Om att hitta viskande elefanter bland skrikande möss

Identifiering av Peer-to-Peer med hjälp av dataströmmar

Jonas Kalderstam

Examensarbete för 30hp
Institutionen för datavetenskap, Naturvetenskapliga fakulteten, Lunds universitet

Thesis for a diploma in computer science, 30 credit points,
Department of Computer Science,
Faculty of Science, Lund University

Om att hitta viskande elefanter bland skrikande möss

Identifiering av Peer-to-Peer med hjälp av dataströmmar

Sammanfattning

I takt med ett ständigt ökande antal bredbandsuppkopplingar och krav på högre bandbredd står Internetleverantörerna inför massiva investeringar om inte Internet ska få slut på kapacitet. Samtidigt har användarna gått från att primärt ladda ner webbsidor eller mejl till att både ladda ner och upp media så som video och ljud. På grund av detta vänder sig flera leverantörer till trafikprioritering för att förbättra sin Quality of Service. I denna uppsats analyserar jag svårigheterna med att identifiera Peer-to-Peer-fildelning i realtid och föreslår en metod baserad på dataströmsmodellen. Med hjälp av en algoritm baserad på bloomfilter så lyckas jag visa på en klar skillnad mellan det genomsnittliga antalet flöden per sekund av P2P-fildelning och andra typer av trafik så som webbsurfning.

Finding Whispering Elephants among Screaming Mice

Identifying Peer-to-Peer with Data Streams

Abstract

With the ever increasing number of broadband connections and demand for higher bandwidth Internet Service Providers are facing massive investments or the Internet might soon run out of capacity. At the same time users have switched from mainly downloading web or mail content to both download and upload media in the form of video and audio. In the light of this several providers are turning to traffic shaping to improve their Quality of Service. In this paper I analyse the difficulties of identifying Peer-to-Peer filesharing in real time and propose a method based on the data stream model. Using an algorithm based on bloom filters I am able to show a clear difference of the average number of flows per second between P2P filesharing and other traffic types such as web browsing.

Innehållsförteckning

Tack

Thore Husfeldt, som stod för den ursprungliga idén till att genomföra ett arbete baserat på dataströmmar.

Maria, för din ovärderliga granskningshjälp.

Tobias, för din assistans vid implementationen och datainsamlingen.

Slutligen tack till Åsa, Magnus och Eva för kommentarer och synpunkter.

1. Inledning

Internet har under sin korta livstid snabbt utvecklats till ett fundamentalt verktyg för utbyte av information. Hastigheten på användarnas uppkopplingar har stadigt ökat från 14,4 Kb/s till dagens hastigheter på mellan 10 Mb/s och 100 Mb/s. Detta har tvingat Internetleverantörer att öka sin egen kapacitet för att tillgodose kundernas behov. De senaste nio åren har dock medfört en revolution av informationsdistribution i och med Peer-to-Peer (P2P). När användare en gång nästan uteslutande laddade ner material, främst i form av text och bilder, laddar de nu i lika stor grad upp som de laddar ner. Samtidigt har typen av data ändrats till att idag även inkludera stora mediafiler som film och musik. För att hantera denna explosion av data tvingas Internetleverantörerna att prioritera vissa typer av trafik. I vissa fall förbjuds P2P helt. Fildelare svarar genom att kryptera sin trafik och skapa förbindelser över portar som inte följer den vedertagna standarden. Även om en leverantör inte har för avsikt att begränsa fildelning inom sitt nät så har de fortfarande ett intresse av att identifiera P2P-trafik för att kunna garantera Quality of Service på centrala tjänster som IP-Telefoni. Men att kunna identifiera P2P-trafik som är dold för tidigare metoder i realtid i de hastigheter som leverantörers routrar arbetar på är en utmaning.

Utvecklingar i flera områden, i synnerhet datorkommunikation och databaser, har skapat ett behov av ett nyanserat synsätt på stora datamängder. Genom att betrakta datan som en ström av information blir det naturligt att skapa algoritmer med O(n) operationer och O(log n) minnesanvänding. Algoritmer med en O(n log n) komplexitet anses normalt inte mycket sämre än en O(n) algoritm, eftersom logaritmen är en så pass långsamt växande funktion. I dataströmssammanhang är det däremot inte ovanligt att log(n) snabbt växer bortom 20. En potentiellt tjugofaldig prestandaförsämring är inget som kan bortses ifrån [20].

Med ett synsätt grundat i dataströmsmodellen undersöker jag i denna uppsats svårigheterna med att identifiera P2P, oberoende av dess protokoll, i realtid. Det är inte min avsikt att identifiera potentiella brott mot upphovsrättslagen. Jag förutsätter i min analys att ingen skillnad mellan tillåten och förbjuden fildelning kan observeras.

Uppsatsen är indelad i två delar. I den första inleder jag med att först redogöra för dataströmsmodellen och TCP/IP-trafik. I kapitel 2.4. introduceras de randomiserade datastrukturer, ”bloomfilter”, som används för att bearbeta dataströmmen. Kapitel 2.5. till 2.6. behandlar P2P och trafikprioritering. I den andra delen går jag först igenom algoritmen som jag utvecklat i kapitel 3.1. tillsammans med förslag på möjliga förbättringar. Slutligen

behandlas tillvägagångssättet och resultat av testkörningarna i kapitel 3.2., 3.3. och 3.4.

1.1. Problemformulering och avgränsning

Syftet med denna uppsats är att undersöka svårigheterna med att identifiera P2P-trafik i realtid, att utveckla en algoritm som potentiellt kan implementeras i SRAM samt att med hjälp av nämnda algoritm utföra experimentella försök för att kontrollera att den lyckas med att identifiera P2P samtidigt som den med låg sannolikhet felklassificerar annan trafik.

Eftersom filöverföringar nästan uteslutande sker med TCP så har jag valt att endast titta på TCP-trafik även om min implementation är ganska enkelt utbyggbar för UDP och alla andra möjliga protokoll. Algoritmen i sig förutsätter inget om protokollet. Det beror bara på hur man definierar ett flöde.

Jag förväntar mig att snabb P2P-trafik kommer att ge upphov till många flöden, medan annan trafik så som snabb http-trafik inte kommer det. Det är min misstanke att även långsammare P2P-trafik genererar fler flöden än annan trafik.

2. Del 1: Rosa och vita elefanter.

Del 1 går igenom motiveringarna bakom uppsatsens syfte och förutsättningarna för att förstå del 2. Den förklarar och definierar koncept så som dataströmmar, Peer-to-Peer och bloomfilter.

2.1. Massiva datamängder och snabba dataströmmar

Information betraktas ofta som en statisk klump, speciellt när den huseras i form av databaser. Det finns tillfällen då ett sådant synsätt begränsas av vad som är praktiskt möjligt att utföra. De två främsta exemplen är dels då mängden data är enorm och dels då datan strömmar in i en hög hastighet där informationen mycket snabbt ska tas om hand, bearbetas och skickas vidare eller för evigt falla i glömska. Vid dessa fall betraktas informationen med fördel som en (potentiellt oändlig) ström. Även om datorströmmar kommer att behandlas utifrån hur de relaterar till datornätverk i största delen av denna uppsats, så kommer jag börja med att titta på deras generella representation.

En dataström kan betraktas som en mängd mindre, potentiellt oändliga, flöden. Hur man definierar vad som skiljer ett flöde från ett annat beror helt på problemet man är intresserad av. Med avseende för datatrafik så kanske man vid en tidpunkt definierar ett flöde som TCP/IP-paket mellan IPadress1/Port1 och IPadress2/Port2, medan man vid en annan definierar det som UDP-paket mellan Dator1 och Dator2. Om vi istället jobbar med ett problem relaterat till diskläsning kanske vi definierar ett flöde som bytes tillhörande filen Z och hela strömmen blir då lika med alla filer på disken. Varje flöde består i sig av element, som är den enhet som bearbetas t.ex. TCP/IP-paket eller bytes från disk. Per definition, kan en dataström vara vilken datamängd som helst. Termen används dock inte för vad som helst. En mängd information, oavsett om den är lagrad på disk, om den överförs över nätverk eller om den kan sägas ha någon annan mer påhittig existensform, betraktas normalt som en dataström först när den är så stor och/eller snabb (i fallet med snabb överföring) vilket kräver att algoritmerna som behandlar strömmen är snabba och ”små”.

Ett grundkrav är att algoritmerna endast läser varje element i dataströmmen en enda gång[1]. Dessutom förväntas det att mängden minne som används är litet i förhållande till vad som läses. Alltså kan vi inte spara en kopia av varje element för framtida läsning. Som ett kort exempel kan vi betrakta följande. Nuförtiden är det vanligt att det under TV-nyheterna rullar

1 För vissa problem kan ett fåtal gånger vara acceptabelt.

en textremsa längst ner i bild som kort berättar om mindre nyheter, valutaförändringar och aktievärden. För att veta när man säkert kan sluta läsa texten och koncentrera sig på de riktiga nyhetsrapporteringarna så krävs det att man kan identifiera när de första nyheterna man läste kommer igen (förutsatt att texten rullar mer än ett varv). Alltså måste man identifiera *kopior*. Textremsorna är korta nog så att detta inte är något problem för de flesta människor, men tänk för ett ögonblick om längden av textremsan var enorm, säg i storleksordning av gemene mans lokala bibliotek. För att säkert identifiera dubbletter krävs det att man minns de nyheter som man läst innan, eller åtminstone en sammanfattning av dem. Om vi dessutom antar att nyheterna på textremsan skrivs ut i en slumpmässig följd så kan vi inte heller koncentrera oss på ett fåtal nyheter att komma ihåg (dvs. den allra första nyheten vi läste).

Som en lätt modifikation kan vi istället låta hastigheten på remsan vara snabb. Att se varje ord är inget problem (bara fixera ögonen på en punkt), men läsning handlar inte bara om att se bokstäver. Om remsan rullar snabbt nog kanske vi inte hinner med att bearbeta varje ord innan vi börjar få problem med att hinna läsa nästa ord. Även om vi skulle hinna med att läsa alla ord är det troligt att om någon skulle fråga oss om vad vi läst skulle vi inte klara av att svara på särskilt ingående frågor om innehållet. Samma problem står datorerna inför fast när informationen är betydligt större och givetvis snabbare.

2.1.1. Olika typer av fönster

Ett sätt att tackla problemet är att endast koncentrera sig på en liten del av strömmen åt gången. Om strömmen är data som samlas in i realtid kanske det är lämpligt att bara betrakta de senaste X elementen, eller elementen som förekommit under de senaste Y minuterna. Det brukar jämföras med att titta på strömmen genom ett litet fönster, där endast en del av strömmen får plats. För elementen de senaste Y minuterna kan vi implementera det på några olika sätt. Det mest uppenbara är att var Y:te minut glömma allt vi vet och i stort sätt börja om från början, något som kallas *Landmark*-fönster [8]. Den konkreta fördelen är att en sådan lösning är enkel att implementera. Nackdelen är det kan ge upphov till olika typer av fel eller osäkerheter beroende på vad vi letar efter i strömmen. Om vi är intresserade av en händelse som är utspridd över ett mindre tidsintervall, så är det möjligt att den inte upptäckts om den inträffar i slutet av ett fönster. I det fallet skärs händelsen av på mitten och vi lyckas kanske inte upptäcka händelsen i något av de intilliggande fönstren.

Ett annat alternativ är att endast glömma element som faktiskt är äldre än Y minuter, istället för att som i *Landmark*-fönster glömma allt som inträffat innan en specifik tidpunkt.

Vi eliminerar felen med fönsterkanter men introducerar istället en annan svårighet. Nu tvingas vi komma ihåg hur gamla enskilda element är så att vi vet när vi ska ta bort dem. Vi måste även vara försiktiga så att vi inte tvingas iterera över en lista om och om igen i jakt på gamla element. Detta är vad som kallas för glidande fönster [8].

Det sista alternativet är vad som kallas för hoppande fönster [9]. Vi utgår ifrån *Landmark*-fönster och delar upp fönstren i mindre subfönster. Medan vi fyller ett nytt subfönster, tar vi bort det äldsta. Själva analysen sker på fönstren emellan. Det är en kompromiss av *Landmark*-fönsters simpelhet och precisionen av de glidande fönstren.

2.2. Datornätverk och TCP/IP

Ett nätverk kan konstrueras på ett oändligt antal sätt. Det normalt mest använda sättet innebär att man använder olika TP-kablar (*twisted pair*) för att koppla samman datorer med växlar och routrar vilka i sin tur kopplas samman med andra datorer. Detta är långt ifrån det enda sätt som används dock. Det finns flera olika typer av TP-kablar och varje sådan typ kan behöva en mer eller mindre unik implementation. En annan väl använd metod för att bygga nätverk är att använda 802.11/a/b/g/n trådlösa basstationer och nätverkskort. Andra metoder inkluderar bland annat telefonledningar (DSL, ISDN), Bluetooth, satellitbaserade radiosignaler och mycket mer. Med tillräckligt mycket tid och besvär skulle man kunna konstruera ett nätverk med hjälp av pappmuggar och snöre. Poängen jag vill förmedla är att nästan alla dessa olika sätt kräver olika implementeringar i datorns mjukvara för att fungera som datornätverk. Att skicka en radiosignal är distinkt annorlunda från att skicka elektriska impulser (eller vibrationer via trådar) via kablar. Utvecklare världen över är evigt tacksamma att de oftast aldrig måste fundera över vilket medium nätverket är uppbyggt av. Detta eftersom processen att upprätta en kontakt hanteras på en lägre nivå än den som de flesta skriver sina program på. Normalt sätt är det operativsystemet som hanterar detta.

En nätverksförbindelse är uppdelad på sju nivåer enligt ISO OSI-referensmodellen (*Open Systems Interconnection*). Dessa är som följer, från botten och upp: den fysiska nivån, datalänknivån, nätverksnivån, transportnivån, sessionsnivån, presentationsnivån och applikationsnivån. Jag kommer inte intressera mig för något annat än transportnivån i denna uppsats. Om de lägre nivåerna nämner jag bara att de tillhandahåller möjligheten för transportnivån att skicka data över en förbindelse (ner till den faktiska omvandlingen till elektriska/optiska/vibrerande signaler beroende på vilket medium vi använde för att bygga vårt nätverk). De övre nivåerna tar hand om den faktiska datan som ska överföras. Det skiljer

sig från applikation till applikation. Eftersom att jag förutsätter att denna information ändå är krypterad och oläslig så ignorerar jag den. Skulle läsaren vara intresserad av att läsa mer så är Tanenbaums Computer Networks [3] en utmärkt startpunkt.

På transportnivån, å andra sidan, existerar de protokoll som konverterar den data som ska skickas till (för en dator) förståeliga bitströmmar. Det finns flera vanligt använda protokoll. Transport Control Protocol (TCP) är en. User Datagram Protocol (UDP), Realtime Transfer Protocol (RTP) och Realtime Control Transfer Protocol (RCTP) är andra. TCP används ofta när en pålitlig överföring av data är det viktiga, så som när man skickar filer över Internet. Det erbjuder garanterad leverans (garanterad på så sätt att datorn kommer att fortsätta försöka skicka data som inte kommit fram så länge den kan) men data som förlorats på vägen kan sakta ner överföringen. UDP erbjuder ingen felkorrigering (dvs. inte garanterad leverans, datorn skickar data endast en gång) vilket betyder att data kan gå förlorad på vägen men UDP är generellt snabb istället. Det används vanligen inom multimedia och onlinespel där några förlorade paket inte är hela världen så länge som överföringen fortlöper stadigt. RTP och RTCP används vanligen inom multimedia på grund av deras möjligheter att skapa en överföring mot flera mottagare. Jag tittar bara närmare på TCP eftersom det är vad nästan alla P2P-applikationer använder för att skicka filer.

Anledningen till att ”garanterad leverans” inte är något självklart är för att när en dator ska skicka information till en annan dator så passerar informationen genom en stor mängd andra maskiner innan den når sin slutdestination. Där någon av maskinerna kanske försvinner från nätverket mitt i överföringen. Att informationen passerar genom andra maskiner är en naturlig konsekvens av att datorer normalt inte har en dedikerad nätverkssladd till alla andra datorer. Fram till slutet av 80-talet hade varje dator i stort sätt en komplett karta över hela nätverket och visste därmed precis vilken väg den skulle skicka informationen för att den skulle komma fram snabbt och säkert. I takt med att nätverken växte och trafiken ökade blev det en ohållbar metod [20]. För att lösa problemet utvecklades några nya protokoll vilket innebar att istället hade varje maskin endast en lokal karta över näraliggande grannar. Denna karta är alltid[2] liten och förändringar kan göras snabbt och enkelt, utan att det påverkar nätverket generellt. Routrar gör uppdateringar i sin karta (eller snarare lista) beroende på tidsfördröjningen till dess grannar för att kunna välja bästa möjliga väg till slutdestinationen för de paket som passerar igenom. Effekten detta har är att paket som tillhör samma

2 Det är en ständig kamp att hålla listan liten. Den har en tendens att växa när man försöker minimera slöseriet med IP-adresser som oundvikligen förekommer på grund av implementationen av nätverk kontra adresser.

överföring mycket väl kan ta olika vägar till destinationen och, beroende på de fördröjningar som råder, anlända i en annan ordning än den de skickades i. Det är upp till destinationsdatorn att pussla ihop paketen i rätt ordning igen.

2.2.1. Transportnivån

När två datorer, eller snarare två applikationer, vill kommunicera med varandra över TCP måste de etablera en förbindelse med varandra. Två olika applikationer som båda kommunicerar över nätverket från samma maskin skiljer sig åt genom att använda olika portar. En port definieras av ett, för maskinen, unikt 16-bitars nummer. En förbindelse upprättas således mellan två portar, som kan vara lokaliserade på olika maskiner eller inte – det spelar ingen roll. Data kan skickas i båda riktningar över förbindelsen. Varje enskild port kan dock vara kopplad till flera förbindelser. Enskilda pakets destination definieras av deras relaterade par av portar (källa och destination). Men hur vet då en applikation vilken port den ska använda om den vill kontakta en annan maskin? Om maskinen inte lyssnar på porten i fråga kommer den bara att förkasta den data som skickas till den. För att underlätta kommunikationen mellan främmande maskiner har populära tjänster, som e-post osv. tilldelats ett portnummer. Alltså kan man vara säker på att en publik tjänst alltid lyssnar på samma portnummer, om inget annat är sagt. En webbserver ligger och lyssnar på port 80 nästan alltid. När en klient vill koppla upp sig mot servern så öppnar den först själv en godtycklig[3] port och skickar sedan en förfrågan till serverns port 80 om att få skapa en förbindelse. När en förbindelse sedan är etablerad tar protokollet datan som ska skickas och delar upp den i mindre delar (refererade till som *paket*), ofta i storlek av 1500 bytes eftersom detta nästan alltid är den största tillåtna överföringsenhetsstorleken (Maximum Transmission Unit, MTU) för datalänknivån (det vill säga Ethernet). Det kan vara så stort som 64 kB enligt TCP-specifikationen. Ett TCP-paket består av två delar: en huvuddel och en datadel. Huvudet innehåller information angående portnummer, sekvensnummer osv.

3 Detta är ofta ett högt nummer eftersom kända tjänster har fått standardiserade portnummer som börjar på 1 och uppåt.

Bit offset	0-3	4-7	8-15	16-32
0	Source port			Destination port
32	Sequence number			
64	Acknowledgment number			
96	Data offset	Reserved	Flags	Window
128	Checksum			Urgent pointer
160	Options (optional)			
160/192+	Data			

Tabell 2.2.1.1.: Ett TCP-pakets struktur, huvdet utgörs av bit 0-160/192, sedan följer datan..

Destinationens protokollimplementation ansvarar för att pussla ihop alla paketen till den ursprungliga datan. Varje paket får ett sekvensnummer och TCP ser till att inga paket går förlorade i överföringen genom att hålla reda på vilka sekvensnummer det mottagit och skickar en bekräftelse för de nummer det har fått. Om en bekräftelse för ett paket inte anlänt inom ett rimligt tidsintervall så skickas det än en gång. På så sätt kan vi vara säkra på att vår data når dess destination, och att eventuell oordning som uppstår på vägen inte påverkar överföringen nämnvärt.

2.3. Dataströmmar i relation till routrar

Vi står inför två huvudsakliga problem, först och främst problemet med beräkningstid. Säg att ett nytt paket anländer till en router och alla tidigare paket är sparade i en sorterad lista (t.ex. ett balanserat binärt sökträd). För att se om detta paket är samma som något som redan finns i listan så måste det jämföras med log(n)[4] antal paket från listan, där n är antalet flöden i listan, i genomsnitt. Den logaritmiska funktionen växer så långsamt att den nästan kan bortses ifrån. Men på Internet så är log(n) ofta större än 20 och i vissa fall uppemot 50 [20]. Tjugo operationer kanske inte låter så mycket, men i samband med att mjukvaran på en router redan är pressad till nära brytningspunkten [20] [46] [47], eftersom den sorterar trafik i hastigheter mellan 2 och 40 Gb/s, så kan det innebära väldigt många operationer per sekund. Om vi för ett ögonblick antar att alla paket är 1500 bytes stora (Internet är i stor utsträckning byggt med Ethernet) och att bandbredden (som utnyttjas fullt ut) är 8 Gb/s, så kan vi enkelt räkna ut det antal paket som anländer varje sekund till att vara fler än 600 000. Det faktum att vi har och göra med paket som är mindre än MTU-storleken emellanåt innebär att antalet paket utan svårighet kan stiga till närmare en miljon per sekund. Således krävs det att routern skickar vidare varje paket och utöver det gör vad vi nu tycker den ska göra på totalt endast en enda

4 Med log(n) menas den binära logaritmen, $\log_2(n)$.

mikrosekund. Konsekvensen av detta blir att det vi vill göra utöver routerns vanliga uppgifter måste göras på ett antal nanosekunder [2]. Om vi helt enkelt vill se om paketet finns i vår sorterade lista så innebär det 20 miljoner operationer varje sekund med de siffrorna jag angav ovan, endast för att hitta dem i listan! Det är klart att det helt enkelt inte finns tid till komplicerade beräkningar.

Det andra problemet vi står inför är minnesanvändning. Vid ovan beskrivna hastigheter är den typ av minne som används som primärminne i de flesta hemdatorer, DRAM[5], för långsamt [2] [4]. Åtkomsttiden ligger på 10 nanosekunder eller mer [4]. Vi behöver alltså en typ av minne som kan hålla takten med processorn och strömmen. Ett sådant minne, SRAM[6], används redan i routrar idag eftersom de har stått inför denna utmaning i årtionden vid det här laget. SRAM används också inuti de x86-processorer som driver de flesta av våra PC- och Mac-datorer. Tiden det tar att läsa ett register i CPU:n ligger på någon enstaka nanosekund [4]. Problemet med SRAM är att dess fördelar i hastighet och i vissa fall energianvändning till trots så är det inte lika tätt som DRAM. Medan primärminnet i de flesta moderna PC-datorer ligger mellan 512 MB och 2 GB så når cacheminnet inuti processorerna i de datorerna sällan över 1 MB [53]. När detta skrivs så har en Intel Xeon processor som används i servrar bland annat ”blott” 16 MB cacheminne till förfogande.

Låt oss återigen föreställa oss en 8 Gb/s länk och att vi vill kunna identifiera paket som tillhör flöden vi tidigare sett. Identifierar vi ett paket tillhörandes ett nytt flöde lägger vi till det i en lista eller liknande. Vår router är placerad på änden av en Internetleverantör och dirigerar trafik för dess DSL-kunder. Varje kund har här en individuell bandbredd upp och ner på 1 MB/s och för enkelhetens skull antar vi att alla kunder associeras med 10 flöden vilka inte förändras. Det finns totalt 1000 kunder som vår router är kopplad till, vilket lagom nog blir 1 GB/s nödvändig bandbredd vilket praktiskt nog är kapaciteten för vår router[7]. Återigen så placerar vi naivt vår information i en sorterad lista. Det finns då 10 000 flöden totalt. Flöden är helt enkelt definierade av källport och destinationsport. En IP-adress kräver 4 bytes för att sparas, en port 2 bytes. Totalt 12 bytes per flöde alltså. Att spara alla i en lista skulle kräva 120 000 bytes, eller cirka 118 kB vilket utan tvekan inte är något exceptionellt ens med SRAM. Men i verkligheten klickar folk på länkar, de spelar spel, laddar ner film och lyssnar på webbradio. Det är inte så otroligt att anta att kunderna kan associeras med några nya flöden (och samtidigt kanske dumpa de gamla flödena för alltid) varje minut. Det beror helt på

5 Dynamic Random Access Memory.
6 Static Random Access Memory.
7 Minns att 8 bitar = 1 byte och bandbredden var 8 Gigabit per sekund.

kundens dåvarande beteende. Läser man en intressant artikel i tio minuter får man antagligen inga nya flöden under den perioden.

Vill vi jämföra de flöden som förekommer vid något tillfälle med en annan tidpunkt måste vi spara mer än en 118 kB lista. Är vi intresserade av att kunna dra några slutsatser över en längre tidsperiod så kommer minnet att ta slut ganska snabbt. Desto mer vi vill komma ihåg om flödena eller kunderna och ju längre vi vill minnas det desto snabbare kommer minnet fyllas. Vitter nämner [4] en praktisk gräns på O(log n) eller O(polylog n) för minnesanvänding.

För att relatera till vad jag skrev i kapitel två om hoppande fönster kan vi notera att vi kan spara nio sekunders data per megabyte ungefär. Med denna lösning är det möjligt att spara en eller ett par minuters data men det skulle vara tämligen dyrt och synnerligen opraktiskt att spara över tio minuters data i SRAM.

2.4. Randomisering, när det lönar sig att vara lite glömsk

Om vi skulle glömma information om ett flöde vi tidigare sett och senare ser det igen så kommer vi tro att det är ett nytt flöde. Detta är vad som benämns som en falsk positiv[8]. Falsk negativ är likvärdigt men en exakt motsats. I detta fall skulle en falsk negativ innebära att ett paket som tillhör ett okänt flöde klassificeras att tillhöra ett känt flöde. Sådana felaktigheter kommer att försämra precisionen av mätresultaten. Programmet kommer inte längre leverera ett korrekt resultat (givetvis finns det en viss sannolikhet att den kommer att leverera ett korrekt svar, beroende på mätdatan). Frågan är hur pass fel resultatet kommer att vara för en viss indata och hur detta påverkar vår användning av det. Randomiserade algoritmer används med fördel för att beräkna ett hyfsat svar under en rimlig tid när det exakta svaret inte kan beräknas inom rimlig tid som för NP-kompletta problem.

Frågan om ett flöde har observerats förut kan uppenbarligen inte besvaras exakt om vi inte har tillräckligt minne att spara information om varenda flöde till vårt förfogande. Som jag tidigare nämnde har vi också en gräns på antalet operationer vi kan utföra. Även om vi hade tillräckligt minne för att komma ihåg varje flöde som observerats så skulle det införa åtminstone O(log n) jämförelser för varje nytt flöde vi observerar, om vi använder ett binärt sökträd eller liknande struktur. Vi har ett krav på O(1) operationer för att garantera snabbhet samt för att se till att vi har tid över till andra uppgifter. Ett sätt att uppnå det vore att spara flödena i en tabell och använda IP-adresser och portnummer för källa och destination som

8 Direkt från engelskans ”False Positive”.

index. Dock tvingas vi då allokera minne för *alla* potentiella flöden. Läsaren kanske frestas med att försöka använda en dynamisk struktur, men jag vill påminna om att detta är en sorterad lista och flödena observeras i en slumpmässig följd vilket medför att en dynamisk struktur tvingas sortera om listan vid insättning vilket skulle påverka vårt tidigare krav på O(1) operationer vid insättning. Med IPv4[9] är detta: $(2^{32}\cdot 2^{16})^2 > 7{,}9\cdot 10^{28}$ möjliga flöden.

2.4.1. Sannolikhetslära

Det kan vara lämpligt med en liten kort genomgång av matematiken som ligger bakom analysen av bloomfilter i följande kapitel och randomiserade algoritmer i allmänhet. Den är på inget sätt heltäckande utan är endast tänkt att ge läsaren ett enkelt sätt att följa de beräkningar som utförs senare.

Definition 2.4.1.1.: *Ett sannolikhetsrum består av följande:*

1. *Ett utfallsrum Ω som betecknar mängden av alla möjliga utfall av den slumpmässiga (stokastiska) process som sannolikhetsrummet beskriver.*
2. *En samling av mängder F, där varje enskild mängd i F kallas för en händelse och är en delmängd av utfallsrummet.*
3. *En sannolikhetsfunktion* $Pr : F \to \mathbb{R}$ *som uppfyller definition 2.4.1.2.*

En slumpmässig process beskrivs av ett sannolikhetsrum och alla beräkningar och påståenden refererar till sannolikhetsrummet. En sannolikhetsfunktion definieras enligt följande:

Definition 2.4.1.2.: *En sannolikhetsfunktion* $Pr : F \to \mathbb{R}$ *lever upp till följande villkor:*

1. *för en händelse E,* $0 \leqslant Pr(E) \leqslant 1$
2. *Pr(Ω) = 1*
3. *för en ändlig eller uppräknelig oändlig följd av parvis inbördes disjunkta händelser* $E_1, E_2, E_3, \ldots$, *så gäller*

$$Pr\left(\bigcup_{i\geq 1} E_i\right) = \sum_{i\geq 1}\left(E_i\right)$$

Notera att den tredje delen av definitionen ovan kräver parvis inbördes disjunkta händelser. Om vi är mindre restriktiva och betraktar alla ändliga eller uppräkneliga oändliga följder av händelser följer:

Lemma 2.4.1.1.: *För en ändlig eller uppräknelig oändlig följd av händelser* $E_1, E_2, \ldots$

9 IPv4 innebär 32 bitars IP-adresser och 16 bitars portar. Den nyare Ipv6 introducerar 128 bitars adresser men används egentligen ännu inte utanför försöksnätverk.

$$Pr\left(\bigcup_{i\geq 1} E_i\right) \leq \sum_{i\geq 1} \left(E_i\right)$$

Ett kort exempel för att tydliggöra skillnaden mellan den tredje delen i definitionen och lemma 2.4.1.1 vore att singla två slantar, varje slant för sig. E_1 är händelsen att den första slanten landar med kronan upp och E_2 händelsen att den andra slanten landar med kronan upp. Utfallsrummet består av fyra möjliga utfall av processen, alla med lika stor sannolikhet: $\frac{1}{4}$.

1. Ingen landar med kronan upp.
2. Endast den första slanten landar med kronan upp.
3. Endast den andra slanten landar med kronan upp.
4. Båda två landar med kronan upp.

Här ser vi att E_1 är händelsen att antingen utfall 2 eller 4 inträffar, och E_2 händelsen att nummer 3 eller 4 inträffar. Ett uppenbart faktum uppstår:

Lemma 2.4.1.2.: *För två händelser E_1 och E_2,*

$$Pr\left(E_1 \cup E_2\right) = Pr\left(E_1\right) + Pr\left(E_2\right) - Pr\left(E_1 \cap E_2\right)$$

För exemplet är det tydligt att det måste vara så eftersom både E_1 och E_2 inkluderar utfall nummer 4 och vi kan inte räkna med samma utfall två gånger. Därmed är det klart att

$$Pr\left(E_1\right) = Pr\left(E_2\right) = \frac{1}{2} \qquad (2.4.1.1.)$$

och varför:

$$Pr\left(\bigcup_{i\geq 1} E_i\right) = Pr\left(E_1 \cup E_2\right) = \frac{3}{4} < 1 = Pr\left(E_1\right) + Pr\left(E_2\right) = \sum_{i\geq 1} \left(Pr\left(E_i\right)\right) \qquad (2.4.1.2.)$$

Händelserna E_1 och E_2 är alltså inte disjunkta enligt definitionen men satisfierar lemma 2.4.1.1. Vi kan dock modifiera exemplet enkelt för att visa när definitionen uppfylls.

Låt E_1 och E_2 fortsätta representera att respektive slant landar med kronan upp. Men istället för att singla med varje enskild slant för sig, så klistrar eller tejpar vi fast slantarna tillsammans. De klistras samman så att om den ena slanten ligger med kronan upp så har den andra slanten alltid kronan ner. Singlar vi nu med denna dubbelslant finns det endast två möjliga utfall, båda med sannolikheten $\frac{1}{2}$.

1. Den första slanten landar med kronan upp, och den andra med klave upp.
2. Den första slanten landar med klave upp, och den andra med kronan upp.

$Pr(E_1)$ och $Pr(E_2)$ är fortfarande lika med $\frac{1}{2}$ eftersom då endast varje enskild slant betraktas men:

$$Pr\left(\bigcup_{i\geq 1} E_i\right) = Pr(E_1 \cup E_2) = 1 = Pr(E_1) + Pr(E_2) = \sum_{i\geq 1} Pr(E_i) \qquad (2.4.1.3.)$$

Att tejpa ihop slantarna har med andra ord gjort händelserna E_1 och E_2 disjunkta.

Definition 2.4.1.3.: *Två händelser E och F är oberoende om och endast om*

$$Pr(E \cap F) = Pr(E) \cdot Pr(F)$$

och mer generellt: händelser E_1, E_2,..., E_k är inbördes oberoende om och endast om, för alla mängder $I \subseteq [1,k]$ *:*

$$Pr\left(\bigcap_{i\in I} E_i\right) = \prod_{i\in I} Pr(E_i)$$

Om två händelser E och F är oberoende, så påverkas inte sannolikheten för E beroende på om F inträffar. Betrakta det tidigare slantexemplet. I det första fallet där slantarna singlades var för sig så var sannolikheten att *båda* slantarna skulle landa med krona upp $\frac{1}{4}$. Vilket är precis vad definition 2.4.1.3 kräver:

$$Pr(E_1 \cap E_2) = Pr(E_1) \cdot Pr(E_2) = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4} \qquad (2.4.1.4.)$$

Betrakta nu det andra fallet, där de två slantarna var ihopklistrade. $Pr(E_1)$ och $Pr(E_2)$ är samma som i det första fallet, men sannolikheten att båda slantarna landar med krona upp är noll eftersom tejpningen innebär att någon slant alltid kommer ha kronan upp:

$$Pr(E_1 \cap E_2) = 0 \neq \frac{1}{4} = Pr(E_1) \cdot Pr(E_2) \qquad (2.4.1.5.)$$

Definitionen stämmer väl överens med den intuitiva innebörden av oberoende: Att en oberoende händelse inte påverkas av andra händelser.

Om man har två tärningar kan det ofta vara så att man är intresserad av summan istället för de enskilda värdena av tärningarna. När man slår en tärning så finns det sex möjliga utfall. Detta innebär att om man slår två tärningar så uppstår 36 möjliga utfall, alla med sannolikhet $\frac{1}{36}$. Summan av de två tärningarna har elva olika möjliga värden: 7-12. De elva händelserna har inte lika stor sannolikhet att inträffa. Till exempel är sannolikheten att få 12 (6+6) mindre än sannolikheten att få 10 (6+4, 5+5 eller 4+6). För att beskriva detta definieras en stokastisk variabel:

Definition 2.4.1.4.: *En stokastisk variabel X är en funktion från utfallsrummet Ω till det reella planet;* $X : \Omega \to \mathbb{R}$ *. En diskret stokastisk variabel X är en stokastisk variabel med ett ändligt eller uppräkneligt oändligt antal värden.*

I fallet med summan av de två tärningarna kan vi definiera en stokastisk variabel

$X : \Omega \to \{7, 12\}$ som antar de möjliga värdena på summan. Händelsen ”X = a” representerar mängden av utfall där summan är lika med *a*. Det vill säga mängden $\{s \subseteq \Omega \mid X(s) = a\}$. Sannolikheten för den händelsen är:

$$Pr(X = a) = \sum_{s \subseteq \Omega : X(s) = a} Pr(s) \qquad (2.4.1.6.)$$

Händelsen ”X = 7” är mängden av utfall där summan är lika med sju: $\{(1,6),(2,5),(3,4),(4,3),(5,2),(6,1)\}$. Sex möjliga utfall som ger:

$$Pr(X = 7) = \frac{6}{36} = \frac{1}{6} \qquad (2.4.1.7.)$$

Stokastiska variabler kan också vara oberoende, precis enligt den tidigare definitionen.

Definition 2.4.1.5.: *Två stokastiska variabler X och Y är oberoende om och endast om för alla värden på x och y:*

$$Pr((X = x) \cap (Y = y)) = Pr(X = x) \cdot Pr(Y = y)$$

Poängen med alla dessa definitioner är för att nu kunna ta upp sannolikhetsfördelningar. Anta att vi utför *n* stycken oberoende experiment, som alla lyckas med sannolikhet *p* (och därmed misslyckas med sannolikhet *(1-p)*). Låt den stokastiska variabeln X representera antalet lyckade experiment. X antar då en binomialisk fördelning.

Definition 2.4.1.6.: *En binomialisk stokastisk variabel X med parametrarna n och p, definieras av den följande sannolikhetsfördelningen med j = 0, 1, 2,..., n;*

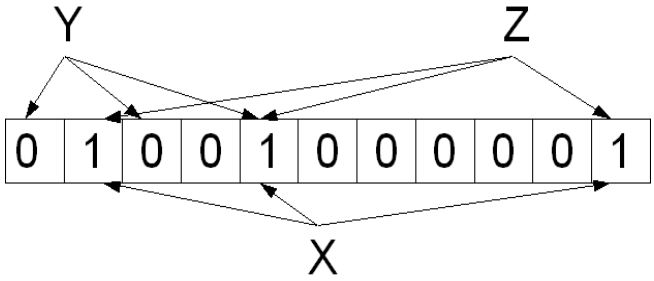
$$Pr(X = j) = \binom{n}{j} \cdot p^j \cdot (1 - p)^{n-j}$$

Med andra ord, X är lika med *j* då det finns exakt *j* lyckade och *(n-j)* misslyckade experiment där varje experiment är oberoende och lyckas med sannolikhet *p*. Det finns $\binom{n}{j}$ sätt att välja vilka experiement som ska lyckas och misslyckas. Fördelningen uppstår senare i analysen av counting bloomfilter där experimenten representeras av ett antal oberoende hashfunktioner.

2.4.2. Bloomfilter

Ett bloomfilter [1], uppkallat efter sin skapare, Burton Bloom, är en ypperligt elegant datastruktur. Sedan dess uppkomst under 70-talet har det använts i allt fler områden [18], även inom datorkommunikation. Det har även vidareutvecklats för att överkomma några av dess fundamentala svagheter och för att klara andra uppgifter. Till exempel så tillåter ett bloomfilter endast insättning och inte borttagning. Dessa vidareutvecklingar har givits

spektakulära namn såsom *Counting Bloom Filters* [11], *Stable Bloom Filters* [13], *Spectral Bloom Filters* [14], *Dynamic Count Filters* [15], *Space Code Bloom Filter* [16] och *Attenuated Bloom Filter* [17].



Figur 2.4.2.1: Ett bloomfilter med 3 hashfunktioner där pilarna indikerar de bitar som bestäms av hashfunktionerna. Först utförs Insert(X). Query(Y) returnerar sedan korrekt 'false' medan Query(Z) felaktigt returnerar 'true', trots att vi inte utfört Insert(Z). Med andra ord är Z en falsk positiv.

Definition 2.4.2.1.: *Ett bloomfilter, BF, är en randomiserad datastruktur som består av följande komponenter:*

En bitvektor av storlek m. Från början är alla bitar 0.

k stycken hashfunktioner: h_1 - h_k, som är definierade: $h_i : X \to \{0, m-1\}$ *, där X är den värdemängd som element är definierade över.*

Ett bloomfilter har två metoder:

Insert(x): De bitar som ges av $h_1(x),..., h_k(x)$ sätts till 1.

Query(x): Bitarna $h_1(x),..., h_k(x)$ kontrolleras. Om någon av bitarna är 0 returneras false. Om alla bitarna är 1 returneras true.

När vi vill veta om *x* finns i filtret och utför *Query(x)*, kontrolleras bitarna $h_1(x),...,h_k(x)$. Om någon av bitarna är 0 är det helt säkert att *x* inte finns i filtret. Skulle alla vara 1 finns det en viss sannolikhet för att det är en falsk positiv eftersom andra element kan ha hashats till samma bitar som *x*. Mitzenmacher har visat [12] att denna sannolikhet minimeras när antalet hashfunktioner $k = \ln(2) \cdot \frac{m}{n}$, till $(\frac{1}{2})^{\ln(2) \cdot \frac{m}{n}}$, där *n* är antalet element hittills insatta i bloomfiltret. Exempelvis kan vi således beräkna att för ett BF på 1000 bitar (*m*), och 160 element (*n*) når vi ett optimum med 4 hashfunktioner av 5 % risk för falska positiva.

Fördelen med bloomfilter är att det endast krävs O(*k*) operationer för insättning och sökning, samt O(*n*) minnesanvändning. Storleken av filtret är inte beroende av storleken av

indataelementen utan endast av deras antal. Med andra ord påverkas inte minnesanvändningen av elementens ordinarie diskstorlek. Nackdelen är givetvis risken för falska positiva. Denna risk kan minimeras till önskad nivå, alternativt kan filtrets storlek modifieras för given felsäkerhet enligt ovan men det beror i slutändan också på de hashfunktioner som används.

2.4.3. Vikten av att välja goda hashfunktioner

En perfekt hashfunktion har samma värdedistribution som en tärning, dvs. helt slumpmässig. På grund av den uppenbara svårigheten med att få något systematiskt och deterministiskt att bete sig på ett sådant sätt är detta sällan möjligt. En god hashfunktion har, något modifierat, därför *nästan* samma värdedistribution som en tärning. Önskade egenskaper är bland annat att en liten förändring av indatan ger upphov till en stor förändring av hashvärdet, och att hashfunktionen är relativt snabb, vilket är speciellt viktigt om ett bloomfilter skall användas för dataströmmar. Att hashfunktionerna är inbördes oberoende är också något önskvärt eftersom det minskar sannolikheten för falska positiva samt för att det förenklar analysen.

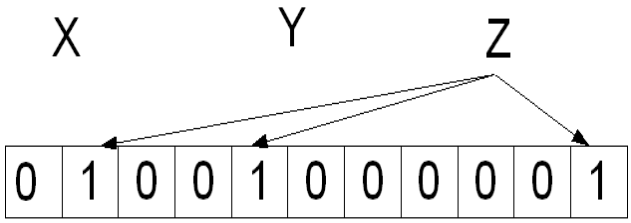
För att ge ett väldigt enkelt exempel på en hashfunktion kan vi ta en funktion baserad på modulus. Vi vill hasha några heltal z till värden från 0 till m. Vi definierar funktionen $h(z) = z \mod m$. Detta är inte en god hashfunktion i de flesta fallen. En liten förändring i indata ger oftast inte upphov till en stor förändring av hashvärdet. Hur snabb den är beror på hur man definierar snabb, men normalt sätt tar modulus ett flertal operationer för en dator att utföra. Det är lätt att tro att vad som endast är en rad kod också betyder en enda operation men så är ofta inte fallet.

Det är inte svårt att föreställa sig indata som ger hashfunktionen en långt ifrån slumpmässig distribution. Välj exempelvis var m:e heltal, så hashas alla element till samma hashvärde med ovan definierade modulusfunktion, de sägs *kollidera*. Detta är ett fundamentalt problem med hashfunktion som inte går att undvika. Även för en god (och med god menar jag de två egenskaper jag nämnde ovan) hashfunktion kan man konstruera indata som får en dålig distribution. Vad som gör en hashfunktion god är att sannolikheten att ett sådant mönster för indatan uppkommer under verkliga omständigheter är låg. Alltså är en hashfunktion väldigt bunden till sin applikation. En hashfunktion som ger en bra värdedistribution för bloomfilter är troligtvis ingen bra hashfunktion för krypterings-

applikationer[10] eller för dataintegritetskontroll[11]. Dessa problem går att undvika så länge som hashfunktionens indata är känd och begränsad. Hashfunktionerna kommer i den här uppsatsen att få IP-adresser och portnummer som indata. Som jag redan nämnt är det totala antalet kombinationer enormt. Men säg för ett ögonblick att vi endast var intresserade av ett specifikt subnät (en Internetleverantör *äger* en mängd IP-adresser som bara de har tillgång till), till exempel vårt lokala hemnätverk. Det finns bara 255 möjliga IP-adresser för datorerna i detta nätverk[12] och att konstruera en bra hashfunktion för den värdedistributionen är trivialt. Vi kan använda sista siffran i IP-adressen direkt som index (för den är unik för varje dator) enklast, och alltså har vi en hashfunktion som är snabb (noll beräkningstid) och som inte ger några kollisioner för indatan. Vill vi sedan titta på ett större nätverk kommer vi dock direkt att stöta på problem eftersom olika IP-adresser kommer att börja kollidera.

2.4.4. Counting bloomfilter

Definition 2.4.4.1: *Ett counting bloomfilter, CBF, består av följande komponenter:*
En vektor av m stycken räknare. Från början är alla räknare 0.
k stycken hashfunktioner: h_1 - h_k, som är definierade: $h_i: X \to \{0, m-1\}$ *, där X är den värdemängd som element är definierade över.*
Ett counting bloomfilter har tre metoder:
Insert(x): De räknare som ges av $h_1(x)$,..., $h_k(x)$ inkrementeras med 1.
Query(x): Bitarna $h_1(x)$,..., $h_k(x)$ kontrolleras. Värdet av den minsta räknaren returneras.
Delete(x): De räknare som ges av $h_1(x)$,..., $h_k(x)$ dekrementeras med 1.

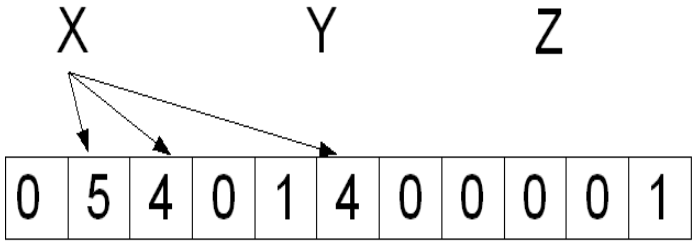


Figur 2.4.4.1.: Ett counting bloomfilter med tre hashfunktioner där Insert(Z) utförs en gång.

Ett CBF [11] skiljer sig från ett vanligt bloomfilter genom att bitvektorn är utbytt mot en

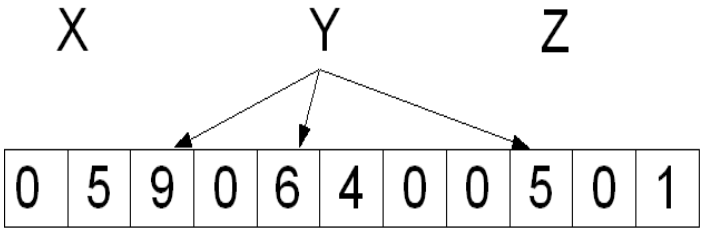
10 Ett viktigt krav i det fallet är också att man inte kan återskapa originalvärdet genom att ta inversen av hashfunktionen eller liknande.
11 Det är vanligt att man kan ladda ner en liten fil med ett MD5-hashvärde i samband med att man laddar ner en större fil som man kan använda för att kontrollera att filen inte korrumperats under överföringen.
12 förutsatt att det är ett C-nät som används, t.ex. 192.168.0.

vektor av räknare. Efter ett antal *Insert*-operationer är man intresserad av hur många gånger *y* har lagts till i filtret. *Query(y)* returnerar då den minsta av räknarna $h_1(y),...,h_k(y)$. Eftersom det är möjligt att något annat värde *z* har kolliderat med någon av *y*:s räknare så kommer räknarna potentiellt ha olika värden. I vilket fall den minsta av räknarna är den som kolliderat minst antal gånger och därför är den *mest* korrekta.



Figur 2.4.4.2.: Insert(X) utförs fyra gånger. Notera att X kolliderar med Z i en räknare, vilken då räknar för högt.

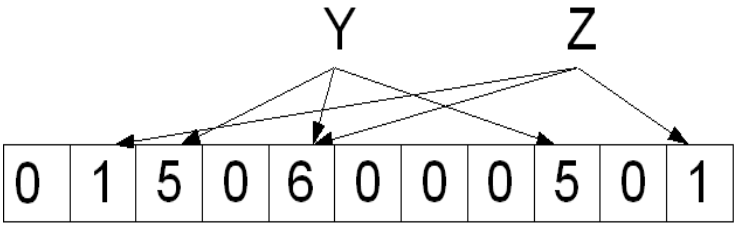
Till skillnad mot ett vanligt bloomfilter så stödjer dock ett CBF även borttagning. Om *Insert(y)* utförs *m* gånger så kommer räknarna $h_1(y),...,h_k(y) >= m$. Vi tolkar detta som att det existerar *m* stycken kopior av *y* i filtret. Vi kan nu ta bort en sådan kopia av *y* genom att i *Delete(y)* minska räknarna $h_1(y),...,h_k(y)$ med 1. Om vi inte utfört *Insert(y)* och det förekommer kollisioner på alla *y*:s räknare så kommer *Delete(y)* innebära att andra elements räknare får för låga värden i filtret. I värsta fall så skapas falska negativa genom att räknare felaktigt nollställs. Eftersom borttagningen i counting bloomfilter inte används i den algoritm som redovisas senare tar jag inte upp sannolikheten för att en räknare har ett för lågt värde efter *Delete()*-operationer.



Figur 2.4.4.3.: Insert(Y) utförs fem gånger. Trots att Y kolliderar med både X och Z så finns det fortfarande räknare som visar rätt.

Om ett element förekommer många gånger, så kommer dess räknare att ha höga värden. Det finns då en chans att räknarna når sitt maxvärde och slår om till noll, vilket gör borttagning omöjligt för de element som kolliderar med de räknarna. Fan et al. menar med sin

analys [1] att 4-bitars räknare (maxvärde 15) skulle vara tillräckligt för de flesta applikationer för att unvika överslag. Det är dock inte tillräckligt för användningen i denna uppsats där värden över 50 inte var något ovanligt. I min implementation använde jag mig av 16-bitars räknare (maxvärde 65 535), men 8 bitar (maxvärde 256) hade varit alldeles lagom.



Figur 2.4.4.4.: Delete(X) har utförts och dess räknare har dekrementerats. Y och Z har korrekt värde i fler räknare än innan.

Precis som för bloomfilter så finns det en risk för falska positiva i ett CBF. *Query(z)* kan returnera ett värde över 0 trots att *Insert(z)* inte har utförts. Sannolikheten för det är dock den samma som för ett vanligt bloomfilter, vilket redovisats tidigare. Ett annat fel som CBF inför är risken att det värde som returneras av *Query(z)* är för högt. Eftersom flera element kan ha kolliderat i sina hashvärden finns det en risk att alla räknare, och därmed även den minsta, har kolliderat. Detta är å andra sidan direkt relaterat till sannolikheten för falska positiva; sannolikheten att andra element hashats till samma räknare som *y*.

Det är således intressant att veta hur stor sannolikheten är att det värde som returneras skiljer sig från det verkliga värdet med mer än ett värde *j*. Detta är mycket beroende av hur många av varje element som kommer att läggas till i filtret. Om vi definierar p_j som sannolikheten att en kollision får räknaren att visa mer än *j* fel, kan vi göra följande observationer. Vi gör även här den ytterligare definitionen att antalet element *n* betecknar antalet *olika* element, inte antalet *förekomster* av element.

Sannolikheten att räknaren *r* inkrementeras har en binomialisk distribution. Dvs.:

$$Pr(r\ inkrementeras\ av\ i\ element) = \binom{n}{i} \cdot p^i \cdot (1-p)^{n-i} \qquad (2.4.4.1.)$$

där p = sannolikheten att någon av de k hashfunktionerna träffar räknaren.

Vilket innebär att sannolikheten att *r* inte inkrementeras av någon av de *n* elementen är:

$$(1-p)^n = (1-\frac{k}{m})^n = p_1 \qquad (2.4.4.2.)$$

Sannolikheten att inkrementeras av exakt *ett* annat element (en enda kollision) är:

$$\binom{n}{1}\cdot p\cdot(1-p)^{n-1}=\binom{n}{1}\cdot\frac{k}{m}\cdot(1-\frac{k}{m})^{n-1}=p_2 \qquad (2.4.4.3.)$$

Således är då sannolikheten att *mer än en* kollision inträffar för räknaren *r* :

$$1-p_1-p_2=p_3 \qquad (2.4.4.4.)$$

Sannolikheten att räknaren *r* då visar *mer än j* fel, p_r, är mindre än:

$$p_j\cdot p_2+p_3>p_r \qquad (2.4.4.5.)$$

Jag gör här förenklingen att mer än en kollision automatiskt gör att räknaren visar mer än *j* fel, vilket inte behöver vara fallet (vilket innebär att den verkliga sannolikheten är lägre). Slutligen då, den totala sannolikheten att ett element *y* rapporteras finnas i filtret mer än *j* för mycket är lika med sannolikheten att *alla* räknarna som bestäms av $h_1(y),...,h_k(y)$ rapporterar mer än *j* för mycket, vilket är lika med

$$(p_j\cdot p_2+p_3)^k=(1-(1-\frac{k}{m})^n-(1-p_j)\cdot\binom{n}{1}\cdot\frac{k}{m}\cdot(1-\frac{k}{m})^{n-1}{}^{k}) \qquad (2.4.4.6.)$$

Som ett specialfall kan vi sätta p_j till 1 (varje felrapportering kommer att överstiga *j*, alternativt $j = 0$, dvs. vad är risken att *Query(y)* returnerar ett felaktigt värde) och ut kommer ekvationen:

$$(1-(1-\frac{k}{m})^{n\ k}) \qquad (2.4.4.7.)$$

Som nästan är lika med $(\frac{1}{2})^{\ln(2)\cdot\frac{m}{n}}$ (2.4.4.8.)

Förutsatt att $k=\ln(2)\cdot\frac{m}{n}$ (2.4.4.9.)

Dvs. chansen för falska positiva i ett bloomfilter. Detta eftersom

$$\lim_{n\to\infty}(1-\frac{\ln(2)}{n})^n=\frac{1}{2} \qquad (2.4.4.10.)$$

enligt exponentialfunktionens definition:

$$e^x=\lim_{n\to\infty}\left(1+\frac{x}{n}\right)^n \qquad (2.4.4.11.)$$

där om vi definierar $x=-\ln(y)$ får:

$$\frac{1}{y}=\lim_{n\to\infty}\left(1-\frac{\ln(y)}{n}\right)^n \qquad (2.4.4.12.)$$

Redan när $n = 10$ så är

$$(1-\frac{\ln(2)}{n})^n>0{,}4875 \qquad (2.4.4.13.)$$

så funktionen konvergerar väldigt snabbt. Den lilla skillnad som uppstår mellan de olika funktionerna kan förklaras av att Mitzenmacher i sin analys uppskattar sannolikheten med en exponentialfunktion, som inte heller är en exakt representation. Vad som är klart är att chansen att ett elements antal felaktigt rapporteras är jämförbar med chansen för falska positiva i ett traditionellt bloomfilter.

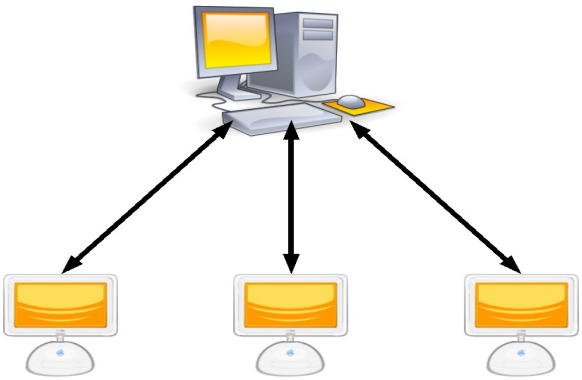
2.5. Peer-to-peer

Innan 1999 låg fildelning på en närmast personlig nivå. Kände man inte någon som kunde bränna det på CDROM, lägga det på en diskett eller mejla det till en så var det både besvärligt och tidsödande att leta reda på det man sökte. I fallet med musik och mp3 så fick man vända sig till webbsidor som enskilt inte hade ett särskilt stort urval. Tillgängligt lagringsutrymme på webbhotell och i allmänhet var tämligen mer begränsat och betydligt dyrare än i dagsläget. Då släppte Shawn Fanning Napster.

Napster var inte ett fullständigt Peer-to-peer-program (P2P) men historiska skäl kräver att det åtminstone nämns. Förutom Bittorrent (som är det mest populära idag) och Napster så har jag valt att även nämna Gnutella och DirectConnect eftersom de båda fyllde en lucka mellan Napsters undergång och Bittorrents popularisering, samt för att illustrera skillnaden mellan Bittorrent och tidigare P2P-protokoll. Dessutom är det värt att ta upp skillnaderna mellan dagens protokoll (Bittorrent) och gårdagens (Napster) eftersom om några år så kommer nya protokoll att ha utvecklats och det är svårt att förutspå hur dessa kommer att bete sig, även om Bittorrent idag verkar vara en indikation. Fler program existerar (och har existerat) som Fasttrack (Kazaa), WinMX, Gnutella2, eDonkey osv. men de kommer inte att tas upp i någon detaljerad form. Det är viktigt att inse att P2P inte bara används för fildelning, även om det är det mest använda. Det används där en decentraliserad struktur är att föredra eller den enda möjligheten.

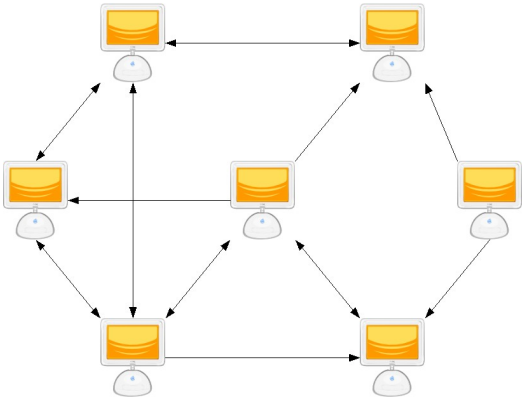
2.5.1. Skillnaden mot Klient-Server

Den vanligaste metoden för att koppla upp datorer mot varandra har traditionellt sätt varit med klient-server-sättet. En dator agerar här server och en eller flera klienter kopplar upp sig mot den. Klienterna har ingen vetskap om varandra och kan inte direkt kommunicera med varandra. All information mellan klienter går först genom servern. Detta är naturligt när klienterna ska ha tillgång till samma information eller när klienterna inte har något behov av att prata med varandra som vid webbsurfning. Framförallt två aspekter är speciellt tydliga.



Figur 2.5.1.1.: Klient-Server.

För det första så måste servern ha väldigt hög bandbredd ner mot klienterna eftersom alla klienter talar med samma server. Blir tjänsterna väldigt populära så som Googles sökmotor tvingas företagen att använda sig av flera maskiner och flera linor ut mot Internet för att kunna erbjuda en snabb tjänst även vid hög belastning. Den andra aspekten är att om servern kraschar eller av någon anledning kopplas ner så försvinner den eller de tjänster som maskinen hanterade för sina klienter. Alltså är servern sårbar för angrepp och fel.



Figur 2.5.1.2.: Peer-to-Peer.

Peer-to-peer å andra sidan förlitar sig inte på en enda server. Här agerar alla klienterna samtidigt server. Klienter kallas istället för *peers*. All information, när klienterna väl har hittat varandra, passerar direkt mellan klienterna utan att först passera genom en tredje part. Detta innebär att i kontrast mot server-klient-metoden så påverkas inte P2P-nätverket av att någon enskild klient kopplas ner. Såvida den klienten inte hade någon unik information som den ännu inte delat med andra klienter. Alltså är det motståndskraftigt mot angrepp och fel. Samtidigt har det även fördelen att ingen klient måste ha en högre bandbredd än andra

klienter för att kommunikationen ska vara snabb. Antingen kan klienten skicka information i sin helhet till andra klienter en och en, eller så kan den skicka till alla samtidigt. Tiden det tar för att alla ska ha all information är den samma, men med den förstnämnda metoden så existerar informationen i fler kopior och är därmed säkrare och snabbare för den enskilda klienten. Varje klient som har en komplett kopia kan i sin tur dela med sig till andra.

2.5.2. Napster

1999 kom så det första riktigt populära fildelningsprogrammet. Det stödde endast överföringar och sökningar av mp3filer. En användare kopplade upp sig mot servern som i sin tur tillhandahöll information om andra uppkopplade användare, samt vilka filer de hade tillgängliga. Om filen man sökte hittades så laddades den ner direkt från en av de användare som delade ut den.

Napsters styrka låg i att alla använde det. Vid tidpunkten fanns inga konkurrerande applikationer och således fanns det ett stort antal användare uppkopplade. Med dagens mått mätt hade det betraktats som ganska långsamt, något som inte märktes då eftersom nästan allas uppkopplingar också var långsammare.

I slutet av samma år stämdes Napster av skivbolagen och 2001 stängdes hela nätverket (kort därefter återuppstod Napster som en betaltjänst). Dess svaghet låg i den centraliserade strukturen (och därmed kvalificerar det inte för verklig P2P i ordets sanna mening) med en server som alla klienter kopplade upp sig mot.

2.5.3. Gnutella

Ursprungligen utvecklat på Nullsoft (men strax övergivet efter att AOL, som köpte upp Nullsoft samma år, satte ner foten [56]) och släppt år 2000. Det är ett decentraliserat system utan en central server och stödjer alla typer av filer. Det är därmed P2P i ordets rätta bemärkelse.

En klient som kopplar upp sig mot Gnutellanätverket måste först hitta en annan klient som är uppkopplad. Det kan göras med en lista över potentiellt fungerande noder från en webbsida eller annan källa (exempelvis har även IRC[13] använts). När den väl är uppkopplad så etablerar den sin egen lista över noder som används nästa gång klienten vill koppla upp sig. Väl uppkopplad till nätverket kan man söka, ladda ner och ladda upp filer. Precis som Napster så beror hastigheten på filöverföringarna helt på bandbredden hos de individuella klienterna

13 Internet Relay Chat.

eftersom man överför en fil endast mellan två klienter åt gången. Men här hanterar de enskilda *peers*:en även sökningarna genom nätverket.

2.5.4. DirectConnect

DC kom ungefär 1999 [55]. Det finns ett flertal tredjeparts klientprogram för protokollet [57]. Precis som när det gäller Napster så finns här en central server, här kallad hub, som klienterna kopplar upp sig mot. Till skillnad mot Napster så finns det dock inte bara en server utan vem som helst kan starta en DC-hub. Hub:en tillhandahåller sökningsmöjligheter över de filer som delas ut av uppkopplade användare samt chatmöjligheter.

En klient kan vara i antingen *active mode* eller *passive mode*. En klient i aktivt läge kan söka och ladda ner från alla andra klienter, medan passiva klienter endast kan ladda ner från aktiva. Aktiva klienter lyssnar på en port och kan direkt få förfrågningar om en viss fil. Passiva klienter måste å andra sidan få en sådan förfrågan från servern. En aktiv användare ber servern säga till den passiva klienten att öppna en förbindelse till den port den lyssnar på, en passiv klient kan inte göra detta eftersom de inte kan lyssna på någon port.

Rent praktiskt är det så att de klienter som sitter bakom brandväggar oftast, om de inte manuellt vidarebefordrar trafik på en viss port, blir passiva. De har därmed inte tillgång till lika omfattande material som aktiva användare har. Själva överföringarna sker, när en förbindelse väl upprättats, med eller utan serverns hjälp, direkt mellan klienterna. Själva överföringen av filer och kontakten med servern sker med TCP och sökningarna genom UDP.

En klient kan specificera exakt hur många samtidiga förbindelser som ska vara tillåtet, så kallade slots (olika för upp- respektive nedladdning). Hub:ar är ofta specialiserade på en viss typ av material, t.ex. just film, anime eller spel. Det är också vanligt att de har krav på hur mycket material som måste delas ut[14], vad materialet ska bestå av, hur många slots som ska vara öppna och dessutom även vilken bandbredd och leverantör användare måste ha för att få koppla upp sig. En hub kan även stödja, och kräva, att användarna registrerar sig. Allt detta är helt upp till administratören bakom servern.

Svagheten är densamma som för Napster, dvs. en central server som alla är beroende utav. Det finns fler servrar visserligen, men detta medför också att allt material sprids ut på flera servrar vilket gör att det kan vara svårt att hitta en speciell fil. Vem som helst kan dessutom inte starta en hub eftersom det behövs en ganska stor bandbredd (speciellt uppåt men även nedåt) då alla sökförfrågningar och passiva nedladdningar går genom servern. Detta

14 En användare bestämmer manuellt vilka filer eller mappar på datorn som ska delas ut till andra användare.

innebär vidare att det finns en övre gräns på antalet användare för varje server, beroende på dess bandbredd och andra resurser så som processor osv.

2.5.5. Bittorrent

Detta är det mest populära P2P-protokollet idag och beräknas ansvara för 35 % [43] av all trafik över Internet i dagsläget. Protokollet har blivit så framgångsrikt att det är det första P2P-protokollet som anammats av kommersiella aktörer för att distribuera filer [26] [50] [51]. Det skapades av Bram Cohen 2001 och utvecklas idag av hans företag Bittorrent Inc [61].

Protokollet fungerar genom att man först laddar ner en så kallad torrent för den fil man är intresserad av. Detta görs vanligen från så kallade torrent-trackers, hemsidor som endast sparar torrent-filer och erbjuder sökningar bland dem. En torrent är en metafil som innehåller information om den eller de filer man egentligen är intresserad utav. Bland annat innehåller den adresser till en eller flera trackers, som inte måste vara samma som den där man laddade ner torrentfilen från första början, information om antal bitar samt hashvärden för bitarna. Därefter öppnar man filen i sin Bittorrent-klient som i sin tur då begär en lista av klienter från någon av de trackers som är specificerade av torrent-filen.

Det som gör Bittorrent unikt är att den eller de filer som delas via torrenten, delas upp i småbitar. När en sådan småbit har laddats ner kan klienten redan då erbjuda den biten till andra klienter. Alltså måste inte klienter vänta på att få ner en komplett fil innan den kan delas vidare till andra klienter. Med hjälp av en algoritm som kallas ”Rare-First” så laddas de sällsyntaste delarna ner först. Det tar därför väldigt kort tid för att en fil ska existera på mer än en plats i nätverket, trots att kanske ingen klient har den kompletta filen. Alla har olika delar.

Eftersom klienter kan dela med sig av enstaka delar av filer så är hastigheten på en Bittorrentöverföring generellt sätt högre än för de andra P2P-protokollen som nämnts. Det medför också att den individuella bandbredden för varje klient spelar betydligt mindre roll eftersom klienter med större bandbredd helt enkelt kan koppla upp sig mot fler peers. De två absolut största fördelarna med protokollet är:

1. Skalningsmöjligheterna. Kapaciteten för nätverket höjs för varje peer som tillkommer, oavsett hur stor bandbredd den har tillgänglig. Det krävs ingen central server med väldigt stora resurser så som för Napster eller DC. Detta är intressant för företag eftersom de därför inte behöver betala någon Internetleverantör för en väldig massa bandbredd och ändå kan de erbjuda snabb överföring av de filer de vill distribuera.

2. Feltoleransen. I fallet för Napster, DC och vanlig klient-server-trafik så är all trafik väldigt beroende på en central server. Skulle den servern av någon anledning försvinna från nätet så kraschar hela P2P-nätet. För Bittorrent spelar det ingen roll om en enskild peer försvinner, så länge den inte hade någon unik bit av filen den inte hunnit dela med sig av ännu. Stabilitet och tillgänglighet är naturligtvis också intressant för företag eftersom de då inte behöver betala någon jourtekniker för att åka in till kontoret och fixa servern.

Den enda tydliga svaghet som Bittorrent har är var man får tag i torrentfilen för att koppla upp sig mot andra peers. Om torrent-trackern försvinner, blir det svårt för nya användare att få tillgång till de peers som finns. Det löses till viss del av att flera torrent-trackers tillhandahåller samma torrentfiler. När man väl fått kontakt med någon annan peer gör det inget om man tappar kontakten med trackern. Detta eftersom en peer kan få kännedom om nya peers genom de peers den redan har kontakt med.

Efter att vissa Internetleverantörer har haft en negativ policy mot just Bittorrent [28] [29] [31] så har de flesta Bittorrentklienter nu implementerat RC4-kryptering [37] [38] [39] [40] av trafiken, samt börjat använda portar som skiljer sig från Bittorrents standard (6881-6889). Ingen av dessa metoder ger anonymitet för användarna, och är inte heller tänkta för det. Målet är att kringgå den begränsning som vissa leverantörer infört i sina nätverk (se kap 7).

2.5.6. Botnät

Det finns andra applikationer för P2P än just fildelning. Instant Messaging är ett enkelt exempel. Ett annat betydligt mer intressant (i spårningssyfte) exempel är botnät. Bot står för robot och det syftar här på ett program som körs på en så kallad zombiedator; en dator som oftast utan ägarens kännedom kör en form av fjärrstyrningsmjukvara. Zombien kan vara vilken dator som helst och botprogrammet används av den som har kontroll över botnätet för diverse syften. Vanliga uppgifter som botnäten har är att leverera spam eller ta del i en DDOS-attack[15], en attack där tusentals datorer samtidigt kopplar upp sig mot t.ex. en hemsida precis som vanliga webbsurfare men mycket mer frekvent i syfte att överbelasta servern. De olika användningsområdena för botnät är bland annat som utpressning av företag genom att hota att initiera en DDOS-attack mot deras nätverk, eller att sälja spammöjligheter till företag i reklamsyfte exempelvis är ett sätt för ”ägaren” av botnätet att tjäna pengar.

Botnät är ett stort problem på Internet och det uppskattas att upp till 150 miljoner datorer

15 Distributed Denial of Service.

[48] kan vara infekterade av botprogram. De sprids vanligtvis genom virus, maskar eller trojaner. Enskilda botnät kan bestå av över en miljon zombies [49]. Vanligtvis har nätverket styrts över IRC. Nyare versioner använder sig dock av ett P2P-protokoll [10] [58], vilket av säkerhetsskäl motiverar identifieringen av sådan P2P-trafik.

2.5.7. Fildelning är olagligt, eller?

Elektronisk distribution av information kan aldrig bli olagligt. Det skulle på ett allvarligt sätt påverka den personliga integriteten. Det är enligt svensk lag inte tillåtet att sprida upphovsskyddat material utan upphovsmannens tillstånd. Upphovsskyddat material står för en stor del av P2P-trafiken över Internet, så mycket är uppenbart. Internetleverantörerna har dock lika mycket och göra med vad deras kunder skickar över Internet som Posten har med vad människor skriver i sina brev. All fildelning genererar väldigt mycket trafik, vilket påverkar leverantörernas förmåga att erbjuda kvalité för sina tjänster genom låg fördröjning och hög bandbredd.

En del Internetleverantörer har valt att helt sonika slå ner på sina kunder genom att antingen begränsa eller sabotera P2P. Amerikanska Comcast upptäcktes under hösten 2007 med att sabotera P2P-trafik för sina kunder [28] [29] [31]. Genom att göra detta kan leverantörerna undvika kostsamma uppgraderingar [46] [47]. Men som jag nämnde ovan så har flera företag tagit till sig av Bittorrent-tekniken för att distribuera stora filer. Blizzard Entertainment använder det för att distribuera uppdateringar till sitt spel World of Warcraft med över nio miljoner spelare världen över. I många fall är dock Comcast de enda tillgängliga leverantörer som kunderna har möjlighet att anlita för att kunna få en någorlunda snabb uppkoppling. Antingen för att de bor utanför storstadsområdena eller för att konkurrensen är låg i just deras område. En situation som är värd att jämföras med den ställning svenska Telia hade för några år sedan. Även inom musikbranschen där motståndet mot fildelning traditionellt sätt varit som starkast har man börjat inse några unika fördelar med att erbjuda materialet fritt utan begränsningar. Bandet Radiohead släppte 2007 skivan *In Rainbows* gratis på sin hemsida och erbjöd sina fans att betala vad de tyckte skivan var värd genom Internet. Hittills har den tjänat in runt 62 miljoner kronor, som utan mellanhänder gick direkt till bandet.

Tiden kommer att utvisa hur frågan med upphovsrättsskyddat material kommer att lösas. Fildelning och P2P är här för att stanna. Inte minst på grund av den juridiska svårigheten att angripa torrent-trackers så som The Pirate Bay. Eftersom inga filmer, ingen musik osv lagrats

på The Pirate Bays servrar så kan de ansvariga inte anklagas för brott mot upphovsrätten. Kryptering och andra metoder kommer i framtiden göra det svårt för myndigheter och branschorganisationer att identifiera fildelare.

Jag vill göra klart att jag inte skrivit denna uppsats i syfte att spåra personer som bryter mot svenska lagar. Den algoritm jag föreslår för att identifiera P2P-trafik kan inte utnyttjas för att skilja laglig fildelning mot olaglig eftersom jag inte alls tittar på den data som skickas och faktiskt förutsätter att den är oläslig i och med någon form av kryptering. Det finns andra anledningar till att man vill identifiera P2P-trafik som jag kort nämnde innan och det är på de anledningarna jag har baserat mina avsikter.

2.6. Trafikprioritering

Information om vilken typ av trafik bredbandsanvändare genererar är av intresse av flera anledningar. Inte minst därför att bredbandsleverantörerna försöker minimera kostnader och maximera sin vinst. Den bästa kunden en leverantör kan ha är någon som betalar för mycket bandbredd men som inte utnyttjar den. Den utan tvekan mest populära betalningsformen idag är en fast månadskostnad. Man betalar för att ha tillgång till en specifik bandbredd oberoende av hur mycket eller lite man använder den. Leverantörerna å andra sidan betalar sina egna leverantörer för den trafik som passerar genom deras nät, inte för vilken bandbredd de utnyttjar [19]. Att skicka data till ett nät på andra sidan världen kostar mer för leverantören än att skicka inom det egna nätet. Var datan färdas är intressant för att kunna minimera konstnader.

Detta motiverar varför majoriteten av leverantörerna i sina kundavtal inkluderar en klausul om att de har rätt att stänga tjänsten för de kunder som utnyttjar sin uppkoppling på ett ”onormalt” eller ”oskäligt” sätt. Vilken mängd data som är ”normalt” eller ”onormalt” förbehåller sig leverantörerna rätten att definiera [36]. Detta finner kunderna ofta förbryllande, eftersom de betalar för en bandbredd och kan tekniskt sätt normalt inte överträffa den.

En mer kundinriktad motivering till att identifiera och påverka trafiken är QoS[16]. En bredbandsleverantör har en begränsad bandbredd till sitt förfogande. Skulle alla kunder utnyttja sin uppkoppling maximalt på en och samma gång skulle leverantörens utrustning sannolikt överbelastas och kunderna skulle lida av fördröjningar i överföringarna, vad som kallas för ”latency”.

16 Quality of Service.

Men det finns vissa tjänster som kunder ändå förväntar sig ska vara funktionella. Om leverantören även utöver en Internetuppkoppling erbjuder IP-telefoni är det rimligt att kräva att telefonerna alltid ska fungera, utom vid strömavbrott. Det vore orimligt om det blev en halv sekunds fördröjning i telefonerna bara för att en miljon svenskar loggar in på Facebook [59]. Detsamma gäller även IPTV, onlinespel, strömmande video och musik och även till viss del vanlig webbtrafik. Många viktiga tjänster tillhandahållna av myndigheter, banker och liknande är numera ofta baserade över Internet och förstärker bara vikten av att vissa tjänster ska fungera i någon mån oavsett belastning.

Peer-to-peer-trafik använder (åtminstone när det används för fildelning) per definition mycket bandbredd. Applikationerna är utvecklade för att utnyttja all oanvänd kapacitet. Dessutom kan *peers*:en vara lokaliserade var som helst i världen. För kunden är varifrån han laddar ner eller laddar upp inte av intresse, bara så länge det går snabbt. För leverantören är det dock, som sagt, väldigt intressant.

Ett sätt att minimera kostnader och möjligtvis öka hastigheten samt minimera fördröjningar för kunden skulle vara att implementera en slags P2P-Proxy hos leverantören med ”Cache Discovery Protocol” [41]. Proxyservern skulle tillhandahålla populära filer och kunde därmed minska antalet överföringar till andra leverantörers nät samtidigt som den kunde erbjuda en antagligen snabbare överföring eftersom proxyservern troligtvis har större bandbredd mot kunden än vad en dator på andra sidan Jorden skulle ha. En sådan implementation på alla typer av filer är dock väldigt osannolik så länge som häxjakten [52] på fildelare fortsätter.

2.6.1. Quality of Service

Vad QoS innebär är ständigt föränderligt. Jag väljer att nämna fyra ganska konstanta faktorer och hur de påverkar olika tjänster.

- Felsäkerhet

  Om vi tar en filöverföring som exempel så måste varenda byte levereras korrekt om inte filen ska bli korrupt under överföringen, vilket troligen skulle göra filen oanvändbar. Detta motiverar varför tjänster som beror mycket på korrekt data primärt går över TCP där information som korrumperas eller försvinner på vägen skickas tillräckligt många gånger tills den anlänt till sin destination. Strömmande video till exempel är inte alls lika beroende av felsäkerhet. Att någon bildruta försvinner på vägen eller korrumperas påverkar inte filmen generellt förrän väldigt många bildrutor

börjar försvinna.

- Bandbredd

Filöverföringar är också ganska beroende av bandbredd. Användare vill att det ska gå snabbt men det går bra även om det går lite långsamt. Strömmande video å andra sidan har strikta krav på bandbredden beroende på kvalitén. Om vi har HD-kvalité så behöver vi väldigt stor bandbredd för att filmen ska kunna spelas upp i normal hastighet. Om hastigheten sjunker under gränsen tvingas vi använda buffring, vilket skapar irriterande fördröjningar under filmen.

- Fördröjning (latency)

Onlinespel är väldigt beroende av en låg latency. Om man skjuter iväg en raket mot en motståndare så måste motståndaren också få veta det inom några hundradels sekunder. Redan vid några hundra millisekunders fördröjning så börjar många spel bli ospelbara eftersom verkan och effekt sätts ur spel. En spelare kan skjuta motståndaren först, på sin egen skärm, men servern registrerade motståndarens skott först och således dör den första spelaren, trots att han aldrig såg sin motståndare avfyra sitt vapen. Filöverföringar å andra sidan har låga krav på fördröjningar. Det spelar ingen roll om det blir några sekunders fördröjning så länge som bandbredden kan hållas hög och stabil. Strömmande video och ljud påverkas inte heller sålänge som fördröjningen är konstant.

- Jitter

Om fördröjningen inte är konstant uppstår vad som kallas för jitter. Om en förbindelse har högt jitter så menar man att variansen i fördröjning är hög. Eftersom olika paket kan skickas över olika vägar så kommer de att utsättas för olika fördröjningar. Telefontjänster är väldigt beroende av ett lågt jitter. Om fördröjningen varierar för mycket blir det svårt att uppfatta tal. För strömmande video och ljud kan man kompensera genom att använda en buffer. Men desto högre jitter desto större buffer tvingas man använda.

Som jag nämnde kan man använda en buffer för att kompensera för några av problemen. En annan enkel metod är att helt enkelt överdimensionera sin utrustning. En leverantör skulle då till exempel placera fem hundra kunder bakom en router som skulle klara av att hantera tusen stycken kunder. Slutligen återstår då ”traffic shaping”, trafikprioritering, som traditionellt använt sig av algoritmer som ”Leaky Bucket”.

”Leaky Bucket” är essentiellt sätt är en stor buffer. När trafik kommer in till routern så

fylls en buffer. Trafik skickas från buffern i en jämn och bestämd takt. Skulle buffern bli full så förkastas all inkommande trafik, alternativt tillåts passera okontrollerat. Detta stabiliserar jitter och bandbreddsanvändning. Dock introducerar man potentiellt sätt större fördröjningar och även fel i trafiken om skulle buffern fyllas och trafik förkastas. Det bästa vore att prioritera olika typer av trafik olika, vilket kräver att man kan identifiera trafiktypen.

2.6.2. Några identifikationsmetoder

Det finns många sätt att identifiera nätverkstrafik. Jag kommer endast att nämna ett par av de mer övergripande metoderna.

2.6.2.1. Portklassificering

De flesta typer av trafik, till exempel webb, FTP, IRC, eller e-post, skickas nästan uteslutande över de välkända portarna[17] [23]. Det samma är sant för de flesta applikationer. Detta gör att man kan enkelt och utan någon egentlig beräkning direkt kan dela upp trafiken efter typ. Den främsta svagheten är att det inte finns några tekniska hinder för att sända till exempel FTP-trafik över port 80, vilket leder till att den trafik som går över icke-standard portar felklassificeras. Fildelningsprotokoll så som Bittorrent och Gnutella har standardportar[18] men den senaste tiden har det blivit mer och mer vanligt att använda icke-standard och även helt slumpmässiga portar [6] [60].

Undersökningar som gjorts har också visat att även om portklassificering identifierar mycket P2P-trafik i dagsläget så återstår det en stor mängd trafik av okänd typ [5]. I takt med att fler och fler P2P-klienter använder icke-standard portar så kommer den okända trafiken att öka.

2.6.2.2. Deep Packet Inspection

Med DPI tittar man på pakethuvudet och paketdatan. Från huvudet kan man fastställa bland annat källa och destination. Från datan kan potentiellt allt annat läsas. Det kan till exempel nämnas att i en Bittorrentöverföring så är det första som skickas ordet ” Bittorrent”. Detta har traditionellt varit en effektiv metod och är vida använd av företag som Cisco, IBM och andra storföretag [54]. Nyare versioner av de mer populära Bittorrent-klienterna implementerar dock RC4-kryptering av datan [37] [38] [39] [40] vilket gör inspektion av paketdatan meningslös. Vidare är etiken av och i vissa fall även lagligheten av inspektion av

17 80, 21, 194 och 110 respektive.
18 6881-6889 och 6347 respektive.

paketdatan också ifrågasättbar [30] [32] [33] [34] [42]. Dessutom tvingas man analysera mer information när man utöver huvudet också tittar på paketets data vilket gör DPI mer resurskrävande än andra metoder.

2.6.2.3. (Shallow) Packet Inspection

Med vanlig SPI tittar man bara på pakethuvudet. Informationen man har tillgång till är inte mycket mer än källa och adress. Det går alltså knappast att dra några slutsatser utav endast detta. Jag vill trots det med denna uppsats visa att man kan med endast SPI med god sannolikhet ändå identifiera P2P-liknande trafik.

SPI introducerar inte heller de etiska dilemman som DPI dras med. Man kan jämföra med att din teleoperatör naturligtvis vet vem du ringde och när, annars skulle de inte kunna koppla ditt samtal rätt. De vet dock ingenting om vad som sades i samtalet. Polisen har intresse av att avlyssna brottslingar, men man har traditionellt sätt endast kunnat göra detta i samband med väldigt grova brottsmisstankar. Detsamma borde rimligtvis även gälla för datatrafik så som e-post och annan kommunikation över Internet.

2.6.2.4. TCP-UDP:par identifiering

Flera P2P-protokoll använder sig av dels TCP för att överföra filer, men också UDP som en kontrollström, och i Bittorrents fall för att ”upptäcka” nya *peers*. I Karagiannis et al.s studie [19] användes denna teknik i kombination med portklassificering för att identifiera P2P. Det finns många program som använder sig av TCP och UDP samtidigt, t.ex. onlinespel. I den nämnda artikeln så användes portklassificering för att sortera bort icke-P2P-trafik. Svagheten är därmed densamma som för den metoden. Det är även inte helt klart huruvida framtidens protokoll kommer att använda sig av både TCP och UDP [45].

2.6.3. Hantering av stora trafikmängder

En studie gjord av brittiska CacheLogic har visat att upp till 35 % av all trafik utgörs av Bittorrent [43]. Det visar sig också att 20 % av användarna står för 80 % av trafiken [5]. Den naturliga slutledningen tordes därför vara att begränsa hastigheten för Bittorrent-trafik och/eller de användare som utnyttjar mycket bandbredd. Något som på engelska benämns ”bandwidth throttling”, bandbreddsstrypning på svenska.

Ett väldigt uppmärksammat fall var, som nämnts tidigare, när den amerikanska bredbandslevantören Comcast under 2007 visade sig begränsa bandbredden för Bittorrent och

Gnutella genom att sabotera sina kunders uppladdningar [28] [31]. Comcast injicerar (vid tillfället då detta skrivs saboterar de fortfarande överföringarna) paket som tillsynes kommer ifrån den andra parten i överföringen och som ber måldatorn att avsluta överföringen. Just i detta fall används ingen regelrätt bandbreddsstrypning, men effekten är den samma eftersom överföringarna gång på gång avbryts och sänker den genomsnittliga hastigheten.

Blockering av icke-önskvärd trafik är något som generellt används på reglerade nätverk, universitetsnätverk till exempel. LDC som är Lunds universitets egen leverantör blockerar en lång rad portar, däribland kända P2P-portar [44]. All trafik som går in och ut ur nätet kontrolleras, och bestäms den vara P2P så blockeras IP-numret. Komplett blockering av viss typ av trafik är knappast något som är acceptabelt för en vanlig leverantör, men inte ovanligt [35].

Ett betydligt mindre saboterande förslag är att återgå till att betala för mängden datatrafik istället för bandbredd. Det är då kunderna som får stå för de kostnader som stor användning av P2P innebär. Ett förslag som ges i Altmanns och Chus rapport [7] baseras på en dynamisk hastighetsbegränsning. Man skulle även kunna tänka sig att icke-P2P-trafik gavs obegränsad hastighet (till den grad tekniken tillåter) medan P2P och liknande ”tung” trafik begränsades till en hastighet som man betalat för. Det finns många möjligheter, om man kan identifiera vad som är och inte är P2P.

3. Del 2: Om att skilja elefanter från möss

Del 2 går igenom hur man kan använda flöden för att identifiera Peer-to-Peer. Det följer jämförelser med andra arbeten och metodiken av undersökningarna förklaras. Slutligen följer resultaten av de undersökningarna och den slutsats jag drar.

3.1. Identifiera Peer-to-peer med hjälp av flöden

När jag under uppsatsens förarbete läste uppsatser och texter om dels dataströmmar och dels P2P-identifiering så undrade jag om man genom att bara titta på mängden flöden som kunde kopplas till en IP-adress (där ett flöde är ett IP:port par), kunde dra någon slutsats huruvida datorn (alternativt datorerna) bakom adressen använde sig av P2P. Viktigare, undrade jag om det kunde göras i realtid i sådana höga hastigheter som förekommer i utkanten av en leverantörs nät (upp till 40 Gbit/s).

Vad jag vet så finns det ingen svensk eller engelskspråkig undersökning som försöker sig på att göra detta i realtid, med en implementationsmöjlighet i SRAM. En liknande offline-undersökning har gjorts av Karagiannis et al. [19].

Eftersom mycket av P2P-trafiken idag är krypterad, och på grund av de etiska och juridiska problem man får när man tittar på paketdatan (man avlyssnar ju bokstavligt talat trafiken) var det givet att endast shallow packet inspection var möjligt att använda. Jag har valt att endast koncentrera mig på TCP-trafik eftersom fildelning sker nästan uteslutande över det. Det finns inga svårigheter med att även inkludera andra protokoll. Det är bara en implementationsfråga eftersom olika protokolls huvud ser olika ut. Principen är dock den samma. Alla paket har en källa och en destination, oavsett dess protokoll.

3.1.1. Utmaningarna

Varje paket måste klassificeras som det tillhör ett flöde som redan observerats eller som det är det första i ett helt nytt flöde. En gigabitlänk kommer potentiellt att hantera över en miljon TCP-paket varje sekund. Vi måste därmed på ett snabbt och effektivt sätt kunna klassificera paketen. Eftersom detta är en fråga om att bestämma om paketet är medlem i mängden ”sedda paket” så är ett bloomfilter väl lämpat för denna uppgift. Det är snabbt, använder lite minne och ger endast en liten del fel sin randomiserade natur till trots. Tiden som används är O(k), där k är antalet hashfunktioner som används. Minnet är m = O(n) bitar, där n är de antal flöden som bloomfiltret förväntas kunna hantera med god sannolikhet.

Med formeln som presenterades i kapitel fem så kan vi beräkna att för ett *n* på 100 000 (antal flöden) och ett *m* på 1 miljon bitar (mindre än 128 kB) så är sannolikheten för falska positiva mindre än 1 % för fem hashfunktioner, och optimal med sju hashfunktioner. Eftersom TCP främst överför data ter det sig rimligt att anta att ett flöde i genomsnitt kommer att använda sig av 10 paket eller mer. Mängden överförd data i 10 paket är nämligen under 15 kB.

När då ett nytt flöde identifierats vill vi använda det för att räkna de antal flöden relaterade till IP-adresserna. Antalet IP-adresser kan vara stort (i värsta fall tillhör varje flöde två stycken nya unika IP-adresser). Därmed kan det fortfarande vara många paket som invokerar den här processen och den måste vara i stort sätt lika effektiv som bloomfiltret innan. Jag har här valt att använda ett counting bloomfilter. Det är snabbt, använder begränsat med minne och har låg sannolikhet för falska positiva.

Ett CBF använder mer minne än ett vanligt bloomfilter eftersom det använder räknare istället för enskilda bitar. Vi kan dock hålla nere storleken genom att observera att långt ifrån alla flöden kommer att tillhöra unika adresser. Enskilda adresser kommer att förekomma i flera flöden. Det hela är beroende av hur många flöden som enskilda IP-adresser genererar. Det beror också på vart flödena är riktade. Om maskinerna inom nätverket endast kommunicerar med varandra så behöver filtret inte vara stort. Men om varje maskin istället har kontakt med främmande datorer i andra nätverk så ökar kravet på filtrets storlek. Vidare undersökningar hade kunnat visa hur många flöden som datorer genererar i genomsnitt. När jag implementerade mitt program överdimensionerade jag filtren kraftigt. Antalet flöden per maskin pendlade mellan 1 och över 50, beroende på vilken typ av aktivitet som rådde.

En länk med kapacitet av 10 Gb/s kan hantera 1000st 10 Mb/s-uppkopplingar (en ganska vanlig hastighet av både ADSL och stadsnät) samtidigt utan problem. Om vi antar att vi använder 16-bitars räknare i vårt CBF, vilket jag gjorde i min implementation, så kan vi med den sedvanliga formeln för bloomfiltret beräkna att för ett filter på 256 kB och 131072 st räknare har en felsäkerhet på mindre än 1 % för 13663 IP-adresser, vilket innebär att medelantalet sedda IP-adresser inte ska överstiga 13663 om vi ska bibehålla 1 % felsäkerhet vid hög belastning. En grundlig undersökning av hur många flöden som observeras kan ge svar på vad den optimala storleken av CBF är. Räknare på 16-bitar innebär ett max antal om 65 535 st flöden, i själva verket skulle antagligen 1-byte-räknare duga (max 255).

När vi således har identifierat en IP-adress som har många flöden, lägger vi till den i den slutgiltiga listan tillsammans med antalet flöden. Denna list har som uppgift att hålla reda på

medelantalet flöden. För att hålla detta värde någorlunda dynamiskt valde jag att reducera medelvärdet till ett enda mätvärde efter ett antal mätintervall. Detta för att motverka situationen att en IP-adress som konstant kan kopplas till många flöden under en lång tid men som plötsligt kopplas ner fortfarande finns kvar i listan och sakta faller mot noll flöden. Så listan håller alltså reda på medelantal flöden för max de senaste *y* sekunderna. Efter *y* sekunder reduceras medelvärdet till att vara jämställt med ett enda mätvärde, och kan därmed fluktuera snabbare igen. Anledningen till att denna lista finns i algoritmen och att CBF:ens värde inte används direkt är eftersom jag ville minska möjligheten för gränsfall där en IP-adress pendlar mellan P2P och icke-P2P.

Vanlig webbtrafik ger upphov till ett litet antal flöden enligt testerna som gjordes, men den kan kopplas till flöden i små toppar med långa bottennoteringar emellan. Om ett cybercafé skulle ligga bakom en NAT-router[1], och alltså ha möjligtvis hundra människor som surfar bakom en och samma IP-adress, så skulle det kunna innebära att dessa toppar blir betydligt högre och kanske passerar gränsen för vad algoritmen skulle klassificera som P2P. Men det skulle snabbt falla under gränsen igen för att stiga snabbt igen. Om routern dirigerar trafiken annorlunda beroende på antal flöden så skulle cybercaféets trafik ständigt skickas annorlunda (detta skulle vara en falsk positiv).

Ständiga uppdateringar i routerns interna routinglista skulle också vara en belastning. Alltså valde jag att bedöma det genomsnittliga antalet flöden per sekund, eller annat mindre intervall, under ett längre fönster. Detta för att bottennoteringar under en längre period skulle balansera ut höga men korta toppar. Samma fel kan givetvis inträffa även här, att en IP-adress ligger på gränsen, men det innebär att eventuella åtgärder endast utförs någon gång per minut eller mindre istället för var eller varannan sekund. Detta sista steg har inte lika höga prestationskrav på sig eftersom P2P-identifierade IP-adresser är begränsade. Men vi har lagt en del tid på bloomfilter och CBF innan detta steg, så det måste ändå vara något effektivt.

Jag valde att representera denna lista över potentiella P2P-adresser med ett balanserat binärt sökträd, specifikt ett rött-svart-träd. Detta trots att minnesåtkomster tar längre tid än att beräkna ett värde i processorn som en hashtabell hade inneburit. Det blir dock aldrig många pekare som måste följas; under tio pekare som är fallet om trädet innehåller mindre än 1024 P2P-identifierade adresser (och i mitt fall skulle detta aldrig överstiga en handfull som mest). Främst valde jag att använda ett träd för att mina undersökningar skulle ske på relativt låga hastigheter samt att det eliminerade alla typer av mätfel som möjligt kan uppkomma av

1 Native adress translation.

bloomfilter. Om algoritmen skulle implementeras i en router så skulle det nästan garanterat krävas något bättre än en trädstruktur. Ett förslag på en bättre lösning ges i avsnitt 3.1.4.

3.1.2. Medelvärdeslistan

Listan har de sedvanliga operationerna *Insert, Delete, Search, Successor, Predecessor*, och så vidare som alla listor har. Utöver det har den även metoderna *Average* och *Reset*.

```
Insert(nyckel k, mätvärde zk):
 Associerar nyckeln k med mätvärdet zk,
 Om k är ett nytt värde så är
     medelvärdet ak = 0 och
     räknaren ck = 0

Average://Beräknar medelvärdet av de mätvärden som kommit in hittills.
 För alla nycklar k i trädet,
     ak =  (ak·ck)+zk / ck+1 ,
     ck = ck + 1,
     zk = 0

Reset:  // Beräknar det totala medelvärdet under den senaste perioden.
     // Reducerar medelvärdet till ett mätvärde
 För alla nycklar k i trädet,
     zk = 0,
     ak =  (ak·ck) / y+1 ,
     ck = 1
```

Insert kallas en eller flera gånger för att lägga in det senaste mätvärdet och *Average* beräknar ett aktuellt medelvärde. I *Reset* så sätts c_k till 1 och ett medelvärde över hela fönstret y beräknas. Om en källa med många flöden tillkommer sent i ett fönster så kommer det rapporteras ha lågt medelantal flöden per sekund över den perioden, men kommer att få ett närmare korrekt värde under nästa fönster. Det är som sagt en kompromiss mellan precision och önskan att tidigare värden alltid ska påverka de senare för att balansera ut kraftiga förändringar. Om en nyckel *k* läggs till i slutet av perioden så kommer dess medelvärde baseras på mindre antal mätvärden än nycklar som tidigare lagts till i listan. Men när en nyckel väl hamnat i listan så kommer *Reset* att beräkna medelvärdet över hela fönstret. Skulle en nyckel därmed helt plötsligt inte få fler mätvärden så beräknas medelvärdet med $z_k = 0$ tills

medelvärdet sjunker under *T* och nyckeln *k* tas bort från listan. Det är därmed enkelt att hamna i listan (genom att i något intervall *x* överstiga *T* flöden), men sådana nycklar som inte lyckas hålla ett medelvärde över *T* (och jag menar här det verkliga medelvärdet) kommer att försvinna ur listan snabbt.

3.1.3. Algoritmen

Här följer pseudokod för algoritmen, den kan lättast tolkas som två trådar. Inom parentes står den datastruktur som är ansvarig för uppgiften i de fall då det kan vara oklart.

```
[TRÅD 1]
För varje paket p
    Om p tillhör ett tidigare ej sett flöde (BF)
        Öka flödesräknarna för de två IP-adresserna som p färdas mellan (CBF)
        Om någon av räknarna överstiger T
         Lägg till IP-adressen i medelvärdeslistan över P2P-adresser
         tillsammans med antalet flöden zk.

[TRÅD 2]
För varje mätintervall x (någon sekund eller kort period)
    Nollställ flödeslistan och flödesräknaren (BF och CBF).
    Beräkna periodens medelvärde för varje IP-adress i listan över
    P2P-adresser.

För varje mätintervall y (större än x)
    För varje adress i P2P-listan,
        Beräkna och skriv ut totalt medelvärde under senaste perioden y.
        Om medel är under T, radera.
        Annars, reducera medelvärdet till ett mätvärde.
```

T = Det minsta antal flöden över vilken en IP-adress klassificeras som möjligt P2P.

x = Ett kort tidsintervall, max ett par sekunder.

y = Ett längre tidsintervall, lämpligen mer än trettio sekunder.

3.1.4. Möjliga förbättringar

De operationer som dominerar arbetet i algoritmen är hashfunktionerna i dess bloomfilter och counting bloomfilter. Om en leverantör endast är intresserad av sina egna kunder och därmed bara är intresserad att räkna antal flöden för de IP-adresser den själv äger, så skulle CBF:et med fördel kunna ersättas av en vanlig hashtabell av räknare om antalet IP-adresser

som är av intresse inte är för högt. En bra hashfunktion som inte ger kollisioner för dessa adresser skulle ganska enkelt kunna konstrueras, som jag nämnde i kapitel 5. Utrymmet skulle därefter vara (om 16-bitars räknare används) $2 \cdot n$ byte, där n är antalet adresser som kommer observeras. Om n då fortfarande är 13663 (som i mitt tidigare exempel för CBF) så kommer denna enkla hashtabell endast att utnyttja 27 kB minne jämfört med 256 kB som motsvarande dimensionerade CBF skulle. Operationer tar O(1) tid istället för O(k). Räknare av storlek 1 byte skulle sannolikt vara alldeles tillräckligt, vilket då skulle innebära en storlek av precis n byte.

Med samma motivering skulle den sista listan med medelvärdena också kunna ersättas av en enkel hashtabell. Det skulle inte innebära samma drastiska förbättringar av minnesanvändingen som för CBF, eftersom vi här endast sparar information om de adresser som vi tror använder P2P, men man skulle åstadkomma en genomgående beräkningstid av O(1) för tråd 1, där det mesta arbetet utförs vilket skulle vara en önskvärd garanti för en väldigt snabb router.

Den optimala längden av perioderna x och y kan också förbättras. Jag valde $x = 1$ sekund och $y = 30$ sekunder eller 60 sekunder på grund av dess enkelhet. Det är möjligt att andra värden på dessa variabler kan ge bättre resultat. Det är lätt att även föreställa sig att andra värden kan ge drastiskt sämre resultat.

Algoritmen, så som den är beskriven ovan, använder sig av *Landmark*-fönster. En lätt modifikation av medelvärdeslistan ändrar algoritmen till att använda glidande fönster istället. En sådan testversion av programmet implementerades men inga experiment utfördes på grund av tidsbrist. Om detta skulle vara en förbättring eller inte kan därmed inte sägas. Nedan följer de förändringar av listan som krävs. Utöver det så är skillnaden att medelvärden aldrig reduceras till mätvärden, dvs. sista raden i algoritmen ovan tas bort.

```
Average://Beräknar ett aktuellt medelvärde
   För alla nycklar k i trädet,
      ak =  (ak·ck)+zk / ck+1 ,
      zk = 0
      Om ck < W så
        ck = ck + 1,
      Om ak < T1 så
        radera k ur listan
```

Där W = längden av det glidande fönstret räknat i antal mindre mätperioder x och

T_l = en lägre tröskel ($T_l < T$) vid vilken elementet raderas ur listan.

T_l behövs eftersom ett element kanske ligger och pendlar runt T och konstant kommer att raderas innan ett stabilt medelvärde kan etableras. *Landmark*-versionen har inte detta problem eftersom ett element tas bort högst en gång per y sekunder.

3.1.5. Jämförelse med en naiv implementation

För att verkligen uppskatta vad det är bloomfilter erbjuder oss så tänker jag här jämföra med en teoretisk implementation med balanserade sökträd. Medelvärdeslistan i min implementation får även den sägas vara ganska naiv så jag betraktar bara skillnaden i de två första stegen. Jag väljer att här jämföra med sökträd därför att de har logaritmisk söktid.

Sökträd har en minnesanvändning på O(n). För att kunna jämföra mellan flöden så måste listan spara information om IP-adress och port, för källan och destinationen. Totalt krävs 12 byte per flöde. Utöver detta måste ett träd också spara ett antal pekare för barnnoder och föräldranoden. Det beror på vilken struktur man använder men jag kan nämna att ett Rött-Svart träd skulle behöva tre pekare om 4 byte, dvs. ytterliggare 12 byte per element. Bloomfiltret å andra sidan använder runt 1,25 byte (10 bitar) per element för att minimera chansen för falska positiva. I första steget handlar det alltså om minst en faktor 10 i minnesanvändningen vi vinner på att använda bloomfilter. Räknar vi dessutom in pekarna för ett Rött-Svart träd så blir det en faktor 20.

I det andra steget är vi intresserade av en IP-adress och en räknare. I en ideal implementation hade 1-byte-räknare använts. IP-adressen kräver 4 byte och summan blir alltså 5 byte per element. Med pekare blir det totalt 17 byte i ett Rött-Svart träd. Counting bloomfilter använder tio räknare per IP-adress vilket summeras till 10 byte per element. En inte lika imponerande prestandavinst eftersom minnet mest tas upp av minnesreferenser i trädimplementationen.

Anledningen till att man inte kan använda exempelvis vektorer för att slippa pekare är eftersom att de har linjär söktid vilket hade varit på tok för långsamt. Även logaritmisk söktid är för långsam när hastigheten blir hög. Dessutom lider sökträd av många minnesreferenser. Flaskhalsen är som sagt just precis minnesåtkomsterna, vilket motiverar önskan att implementera algoritmen i SRAM. Bloomfilter erbjuder vad som kan tyckas vara den perfekta kompromissen mellan minnesanvändning och antal minnesreferenser (konstant antal). Allt som krävs är att vi tillåter att algoritmen med en liten sannolikhet levererar fel svar emellanåt.

3.1.6. Relaterat arbete

Andra personer har de senaste åren försökt utnyttja trafikens flödesmönster för att identifiera P2P och annan trafik. De arbeten som jag nämner här är de som jag känner till.

Remco van de Meent, Aiko Pras, "Assessing Unknown Network Traffic" [21]

van de Meents och Pras idé är att identifiera inducerade flöden som annars kanske inte skulle kategoriserats korrekt. Man ger som exempel en FTP-överföring där en kontrollförbindelse över port 21 inducerar en överföringsförbindelse över port 22 där själva datan skickas. Man tittar endast på pakethuvudet och baserar sin identifikation på en jämförelse med de välkända portarna. Deras algoritm är inte anpassad för att användas i realtid. Experimenten som utfördes på ett universitetsnätverk av cirka 2000 uppkopplade studenter visar slutligen att deras algoritm endast ger en marginell förbättring över vanlig portklassificering.

Kim et al., "Towards Peer-to-Peer Analysis Using Flows" [6]

Även hos Kim et al. använder man sig i hög utsträckning av portklassificering. Om någon av IP-adresserna utnyttjar en port som finns i deras P2P-portlista så klassificeras flödet som P2P. Denna lista över P2P-portar genereras genom en ingående analys av paketen i dumpfiler över trafik. Detta gäller potentiellt även paketdatan. Själva identifieringen sker i realtid och lyckas identifierea en stor del P2P på universitetsnätet där experimenten utfördes. Man lyckas likväl inte identifiera flöden där båda parter använder tidigare osedda portar och sorterar bort trafik som passerar på välkända portar för andra tjänster än P2P.

Wagner et al., "Flow-Based Identification of P2P Heavy-Hitters" [22]

Här har man implementerat sin algoritm för realtidsundersökning av Netflow-data. Netflow är något som Ciscos routrar använder. En del av trafiken väljs slumpmässigt och dess flödesrepresentation skickas som en UDP-ström för analys (till en av administratören bestämd destination). Det faktum att Netflow inte analyserar all trafik utan bara en delmängd skapar från början möjligheten för fel och falska negativa. Det är denna UDP-ström man har analyserat i den experimentella delen.

Algoritmen baseras även här på portklassificering. Man motiverar det genom att om en *peer* använder en okänd port så kommer den fortfarande ofta att kommunicera med andra *peers* som använder standardportarna. Man sparar under en längre tid (en timme) vilka portar varje *peer* har kommunicerat över och klassificerar en *peer* som P2P om den har ett flöde som utnyttjar en P2P-port under denna tid eller om den potentiellt kommunicerat med en P2P-identifierad *peer*.

För att bekräfta pålitligheten av sin algoritm så introducerar man även en valideringsmetod i tre steg. Först kontrollerar man att den *peer* man vill kontrollera är tillgänglig med ett ICMP-eko, en pingförfrågan. Sedan försöker man skapa en förbindelse mot den port man misstänker vara P2P med TCP. I sista steget försöker man faktiskt skapa en förbindelse över det P2P-protokoll man misstänker att förbindelsen använder. Något som inte fungerade för Bittorrent eftersom det kräver att man har kännedom om den fil som delas via torrentfilen.

Den absolut största svagheten anser jag vara att Wagner et al. helt ignorerar alla flöden som har en eller båda portarna utanför intervallet 1024-30000 för att undvika falska positiva. Det framkom att den mesta P2P-trafiken sker i detta intervall. Skulle deras algoritm användas i stor skala skulle P2P-nätverken dock säkerligen anpassa sig genom att potentiellt skicka all trafik över välkända portar mellan 1 och 1024, något jag förutsätter som en möjlighet i min egen analys.

Karagiannis et al., "Transport Layer Identification of P2P Traffic" [19]

Slutligen har vi då Karagiannis et al. som försöker identifiera P2P-trafik oberoende av vilka portar den färdas över. Algoritmen är dock inte anpassad för realtidsbruk. Man tittar bara på pakethuvudet och jämför de resultaten senare med en analys som baseras på de första 16 byte:sen av paketens data där man söker efter kända bitsträngar som skickas i P2P-protokoll. Man lyckas identifiera en stor andel P2P-trafik och tidigare okända protokoll men nämner att kryptering av data innebär att en del av resultaten inte kunde verifieras.

Deras identifiering har två huvudsakliga faser. I den första fasen så identifierar man de IP-adresser som har både ett TCP-flöde och UDP-flöde mellan sig. Sex av nio protokoll i försöken använder sig av både TCP och UDP, bland annat Bittorrent, Direct Connect och Gnutella. I den andra fasen så betraktar man alla flöden relaterade till adresser där antalet portar som används är lika med antalet unika IP-adresser. Man noterar här att exempelvis webbtrafik har en högre andel portar än IP-adresser eftersom en webbläsare initialt öppnar flera förbindelser för att ladda ner sidans material parallellt. Antalet portar överstiger i det fallet antalet unika IP-adresser och undviker därmed felklassificering i det fallet. För att minimera andelen falska positiva så utesluter man flöden vars portar och i viss mån även beteende överensstämmer med en del välkända tjänster som mejl, e-post, FTP, SSL och DNS inom TCP. Man noterar även att det inte finns något som hindrar P2P-klienter från att använda sig av dessa portar, vilket jag nämnde tidigare. Om en IP-adress i ett flöde är klassad som P2P så blir den andra adressen i flödet också klassat som P2P. På samma sätt så markeras

IP-adresser som kommunicerar med icke-P2P som att inte vara P2P. För IP-adresser som har många förbindelser (fler än 20) kan man med väldigt god precision klassificera den som antingen P2P eller inte.

Denna uppsats är det enda arbete jag känner till där man försöker identifiera P2P utan att analysera vare sig paketdata eller portnummer av P2P-trafik. Att man jobbar baklänges, det vill säga först klassificerar en stor del av trafiken och sedan sorterar bort icke-P2P-trafik som är enkel att identifiera gör det möjligt att identifiera tidigare okända protokoll. Något som är viktigt om en algoritm ska kunna användas i framtiden med idag outvecklade protokoll. Algoritmen är däremot som sagt inte tänkt att användas i realtid och kan inte modifieras utan väldigt stora ingrepp eftersom den grundar sig på en stor mängd jämförelser vilket innebär en stor mängd minnesreferenser och beräkningar.

3.2. Implementering och tillvägagångssätt

Här förklaras metodiken bakom undersökningarna samt hur algoritmen implementerades.

3.2.1. Implementering

Algoritmen implementerades i C++. Bloomfilter och counting bloomfilter samt det Rött-Svarta sökträdet som användes som medelräknare implementerades helt på egen hand, med bortseende från hashfunktionerna i bloomfiltren som jag använde ett färdigt litet bibliotek med diverse funktioner för [27]. För att fånga upp TCP-paket på nätverket och läsa dumpfiler av trafik användes libpcap [24].

Storleken på bloomfiltret och CBF:et dimensionerades för att klara av totalt hundra tusen flöden (128 kB) och tio tusen IP-adresser (256 kB) respektive. Eftersom ett binärt träd användes för medelvärdeslistan var storleken inte konstant. Men den totala storleken skulle inte överstiga 1 MB utan att även överstiga hundra tusen flöden.

I fallet då trafik fångas i realtid från nätverket så användes två trådar så som pseudokoden visar. Tråd 1 är då en callback-funktion som kallas när ett paket fångas. När trafik lästes från dumpfiler användes dock endast en tråd. Dumpfilerna sparades av TCPdump och då sparas tiden då paketet ankom i dumpfilen. Detta användes för att hålla reda på var på tidslinjen programmet befann sig och om den borde nollställa bloomfilter osv. Eftersom TCPdump använder pcap så är denna metoden även applicerbar i realtidsmätning och jag implementerade även en sådan version av programmet. Dock föredrog jag att använda två separata trådar för att sprida ut arbetet över tid och inte bara jobba när ett paket fångades. Alla

resultat som redovisas är dock gjorda på dumpfiler, men skulle även ha kunnat göras i realtid givetvis.

Synkroniseringen mellan trådarna valde jag att hantera genom att låsa datastrukturerna. Det är inget problem för de experiment jag gjorde i realtid eftersom belastningen var så låg. I en verklig implementation med betydligt högre hastigheter skulle synkroniseringen nog behöva närmare eftertanke. Det borde gå att lösa. Man kan tex. tänka sig att man använder en dubbel uppsättning datastrukturer och helt enkelt byter ut dem varannat intervall.

För att kunna få en bättre bild över trafikmönstret så raderades aldrig nycklar ur medelvärdeslistan. Jag valde att behålla alla som någon gång hamnar i listan för att kunna se hur trafiken beter sig mellan två toppar.

3.2.2. Mätdata

De antal flöden som krävdes för att hamna i medelvärdeslistan sattes till 2. Mätdatan samlades från en dator åt gången med undantag från några där mätdatan samlades från en dator som agerade NAT-router åt en annan dator. Det hände att vissa IP-adresser förutom de lokala hamnade i listan men de sorterades bort från diagrammen. Detta eftersom dessa datorer är endast de som testdatorn hade kontakt med och inget kan sägas om dem egentligen.

Datan samlades in av mig, men några stycken även av två andra studenter. Jag försökte få data ifrån så många olika typer av trafik jag kunde tänka mig, P2P, Webb, mejl, FTP, onlinespel, VPN osv. I de flesta fallen förekommer flera, till exempel Webb, mejl och instant messaging-trafik för de flesta webbtesterna. Detta eftersom dessa vanligen körs samtidigt. Dessutom så söker diverse program man normalt inte relaterar till Internettrafik efter uppdateringar, så som Windows Update eller liknande och kan därför ha påverkat resultaten i en väldigt begränsad utsträckning.

Webbsurfning och FTP-överföringar genomfördes med Firefox v2. Instant Messaging skedde med programmen Adium (Mac OS X) och Pidgin (Windows XP). Mejl hämtades via Thunderbird v2 eller via webbmejl. P2P representerades av Bittorrentklienterna Transmission v0.96 (Mac OS X), µTorrent v1.7.5(Windows XP) och Blizzard Downloader(Mac OS X).

Datan som samlades in var som sagt med hjälp av TCPdump och det gjordes över varierande tider, ofta 10 minuter eller 200 000 paket. Således är tidsspannet varierande i de olika diagrammen. 200 000 paket valdes eftersom i höga hastigheter så växer dumpfilen snabbt, även när endast TCP-huvudet sparades. I något fall analyserades drygt fyra miljoner paket, för att få en bild av en Bittorrent-överföring från början till slut. Om inget annat nämns,

är det min egen trafik jag analyserat och trafiken registrerades på samma maskin som skickade och tog emot den. Maxhastigheten för uppkopplingen där överföringarna skedde låg på 10 Mb/s upp och ner.

3.3. Resultat

3.3.1. redovisar de positiva resultaten. 3.3.2. visar de resultat som var klart negativa, och talar kort om varför och hur Bittorrent kan undgå att identifieras.

3.3.1. Webbtrafik och Bittorrent

Som vi kan se i diagram 3.3.1.1. så är vanlig webbtrafik långt ifrån stabil i antalet flöden per sekund. Även om det någon sekund registreras mer än två flöden så kommer varken Surf 1 eller Surf 2 upp i ett medelvärde av två. Det ska nämnas att Surf 1 inte är utförd av mig själv.

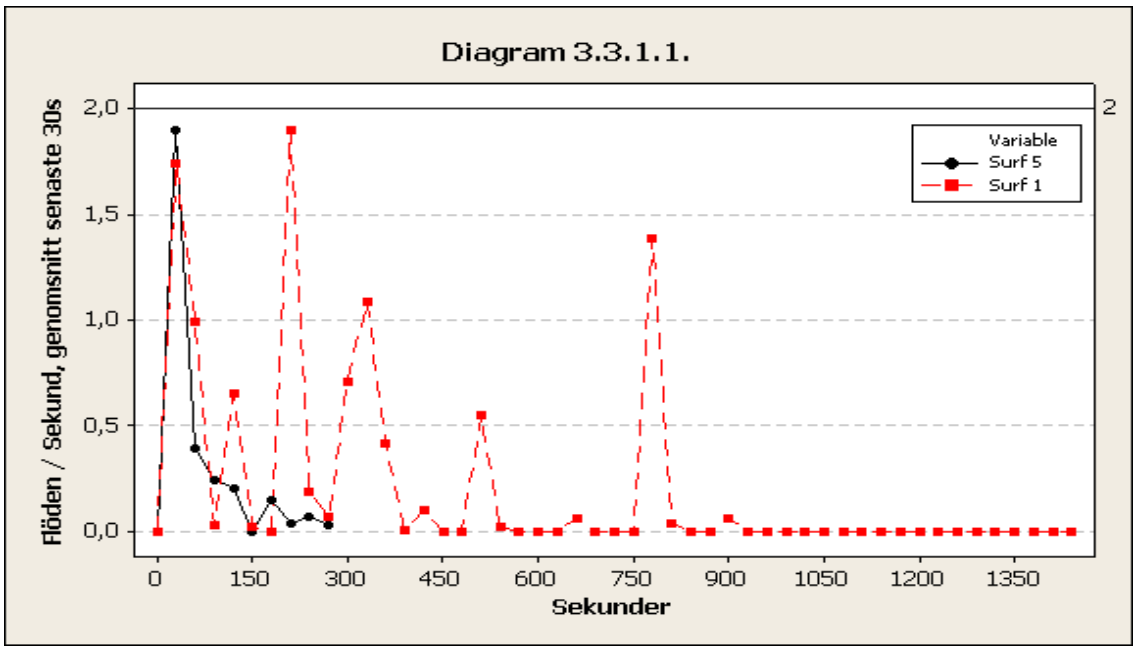


Diagram 3.3.1.1.: Webbsurfning utförd av två olika personer.

I diagram 3.3.1.2. ser vi den första Bittorrent-datan. Hastigheten var mycket låg, cirka 10 kB/s ner och 60-70 kB/s upp. Svärmen bestod bara av 4-5 *peers*. I samma diagram finns också två kurvor över webbsurfning. I Surf 2 försökte jag klicka på länkar ofta och snabbt i ett försök att höja antalet registrerade flöden, vilket också lyckades. Surf 3 är främst en FTP-överföring.

Precis som i diagram 3.3.1.1. lyckades inte surfningen hålla ett stabilt flödesantal. Detta stämmer väl överens med vad jag trodde om webbtrafik. Bittorrent-trafiken lyckas dock

nästan uteslutande hålla sig över 2 flöden per sekund i genomsnitt, detta trots den väldigt låga hastigheten.

Ännu ett exempel på en väldigt långsam Bittorrent-överföring kan ses i diagram 3.3.1.3.. I mitten av överföringen kan man se att antalet flöden per sekund dyker mot noll. Detta beror på att jag råkade stänga av datorn och den fick därmed söka upp och ta kontakt med *peers* igen efter att jag startat upp den. Svärmen bestod av cirka 13 *peers* och hastigheten låg på 25 kB/s ner och 4 kB/s upp i genomsnitt under de två timmarna. Även denna lyckas hålla sig ganska konstant över två flöden per sekund.

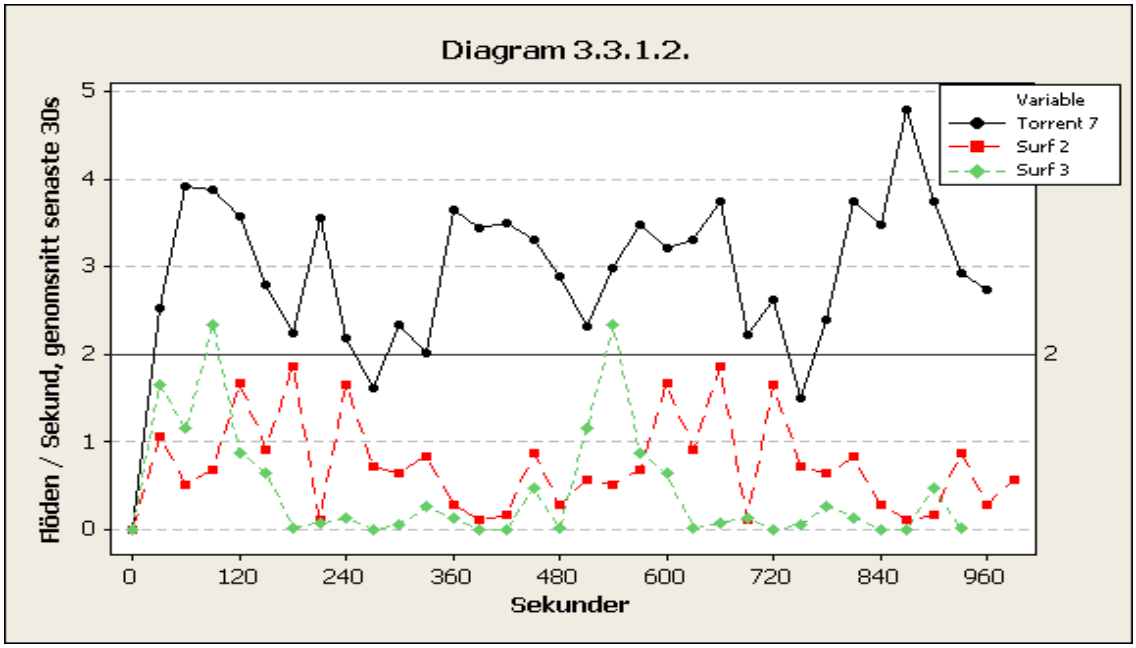


Diagram 3.3.1.2.: Webbsurfning i jämförelse med långsam Bittorrent.

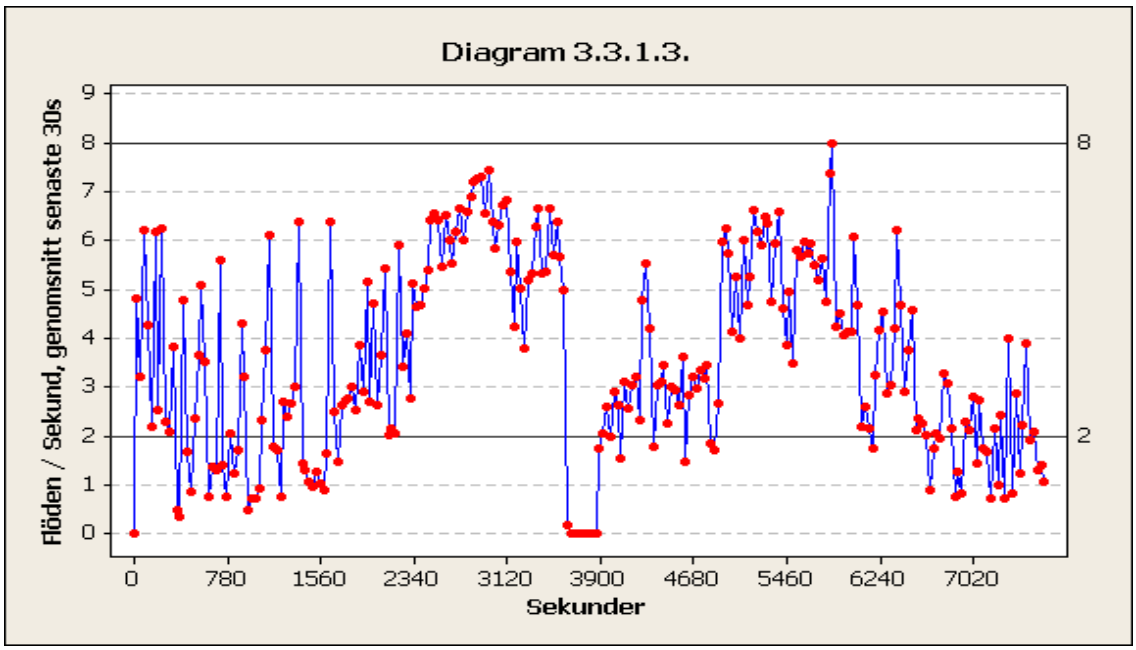


Diagram 3.3.1.3.: Långsam Bittorrent.

När det gäller snabb Bittorrent är antalet flöden per sekund mycket högre. I diagram 3.3.1.4. ser vi exempel på tre snabbare överföringar. Torrent 4 är inte data insamlad av mig själv, men enligt uppgift gick hastigheten kraftigt ner mot slutet av perioden och det rörde sig främst upp uppladdning. Torrent 5 höll en hastighet av cirka 100 kB/s och svärmen bestod av 20 *peers* ungefär. Torrent 6 är den trafik som genererades av Blizzard Downloader, som används för att skicka ut uppdateringar för spelet World of Warcraft. Vad som är unikt för den jämfört med andra Bittorrentklienter är att den samtidigt laddar ner via http parallellt med Bittorrent. Hastigheten låg på 1 MB/s och av det stod http-delen för ungefär 90 %. P2P-delen är alltså jämförbar med Torrent 5. Efter ett tag tvångsavslutades programmet, vilket förklarar den kraftiga minskningen av antalet flöden vid 300 sekunder.

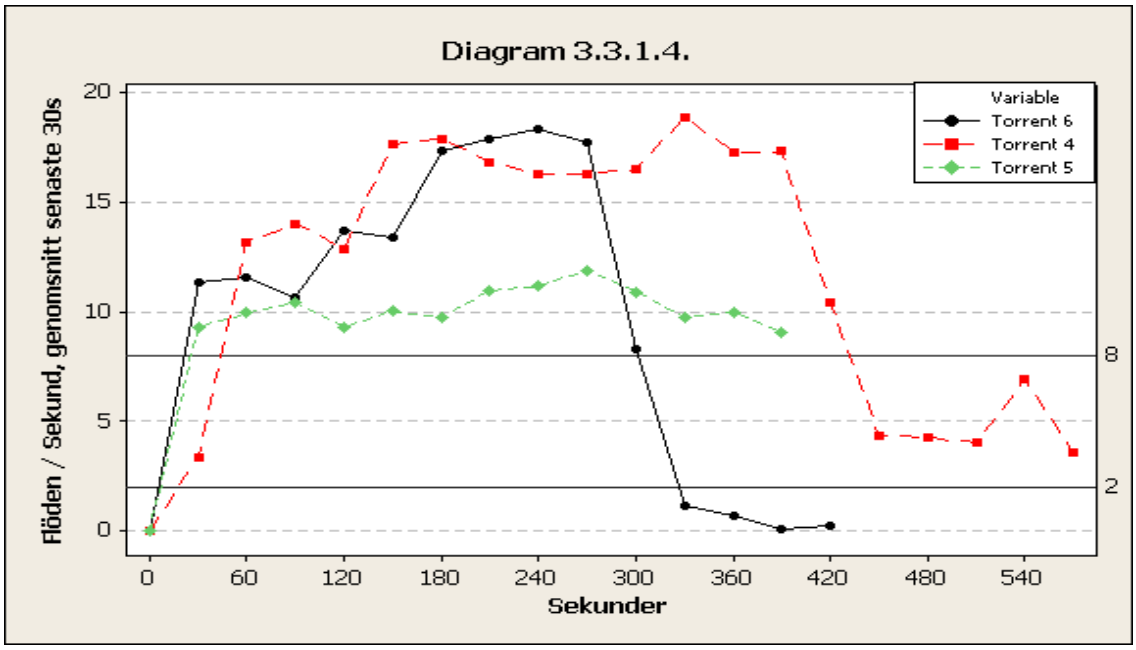


Diagram 3.3.1.4.: Snabbare Bittorrent, 100 kB/s - 1 MB/s.

Vad som är gemensamt för serierna i diagram 3.3.1.5. är att de hade alla väldigt stora svärmar. Torrent 1,2 och 3 hade svärmar på nära 2000 *peers* och Torrent 8 hade runt 400 *peers*. Likaså gav alla upphov till väldigt höga hastigheter. De låg alla stabilt på 1 MB/s ner och flera hundra kB/s upp. Som vi ser innebar detta även väldigt höga genomsnittsvärden av antal flöden per sekund.

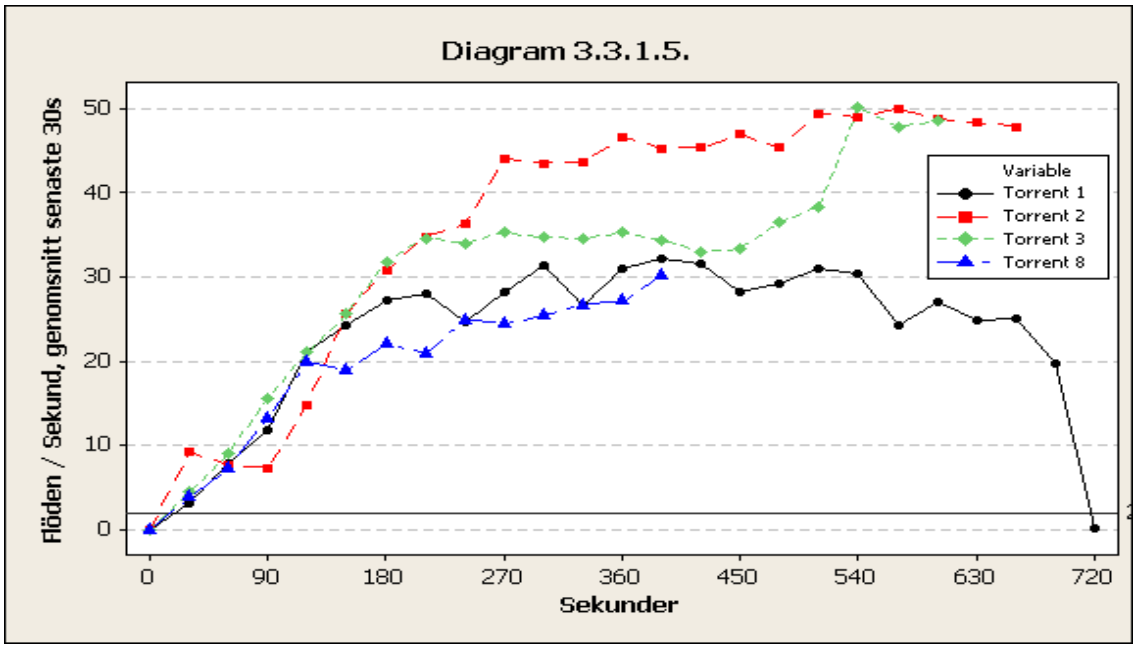


Diagram 3.3.1.5.: Väldigt snabb Bittorrent, runt 1 MB/s och stora svärmar.

3.3.2. Felkällor och metoder för att undvika upptäckt

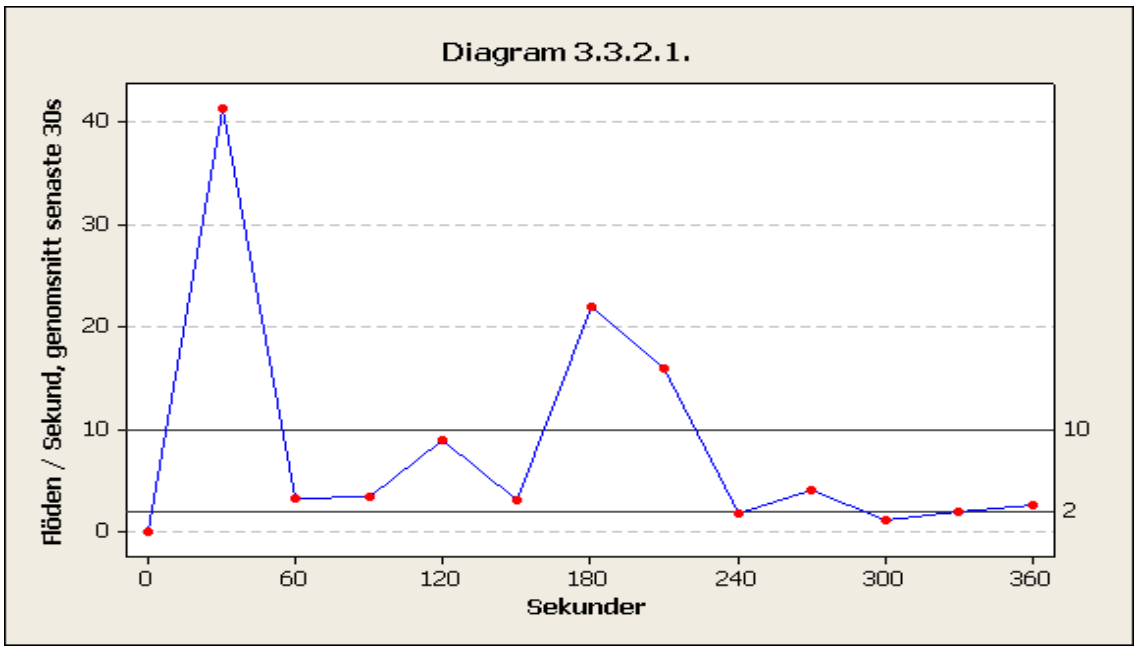


Diagram 3.3.2.1.: Samtidigt öppnande av 23 bokmärken i webbläsare.

Jag misstänkte att om man samtidigt öppnade ett stort antal webbsidor skulle man ge upphov till tillräckligt höga antal flöden att genomsnittet skulle kunna förväxlas med P2P. Resultatet kan observeras i diagram 3.3.2.1. där jag samtidigt öppnade tjugotre bokmärken i Firefox och kort därefter öppnade dem allihop igen. Som kan ses så är den första toppen ungefär dubbelt så hög som den andra. Men den andra toppen är utspridd över sextio sekunder

istället för trettio. Detta kan ses som ett mycket bra exempel av fel som *Landmark*-fönster kan generera när mätdata hamnar mitt emellan två fönster.

Ett annat sätt att generera en liknande typ av resultat som i 3.3.2.1. skulle vara om man analyserade trafiken bakom en NAT-router. Om nätverket bakom är stort nog och tillräckligt många människor surfar eller liknande, som ett cybercafé, kan antagligen en jämnare kurva uppnås. Speciellt om många av personerna samtidigt klickar på länkar så kommer toppar i stil med den i 3.3.2.1. troligtvis uppstå. Populära servrar skulle också kunna få ett liknande mönster.

Det finns även metoder för att dölja all flödesrelaterad information. En sådan metod är att skicka trafiken först genom en annan dator med hjälp av en VPN-tunnel alternativt SSH. I diagram 3.3.2.2. ser vi att en ganska snabb Bittorrent-överföring blir i det närmaste osynlig när den skickas via en VPN-tunnel. För detta test använde jag två datorer, den ena agerade NAT-router för den som använde VPN-tunneln. Själva trafiken registrerades sedan hos routern.

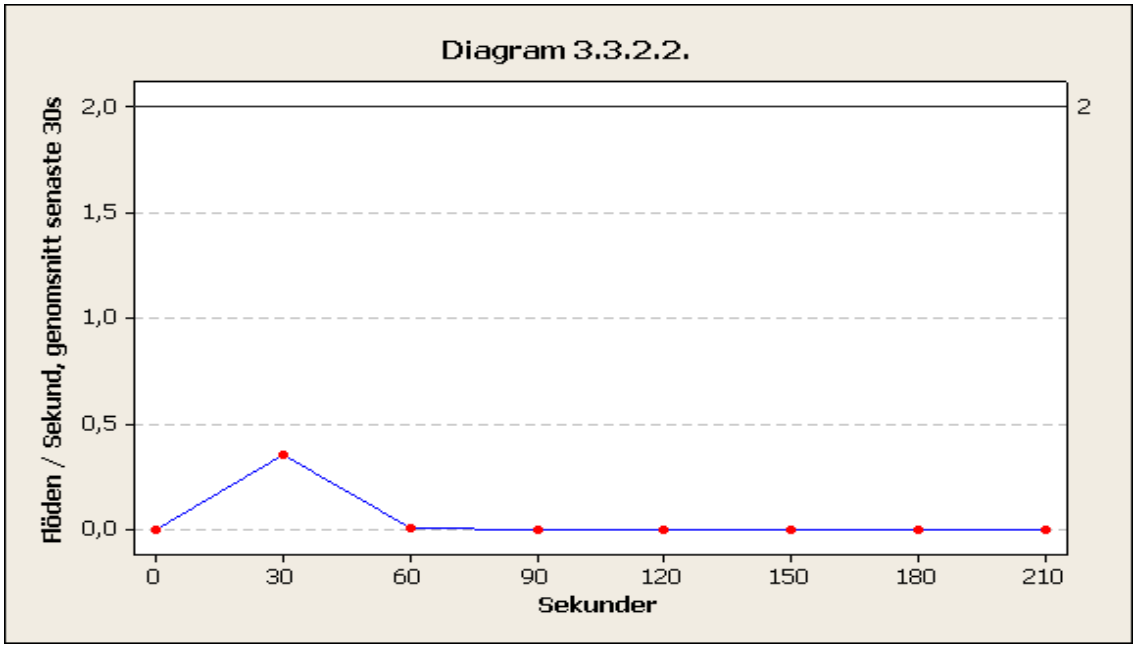


Diagram 3.3.2.2.: Bittorrent-trafik över en VPN-tunnel via GRE-protokollet.

Svärmen var på 20 *peers* och hastigheten låg på 300-400 kB/s både upp och ner. Som en jämförelse kan vi titta på Torrent 5 i diagram 3.3.1.4., som är exakt samma Bittorrent-överföring fast efter jag stängde av VPN. Efter att jag stängde av VPN lyckades den inte uppnå samma hastighet vilket antagligen beror på att den då hamnade bakom en brandvägg som var aktiv. Jag tittar endast på TCP-trafik och eftersom VPN i detta fall utnyttjade GRE-

protokollet så blir det naturligtvis i det närmaste osynligt. Endast en kontrollström gick över TCP. Men eftersom all trafik skickas mot VPN-servern skulle det högst ge upphov till ett flöde precis som andra klient-server-applikationer.

För onlinespel där latency spelar en väldigt stor roll så finns det en mycket liten risk för falska positiva. Min implementation grundade sig på TCP-trafik och jag utförde experiment med spelet World of Warcraft som använder TCP. Medelantalet flöden var mindre än 0.01 i varje intervall och jag valde därför att inte visa datan som diagram.

3.4. Slutsatser

Enligt de situationer jag analyserat så uppträder Bittorrent med flera flöden per sekund stabilt övre längre perioder, medan annan trafik än P2P inte gör det. Webbtrafik i synnerhet präglas av små toppar med längre dalar emellan. Enda gången det blir en fråga om falska positiva är när många webbsidor öppnas samtidigt, eller när många personer surfar samtidigt bakom en NAT-router. I åtminstone det första fallet skulle det antagligen inte betyda allt för mycket om man för någon minut dirigeras med högre latency hos sin leverantör. Tiden det tar att öppna tjugo sidor, samt tiden det tar att läsa tjugo sidor är sådan att det antagligen inte spelar någon roll ifall det tar hundra millisekunder extra att få kontakt med servern.

Servrar, om de är populära, kan också misskvalificeras. Men eftersom Internetleverantörer ofta i sina avtal specificerar att servrar är förbjudna eller endast tillåtna för privat bruk, kan det kanske snarare ses som en positiv bieffekt att sådan trafik inte heller prioriteras.

Ett spel, som är väldigt beroende av latency och att ”dirigeras rätt”, skulle ge upphov till maximalt ett flöde per sekund. Det är trots allt en enkel klient-server-applikation. Det är dessutom redan naturligt att stänga ner så många andra program och tjänster som möjligt för att frigöra så mycket resurser som möjligt (processor, minne) till spelet och för att minimera latency i sådana applikationer.

Bittorrent verkar ge upphov till väldigt många flöden så länge som hastigheten är ganska stor, cirka 100 kB/s eller mer. Nu när till och med villor ute i glesbygden har tillgång till 8 Mb/s ADSL är hastigheter av 1 MB/s på P2P-överföringar absolut inte reserverat för de med bäst uppkoppling längre. Därför spelar det antagligen inte så stor roll om långsam P2P-trafik inte identifieras. Den snabba trafiken som också är den dyraste identifieras med stor sannolikhet.

Det är också inte heller svårt att undgå upptäckt. Genom att använda till exempel en

VPN-tunnel kan man med 100 % sannolikhet undgå identifiering av sin Internetleverantör. I mina försök använde jag en VPN-tunnel till företaget Relakks [25] som erbjuder VPN i anonymiseringssyfte mot en månadskostnad. Men oavsett vad för VPN man använder så måste flödena ”sättas fria” någonstans för att kunna nå destinationerna. Där är det möjligt att identifiera trafiken. Det är dessutom inte helt omöjligt att tänka sig att Relakks eller andra företag som erbjuder VPN skulle vara intresserade av att prioritera trafiken olika beroende på typ.

En liknande metod vore att använda ett P2P-nätverk som Tor eller Onion för att dölja sitt trafikmönster. Tor fungerar så att ens trafik krypteras och skickas genom ett antal *peers* innan den skickas vidare mot sin destination. Samtidigt så delar man själv ut en del av sin egen bandbredd för att andra ska kunna vidarebefordra trafik genom ens dator. Jag har inte gjort några tester med Tor, men jag tror att vid låg belastning kommer mönstret att likna VPN väldigt mycket. Men skulle det vara så att hastigheten blir hög, så kanske antalet Tor-*peers* man är uppkopplad mot kommer att ge ett mönster liknande Bittorrent. Fast man skulle antagligen behöva vara uppkopplad mot ganska många Tor-*peers* för att detta skulle kunna inträffa.

I min implementation använde jag ett gränsvärde på 2 flöden per sekund för att initialt misstänka P2P. Ett högre gränsvärde skulle höja tröskeln för falska positiva, men även risken att långsam P2P inte identifieras. I vissa av diagrammen, där mönstret kan anses vara otydligt, har jag markerat 2 och 10 med en horisontell linje. Om antalet flöden håller sig stabilt över 10 anser jag att det säkert är identifierat som P2P. Dessa två linjer skulle behöva förenas någonstans emellan 2 och 10 flöden per sekund. Det optimala värdet av detta gränsvärde kan antagligen bara finnas genom omfattande experiment med verklig trafik hos en leverantör.

Tester i högre hastigheter skulle också vara en nödvändighet för att verifiera prestandan hos algoritmen. Eftersom jag har varit begränsad till 100 MB/s (hastigheten på det lokala nätverket) har jag inte kunnat genomföra några relevanta experiment för att bekräfta effektiviteten. Trots det är jag säker på att, med hjälp av de förbättringar som jag föreslog i kapitel 8, algoritmen kan göras snabb nog för att klara av även hastigheter uppemot 40 Gb/s i realtid. Det är helt klart att den rent storleksmässigt kan implementeras i SRAM.

Referenser

Alla refererade webbsidor finns sparade och kan skickas mot begäran.

Litteratur

[1] Burton H. Bloom, *Space/time trade-offs in hash coding with allowable errors*, Communications of the ACM, 13(7):422-426, Juli 1970.

[2] Cristian Estan, George Varghese, *New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice*, ACM Transactions on Computer Systems, Vol. 21, No. 3, Augusti 2003, sid. 270–313.

[3] Andrew S. Tanenbaum, *Computer Networks*, 4th Edition, ISBN: 0-13-066102-3, Prentice Hall, 2002.

[4] Jeffrey Scott Vitter, *External Memory Algorithms and Data Structures: Dealing with Massive Data*, ACM Computing Surveys, Vol. 33, No. 2, Juni 2001, sid. 209–271.

[5] A. Gerber, J. Houle, H. Nguyen, M. Roughan, S. Sen, *P2P, The Gorilla In The Cable*, Proc. National Cable & Telecommunications Association (NCTA), Juni 2003.

[6] Myung-Sup Kim, Hun-Jeong Kang, James W. Hong, *Towards Peer-to-Peer Traffic Analysis Using Flows*, DSOM 2003, sid. 55–67.

[7] Jörn Altmann, Karyen Chu, *A Proposal for a Flexible Service Plan that is Attractive to Users and Internet Service Providers*, IEEE INFOCOM 2001.

[8] Ahmed Metwally, Divyakant Agrawal, Amr El Abbadi, *Duplicate Detection in Click Streams*, WWW: Proc. of the 14th international conference on World Wide Web, sid. 12–21, Maj 10–14, 2005, Japan.

[9] Y. Zhu ,D. Shasha, *StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time*, Proceedings of the 28th ACM VLDB International Conference on Very Large Databases, sid. 258-369, 2002.

[10] T. Peng, C. Leckie, K. Ramamohanarao, *Survey of Network-Based Defense Mechanisms Countering the DoS and DDoS Problems*, ACM Computing Surveys, Vol. 39, No. 1, Article 3, April 2007.

[11] L. Fan, P. Cao, J. Almeida, A. Broder, "*Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol*, IEEE/ACM Transactions on Networking 8:3 (2000), sid. 281–293.

[12] Michael Mitzenmacher, Compressed Bloom Filters, IEEE/ACM Transactions on networking, Vol. 10, No. 5, Oktober 2002

[13] Fan Deng, Davood Rafiei, *Approximately Detecting Duplicates for Streaming Data using Stable Bloom Filters*, SIGMOD 2006, Juni 27–29.

[14] Saar Cohen, Yossi Matias, *Spectral Bloom Filters*, SIGMOD 2003, Juni 9–12.

[15] J. Aguilar-Saborit, P. Trancoso, V. Muntes-Mulero, *Dynamic Count Filters*, SIGMOD Record, Vol. 35, No. 1, Mars 2006.

[16] Abhishek Kumar, Jun Xu, Li Li, Jua Wang, *Space-Code Bloom Filter for Efficient Traffic Flow Measurement*, IMC'03, Oktober 27–29, 2003.

[17] Rhea, S.C., Kubiatowicz, J, *Probabilistic location and routing*, Proceedings of INFOCOM 2002.

[18] Andrei Broder, Michael Mitzenmacher, *Network Applications of Bloom Filters: A Survey*, Internet Mathematics Vol. 1, No. 4, 2005: sid. 485–509.

[19] T. Karagiannis et al, *Transport Layer Identification of P2P Traffic*, IMC'04, Oktober 25-27, 2004, Italien.

[20] Alejandro López-Ortiz, *Algorithmic Foundations of the Internet*, ACM SIGACT News, Vol. 36, No. 2, Juni 2005.

[21] Remco van de Meent, Aiko Pras, *Asessing Unknown Network Traffic*, CTIT Technical Report 04-11, University of Twente, Nederländerna, februari 2004.

[22] A. Wagner et al, *Flow-Based Identification of P2P Heavy-Hitters*, International Conference on Internet Surveillance and Protection (ICISP), 2006.

Internet

[23] *Internet Assigned Numbers Authority*, Senast besökt: 2007-12-13, http://www.iana.org/

[24] *TCPDump / Libpcap*, Senast besökt: 2007-12-13, *http://www.tcpdump.org/*

[25] *Relakks*, Senast besökt: 2007-12-13, https://www.relakks.com/

[26] *Blizzard Downloader*, Senast besökt: 2007-12-13, http://www.blizzard.co.uk/wow/faq/bittorrent.shtml

[27] *General Purpose Hash Function Algorithms*, Senast besökt: 2007-12-13, http://www.partow.net/programming/hashfunctions/index.html

[28] Sarah Lai Stirland, *Comcast Using Malicious Hacker Technique Against Own Customers, New Report Says*, Senast besökt: 2007-12-13, http://blog.wired.com/27bstroke6/2007/11/comcast-using-m.html

[29] Peter Svensson, *Comcast blocks some Internet traffic*, Senast besökt: 2007-12-13, http://www.msnbc.msn.com/id/21376597/

[30] Chris Soghoian, *Comcast to face lawsuits over Bittorrent filtering*, Senast besökt: 2007-12-13, http://www.cnet.com/8301-13739_1-9802410-46.html?tag=nefd.blgs

[31] Brad Stone, *Comcast: We're Delaying, Not Blocking, Bittorrent Traffic*, Senast besökt: 2007-12-13, http://bits.blogs.nytimes.com/2007/10/22/comcast-were-delaying-not-blocking-bittorrent-traffic/

[32] Chris Soghoian, *Congressman to Comcast: Stop interfering with Bittorrent*" Senast besökt: 2007-12-13. http://www.news.com/8301-10784_3-9804158-7.html

[33] Eric Bangeman, *Advocacy group to FCC: Comcast's traffic blocking defense is bogus*, Senast besökt: 2007-12-13, http://arstechnica.com/news.ars/post/20071101-advocacy-group-to-fcc-comcasts-traffic-blocking-defense-is-bogus.html

[34] Eric Bangeman, *Comcast hit with class-action lawsuit over traffic blocking*, Senast besökt: 2007-12-13, http://arstechnica.com/news.ars/post/20071114-comcast-hit-with-class-action-lawsuit-over-traffic-blocking.html

[35] Stephen Withers, *Block P2P, Belgian court tells ISP*, Senast besökt: 2007-12-13, http://www.itwire.com/content/view/13351/53/

[36] Carey Greenberg-Berger, *Comcast Customer Uses "Unlimited Service*" Excessively, Gets Disconnected For A Year, Senast besökt: 2007-12-13, http://consumerist.com/consumer/comcast/comcast-customer-uses-unlimited-service-excessively-gets-disconnected-for-a-year-235585.php

[37] Chris Williams, *Surge in encrypted torrents blindsides record biz*, Senast besökt: 2007-12-13, http://www.theregister.co.uk/2007/11/08/bittorrent_encryption_explosion/

[38] *µTorrent*, Senast besökt: 2007-12-13, http://www.utorrent.com/faq.php

[39] *Azureus*, Senast besökt: 2007-12-13, http://azureus.sourceforge.net/faq.php

[40] Adam Livingstone, *A bit of Bittorrent bother*, Senast besökt: 2007-12-13, http://news.bbc.co.uk/2/hi/programmes/newsnight/4758636.stm

[41] ”*CacheLogic and Bittorrent Introduce Cache Discovery Protocol*, Senast besökt: 2007-12-13, http://torrentfreak.com/cachelogic-and-bittorrent-introduce-cache-discovery-protocol/

[42] John Leyden, *Germany enacts 'anti-hacker' law*, Senast besökt: 2007-12-13, http://www.theregister.co.uk/2007/08/13/german_anti-hacker_law/

[43] Adam Pasick, *LIVEWIRE - File-sharing network thrives beneath the radar*, Senast besökt: 2007-12-13, http://in.tech.yahoo.com/041103/137/2ho4i.html

[44] ”*Användning av P2P-program på LUNET*, Senast besökt: 2007-12-13, http://www2.ldc.lu.se/security/P2P-services.shtml

[45] Chris Williams, *Pirate Bay aims to sink Bittorrent*, Senast besökt: 2007-12-13, http://www.theregister.co.uk/2007/11/01/pirate_bay_new_protocol/

[46] Lawrence G. Roberts, *Routing Economics Threaten the Internet*, Senast besökt: 2007-12-13, http://www.internetevolution.com/author.asp?section_id=499&doc_id=136705&

[47] Grant Gross, *Study: Internet could run out of capacity in two years*, Senast besökt: 2007-12-13, http://www.macworld.com/news/2007/11/19/internetcapacity/index.php

[48] Tim Weber, *Criminals 'may overwhelm the web*' , Senast besökt: 2007-12-13, http://news.bbc.co.uk/2/hi/business/6298641.stm

[49] Gregg Keizer, *Dutch Botnet Suspects Ran 1.5 Million Machines*, Senast besökt: 2007-12-13, http://www.techweb.com/wire/security/172303160

[50] Peter Bowes, *Warner to start movie downloads*, Senast besökt: 2007-12-13, http://news.bbc.co.uk/1/hi/business/4753435.stm

[51] Burt Helm, Bittorrent *Goes Hollywood*, Senast besökt: 2007-12-13, http://www.businessweek.com/technology/content/may2006/tc20060508_693082.htm

[52] Andrew Orlowski, *RIAA sues the dead*, Senast besökt: 2007-12-13, http://www.theregister.co.uk/2005/02/05/riaa_sues_the_dead/

[53] *Static random access memory*, Senast besökt: 2007-12-13, http://en.wikipedia.org/wiki/Static_random_access_memory

[54] *Deep packet inspection*, Senast besökt: 2007-12-13, http://en.wikipedia.org/wiki/Deep_packet_inspection

[55] *Direct Connect*, Senast besökt: 2007-12-13, http://en.wikipedia.org/wiki/Directconnect

[56] *Gnutella*, Senast besökt: 2007-12-13, http://en.wikipedia.org/wiki/Gnutella

[57] *DC++*, Senast besökt: 2007-12-13, http://dcplusplus.sourceforge.net/

[58] *Botnet*, Senast besökt: 2007-12-13, http://en.wikipedia.org/wiki/Botnet

[59] Nelly Visanji, *En miljon svenskar använder Facebook*, Senast besökt: 2007-12-13, http://www.idg.se/2.1085/1.129990

[60] *Meeting the Challenge of Today's Evasive P2P Traffic, Service Provider Strategies for Managing P2P Filesharing, an industry white paper,* 2004, Sandvine Incorporated. http://www.sandvine.com

[61] *Bittorrent Inc*, Senast besökt: 2007-12-13, http://www.bittorrent.com/