**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

**Finding Whispering Elephants Among Screaming Mice**

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

***Abstract***

With the ever increasing number of broadband connections and demand for higher bandwidth Internet Service Providers are facing massive investments or the Internet might soon run out of capacity. At the same time users have switched from mainly downloading web or mail content to both download and upload media in the form of video and audio. In the light of this several providers are turning to traffic shaping to improve their Quality of Service. In this paper I analyse the difficulties of identifying Peer-to-Peer 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**

About the translation 4

1. Introduction 5

1.1. Statement of purpose 6

2. Part 1: Pink and white elephants 7

2.1. Massive amounts of data and fast data streams 7

2.1.1. Different types of windows 8

2.2. Computer networks and TCP/IP 9

2.2.1. The transport level 11

2.3. Data streams in relation to routers 12

2.4. Randomization: when it pays to forget. 14

2.4.1. Probability theory 15

2.4.2. Bloom filter 18

2.4.3. The importance of choosing good hash functions 19

2.4.4. Counting Bloom filter 20

2.5. Peer-to-Peer 24

2.5.1. Difference from Client-Server 24

2.5.2. Napster 26

2.5.3. Gnutella 26

2.5.4. Direct Connect 27

2.5.5. Bittorrent 28

2.5.6. Botnets 29

2.5.7. File sharing is illegal, right? 30

2.6. Traffic shaping 31

2.6.1. Quality of Service 32

2.6.2. Some identification methods 34

2.6.3. Handling large amounts of traffic 35

3. Part 2: Separating the elephants from the mice 37

3.1. Identifying Peer-to-Peer by counting flows 37

3.1.1. Challenges 37

3.1.2. The average value list 40

3.1.3. The algorithm 41

3.1.4. Possible improvements 42

3.1.5. Comparison with a naïve implementation 43

3.1.6. Related work 44

3.2. Implementation and approach 46

3.2.1. Implementation 46

3.2.2. Measurement data 47

3.3. Results 48

3.3.1. Web traffic and Bittorrent 48

3.3.2. Error sources and methods to avoid detection 52

3.4. Conclusions 54

Bibliography 56

# About the translation

This is a translation from the original Swedish text, translated by myself. The translation follows the original except for a few places where I felt that I had to switch parts of the text around to better suit the flow in English. Naturally, as with any translation, the quality of the text will invariably be worse than if it were written in English in the first place. I hope the reader can overlook this.

# 1. Introduction

The Internet has in its short life quickly evolved to an essential means for exchange of information. The speeds 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 they now 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, ISPs 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 providers' routers operate in is a challenge.

Advances in several areas, especially communication and databases, has created the need for a fresh view on 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 slowly growing function. In the context of data streams however, it is not unusual for log(n) to quickly grow beyond 20. A potential twentyfold decrease in performance is not something that can be ignored [[1](#Blo70)].

With a viewpoint based on 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 explain TCP/IP traffic. The randomised data structures, Bloom filters, which are used to process the stream are introduced in chapter 2.4. Chapter 2.5 and 2.6 deal with P2P and traffic prioritization. The algorithm I have used is explained in chapter 3.1, together with some suggestions for improvements. Finally, the experimental procedure and results are presented in chapters 3.2, 3.3 and 3.4.

## 1.1. 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.

# 2. Part 1: Pink and white elephants

Part 1 goes over the motivations behind the purpose of the paper and the foundations to understand part 2. It explains and defines concepts such as data streams, Peer-to-Peer and Bloom filter.

## 2.1. 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 coming in very fast and has to be processed and re-transmitted quickly 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-1). 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 of them (like the first piece of text we 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.

### 2.1.1. 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 be appropriate to only 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 we would potentially detect it in none of the windows.

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 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.

## 2.2. Computer networks and TCP/IP

A network can be constructed in any number ways. The most common approach is to use *twisted pair* cabling to connect computers, switches and routers which in turn are connected to other computers. This is far from the only way to construct networks. Multiple types of cabling exists and every one of those types could require a more or less unique implementation. Another commonly used method to build networks is by using 802.11/a/b/g/n wireless access points and network cards. Other methods include telephone wires (DSL, ISDN), Bluetooth, satellite based radio signals and much more. It would even be possible, given enough time and effort, to construct a network using only paper cups and a piece of string. The point I wish to make is that nearly all of these different methods require different implementations in the computer's software to function as a computer network. Sending a radio signal is quite different from sending electrical impulses through cables (or vibrations through strings). Developers the world over are forever grateful that they usually don't have to consider on what medium the network is constructed. This since the process of establishing a connection is handled at a lower level than the one most write their programs on. Normally, it is the operating system that handles this.

A network connection is made up of seven levels according to the ISO OSI reference model (*Open Systems Interconnection)*. These are as follows, from the bottom up: the physical level, the data link level, the network level, the transport level, the session level, the presentation level and the application level. I will only concern myself with the transport level in this paper. Of the lower levels I will only mention that they provide the ability for the transport level to transmit data over a connection (down to the actual physical conversion to electrical/optical/vibrating signals depending on the medium). The upper levels handles the actual data that is transferred. This differs from application to application. Since I assume this information to be encrypted and unreadable, I ignore it. Would the reader be interested in reading more about this, then Tanenbaum's Computer Networks [[4](#Tan02)] would be an excellent starting point.

At the transport level on the other hand, exists those protocols that convert the data that is to be transmitted to (for a computer) understandable bit streams. There are several commonly used protocols. Transport Control Protocol (TCP) is one. User Datagram Protocol (UDP), Real-time Transfer Protocol (RTP) and Real-time Control Transfer Protocol (RTCP) are others. 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. But data that becomes lost can reduce 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 is generally 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.

The reason "guaranteed delivery" isn't something one can expect from all protocols is because when a computer sends information to another computer, that information will pass through an amount of different computers and devices before reaching its destination. One of those devices might disappear from the network unexpectedly in the middle of the transfer. Information passing through third-party machines is a natural consequence of the fact that computers usually don't have a dedicated network cable to all other computers. Up til the end of the 80s, every computer had basically a complete map of the entire network and thus knew exactly which path it should send the data on to make it arrive quickly and safely. But as networks grew and traffic increased, this became unmanageble [[5](#Lóp05)]. To resolve the problem, new protocols were developed which meant that computers instead only had a local map of nearby neighbours. This map is always small[[2]](#footnote-2) and changes can be made quickly and simply, without affecting the network as a whole. Routers perform updates in their map (or rather list) depending on the time delay to its neighbours in order to be able to select the best possible route to the destination for the packages that pass through. This has the effect of making packages part of the same transfer arrive in a different order than the one they were sent in because they might very well take different paths across the network. It is up to the destination computer to piece the packages back together again.

### 2.2.1. The transport level

When two computers, or rather two applications, wish to communicate with eachother over TCP, they must establish a connection with eachother. Two different applications situated at the same computer differentiate their communications by using different ports. A port is defined by a, for the computer, unique 16 bit number. A connection is thus established between two ports, that could be localized as different machines or not. Data can be sent in both directions across the connection. But each port can be associated with several connections. The destinations of single packets are defined by their port pairs (source and destination). So how does an application know which port it should use if it wishes to contact another machine? If the computer isn't listening on the port in question, it will discard the data sent to it. To facilitate the communication between foreign computers, popular services such as email etc have been assigned (or have annexed) a specific port number. So one can be sure that a public service always on the same port number, unless something else is specified. A web server almost always listens on port 80. When a client wishes to connect to the server, it first opens an arbitrary port[[3]](#footnote-3) and sends a request to port 80 at the server to establish a connection. Once a connection is established, the protocol takes the data that is to be sent and divides it into smaller pieces (referred to as *packages*), often in sizes of 1500 bytes since this almost always is the value of the Maximum Transmission Unit (MTU) of the data link level (Ethernet that is). It could be as large as 64 kB according to the TCP specification. 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 | | | |

**Table 2.2.1.1.: The structure of a TCP packet, the head is made up of bit 0 to 160/192.**

The protocol implementation of the destination is responsible for piecing together the packages to the original data again. Every package recieves a sequence number and TCP makes sure no packages are lost in the transfer by keeping track of which sequence numbers have arrived and sends a confirmation for those numbers it has received. If a confirmation for a package isn't received within a reasonable time limit it is re-transmitted. That way we can be certain that our data reaches its destination, and that any possible disorder doesn't affect the transfer significantly.

## 2.3. Data streams in relation to routers

We face two primary problems, firstly the problem of computation time. 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)] [[5](#Rob07)] [[6](#Gro07)], 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 according to 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.

The second primary problem is memory use. At the speeds described above, the type of memory commonly used as primary memory in most personal computers, DRAM[[4]](#footnote-4), 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[[5]](#footnote-5), 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.

## 2.4. Randomization: when it pays 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 maximum of (232⋅216)2 > 7.9⋅1028 possible flows with IPv4[[6]](#footnote-6). 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.

### 2.4.1. Probability theory

It might be appropriate with a small summary of the mathematics behind the analysis of Bloom filters in coming chapters and randomized algorithms in general. It is no regards complete but is only included to give the reader a simpler way to follow the calculations that are done later.

**Definition 2.4.1.1.:** *A probability space is composed of the following:*

***1.*** *A result space Ω denoting all possible outcomes of the random (stochastic) process that the probability space describes.*

***2.*** *A collection of sets F, where each and every set in F is called an event and is a subset of the result space.*

***3.*** *A probability function Pr : F→ℝ as defined by 2.4.1.2.*

A random process is described by a probability space and all calculations and statements refer to the probability space. A probability function is defined like so:

**Definition 2.4.1.2.:** *For a probability function Pr : F→ℝ the following is true:*

***1.*** *For an event E, 0≦Pr(E)≦1*

***2.*** *Pr(Ω) = 1*

***3.*** *For a finite or countable infinite set of pairwise disjoint events E1, E2, E3,..., :*

Note that the third part of the definition above demands pairwise disjoint events. If we are less restrictive and considers all finite or countably infinite sets of events, it follows that:

**Lemma 2.4.1.1.:** *For a finite or countably infinite set of events E1, E2, E3,..., :*

A short example to clearify the difference between the third part of definition 2.4.1.2. and lemma 2.4.1.1. would be to flip two coins, each seperately. E1 is the event that the first coin is heads and E2 is the event that the second coin is heads. The sample space consists of four possible outcomes, all with an equal probability of .

1. Neither is heads.
2. Only the first coin is heads.
3. Only the second coin is heads.
4. Both coins are heads.

Here we see that E1 is equal to the event that either outcome 2 or 4, and E2 is equal to the event of either 3 or 4. An obvious fact appears:

**Lemma 2.4.1.2.:** *For two events E1 and E2 :*

For the example it is clear that it must be so, since both E1 and E2 include outcome 4 and we can't count the same outcome twice. So then it's clear that

(2.4.1.1.)

and why:

(2.4.1.2.)

The events E1 and E2 are not disjoint according to the definition but satisfies lemma 2.4.1.1. However, we can easily modify the example to show when the definition is satisfied. Let E1 and E2 still represent that each respective coin is heads. But instead of flipping the coins seperately, we glue the coins together in such a way so that if one coin is heads up then the other coin is heads down. Flipping this double coin now can only give rise to two possible outcomes, both with probability .

1. The first coin is heads, the second one is tails.
2. The first coin is tails, the second one is heads.

Pr(E1) and Pr(E2) are still equal to since each coin is still regarded seperately but:

(2.4.1.3.)

Gluing the coins together have in other words made the events E1 and E2 disjoint.

**Definition 2.4.1.3.:** *Two events E and F are independent if and only if*

*and more generally: events E1, E2, ..., Ek are independent if and only if, for all sets :*

If two events E and F are independent, then the probability of E does not depend on the occurance of F. Consider the example of the coins earlier. In the first case where each coin was flipped seperately, the probabity for *both* coins being tails was . Which is exactly was definition 2.4.1.3. demands:

(2.4.1.4.)

Consider now the second case, where the two coins were glued together. Pr(E1) and Pr(E2) are the same as in the first case, but the probability of both being heads is zero since the glue forces one coin always to be tails:

(2.4.1.5.)

The definition corresponds well with the intuitive meaning of independence: that an independent event is not affected by other events.

If you have two dice, you are often interested in the sum instead of the individual die values. Throwing a die has six possible outcomes. This means that throwing two dice generate 36 possible outcomes, all with probability . The sum of the two dice has eleven possible values: 7-12. The eleven events do not have equal probabilities. For example, the probability of throwing 12 (6+6) is less than the probability to throw 10 (6+4, 5+5 or 4+6). To describe this, a stochastic variable is defined:

**Definition 2.4.1.4.:** *A stochastic variable X is a function which maps values in the sample space Ω to the real plane; . A discrete stochastic variable X is a stochastic variable with a finite or countably infinite number of values.*

In the case of the sum of the two dice, we can define a stochastic variable which takes the possible values of the sum. The event "X = a" represents the set of outcomes where the sum equals *a*. That is the set . The probability for that event is:

(2.4.1.6.)

The event "X = 7" is the set of outcomes where the sum equals seven:

{(1, 6), (2, 5), (3, 4), (4, 3), (5, 2), (6, 1)}. Six possible outcomes that gives:

(2.4.1.7.)

Stochastic variables can also be independent, just as previously defined.

**Definition 2.4.1.5.:** *Two stochastic variables X and Y are independent if and only if for all values of x and y:*

The point of all these definitions is to now be able to talk about probability distributions. Assume that we perform *n* independent experiments, all successful with probability *p* (and thus fail with probability (*1-p*)). Let the stochastic variable X represent the number of successful experiments. X then assumes a binomial distribution.

**Definition 2.4.1.6.:** *A binomial stochastic variable X with parameters n and p, is defined by the following probability distribution with j = 0, 1, 2, ..., n;*

In other words, X equals *j* when there are exactly *j* successful and (*n-j*) failed experiments where each experiment is independent and successful with probability *p*. There are ways to select which experiments should succeed and which should fail. The distribution appears later in the analysis of counter Bloom filters where the experiments are represented by a number of independent hash functions.

### 2.4.2. Bloom filter

A Bloom filter [[1](#Blo70)], 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 [[9](#Bro05)], even in computer communications. It has also been further developed to overcome some of its fundamental weaknesses and to accomplish different tasks. A Bloom filter supports only insertion but not deletion for example. These variants have been given such spectacular names as *Counting Bloom Filter* [[10](#Fan98)], *Stable Bloom Filter* [[12](#Den06)], *Spectral Bloom Filter* [[13](#Coh03)], *Dynamic Count Filter* [[14](#Agu06)], *Space Code Bloom Filter* [[15](#Kum03)] and *Attenuated Bloom Filter* [[16](#Rhe02)].

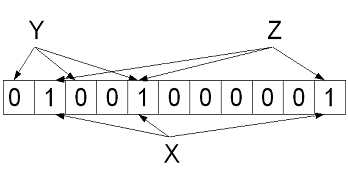


Figure 2.4.2.1: A Bloom filter with three hash functions where the arrows indicate the bits determined by the hash functions. First, Insert(X) is performed. Query(Y) then returns "false" correctly while Query(Z) erroneously returns "true", despite Insert(Z) not having been performed. In other words, Z is a false positive.

**Definition 2.4.2.1.:** *A Bloom filter, BF, is a randomised data structure consisting of:*

*A bit vector of size m. By default, all bits are equal to 0.*

*k hash functions, h1 - hk, that are defined as: , where X is the space that elements are defined on.*

*A Bloom filter has two operations:*

*Insert(x): The bits determined by h1(x),..., hk(x) are set to 1.*

*Query(x): The bits determined by h1(x),..., hk(x) are checked. If atleast one of the bits is 0, false is returned. If all bits are 1, true is returned.*

When we want to know if *x* has been inserted into the filter and perform *Query(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 filter. 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 [[11](#Mit02)] that the probability of this is minimized when the number of hash functions is , to , 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 a 5% risk of false positives.

The advantage of Bloom filters is that only O(k) operations are required for insertion and search, and the amount of memory which must be allocated does not depend on the size of the input elements but only on their number, 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.

### 2.4.3. The importance of choosing good hash functions

A *perfect* hash function has the same value distribution as a die, e.g. completely random. Because of the obvious difficulty with generating truly random values from a systematic and deterministic process, this is rarely possible. A *good* hash function will have an *almost* completely random value distribution. Desired properties are for instance that a small change in the input data should give rise to a large change in the output data and that the hash function is relatively fast, which is especially important if it is to be used with data streams.

To give an example, consider a hash function based on the modulo operator. We want to hash a few integers z to values from 0 to m. We define the function h(x) = z mod m. This is not a good hash function for most cases. A small change in the input data does not generate a large change in the hash value. Its speed is dependent on how we define a function to be fast and how the modulo operator is implemented in the software.

It's not difficult to imagine some input data which will have a far from random distribution. Choose every *m*th integer for example, and all elements will be hashed to the same value and *collide*. This is a fundamental problem with hash functions which is unavoidable. Even for a good hash function, and by good I mean the properties mentioned before, it is theoretically possible to construct a set of input elements that will be hashed badly. The primary difference between a good and a bad hash function, is that the likelihood of such an input set actually occurring under real circumstances is low. These problems are avoidable as long as the input data of the hash function is known and limited. The hash functions used for this paper was given IP addresses and port numbers as input data. The total number of combinations is huge but consider for a moment that we were only interested in a specific subnet (an ISP controls an amount of IP addresses only they have access to), for example our local home network. There are only 255 possible IP addresses available to the computers in this network[[7]](#footnote-7) and constructing a good hash function for that input data is trivial. We can use the last number of the IP address directly as an index for a table since it is unique for each computer and the total number of computers (255) is a very small table indeed. Now we have a fast and non-colliding hash function for these specific networks. But if we were to apply that function to a different, larger network, we would instantly run into trouble since there would be too many collisions.

Hence a hash function might be quite bound to its application. A hash function that works well for Bloom filters does not necessarily work well for encryption[[8]](#footnote-8) or data integrity[[9]](#footnote-9).

### 2.4.4. Counting Bloom filter

**Definition 2.4.4.1:** *A counting Bloom filter, CBF, consists of:*

*A vector of m counters. By default, all counters are set to 0.*

*k hash functions, h1 - hk, defined as: , where X is the space that elements are defined on.*

*A counting Bloom filter has three operations:*

*Insert(x): The counters determined by h1(x),..., hk(x) are incremented by one.*

*Query(x): The bits determined by h1(x),..., hk(x) are checked. The value of the smallest counter is returned.*

*Delete(x): The counters determined by h1(x),..., hk(x) are decremented by one.*

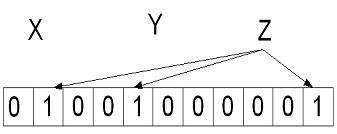


Figure 2.4.4.1.: A counting Bloom filter with three hash functions where Insert(Z) has been performed once.

A CBF [[10](#Fan98)] differs from a normal Bloom filter by using an array of counters instead of an array of bits. After a number of *Insert* operations, you might be interested in how many times y has been added. *Query(y)* then returns the smallest value among the counters decided by h1(y),...,hk(y). Because it is possible, even probable, that another value z has collided with some of y's counters, the counters will potentially have different values. In which case the counter with the smallest value is the one that has collided the least and therefore is the *most* correct.

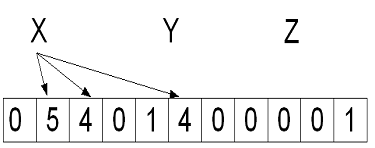


Figure 2.4.4.2.: Insert(X) is performed four times. Note that X collides with Z for one counter.

A CBF also supports deletion, which a normal Bloom filter does not. If *Insert(y)* is performed *m* times, then the counters will have values greater than or equal to *m* (depending on collisions). We say that there are *m* instances of *y* in the CBF. It is now possible to remove one instance of y by decrementing said counters by one with *Delete(y)*. If *Insert(y)* hasn't been performed and there are collisions for all of *y*'s counters, *Delete(y)* will cause the counters of other elements to have too small values. In the worst case, false negatives are created by wrongfully setting counters to zero. Since the deletion in counter Bloom filters is not used in the algorithm demonstrated later, I will not concern myself with the probability of a counter having a value that is too low after *Delete()* operations.

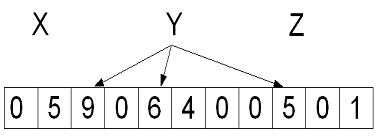


Figure 2.4.4.3.: Insert(Y) is performed five times. Despite Y colliding with both X and Z, there are still correct counters.

If an element occurs many times, its counters will have high values. There is then a chance that the counters will reach their maximum and roll over to zero, which would make deletion for those elements colliding with that that element impossible. Fan et al. [[10](#Fan98)] say that 4 bit counters (maximum 15) would be sufficient for most applications. That is not the case here where values over 50 were nothing unusual. In my implementation I have used 16 bit counters (maximum 65 535), but 8 bits (maximum 255) would have been perfectly sufficient.

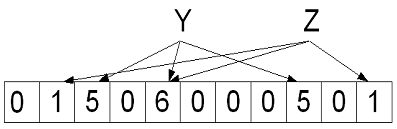


Figure 2.4.4.4.: *Delete(X)* has been performed and its counters have been decremented. Y and Z have correct values in more counters than before.

Just as for Bloom filters there is a risk for false positives in a CBF. A value z might be reported as being contained in the CBF when that is not the case. The probability for that is the same as for a regular Bloom filter though, as described earlier. Another error introduced by the CBF is the risk that the value returned by checking the number of instances of y is wrong. Since multiple elements might have collided in their hash values there is a risk that all counters, and thereby also the one with the smallest value, have collided. This is however directly related to the probability of false positives: the probability that other elements are hashed to the same counters as y.

It is then interesting to know how large the probability is that the value returned differs from the actual value by more than *j*. This is very dependent on the number of instances of each element that will be added to the CBF. If we define *pj* as the probability that a collision makes the counter more than *j* wrong, we can make the following observations:

The probability for counter *r* to be incremented has a binomial distribution. That is:

(2.4.4.1.)

where *p* equals the probability of any of the *k* hash functions to hash to the counter. Which means that the probability of *r* not being incremented by any of the *n* elements is:

(2.4.4.2.)

The probability of being incremented by exactly *one* other element (a single collision) is:

(2.4.4.3.)

Hence the probability that *more than one* collision occurs for the counter *r* is:

(2.4.4.4.)

The probability that the counter *r* is *more than j* wrong, pr, is less than:

(2.4.4.5.)

Here I make the simplification that more than one collision automatically makes the counter more than *j* wrong which doesn't have to be the case of course, meaning that I am overestimating the error slightly. Finally then, the total probability of an element *y* to be reported to have more than *j* instances too many in the CBF is equal to the probability that *all* the counters decided by h1(y),...,hk(y) is more than *j* wrong, which equals:

(2.4.4.6.)

As a special case we can set *pj* to 1, every error will exceed *j*, alternatively *j* = 0. In other words, what is the probability that *Query(y)* returns an incorrect value. What we get is:

(2.4.4.7.)

Which is almost equal to (2.4.4.8.)

Assuming (2.4.4.9.)

That is, the chance for false positives in a Bloom filter. This since

(2.4.4.10.)

by the definition of the exponential function:

(2.4.4.11.)

where, if we define , get:

(2.4.4.12.)

Already when *n* = 10, this becomes:

(2.4.4.13)

So the function converges very quickly. The small difference that arises between the different functions is explained by Mitzenmacher's use of an exponential function to estimate the probabilty, and that it is not an exact representation. What is clear however is that the chance of an element being reported as having a wrong instance count is comparable to the chance of false positives in a traditional Bloom filter.

## 2.5. Peer-to-Peer

Before 1999, file sharing was at an almost personal level. If you didn't know someone that could burn it on CDROM, put it on a floppy or mail it to you, it was both cumbersome and time consuming to find what you were looking for. In the case of music and mp3, you had to turn to web pages that individually didn't have a particularly large selection. Available storage space on the web and in general were significantly more limited and expensive than today. Then Shawn Fanning released Napster.

Napster was not a true Peer-to-Peer program (P2P) in the strictest sense but historical reasons demand it to be mentioned. Except Bittorrent (which is the most popular today) and Napster, I have chosen to also mention Gnuttella and Direct Connect since they both filled a gap between the fall of Napster and the rise of Bittorrent and to illustrate the difference between Bittorrent and earlier P2P protocols. It is also worth mentioning this difference between today's protocol (Bittorrent) and yesterday's (Napster) since in a few years new protocols will have been developed and it is hard to predict how these will function, even if Bittorrent today seems to be an indication of it. More programs exist (and have existed) like Fast track (Kazaa), WinMX, Gnutella2, eDonkey etc but they will not be dealt with in any detailed manner. It is important to realise that P2P not only is used for file sharing, even if it is the most common use for it. It is used where a de-centralized structure is either preferable or the only possibility.

### 2.5.1. Difference from Client-Server

The most common method of connecting computers with each other has traditionally been the client-server way. One computer acts as a server and one or more clients connect to it. The clients have no knowledge of each other and cannot directly communicate with each other. All information between clients must first pass through the server. This is natural where clients need access to the same data or when clients have no need to communicate with each other such as during web browsing. Primarily two aspects are especially clear.



**Figure 2.5.1.1: Client-Server.**

Firstly, the server needs a high bandwidth to the clients since all clients communicate with the same server. If the services become very popular such as Google's search engine, companies are forced to use several machines and multiple connections to the Internet to be able to offer a fast service even during high load. Secondly, if the server crashes or for some reason goes offline the services it offers to its clients disappears. So the server is vulnerable to attacks and errors.



**Figure 2.5.1.2.: Peer-to-Peer.**

Peer-to-Peer on the other hand doesn't rely on any single server. Here all the clients simultaneously act as servers. Clients are instead called *peers*. All data, once peers have found each other, pass directly between the peers without first passing through a third party. This means that in contrast to the client-server way, a P2P network is not affected by one peer going offline, unless that peer doesn't possess some unique piece of data which it hasn't shared with other peers yet. So it is resistant against attacks and errors. At the same time it also has the advantage that no peer must have a higher bandwidth than other peers for the communication to be fast. Either the peer can send data in its completeness to other peers one by one, or it can send to all of them at once. The time it takes for everyone to have all the data is the same, but in the first way the information exists in more copies and is therefore more secure and faster for the individual peer. Every peer that has a complete copy can in turn share with others.

### 2.5.2. Napster

In 1999 the first really popular file sharing program arrived. It supported only transfers and searches of mp3-files. A user connected to the server which provided information about other connected users, and which files they had available. If the file you searched for was found it was downloaded directly from one of the users sharing it.

Napster's strength was that everyone used it. At the time there were no competing applications and thus there was a large amount of users connected. By today's standard it was fairly slow, but everyone's connection was slower in those days also so it wasn't noticeable.

In the end of the same year, Napster was sued by the record companies and in 2001 the whole network was closed down (shortly thereafter Napster was resurrected as a commercial service). Its weakness was the centralized structure (strictly disqualifying it from being true P2P) with one server that all clients connected to.

### 2.5.3. Gnutella

Originally developed at Nullsoft (but promptly abandoned after AOL, which bought Nullsoft in the same year, put their foot down [[12](#Gnu07)]) and released in 2000. It is a de-centralized system without a central server and supports all types of files. It thereby qualifies as P2P in the strictest sense of the word.

A peer that connects to the Gnutella network must first find another peer that is connected. This can be done with a list of potentially functioning nodes from a web page or some other source (IRC[[10]](#footnote-10) has also been used, for example). Once it is connected it establishes its own list of nodes which are used the next time the peer tries to connect. While connected, you can search, download and upload files. Just as for Napster, the speed of the transfers depend entirely on the bandwidth of the individual peers since you transfer a file only between two peers at a time. Here the individual peers also handle the search in the network.

### 2.5.4. Direct Connect

DC was released around 1999 [[13](#Dir07)]. There are several third party client programs for the protocol. Just as is the case for Napster there is a central server, in this case called a hub, that the clients connect to. Unlike Napster though there is not only one server, anyone can start a DC hub. The hub provides search possibilities of the files being shared by connected users and a chat service.

A client can be in either *active mode* or *passive mode*. A client in active mode can both search and download from all of the other clients, while passive clients are limited to only download and search from active users. Active users listen on a port and can directly receive requests about certain files. Passive clients on the other hand must get such a request from the server. An active client asks the server to instruct the passive client to open a connection to the port it, the active client, is listening to. A passive client cannot do this since they do not listen on any port.

In practice, usually the clients behind firewalls, unless they have manually opened a port, become passive clients. They thereby don't have access to as much data as active users do. The actual transfers, once a connection has been established with or without the server's assistance, takes place directly between clients. The transfers and connection with the server use TCP and the searches use UDP.

A client specifies exactly how many concurrent connections should be allowed, so called slots (different for upload and download). Hubs are usually specialized on a certain type of data, for example movies, anime or games. It is also common that they have demands on how much data must be shared[[11]](#footnote-11), what the data should be composed of, how many slots should be open and also what bandwidth and ISP users are required to have in order to be allowed to connect. All this is up to the administrator of the server.

The weakness is the same as for Napster, e.g. a central server that everyone depends on. Granted, there are several servers, but this also means that all of the data is spread out between several servers which means it can be hard to find a particular file. Additionally, it is in reality not possible for just anyone to start a hub because quite a large bandwidth is needed (especially for uploads, but also for downloads) since all search queries and passive downloads must pass through the server. This means that there is an upper limit on the number of users for every server, depending on its bandwidth and other resources such as CPU etc.

### 2.5.5. Bittorrent

This is the most popular P2P protocol today and is estimated to be responsible for 35% [[14](#Pas07)] of all the Internet's traffic at current. The protocol has become so successful that it is the first P2P protocol to be embraced by commercial entities to distribute files [[15](#Bli07)] [[16](#Bow07)] [[17](#Hel07)]. It was created by Bram Cohen in 2001 and is developed today by his company Bittorrent Inc [[18](#Bit07)].

First you download a so called torrent for the file you are interested in. This is usually done from web pages dedicated to distributing torrent files which offers users to search among them. A torrent is a meta file that contains information about the file or files you're really interested in. Among others it contains addresses to one or several so called trackers, information about the number of pieces and hash values for those pieces. Then you open the torrent file in your Bittorrent client program which in turn then requests a list of peers from the trackers specified in the torrent file.

What makes Bittorrent unique is that the file or files that are shared by the torrent, are split into smaller pieces. Once such a piece has been downloaded the peer can then start to offer the piece to other peers. So peers must not wait for the entire file to finish before sharing with others. With an algorithm named "Rare-First" the most rare pieces are downloaded first. It subsequently takes a very short time for a file to have more than one copy in the network, even though no individual peer might actually have the complete file. Everyone can have different pieces.

Since clients can share individual pieces of files, the speed of a Bittorrent transfer is generally higher than for the other P2P protocols mentioned. It also means that the individual bandwidth for every peer plays a much smaller role since peers with higher bandwidth simply can connect to more peers. The two most significant advantages of the protocol are:

1. The possibilities for scaling. The capacity of the network is raised for every peer that is added, regardless of its bandwidth. No central server with large resources is required as for Napster or DC. This is interesting for companies since they then don't have to pay an ISP for a lot of bandwidth but can still offer fast transfers of the files they want to distribute.
2. Fault tolerance. In the case for Napster, DC and any client-server system, all the traffic is highly dependent on the central server. Would that server for any reason go offline the entire network will come to a halt. For Bittorrent it doesn't matter if an individual peer disappears, as long as it doesn't have a unique piece of the file not found elsewhere. Stability and accessibility are naturally also interesting for companies since they then don't have to pay a technician to be on call to go to the office and fix the server.

The only obvious weakness Bittorrent has is where you get a hold of the torrent file to connect with other peers in the first place. If the torrent tracker goes offline, it becomes difficult for new peers to connect to the already connected peers. It is solved in some part by torrent files specifying more than one tracker. Once a connection is made to another peer it doesn't matter if the tracker goes down since a peer can learn about new peers through the peers it is already connected to.

Most Bittorrent client programs now also implement RC4-encryption [[19](#Wil07)] [[20](#μTo07)] [[21](#Azu07)] [[22](#Liv07)] of the traffic and use ports that differ from the Bittorrent standard (6881-6889) after a few ISPs having a negative policy towards Bittorrent [[23](#Sti07)] [[24](#Sve07)] [[25](#Sto07)]. None of these methods offer anonymity for the users and are not intended to. The goal is to bypass the limitations imposed by certain ISPs in their networks (see 2.6).

### 2.5.6. Botnets

Other applications than file sharing exists for P2P. Instant Messaging is one simple example. Another far more interesting (for tracking purposes) example is botnets. Bot is short for robot and refers here to a program that runs on a so called *zombie computer*; a computer running a form of remote control software, usually without the owner knowing about it. The zombie could be any computer in the world and the bot program is used by the person in control of the botnet for various purposes. Usual tasks for the botnets are to deliver spam or take part in a DDOS[[12]](#footnote-12) attack, an attack where thousands of computers simultaneously connects to for example a web page just like regular web users except much more frequent with the purpose of overloading the server. The different uses of botnets, among others to extort companies by threatening to engage a DDOS attack on their network or to sell spam possibilities to companies for advertising, present ways for the botnet "owner" to earn money.

Botnets are a big problem on the Internet and it is estimated that up to 150 million computers [[26](#Web07)] could be infected by bot programs. They usually spread through computer viruses, worms or trojans. Individual botnets can consist of over a million zombies [[27](#Kei07)]. IRC has been a common way of controlling the networks. Newer versions however use a P2P protocol [[28](#Pen07)] [[29](#Bot07)], which for security reasons motivate the identification of such P2P traffic.

### 2.5.7. File sharing is illegal, right?

Electronic distribution of information can never be illegal. If it were, it would seriously affect the right to privacy. It is by Swedish law forbidden to distribute copyrighted material without the approval of the copyright holder. Copyrighted material make up a large part of the P2P traffic on the Internet, that much is clear. But ISPs have as much responsibility for what their customers send over the Internet as the postal service has about what people write in their letters. File sharing generate a lot of traffic, which affects the capability of the ISPs to offer quality in their services with low latency and high bandwidth.

Some ISPs have opted to punish their customers by either limiting or sabotaging P2P. American Comcast was during the fall of 2007 discovered in sabotaging P2P traffic for its customers [[23](#Sti07)] [[24](#Sve07)] [[25](#Sto07)]. By doing this the ISPs can avoid costly upgrades [[5](#Rob07)] [[6](#Gro07)]. But as I mentioned earlier, several legitimate companies have embraced the Bittorrent technology to distribute large files. Blizzard Entertainment uses it to distribute updates for its game World of Warcraft with over nine million players the world over for example. In many cases Comcast is the only ISP with bandwidth available to customers. Either because they live outside of the metropolitan areas or because competition is low in their specific neighbourhood. A situation worth comparing with the position that Swedish Telia enjoyed a few years ago. Even in the music business where the resistance against file sharing traditionally has been the fiercest have they started to realize a couple of the benefits of offering the material free without limitation. The band Radiohead released in 2007 the album *In Rainbows* free of charge on their web page and offered their fans to pay what they thought the album was worth by donating over Internet. So far the album has earned 62 million Swedish kroner, which went directly to the band without any middle men.

Time will tell how the question of copyrighted material will be solved. File sharing and P2P is here to stay. Not least because of some legal difficulty in attacking torrent trackers such as *The Pirate Bay*. Since no movies, no music etc are stored on the servers of The Pirate Bay, the people behind it cannot be accused of violating copyright directly. Encryption and other methods will in the future make it hard for authorities and trade associations to identify file sharers.

I want to make it clear that I have not written this thesis with the purpose of tracking people breaking Swedish law. The algorithm I propose to identify P2P traffic cannot be used to distinguish between legal and illegal file sharing because it does not look at the data being sent and I actually presume it to be unreadable by being encrypted in some way. There are other reasons for wanting to identify P2P traffic that I briefly mentioned earlier and it is because of those reasons I have based my intentions.

## 2.6. Traffic shaping

Information about which type of traffic a broadband user is generating is of interest for several reasons. Not least because ISPs try to minimize costs and maximize profits. The best customer a provider can have is someone who pays for a lot of bandwidth but doesn't use it. Without a doubt, the most popular form of paying today is a fixed monthly rate. you pay to have access to a certain bandwidth regardless of how much or little you use it. Providers on the other hand pay their own providers based on the amount of data passing through their network and not based on the bandwidth [[30](#Kar04)]. To send data to a network on the other side of the world costs more for the provider than to send it within their own network. Where the data is going is interesting in order to be able to minimize costs.

This motivates why the majority of ISPs in their terms of agreement include a clause saying basically that they have the right to terminate the service if the customer use their connection in an "abnormal" or "unreasonable" way. The provider reserves the right to define which amount of data is "normal" or "abnormal" [[31](#Gre07)]. This is often confusing for customers since they believed that they paid for a bandwidth, and normally it is technically impossible for them to exceed that.

A more customer friendly motivation to identify and reshape traffic is QoS[[13]](#footnote-13). An ISP has a limited bandwidth available. If all customers would use their connections maximally at once, the equipment of the provider would likely be overloaded and the customers would experience delays in transfers, what is known as "latency".

But there are services which customers still expect to function. If the provider in addition to broadband also offers an IP-telephony service it is reasonable to demand that the phones should always work, except in the case of a power outage. It would be unreasonable if there was half a second delay in phones just because a million Swedes log in to Facebook. The same goes for IPTV, online games, streaming video and music and to some part also regular web traffic. Several important services provided by government, bank and so on are now often available on the Internet and only enhances the importance that some services remain functional regardless of traffic load.

Peer-to-Peer traffic uses (at least in the case of file sharing) per definition a lot of bandwidth. The applications are designed to use all available capacity. In addition, peers can be localized anywhere in the world. The customer doesn't care from which part of the world he downloads from or uploads to, as long as it's fast. For the provider however it is, as said, very interesting.

One way to minimize costs and possibly increase speeds and minimize delays for the customer would be to implement a form of P2P proxy at the ISP with "Cache Discovery Protocol" [[32](#Cac07)]. The proxy server would provide popular files and could thereby reduce the number of transfers to other providers' network while at the same time probably offering faster transfers since the proxy server is likely to have a greater bandwidth to the customer than a computer on the other side of the Earth would have. Such an implementation for all types of files is quite improbable as long as the witch hunt [[33](#Orl07)] on file sharers keep on.

### 2.6.1. Quality of Service

What QoS means is in constant flux. I have chosen to mention four quite constant factors and how they affect different services.

* Error handling

If we take a file transfer as an example then every byte must be delivered correctly if the file is not to become corrupt during the transfer, which probable makes the file useless. This motivates why services that depend on correct data primarily go through TCP where data that is corrupted or lost along the way is retransmitted enough times until it arrives at the destination. Streaming video for example is not at all as dependant on error free transfers. Losing a frame will generally not affect the video viewing experience until enough frames are lost or corrupted.

* Bandwidth

File transfers are also quite dependant on bandwidth. Users want them to go fast but are content even if they are "a bit slow". Streaming video on the other hand has very strict requirements on bandwidth depending on the quality. In the case of HD video a very high bandwidth is needed for the video to be able to be played at normal speed. If the speed is decreased below the limit we have to use buffering, which creates annoying delays during the video.

* Latency

Online games are very dependent on low latency. If you shoot a rocket toward an opponent the opponent must be notified of that within a few hundredths of a second. Already at a delay of a few hundred millisecond many games start to become unplayable since cause and effect is removed. A player can shoot the opponent first, on his own screen, while the server registered the shot of the opponent first and thus the player dies, even though he never saw his opponent fire his gun. File transfers on the other hand have very low demands on latency. It doesn't matter if there's a delay of a few seconds as long as the bandwidth can be kept high and stable. Streaming video and audio are not affected greatly by latency either as long as it is constant.

* Jitter

If the delay isn't constant an effect known as jitter is generated. If a connection has a lot of jitter it means that deviation of the latency is high. Since different packets can be sent over different routes they will be subjected to different delays. Telephone services are very dependent on jitter being low. If the delay varies too much it becomes difficult to recognize speech. For streaming video and audio you can compensate by using a buffer, more jitter means a larger buffer is required.

As I mentioned, you can use a buffer to compensate for some of the problems. Another more simple method would be to simple overcompensate in terms of equipment. A provider would in that case for example place five hundred customers behind a router that could handle a thousand. Finally you could use traffic shaping, which traditionally have used algorithms such as "Leaky Bucket".

"Leaky Bucket" is essentially a large buffer. When traffic arrives to the router a buffer is filled. Traffic is sent from the buffer in an even and fixed rate. If the buffer were to become full then all incoming traffic is rejected, or alternatively allowed to pass unchecked. This stabilizes jitter and bandwidth use but you potentially introduce larger delays and even errors in the traffic if the buffer were to be filled and data discarded. The best thing would be to prioritize different types of traffic differently, which would demand that your are able to identify the type of traffic in the first place.

### 2.6.2. Some identification methods

There are many ways to identify network traffic. I will only mention a few of the overall methods.

#### 2.6.2.1. Port identification

Most types of traffic, for example web, FTP, IRC or email, are sent almost exclusively through the well known ports[[14]](#footnote-14) [[34](#Int07)]. The same is true for most applications. This makes it easy to divide the traffic by type without any real processing. The foremost weakness is that there are no technical obstacles to send for example FTP traffic over port 80, which leads to traffic using non-standard ports are misidentified. File sharing protocols like Bittorrent and Gnutella have standard ports[[15]](#footnote-15) but lately it is increasingly common to use non-standard and even completely random ports [[35](#Kim03)] [[36](#San04)].

Surveys have shown that even if port identification manages to identify a lot of P2P traffic today, a large amount of unknown traffic remains [[37](#Ger03)]. As more and more P2P clients use non-standard ports this unknown traffic will increase.

#### 2.6.2.2. Deep Packet Inspection

In DPI, you look at both the packet header and packet data. From the head you can determine source and destination among others. From the data you can potentially determine everything else. It could be mentioned for example that in a Bittorrent transfer the first piece of data that is sent is the word "Bittorrent". This has traditionally been an effective method is widely used by companies such as Cisco, IBM and other major corporations [[38](#Dee07)]. However, newer versions of the more popular Bittorrent clients implement RC4-encryption of the data [[19](#Wil07)] [[20](#μTo07)] [[21](#Azu07)] [[22](#Liv07)] which makes an inspection of the package data meaningless. Further, the ethics and in some cases also the legality of data inspection is questionable [[39](#Sog07)] [[40](#Sog071)] [[41](#Ban07)] [[42](#Ban071)] [[43](#Ley07)]. In addition, you are forced to analyze more data when you in addition to the package head also inspect the data which makes DPI more resource demanding than other methods.

#### 2.6.2.3. (Shallow) Packet Inspection

With ordinary SPI you only inspect the package head. The information you have access to is not much more than source and destination addresses. It is hardly possible to draw many conclusions from this alone but I want to demonstrate in this paper that despite this, it is possible using only SPI with good probability still identify P2P-like traffic.

SPI also don't introduce the ethical (or possibly legal) dilemma of DPI. One can compare it to the fact that your phone company naturally knows who you called and when, or it would be difficult for them to connect your call. They however know nothing about what was said during the actual conversation. Police have in interest in being able to listen in on criminals, but have traditionally only been able to do so in connection to a significant suspicion. The same should reasonably apply also to traffic such as email and other communication over the Internet.

#### 2.6.2.4. TCP-UDP pair identification

Several P2P protocols use both TCP to transfer files, as well as UDP as a control stream. And in the case of Bittorrent: to "discover" new peers. This was used by Karagiannis et al [[30](#Kar04)] in combination with port identification to identify P2P. There are many programs that use both TCP and UDP simultaneously, for example online games. In the mentioned article port identification was used to identify and remove non-P2P traffic. The weakness is thus the same as for that method and it is not clear if the protocols of the future will use both TCP and UDP [[44](#Wil071)].

### 2.6.3. Handling large amounts of traffic

A study by British CacheLogic has shown that up to 35% of all traffic is due to Bittorrent [[14](#Pas07)]. It also turns out that 20% of the users are responsible for 80% of the traffic [[37](#Ger03)]. The natural conclusion ought therefore be to limit the speed of Bittorrent traffic and/or the users utilizing a lot of bandwidth. Something that is known as *bandwidth throttling*.

One very notable case was, as mentioned, when the American ISP Comcast in 2007 was revealed to limit the bandwidth of Bittorrent and Gnutella by sabotaging the uploads of their customers [[23](#Sti07)] [[25](#Sto07)]. Comcast injects (at the time of writing, this was still going on) packets appearing to originate from the other party in the transfer and asks the target computer to terminate the transfer. In this case no bandwidth throttling was used but the effect is the same since transfers are repeatedly cancelled, lowering the average speed.

Blocking unwanted traffic is something that is generally used on restricted networks, for example university networks. LDC, Lund University's own provider, blocks a large range of ports, among them well known P2P ports [[45](#Anv07)]. All traffic going in and out of the network is checked, and if it is determined to be P2P the IP-address is blocked. Complete blocking of a certain type of traffic is hardly something that could be considered acceptable for a regular ISP, but not unusual [[46](#Wit07)].

A much less incursive proposal is to return the idea that you pay for the amount of data you transmit instead of bandwidth. Then the customers would pay for the costs involved with a significant P2P use. One proposal which is given by Altmann and Chu [[47](#Alt01)] is based on a dynamic speed limit. You could also imagine that non-P2P traffic was given unlimited speed (as limited by technology) while P2P and other "heavy" traffic is limited to a speed you have paid for. There are many possibilities, if you are able to identify what is P2P and what is not.

# 3. Part 2: Separating the elephants from the mice

Part 2 goes through how you can use flows to identify Peer-to-Peer. Comparisons with previous work are made and the methodology of the investigations are explained. Finally follows the results of the investigations and my final conclusions.

## 3.1. Identifying Peer-to-Peer by counting flows

While reading papers and articles about data streams and P2P identification, I wondered if it were possible to draw any conclusions regarding P2P use by only looking at the number of flows that could be associated to an IP-address (where a flow is defined as an IP:port pair). More importantly, I wondered if it could be done in real time at such high speeds that occur at the edge of a provider's network (up to 40 Gbit/s).

There are no Swedish or English investigations, as far as I know, that try to do this in real time with a possibility of being implemented in SRAM. A similar offline investigation was done by Karagiannis et al [[30](#Kar04)].

Since a lot of P2P-traffic is encrypted today, and because of the ethical and potential legal issues arising from inspecting the packet data (you're literally wire-tapping the traffic), it was obvious that only shallow packet inspection could be used. I have chosen to limit myself to TCP-traffic since that is what file sharing almost exclusively uses. There are no difficulties in also including other protocols. It is only a question of implementation since package heads differ between protocols. The principle is the same; all packages have a source and a destination, regardless of protocol.

### 3.1.1. Challenges

Every package must be identified as belonging to a flow that has already been observed or as belonging to a completely new flow. A gigabit link will potentially handle over a million TCP-packages each second. We must therefore identify the packages quickly and efficiently. Because this is a question whether the package is a member of the set of "already seen packages", a Bloom filter is well suited for this task. It is fast, it uses little memory and despite its randomized nature it only gives rise to a small amount of errors. The time used is O(k), where *k* is the number of hash functions used. The memory required is *m* = O(n) bits, where *n* is the number of flows that the Bloom filter is expected to be able to handle with good probability.

With the formula presented in chapter 2.4 we can calculate that for an *n* of 100 000 (number of flows) and an *m* of 1 million bits (less than 128 kB) the probability of false positives is less than 1% for five hash functions, and is optimal for seven hash functions. Since TCP primarily transmits data it seems reasonable to assume that a flow will, on average, use 10 packets or more. The amount of data transmitted in 10 packets is less than 15 kB.

When a new flow has been identified, we wish to use it to calculate the total number of flows related to the different IP-addresses. The number of IP-addresses can be large (in the worst case every flow belongs to two new and unique IP-addresses). Hence there can be a lot of packages invoking this process and it must be basically as fast as the previous Bloom filter. I have chosen to use a counting Bloom filter here. It is fast, uses a limited amount of memory and has a low probability of false positives.

A CBF uses more memory than a regular Bloom filter because of its use of counters instead of single bits. It is possible however to limit the size by noting that far from all flows will belong to unique addresses. Individual addresses will occur in several flows. It's all dependant on the number of flows generated by individual addresses. It also depends on the direction of the flows. If the machines within the network only communicate with each other there is no need for a large filter, but if the machines instead communicate with machines on foreign networks the size requirement increases. Further investigations could demonstrate how many flows are generated by computers on average. For my implementation I substantially oversized the filters. The number of flows per computer varied between 1 and over 50, depending on the type of activity.

A link with a 10 Gb/s capacity can handle 1000 10 Mb/s connections (a fairly common speed for both ADSL and LAN connections) concurrently without problems. Assuming that 16 bit counters are used in the CBF, which I used in my implementation, we can use the regular formula for Bloom filters to calculate that a filter of 256 kB and 131072 counters has an error probability of less than 1% for 13 663 IP-addresses. Which means that the average number of IP-addresses should not exceed 13 663 if we are to stay below a 1% error rate at high load. A thorough investigation of the number of observed flows could answer what the optimal size of the CBF is. Counters of 16 bits mean a maximum of 65 535 flows, while in reality 8 bit counters (255 flows) would probably have sufficed.

When thus an IP-address belonging to many flows has been identified, it is added to the final list together with the number of flows. The purpose of this list is to keep track of the average number of flows. To keep this value fairly dynamic I chose to reduce the average to a single measurement (normally an average value represents several measurements) after a few measurement intervals. This in order to counter the situation where an IP-address that is constantly associated with a lot of flows for a long time but is suddenly disconnected remains in the list and slowly decreases to zero average flows. So the list contains the average number of flows for at most the latest *y* seconds. After *y* seconds the average value is reduced to be equal to a single measurement, and is then free to fluctuate faster once more. The reason for this list and not using the value of the CBF directly is because I wanted to reduce the possibility of borderline cases where an IP-address varies between P2P and non-P2P.

Regular web traffic give rise to a small number of flows according to the tests made, but it can be linked to flows in small peaks with long minimum values in between. If a cyber café were to be situated behind a NAT router[[16]](#footnote-16), and potentially have a hundred people surfing the web from the same IP-address, it might mean that these peaks add up and surpass the limit for what the algorithm would identify as P2P. But it would quickly fall below the limit only to quickly rise again. If the router of ISP directs traffic differently depending on the number of flows the traffic of the cyber café would constantly be sent differently (this would be a false positive).

Constant updates in the internal routing list would also be un-desirable. So I chose to judge the average number of flows per second, or some other small period of time, during a longer window. This in order to allow long periods of inactivity to balance out short but high peaks. The same error could of course arise here as well, that an IP-address is on the limit. But that means that possible actions are only taken a few times per minute or less instead of every few seconds. This last step does not have as high demands on it because the number of P2P identified IP-addresses are limited. But we have already spent some processing time on the Bloom filter and CBF before this step, so it must still be fairly efficient.

I chose to represent this list of potential P2P addresses with a balanced binary search tree, specifically a red black tree. This despite memory access taking longer time than calculating a value in the CPU which a hash table would have required. There are however never that many pointers that have to be followed; less than 10 pointers which is the case if the tree contains less than 1024 P2P identified addresses (and in my tests this would never go beyond a handful). Primarily, I chose to use a binary tree because my investigations would take place at fairly low speeds and because it eliminated all kinds of measurement errors that might arise from Bloom filters. If the algorithm were to be implemented in a router it would almost surely demand something better than a binary tree. One suggestion for improvement is given in 3.1.4.

### 3.1.2. The average value list

The list has the usual operations *Insert, Delete, Search, Successor, Predecessor*, and so on that all lists do. In addition it also supports the operations: *Average* and *Reset*.

Insert(key k, measurement value zk):

Associate key k with measurement value zk,

If k is a new value then

the average value ak = 0 and

the counter ck = 0

Average: // Calculates the average value of inserted measurement values.

For all keys k in the tree,

ak = ,

ck = ck + 1,

zk = 0

Reset: // Calculates the total average during the latest period.

// Reduces the average to a measurement.

For all keys k in the tree,

zk = 0,

ak = ,

ck = 1

*Insert* is called once or several times to add the latest measurement and *Average* calculates a current average. In *Reset* ck is set to 1 and an average over the entire window *y* is calculated. If a source with a lot of flows is added late in a window it will be reported as having a low average number of flows per second over that period, but will get a more correct value during the next window. As said, it is a compromise between precision and the wish to have previous values always affecting later ones, in order to compensate for drastic variations. If a key *k* is inserted at the end of the period its average will be based on a smaller number of measurements than keys that have been inserted earlier. But when a key finally ends up in the list, *Reset* will be able to calculate the average value for the entire window. Would a key suddenly not get more measurements the average is calculated with *zk* = 0 until the average value drops below *T* and the key *k* is removed from the list. It is therefore easy to be inserted in the list (by in some period *x* exceed *T* flows), but keys that are unable to keep an average value above *T* (and here I mean the true average value) will quickly be removed from the list.

### 3.1.3. The algorithm

Here is the pseudo code for the algorithm. It is easiest interpreted as two threads. In parenthesis the responsible data structure is specified in those cases where it might be unclear.

**[Thread 1]**

For every package p

If p belongs to a previously unseen flow (BF)

Increase the flow counters for the two IP-addresses that p is transmitted between (CBF)

If any of the counters exceed T

Insert the IP-address in the average value list of P2P-addresses together with the number of flows zk.

**[Thread 2]**

For every measurement interval x (a second or some short period)

Clear the list of flows and flow counters (BF and CBF).

Calculate the average value of the period for every IP-address in the list of P2P-addresses.

For every measurement interval y (larger than x)

For every address in the P2P list,

Calculate and print the total average during the latest period y.

If the average is less than T, delete.

Else, reduce the average to a measurement.

*T* = The smallest number of flows above which an IP-address is designated as possible P2P.

*x* = A short period of time, maximum a few seconds.

*y* = A longer period of time, ideally more than thirty seconds.

### 3.1.4. Possible improvements

The operations that dominate the work load in the algorithm are the hash functions of the Bloom filter and counting Bloom filter. If a provider is only interested in its own customers and thus is only interested in counting the number of flows for those IP-addresses they themselves own, the CBF could be replaced by a regular hash table of counters if the number of relevant IP-addresses isn't too high. A good hash function that doesn't collide for these addresses could quite easily be constructed, as I mentioned in 2.4. The space would then be (if 16 bit counters are used) *2∙n* byte, where *n* is the number of addresses that will be observed. If *n* still is 13 663 (as in my previous example for CBF) this simple hash table will only use 27 kB of memory compared to 256 kB for the comparatively dimensioned CBF would. Operations take O(1) time instead of O(k). Counters of size 1 byte would likely suffice, which would mean a size of exactly *n* bytes.

With the same argument, the last list of average values could also be replaced by a simple hash table. It would not entail the same drastic improvement in memory usage as for the CBF, since here we only save information about those addresses which we believe are using P2P. But you would achieve a consistent running time of O(1) for thread 1 where most of the work is done, which would be a desirable guarantee for a very fast router.

The optimal length of the periods *x* and *y* can also be improved. I chose *x* = 1 second and *y* = 30 seconds or 60 seconds arbitrarily because it was simple. It is possible that other values for these variables could yield better results. It is also easy to imagine other values that could give drastically worse results.

The algorithm, as it is described above, makes use of Landmark windows. A slight modification of the list of averages changes the algorithm to use sliding windows instead. I implemented such a test version of the program but no experiments were made because of lack of time. So it is not known whether this would be an improvement or not. Below follows the changes for list required. In addition to that, the difference is that average values never are reduces to measurement values, e.g. the final row of the algorithm above is removed.

Average: // Calculates a current average value.

For all keys k in the tree,

ak = ,

zk = 0,

If ck < W

ck = ck + 1,

If ak < T1

delete k from the list

Where *W* = the length of the sliding window in number of smaller periods *x* and *T1* = a lower threshold (*T1* < *T*) at which the element is deleted from the list.

*T1* is needed since an element might be varying around *T* and will be constantly deleted before a stable average value can be established. The Landmark version does not have this problem because an element is removed at most once every *y* seconds.

### 3.1.5. Comparison with a naïve implementation

To really appreciate what Bloom filters have to offer, here I compare with a theoretical implementation using balanced search trees. The list of average values in my implementation has to be considered pretty naïve as well so I will only compare the difference in the first two steps. I have chosen to compare with balanced trees because they have logarithmic search times.

Balanced search trees have a memory use of O(n). To be able to compare flows the list must save information about IP-address and port number, for the source and destination. In total 12 bytes are required per flow. In addition a tree must also save a couple of pointers for children and parent nodes. It depends on what specific structure you use but for a red black tree, three pointers of 4 bytes each would be required, that is an additional 12 bytes per element. The Bloom filter on the other hand use around 1.25 bytes (10 bits) per element to minimise the chance of false positives. So in the first step, we gain at least a factor of 10 in memory efficiency if we use a Bloom filter. If we also include the pointers of a red black tree, it becomes a factor of 20.

The reason you can't use for example arrays to get rid of the pointers is because they have a linear search time which would have been far too slow. Even logarithmic search time is too slow when the speed becomes high. Also balanced search trees suffer from a lot of memory references. The bottleneck is, as I have said, precisely memory access, which motivates the desire to implement the algorithm with SRAM. Bloom filters offer what seems to be the perfect compromise between memory use and number of memory references (a constant number). All that is required is that we allow the algorithm with some small probability to occasionally return the wrong answer.

### 3.1.6. Related work

Others have in the last years attempted to utilize the flow pattern of the traffic to identify P2P and other traffic. The work I mention here are the ones I am aware of.

*Remco van de Meent, Aiko Pras, "Assessing Unknown Network Traffic"* [[48](#van04)]

van de Meent's and Pras's idea is to identify induced flows which otherwise perhaps would not be identified correctly. They give as an example an FTP-transfer where a control connection on port 21 induces a transmission connection on port 22 where the actual data is transmitted. They only inspect the package heads and base their identification on a comparison with the known ports. Their algorithm is not adapted to be used in real time. The experiments that were perform on a university network of about 2000 connected students ultimately show that their algorithm only offers a marginal improvement over ordinary port identification.

*Kim et al., ”Towards Peer-to-Peer Analysis Using Flows”* [[35](#Kim03)]

Also Kim et al. makes heavy use of port identification. If any of the IP-addresses uses a port that exist in their list of P2P-ports, the flow is identified as P2P. This list of P2P-ports is generated by an extensive analysis of the packages found in dump files of traffic. This also potentially applies to the package data. The actual identification occurs in real-time and is able to identify a large portion of P2P on the university network where the experiments where done. Still, they are unable to identify flows where both parties use previously unseen ports and they ignore traffic that flows across known ports for non-P2P services.

*Wagner et al., ”Flow-Based Identification of P2P Heavy-Hitters”* [[49](#Wag06)]

Here an algorithm for real-time investigation of Netflow data has been implemented. Netflow is something used by Cisco's routers. Part of the traffic selected at random and its flow representation is sent as a UDP stream for analysis to a destination decided by the administrator. The fact that Netflow does not analyse all traffic but only a small part of it creates from the outset possibilities for errors and false negatives. It is this UDP stream that has been analysed in the experimental section.

The algorithm here as well is based on port identification. They motivate it by saying that while a peer might use an unknown port it will still often communicate with other peers using the standard ports. During a longer period of time (an hour) data of which ports each peer has communicated over is saved and a peer is classified as P2P if it has a flow that uses a P2P-port during this time or if it potentially has communicated with a P2P identified peer.

To confirm the reliability of their algorithm, they also introduce a validation method in three steps. First, they confirm that the peer they want to check is available by an ICMP echo, a ping request. Then they try to establish a connection to the port they suspect being related to P2P through TCP. In the final step, they actually try to initiate a connection with the P2P-protocol they suspect is being used by the peer. Something that didn't work for Bittorrent since that requires you to have knowledge about the file that is being shared through the torrent.

I feel that the biggest weakness is that Wagner et al. completely ignores flows that have one or both ports outside the 1024 - 30 000 interval to avoid false positives. It was shown that most P2P-traffic takes place in this interval. Would their algorithm be used at large scale, the P2P networks would surely adapt by potentially sending all traffic over known ports between 1 and 1024, something I assume to be a possibility in my own analysis.

*Karagiannis et al., ”Transport Layer Identification of P2P Traffic”* [[30](#Kar04)]

Finally we have Karagiannis et al., who try to identify P2P traffic regardless of what ports it is transmitted on. The algorithm is however not intended for use in real-time. Only the first 16 bytes of the package data is investigated where they search for known bit strings that are transmitted in P2P-protocols. They are able to identify a large portion of P2P and previously unknown protocols but mention that data encryption prevent some of the results to be verified.

Their identification have two main phases. In the first phase IP-addresses that share both a TCP flow and a UDP flow are identified. Six out of nine protocols in the experiments use both TCP and UDP, among them Bittorrent, Direct Connect and Gnutella. In the second phase, all flows related to addresses where the number of ports used is equal to the number of unique IP-addresses are investigated. They note here that for example web traffic have a larger number of ports than IP-addresses since a web browser will initially open several connections to download the content of the page in parallel. The number of ports in that case exceeds the number of unique IP-addresses and therefore avoids being wrongly identified. To minimize the number of false positives they exclude flows with ports, and to some extent also behaviour, that correspond to known services such as email, FTP, SSL and DNS within TCP. It is also noted that there is nothing preventing P2P-clients to use these ports, which I mentioned earlier. If one IP-address in a flow is identified as P2P then the other address is also identified as P2P. In the same way, IP-addresses communicating by non-P2P is marked as non-P2P. For IP-addresses with a lot of connections (more than 20) they are able to identify with very good precision if they use P2P or not.

This is the only work I know of where an attempt is made to identify P2P without analysing either package data or port numbers of P2P-traffic. Working backwards, by first designating a large portion of the traffic as P2P and then removing easily identified non-P2P traffic, makes it possible to identify previously unknown protocols. Something that is important if the algorithm is to be used in the future with currently unknown protocols. The algorithm is however not intended to be used in real-time and cannot be modified without very large alterations because it is based on an amount of comparisons which would entail a large amount of memory references and calculations.

## 3.2. Implementation and approach

Here the methodology behind the investigations together with the implementation of the algorithm is explained.

### 3.2.1. Implementation

The algorithm was implemented in C++. The Bloom filter and counting Bloom filter, together with the red black search tree used for average values, were implemented completely by myself, excluding the hash functions for the Bloom filters for which I used an external library for [[50](#Gen07)]. To capture TCP-packages from the network and to read dump files of traffic, libpcap was used [[51](#TCP07)].

The sizes of the Bloom filter and the CBF were dimensioned to handle a total of one hundred thousand flows (128 kB) and ten thousand IP-addresses (256 kB) respectively. Since a binary search tree was used for the list of average values, the total size was not constant. But the total size would not exceed 1 MB without also exceeding one hundred thousand flows.

In the case where traffic was captured in real time from the network, two threads were used as demonstrated by the pseudo code. Thread 1 was then a callback function which was called when a package was captured. When traffic was read from dump files however, only one thread was used. The dump files were saved by TCPdump which also saves the time of arrival of the package in the dump file. This was used to keep track of where on the time axis the program was located and if it should reset Bloom filters etc. Since TCPdump uses pcap, this method is also applicable in a real time measurement and I also implemented such a version of the program. However I preferred to use two separate threads to spread the work load over time and not just do work when a package was captured. All results that are presented come from dump files, but could also have been made in real time of course.

I chose to handle the synchronization of the threads by locking the data structures. This was not an issue for the experiments I performed since the load was so low. In a real implementation with far higher speeds the synchronization would probably need a more careful consideration. It should be solvable. One can for instance imagine using a double set of data structures and simply replace them every other time.

To get a better picture over traffic patterns keys were never deleted from the list of average values. I chose to keep all that at some time ended up in the list in order to see how the traffic behaved between peaks.

### 3.2.2. Measurement data

The number of flows required to end up in the list of average values was set to 2. The measurement data was collected from one computer at a time with exception from a few where the data was collected from a computer acting as a NAT router for a different computer. It happened that some IP-addresses except the local ones ended up in the list but these were removed from the diagrams, since these computers were only the ones that the test computer was in contact with and nothing can really be said about them.

The data was collected by me, but a few also by two other students. I tried to get data from as many possible different types of traffic I could imagine, P2P, web, mail, FTP, online games, VPN, etc. In most cases several occur, for example web, mail and instant messaging traffic for most web tests. This because it is common for these to be active at the same time. Also a few program normally not related to Internet traffic would search for updates, such as Windows Update or similar and could therefore have affected the results in a very limited fashion.

Web surfing and FTP-transfers were done with Firefox v2. Instant messaging occurred with the programs Adium (Mac OS X) and Pidgin (Windows XP). Mail was retrieved by Thunderbird v2 or through web mail. P2P was represented by the Bittorrent clients Transmission v0.96 (Mac OS X), μTorrent v1.7.5 (Windows XP) and Blizzard Downloader (Mac OS X).

As said the data was collected by TCPdump and was done in a varying amount of times, often 10 minutes or 200 000 packages. Hence the length of time varies between the different diagrams. 200 000 packages was selected because at high speeds the dump file grows quickly, even when only TCP-heads are saved. In some case some four million packages were analysed, to get a picture of a Bittorrent transmission from beginning to end. If nothing else is mentioned, it is my own traffic that I have analysed and the traffic was registered on the same machine that sent and received it. The maximum speed for the transmissions was 10 MB/s for both upload and download.

## 3.3. Results

3.3.1 demonstrates the positive results. 3.3.2 shows those results that were clearly negative, and briefly explains why and how Bittorrent can avoid detection.

### 3.3.1. Web traffic and Bittorrent

As we can see in diagram 3.3.1.1, ordinary web traffic is far from stable in the number of flows per second. Even if for some second more than two flows are registered, neither of them reaches an average value of two. It should be mentioned that "Surf 1" was not done by myself.

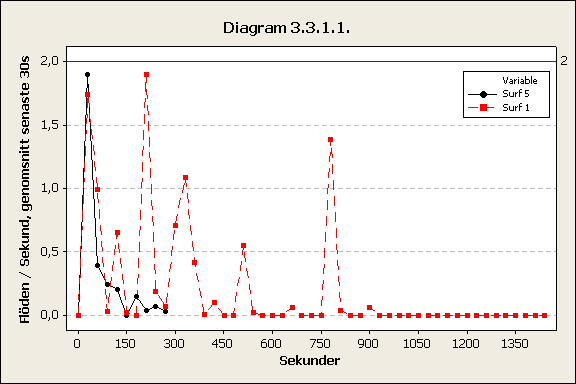


Diagram 3.3.1.1.: Web surfing performed by two different people.

In diagram 3.3.1.2. we see the first piece of Bittorrent data. The speed was very low, around 10 kB/s down and 60-70 kB/s up. The swarm was only composed of 4-5 peers. In the same diagram are also two plots of web surfing. In "Surf 2" I tried to click on links often and quickly, in an attempt to raise the number of registered flows which succeeded. "Surf 3" is primarily an FTP-transfer.

Just as in diagram 3.3.1.1. the web surfing was not able to keep a stable average flow number. This coincides well with what I expected about web browsing. The Bittorrent traffic however is able to almost consistently stay above 2 flows per second on average, despite the very low speed.

Another example of a very slow Bittorrent transfer can be seen in diagram 3.3.1.3. In the middle of the transfer one can see that the number of flows drop to zero. This is because I accidentally shut down the computer and it was therefore forced to once again search for peers once I had restarted it. The swarm consisted of about 13 peers and the speed was at about 25 kB/s down and 4 kB/s up on average during the two hours. Even this transfer is able to stay fairly consistently above two flows per second.

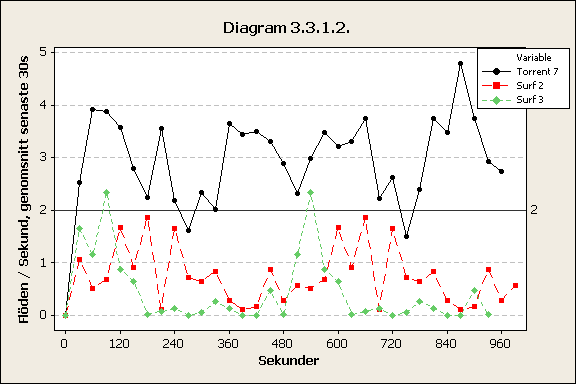


Diagram 3.3.1.2.: Web browsing in comparison with slow Bittorrent.

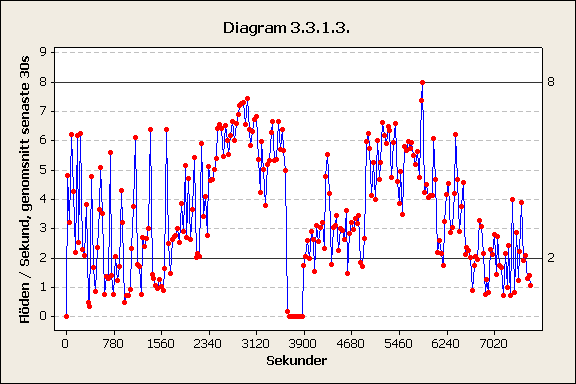
****

Diagram 3.3.1.3.: Slow Bittorrent.

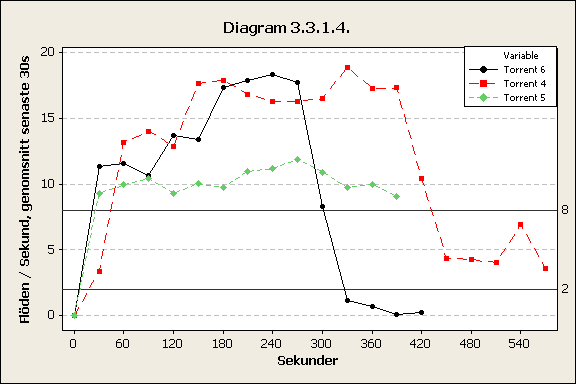
When it comes to fast Bittorrent the number of flows per second are a lot higher. In diagram 3.3.1.4. we see examples of three fast transfers. "Torrent 4" is not data collected by myself, but from what I was told the speed decreased significantly towards the end of the period and it was mostly uploading taking place. "Torrent 5" kept a speed of about 100 kB/s and the swarm consisted of about 20 peers. "Torrent 6" is the traffic generated by Blizzard Downloader, which is used to download updates for the online game World of Warcraft. What is unique about that compared to other Bittorrent clients is that is simultaneously downloads through http in parallel with Bittorrent. The speed was around 1 MB/s and http was responsible for about 90% of that. The P2P part is then comparable to "Torrent 5" with about 100 kB/s. After a while the program was terminated, which explains the sudden drop in number of flows at 300 seconds.

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

What the plots in diagram 3.3.1.5 have in common is that they all had very large swarms. "Torrent 1, 2, and 3" had swarms of nearly 2000 peers and "Torrent 8" hade around 400 peers. Also, all of them gave rise to very high speeds. They all stayed steady at 1 MB/s down and several hundred kB/s up. As we see, this also means very high average values of flows per second.

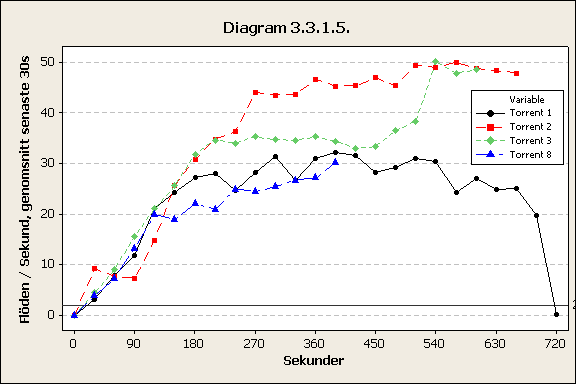


Diagram 3.3.1.5.: Very fast Bittorrent, around 1 MB/s and large swarms.

### 3.3.2. Error sources and methods to avoid detection

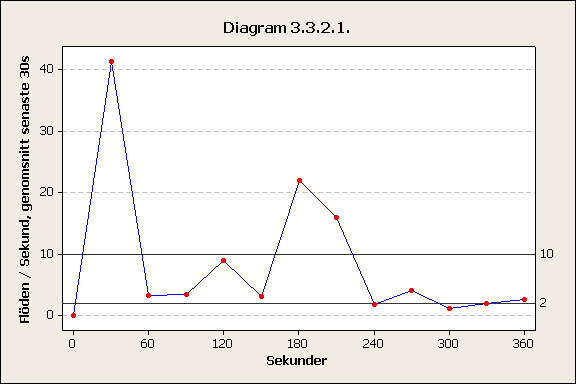


Diagram 3.3.2.1.: Simultaneously opening 23 bookmarks in a web browser.

I suspected that if you were to simultaneously open a large number of web pages, you would give rise to enough flows so that the average could be confused with P2P. The result can be observed in diagram 3.3.2.1 where I simultaneously opened twenty three bookmarks in Firefox and shortly thereafter opened them all again. As can be seen, the first peak is around twice as high as the second one. However, the second peak is spread across sixty seconds instead of thirty. This can be seen as a very good example of the error that can appear when measurement data ends up in two windows for Landmark windows.

Another way to generate a similar type of result as in 3.3.2.1 would be if you analyzed the traffic behind a NAT router. If the network behind the router is large enough, and enough people are browsing or similarly, as for a cyber café, a smoother curve could probably be achieved. Especially if enough of the people simultaneously click on links, peaks like those in 3.3.2.1 are likely to arise. Popular servers could also get a similar pattern.

There are also methods to conceal all flow related information. One such method is to send the traffic through another computer with the help of a VPN or SSH-tunnel. In diagram 3.3.2.2 we see that a fairly fast Bittorrent transfer becomes practically invisible when routed through a VPN-tunnel. For this test I used two computers, where one acted as a NAT router for the one using the VPN-tunnel. The traffic was registered at the router.

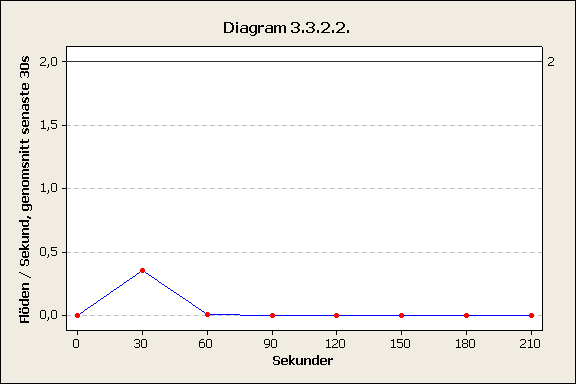


Diagram 3.3.2.2.: Bittorrent traffic through a VPN-tunnel via the GRE protocol.

The swarm was composed of 20 peers and the speed lay at 300-400 kB/s up and down. As a comparison we can look at "Torrent 5" of 3.3.1.4, which is the exact same Bittorrent transfer except with VPN turned off. After I shut down the VPN it was never able to reach the same speed which probably is because it ended up behind an active firewall. I only concern myself with TCP-traffic and since VPN in this case used the GRE protocol, it becomes basically invisible. Only a control stream was sent through TCP. But since all the traffic is directed towards the VPN server it would at most give rise to one flow just as other client-server applications.

For online games where latency matters there is a very small chance for false positives. My implementation was based on TCP and I did experiments with the game World of Warcraft that uses TCP. The average number of flows was less than 0.01 in each interval and I therefore decided not show that data as a diagram.

## 3.4. Conclusions

According to the situations I have analysed, the behaviour of Bittorrent traffic can be associated with multiple flows per second consistently over longer periods of time, while other non-P2P traffic does not. Web browsing in particular is characterized by small peaks with longer valleys in-between. The only time there is a question of false positives is when a lot of web pages are opened simultaneously. or when a lot of people simultaneously browse the web behind a NAT router. At least in the first case it would probably not matter greatly if you were redirected with a higher latency by you ISP for a few minutes. The time it takes to open twenty pages, together with the time it takes to actually read twenty pages, is so high that it probably doesn't matter if the server responds in an additional few hundred milliseconds.

Servers, if they are popular, can also be misidentified. But since ISPs often in their terms of agreement forbid the customer to run servers except for personal use, this would perhaps be considered a positive side effect that such traffic wasn't prioritized.

A game, which is very dependent on latency and correct routing, would give rise to at most one flow per second. It is after all a simple client-server application. It is also natural to shut down most non-essential programs to free as much resources as possible (CPU, memory) for the game and to minimise latency.

Bittorrent seems to give rise to a lot of flows as long as the speed is fairly large, about 100kB/s or more. Now when even homes in the countryside have access to 8 Mb/s ADSL, speeds of 1 MB/s on P2P-connections are no longer reserved for those with the best connections. It then probably doesn't matter if slow P2P-traffic isn't identified. The fast traffic, which also is the more expensive traffic, is identified with high probability.

It is also not hard to avoid detection. By using for example a VPN-tunnel, you can with 100% certainty avoid identification by your ISP. In my experiments I used a VPN-tunnel provided by the company Relakks [[52](#Rel07)] which offers VPN for the sake of anonymity for a fixed monthly charge. But regardless what kind of VPN is used, the flows must always be "set free" somewhere to reach their destinations. There it will be possible to identify the traffic. It also not impossible to imagine that Relakks or other companies offering VPN-services would be interested in prioritizing traffic differently depending on type.

A similar method would be to use a P2P-network such as Tor or Onion to hide your traffic pattern. Tor works by encrypting your traffic and routing it through a number of peers before it is sent to its destination. At the same time, you share part of your own bandwidth for others to route their traffic through. I have not done any experiments with tor, but I think that at a low load the pattern will look a lot like VPN. But if the speed would become high, maybe the number of connected Tor peers could give rise to a pattern similar to Bittorrent. But you would probably have to be connected to quite a few Tor peers for this to occur.

In my implementation I used a limit of 2 flows per second to initially suspect P2P. A higher limit would increase the threshold for false positives, but also increase the risk that slow P2P goes unidentified. In some of the diagrams, where the pattern can be considered unclear, I have marked 2 and 10 with horizontal lines. If the number of flows stays consistently above 10, I consider it to be clearly identified as P2P. These two lines would have to be joined somewhere between 2 and 10 flows per second. The optimal value of this threshold could probably only be found through extensive experiments using real traffic from an ISP.

Tests at higher speeds would also be a necessity to verify the performance of the algorithm. Since I have been limited to 100 MB/s (the speed of the local network) I have been unable to perform any relevant experiments to confirm the efficiency. Despite that I am sure that, with the help of the suggestions I made in chapter 3.1.4, the algorithm could be made fast enough to be able to handle speeds even at 40 Gb/s in real time. It is clear from its size that it can be easily implemented with SRAM.

# Bibliography

All referenced web pages are saved and can be sent upon request.

x

|  |  |
| --- | --- |
| [1] | Burton H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422-426, July 1970. |
| [2] | Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi, "Duplicate Detection in Click Streams," in *Proc. of the the 14th international conference on World Wide Web*, Japan, 2005, pp. 12-21. |
| [3] | Y. Zhu and D. Shasha, "StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time," in *Proc. of the 28th ACM VLDB International Conference on Very Large Databases*, 2002, pp. 258-369. |
| [4] | Andrew S. Tanenbaum, *Computer Networks*, 4th ed. United States of America: Prentice Hall, 2002. |
| [5] | Alejandro López-Ortiz, "Algorithmic Foundations of the Internet," *ACM SIGACT News*, vol. 36, no. 2, June 2005. |
| [6] | Lawrence G. Roberts. (2007, Dec.) Routing Economics Threaten the Internet. [Online]. <http://www.internetevolution.com/author.asp?section_id=499&doc_id=136705&> |
| [7] | Grant Gross. (2007, Dec.) Study: Internet could run out of capacity in two years. [Online]. <http://www.macworld.com/news/2007/11/19/internetcapacity/index.php> |
| [8] | Cristian Estan and 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, pp. 270-313, August 2003. |
| [9] | Jeffrey Scott Vitter, "External Memory Algorithms and Data Structures: Dealing with Massive Data," *ACM Computing Surveys*, vol. 33, no. 2, pp. 209-271, June 2001. |
| [10] | Andrei Broder and Michael Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485-509, 2005. |
| [11] | L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281-293, 1998. |
| [12] | Fan Deng and Davood Rafiei, "Approximately Detecting Duplicates for Streaming Data using Stable Bloom Filters," in *SIGMOD*, 2006. |
| [13] | Saar Cohen and Yossi Matias, "Spectral Bloom Filters," in *SIGMOD*, 2003. |
| [14] | J. Aguilar-Saborit, P. Trancoso, and V. Muntes-Mulero, "Dynamic Count Filters," *SIGMOD Record*, vol. 35, no. 1, March 2006. |
| [15] | Abhishek Kumar, Jun Xu, Li Li, and Jua Wang, "Space-Code Bloom Filter for Efficient Traffic Flow Measurement," in *Proc. of the 203 ACM SIGCOMM conference on Internet measurement*, 2003, pp. 167-172. |
| [16] | S.C. Rhea and J. Kubiatowicz, "Probabilistic location and routing," in *Proc. of INFOCOM*, 2002. |
| [17] | Michael Mitzenmacher, "Compressed Bloom Filters," *IEEE/ACM Transactions on Networking*, vol. 10, no. 5, October 2002. |
| [18] | (2007, Dec.) Gnutella. [Online]. <http://en.wikipedia.org/wiki/Gnutella> |
| [19] | (2007, Dec.) Direct Connect. [Online]. <http://en.wikipedia.org/wiki/Directconnect> |
| [20] | Adam Pasick. (2007, Dec.) LIVEWIRE - File-sharing network thrives beneath the radar. [Online]. <http://in.tech.yahoo.com/041103/137/2ho4i.html> |
| [21] | (2007, Dec.) Blizzard Downloader. [Online]. <http://www.blizzard.co.uk/wow/faq/bittorrent.shtml> |
| [22] | Peter Bowes. (2007, Dec.) Warner to start movie downloads. [Online]. <http://news.bbc.co.uk/1/hi/business/4753435.stm> |
| [23] | Burt Helm. (2007, Dec.) Bittorrent Goes Hollywood. [Online]. <http://www.businessweek.com/technology/content/may2006/tc20060508_693082.htm> |
| [24] | (2007, Dec.) Bittorrent Inc. [Online]. <http://www.bittorrent.com/> |
| [25] | Chris Williams. (2007, Dec.) Surge in encrypted torrents blindsides record biz. [Online]. <http://www.theregister.co.uk/2007/11/08/bittorrent_encryption_explosion/> |
| [26] | (2007, Dec.) μTorrent. [Online]. <http://www.utorrent.com/faq.php> |
| [27] | (2007, Dec.) Azureus. [Online]. <http://azureus.sourceforge.net/faq.php> |
| [28] | Adam Livingstone. (2007, Dec.) A bit of Bittorrent bother. [Online]. <http://news.bbc.co.uk/2/hi/programmes/newsnight/4758636.stm> |
| [29] | Sarah Lai Stirland. (2007, Dec.) Comcast Using Malicious Hacker Technique Against Own Customers, New Report Says. [Online]. <http://blog.wired.com/27bstroke6/2007/11/comcast-using-m.html> |
| [30] | Peter Svensson. (2007, Dec.) Comcast blocks some Internet traffic. [Online]. <http://www.msnbc.msn.com/id/21376597/> |
| [31] | Brad Stone. (2007, Dec.) Comcast: We’re Delaying, Not Blocking, Bittorrent Traffic. [Online]. <http://bits.blogs.nytimes.com/2007/10/22/comcast-were-delaying-not-blocking-bittorrent-traffic/> |
| [32] | Tim Weber. (2007, Dec.) Criminals 'may overwhelm the web'. [Online]. <http://news.bbc.co.uk/2/hi/business/6298641.stm> |
| [33] | Gregg Keizer. (2007, Dec.) Dutch Botnet Suspects Ran 1.5 Million Machines. [Online]. <http://www.techweb.com/wire/security/172303160> |
| [34] | T. Peng, C. Leckie, and K. Ramamohanarao, "Survey of Network-Based Defense Mechanisms Countering the DoS and DDoS Problems," *ACM Computing Surveys*, vol. 39, no. 1, April 2007. |
| [35] | (2007, Dec.) Botnet. [Online]. <http://en.wikipedia.org/wiki/Botnet> |
| [36] | Thomas Karagiannis, Andre Broido, Michalis Faloutsos, and Kc claffy, "Transport Layer Identification of P2P Traffic," in *IMC'04*, Italy, 2004. |
| [37] | Carey Greenberg-Berger. (2007, Dec.) Comcast Customer Uses "Unlimited Service" Excessively, Gets Disconnected For A Year. [Online]. <http://consumerist.com/consumer/comcast/comcast-customer-uses-unlimited-service-excessively-gets-disconnected-for-a-year-235585.php> |
| [38] | (2007, Dec.) CacheLogic and Bittorrent Introduce Cache Discovery Protocol. [Online]. <http://torrentfreak.com/cachelogic-and-bittorrent-introduce-cache-discovery-protocol/> |
| [39] | Andrew Orlowski. (2007, Dec.) RIAA sues the dead. [Online]. <http://www.theregister.co.uk/2005/02/05/riaa_sues_the_dead/> |
| [40] | (2007, Dec.) Internet Assigned Numbers Authority. [Online]. <http://www.iana.org/> |
| [41] | Myung-Sup Kim, Hun-Jeon Kang, and James W. Hong, "Towards Peer-to-Peer Traffic Analysis Using Flows," in *DSOM*, Heidelberg, Germany, 2003, pp. 55-67. |
| [42] | Sandvine Incorporated, "Meeting the Challenge of Today’s Evasive P2P Traffic, Service Provider Strategies for Managing P2P Filesharing, an industry white paper," 2004. |
| [43] | A. Gerber, J. Houle, H. Nguyen, and S. Sen, "P2P, The Gorilla In The Cable," in *Proc. National Cable & Telecommunications Association (NCTA)*, 2003. |
| [44] | (2007, Dec.) Deep packet inspection. [Online]. <http://en.wikipedia.org/wiki/Deep_packet_inspection> |
| [45] | Chris Soghoian. (2007, Dec.) Comcast to face lawsuits over Bittorrent filtering. [Online]. <http://www.cnet.com/8301-13739_1-9802410-46.html?tag=nefd.blgs> |
| [46] | Chris Soghoian. (2007, Dec.) Congressman to Comcast: Stop interfering with Bittorrent. [Online]. <http://www.news.com/8301-10784_3-9804158-7.html> |
| [47] | Eric Bangeman. (2007, Dec.) Advocacy group to FCC: Comcast's traffic blocking defense is bogus. [Online]. <http://arstechnica.com/news.ars/post/20071101-advocacy-group-to-fcc-comcasts-traffic-blocking-defense-is-bogus.html> |
| [48] | Eric Bangeman. (2007, Dec.) Comcast hit with class-action lawsuit over traffic blocking. [Online]. <http://arstechnica.com/news.ars/post/20071114-comcast-hit-with-class-action-lawsuit-over-traffic-blocking.html> |
| [49] | John Leyden. (2007, Dec.) Germany enacts 'anti-hacker' law. [Online]. <http://www.theregister.co.uk/2007/08/13/german_anti-hacker_law/> |
| [50] | Chris Williams. (2007, Dec.) Pirate Bay aims to sink Bittorrent. [Online]. <http://www.theregister.co.uk/2007/11/01/pirate_bay_new_protocol/> |
| [51] | (2007, Dec.) Användning av P2P-program på LUNET. [Online]. <http://www2.ldc.lu.se/security/P2P-services.shtml> |
| [52] | Stephen Withers. (2007, Dec.) Block P2P, Belgian court tells ISP. [Online]. <http://www.itwire.com/content/view/13351/53/> |
| [53] | Jörn Altmann and Karyen Chu, "A Proposal for a Flexible Service Plan that is Attractive to Users and Internet Service Providers," in *Proc. IEEE INFOCOM*, 2001. |
| [54] | Remco van de Meent and Aiko Pras, "Assessing Unknown Network Traffic," University of Twente, Netherlands, CTIT Techincal Report 04-11 2004. |
| [55] | Arno Wagner, Thomas Dübendorfer, Lukas Hümmerle, and Bernhard Plattner, "Flow-Based Identification of P2P Heavy-Hitters," in *International Conference on Internet Surveillance and Protection (ICISP)*, 2006. |
| [56] | (2007, Dec.) General Purpose Hash Function Algorithms. [Online]. <http://www.partow.net/programming/hashfunctions/index.html> |
| [57] | (2007, Dec.) TCPDump / Libpcap. [Online]. <http://www.tcpdump.org/> |
| [58] | (2007, Dec.) Relakks. [Online]. <https://www.relakks.com/> |
| [59] | (2007, Dec.) Static random access memory. [Online]. <http://en.wikipedia.org/wiki/Static_random_access_memory> |
| [60] | (2007, Dec.) DC++. [Online]. <http://dcplusplus.sourceforge.net/> |
| [61] | Nelly Visanji. (2007, Dec.) En miljon svenskar använder Facebook. [Online]. <http://www.idg.se/2.1085/1.129990> |

x

# 

1. For some problems a few times could be acceptable. [↑](#footnote-ref-1)
2. It's a constant struggle to keep the list small. It has a tendency to grow when one tries to minimise the waste of IP-addresses that inveriably occurs because of the implementation of networks compared to addresses. [↑](#footnote-ref-2)
3. This is usally a very high number since known services have standardized ports beginning at 1 and upwards. [↑](#footnote-ref-3)
4. Dynamic Random Access Memory [↑](#footnote-ref-4)
5. Static Random Access Memory [↑](#footnote-ref-5)
6. IPv4 uses 32 bit IP-addresses and 16 bit ports. The newer IPv6 uses 128 bit addresses but is really not in general use. [↑](#footnote-ref-6)
7. Assuming a C subnet is used, for example 192.168.0.X [↑](#footnote-ref-7)
8. An important demand in that case is that the hash function is not reversible. [↑](#footnote-ref-8)
9. MD5 hash value checksums are used to verify that a file has been copied without errors. It is not uncommon for a few bits to be scrambled during network transmissions, hence the need to verify the integrity. [↑](#footnote-ref-9)
10. Internet Relay Chat. [↑](#footnote-ref-10)
11. A user manually decides what files or folder on the computer that should be shared with other users. [↑](#footnote-ref-11)
12. Distributed Denial of Service. [↑](#footnote-ref-12)
13. Quality of Service. [↑](#footnote-ref-13)
14. 80, 21, 194 and 110 respectively. [↑](#footnote-ref-14)
15. 6881-6889 and 6347 respectively. [↑](#footnote-ref-15)
16. Native address translation. [↑](#footnote-ref-16)