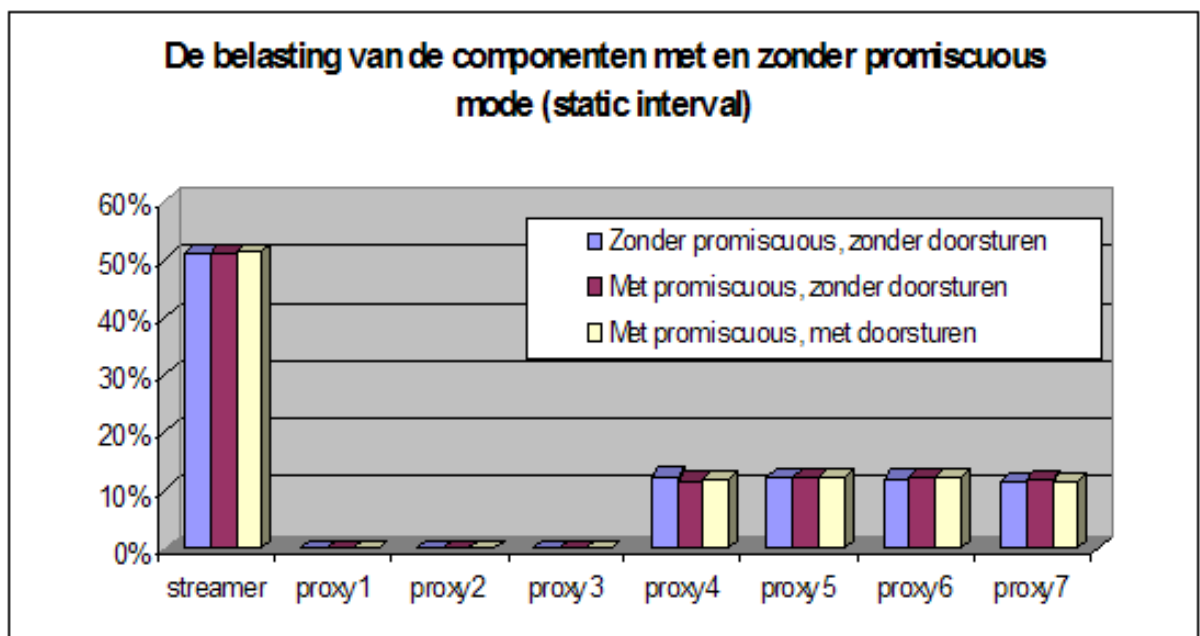


Figuur 5.2: Vergelijking belasting met en zonder promiscuous mode (SoF)

Indien we geen gebruik maken van promiscuous mode dan wordt de belasting mooi verdeeld tussen de streamer en proxy 4 tot 7, dit zijn immers de proxies het dichtste bij de clients. De andere proxies krijgen geen aanvragen en zullen hun cache dus ook niet vullen met segmenten.

Zetten we promiscuous mode aan bij alle proxies, maar veranderen we de doorstuurstrategie niet dan zullen proxy 1 tot 3 zich wel vullen met segmenten maar ze zullen nog steeds geen aanvragen krijgen, vandaar dat de belasting van deze proxies op nul blijft staan. Als we een iets meer gecompliceerde doorstuur strategie gebruiken in combinatie met de promiscuous mode dan zien we dat de belasting op de streamer daalt en dat de hoger gelegen proxies een deel van het werk op zich nemen, de proxies het dichtste bij de clients blijven het meeste werk doen.

Met Sliding Interval algoritme met vaste intervallengte:



Figuur 5.3: Vergelijking belasting met en zonder promiscuous mode (Static interval)

We doen nu dezelfde simulaties, maar dan met het Sliding Interval algoritme met vaste intervallengte, voor de eerste twee simulaties geeft dit een heel gelijkaardig resultaat, proxy 1 tot 3 nemen zoals verwacht geen belasting over van de andere componenten. Bij de derde simulatie zien we wel een groot verschil met het vorige voorbeeld. Proxies 1 tot 3 hebben schijnbaar geen enkel nut. Het moet wel gezegd worden dat de grafiek op dit vlak wat bedriegt, de outputbestanden van de simulator leren ons dat deze proxies telkens 0.01% van de totale belasting op zich nemen.

Dit verschil is ontstaan doordat het Sliding Interval algoritme met vaste intervallengte een wei-

nig flexibel caching algoritme is, dit zorgt ervoor dat als men dergelijke proxies op verschillende dieptes zet in een boom en als die proxies dezelfde cachegrootte hebben dat die caches bijna altijd identiek dezelfde segmenten zullen bevatten. Wanneer een segment niet lokaal kan gevonden worden bij de proxy op het diepste niveau dan is het heel onwaarschijnlijk dat één van de caches op hoger niveau dit segment wel zal hebben. Het Survival of the Fittest algoritme is dan weer veel meer flexibel en zal enkel de segmenten cachen waar het veel aanvragen voor krijgt. Een proxy op hoger niveau zal meestal aanvragen krijgen naar segmenten die de proxies op lager niveau niet lokaal hebben dus op die manier zijn die hoger gelegen proxies wel nuttig.

5.2.3 Requests intelligent doorsturen

Wat?

Als een proxy een aanvraag krijgt naar een bepaald segment en hij heeft dat segment op dat moment niet lokaal in zijn cache dan is die proxy verplicht om die aanvraag door te sturen naar een andere component, ofwel een streamer ofwel een andere proxy. Het doorsturen kan hij doen op drie manieren: ofwel stuurt hij steeds al zijn aanvragen door naar een streamer. Hiervoor is geen samenwerking nodig maar we vinden dit geen goede strategie, het is tenslotte ons doel om de belasting van de streamer te verminderen. Een andere mogelijkheid is om andere proxy servers te zoeken waar het fragment wel aanwezig is en de aanvraag daar naar toe te sturen. Een derde mogelijkheid is om de aanvraag door te sturen naar een proxy server die dat fragment niet heeft. Dit kan gedaan worden om de andere proxy te suggereren dat hij het fragment zoekt en cachet. Om de laatste twee mogelijkheden te kunnen doen is er samenwerking nodig tussen de proxies.

Uiteindelijk is geen enkel van die drie mogelijkheden op zichzelf adequaat om tot goede prestaties te komen. Wat een proxy nodig heeft is een goede strategie voor het doorsturen van segmenten die afwisselt tussen elk van deze mogelijkheden. We bespreken vier Topologie algoritmen, elk met zijn eigen strategie.

Doorsturen naar de streamer: Een proxy die dit topologie algoritme gebruikt zal steeds alle requests doorsturen naar de dichtstbijzijnde streamer. Er is absoluut geen vorm van samenwerking tussen de proxies. Uiteraard zal dit algoritme leiden tot onnodige belasting van de streamer.

Normaal Topologie Algoritme: Als een request niet lokaal afgehandeld kan worden dan zal een proxy gezocht worden die het segment wel lokaal heeft. Als er meerdere dergelijke proxies werden gevonden dan zal de request doorgestuurd worden naar de dichtstbijzijnde. Als er geen enkele dergelijke proxy kan gevonden worden dan wordt de request doorgestuurd naar de streamer.

Shared Topologie Algoritme: Dit algoritme is gelijkaardig aan het Normale Topologie Algoritme maar gaat nog een stapje verder. Als een request niet lokaal afgehandeld kan worden dan zal een proxy gezocht worden die het segment wel lokaal heeft. Als er meerdere dergelijke proxies worden gevonden dan zal de request doorgestuurd worden naar de dichtstbijzijnde. Als er geen enkele dergelijke proxy kan gevonden worden dan wordt de aanvraag doorgestuurd naar de proxy die op dat moment de meeste plaats vrij heeft in zijn cache. Op deze manier kunnen de proxies elkaar de suggestie doen om een bepaald segment te cachen. Als er meerdere proxies zijn met evenveel plaats vrij in hun cache dan wordt de aanvraag gestuurd naar degene die het dichtste bij is. Om te verhinderen dat op deze manier aanvragen blijven circuleren in het netwerk zal een proxy eerst controleren of de hoeveelheid vrije plaats in zijn eigen cache niet groter of gelijk is aan dat van de andere proxy waarvan hij denkt dat die het meeste plaats vrij heeft. In dat geval stuurt hij de aanvraag door naar de dichtstbijzijnde streamer. Bij dit algoritme zal een stream dus nooit meer dan 2 proxies passeren voor hij bij de client toekomt.

Complex Topologie Algoritme: Dit algoritme is gelijkaardig aan het Shared Topologie Algoritme maar gaat opnieuw een stapje verder. Opnieuw zal eerst een proxy worden gezocht die het fragment wel lokaal heeft, indien er geen dergelijke proxy gevonden kan worden dan wordt de aanvraag doorgestuurd naar een proxy met de kleinste score volgens de volgende formule:

$$-c_A \frac{A}{A_g} + c_B \frac{B}{B_g} + c_C \frac{C}{C_g} + c_D \frac{D}{D_g}$$

met:

A het aantal plaatsen vrij in de cache van die andere proxy

B de belasting van die andere proxy in de voorbije minuut

C het gewicht van de route van die andere proxy tot aan de dichtste streamer

D het gewicht van de route van hier tot aan die andere proxy

en:

A_g, B_g, C_g & D_g de gemiddelden over alle andere proxies,
 c_A, c_B, c_C & c_D de gewichten, positieve gehele getallen.

De aanvraag wordt doorgestuurd naar de proxy met de kleinste score. Indien een proxy zelf de proxy is met de kleinste score stuurt hij de aanvraag door naar een streamer. Voor deze manier van werken is het natuurlijk belangrijk dat alle proxies dezelfde formule gebruiken. Deze formule is heel erg instelbaar door de provider, de gewichten kunnen vrij gekozen worden. Door een gewicht gelijk te stellen aan nul wordt die factor niet meer meegerekend in het algoritme

Vergelijkende simulaties

In totaal doen we zes simulaties:

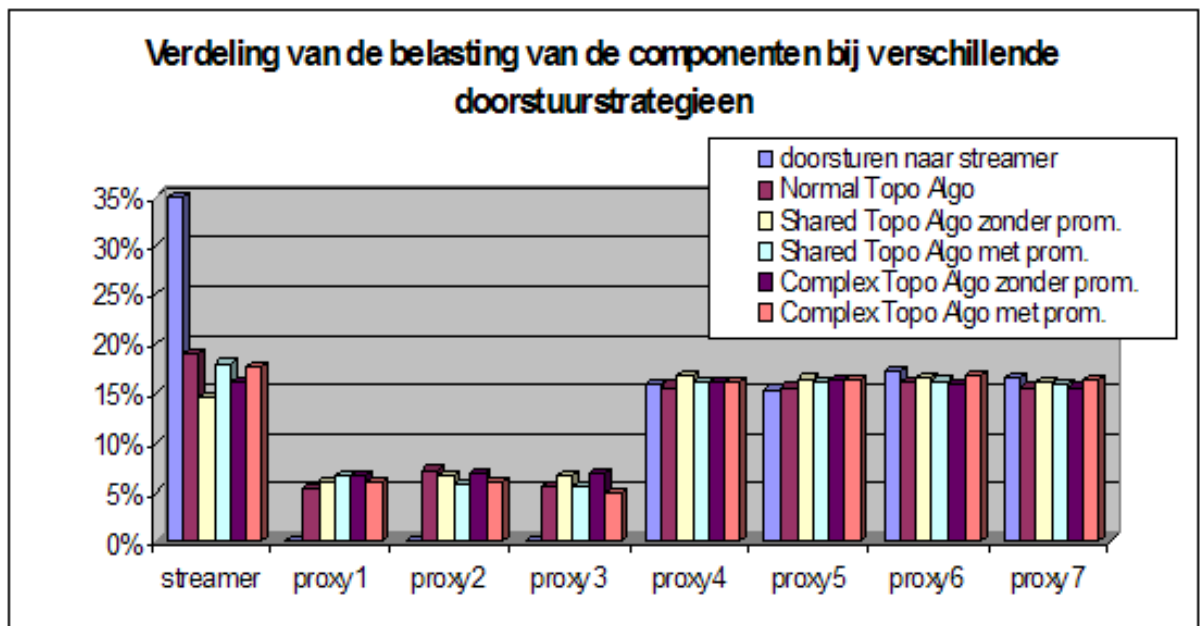
- Doorsturen naar streamer
- Normal Topo Algo met promiscuous mode
- Shared Topo Algo zonder promiscuous mode
- Shared Topo Algo met promiscuous mode
- Complex Topo Algo met alle coëfficiënten gelijk aan 1, zonder promiscuous mode
- Complex Topo Algo met alle coëfficiënten gelijk aan 1, met promiscuous mode

De caching algoritmes op de proxies zijn survival of the fittest.

Er zijn een drietal conclusies die we kunnen trekken uit deze simulaties:

Een **doorstuur strategie is essentieel**. Enkel doorsturen naar de streamer zal nooit tot goede resultaten leiden omdat proxies op kleinere dieptes zo ongebruikt blijven.

Als er wel een doorstuur strategie gebruikt wordt die op zoek gaat naar andere proxies die een bepaald segment wel lokaal hebben, dan zijn de details van die doorstuurstrategie van niet veel belang. We ondervinden maar **heel weinig verbeteringen** naar mate de doorstuur strategieën nog complexer worden. We merken een verbetering van enkele procenten tussen Normal Topo



Figuur 5.4: Vergelijking van verschillende doorstuur strategieën

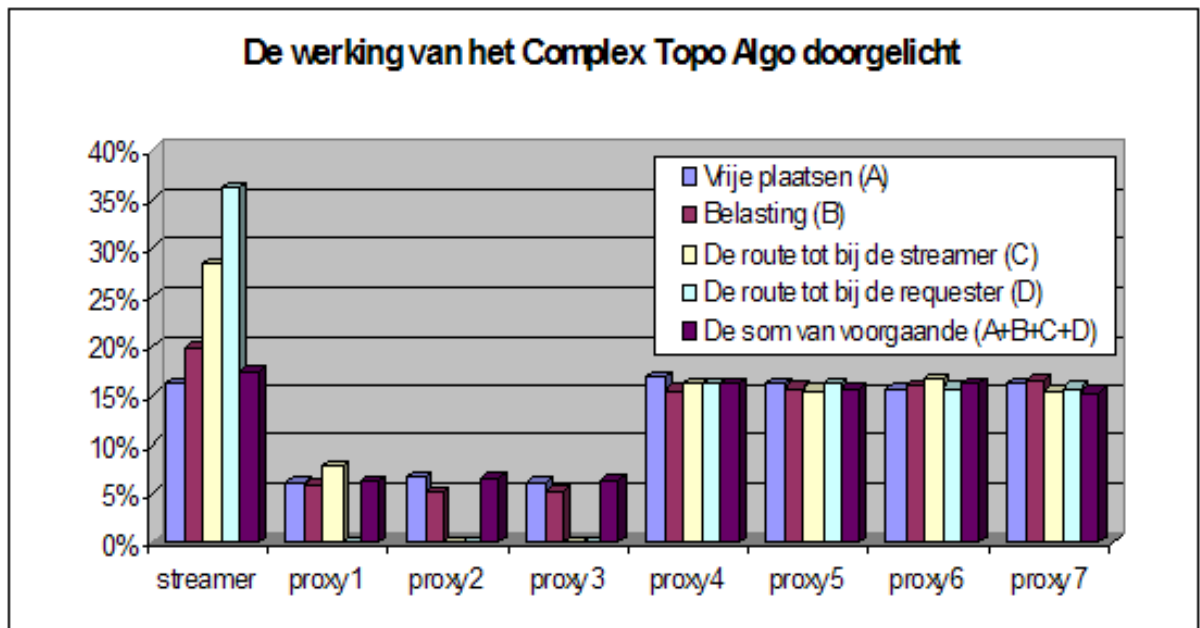
Algo en Shared Topo Algo, maar we zien al helemaal geen verbetering tussen Shared Topo Algo en Complex Topo Algo. Verder in dit hoofdstuk zullen we meer uitgebreid de gewichten van Complex Topo Algo bestuderen.

Het **combineren van de promiscuous mode** met de verschillende doorstuur strategieën blijkt geen verbetering op te leveren. Het normal topo algo is het enige dat verbeteringen ondervindt van de mode, dit is logisch want zonder promiscuous mode werkt deze strategie niet goed. De meer complexe doorstuur strategieën ondervinden helemaal geen verbetering meer, in tegendeel het lijkt erop dat de promiscuous mode de goede werking van deze strategieën tegen werkt. Dit komt doordat, als een proxy op hoger niveau in de boom de promiscuous mode gebruikt, de cache van die proxy het grootste deel van de simulatie volledig vol zal zitten. Als alle proxies aanwezig in het netwerk volledig volzet zijn dan heeft het doen van een suggestie aan een andere proxy om een bepaald segment te cachen helemaal geen zin meer.

Het complex topologie algoritme instellen

Er zijn vier gewichten, we zullen ze voor het gemak gewicht A, B, C & D noemen, zoals hierboven gedefinieerd. We doen vijf simulaties: één waarbij iedere gewicht is gelijk gesteld aan

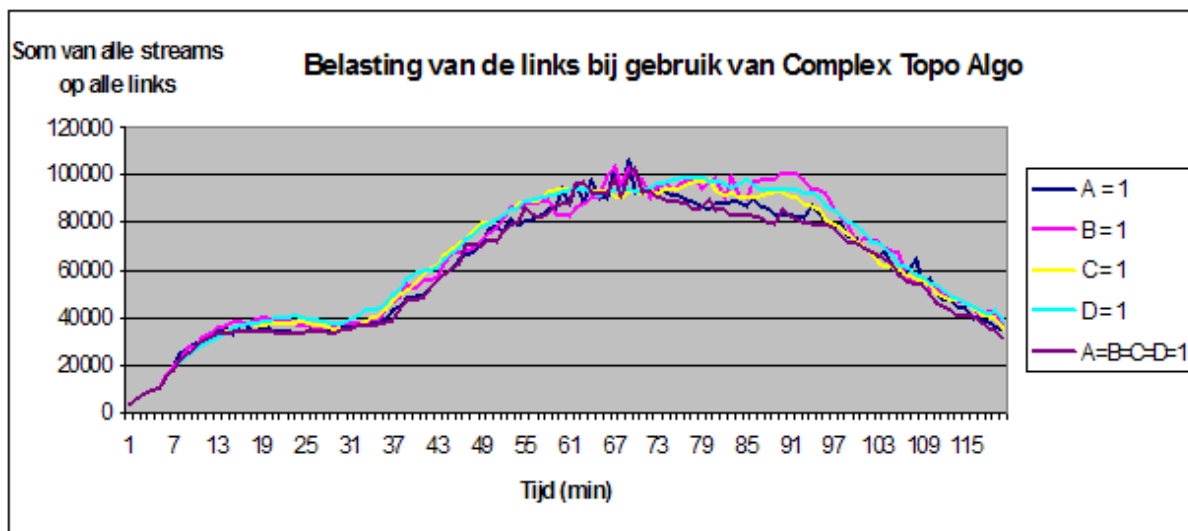
1 en de andere gewichten aan 0 (om zo het effect van enkel die factor te onderzoeken) en een vijfde simulatie waar we alle gewichten gelijk stellen aan 1 (en dus het gemiddelde onderzoeken van alle factoren). We gebruiken geen promiscuous mode en het gebruikte caching algoritme is Survival of the Fittest.



Figuur 5.5: De werking van het Complex Topo Algo doorgelicht

De conclusie die we uit deze simulatie moeten trekken is dat de meeste van deze factoren niet heel nuttig zijn als het gaat om belasting verdeling. Als we de eerste vier simulaties vergelijken zien we dat de eerste veruit de beste resultaten geeft, en dit geval is eigenlijk niets anders dan het Shared Topo Algoritme. Het doorsturen volgens belasting van de proxies lijkt niet zo effectief als in de eerste simulatie. In de derde simulatie wordt enkel doorgestuurd naar die ene proxy die het dichtste bij de streamer zit en in de vierde simulatie wordt helemaal niet doorgestuurd. De simulatie waar het gemiddelde van de factoren gebruikt wordt toont iets betere resultaten maar nog steeds niet zo goed als die van het Shared Topo Algo

Als we de grafiek met de belasting van de links in het netwerk bekijken zien we opnieuw dat hier geen enkele factor zorgt voor resultaten die uitblinken ten opzichte van anderen. Het gebruik van het gewogen gemiddelde zorgt voor een iets beter globaal resultaat maar de verschillen zijn te klein om een eventueel hogere belasting van de streamer te kunnen verantwoorden.



Figuur 5.6: De belasting op van de links over de tijd bij Complex Topo Algo

Conclusie: In een boom-topologie zal het Complex Topo Algo zeker niet beter functioneren als het gewone Shared Topo Algo. Doorsturen op basis van het aantal vrije plaatsen in de cache lijkt de meest succesvolle strategie te zijn.

5.2.4 Intelligente caching beslissingen nemen

Wat?

Ook de beslissing of een segment al dan niet in de cache wordt opgeslagen is een beslissing die we kunnen verbeteren door gebruik te maken van samenwerking. We hebben één algoritme dat bij het nemen van deze beslissing gebruik maakt van gegevens van andere proxies, we noemen dit algoritme het “Shared Cache Algo”.

Shared Cache Algo: Als de provider gebruik wil maken van dit caching algoritme moet hij eerst specificeren hoeveel kopieën van elk segment hij toelaat in het netwerk. We noemen het maximaal aantal toegelaten kopieën van dat segment K . Het algoritme werkt als volgt: als een bepaald segment gecaptured wordt dan moet het aan twee voorwaarden voldoen om gecached te kunnen worden: er mogen op dat moment niet meer dan $K - 1$ kopieën van dat segment aanwezig zijn in het netwerk en het segment voldoet aan de voorwaarden die het Survival of the Fittest

algoritme op legt. Dit algoritme is dus eigenlijk niet meer een uitbreiding van het Survival of the Fittest algoritme dat een extra voorwaarde stelt bij het cachen. We hebben gekozen om dit algoritme uit te breiden omdat het het meest flexibele was van alle niet-coöperatieve algoritmes en het dus de beste prestaties had in een niet samenwerkende context.

Door het aantal kopieën van elk segment gelijk te stellen aan 1 vormt men het netwerk als het ware om tot één grote proxy met een enorm grote cache. Als die globale cache zodanig groot is dat alle segmenten erin passen dan kan men in principe de belasting van de streamer herleiden tot bijna nul. (Er is natuurlijk altijd minstens 1 stream per segment nodig tussen de streamer en de proxy die dat segment cacht). Nog voor we één simulatie gedaan hebben kunnen we dus al stellen dat dit algoritme belasting van de streamer zal inruilen voor meer verkeer op de links. Een groter wordende K zal dat verkeer op de links verminderen maar zal meer of grotere caches vereisen om nog steeds alle segmenten te kunnen cachen.

Stel:

Het maximaal aantal kopieën van ieder segment is K ,

De maximale lengte van een programma is M ,

Het aantal zenders is Z ,

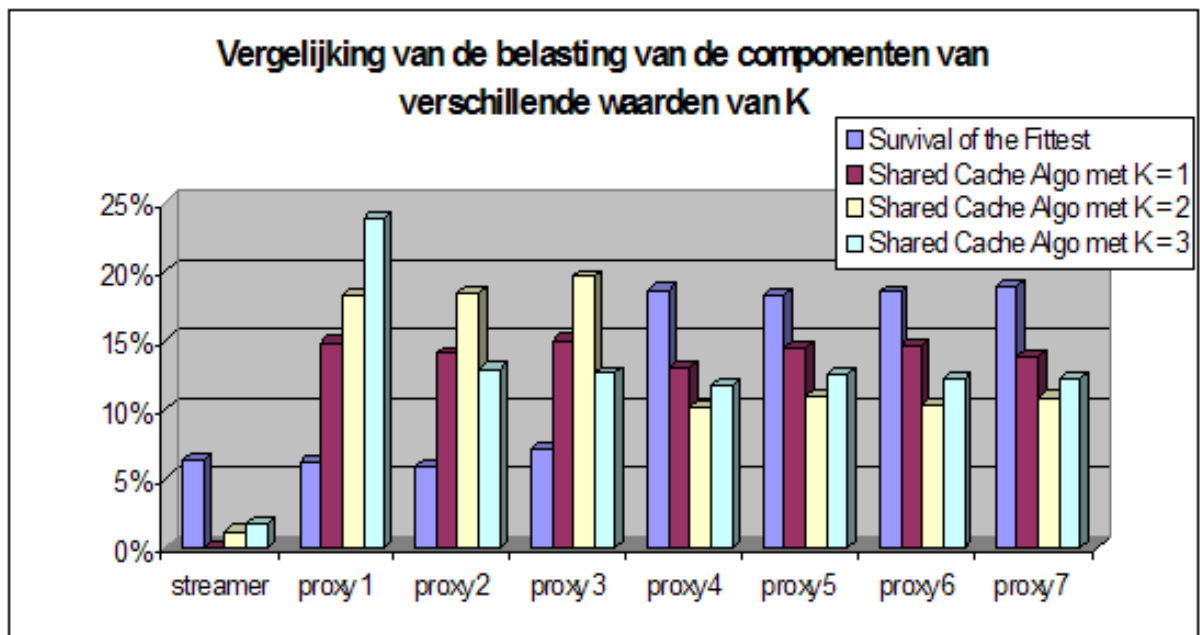
Het aantal caches in het netwerk is C

Dan zal de een cachegrootte per cache van G volstaan om altijd alle segmenten te kunnen opslaan in een cache, met:

$$G = \frac{ZMK}{C}$$

Vergelijkende simulaties

Bij deze simulaties gebruiken we de Shared Topo Algo doorstuurstrategie en hebben we de promiscuous mode aangezet. We gebruiken ook nog steeds dezelfde boom. We doen vier simulaties, één waarbij het cache algoritme survival of the fittest is, en één met het Shared Cache Algo met een K gelijk aan 1, 2 en 3. Om duidelijkere simulatieresultaten te bekomen hebben we de demandfile aangepast zodat beide televisiezenders nu enkel programma's uitzenden van 30 minuten. De cachegrootte staat nog steeds op 10, er zijn 7 caches in het netwerk dus dit staat ons toe om bij $K = 1$ alle segmenten in caches op te slaan, bij $K = 2$ en $K = 3$ zal dit niet meer mogelijk zijn.

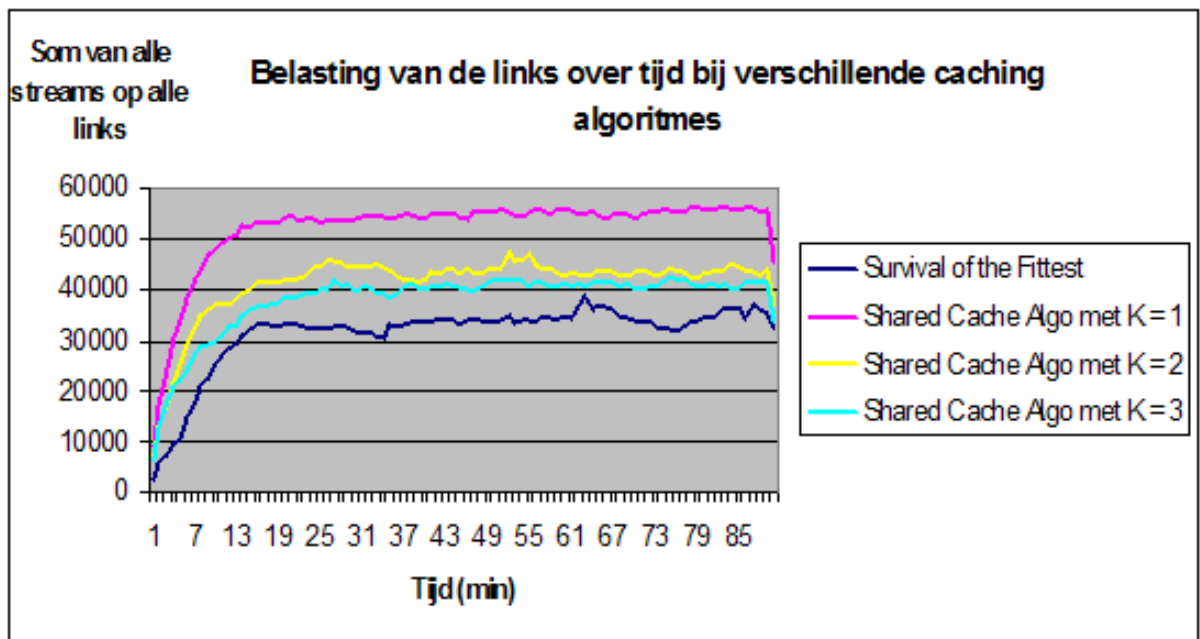


Figuur 5.7: Vergelijking verschillende waarden van K bij Shared Cache Algo

We merken op dat we de belasting van de streamer bijna helemaal kunnen weg nemen door over te schakelen van Survival of the Fittest naar Shared Cache Algo met $K = 1$, dit is logisch want als alle segmenten een plaats krijgen in één van de caches dan moet elk segment maar één keer gevraagd worden aan de streamer. Als we het aantal aanwezige kopieën verhogen tot 2 en 3 merken we dat de belasting van de streamer iedere keer met een kleine hoeveelheid toeneemt, maar wel nog steeds redelijk klein blijft in vergelijking met Survival of the Fittest. De reden hiervoor is dat zelfs bij dit caching algoritme van de meest recente segmenten 2 of 3 kopieën zullen aanwezig zijn, het tekort aan plaats in de caches zal ervoor zorgen dat oudere segmenten te vroeg verdwijnen maar zal er niet voor zorgen dat een live segment niet gecached wordt. De vorm van de demandcurves zal ervoor zorgen dat de schade eerder beperkt blijft.

Op de grafiek met de belasting van de links zien we dat bij Survival of the Fittest het minste verkeer is op het netwerk. Bij het Shared Cache Algo met $K = 1$ is er gedurende de hele simulatie veel meer belasting op de links. Hoe groter men de waarde van K kiest hoe minder verkeer op het netwerk, maar ook hoe meer belasting bij de streamer. Bij $K = \infty$ zullen we uiteindelijk het zelfde gedrag zien als bij Survival of the Fittest.

Een provider die kiest om gebruik te maken van het Shared Cache Algo zal een keuze moeten



Figuur 5.8: De belasting op van de links over de tijd bij Shared Cache Algo

maken voor K afhankelijk van de prijs van de links ten opzichte van de prijs van de opslagplaats in de caches.

Hoofdstuk 6

Conclusies

6.1 Conclusies

Als er één ding is wat men zeker moet onthouden uit deze scriptie dan is het wel dat caches in een transportnetwerk heel nuttig kunnen zijn, in alle delen van het netwerk, van het core-netwerk tot dicht bij de clients. Toch verdient deze scriptie een meer uitgebreide conclusie. Daarom geven we hier een samenvatting van de belangrijkste conclusies die we kunnen maken hebben in de loop van deze scriptie:

Het Prefix Algoritme is een belangrijke manier om de setuptijd van streams te verminderen maar het is helemaal onbruikbaar in de context van het verminderen van de belasting op streamer. De andere geziene algoritmen zijn wel geschikt om de netwerk belasting te verminderen. Als we de algoritmes sorteren volgens stijgende prestaties dan krijgen we: Sliding Interval algoritme met constante intervalgrootte, Sliding Interval algoritme met variabele intervalgrootte & Survival of the Fittest algoritme. De betere prestaties zijn het gevolg van meer flexibiliteit van de algoritmes.

Indien er geen enkele vorm van samenwerking is tussen de proxies dan zal de topologie waarmee wordt gewerkt geen invloed uitoefenen op de prestaties van die proxies.

In een boom waar caches aanwezig zijn op verschillende dieptes is een intelligente vorm van het doorsturen van de aanvragen strikt noodzakelijk opdat de proxies op kleinere diepte van een dergelijke boom nuttig zouden zijn.

Alle geziene algoritmes zullen beter presteren bij een groter wordende cache. Toch zal de prestatie stijging het sterkste zijn bij het Survival of the Fittest algoritme, beide Sliding Interval algoritmes zullen een prestatieverbetering ondervinden die relatief gezien minder groot is.

Bij een mooie exponentieel dalende demandcurve zullen algoritmes die gebruik maken van een ΔT het beste presteren met een zo klein mogelijke waarde voor ΔT . Een grote waarde voor ΔT is enkel te verantwoorden in situaties waar de demandcurve heel onvoorspelbaar gedrag vertoont. Extreem grote waarden van ΔT zal de werking van het gebruikte algoritme volledig lam leggen.

Indien een eenvoudige doorstuur strategie gebruikt wordt zal het gebruiken van een promiscuous mode leiden tot sterke prestatie verbeteringen. Bij het gebruik van meer ingewikkelde doorstuurstrategieën zal het gebruik van de promiscuous mode eerder leiden tot prestatie vermindering.

De grootte van de prestatie verbetering geïntroduceerd door de promiscuous mode zal in grote mate afhangen van het de flexibiliteit van het gebruikte caching algoritme. Hoe meer flexibel het caching algoritme hoe meer waardevol de promiscuous mode.

Van alle geziene Topologie Algoritmes zorgt het Shared Topo Algo voor de beste prestaties. Dit betekent dat het zoeken naar de cache met de meeste vrije plaats, de beste doorstuurstrategie is.

Als men wil kan men de streamer bijna volledig ontlasten door het Shared Cache Algoritme te gebruiken, hierdoor wordt ieder segment één of meerdere keren gecached in één van de caches. De kost die men betaalt voor deze verbetering is sterke stijging in netwerkverkeer op de links.

6.2 Toekomstig werk

We beëindigen deze scriptie met enkele suggesties over onderwerpen die in de lijn liggen van deze thesis die interessant zijn voor verder onderzoek:

Vooraf in de context van samenwerkende proxies ligt nog veel ruimte voor onderzoek. Men zou zich bijvoorbeeld kunnen bezig houden met het ontwikkelen van het protocol dat communicatie

tussen de proxies moet mogelijk maken. Op deze manier zou men kunnen onderzoeken hoeveel overhead deze samenwerking veroorzaakt en of het in de praktijk haalbaar is om alle proxies in het netwerk op de hoogte te houden van de inhoud van alle andere proxies.

Verder zou men met minimale aanpassingen aan de simulator kunnen onderzoeken in welke mate RTSP proxy servers nuttig zijn in meer algemene netwerken, en dus niet enkel in toegangsnetwerken van providers.

Een ander, heel interessant onderzoek zou één kunnen zijn naar de werking en het nut van peer-to-peer streaming tussen de verschillende settopboxes. Hierbij zouden settopboxes met interne caches onderling verbindingen kunnen opzetten om zo programmafragmenten uit te wisselen. Men zou zich kunnen inspireren op bestaande peer-to-peer file-sharing strategieën om dit te verwezelijken. Dit zou kunnen plaats vinden in een netwerk met of zonder RTSP proxy servers.

Bijlage A

De TsTV Simulator

In deze bijlage wordt extra uitleg gegeven over de werking van de simulator. De lezer van deze scriptie zal gemerkt hebben dat deze simulator veruit het belangrijkste hulpmiddel is geweest bij dit onderzoek. Het is dan ook wenselijk dat - indien men zelf een bepaalde situatie wil onderzoeken - men gebruik kan maken van deze simulator. In deze bijlage volgt een uitgebreid overzicht over hoe men simulaties kan uitvoeren, hoe men de simulator configureert en hoe men de gegenereerde output bestanden interpreteert.

A.1 Uitvoeren

De Simulator is geschreven in Java 1.5. Iedere PC met een java runtime environment recenter dan 1.5 volstaat dus om de simulator uit te voeren. Verder moet gezegd worden dat in bepaalde omstandigheden de simulator gedurende enkele seconden net na het starten heel veel geheugen verbruikt. In die omstandigheden is het nodig dat de gebruiker de maximale heapgrootte van de virtuele machine aanpast. Dit gebeurt indien men netwerken probeert te simuleren met extreem veel clients. De gebruiker kan de maximale heapgrootte verhogen tot bvb. 512 MB door java het argument “*-Xmx512m*” mee te geven bij uitvoeren van de simulator.

De uitvoerbare klasse van de simulator heet *Sim.class* en voor de gewone werking moeten minstens drie argumenten meegegeven worden. Dit zijn achtereenvolgens de namen van: het te gebruiken topologiebestand, het te gebruiken demandbestand & het te gebruiken configuratiebestand. Uitgebreide informatie over de inhoud van deze bestanden wordt hieronder gegeven. Normaal zal de simulator zijn outputbestanden weg schrijven in de directory “*Output*”, maar

indien de gebruiker een vierde argument mee geeft dan zal er een subdirectory gecreëerd worden in “*Output*” met die naam waarin dan de output bestanden komen. Deze feature werd toegevoegd om het mogelijk te maken verschillende simulaties na elkaar in batch uit te voeren zonder dat de output bestanden zichzelf zouden overschrijven. Die output subdirectory kan ook opgegeven worden in het configuratiebestand, maar daarover dus later meer.

De simulatie wordt gestart door:

```
java [-Xmx512m] Sim <topologiebestand> <demandbestand> <configuratiebestand> [<outputdir>]
```

A.2 Inputfiles maken

A.2.1 Het configuratiebestand

De belangrijkste van alle input bestanden is duidelijk het configuratiebestand. Het bevat informatie over de caching algoritmes en samenwerkingsvormen die de verschillende proxy servers moeten hanteren. Verder is hier ook de mogelijkheid om de outputdirectory en het type van inputbestand te specificeren. We zullen nu het configuratiebestand regel per regel overlopen.

De eerste twee regels van de inputfile worden gebruikt om te specificeren welke caching algoritme de proxy servers hanteren en hoe dat algoritme werkt.

cachealgo x

cachevar y

Hierbij zijn x & y gehele getallen. Cachealgo wordt gebruikt om het te gebruiken caching algoritme te specificeren en y heeft betekenis afhankelijk van het gekozen cachealgo. Hieronder geven we een overzicht van de geïmplementeerde caching algoritmes.

- (1) **First Come, First Serve algoritme:** De waarde van cachevar heeft geen belang.
- (2) **Sliding Interval met vaste grootte:** De waarde van cachevar heeft geen belang.
- (3) **Sliding Interval met variabele grootte:** De waarde van cachevar is hier ΔT . Dit is het tijdsinterval uitgedrukt in aantal minuten waarna de grootte van de intervallen van de verschillende televisiezenders herberekend wordt.

- (4) **Prefix algoritme:** De waarde van cachevar heeft geen belang.
- (5) **Survival of the Fittest algoritme:** De waarde van cachevar is hier ΔT . Dit is het tijdsinterval uitgedrukt in aantal minuten waarna de populariteit van ieder programmasegment herberekend wordt.
- (6) **Shared algoritme:** De waarde van cachevar is hier: het maximaal aantal toegelaten kopieën van een bepaalde segment die mogen aanwezig zijn in de verzameling van proxy servers die bevat zit in de bronnen verzameling van iedere proxy server.

De volgende regels worden gebruikt om te specificeren welk topologie algoritme de proxy servers gebruiken.

topoalgo x
topovar y
coefA c_a
coefB c_b
coefC c_c
coefD c_d

Hierbij zijn x & y gehele getallen. Topoalgo wordt gebruikt om het te gebruiken topologie algoritme te specificeren en y heeft betekenis afhankelijk van het gekozen topoalgo. De coëfficiënten zijn rationale getallen en hebben enkel een betekenis indien het “*Complex Shared Topo Algorithm*” gebruikt wordt. Hier volgt een overzicht van geïmplementeerde topologie algoritmes:

- (1) **Normal Topo Algoritme:** Indien er geen samenwerking is tussen de proxy servers.
- (2) **Shared Topo Algoritme:** De proxy servers hanteren een eenvoudige samenwerkingsvorm gebaseerd op de resterende vrije plaats in hun caches.
- (3) **Complex Shared Topo Algoritme:** De proxy servers hanteren een meer gecompliceerde samenwerkingsvorm gebaseerd op: de resterende vrije plaats in hun caches (A), hun belasting op dat moment (B), de lokaliteit van de proxy server ten opzichte van de dichtste streamer (C), de lokaliteit van de proxy server ten opzichte van de proxy server die de request doorverwijst (D). De relatieve grootte van c_a , c_b , c_c & c_d bepalen de belangrijkheid van achtereenvolgens (A), (B), (C) & (D).

tussenliggend *b*

De variabele “*tussenliggend*” kan de waarde 0 of 1 dragen en geeft aan of de interfaces van de proxy servers al dan niet in promiscuous mode gezet kunnen worden. Dit wil zeggen, als een proxy server dat wil kan een passerende stream gecaptured en gecached worden.

outputdir *directory*

De variabele “*directory*” kan een naam van een directory dragen die door de simulator binnen in “*Output*” gemaakt wordt, waarin de output files terecht komen. Deze output directory kan ook direct in de commandline mee gegeven worden. Indien via beide verschillende directories ingevuld werden, dan zal die uit de inputfile gebruikt worden.

soortdemand *b*

De variabele “*soortdemand*” kan de waarde 0 of 1 dragen en geeft aan welk type demandbestand gebruikt wordt. Deze variabele zal bij normaal gebruik van de simulator waarde “1” hebben. Meer informatie hierover volgt.

A.2.2 Het topologiebestand

Het topologiebestand is het bestand waarin het netwerk wordt beschreven dat moet worden gesimuleerd. Een netwerk bestaat typisch uit: clients, streamers, proxies, nodes & links. Elk van deze componenten worden beschreven door één regel in het topologiebestand. Hieronder volgt kort een overzicht van elk van deze regels.

SERVER[*naam*]

NODE[*naam*]

Server (dit is de streamer) en node (dit is een gewone router, zonder caching capaciteiten) hebben enkel een naam nodig. Deze is uniek over alle componenten!

PROXY[*naam, capaciteit, #ALL#*]

PROXY[*naam, capaciteit, bron1, bron2, ...*]

Een proxy heeft drie of meerdere attributen nodig. De naam is uniek over alle componenten. De capaciteit is een geheel getal dat de cachegrootte uitdrukt in aantal minuten opgeslagen video. De derde en volgende variabelen geven de bronnen van deze proxy. Dit zijn de namen van streamers of andere proxyservers die deze proxy server mag aanspreken indien hij een bepaald fragment niet lokaal kan bedienen. Een makkelijkere manier om te schrijven dat een bepaalde proxy alle andere proxies en alle streamers in het netwerk mag gebruiken als bron is door “#ALL#” te schrijven. “#ALL#” mag uiteraard niet gebruikt worden als naam van een component.

CLIENT[*naam, bron*]

Een client heeft twee attributen nodig. De eerste is zijn naam die uniek is over alle componenten. De tweede is de bron van deze clients. In tegenstelling tot een proxyserver wordt een client “dom” geacht. Hij heeft dan ook maar één bron waar hij al zijn requests naar stuurt.

LINK[*naam, van, naar, gewicht-heen, gewicht-terug, capaciteit-heen, capaciteit-terug*]

Een link heeft steeds zeven attributen nodig. De eerst is de naam, die uniek is over alle componenten. De volgende twee attributen zijn de namen van de twee componenten die deze link verbinden. De attributen die daarop volgen zijn de gewichten van de link in de ene en de andere richting. De laatste twee attributen geven de capaciteit van die link aan in beide richtingen. De capaciteit heeft echter nog geen betekenis gekregen in de simulator, en mag dus een willekeurige waarde hebben.

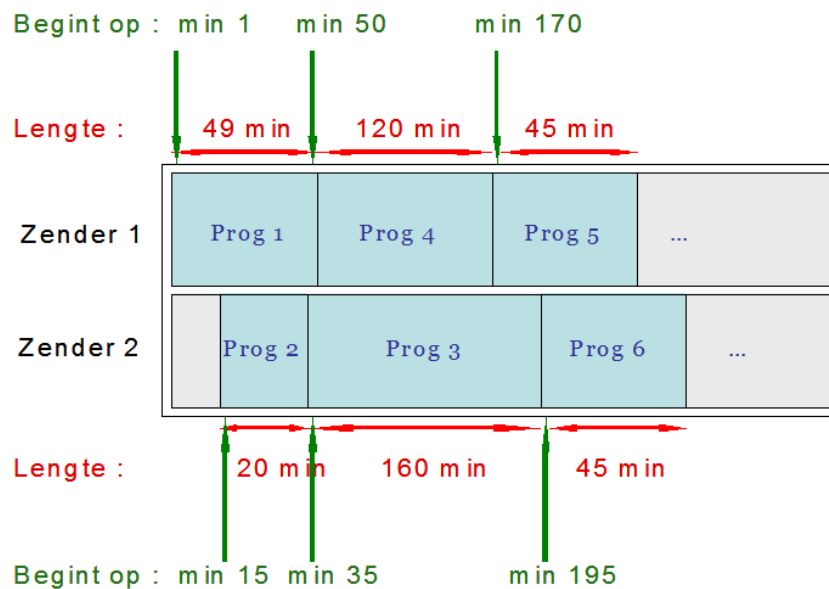
Het is belangrijk om te wijzen op het belang van de volgorde van de regels in het topologiebestand. De componenten worden aangemaakt in volgorde dat ze gedefinieerd worden in dit bestand en dit leidt tot ongewenst gedrag indien de regels in een onlogische volgorde staan. Bijvoorbeeld: een client heeft een bron die pas de regel daaronder gedefinieerd wordt.

A.2.3 Het demandbestand

Zoals gezegd zijn er eigenlijk twee types van demandbestanden, waarvan slechts één veelvuldig werd gebruikt in onze simulaties. Dit is het type waarbij in het bestand wordt aangegeven welke de populariteit is van de verschillende programmas, maar de aanvragen door de simulator zelf willekeurig verdeeld worden over de clients. Indien men voor een bepaalde simulatie die verdeling zelf wil specificeren (een bepaald programma is enkel populair in een bepaalde regio van het land. Vb. regionale televisie) kan men dat doen via dat tweede type.

Definiëren van de programmas

Op dit moment moet het aantal programmas, hun begintijdstip, hun lengte en tot welke zender ze behoren gedefinieerd worden. Dit gebeurt aan de hand van een B-, S- & Z-regel, die altijd in die volgorde bovenaan het demandbestand moeten staan. Aan de hand van het onderstaande voorbeeld zullen we illustreren hoe die regels opgebouwd worden.



Figuur A.1: Twee televisiezenders zenden elk 3 programma uit

1. Noteer van elk programma het begintijdstip en de lengte van het programma uitgedrukt in minuten. (Merk op dat het programma dat het eerste begint steeds op minuut 1 begint en niet op minuut 0)
2. Noteer van elk programma tot welke zender het behoort door de zenders te nummeren. De nummering begint vanaf 1. Het is de verantwoordelijkheid van de schrijver van het

demandbestand dat programmas van een bepaalde zender elkaar niet overlappen. Programmas op dezelfde zender hoeven niet per se aan te sluiten.

3. Sorteert de programmas volgens stijgend begintijdstip.
4. Bouw regel B op door in die volgorde de lengte van de programmas op te schrijven
5. Bouw regel S op door in die volgorde de begintijdstippen van de programmas op te schrijven
6. Bouw regel Z op door in die volgorde de nummer van de zender van de programmas op te schrijven
7. Vooraan regel B wordt het aantal programmas geschreven

De tekening zou onderstaande regels geven:

```
B 6 49 20 160 120 45 45
S 1 15 35 50 170 195
Z 1 2 2 1 1 2
```

Demandverdeling over de clients gebeurt random

Dit geval is er nood aan een P-regel in het bestand. Deze regel beschrijft de populariteit van de verschillende programmas. De kijkers van ieder programma worden dan verdeeld over de clients door de simulator zelf.

Als we verder gaan op het voorbeeld van de sectie hierboven en stellen dat zender 1 die avond dubbel zo veel kijkers heeft als zender 2 dan zou de P-regel er als volgt uit kunnen zien.

```
P 3000 1500 1500 3000 3000 1500
```

Demandverdeling over de clients gebeurt manueel

In dit geval gebeurt de verdeling dus manueel. De schrijver van het demandbestand moet specificeren voor iedere client hoeveel kijkers er zijn voor elk programma. Dit gebeurt in het bestand via verschillende N-regels, één per programma. (Herinner dat programmas vanaf 1 genummerd worden). “N” wordt gevolgd door het programmanummer gevolgd door verschillende duo’s van clientnamen en het aantal aanvragen van die client voor dat programma. Moest de werkwijze nog steeds niet duidelijk zijn, hier volgt een mogelijk voorbeeld van de N-regels voor een topologie met vier clients.

```
N 1 client1 1000 client2 500 client3 800 client4 700
N 2 client1 500 client2 200 client3 700 client4 100
N 3 client1 500 client2 200 client3 700 client4 100
N 4 client1 1000 client2 500 client3 800 client4 700
N 5 client1 1000 client2 500 client3 800 client4 700
N 6 client1 500 client2 200 client3 700 client4 100
```

Voor grote topologieën kan dit enorm lastig worden, vandaar het belang van random verdeelde demand.

De demandcurve instellen

Het feit dat de demandcurve exponentieel daalt werd uitgebreid besproken in deze thesis. Het tempo waarmee die daalt is van groot belang is voor de werking van de gesimuleerde algoritmen. Vandaar dat dit instelbaar is in de simulator. Dit wordt gedaan aan de hand van één enkele variabele genaamd C. Meer informatie over de betekenis van deze variabele is te vinden in Hoofdstuk 3.

A.3 Output files lezen

We gaan niet veel tijd besteden aan het overlopen van de output files omdat hun structuur meestal zo eenvoudig is. We zullen per type heel kort wat uitleg geven met een voorbeeldje.

A.3.1 De clients

De output bestanden zijn alle bestanden wiens naam begint met de prefix “cl-”. Voor iedere client zijn er twee output files genaamd: “cl-clientnaam-abs.txt” & “cl-clientnaam-rel.txt”. Deze tonen in absolute of relatieve cijfers door welke proxy of streamer de requests afgehandeld zijn geweest. Bovendien bestaat er ook nog een “cl-totaal-abs.txt” & “cl-totaal-rel.txt” waarin de gesomeerde gegevens van alle clients staan.

<i>prognr</i>	<i>strea</i>	<i>proxyA</i>	<i>proxyB</i>	<i>proxyC</i>	<i>proxyD</i>	<i>proxyE</i>	<i>proxyF</i>	<i>totaal</i>
0	11334	2356	2190	948	810	882	740	19260
1	6600	40	20	3360	2720	2720	3000	18460
2	64980	5652	4644	5562	3832	6186	4114	94970
3	13700	40	20	6926	7754	6182	7318	41940
4	20894	60	30	6987	8074	5769	7317	49131
5	1610	732	208	448	740	660	802	5200
<i>tot</i>	119118	8880	7112	24231	23930	22399	23291	228961

Tabel A.1: Een client output bestand

A.3.2 De proxies

Voor ieder proxy wordt een output bestand gegenereerd met de naam “pr-proxynaam.txt” waarin afgelezen kan worden hoeveel binnenkomende request konden afgehandeld worden uit de locale cache (Local), afgehandeld worden door die request door te sturen naar een andere proxy of een streamer (Remote) en hoeveel van die requests helemaal niet afgehandeld konden worden (NoFo). Verder wordt er per minuut en per categorie een totaal berekend. Er wordt ook één bestand gemaakt met naam “pr-totaal.txt” waarin de gegevens van alle proxy servers in opgeteld wordt.

<i>tijd</i>	<i>NoFo</i>	<i>Local</i>	<i>Remote</i>	<i>Totaal</i>
1	0	0	18	18
2	0	0	34	34
3	0	99	70	169
4	0	88	115	203
5	0	76	153	229
6	0	264	147	411
7	0	219	180	399
8	0	181	208	389
9	0	347	192	539
10	0	283	218	501
<i>tot</i>	0	1557	1335	2892

Tabel A.2: Een proxy output bestand

A.3.3 De links

Logisch gezien is er voor iedere link die ingegeven is het topologiebestand eigenlijk twee links, één in elke richting. Voor iedere link wordt er dan ook twee output bestanden aangemaakt met de namen. “li-linknaam-heen.txt” & “li-linknaam-terug.txt”. In zo’n bestand staat dan per minuut hoeveel streams er liepen over die link in die richting.

<i>tijd</i>	<i>data</i>
1	16
2	30
3	75
4	112
5	143
6	169
7	190
8	207
9	221
10	232

Tabel A.3: Een link output bestand

A.3.4 De streamers

Voor ieder streamer wordt een output bestand gemaakt met als naam “se-streamernaam.txt”. Daarin staat per minuut hoeveel requests die streamer heeft afgehandeld. Er wordt ook een totaal gemaakt over de hele duurtijd van de simulatie.

<i>tijd</i>	<i>data</i>
1	1
2	1
3	2
4	2
5	2
6	2
7	2
8	2
9	2
10	2
<i>tot</i>	18

Tabel A.4: Een streamer output bestand

Bibliografie

- [1] S. Chen, Y. Yan, S. Basu, X. Zhang, “SRB: Shared Running Buffers in Proxy to Exploit Memory Locality of Multiple Streaming Media Sessions”, in *Proceedings of the 24th International Conference on Distributed Computing Systems*, 2004.
- [2] S. Gruber, J. Rexford, A. Basso, “Design Considerations for an RTSP-Based Prefix-Caching Proxy for Multimedia Streams”, in *AT&T Labs - Research*, September 1999.
- [3] J. Liu, J. Xu, “Proxy Caching for Media Streaming over the Internet”, *IEEE Communications, Feature Topic on Proxy Support for Streaming on the Internet*, August 2004.
- [4] T. Wauters, W. Van de Meersche, F. De Turck, Bart Dhoedt, P. Demeester, T. Van Caenegem, E. Six, “Co-operative Proxy Caching Algorithms for Time-Shifted IPTV Services”, *32nd EUROMICRO CONFERENCE, Track on “Multimedia & Telecommunications: Dependable Adaptive Systems”*, Cavtat/Dubrovnik, Croatia - Aug. 28th - Sept. 1st, 2006
- [5] M. Strobbe, V. Verstraete, “MPEG”, *IBCNWiki*
- [6] JGraphT <http://jgrapht.sourceforge.net>

Lijst van figuren

2.1	Een typische demandcurve	7
2.2	Kijkers curve afgeleid van de demandcurve	7
2.3	Een transportnetwerk	9
2.4	Settopboxes met caching capaciteiten	10
2.5	Een boom met caches in tussenliggende nodes	10
2.6	Een mogelijke implementatie van een RTSP cache zoals die geïmplementeerd is binnen de onderzoeksgroep	11
2.7	De logische architectuur van een RTSP Proxy Server	12
2.8	Beslissingsboom 1 van een RTSP Proxy Server	13
2.9	Beslissingsboom 2 van een RTSP Proxy Server	13
2.10	Beslissingsboom 3 van een RTSP Proxy Server	14
2.11	De stream neemt een andere weg als de aanvraag	15
2.12	Twee streams vs. Eén stream, één aanvraag	16
4.1	Demandcurve met interval	26
4.2	Een gebruiker bevindt zich in het venster en pauzeert	27
4.3	De drie gesimuleerde topologieën	38
4.4	Vergelijking belasting in een boom met proxies dicht bij de clients	39
4.5	Vergelijking van de belasting van de componenten in een boom vol proxies	40
4.6	Vergelijking belasting in een ring	41
4.7	Prefix algo met variërende cachegroottes	42
4.8	Sliding interval met vaste intervalgrootte met variabele cachegroottes	43
4.9	Sliding interval met variabele intervalgrootte met variabele cachegroottes	43
4.10	Sliding interval met variabele intervalgrootte met variabele Delta T	44
4.11	Survival of the Fittest met variabele cachegroottes	45

4.12	Survival of the Fittest algoritme met variabele Delta T	46
5.1	Tussenliggende proxies met promiscuous mode	50
5.2	Vergelijking belasting met en zonder promiscuous mode (SoF)	51
5.3	Vergelijking belasting met en zonder promiscuous mode (Static interval)	52
5.4	Vergelijking van verschillende doorstuur strategieën	56
5.5	De werking van het Complex Topo Algo doorgelicht	57
5.6	De belasting op van de links over de tijd bij Complex Topo Algo	58
5.7	Vergelijking verschillende waarden van K bij Shared Cache Algo	60
5.8	De belasting op van de links over de tijd bij Shared Cache Algo	61
A.1	Twee televisiezenders zenden elk 3 programma uit	70

Lijst van tabellen

A.1 Een client output bestand	73
A.2 Een proxy output bestand	74
A.3 Een link output bestand	75
A.4 Een streamer output bestand	76