**Finding Whispering Elephants Among Screaming Mice**

**Identifying Peer-to-Peer With Data Streams**

**Jonas Kalderstam**

Thesis for a diploma in computer science, 30 credit points,

Department of Computer Science,

Faculty of Science, Lund University

## Abstract

**Finding Whispering Elephants Among Screaming Mice**

**Identifying Peer-to-Peer With Data Streams**

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 file sharing 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 file sharing and other traffic types such as web browsing.

# Table of contents

Abstract 2

Table of contents 3

Translation 4

1. Introduction 5

1.1. Statement of purpose 6

2. Massiva datamängder och snabba dataströmmar 7

2.1. Olika typer av fönster 8

3. Datornätverk och TCP/IP 9

3.1. Transportnivån 10

4. Dataströmmar i relation till routers 12

5. Randomisering, när det lönar sig att vara lite glömsk 14

5.1. Bloom filter 14

5.1.1. Vikten av att välja goda hashfunktioner 15

5.2. Counting Bloom Filter 16

6. Peer-to-peer 19

6.1. Skillnaden mot Klient-Server 19

6.2. Napster 20

6.3. Gnutella 21

6.4. DirectConnect 21

6.5. Bittorrent 22

6.6. Botnät 23

6.7. Fildelning är olagligt, eller? 23

7. Trafikprioritering 25

7.1. Quality of Service 25

7.2. Några identifikationsmetoder 27

7.2.1. Portklassificering 27

7.2.2. Deep Packet Inspection 27

7.2.3. (Shallow) Packet Inspection 27

7.2.4. TCP-UDP:par identifiering 27

7.3. Hantering av stora trafikmängder 28

8. Identifiera P2P med hjälp av flöden 29

8.1. Utmaningarna 29

8.2. Medelvärdeslistan 31

8.3. Algoritmen 32

8.4. Möjliga förbättringar 32

8.5. Jämförelse med en naiv implementation 33

8.6. Relaterat arbete 34

9. Implementering och tillvägagångssätt 36

9.1. Implementering 36

9.2. Mätdata 36

10. Resultat 38

10.1. Webbtrafik och BitTorrent 38

10.2. Felkällor och metoder för att undvika upptäckt 41

11. Slutsatser 43

12. Referenser 45

12.1. Litteratur 45

12.2. Internet 46

# Translation

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

Maria, för din hjälp, ditt tålamod och för att du ständigt får mig att vilja förbättra mig själv.

Tobias, för din ovärderliga assistans vid implementationen och datainsamlingen.

Slutligen tack till min familj, i synnerhet min syster Åsa, för ändlösa kommentarer, synpunkter och tålamod.

## Introduction

The Internet has in its short life quickly evolved to an essential means for exchange of information. The speed of client connections have steadily increased from 14.4 Kb/s dial-up to speeds between 10 Mb/s and 1000 Mb/s. This has forced Internet service providers to increase their own capacity to satisfy customer needs. However, the last nine years have meant a revolution in information distribution with Peer-to-Peer. Once, users almost exclusively downloaded data, primarily text and images, while now they upload as much as they download. Simultaneously, the type of data has shifted to a large extent to mean large audio and video files. To handle this explosion of data, I.S.P.s are forced to prioritise some types of traffic during high load. Some go as far as to try and ban the use of P2P software. File sharers respond by encrypting their traffic and using non-standard ports. Even if a provider has no intention of limiting file sharing within their network, it might still benefit from identifying P2P traffic in order to maximise quality of service for vital services such as voice-over-IP. But to be able to identify P2P traffic, in real time, that is invisible to previous methods and at the speeds that the routers of providers operate in is a challenge.

Advances in several areas, especially communication and databases, has created the need for a fresh view of large amounts of data. By considering the data as a stream of information it becomes natural to create algorithms with O(n) time and O(log n) memory use. Algorithms with a complexity of O(n log n) are usually not considered much worse than O(n), since the logarithm is such a slow growing function. In the context of data streams however, it is not unusual for log(n) to quickly grow beyond 20. A potentially twentyfold decrease in performance is not something that can be ignored.

With a viewpoint grounded in the data stream model, I investigate in this paper the difficulties of identifying P2P, independent of protocol, in real time. The purpose has not been to identify potential violations of copyright laws. In my analysis I assume that no difference between sharing of illicit and allowed material can be observed.

I begin by accounting for the data stream model and quickly explain some points of TCP/IP. The randomised data structures, bloom filters, which are used to process the stream are introduced in chapter five. Chapter six to seven deal with P2P and traffic prioritisation. The algorithm I have used is explained in chapter eight together with some suggestions for improvements. Finally, the experimental procedure and results are presented in chapters nine, ten and eleven.

### Statement of purpose

The purpose of this thesis is to investigate the difficulties of identifying P2P traffic in real time, to develop an algorithm that could potentially be implemented in SRAM and to carry out experiments with said algorithm in an attempt to verify that it is able to identify P2P whilst minimising the risks of falsely identifying other traffic as P2P.

Since file transfers almost exclusively are carried out over TCP I have chosen to only consider that in my implementation but it is easily expandable to include UDP and all other kinds of protocols. The algorithm itself assumes nothing about protocol. It only depends on how you define a flow.

I expect that fast P2P traffic will give rise to many flows, while other traffic including fast HTTP will not. I also suspect that slower P2P traffic will still generate more flows than other kinds of traffic.

## Massive amounts of data and fast data streams

Information is often considered as a static mass, especially when housed in the form of a database. There are times when such a view is limited by what is possible and practical to do. The two primary examples are when the amount of data is huge, and when the data is streaming in very fast and has to be processed and re-transmitted or be lost forever. At these times the data is best viewed as a (potentially infinite) stream. Although data streams will mostly be considered from how they relate to computer networks in this paper, I will begin by consider their general representation.

A data stream can be viewed as an amount of lesser, potentially infinite, flows. How one flow is defined from another is dependent on the problem at hand. With respect to network traffic, one might define a flow as those TCP/IP packets originating from IP-address 1, port X, sent to IP-address 2, port Y. If the problem instead is related to reading files from disc, it might be defined as bytes belonging to file Z. In this last case, the stream would be all files on the entire disc. Every flow is itself composed of elements, which is the unit being manipulated e.g. TCP-packets or bytes read from disc. So by definition, a data stream could be any collection of data. The term is usually reserved only for data that is so huge and/or fast that it demands the processing algorithms to be small and fast.

One requirement is that the algorithms only read each element in the stream once[[1]](#footnote-2). Additionally, it is expected that the amount of memory used is small in relation to what is read. In other words, it is not possible to save a copy of each element for future reading in a table or similar.

As a short example consider the following: Nowadays it is common that during broadcast news a small strip of text scrolls at the bottom of the frame, telling of minor events around the world or stock market changes. For the viewer to know if he or she has already read all of the information contained in that strip, the viewer will have to be able to identify the first thing he or she read (assuming the information scrolls by more than once). So the viewer must be able to identify *copies*, when the strip starts to repeat itself.

The amount of information scrolling by during the news is limited enough that this is not a problem for most people but imagine for a moment if the amount of information in that scrolling text was enormous, say, the size of your local library. To be able to identify duplicates you'd be required to remember what you already read, or at least a summary of it. If we also assume that the news in the strip of text is printed randomly, we cannot concentrate our efforts to just remember a few, that is the first piece of text read. Instead of increasing the amount of information displayed, we can also increase its speed. Seeing every letter is not a problem, all you have to do is fixate your eyes on one point on the strip, but reading is more than just seeing letters. It takes a minute amount of time to process the word before a meaning is associated with the shapes seen by the eye. It takes additional time to associate this word with the previous word we read, and so on. Much like how the computer has to process information and store it. If the text scrolls fast enough, we might have trouble processing the meaning of it before we have to start reading the next sentence. Even if we were able to read all the words, if someone were to ask us to summarize what we read we would probably do a lousy job of it. The same problem is faced by computers when the amount of data grows larger and faster.

### Different types of windows

One way to tackle the problem is to only concentrate on a small part of the stream at any given time. If the stream is data being collected in real time, maybe it would appropriate only to consider the last X elements, or elements that were recorded in the last Y minutes. It can be said that the stream is viewed with the help of a small window. where only part of the stream is visible.

It can be done in a number of ways. The most obvious way is to forget everything every Y minutes, and basically start over. This is known as a *Landmark* window [[2](#Met05)]. The primary advantage is that such a solution is easy to implement. The drawback is that different kinds of errors or uncertainty can be introduced depending on what is being investigated in the stream. If we were to be interested in an event that would be spread out in time, it is possible that it would not be detected if it was recorded at the end of a window. In that case the event would be cut in half, and in none of the adjacent windows would we potentially detect it.

An alternative would be to only forget elements that actually are older than Y minutes. We eliminate the errors associated with window limits but instead introduce another difficulty. We are now forced to remember how old individual elements are in order to forget them at the right time. Care must be taken so that we are not forced to iterate of a list searching for old elements. This is called a sliding window [[2](#Met05)].

A compromise would be what is called a jumping window [[3](#Zhu02)]. Taking a *Landmark* window, we can divide it into smaller sub windows. While a new sub window is being filled, we remove the oldest. The actual processing takes place in the sub windows in between the two. It is a compromise of the simplicity of the *Landmark* window and the precision granted by the sliding windows.

## Computer networks and TCP/IP

As I have mentioned, flows can be defined in any number of ways. I will only deal with flows which can be defined in the transport level of the ISO Open Systems Interconnection reference model. This is the level where protocols such as TCP/IP lives. I do not concern myself with the higher levels dealing with application specific data because I assume it to be encrypted or simply to massive to process.

In addition to Transport Control Protocol (TCP), several other protocols are commonly used. For instance User Datagram Protocol (UDP), Realtime Transfer Protocol (RTP) and Realtime Control Transfer Protocol (RTCP). TCP is usually used when a reliable transfer of data is the most important aspect. It offers guaranteed delivery. Guaranteed in the sense that the computer will keep trying to transmit data that hasn't arrived at the destination. Data that becomes lost gives rise overhead and can subsequently diminish the speed of the transfer.

UDP on the other hand offers no error correction so data is only sent once. Data can be lost in the transfer, but is0generally transmitted quickly instead. It is commonly used for streaming audio/video and online gaming where a few lost packets won't mean the end of the world as long as the data has a steady flow.

RTP and RTCP is commonly used for multimedia because of their abilities to transfer data to several recipients at once. P2P applications generally use either TCP or UDP (or both). I will restrict myself to TCP in my implementation because that is the most common protocol.

Another difference between UDP and TCP is that UDP lacks ports. TCP is designed that in addition to source/destination IP addresses, a port must also be specified. The port is merely an unsigned 16-bit integer and does not have to be the same between source and destination. To keep it simple, many applications have been granted (or have annexed) a standard port for its use. A web server listens on port 80 and your web browser assumes this unless you specify otherwise. Packets with a designated destination port that doesn't match a corresponding server on the destination IP address are discarded.

The data is made up of packets because once a computer intends to transmit data over the network, it will first chop it up into a bunch of individual pieces. Each packet is usually 1500 bytes because of the Ethernet standard with which most networks are built. The TCP-specification itself allows for 64kB packets. A TCP packet consists of a head, and some data. The head contains information regarding source, destination etc.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 3.1.1: The structure of a TCP packet, the head is made up of bit 0 to 160/192.**

Because of the way the internet is constructed, packets can, and do, take different routes to their destination. It is then not surprising that they might arrive out of order compared to when they were sent. Routers update and order their internal map of neighbours based on their latencies in order to choose the best path for each passing packet. This is generally invisible to users and applications. However when one is interested in understanding what kind of traffic is flowing through a router, the differing paths of packets is a potential source of error. But because my algorithm isn't concerned with the application data, and hence, is invariant to the order of arriving packets, this isn't much of a problem. If all applications adhered to using standard ports, there would be no problem in identifying traffic since a simple look-up table would suffice. Many applications do not however, and in many cases for P2P, a completely random port is used. This means that we must consider all ports if we are to find all relevant flows.

## 4. Data streams in relation to routers

Say a new packet arrives at a router and all previous packets are saved in some kind of list, for instance a binary search tree. In order to verify if this packet belongs to a flow which has already been identified (which has saved packets in the list), we'd have to compare it with all the saved packets. For a search tree, the search cost will be log(n) on average. The logarithmic function grows slowly, but with the traffic volumes that are of interest it can easily grow to 20 or even 50 [[4](#Lóp05)].

Twenty or fifty operations might not sound so bad, but combined with the fact that router software is already operating near the limit [[4](#Lóp05)][REF$46,47], since it is sorting traffic at between 2 and 40 Gb/s, those mere twenty operations will quickly scale to the millions each second. A quick calculation easily demonstrates: if each packet is 1500 bytes (maximum by the Ethernet-standard), and the router handles traffic at 8Gb/s, the number of packages arriving each second is more than 600 000. Performing 20 operations on each package means 12 million operations each second. Many packets will naturally be smaller than 1500 bytes, meaning the number of packets more likely lies closer to a million per second, meaning 20 million operations per second.

So the router is forced to pass along each incoming packet and in addition do whatever we need it to do, in a single microsecond. Hence whatever we need it do to, it must be able to do in a few nanoseconds [[5](#Est03)]. With the numbers given previously, simply checking if the packet somehow already is a member of our list would mean 20 million operations, which clearly is impossible to achieve in a few nanoseconds today. There certainly isn't any time left for additional computations.

Besides the lack of time, we also face a lack of memory. At the speeds described above, the type of memory commonly used as primary memory in most personal computers, DRAM, is too slow [[5](#Est03)] [[6](#Vit01)]. The access time is at 10 nanoseconds or more. [[5](#Est03)]. We need a type of memory that can keep pace with the processor and the data stream. Such memory, SRAM, is already used in routers today because this is not a new problem for routers. It is also used inside the processors driving most of our personal computers. The time to read a registry in the CPU takes only a few nanoseconds [[6](#Vit01)]. The problem with SRAM is, its advantages in speed and power usage notwithstanding, that it is not as dense as DRAM. While your computer probably has several GBs of DRAM, it is unlikely to have more than one or two MBs at the most of SRAM in the CPU. At the time of this writing, the power house of server processors, the Intel Xeon, has a mere 16 MB of cache at its disposal.

If we again consider a 8Gb/s link where we as before want to be able to identify packets belonging to flows already seen. If a new flow is identified, we put it in a list or similarly. The router is placed at the end of an ISP and is directing traffic for its broadband customers. Every customer has an individual bandwidth of 1 MB/s both up and down and for simplicity's sake, we assume every customer is associated with 10 flows which do not change. There is a total of 1000 customers connected to the router, which conveniently makes 1GB (8Gb) per second. Once again, we naïvely place the information about the flows in some kind of sorted list. There are 10 000 flows total.

Flows are defined by their sources and destinations, which in turn are defined by their ports and IP-addresses. An IP-address requires 4 bytes, a port 2 bytes. Each flow can then be described by 12 bytes. To save them all in a list would require 120 000 bytes, about 118 KB which without a doubt is nothing exceptional even with SRAM. But in reality, flows don't stay the same. People surf the web, click on links, and do all sorts of stuff relating to new sites all the time. It is quite natural to assume that customers can be associated with some new flows, and not with some older ones anymore, every minute or so all depending on the behaviour of the customer. If you're reading an interesting article for 10 minutes, you're perhaps unlikely to be related to any new flows during that time.

So if we want to compare the flows at one time with another time, we have to save more than 118 KB. If we are interesting in being able to draw conclusions over a longer time period, the memory will probably run out pretty quick. The more we want to remember about the flows or the customer and the longer we want to remember, the fast the memory will be filled. Vitter, J. S. mentions in [[6](#Vit01)] a practical limit of O(log n) or O(polylog n) for the memory use. So that for n flows, we may only remember a log(n) amount of information.

To relate to what I wrote previously about jumping windows, we can note that we can save about nine seconds of data per MB. With this solution it is possible to save a couple of minutes worth of data but it would be quite costly ultimately impractical to save more than ten minutes worth in SRAM. Ideally, we'd like to use less than 12 bytes to save a flow, a lot less.

## 5. Randomization: when it payes to forget.

The question “is this a new flow?” can, as I have explained, not be answered exactly if we don’t have enough memory available to save enough information about each flow. There is also a limit on the amount of operations we are able to perform for each packet. It is definitely a must to stay within O(1) operations since even O(log n) will grow too large. The obvious solution is some form of hash table but this immediately collides with the memory constraints. We have to allocate a table large enough to hold all the potential flows which is a very large table indeed if we expect all computers to potentially speak with all other computers. A dynamic structure would be forced to resort the list, running into the constraint on O(1) operations.

We can save some space by allowing ourselves to forget. This does however introduce a certain amount of errors. Flows which have been previously been identified and forgotten will be counted once again, generating false positives. Likewise, a packet might be wrongfully assigned to an existing flow. This I designate a false negative. The designation of positive/negative is the answer to the question: “Is this a new flow?” These errors will affect the precision of the measurements. The algorithm can no longer be trusted to return a completely correct result, though with it might with certain probability of course. The question is how wrong the result will be for a certain input and how this affects our use of it. Randomized algorithms are practical to use when an almost correct answer is good enough and the answer is needed within a “reasonable” amount of time, as for example is the case for problems which are NP-complete. More important for our problem however is that if we allow for an answer that is “good enough”, we can drastically reduce the amount of data we are forced to save about each flow.

### 5.1 Bloom filter

A bloom filter[$REF 1], named after its creator Burton Bloom, is quite an elegant data structure. Since its creation during the seventies it has been used in an increasing amount of fields[$REF 18], even in computer communications. It has also been further developed to overcome some of its fundamental weaknesses and to accomplish different tasks. One version, which I use in my implementation, is called Counting Bloom Filter[$REF 11].

A bloom filter, from now on referred to as BF, is a bit vector of size *m* with two operations: add and check. In the beginning all bits are 0. For every element added to the BF, a total of k bits, determined by the hash functions h1-hk, are set to 1. A bit might possibly be set to 1 multiple times, which does not affect its value. When the check function is called with the input x, the bits h1(x),…, hk(x) are checked. If any of the bits are 0, there is no chance that x has been added to the BF. If all the bits were to be 1, there is a chance that it is a false positive since other elements might have been hashed to the same bits as x. Mitzenmacher has shown[$REF 12] that probability of this is minimized when the number of hash functions is $EQUATION, to $EQUATION, where *n* is the number of elements so far added to the BF.  We can thus calculate, for example, that for a BF of 1000 bits (*m*), and 160 elements (*n*), we reach an optimum with four hash functions with 5% risk of false positives.

The advantage of bloom filters is that only O(k) operations are required for insertion and search, and that amount of memory which must be allocated is not dependant on the size of the input elements but only on their numbers, e.g. O(n). The downside is of course the risk of false positives. The risk can be minimized to a desired level, or the size can be modified to give a certain error probability as above but depends ultimately on the hash functions used.

#### 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 likt en tärning är detta sällan möjligt. En god hashfunktion har *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 BF skall användas för dataströmmar.

För att ge ett väldigt enkelt exempel kan vi ta en hashfunktion baserad på modulus. Vi vill hasha några heltal z till värden från 0 till m. Vi definierar funktionen h(x) = z mod m. Detta är inte en god hashfunktion i de flesta fallen. En liten förändring i indata är 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 alltså inte fallet.

Det är inte svårt att föreställa sig indata som får en långt ifrån slumpmässig distribution. Välj exempelvis var m:e heltal, så hashas alla element till samma hashvärde med ovan definierade funktion, det sägs att de *kolliderar*. Detta är ett fundamentalt problem med hashfunktion som inte går att undvika. Även för en god (och med god menar 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 bloom filter är troligt ingen bra hashfunktion för krypteringsapplikationer[[2]](#footnote-3) eller för dataintegritetskontroll[[3]](#footnote-4).

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 den har tillgång till), till exempel vårt lokala hemnätverk. Det finns bara 255 möjliga IP-adresser för datorerna i detta nätverket[[4]](#footnote-5) 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 direkt att stöta på problem eftersom olika IP-adresser kommer att börja kollidera.

### Counting Bloom Filter

Ett CBF[11] skiljer sig från ett vanligt bloom filter genom att bitvektorn är utbytt mot en vektor av räknare. Precis som för ett bloom filter så används k st hashfunktioner. När ett värde y läggs till så inkrementeras räknarna som bestäms av h1(y),...,hk(y). Efter ett antal operationer är man intresserad av hur många gånger y har lagts till i CBF:et. CBF:et returnerar då den minsta av räknarna h1(y),...,hk(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 då en minsta av räknarna är den som kolliderat minst antal gånger och därför är den *mest* korrekta.

-BILD

Till skillnad mot ett vanligt bloom filter så stödjer dock ett CBF även bortagning. Om ett värde y läggs till i CBF:et m gånger så kommer räknarna h1(y),...,hk(y) >= m. Vi tolkar detta som att det existerar m stycken kopior av y i CBF:et. Vi kan nu ta bort en sådan kopia av y genom att minska räknarna h1(y),...,hk(y) med ett. Försöker vi åter ta bort y kontrolleras först (detta gjordes även förra gången, men då visste vi att y fanns i CBF:et) om någon av räknarna h1(y),...,hk(y) är nollställd. Om så är fallet, dekrementeras ingenting (vi kommer bara att minska antalet av andra värden som potentiellt kolliderat om vi dekrementerar räknarna). Vi gör även här den ytterligare definitionen att antalet element n betecknar antalet *olika* element, inte antalet *förekomster* av element.

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 bortagning 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. Det är inte tillräckligt för användningen i denna uppsatsen 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.

Precis som för bloom filter så finns det en risk för falska positiva i ett CBF. Ett värde z kan rapporteras finnas i CBF:et när så inte är fallet. Sannolikheten för det är dock den samma som för ett vanligt bloom filter, så som redovisats tidigare. Ett annat fel som CBF inför är risken att det värde som returneras när antalet y som finns i filtret kontrolleras. 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 tillsammans hashats till samma räknare som y.

Det är då intressant att veta hur stor sannolikheten är att det värde som returneras skiljer sig från det verkliga värdet med mer än *j*. Detta är väldigt beroende på hur många av varje element som kommer läggas till i filtret. Om vi definierar *pj* som sannolikheten att en kollision får räknaren att visa mer än *j* fel, kan vi göra följande observationer.

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

 (5.2.1)

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:  (5.2.2)

Sannolikheten att inkrementeras av exakt *ett* annat element (en enda kollision) är:  (5.2.3)

Således är då sannolikheten att *mer än en* kollision inträffar för räknaren *r* :  (5.2.4)

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

 (5.2.5)

Jag gör här förenklingen att mer än en kollision automatiskt gör att räknaren visar mer än *j* fel, vilket inte behöver vara fallet (vilket i sin tur minskar den totala sannolikheten för fel). Slutligen då, den totala sannolikheten att ett element *y* rapporteras finnas i CBF:et mer än *j* för mycket är lika med sannolikheten att *alla* räknarna som bestäms av h1(y),...,hk(y) rapporterar mer än *j* för mycket, vilket är lika med  (5.2.6)

Som ett specialfall kan vi sätta *pj* till 1 (varje felrapportering kommer att överstiga *j*, alternativt *j = 0,* alternativt vad är chansen att ett element uppges ha fel antal i filtret) och ut kommer ekvationen:

 (5.2.7)

Som nästan är lika med  (5.2.8)

Förutsatt att  (5.2.9)

Dvs chansen för falska positiva i ett bloom filter. Detta eftersom

 (5.2.10)

när *n* växer.

-Visa med exponentialfunktionens definitioner

Redan när *n* = 10 så är

 (5.2.11)

Den lilla skillnad som uppstår mellan de olika funktionerna kan förklaras av att Mitzenmacher i sin analys uppskattar sannolikheten med en *e*-funktion, och 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 bloom filter.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 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 möjliga.

### Skillnaden mot Klient-Server



**Figur 6.1.1: Klient-Server.**

Den vanliga metoden för att koppla upp datorer mot varandra har traditionellt sätt varit med klient-server. En dator agerar 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. Två aspekter framförallt är speciellt tydliga.

För det första så måste servern ha väldigt hög bandbredd ner mot klienterna. Populära tjänster så som Googles sökmotor tvingar 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 kopplas ner av någon anledning så försvinner den eller de tjänster som maskinen hanterade för sina klienter. Alltså är servern sårbar för angrepp och fel.



**Figur 6.1.2: Peer-to-Peer.**

Peer-to-peer å andra sidan förlitar sig inte på en enda server. Här agerar alla klienterna samtidigt server (även kallade *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 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 där med säkrare och snabbare för den enskilda klienten. Varje klient som har en komplett kopia kan i sin tur dela med sig till andra.

### 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 tillhandahöll information om andra uppkopplade användare och vilka filer de hade tillgängliga. Om filen man sökte efter 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 dess centraliserade struktur (och därmed kvalificerar det inte för verklig P2P i ordets sanna mening) med en server som alla klienter kopplade upp sig mot.

### 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[[5]](#footnote-6) 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 eftersom man överför en fil endast mellan två klienter åt gången.

### DirectConnect

DC kom ungefär 1999[55]. Det finns ett flertal tredjeparts klientprogram för protokollet[57]. Precis som Napster så finns 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 sökningsmöjlighet över de filer som delas ut av uppkopplade användare samt chatmöjlighet.

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 vidarebefodrar trafik på en viss port, blir passiva. De har därmed inte tillgång till samma omfattning 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[[6]](#footnote-7), 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 mot 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. Vilket också innebär att det finns en övre gräns på antalet användare för varje server, beroende på dess bandbredd och andra resurser så som processor osv.

### Bittorrent

Detta är det mest populära P2P-protokollet idag och beräknas ansvara för 30%[43] av all trafik över Internet idag. 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 som så 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 sökning 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 de 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. Dvs alla har olika delar.

Eftersom klienter kan dela med sig av 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 vanliga webbtrafik 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 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).

### 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. 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[[7]](#footnote-8), 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 användningsområden för botnät 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” att tjäna pengar.

Botnät är ett stort problem på Internet och det uppskattas att upp till 150 miljoner datorer[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 av ett P2P-protokoll[10][58]. Vilket motiverar av säkerhetsskäl att identifiera sådan P2P-trafik.

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

Jag vill göra klart att jag inte skrivit denna uppsatsen 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 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 med de anledningarna jag har baserat mina avsikter.Trafikprioritering

Information om vilken typ av trafik bredbandsanvändare genererar är av intresse av flera anledningar. Inte minst av 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 leverantörer för den trafik som passerar genom deras nät, inte för vilken bandbredd de utnyttjar[19].

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 inte överträffa den normalt.

En mer kundinriktad motivering till att identifiera och påverka trafiken är QoS[[8]](#footnote-9). 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”.

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. 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.

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 peer:sen 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 å andra sidan väldigt intressant enligt ovan.

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 antagligen 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.

### 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 sådana tjänster primärt går över TCP där information som korrumperas eller försvinner på vägen skickas tillräckligt många gånger tills den anlänt till sin destination. Strömmande video å andra sidan är inte alls lika beroende av felsäkerhet. Att någon bildruta försvinner på vägen eller korrumperas påverkar inte filmen generellt förren 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 det 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 ens några sekunders fördröjning så länge som bandbredden kan hållas hög och stabil. Strömmande video och ljud påverkas inte heller sålänge 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 högre 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” som traditionellt använt sig av algoritmer som ”Leaky Bucket”.

Essentiellt sätt är det 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 högre fördröjningar och även fel i trafiken om buffern fylls. Det bästa vore att prioritera olika typer av trafik olika, vilket kräver att man kan identifiera trafiktypen.

### Några identifikationsmetoder

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

#### Portklassificering

De flesta typer av trafik, till exempel webb, ftp, IRC, eller e-post, skickas nästan uteslutande över de välkända portarna[23][[9]](#footnote-10). Det samma är sant för de flesta applikationer. Detta gör att man kan enkelt och utan någon egentlig beräkning direkt dela upp trafiken efter typ. Den främsta svagheten är att det finns inga tekniska hinder för att sända till exempel ftptrafik över port 80, och därmed felklassificeras den trafik som går över icke-standard portar. Fildelningsprotokoll så som Bittorrent och Gnutella har standardportar[[10]](#footnote-11) 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 P2Ptrafik i dagsläget så återstår det en stor mängd trafik av okänd typ[5]. I takt med att fler och fler klienter använder icke-standard portar så kommer den okända trafiken att öka.

#### Deep Packet Inspection

Med DPI tittar man på pakethuvudet och paketdatan. Från huvudet kan man utröna källa och destination bland annat. 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 klienterna implementerar dock RC4-kryptering av datan[37][38][39][40]. Vidare är etiken och i vissa fall lagligheten av inspektion av paketdatan också ifrågasättbar[30][32][33][34][42]. Dessutom tvingas man analysera mer information när man utöver huvudet också tittar på paketets data.

#### (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 endast utav detta. Jag vill trots det med denna uppsatsen visa att man kan med endast SPI med god sannolikhet ändå identifiera P2Pliknande trafik.

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

#### 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 als 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 sådan 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].

### 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]. En naturlig idé som uppenbarar sig är 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 nämnt, 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 fallet användes ingen regelrätt bandbreddsstrypning, men effekten var den samma eftersom överföringarna inte rent av blockerades.

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-P2Ptrafik gavs obegränsad hastighet (till den grad tekniken tillåter) medan P2P och liknande ”tung” trafik begränsades till den hastighet som man betalat för. Det finns många möjligheter, om man kan identifiera vad som är och inte är P2P.Identifiera P2P med hjälp av flöden

När jag under förundersökningar läste uppsatser och texter om dels dataströmmar och dels om P2Pidentifiering 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 den (eller de) datorn bakom adressen var involverad i 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 P2Ptrafiken 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.

### Utmaningarna

Varje paket måste klassificeras om det tillhör ett flöde som redan observerats eller om det är det första i ett helt nytt flöde. En gigabitlänk kommer potentiellt 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 uppgiften. 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 ett flöde två stycken 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 (eller negativa, beroende på hur man ser på det).

Ett CBF använder mer minne än ett vanligt bloomfilter eftersom det använder räknare istället för enskilda bitar. Dock kan vi hålla nere storleken genom att observera att långt ifrån alla flöden kommer 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 var 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 10gbit/s kan hantera 1000st 10mbit/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 131072st räknare har en felsäkerhet på mindre än 1% för 13663 IP-adresser, vilket innebär att medelantalet sedda IP-adresser inte ska överstiga 13,663 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 1byte 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 listan 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 inte finns kvar i listan och sakta faller mot noll. 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 listan 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[[11]](#footnote-12), 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 P2Pidentifierade ip-adresser är begränsade. Men vi har lagt en del tid på bloom filter 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 röd-svart-träd. Detta trots att minnesåtkomst tar längre tid än att beräkna ett värde i processorn.

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 bloom filters. Om algoritmen skulle implementeras i en router så skulle det nästan garanterat att krävas något bättre än en trädstruktur. Ett förslag på en bättre lösning ges i avsnitt 8.4.

### 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 = ,

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 = ,

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 ck 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/s 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 zk = 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.

### Algoritmen

Här följder 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 P2Padresser

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

P2Padresser.

För varje mätintervall y (större än x)

För varje adress i P2Plistan,

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.

### Möjliga förbättringar

De operationer som dominerar arbetet i algoritmen är hashfunktionerna i bloomfiltret och CBF:et. 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\*n* bytes, 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, och 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 simplicitet. 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 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 = ,

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 mindre mätperioder x och

*T*1 = en lägre tröskel (*T*1 < *T*) vid vilken elementet raderas ur listan. Detta 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.

### Jämförelse med en naiv implementation

För att verkligen uppskatta vad det är bloom filter 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 för 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 föräldernoden. Det beror på vilken struktur man använder men jag kan nämna att ett Röd-Svart träd skulle behöva tre pekare om 4 byte, dvs ytterliggare 12 byte per element.

Bloom filtret 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 bloom filter. Räknar vi dessutom in pekarna för ett Röd-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-bytes räknare använts. IP-adressen kräver 4 byte och summan blir alltså 5 byte per element. Med pekare blir det totalt 17 byte.

Counting bloom filter använder tio räknare per IP-adress vilket summeras till 10 byte per element. Inte en 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. Bloom filter 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.

### 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 inducerar en överföringsförbindelse 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 universitetnätverk 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 här använder man sig i hög utsträckning av portklassificering. Om någon IP-adresserna utnyttjar en port som finns i deras portlista så klassificeras flödet som P2P. Denna lista över P2P-portar genereras genom en ingående analys av paketen i dumpfiler över trafik. Potentiellt även paketdatan. Själva identifieringen sker i realtid och lyckas identifierea en stor del P2P på universitetnätet. Man lyckas inte identifiera flöden där båda parter använder tidigare osedda portar och sorterar bort trafik som passerar på välkända portar för andra tjänster.

*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 någonstans. 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 kommunciera 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 tiden 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 man helt ignorerar alla flöden som har en eller båda portar utanför intervallet 1024-30000 för att undvika falska positiva. Det framkom att den mesta P2P-trafiken sker i detta intervallet. 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 med en analys som baseras på de första 16 bytesen 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 kopplade 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.

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 är inget som hindrar P2P-klienter från att använda sig av dessa portar, som 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 klassficera som antingen P2P eller inte.

Detta är det enda arbetet 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.Implementering och tillvägagångssätt

### Implementering

Algoritmen implementerades i C++. Bloomfilter och counting bloomfilter samt det röd-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 dimensionerades för att klara av totalt hundra tusen flöden (128 KB) och tio tusen IP-adresser (256 KB). 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. De 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 kunnat ha gjorts i realtid givetvis.

Synkroniseringen mellan trådarna valde jag att hantera genom att bara 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. Noggrann eftertanke borde avslöja en lösning.

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.

### 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. Det hände att vissa IP-adresser förutom de lokala hamnade i listan men de sorterades ut 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.

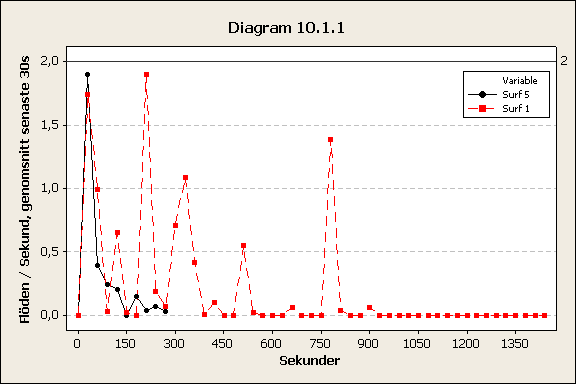
Webbsurfning och FTP-överföringar genomfördes med Firefox v2. Instant Messaging skedde med programmen Adium (Mac OS X) och Pidgin (Windows XP). Mejl hämtades via Thunderbird v2 eller via webbläsare. P2P, BitTorrentklienterna Transmission v0.96 (Mac OS X), μTorrent v1.7.5(Windows XP) och Blizzard Downloader(Mac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

## Resultat

### Webbtrafik och BitTorrent

Som vi kan se i diagram 10.1.1 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 är inte utförd av mig själv.

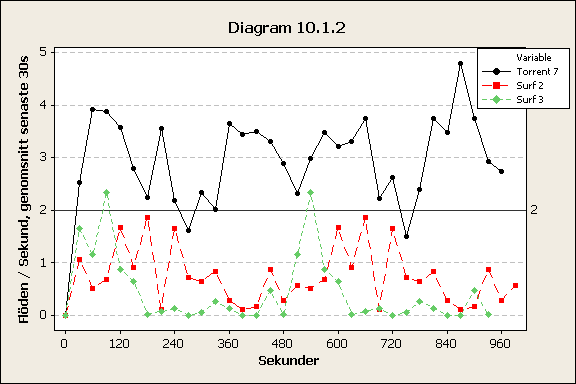


**Diagram 10.1.1: Webbsurfning utförd av två olika personer.**

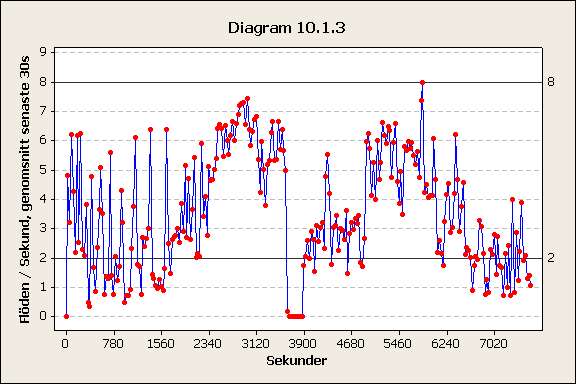
I diagram 10.1.2 ser vi den första BitTorrent-datan. Hastigheten var mycket låg, ca 10 KB/s ner och 60-70 KB/s upp. Svärmen bestod bara av 4-5 peers. I samma diagram finns också två kurvor över webbsurfning. I Surf 2 försökte jag klicka på länkar ofta och snabbt i ett försök att höja antalet registrerade flöden, vilket också lyckades. Surf 3 är främst en FTP-överföring.

Precis som i diagram 10.1.1 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 10.1.3. I mitten av överföringen kan man se att antalet flöden per sekund dyker mot noll. Detta beror på att jag råkade stänga datorn och den fick därmed söka upp och ta kontakt med peers igen efter att jag startat upp den. Svärmen bestod av ca 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.



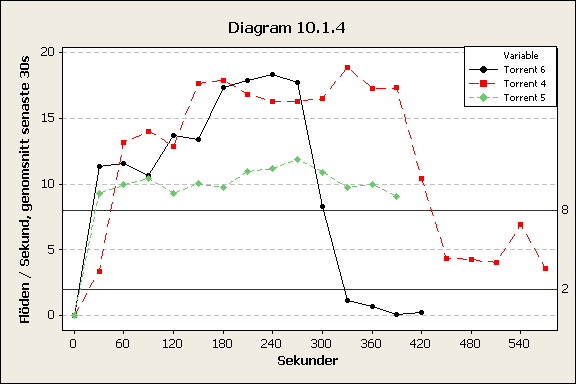
**Diagram 10.1.2: Webbsurfning i jämförelse med långsam BitTorrent.**



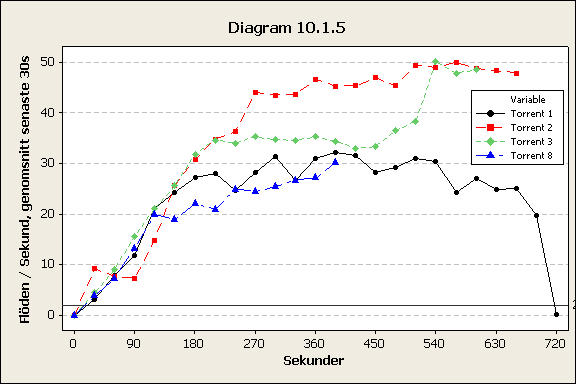
**Diagram 10.1.3: Långsam BitTorrent.**

När det gäller snabb BitTorrent är antalet flöden per sekund mycket högre. I diagram 10.1.4 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 ca 100 KB/s och svärmen bestod av 20 peers ungefär. Torrent 6 är den trafik som genereras av Blizzard Downloader, som används för att skicka ut uppdateringar för World of Warcraft.

Vad som är unikt för den jämfört med andra BitTorrentklienter är att den samtidigt laddar ner via http som P2P. 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.



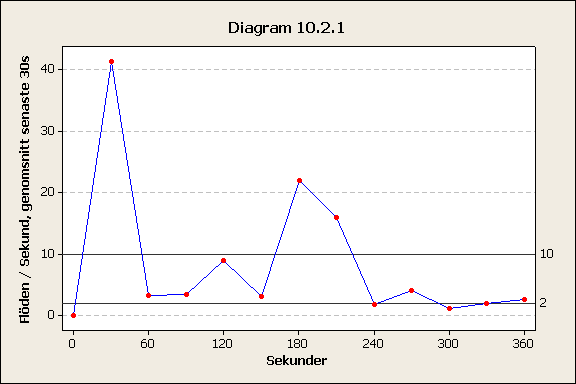
**Diagram 10.1.4: Snabbare BitTorrent, 100 KB/s - 1 MB/s.**



**Diagram 10.1.5: Väldigt snabb BitTorrent, runt 1 MB/s och stora svärmar.**

Vad som är gemensamt för serierna i diagram 10.1.5 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.

### Felkällor och metoder för att undvika upptäckt



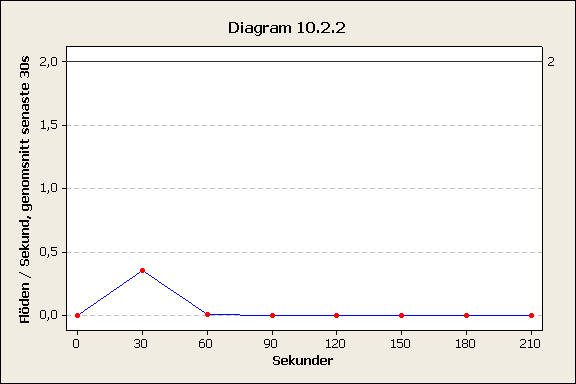
**Diagram 10.2.1: Samtidigt öppnande av 23 bokmärken i webbläsare.**

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 10.2.1 där jag 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 10.2.1 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 10.2.1 troligtvis uppstå. Populära servrar skulle också kunna få ett liknande mönster.

Det finns även metoder för att dölja all flödesrelaterad information. En sådan är att skicka trafiken först genom en annan dator med hjälp av en VPN-tunnel alternativt SSH. I diagram 10.2.2 ser vi att en ganska snabb BitTorrent-överföring blir i det närmaste osynlig när den skickas via en VPN-tunnel. För detta testet använde jag två datorer, den ena agerade router för den som använde VPN-tunneln. Själva trafiken registrerades sedan hos routern.



**Diagram 10.2.2: 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 jämförelse kan vi titta på Torrent 5 i diagram 10.1.4, som är exakt samma BitTorrent 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. Jag tittar endast på TCP-trafik, och eftersom VPN i detta fallet 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. Slutsatser

Enligt de situationer jag analyserat så uppträder BitTorrent med flera flöden per sekund stabilt övre längre perioder, medan icke P2P-trafik 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 routas 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 ”routas 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, ca 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 inget 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 vidarebefodra trafik genom min 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 identfieras. 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 av algoritmen. Eftersom jag har varit begränsad till 100 MB/s 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 40Gb/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.

1. 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.

1. Andrew S. Tanenbaum, *Computer Networks*, 4th Edition,

ISBN: 0-13-066102-3, Prentice Hall, 2002.

1. 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.

1. A. Gerber, J. Houle, H. Nguyen, M. Roughan, S. Sen, *P2P, The Gorilla In The Cable*,

Proc. National Cable & Telecommunications Association (NCTA), Juni 2003.

1. Myung-Sup Kim, Hun-Jeong Kang, James W. Hong, *Towards Peer-to-Peer Traffic Analysis Using Flows*,

DSOM 2003, sid. 55–67.

1. Jörn Altmann, Karyen Chu, *A Proposal for a Flexible Service Plan that is Attractive to Users and Internet Service Providers*,

IEEE INFOCOM 2001.

1. 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.

1. 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.

1. 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.

1. 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.

1. Michael Mitzenmacher, Compressed Bloom Filters,

IEEE/ACM Transactions on networking, Vol. 10, No. 5, Oktober 2002

1. Fan Deng, Davood Rafiei, *Approximately Detecting Duplicates for Streaming Data using Stable Bloom Filters*,

SIGMOD 2006, Juni 27–29.

1. Saar Cohen, Yossi Matias, *Spectral Bloom Filters*,

SIGMOD 2003, Juni 9–12.

1. J. Aguilar-Saborit, P. Trancoso, V. Muntes-Mulero, *Dynamic Count Filters*,

SIGMOD Record, Vol. 35, No. 1, Mars 2006.

1. Abhishek Kumar, Jun Xu, Li Li, Jua Wang, *Space-Code Bloom Filter for Efficient Traffic Flow Measurement*,

IMC’03, Oktober 27–29, 2003.

1. Rhea, S.C., Kubiatowicz, J, *Probabilistic location and routing*,

Proceedings of INFOCOM 2002.

1. Andrei Broder, Michael Mitzenmacher, *Network Applications of Bloom Filters: A Survey*,

Internet Mathematics Vol. 1, No. 4, 2005: sid. 485–509.

1. T. Karagiannis et al, *Transport Layer Identification of P2P Traffic*,

IMC'04, Oktober 25-27, 2004, Italien.

1. Alejandro López-Ortiz, *Algorithmic Foundations of the Internet*,

ACM SIGACT News, Vol. 36, No. 2, Juni 2005.

1. Remco van de Meent, Aiko Pras, *Asessing Unknown Network Traffic*,

CTIT Technical Report 04-11, University of Twente, Nederländerna, februari 2004.

1. A. Wagner et al, *Flow-Based Identification of P2P Heavy-Hitters*,

International Conference on Internet Surveillance and Protection (ICISP), 2006.

### Internet

1. *Internet Assigned Numbers Authority*, Senast besökt: 2007-12-13,

http://www.iana.org/

1. *TCPDump / Libpcap*, Senast besökt: 2007-12-13,

*http://www.tcpdump.org/*

1. *Relakks*, Senast besökt: 2007-12-13,

https://www.relakks.com/

1. *Blizzard Downloader*, Senast besökt: 2007-12-13,

http://www.blizzard.co.uk/wow/faq/bittorrent.shtml

1. *General Purpose Hash Function Algorithms*, Senast besökt: 2007-12-13,

http://www.partow.net/programming/hashfunctions/index.html

1. 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

1. Peter Svensson, *Comcast blocks some Internet traffic*, Senast besökt: 2007-12-13,

http://www.msnbc.msn.com/id/21376597/

1. 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

1. 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/

1. Chris Soghoian, *Congressman to Comcast: Stop interfering with BitTorrent*” Senast besökt: 2007-12-13.

http://www.news.com/8301-10784\_3-9804158-7.html

1. 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

1. 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

1. Stephen Withers, *Block P2P, Belgian court tells ISP*, Senast besökt: 2007-12-13,

http://www.itwire.com/content/view/13351/53/

1. 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

1. 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/

1. *μTorrent*, Senast besökt: 2007-12-13,

http://www.utorrent.com/faq.php

1. *Azureus*, Senast besökt: 2007-12-13,

http://azureus.sourceforge.net/faq.php

1. Adam Livingstone, *A bit of BitTorrent bother*, Senast besökt: 2007-12-13,

http://news.bbc.co.uk/2/hi/programmes/newsnight/4758636.stm

1. ”*CacheLogic and BitTorrent Introduce Cache Discovery Protocol*, Senast besökt: 2007-12-13,

http://torrentfreak.com/cachelogic-and-bittorrent-introduce-cache-discovery-protocol/

1. John Leyden, *Germany enacts 'anti-hacker' law*, Senast besökt: 2007-12-13,

http://www.theregister.co.uk/2007/08/13/german\_anti-hacker\_law/

1. Adam Pasick, *LIVEWIRE - File-sharing network thrives beneath the radar*, Senast besökt: 2007-12-13,

http://in.tech.yahoo.com/041103/137/2ho4i.html

1. ”*Användning av P2P-program på LUNET*, Senast besökt: 2007-12-13,

http://www2.ldc.lu.se/security/P2P-services.shtml

1. 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/

1. 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&

1. 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

1. Tim Weber, *Criminals 'may overwhelm the web*' , Senast besökt: 2007-12-13,

http://news.bbc.co.uk/2/hi/business/6298641.stm

1. Gregg Keizer, *Dutch Botnet Suspects Ran 1.5 Million Machines*, Senast besökt: 2007-12-13,

http://www.techweb.com/wire/security/172303160

1. Peter Bowes, *Warner to start movie downloads*, Senast besökt: 2007-12-13,

http://news.bbc.co.uk/1/hi/business/4753435.stm

1. Burt Helm, *BitTorrent Goes Hollywood*, Senast besökt: 2007-12-13,

http://www.businessweek.com/technology/content/may2006/tc20060508\_693082.htm

1. Andrew Orlowski, *RIAA sues the dead*, Senast besökt: 2007-12-13,

http://www.theregister.co.uk/2005/02/05/riaa\_sues\_the\_dead/

1. *Static random access memory*, Senast besökt: 2007-12-13,

http://en.wikipedia.org/wiki/Static\_random\_access\_memory

1. *Deep packet inspection*, Senast besökt: 2007-12-13,

http://en.wikipedia.org/wiki/Deep\_packet\_inspection

1. *Direct Connect*, Senast besökt: 2007-12-13,

http://en.wikipedia.org/wiki/Directconnect

1. *Gnutella*, Senast besökt: 2007-12-13,

http://en.wikipedia.org/wiki/Gnutella

1. *DC++*, Senast besökt: 2007-12-13,

http://dcplusplus.sourceforge.net/

1. *Botnet*, Senast besökt: 2007-12-13,

http://en.wikipedia.org/wiki/Botnet

1. Nelly Visanji, *En miljon svenskar använder Facebook*, Senast besökt: 2007-12-13,

http://www.idg.se/2.1085/1.129990

1. *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](http://www.sandvine.com/)

1. *BitTorrent Inc*, Senast besökt: 2007-12-13,

http://www.bittorrent.com/

1. For some problems a few times could be acceptable. [↑](#footnote-ref-2)
2. Ett viktigt krav i det fallet är också att man inte kan återskapa originalvärdet genom att ta inversen av hashfunktionen eller liknande. [↑](#footnote-ref-3)
3. 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. [↑](#footnote-ref-4)
4. förutsatt att det är ett C-nät som används, t.ex. 192.168.0. [↑](#footnote-ref-5)
5. Internet Relay Chat. [↑](#footnote-ref-6)
6. En användare bestämmer manuellt vilka filer eller mappar på datorn som ska delas ut till andra användare. [↑](#footnote-ref-7)
7. Distributed Denial of Service. [↑](#footnote-ref-8)
8. Quality of Service. [↑](#footnote-ref-9)
9. 80, 21, 194 och 110 respektive. [↑](#footnote-ref-10)
10. 6881-6889 och 6347 respektive. [↑](#footnote-ref-11)
11. Native adress translation. [↑](#footnote-ref-12)