Department of
Computer Science

Technical Report

Undergraduate Dissertation: Creating a Distributed Network Traffic Analyser

Sam Crawford

Technical Report 2007-08 July 2007

ISSN 1740-9497

Contact Address:
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

CREATING A DISTRIBUTED NETWORK TRAFFIC ANALYSER

Submitted by Sam Crawford
for the degree of
BSc (Hons) Computer Science
2007

Creating a distributed network traffic analyser

Submitted by: Sam Crawford

COPYRIGHT

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Batchelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:

This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed:

Abstract

Internet traffic is rising dramatically each year, forcing ISPs to look deep into their networks to better understand the data they carry. Traditional network monitoring tools provide insufficient resolution to identify the true source and type of the traffic. To make matters worse, many modern protocols now randomise their port activity too, making their traffic much harder to classify correctly.

This project presents a means of supporting effective low-level monitoring of high-traffic networks. Signature analysis and flow tagging are used to classify packets, permitting identification of all but encrypted traffic. The system has been developed to support large, distributed networks using only commodity hardware, making it more attractive to the smaller ISPs who cannot afford commercial ASIC-based traffic analysers. The finished solution was tested on an ISP's network in London for two months with very positive results.

Contents

List of figures

List of tables

Acknowledgements

I wish to thank Chris Rogers at Fluidata for allowing me to test the project on his network; this testing and the resulting user feedback have proven invaluable. Thanks also go to my supervisor, Dr Bradford, for his hands-off support throughout and to Rich and Sandy for proof reading this dissertation.

Chapter 1

Introduction

Since the earliest days of the Internet, network operators have been interested in measuring and monitoring their traffic. The widespread usage of tools such as MRTG and Cacti amongst ISPs, businesses and technical-minded home users is a testament to this fact. Caceres (1989) demonstrated in his publication an early interest in monitoring not only traffic volume, but also the type of traffic. Whilst protocol-level traffic analysers are still relatively uncommon in smaller ISPs and businesses, the demand for them has increased significantly in recent years.

This increase in demand, particularly in the UK and US, can likely be attributed to the growth in broadband availability and speeds, as well as the ever-increasing usage of file sharing applications. Just four years ago only 67% of the UK was able to receive broadband, often at less than 1Mbps (BT Plc, 2003). Take-up was understandably just 4.97% of households (Point Topic, 2007). Nowadays 99% of the UK can receive broadband, with an average available speed of over 4.5Mbps (BT Wholesale, 2007), and nearly 50% of homes now have it installed (Point Topic, 2007). In 2001 Napster, widely acknowledged as the first major peer-to-peer file sharing application, peaked at 26.1 million users worldwide (ComScore, 2001). Six years later, BitTorrent dominates the peer-to-peer market and claims some 135 million users (The Register, 2007). There is also the growth of Internet radio, video streaming websites (such as YouTube) and online gaming that also plays a major role in Internet traffic nowadays.

With this massive increase in available bandwidth and the applications to make use of it, network traffic volumes have unsurprisingly soared. The cost of “transit bandwidth” (bandwidth between two major network providers) now plays a big part in all ISP’s costs and is therefore something that must be watched carefully. Indeed, this is why the majority of broadband connections are contended. For example, you may have an 8Mbps 50:1 contended broadband connection at home, which would imply that if the network was fully utilised you would only receive 1/50th of that total 8Mbps bandwidth. This may seem unfair, but contention is present in nearly all networks (For example: consider a 16 port 100Mbps switch with a single 1000Mbps uplink port: 15 * 100Mbps = 1500Mbps, so the uplink must be contended). Problems arise with this contention model when a handful of users use more than their “fair share” of bandwidth for extended periods. This has led to the introduction of usage caps and “fair usage policies” amongst ISPs – concepts unheard of in the broadband market only 3 years ago.

File sharing applications have now evolved in to their third generation (as described by Madhukar et al (2006)), and are now employing port randomising activity and data obfuscation in an attempt to thwart traffic shaping and other restrictions. Traditional traffic analysers that use only port-analysis to determine traffic types have struggled to adapt to such developments.

There have been other growth areas in Internet usage too. Denial of Service (DoS) and Distributed DoS (DDoS) attacks attempt to overwhelm servers and network infrastructure by swamping them with perfectly valid requests (CERT, 2001). Open email relays are also a big problem for ISPs nowadays as they are often targeted by "spammers" (distributors of unwanted email) to distribute their wares.

It is with these issues in mind that the proposed system has been conceived. By determining the true source and type of data flowing over the network, one can achieve a number of very useful things that were previously impossible or very hard. For example, armed with the knowledge that a small portion of your user-base is consuming a large amount of your bandwidth, one could seek to restrict these users or a certain type of traffic such that the majority of users would continue to experience good service. Latency sensitive applications such as Voice over IP (VoIP) are affected particularly badly by high bandwidth consumption. One could also more easily identify large spikes in certain *types* of traffic (such as email) from a particular host on the network, which may be indicative of a compromised host. Without protocol-level classification, such spikes may simply be lost in the noise or incorrectly assumed to be something else (such as a large file download). There are of course countless other examples where this additional level of granularity would be pivotal in cutting costs, improving user experience, planning future network growth or just normal network monitoring.

It should be noted at this point that systems that perform these kinds of functions do already exist and have done so for some time. Vendors such as Cisco and Packeteer offer protocol-level traffic analysers and traffic shapers that can classify hundreds of different applications. However, these products are often well out of the price range of the smaller and medium ISPs, and completely unviable for individuals and businesses simply wanting to monitor their network activity more closely.

This project aims to provide a means for high speed protocol-level traffic classification, using only commodity computing hardware. By achieving this, tangible benefits could be realised by both direct users of the system (e.g. ISPs making cost savings) and indirect users of the system (e.g. customers of the ISPs or business, as their quality of service may be improved as a result of observed problems being rectified). This project by no means attempts to address a theoretical problem only; a solution to the problem would have numerous real-world applications, and this is perhaps the most appealing reason for undertaking this project.

Chapter 2

Literature Survey

Introduction

The literature survey shall analyse existing work that is pertinent to the design of the final product. Five broad subject areas (shown below) are described. Of course, further relevant subject areas may be discovered in the process of reviewing this preliminary material, and by the same token, some areas may also be narrowed in focus or even eliminated.

1. Traffic analysis and graphing
2. Packet capture and flow analysis
3. Large scale data storage
4. Legal and ethical issues relating to network monitoring
5. Supporting network infrastructure

Traffic analysis and graphing

Ever since the early days of IP networks, researches have spent considerable time and effort studying the flow of network traffic. Organisations, too, have invested huge resources in monitoring their network, particularly as it became more and more important to their ongoing operations.

After reading a number of papers on the broad subject of network traffic analysis, it seems that there are currently two main methods of achieving this (Barford et al, 2002). Each has its advantages and disadvantages, and is thus suited to different applications.

1. Polling network equipment
   Protocols such as SNMP (Simple Network Management Protocol) allow us to actively query a router or switch (or indeed many other network devices) for traffic statistics. However, as noted in (Estan et al, 2004), SNMP “counters” only give us the total amount of traffic transmitted – they provide no information about the type of data going over the network.

2. Packet capture and analysis
   By studying packet captures and knowing the protocols they adhere to, it is possible to construct packet flows for IP traffic, following data from source to destination. Flows, defined here as sequences of packets used in the

construction of sessions and data transfers, provide detailed information about how the network is being used, and ultimately how the network will be used in the future. Of course, this type of traffic analysis requires more resources and is also subject to certain privacy and legal implications, as will be seen later.

Whilst SNMP polling is a good solution for providing overall traffic statistics, it gives us no visibility on the data inside the network. Because of this, should a traffic spike or fall occur within the network it would be much harder to track the issue down to the application and host generating it, negating much of the usefulness of having the system in the first place. It will be seen later that performing live packet capture at 100Mbps is quite possible using modern commodity hardware, but the same is not yet true of 1Gbps. Despite this, the increased flexibility of the packet capture method makes it the most suitable candidate for use in the system.

Regardless of the data collection method used, graphs can be generated from any data with a time series attached. Of course, there are a great many packages that generate graphs already and since writing a graphing program is not the focus of this project, it is expected that an existing program will be used to generate traffic graphs. Whilst MRTG is popular in many papers, the authors of (Hiremagalur et al, 2005) chose to develop their own graphing system stating that "using MRTG, it is not easy too (sp) monitor and visualise traffic…". It should also be noted that MRTG is only suitable for SNMP based polling, and as noted above, this is too inflexible for the desired purposes. A variety of other graphing packages designed specifically for graphing network traffic exist, such as RTG (Beverly, 2006) and RRDTool (Oetiker, 2006). RRDTool is by far the most commonly used graphing tool amongst the papers studied, and it was most notably used to generate the graphs for the paper on "The Impact of Residential Broadband Traffic" which graphed data at up to 250Gbps (Fukuda et al, 2005).

Packet capture

One of the earliest examples of IP network monitoring and analysis can be found in a paper published in 1989 by (Caceres, 1989). The authors of this paper recorded traffic through their WAN link for 1 day and analysed not just the volume of traffic passing through the link, but also the *type* of traffic. The authors then continue to note some interesting statistics about the contents of the network traffic, including the fact that SMTP (Email) was by far the most prevalent protocol in use at the time (HTTP traffic was not to appear until around 1990).

Whilst these findings are interesting from a historical point of view, much more can be learned from their work. For example, the data set they operated on consisted of one day's worth of complete network traffic – circa 500,000 packets. Today, some 17 years later, some of the papers studied here discuss lossless capture of 500,000 packets per second using commodity hardware (Deri, 2002).

However, whilst the amount of data to be analysed may have increased, the same basic techniques are still in use today (partly owing to the fact that TCP/IP and UDP/IP are the primary communication protocols for Internet traffic, just as in 1989). The authors of Caceres (1989) wrote their own kernel level packet capture filter for the VAX hardware. Nowadays a portable C library known as libpcap (LBL Network Research Group, 2006) is available that interfaces directly with the network drivers to produce raw packet captures. This library, originally developed by the Network Research Group at the Lawrence Berkeley National Laboratory in 1994, relies upon a kernel level filter copying all packet data to special buffers, which libpcap then presents to the programmers via a small API. Many well known applications use libpcap at their core, including Snort, Wireshark (formerly Ethereal), nmap and tcpreplay (LBL Network Research Group, 2006).

It should be noted that libpcap operates at a very low level and its acceptance has been so universal that there are no alternatives with anywhere near the same level of user take-up. This is not to say that attempts have not been made though. Fisk et al (2002) tease us by suggesting a possible improved replacement for libpcap, called SMACQ. However, further reading reveals that SMACQ is also built on top of libcap and therefore is wholly dependant on it. However, this does not detract from their achievement. SMACQ provides users with a structured query language which they can use to build queries to perform on network data (either captured live, or post-processed from an existing pcap format file). Much like a real database, their system allows users to perform relational queries on network traffic. Whilst this is very useful in a wide range of applications, sadly it does not support inspection of the packets payload, which, as will be seen later, is required to track applications that randomise their port activity.

NetFlow (Cisco Systems, NetFlow, 2006) is another method of performing traffic analysis across busy networks. Originally a proprietary protocol defined by Cisco, it has since been opened up and accepted by the majority of network vendors, albeit under different names such as cflowd (Juniper Networks) and NetStream (Huawei Technology). When enabled on a router, NetFlow will capture traffic streams and send back information relating to the stream to a flow collector. There are many different versions of the NetFlow protocol, but the most recent is the NetFlow record version 9 A typical version 9 flow record contains the following information (Cisco Systems, NetFlow, 2006):

- Version number
- Sequence number
- Input and output SNMP indices
- Timestamps for the flow start and finish time
- Number of bytes and packets in the flow
- Layer 3 headers:
    - Source and destination IP addresses
    - Source and destination port numbers
    - IP protocol
    - Type of Service value

Nowadays, NetFlow is primarily used for traffic accounting in large hosting providers as it provides a means for many busy routers and switches to report approximate bandwidth usage to one central accounting server. The term ‘approximate’ here is important though, as Estan et al highlight in their paper “Building a Better NetFlow” (Estan et al, 2004). NetFlow is designed to be used in a sampled manner – that is, only 1 flow in every 10 may be recorded and sent to the collector. The reason behind this is that on high traffic routers, monitoring live TCP flows from potentially thousands of concurrent sessions would consume too much memory and CPU power, and could potentially harm routing performance. As detailed in Estan et al (2004), this issue is only exacerbated during a DoS (Denial of Service) attack. Of course, by not monitoring every flow, data is being lost that may have otherwise revealed important information. Estan et al (2004) provide a basis for creating an improved NetFlow engine that attempts to solve the problems of incorrect sampling rates by creating a self-adaptive sampling rate and also attempts to provide accurate flow count statistics.

However, neither NetFlow nor ANF (Adaptive NetFlow, presented by Estan et al) are suitable for the purposes of this project. As discussed earlier, NetFlow is only sampled. Whilst this is suitable for estimating traffic for billing information, it does not provide the level of granularity required in order to monitor spikes and inconsistencies – these would likely be masked by even a small amount of under-sampling. Secondly, NetFlow is typically only available on higher end routers and switches, none of which are readily available for use in this project. The biggest issue of them all though is that, like SMACQ, NetFlow does not allow payload inspection. Once again this means that it is not possible to track protocols that randomise their port numbers. As detailed later in this document in more depth, the Internet2 consortium (Shalunov, 2003, Pg 15) has exactly this issue with passive FTP.

Therefore, having excluded the possibility of collecting traffic data from the only two viable alternatives, SMACQ and NetFlow (excluding SNMP completely), the only remaining option is to pursue the low level and venerable libpcap (LBL Network Research Group, 2006). libpcap provides a low level C API that can be used to capture individual packets in their raw form. It does not provide any decoding functionality, nor does it provide and functionality for following flows or streams of data. In this sense it gives the programmer freedom to develop whatever higher level view on the data they like. Of course, this also requires more work.

One area surrounding libpcap that has attracted a lot of research in recent years is its performance. This is something that is crucial to this project too. Given that the majority of modern networks operate at 100Mbps, the ability to handle anything less than this would make the system useless for anything but the most trivial tasks.

Schneider and Wallerich (Schneider et al, 2005) presented a paper on packet capture at gigabit speeds. At full speed (1Gbps) they find major packet drops on both dual processor Xeon and Opteron systems running Linux and FreeBSD. However, it is easy to deduce from the graphs presented that no such problem is experienced at 100Mbps, or indeed anywhere before 450Mbps. Deri (2002) observes similar issues in his own tests and has implemented a solution that aims to resolve this issue. His solution creates a new type of socket, known as PF_RING, which requires a driver level modification to operate correctly. Used in conjunction with the NAPI present in recent Linux kernels, he is able to achieve 970Mbps when capturing 1500 byte packets. Whilst Schneider et al (2005) has shown us that such modifications are unnecessary for handling traffic at 100Mbps, it does provide an insight in to the extra work that would have to be conducted should the application need to operate at gigabit speeds and beyond.

Traffic categorisation

Simply being able to capture IP flows is only a small part of the traffic analysis problem. A big problem that is affecting even the largest organisations is traffic categorisation. Traditionally, it has been sufficient to take the destination port of the packet and look it up in a service mappings table (such as /etc/services in *UNIX* and Linux) to determine the application generating the traffic. This technique is known as 'Port Analysis'. However, many newer protocols are randomising their port activity and exploiting the properties of uPnP (UPnP Forum, 2006) to make these services visible to outside networks, even behind a NAT router and firewall.

Perhaps the most common example of this is the File Transfer Protocol (FTP). FTP can run in two modes – active (also known as PORT) and passive (also known as PASV). Active FTP uses port 21 for commands and 20 for data, with the server connecting back to the client (from port 20) to transfer data. Allowing arbitrary incoming connections is a security risk and is blocked by most modern firewalls. As a consequence, passive FTP was introduced. This still uses port 21 for commands, but now has the client connect to an unprivileged port (i.e. > 1024) on the server for data transfer. The server tells the client which port it should connect back to it on for data transfer in the response to the "PASV" command. Of course, an outgoing connection from the client will also originate from an unprivileged port, so we have communications between client and server on two unprivileged ports.

This presents a problem for traditional traffic analysers that look only at ports 20 and 21, as they will only capture the very minimal command data (as port 20 will be unused). Indeed, even the Internet2 consortium is unable to categorise passive FTP traffic due to exactly this issue (Shalunov, 2003, page 15). Of course, if this issue was restricted to Passive FTP alone then it would not be such a problem, but newer peer-to-peer protocols in particular, such as BitTorrent, use similar data transfer setups – and these are now accounting for a very large proportion of overall Internet traffic (CacheLogic Research, 2006).

One viable alternative to port based analysis is signature analysis. This involves understanding the protocol of the application in question, and then writing a regular expression (or some other form of match) that could be applied to the data within the packet and return true if it matched that applications signature, and false otherwise. Several have attempted to implement such a solution, some successfully (Related projects to libpcap, LBL Network Research Group, 2006), but none have yet made it to readily available general purpose traffic analysers.

It is important to realise that this approach involves delving in to the packet payload (the data), which will in turn incur more processing overhead per packet. Furthermore, the method of detection will vary drastically between protocols (e.g. the signatures for BitTorrent and passive FTP are very different). However, it may be possible to optimise the process for certain protocols by being stateful. In passive FTP, for example, the communication flow is of the form:

```
[… FTP authentication …]
> PASV
< Entering Passive Mode (192,168,150,90,195,149).
```

Here the client is requesting a passive mode data transfer, and the server is telling the client that he should connect back to 192.168.150.90 on port 50069 (195*256+149). By merely monitoring the command port (21) and watching for lines of the form “Entering Passive Mode (.*)” the data port that will be used for the data transfer session can be determined. Theoretically, this could then be stored in some state table and have a flag attached to it stating that traffic going to and from 192.168.150.90 on port 50069 is in fact FTP data. This would provide a constant-time lookup for FTP traffic, but it does require that the capture application maintain a state.

Maintaining state for a potentially large amount of traffic flows will require significantly more resources (both CPU and memory) and will ultimately result in the system being able to handle a lower throughput. Consideration must also be given to when it can safely be said that a flow has terminated, and how often to send back data about that flow. After all, a flow may continue for many hours and it is likely that the user would want to see some data about that flow before it finishes (that is, whilst the network operator could do something about it).

The research in to protocols above has shown that traffic identification is not always a trivial problem. For this reason, it makes sense to implement traffic identification modularly. When a packet is detected it could first be passed through known services, and if no match was found then it could be passed on to the more complex modules for identification (by signature, or the method described above). By doing this, support for future protocols may be incorporated into the system as and when they emerge.

Legality and Ethics

Monitoring traffic on a live production network raises numerous legal, privacy and ethical concerns. To begin with the structure of a typical TCP/IP packet is considered (Cisco Systems, IP, 2006). This contains a large number of fields including, but not limited to, source and destination addresses, source and destination port numbers, a checksum, and of course the packet payload (data) itself. Peuhkuri (2001) highlights that there are only two header fields that contain sensitive data – the source and destination addresses. On a LAN, it is often a trivial matter to trace a workstation and in turn a person using this information. It is much harder over the Internet, but it will at least drastically narrow the search space if this information were available. On the premise that live network data is “one of the basic tools” of network research, Peuhkuri (2001) sets about to devise a method of securely encrypting the source and destination addresses so that researchers may continue to investigate the data flows without compromising an individual’s privacy. Without such a level of protection, anyone viewing the headers could easily determine that computer A was communicating with computer B, and therefore it could be inferred that there exists some relationship between the two.

Peuhkuri (2001) also notes that the packet payload may be sensitive too, but does not consider encrypting or masking this data in any way (as that would defeat the purpose of using it for research). Indeed, it is this portion of the packet that contains the raw data – namely web traffic, email, authentication data (both encrypted and clear-text), and so on. One could argue that protecting the packet payload is even more important than protecting the header information, as the information it may contain is almost limitless.

Hussain et al (2005) also annonymises their packet traces, this time using tcpdriv (Minshall, 2005). Users with sufficient access are able to see the data in its raw form, whereas others will only see the annonymised information. Users subject to such restrictions will have the payload hidden from them as well.

All of the above raises a number of concerns about the proposed project, which will almost certainly involve capturing raw packets from live production networks. However, it is important to remember that the proposed system will be a closed one. That is to say that data from the proposed system is designed only to be used within the confines of the organisation operating the system. Given that this is also aimed at networking staff within the organisation, who will not only have had to sign confidentiality agreements already, but these are exactly the people who need to see this data – hiding the source and destination information will only hinder their use of the system. Furthermore, it is very common nowadays for ISPs (who would be a major target user of the system) to state in their terms and conditions that they reserve the right to monitor users network traffic in order to maintain and improve the network.

Capturing the payload of the packets is a different story though. Networking staff would not normally have a requirement to see such data, unless they were debugging a specific issue, in which case more traditional packet capture systems could be deployed. There is certainly no need for this system to store packet payloads either, and to do so would require vast amounts of storage and, more importantly, effectively double network traffic as every packet would be duplicated on to the storage system.

This is not to say that packet payloads need not be examined though. In the packet capture section above it was discovered that certain protocols that use port randomisation require us to inspect the packet payloads to determine which ports will be used next. This technique needs only to be applied to certain packets though and any inspection is performed in-memory and then the packet payload would be discarded immediately. None of the payload (apart from the next port, if indeed there is one), need be stored permanently.

For the above reasons it is not intended that packet encryption or masking techniques will be implemented. This will instead be left as future work, and the system will be designed with this in mind, such that it can be easily bolted on at a future date if necessary. Of course, there will still be security on the front end, but those who have access will have full visibility of the entire system. Should networking staff ever need to export data for others to see, they could easily employ tools such as tcpdpriv (Minshall, 2005) to mask sensitive information.

Data storage

Data storage in traffic analysis systems is always an important issue. Hussain et al (2005) notes that 64-byte packets at gigabit speeds sustained for one hour would generate 46GB of data. Of course, the target of 100Mbps is only a tenth of this, but even so it is not realistically possible store 4.6GB of data every hour. Furthermore, and unlike Hussain et al (2005) who need to store all the data in its entirety, the intention here is to only analyse and store traffic levels at regular time intervals - not every single raw packet.

ntop (Pras, 2000) is a popular Linux-based traffic analyser that employs the use of libpcap. Modifications have been developed that enable it to use a MySQL database to store persistent summary traffic statistics. This data can then be used for reporting on, but it is important to note that this is all it is used for. Furthermore, the data is only stored per host, and not per host and protocol (although the graphs in ntop do allow for this). This makes analysis of host and protocol data considerably harder, and makes the exporting of such data almost impossible.

Cacti (Cacti Group, 2006) is another popular traffic analyser, although this employs SNMP as its data gathering technique and is therefore limited to traffic totals only (it cannot drill down to protocol level). Instead of a full RDBMS (Relational Database Management System) it employs the use of RRDTool's RRD (Round-Robin-Database) files. These store data at a specified resolution (e.g. 5 minutes) for a pre-defined time period (e.g. 2 days). Once this period has expired, the old data is removed and replaced by fresh data. In this sense the data storage is cyclic, or "round-robin". The result is that the size of the database files and thus the access times are always constant, which is a useful thing to know when predicting growth.

The data stored within an RRD file can then be represented visually using RRDTool, which allows one to generate graphs, or exported as raw numbers using RRDFetch or the C API. However, even the briefest examination of the core requirements for the proposed system will reveal that there is the need to amalgamate traffic information from many hosts/networks and potentially analyse them simultaneously to produce the most interesting results (Just looking at one hosts data will only ever tell you about that host. But by looking at a host relative to all other hosts, statements can be made about the activity of those hosts and the overall state of the network). Since RRDTool lacks the necessary relational functions required to achieve such a task, it does limit its use within the system to that of storing graphing data for display purposes only (that is to say one cannot expect it to perform any further manipulation on this data once it was stored in RRD format).

With this in mind, a method for storing data permanently and performing relational queries upon it is still required. Hiremagalur et al (2005) seem to have encountered precisely this issue. Within their developed solution they chose to make use of the open source database MySQL. They claim that it is a very fast, robust and scalable database, well suited to the purposes of storing and querying large volumes of data. Whilst it may have lacked some features commonly found in other RDBMSs until recently, it is undeniably a very popular choice amongst developers nowadays. Personal experience of the database has shown it to be able to handle databases in excess of 4GB in size, with over 25,000,000 records spread across multiple tables.

One important factor to consider when choosing a database is its accessibility. Whilst RRDTool looked very promising initially, the lack of data accessibility prompted a search elsewhere for permanent data storage. By the same token, there is the need to be able present this data somehow, so it is important that there are suitable APIs available for our chosen programming languages. MySQL has a very complete and robust API available for most modern-day programming languages, but most importantly to us it has an API for both C and PHP (MySQL AB, 2006; PHP Group, 2006).

It should also be noted that MySQL databases are supported directly by the RTG graphing application. Should it later be decided that generating graphing data in RRDs and then graphing them with RRDTool is not the best option, it would be easy to switch to generating graphs directly from the database and displaying them using RTG.

Architecture

A problem highlighted earlier with NetFlow centred on the fact it required the router or switch to engage in data processing in order to report the flows. Such processes consume valuable CPU and memory that could be better served handling the routing and switching for its clients. Under a heavy DoS (Denial of Service) attack, NetFlow may cause more harm than good as it busily attempts to record all of the flows occurring over the network (including the attacking flows), thus depriving real users of the system the CPU and memory required to route their packets.

A solution to this problem is to use port mirroring, as suggested and used in Dreger et al (2006). Port mirroring, also known as “spanning” in Cisco terminology, operates at the router/switch level and copies all of the traffic on one interface to another. Thus, if a client machine was connected to port 1 on a switch, the switch could be instructed to mirror all traffic from port 1 to port 2, and then attach a sniffer to port 2 to record all of the client’s traffic. There are a number of benefits to using this technique:

1. It is entirely passive. The sniffer will not interfere with the client’s live traffic, as it is only receiving a copy on an entirely different switch port.
2. It uses less resources than NetFlow (but still does use some)
3. It allows us to selectively mirror only certain ports, rather than all of the router/switches traffic
4. It allows us to selectively mirror the direction of the traffic (inbound / outbound / both)

Figure 1, below, depicts how this is commonly deployed (Network Instruments, 2004).

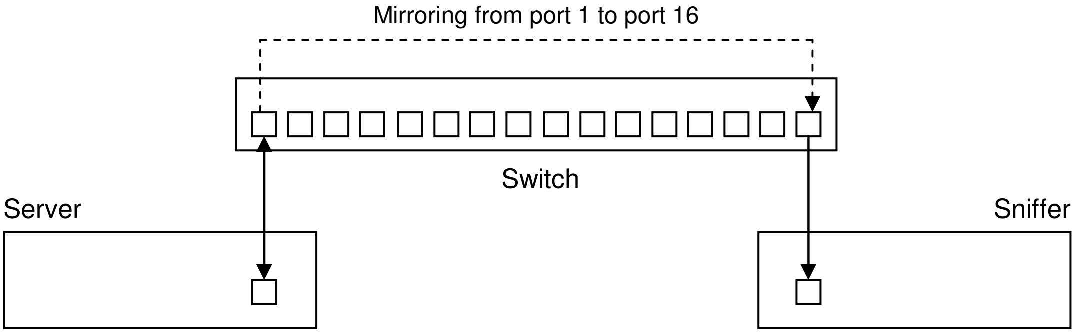


Figure 1 - Sample port mirroring infrastructure diagram

In Figure 1 above, the switch is simply mirroring all traffic from one server connected on port 1 to a sniffer connected on port 16. Of course, if the switch were to instead mirror all of the traffic from its uplink port to its router (as most switches will have, as it allows them to connect to remote networks), it would then be possible to monitor all inbound and outbound traffic from the switch, but not traffic local to the switch.

However, as Network Instruments (2004) notes in depth, port mirroring is not perfect. The switch or router performing the mirror must execute the copy operation for each packet to the capture port. Whilst this is faster than maintaining state information as NetFlow must do, it nonetheless still consumes CPU and memory, and at very high speeds (over 1Gbps) this can detract from users' performance. Network Instruments (2004) also discusses the buffer overflows that may occur when mirroring multiple ports on to a single capture port. Clearly this situation is unavoidable in certain circumstances. For example, if we are working with a 100Mbps switch and are mirroring ports 1 and 2 to port 16, and there is 100Mbps of traffic flowing through ports 1 and 2, then we wish to copy 200Mbps worth of data to port 16. However, this is clearly not possible as port 16 is only capable 100Mbps transfer.

The only solution to guarantee complete packet capture on a port is to use a tap (Network Instruments, 2004). This, as the name suggests, taps directly in to the line and passively copies every byte travelling over the link. The concept of a tap is represented in Figure 2, below.

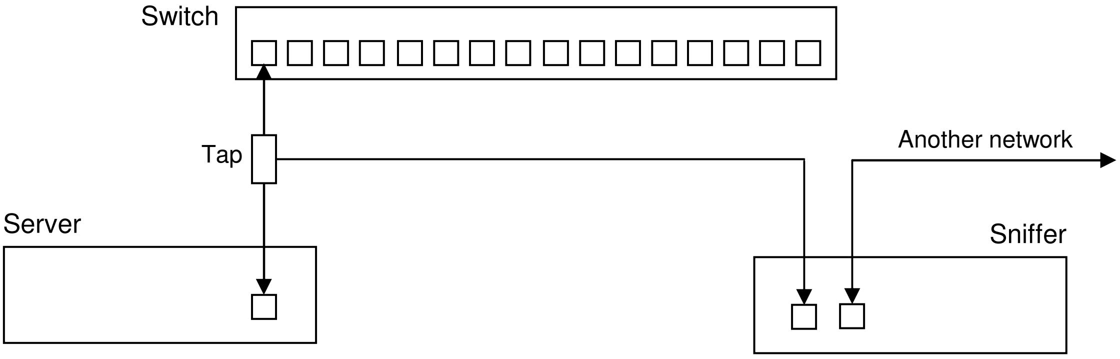


Figure 2 - Packet capture using network taps

Whilst a tap would clearly result in a better overall solution for this project, it will not be used within the project for the following reasons:

1. It is feasible that the application may need to mirror multiple ports simultaneously. A single tap is only able to monitor a single port, whereas a port mirror can mirror an arbitrary number of ports (subject to buffer overflow restrictions discussed earlier).
2. The volume of traffic the proposed system is designed to handle (circa 100Mbps) should not present a problem for most common routers and switches to copy. The CPU and memory issues described in (Network Instruments, 2004) are pertinent to backbone routers and switches handling 1Gbps above only.

3. The cost of a single Ethernet tap is prohibitive to this project (in excess of £200), and in order to demonstrate its usefulness effectively multiple networks would have to be monitored, thus requiring the use of multiple taps.

Conclusion

Within this document, an explanation of the different network traffic analysis techniques in use today have been presented, as well as the advantages and disadvantages of each. The SNMP protocol has been discussed, which is well supported and fast, but provides no visibility on the types of traffic. The primary alternative, packet capture and analysis, has been discussed in depth and its advantages and disadvantages weighed-up. Whilst these are too numerous to summarise here, it has been noted that packet capture and analysis can provide far greater visibility on the traffic flowing over the network and the way in which this traffic flows, but at the cost of increased infrastructure and computational overhead – particularly in the area of traffic classification. This is a problem unto itself, particularly when one considers protocols that randomise their port usage; requiring one to maintain state, which itself is subject to a series of problems.

Given that the system's usefulness depends on its ability to present as much detail about the data flowing over the network as possible, it seems clear that packet capture and traffic classification techniques will have to be used to obtain this data.

The advantages and disadvantages of a number of different methods for storing and displaying the data the system collects have also been examined. Other applications have also been explored, such as MRTG (which generates static graphs from SNMP polls), RTG (which generates dynamic graphs from MySQL databases) and RRDTool (which generates graphs from a proprietary 'Round-Robin-Database' file format). An almost foregone conclusion is that a more complete database will also been required to support the inherent relationships between network devices and the vast amount of data that the system will collect. MySQL has been briefly examined, along with its use within other systems requiring large database support – all of which have been satisfied with its performance and some have even switched to it from other database products.

Clearly some form of database is required to store data, and for the moment at least, MySQL appears to be the logical choice due to its demonstrable performance and well supported API within a number of programming languages. Graphing is somewhat less clean-cut, and it is quite feasible that different graphing systems will have to be used to represent different things (for example, live and summarised data). The performance hit of generating graphs in real-time will be the deciding factor here, but this is not something that can truly be known until implementation.

Privacy and legal concerns have also been considered, and a number of solutions have been examined that researchers have employed when analysing traffic from live production networks with real user traffic flowing over them. A conclusion was drawn at that point that the issues discussed in the papers examined were likely to be non-issues for the expected user base – networking professionals who routinely had access to user traffic anyway. For this reason, coupled with the additional overhead of developing encryption for the databases, this area will receive no further attention and could instead be targeted as future work.

Finally, the infrastructure that would be required to support the system was touched upon, along with the advantages and disadvantages that using alternatives would bring. Indeed, it was found that the raw packet capture method is imperfect and can result in missed packets when under heavy load. Using network taps is the only method guaranteed to preserve complete packet capture, but the costs involved in obtaining these make them unviable for the project. In the packet capture scenario the switches and routers require some additional functionality (such as port mirroring) that would not normally be found in commodity home devices. However, this functionality is available in low-end business hardware and certainly for any ISP of a non-trivial size. This hardware is also available for the purposes of this project too.

Whilst recognising these imperfections, packet capture in conjunction with port mirroring on a switch will still be used in order to develop and test the solution. For the targeted traffic volumes (circa 100Mbps per link), it is not envisaged that the potential traffic loss will be anything other than a minor problem. Furthermore, the ability to develop and test the system on commodity PCs using readily-available business network switches should also reduce development time and increase the overall quality of the end product.

In some preliminary discussions held with two London based Internet service providers, great interest has been expressed in the project and its ability to monitor disparate network traffic right down to the application level. Whilst these discussions are only at the preliminary stages, both ISPs have made initial offers to host a test solution on their networks, which would be extremely useful for testing purposes – both technically and from a usability standpoint too, as they would be keen to try the application themselves.

In conclusion, it has been shown that the proposed project is feasible by adapting current tools and techniques, and it is useful too. The research presented in this document has also refined the projects goals and focussed the scope more intensely on the problem of packet classification. It is now known that packet capture techniques and a modified signature-based categorisation algorithm will be required to adequately gather the data even before the analysis phase. An appreciation of the data storage that will be required has also been garnered, along with the knowledge that all of this can be performed on commodity hardware providing the traffic is not too high. Finally, research in to infrastructure has shown that whilst everything required to adequately develop and test the proposed solution is available, there is potential room for improvement (e.g. network taps, faster switches). Even at this stage it has been recognised that there are performance gains to be made by using specialised hardware, and potential regulatory gains from encrypting the data stored and displayed in the front-end. At this stage though these are left as future areas for work, as neither relates directly to the core focus of the project and would also significantly increase cost and time required to develop the system.

Chapter 3

Requirements

There are two core goals of the project:

1. To be able to correctly classify network traffic that was previously difficult to classify or unclassifiable using traditional means
2. To be able to report and alert upon this data using a user-friendly interface

It should be noted at this point that the design and completeness of the user interface is secondary in importance to the process of classifying the traffic. This problem has been the focus of the Literature Survey, and numerous problems and concerns surrounding this process have been identified. By contrast, only a relatively small amount of research was carried out in to the interfaces of existing network monitoring systems.

The requirements process will begin with requirements elicitation and analysis. The majority of requirements will be derived from precisely the issues identified with existing systems, although a number will also be taken from analysing questionnaire responses and consultation with Fluidata Limited - a London-based business ISP and potential real-life user of the system.

Based upon the requirements elicitation and analysis, a complete requirements specification will be formed, consisting of both functional and non-functional requirements, as well as any optional requirements that are not core to the system but would certainly add value.

Finally, the requirements validation will then seek to ensure that the requirements specification satisfies the original problem, and that all requirements are complete, consistent, verifiable and realistic.

Requirements elicitation and analysis

Methodology

The requirements elicitation phase of development will focus on three primary sources of requirements.

Firstly, the abstract and Literature Survey have identified the core goals of the project and from these alone a number of requirements can be derived. The Literature Survey has also explored how some existing systems have tackled the problem of application detection in network traffic, and by examining the inherent flaws exhibited by other techniques, further requirements and constraints upon how it should be constructed can also be ascertained.

Secondly, a short questionnaire has been used to back up requirements gathered already and expose new ones. This has been sent to three companies – Fluidata Limited, Xifos Limited and Brightstar Associates Limited. Fluidata is a London based ISP focussed upon providing business connectivity and hosting, complete with service level agreements and a high degree of reliability and redundancy. Brightstar is a relatively new London-based ISP providing business connections along with bespoke solutions. Xifos Limited is a Reading-based ISP that caters more towards the home market, and has an entirely different set of user demographics to both Brightstar and Fluidata. This fairly broad spread of ISPs should help identify requirements that are of interest to all sectors of the ISP market. The original questionnaire issued to the ISPs can be found in Appendix A, and their responses in Appendix B.

Finally, one of the afore-mentioned ISPs – Fluidata Limited – has kindly agreed to help in more depth with the project. The original project proposal was sent to them and they have responded with detailed feedback (which can be found in Appendix C) that further bolsters the demand for certain requirements. Furthermore, they have also offered to host the developing system on their network, and provide a live network to test upon. The legal issues inherent in such an implementation are a moot point in this case, due to a clause in the contracts they hold with their customers (Appendix C, page 2). Their continuing feedback on the developing system will also be extremely useful in later phases, as new requirements emerge and old ones change in priority.

Hardware and software considerations

The Literature Survey identified that port based traffic classification produces poor results, particularly since the advent of protocols without their own standard ports (e.g. VoIP, passive FTP) and those that purposefully randomise their port activity in an attempt to avoid classification (e.g. peer-to-peer protocols). The inability of commodity traffic analysers to correctly classify such traffic has led to the core requirement of this project. Whilst there are a handful of enterprise traffic analysers from the likes of Cisco, Naurus and CacheLogic, these are all well out of reach of small to medium businesses and ISPs, who are the target users of the application.

From the above a number of loose requirements can be inferred. Firstly, it is stated that the application must run on commodity hardware. "Commodity" in this instance is defined as commonplace x86 based machines produced within the last three years. It should be noted that the system itself will be developed on a dual 1.4Ghz Pentium III machine that is approximately four years old, so if it performs well enough on this, then it can clearly only do better on more modern (and faster) hardware. The aforementioned enterprise traffic analysers typically operate on ASIC (Application Specific Integrated Circuit) hardware, which is traditionally very expensive to produce.

The research conducted in the Literature Survey has also shown that packet capture (core to the ability of traffic analysis) is easily achievable at 100Mbps, but is much harder at 1000Mbps (gigabit). Indeed, Schneider et al (2005) have shown that even on modern x86 servers a significant percentage of packets will be dropped – particularly when the packet sizes are smaller. Graham (2006) identifies specific DAG network cards that utilise hardware acceleration to truly capture at wire-speed gigabit (full speed, full duplex), but sadly these are prohibitively expensive for the purposes of this project. Given that being able to capture packets is only half the problem (with analysis being the other half), it appears that wire-speed gigabit traffic analysis is still some way off. For the above reasons, it will only be required that each capture machine be able to capture and analyse traffic in real-time at 100Mbps.

Distributed connectivity and traffic volumes

Staying with the process of traffic capture, the Literature Survey and questionnaire responses indicated that many of the target users have multiple WAN (Wide Area Network) links that they may wish to monitor. These may be based at a single site or spread over many – there is no formal network design process that limits them to any one specific method. For this reason it was noted that the system most likely needs to be distributed, with multiple capture/analysis servers and a single aggregation server. By employing a technique such as this, the infeasibility of gigabit speed analysis can also be partially overcome. This is essentially spreading the load, and then aggregating the results in to a single database and subsequent visible output.

The previous two observations are reinforced by again looking at the target audience for the application. Small to medium sized businesses will typically employ xDSL or leased lines, sitting somewhere between 2Mbps and 30Mbps per connection. This is substantially less than the maximum supported speed of 100Mbps cited earlier. ISPs have a considerably larger requirement (their connections to other ISPs and core Internet exchanges will typically be 100Mbps or 1000Mbps), and indeed Fluidata cited in their formal response that 100Mbps aggregated would likely be insufficient for their needs (See Appendix C). However, like most ISPs, Fluidata maintain more than one Internet connection and spread their traffic over these links to ensure redundancy and help reduce congestion during peak times.

The results from the questionnaires back up this claim. Each respondent operated multiple links, all at less than 50Mbps. Therefore the need for a distributed client / server system to handle large volumes of data from disparate networks can already be observed at this early stage. So for the time being at least, it appears that the limitation of 100Mbps per link will satisfy the target audience, although the application design should account for a future move to gigabit when it becomes viable.

Modularity and considerations for the future

An observation made in the Literature Survey highlighted the explosive growth of peer-to-peer traffic on the Internet in recent years. The massive increase in usage has been coupled with a clear increase in the number of protocols and applications facilitating such file-sharing. Furthermore, although VoIP has been in operation in parts of the world for many years now, it is only within the past 18-24 months that its usage has really exploded in the UK and US (at least partially fuelled by the boom in broadband availability and take-up). Not only does this increase in new protocols and usage push ISPs and businesses to monitor their traffic more closely, it also impresses upon us the fact that the Internet and its usage is constantly evolving, and new protocols come and go over time.

Indeed, it may be that next year BitTorrent (currently the most popular peer-to-peer application) is toppled by a new and as yet unheard of application. Such a new application protocol will likely require a completely new detection and classification mechanism. Therefore it makes sense to modularise the classification process, allowing future protocols to be added simply by installing a new module. To do otherwise would result in the system becoming obsolete rather quickly, or at the very least requiring recompiling and redevelopment each time a new protocol was discovered.

The project only aims to demonstrate that this kind of traffic classification is practical given the requirements and constraints. It is not aimed at categorising *every* protocol in use today, there are simply too many! For this reason it expected that modules for the most common Internet protocols will be developed as a part of the project, and the rest will be left as future work or development. These most common Internet protocols will include at least the following: HTTP, HTTPS, FTP, SMTP, POP3, IMAP, DNS, BitTorrent, VoIP (SIP) and SSH. Whilst all of these protocols run at the application layer (layer 7 in the OSI model), two of the questionnaire respondents also expressed an interest monitoring lower layer protocols, including ICMP, IPSec and L2TP. Again, this will be included providing there is sufficient time following the core development and testing.

Any traffic that cannot be classified should, of course, be marked as such. However, this alone does not tell us much, and should a large amount of “Unknown” or “Unclassified” traffic accumulate, we would still be none-the-wiser as to its true source. Fluidata observed precisely this when reviewing a prototype of the system. They have suggested that an additional feature be included that allows the user to drill down in to the “Unknown” traffic and see precisely the addresses, ports and transport layer protocols involved (See Appendix M for preliminary feedback from Fluidata). Ultimately, this should allow the user to determine what protocol(s) are causing all of this unknown traffic and enable them to install new classification modules to bring the traffic in to line with the rest of the system.

User interaction

As highlighted in the introduction, even the most advanced traffic analysis scheme is pointless without a means of visualising the results. Therefore a graphical interface to display the analysed data will also be created as a part of the project. However, some attention to styling will be sacrificed in order to allow additional effort to be placed upon the primary focus of the project.

Traditional graphical interfaces for network traffic monitors utilise a range of graphing tools to visually represent inbound and outbound traffic. The intention here is to continue to present such graphs, but to extend them by splitting the graphs data points down in to their individual application components. Additionally, a tabular summary of traffic volume is also employed by existing applications such as nTop. This is again something that should be carried over to the proposed interface.

Before moving to the next point, it is worth noting that the Internet, and indeed many local networks, are divided in to subnets. The University of Bath, for instance, owns an entire /16 CIDR block of IP addresses, but does not place all hosts on the same physical network in the same logical network. Instead, the network is subdivided in to small sub-networks (typically one or more per building in the case of the University of Bath) that create a logical separation of departments.

Given the above, and the existing requirement that the system be able to aggregate from multiple capture/analysis hosts, it seems only logical that the interface be able to segregate the monitored network in to sub-units, allowing for easier interpretation and thus problem identification. Furthermore, some network operators may desire a finer granularity than others, so it also makes sense to allow users of the system to define their own groupings of hosts. This claim is further backed up by the observation that all of the questionnaire respondents used multiple network links and would therefore likely want to monitor each one separately.

Four basic levels of viewing the network activity within the graphical interface can now be defined:

1. Totals – Graphs and tabular data are generated across all groups monitored in the network, accounting for all applications and all traffic both in and out.
2. Groups – Graphs and tabular data are generated, grouped by all of the hosts within each group.
3. Hosts – Graphs and tabular data and generated, grouped by application (both inbound and outbound traffic) for each host on the network.
4. Applications – Graphs and tabular data are generated per application, across the totals, each group and each individual host.

Cacti and MRTG, two popular traffic graphing tools studied in the Literature Survey and in use by the questionnaire respondents, both graph data at different resolutions over different intervals. By default, they both graph the current day’s data over 5 minute averages, and the current year’s data over 1 day averages. Naturally they include periods in between, ranging from weeks to months. The short interval graphs with the higher resolution will clearly highlight traffic spikes and anomalies, whilst the longer interval graphs allow users to see the growth or decline of traffic over extended periods. This is a valuable feature and the fact that tools that provide this are already in use by our target users suggests that this would certainly be missed by users should it not be included in the requirements.

However, given that the proposed system is intended for use within environments such as an ISP’s NOC (Network Operations Centre) where every minute counts (particularly when it comes to customers’ service level agreements), it would be desirable to have a greater resolution than 5 minutes. But this could lead to computational issues – in particular having to aggregate and graph over 100Mbps of analysed traffic every minute. Testing is clearly required to understand this issue better, but at this point it will suffice to say that a resolution of at least 5 minutes will be required, with a higher resolution possible on smaller networks.

Alerting

Having high resolution traffic monitoring is a very useful tool if you have people watching the system at least that resolution themselves. Of course, it is unrealistic to expect anyone to stare at a series of graphs for any period of time constantly on alert for anomalies. For example, one may have a system capable of per-minute traffic graphing, but unless people are watching the graphs at least once per minute, this is overkill. Thus, as the results of the questionnaires have shown, ISPs would find it useful to receive alerts when a certain type of traffic falls out of expected bounds. Whilst this is a slight deviation from the core goal of the project, it will demonstrate the system's potential usefulness in a real-world scenario.

These alerts would be triggered by some rules, as defined by users of the system. These rules could be composed of a large number of variables, leading to extremely complex rules and an overly complicated user interface. Because of this, and the fact that generating alerts is not a core goal of the system, these rules should be kept relatively simple, whilst still trying to maintain nearly all of their functionality. It is envisaged that rules would be composed of constraints defined by all or some of the following:

- IP address or ranges of IP addresses
- Application classification (e.g. HTTP, FTP, BitTorrent, or a combination)
- Traffic volume (either current or a historical average)
- Time of day

Fluidata's response to the project proposal went in to considerable depth suggesting an addition to the proposed alerting system. They desired the additional ability to be able to trend traffic data and thus automatically modify alerting rules according to observed patterns in network traffic. This would essentially be composed of two main parts;

(1) A learning process that monitored traffic levels and built a model of expected traffic levels not only for the current day, but also for the future.
(2) A second process that would modify the rules continuously in response to changes to the model from the first process.

As they highlighted in depth in their feedback, traffic levels evolve over time, but a clearly recognisable pattern can be seen from the graphs. By modelling this and learning from it, future traffic levels can be better predicted, allowing tolerances to be adjusted accordingly. Another respondent to the questionnaires also briefly discussed trending as a desirable feature, reasoning that it would help reduce the number of erroneous alerts generated by the system.

With two independent sources both citing this as a feature they would like to see included, there is clearly the necessary demand for it. However, the alerting aspect is not the focus of this project. For this reason alone this feature will be left as an optional requirement that will be fulfilled if time permits. Naturally, the design of the system will account for the fact that this requirement may need to be implemented in the future.

Respondents of the questionnaire also specified that they would like to be alerted via both emails and SMS messages (to mobile phones). Previous personal experience with monitoring systems has shown that sending emails is trivial and even sending SMS messages is now relatively easy, with the advent of APIs provided by the likes of Clickatel (Clickatel, 2007). Fluidata in their questionnaire response highlighted the fact that neither email nor SMS were guaranteed forms of communication, and thus could not be relied upon alone in a mission-critical environment. They went on suggest that an "automatic dial-plan" might be put in place that dialled a series of phone numbers until one responded, indicating that they were aware that an alert had been raised. Again, whilst this is a very interesting and useful feature, it will be left as an optional extra that will be completed providing time and resources are available. SMS and email alerts will be considered as a core requirement though.

Integration with existing systems

After reviewing an early prototype, Fluidata were keen to point out the potential benefits from integrating this new system with their existing customer facing web-portal. At present, users of their web-portal are presented with traffic graphs and historical bandwidth usage figures. By integrating with this system, detailed breakdowns of the traffic for individual customers could be generated, giving the end-user a far more detailed view of their traffic. Whilst this may be over-kill for the majority of users, Fluidata felt that a sufficient number of their users would find the addition a useful one.

The concept of grouping hosts came up again here, as Fluidata asked if it would be possible to present a graph for a customer's entire network (as there would often be multiple hosts, IP addresses and even WAN links associated with one customer). This reinforced the earlier idea that the ability to define arbitrary groups was an important one, and certainly warranted inclusion as a functional requirement.

Requirements specification

Functional requirements

The functional requirements are a specification explicitly stating what the system will and will not do.

The system will:

1. Monitor and analyse network traffic of at least the following basic network protocols: HTTP, HTTPS, FTP, DNS, IMAP, POP, SMTP, POPS, IMAPS, SSH, Telnet, RDP
2. Monitor and analyse network traffic of at least the following dynamic network protocols: Passive FTP, BitTorrent, SIP (VoIP)
3. Accurately classify at least 99% of traffic, where the traffic type should be classifiable according to the protocols supported
4. Classify and record any traffic that is unknown to us as being of type "Unknown"
5. Allow for classification of new protocols with the addition of traffic classification modules
6. Operate a client/server architecture, with:
   - 6.1. Capture and analysis machines acting as clients. These should capture and classify the traffic at each of the individual WAN (Wide Area Network) links and periodically report the results back to the server.
   - 6.2. Aggregators and reporting machines acting as the servers. These should receive periodic reports from the clients, and using this data should maintain a database of traffic and generate reports and graphs.
7. Be able to capture and analyse traffic at up to 100Mbps continuous data throughput per WAN link on commodity computing hardware
8. Support a data resolution of at least:
   - 8.1. Five minutes per day
   - 8.2. One hour per week
   - 8.3. Six hours per month
   - 8.4. One day per year
9. Provide a single web-based front-end that collates information from all capture sources
10. Generate traffic graphs of at least the following type:
    - 10.1. Accumulated total traffic across all groups of hosts, split by application, and inbound/outbound directions
    - 10.2. Accumulated traffic per group, across all hosts, split by application, and inbound/outbound directions
    - 10.3. Accumulated traffic per application, split by host and inbound/outbound directions
    - 10.4. Accumulated traffic per host, split by application
    - 10.5. Traffic per host per application, split by inbound/outbound directions
11. All graphs should be generated displaying at least the following levels:
    - 11.1. Daily
    - 11.2. Weekly
    - 11.3. Monthly (Going back for the period of at least one year)

12. All graphs should include at least the following in their display:
    12.1. A title, relevant to the data being displayed
    12.2. A Y-axis labelled with transfer rate in suitable units
    12.3. A X-axis labelled with time, increasing linearly
    12.4. All data series labelled suitably (i.e. as an application, group, IP address)
    12.5. Minimum, maximum, current, and average transfer rates per data series
13. Generate tabular data suitable for display relating to at least the following intervals:
    13.1. Current transfer rate inbound
    13.2. Current transfer rate outbound
    13.3. Hourly average transfer rate inbound
    13.4. Hourly average transfer rate outbound
    13.5. Six-hours average transfer rate inbound
    13.6. Six-hours average transfer rate outbound
14. Generate tabular data suitable for display relating to at least the following levels:
    14.1. Accumulated total traffic across all groups of hosts, split by application
    14.2. Accumulated traffic per group, across all hosts, split by application
    14.3. Accumulated traffic per application, split by host
    14.4. Accumulated traffic per host, split by application
15. Allow end users to define named groupings of hosts based upon:
    15.1. A single IP address
    15.2. A range of IP addresses
    15.3. A CIDR style range (e.g. 80.77.250.230/29)
16. Allow users to define automated alerts based upon:
    16.1. Traffic classified as a certain type of application or from a class of applications
    16.2. Traffic originating from or destined to a certain hosts
    16.3. Traffic inbound, or traffic outbound exceeding a certain transfer rate based upon
        16.3.1. Current transfer rates
        16.3.2. Past hours averaged transfer rates
        16.3.3. Past six hours averaged transfer rates
    16.4. Traffic inbound, or traffic outbound falling below a certain transfer rate based upon
        16.4.1. Current transfer rates
        16.4.2. Past hours averaged transfer rates
        16.4.3. Past six hours averaged transfer rates
    16.5. Traffic occurring a certain time of the day
    16.6. Any combination of the above (Requirements 16.1 to 16.5)
17. Allow users to specify recipients of alerts, giving each a:
    17.1. Name
    17.2. Email address or SMS mobile phone number
    17.3. Flag determining whether or not this contact is currently active
18. Be able to send alerts to contacts via at least:
    18.1. Email
    18.2. SMS
19. Employ a basic means of security, limiting access by username and password to the front end

20. Not interfere with any of the network traffic it monitors (i.e. It is passive)
21. Provide a detailed breakdown of any unclassified traffic, such that administrators may then be able to infer the true type and install new classification modules

Optional functional requirements

These functional requirements have been identified by the requirements elicitation process as being useful to the project and its intended users, but are not core to the goal of the project itself. For this reason they have been designated as optional and will be completed if time and resources permit.

The system will, should time and resources permit:

22. Trend traffic data to produce a model of model of the type and volume of traffic expected at different times of the day. The system will then permit users to define a variance margin by which traffic may deviate before alerts are sent out.
23. Monitor and graph non-application layer protocols such as ICMP, IPSec, L2TP and BGP.
24. Provide a method of guaranteed alerts, by means of automatically dialling a series of pre-defined phone numbers and waiting for a user's acknowledgement.

Non-functional requirements

The non-functional requirements specify those requirements that cannot be precisely measured by success/fail criteria – they are, in a sense, subjective.

The system will:

25. Operate effectively on commodity hardware. Namely, the system should not require specialised hardware (such as ASIC circuitry or advanced capture cards) to operate within the specified parameters
26. The front end should be easily navigable and respond fast enough such that it is usable
27. The system should be both stable and reliable
28. Future improvements to the system should be accounted for in the design such they it is easily upgradeable. These may include, but are not limited to:
    28.1. Support for gigabit capture and analysis per link, once the hardware and operating systems have matured to a suitable level
    28.2. Trending of network traffic and automatic rule adjustment based upon this learning process
    28.3. Inclusion of new alerting mechanisms, such as persistent calling / SMSs until acknowledgement
    28.4. Support for a different databases on the back-end
    28.5. Support for modelling traffic outside of the local network
    28.6. Support for integration with existing customer-facing systems to improve client visibility of network usage

Requirements validation

The process of requirements validation seeks to show that the requirements defined in the requirements specification solve the problem defined. One common way of going about this is to produce prototypes and have the intended users review these against the set of requirements. This is precisely the process that has been (and will continue to be) used here.

Shortly after drafting the first set of requirements, a semi-functioning mock-up of the system was produced. Screenshots of this early version can be found in Appendix C. Reviewing this with two of the three respondents of the questionnaires, a number of additional requirements came to light.

First and foremost they desired the ability to not only group hosts by the WAN connection they were operating over, but instead by their own custom criteria. For example, instead of having a group "WAN Connection Bournemouth", they wanted to be able to subdivide that down further. Ultimately this request was born out of the fact an ISP carries data for many clients down a single WAN link, and instead of monitoring them all together; they would quite understandably like to group them by client. Based upon this, functional requirement 15 was born.

Reviewers of the prototype also noted that the graphs were sometimes hard to interpret, particularly when representing large volumes of data. One suggestion was to limit the number of data points that could be plotted on the graph and introducing figures for minimum, maximum, average and current transfer rates per data point. These suggestions formed the basis of functional requirement 12.

One of the reviewers pointed out that a large percentage of their users used VPN traffic between multiple sites. Traffic such as this operates at below the application layer, and often operates on top of the IPSec protocol which sits at layer 3 in the OSI model. This further backed up support for the optional functional requirement 22. However, at this point in development there was no way to test IPSec traffic capture, so it could not definitively be included as a full requirement.

The remaining comments around the prototype surrounded the styling of the interface and the layout the data. These comments have been noted and will play a role in helping to shape the interface when it comes to designing and building the final version, but they will not form specific requirements.

Chapter 4

Design and implementation

This section focuses upon the design and implementation of the proposed system, based upon the requirements outlined earlier. In keeping with the mantra of the evolutionary development model chosen, the design and implementation are being performed almost in tandem, with a design being drafted, partially implemented, redrafted, re-implemented, and so on. To reflect this methodology, the design and implementation will be discussed simultaneously throughout this section, with the exception of a high-level design overview being presented first.

Throughout the design references are made to the “Bath” and “Plymouth” networks. These are two small home networks that facilitated early development and testing of the system. Later development and testing occurred on Fluidata’s network, which is a real-world ISP network. In many cases changes in design were required when moving from the low traffic networks of Bath and Plymouth to the high traffic network of Fluidata. Each of these design changes is described in detail within the respective section. A more detailed description of these three networks, the computer systems monitoring them and the network architecture is available in the testing section.

High-level preliminary design

In keeping with one of the core ideas of both the Literature Survey and requirements, the system has been designed to be as modular as possible. Figure 3 presents a conceptual model of how the different components will interact.

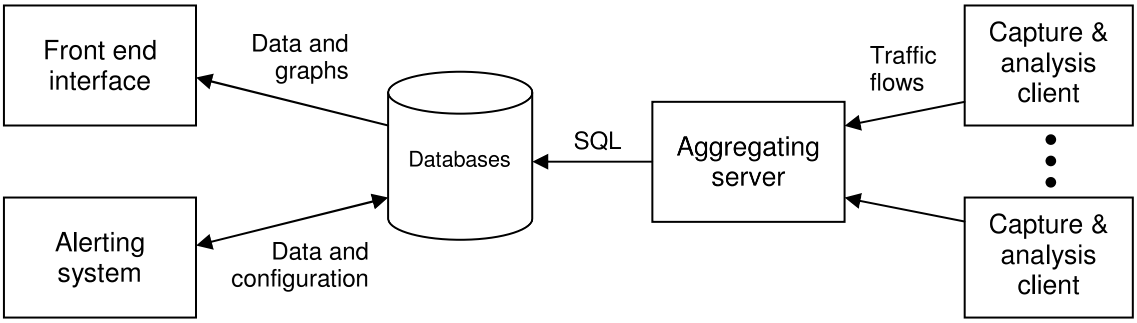


Figure 3 - High level system architecture diagram

The remainder of the high level design overview will focus on detailing the purpose of each of the components in the above diagram.

Capture and analysis client

The capture and analysis client is arguably the most complex part of the system. Passively intercepting all inbound and outbound WAN traffic on a network, its job is to classify each packet it sees and report its findings back to the aggregating server.

A principal idea of the client is to only report back a summary of the traffic it has discovered – not the entire volume of traffic itself (otherwise there would be little point in having distributed capture clients). This summary will incorporate the key ideas of the Cisco NetFlow protocol – namely, record traffic flows between source and destination and purge them (report them back) when they expire. The key difference between the method used in NetFlow and the proposed solution here is two fold:

(1) NetFlow sends only source address, destination address, source port, destination port, protocol, number of bytes and the start and end timestamps. The proposed solution would additionally send an application field, containing the results of the application classification. This classification would not be possible with the NetFlow information alone.
(2) NetFlow only expires a flow when it sees a TCP FIN (Finish) or RST (Reset) message. The proposed solution here would also expire flows at a certain time interval and also when the number of bytes exceeds a certain threshold. This would ensure that large traffic flows could not continue for hours without appearing on the front end.

Detecting the source application of an individual packet at first appears a daunting task. The research in the Literature Survey has shown that current systems, such as nTop, simply match the port number in use to the port numbers defined in the IANA (IANA, 2007) protocol assignment lists. As was shown in the Literature Survey, this often leads to inaccurate and incomplete results.

The proposed solution involves using application signature detection to delve in to the packet payload and attempt to find a match against known application signatures. If a match is found, then mark that particular packet as being of the matching application type. However, also mark any *future* packets involving the same source address and source port or the same destination address and destination port as being of that application type. This is the key idea that allows us to classify packets even if their contents are unrecognisable or encrypted – only a single recognisable setup packet is required to distinguish the stream. For example, consider the following TCP session:

Host A		**Host B**	**Packet Payload**
192.168.254.200:32582	>	80.77.246.42:41820	.BitTorrent Protocol…X..aF0x..
192.168.254.200:32582	<	80.77.246.42:41820	[Handshake response]
192.168.254.200:32582	>	80.77.246.42:41820	[Request piece 0x53]
192.168.254.200:32582	<	80.77.246.42:41820	[Binary piece data]
…			
192.168.254.200:32582	>	77.6.58.22:41820	[Binary data]

The first packet in the above session contains the key information we are looking for – the first 19 bytes are instantly recognisable as being the handshake message for the BitTorrent protocol. At this point we can record that 192.168.254.200 on port 32582 is handling BitTorrent and so is 80.77.246.42 on port 41820. The subsequent three packets contain no recognisable application signatures, but they are classified as BitTorrent by looking up "192.168.254.200:32582" and "80.77.246.42:41820" and finding that they have already been tagged with an application. The final packet is destined to an unrecognised host and port and contains completely unrecognisable data. But because it is already known that "192.168.254.200:32582" is carrying BitTorrent traffic, it can be inferred that every session using that host and port is also carrying BitTorrent traffic.

This inference process saves us from checking the payload of every single packet, thus reducing overhead. Naturally, some expiry time will also need to be associated with the tag that ties an application ID to a specific host and port. To allow it to exist indefinitely would result in erroneous data later on, as dynamically allocated ports (1024 to 65535) can be reused for other purposes once they are no longer in use.

Finally, an optimisation to this process would involve using a traditional port-based classification to find the application when the port was less than 1024. Ports below 1024 are known as "well known ports" and their use is regulated by the IANA (IANA, 2007). Applications started by normal users cannot bind to such ports - they must use ports greater than 1024.

The flow chart in Figure 4 details the proposed packet classification process.

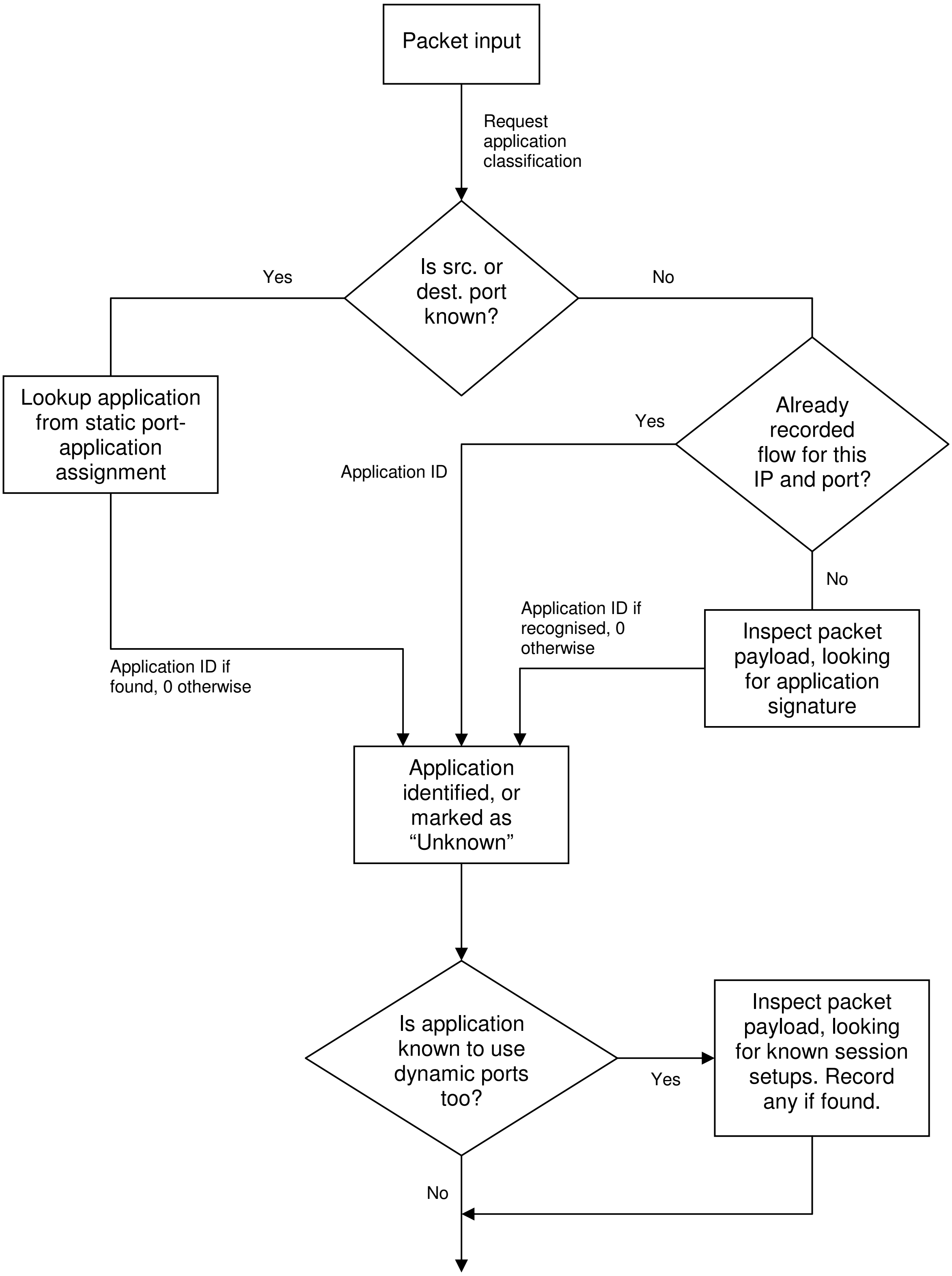

Packet input
Request application classification
Is src. or dest. port known?
Yes
No
Lookup application from static port-application assignment
Already recorded flow for this IP and port?
Yes
Application ID
No
Application ID if recognised, 0 otherwise
Inspect packet payload, looking for application signature
Application ID if found, 0 otherwise
Application identified, or marked as "Unknown"
Is application known to use dynamic ports too?
Yes
Inspect packet payload, looking for known session setups. Record any if found.
No

Figure 4 - Preliminary packet classification process

The requirements clearly stated that the packet classification process should be as modular as possible. The purpose of this is two fold:

(1) To allow classifications for new applications to be incorporated very easily (i.e. without having to recompile the application).

(2) To allow users to remove certain modules that they know, due to their knowledge of their network, will rarely or never be used. Thus they can save on computational overhead and potentially push the application to handle more traffic. For example, a business ISP is not likely to want to classify all online computer games traffic (at least not individually anyway), but a consumer ISP may wish to do exactly this.

Thus installing a module should be as simple as compiling it and dropping it in to a directory, and then, perhaps, enabling it within the application. This may simply involve entering a line in to a configuration file. Disabling a module would then just involve commenting out or deleting that line from the configuration file.

As depicted in the flow chart in Figure 4, any traffic that cannot be classified after all of the different processes will simply be marked as “Unknown”. It is important not to simply discard the traffic, as firstly it may represent a sizeable volume and secondly, it may be the only clue the user has to the emergence of a new and as yet unidentified protocol.

Aggregation server

The aggregation server performs a trivial task compared to the client. It simply waits for flow information on a specific port and, upon receiving said information, splits it up in to relevant chunks and inserts them as records in to a database.

The aggregation server should do no more and no less than this. The only consideration that is required here is that multiple clients will be posting data to the same server, so some form of receiving concurrency needs to be accounted for.

Databases (and associated processes)

The purpose of the databases is three fold:

(1) To store the flow information from the aggregation server temporarily (i.e. before processing)
(2) To store processed the information suitable for interpretation and display on the front end
(3) To store graphing information suitable to generate graphs from

It is intended that the first two of the above will be handled by a traditional database such as MySQL. This database was highlighted as both a reliable and strong performer in the Literature Survey. Flow records would be inserted in to a single large database table, and then periodically (most likely every minute) a process would collate these records and record new averages per host, per group, per application, and also in total. As specified in the requirements, these averages should be recorded for at least the current traffic volume, the past hours traffic volume, and an average of the past six hours traffic.

The third point (the storing of the graphing data) can be looked at slightly differently. Unlike data on the front end, the data in the graphs will be static – that is, once the image has been generated it cannot be manipulated (although it will likely be overwritten on the next generation cycle). Furthermore, the data used to generate the graphs is just simple time-series data, and no complex queries are required to acquire the data. Therefore, a full RDBMS (Relational Database Management System) is not required here, and a smaller and more efficient alternative would be more appropriate.

As highlighted in the Literature Survey, this is precisely the area that RRDTool targets. This application utilises RRDs (Round Robin Databases) that store time-series data at specific intervals for finite periods. For example, one might configure an RRD to store data for one day at five minute intervals. After each day, when the data has been stored, it would wrap around and begin storing data from the beginning again. This ensures a constant database size and predictable lookup time for queries.

In addition to being a useful method for storing time-series data, the RRDTool application itself uses these RRDs to generate highly customisable graphs. It is envisaged that this will be used to generate graphs every minute for all hosts on the network, although this could prove too computationally challenging on larger networks.

Front end

The front end is intended to be a relatively simple web-based system that allows users to browse the various statistics at a range of different levels. These levels, as defined in the requirements, are:

(1) In total (i.e. across all groups and applications)
(2) Per group of hosts
(3) Per application
(4) Per host

Whilst less attention is being paid to the front end than the back end, it is intended that it be simple to use, highly functional and fast. In keeping with the availability of other traffic monitoring systems and the current trend in application development, it is also intended that the front end be web-based. A number of standard practices in web-based design should also be observed:

(1) All pages should be consistent in style and appearance (commonly solved using a Cascading Style Sheet)
(2) All pages and sub-sections should be titled, and such headings should be easily distinguishable from body text
(3) All pages should be easily navigable, with clear links back to the previous and top levels

(4) All data tables should be orderable by their column headings

A preliminary mock-up of the appearance of the front end can be found in Appendix D, with an even earlier version visible in the prototypes used during requirements validation available in Appendix C.

Alerting system

The front end of the alerting system should allow users to configure rules that define when an alert should be triggered and who should be contacted. The back end should periodically attempt to match these rules against the current database, and if a match is found, send out the alerts. At the very highest level, the above defines the alerting part of the system. This has largely glossed over the most important points though. Each point will now be discussed in more depth.

As highlighted in the requirements elicitation, every minute a service is operating outside of its normal parameters, the ISP could be losing money, customers or both. Clearly the imperative is to get alerts out as quickly as possible. So, the most logical interval at which to match against the alerting rules is precisely the same interval at which the data is processed by the data processing scripts. To check less frequently may result in us missing traffic spikes, and to check more frequently would result in wasted checks and CPU cycles (as it would involve checking the same data multiple times, because it would not have changed). At present it is envisaged that data will be recorded and processed once per minute, so therefore this is the logical interval at which to check for alerts.

It can also be seen from the requirements surrounding the alerting rules (requirement 16.1 through 16.6) that these could become rather complex to specify if done on a single page. Therefore a multi-page "wizard" interface is envisaged, with each page defining a distinct part of the rule. Thus the following pages would exist:

(1) A page to define the hosts that the rule would encompass
(2) A page to define the application protocols that the rule would run for
(3) A page to define the transfer rate (at the time-series average) that the rule should trigger at (for example, one might say "60KB/s average over the past hour")
(4) A page to define the time of day that the rule should be active for
(5) A page to define who should be contacted when the alert is triggered
(6) A summary page that allows the user to confirm or edit each section, enter a name for the rule, and finally activate it

The usability of this envisaged interface remains to be seen, as it is possible that a user with many rules to define could find the interface rather slow. Furthermore, having to edit the rules to account for acceptable changes in network traffic would also consume considerable time. In such a scenario the trending of traffic data, as suggested by two of the questionnaire respondents, would be very useful.

It should also be noted that the contacts would only need to be defined once and could then be reused in many rules. Each contact would be specifiable with a mobile phone number or an email address, as well as a flag indicating whether or not that contact was active. A draft of the alerting interface can be seen in Appendix D.

Detailed design and implementation

The preliminary high level design presented in the previous section was used as a basis upon which to begin development of the application. As is often the case with software development projects, once development began a number of elements from the original design had to be modified. That said, the overall structure of the system (as presented previous in Figure 3) remains unchanged.

This section will now focus upon the evolution of the design from the preliminary one presented earlier to the final one, and then the implementation of this final design. As this is an iterative process, both the design and implementation will be discussed simultaneously throughout the following section.

Capture and analysis client

Packet capture

The purpose of the capture and analysis client is to capture and classify packets and report these classifications back to the aggregation server.

Speed really is of the essence in this part of the application, so it is important to choose a language that is proven to be a good performer in such scenarios. Support for a stable and well documented packet capture library is also a necessity. As stated in the Literature Survey, the single most common way of capturing packets is to use the libpcap library. This very fast and very stable library has been ported to a whole host of languages and operating systems since its inception, but C is still the development language of the main branch. Given this knowledge and the fact that C is a tried and tested high-performance language, the obvious choice of development language for the capture client is C. Whilst it may be lacking in terms of pre-defined data structures that other more modern languages may offer, it more than makes up for this here with its raw speed and flexibility.

It is also worth noting that C lacks a garbage collector, thus requiring the programmer to free any memory they use once it is no longer required. In order to help ensure that memory leaks do not occur, Hans Boehm's garbage collector (commonly known as "GC.h") will be used to handle all memory management (Boehm, 2007).

As indicated above, the libpcap C library will be used for packet capture. This allows the programmer to intercept *raw* packets on a specific network interface, using a very simple control loop presented in pseudo code in Figure 5.

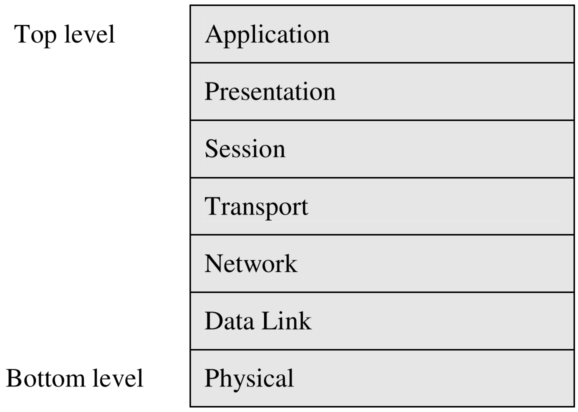
```
desc = pcap_open_live("eth0", ...)
while (packet = pcap_next(desc, header))
{
   // Work with packet
}
pcap_close(desc)
```

Figure 5- Simple libpcap control loop

Note here that there are a number of stipulations:

(1) A captured packet is presented to us as a blob of binary data. libpcap does not decode packets and provides no structures to facilitate this.
(2) Only a single interface can be captured from at any one time
(3) In order to capture packets from a given interface, the interface must first be placed in *promiscuous mode* (This allows the Ethernet driver to receive packets that were not destined for it). In order to do this root or administrative privileges will be required over the capturing machine.

The stipulation that root privileges are required to use the libpcap library is not of great consequence here – the machine running the application is likely to be specialised for this purpose and segregated from the rest of the network. However, the fact that libpcap provides no packet decoding means that the application will need to handle this process itself. In order to do this, the structure of packets needs to be understood – a process that is made relatively painless thanks to the seven-layer OSI model, as shown in Figure 6.



Top level

Application

Presentation

Session

Transport

Network

Data Link

Bottom level

Physical

Figure 6 - Seven layer OSI model

The packets presented to us by libpcap will typically include everything from the Data Link layer upwards. At this point it is worth looking back to the requirements and observing that all of the required protocols operate over TCP or UDP, which sit at the Transport layer. (Some of the protocols from the optional requirements do not fall under this criterion, but this will be dealt with later). For the moment, it is also observed that TCP and UDP both sit on top of IP, which in turn sits on top of the Ethernet protocol. (Although TCP and UDP can sit atop other protocols, IP and Ethernet are the only ones being considered for the moment).

Therefore, in order to be able to find the data that the applications at either end are sending one another, a mechanism for decoding TCP and UDP packet headers must be devised, which in turn requires us to be able to decode IP and Ethernet headers. The structure of Ethernet, IP, TCP and UDP headers can all be found in Appendix F, but for the purposes of continuity, summaries are shown below in Figures 7 and 8.

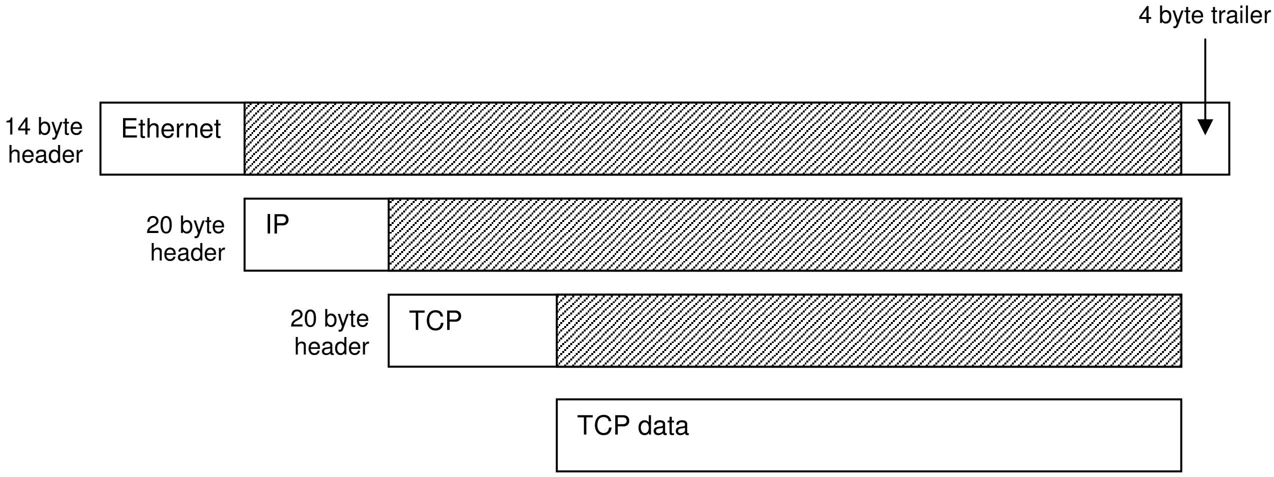


Figure 7 - TCP/IP packet format (Excluding options fields)

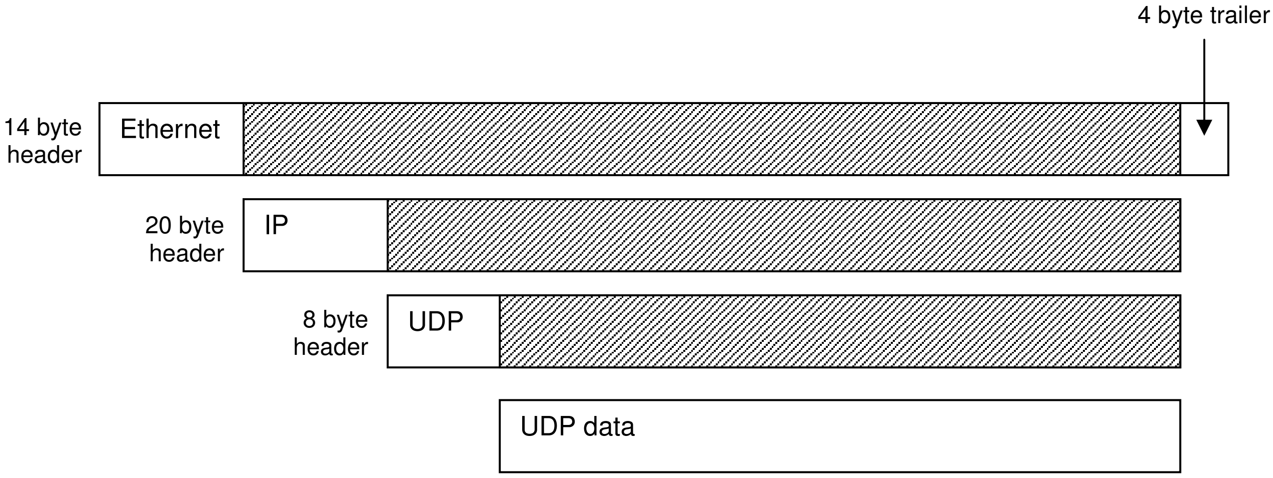


Figure 8 - UDP/IP packet format (Excluding option fields)

It is the knowledge of these structures that has facilitated the decoding of the raw packets into a more usable format. This format consists of a packet structure (defined as “struct packet” in the code), which is constructed according to Figure 9.

```
 struct packet {
        struct ethhdr *eh;
        union {
                struct iphdr *iph;
                void *raw;
        } nh;
        union {
                struct tcphdr *th;
                struct udphdr *uh;
                void *raw;
        } h;
        void *data;
        int data_len;
};
```

Figure 9 - Packet structure used in client application

Here it is noted that a packet *must* contain an “ethhdr” field – the Ethernet header. It may then contain either an “iphdr” field (the IP header) or a “raw” value. At the next level down the packet may contain either a “tcphdr”, “udphdr” or again a raw representation of the data. Finally, there is the actual data contained within the TCP or UDP packet (this is essentially just an offset from the start of the packet). The various other structures relied upon in this structure (ethhdr, iphdr, tcphdr, udphdr) are all defined by the host operating system as a part of the standard C library.

It should be noted that the intricacies of the packet decoding mechanism have been adapted from an example by Shankhar (2006), and tested thoroughly by comparing output against that of tcpdump (2006). To have an error at this level in the application would render all higher levels useless, so it has been necessary to ensure that packet decoding was absolutely correct before proceeding.

Changes required following implementation on a high traffic network

The packet decoding mechanism described above is relatively simplistic and does not account for protocols and additional layers that may be found on an ISP-class network. Indeed, whilst this method worked perfectly during testing on home networks in Bath and Plymouth, it failed to decode many packets from Fluidata’s network in London. Investigation in to the source of this issue soon revealed two problems that had not previously been considered.

Firstly, Fluidata’s network uses VLANs (Virtual LANs) throughout. VLANs allow network administrators to segment a single physical LAN in to logical sub-networks at the second layer of the OSI model, using a protocol known as IEEE 802.1Q (IEEE, 2003). The aim of this is to reduce broadcast traffic, reduce interference between logically diverse networks and ease manageability. It is important to note though that VLAN tagging simply adds four bytes to each packet – it does not modify any of the existing headers (Ethernet, IP, transport layer, etc). Therefore, for the purposes of this project, it is sufficient to simply read over this header and ignore the VLAN ID.

The VLAN header appears immediately after the Ethernet header in a packet, and its presence is signified by the *type* field in the Ethernet header being set to ETH_VLAN.

Should this be found, the packet decoding code simply reads over four bytes and continues as normal.

Secondly, it was later noted that some packets could still not be decoded successfully. Capturing a handful and then analysing them using tcpdump, it became apparent that they used a protocol at the transport layer known as GRE (Generic Routing Encapsulation). Unlike UDP or TCP, which are the normal candidates to be found at the transport layer, the payload of GRE packets are IP packets themselves. This protocol, developed primarily by Cisco, was designed as a simple way for tunnelling IP packets over a *virtual* point-to-point link (RFC2784, Network Working Group, 2000). A typical GRE packet is illustrated below in Figure 10.

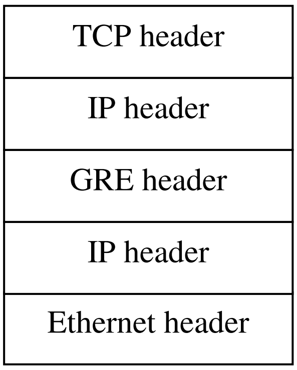


Figure 10 - Generic Routing Encapsulation in use

Note the occurrence of an IP header twice within the GRE packet. The first IP header (at the lowest level) would contain information pertinent to the two routers handling the tunnel. The second IP header would contain the real data relating to the packet being encapsulated by GRE. It is this second, inner packet that the system needs to use in order to classify the data inside the packet correctly. This was achieved by recognising that the first IP header encapsulated a GRE packet (the type field would be set to 47), reading over the GRE header, and then recursively calling the function to handle IP packet decoding on the second IP header. The only complication with this method is that the GRE header may be variable in length, which requires additional checks to determine how many bytes one should read over to find the second IP header.

Once support for VLANs and GRE had been integrated in to the packet decoding mechanism, no more unidentified packets were reported by the system.

Module support

Before going on to discuss how large volumes traffic are classified through the use of flows, it is first necessary to discuss the modules that classify individual packets and perhaps just as importantly, how these modules are kept modular. This was identified as a key requirement in the requirements section, primarily due to the fact that new protocols are emerging at an alarming rate.

Even at the beginning of development, the goals of this section were already known. These were as follows:

(1) The main client capture/analysis program should be compiled distinctly from the modules that perform packet classification
(2) New modules should be installable simply by adding a new file (and compiling the module in to a library)
(3) A mechanism for marking modules as active or inactive should be available, ideally via a simple configuration file

Some research on the Internet surrounding modular C programming soon turned up the dynamic linking library, also known as dlfcn.h. This allows the programmer to build a compiled shared library of code in one location, and then have another program use that library at *runtime*. The key word here is of course "runtime", as it means that the program using the library need not have the library available to it at compile time. However, this results in weaker compile-time checks, but this is a sacrifice that must be made for the sake of flexibility.

Using the dynamic linking library is straight-forward. One simply opens the shared library (a .so file) using dlopen, and then searches the library for the desired function by name using dlsym, which returns a pointer to the desired function. Of course, to search by name requires the program to know the name of the function it is searching for first, which may initially be seen as a problem. Furthermore, there may be performance issues when searching for functions in a dynamic library thousands of times per seconds (the rate at which packets are expected to arrive).

The finished solution overcomes these problems (the code is presented in Appendix R, appdetect.c, function init_modules()). A configuration file called "modules" is used that defines which modules should be loaded at start-up of the client application. In traditional UNIX configuration style, a line beginning with "#" is a comment and is ignored, so one can easily comment out troublesome modules or those they know will never be used. At start-up this file is read, and whilst iterating over each line a search is made in the dynamic library for the lines contents prefixed by "detect_". Should a matching function be found, its function pointer is added to an array of known modules. Thus, for example, a line in the configuration file containing "ftp" would initiate a search of the dynamic library at start-up for a function called "detect_ftp", and should the function be found, it would be recorded for use later in the array. Therefore, once this start-up process is done, the library handle can be discarded and never again be used, simply because pointers to all of the module functions are stored in an array already.

Each module is individually compiled in to an object file (.o), and then all of these together are built using "gcc –shared" in to a shared library (.so). This can then be installed in to the systems /usr/lib directory to make the library available to any application. By doing this, different modules can be added to the library simply by compiling the individual object file and rebuilding the shared library.

The observant reader will note though that all of this imposes another requirement upon us – each classification module must have a common interface. This is a good thing as it ensures consistency, but there is always the concern that this is at the cost of future flexibility. The interface used here is described below, using the FTP classification header as an example:

```
u_int16_t detect_ftp( struct packet *p, u_int16_t app,
                      DynamicApp **hashtable )
```

The first argument to the function detect_ftp is the packet being inspected. This structure is exactly the one discussed in the previous section, thus giving us access to the Ethernet, IP, TCP/UDP headers, and of course the packets data.

The second argument is the application ID, which will be non-zero should it have already been classified by another module. As will be seen later in the modified classification flow-chart, rather than terminate as soon as the application ID is found, it is sometimes necessary to fall through additional modules to either to find a more accurate match or to discover session setups within the data for other protocols.

For example, the HTTP module might trivially classify anything running over port 80 as HTTP data. However, if the traffic on that port was actually BitTorrent and we did not fall through to the later BitTorrent module, then the traffic would be incorrectly identified as HTTP. By falling through we would not only correctly identify BitTorrent as the protocol in use, but we would also gain the chance to scan the data for future session setups (which is a common occurrence in the BitTorrent protocol).

The third and final argument is a hash table of DynamicApp structures, shown below in Figure 11. These structures are a mapping of host, port and transport protocol (e.g. TCP, UDP) to an application, coupled with an expiry time for the mapping (after which time the contents of the structure should be ignored and it can be safely removed from the hash table). The hash function hashes upon the host, port and protocol. This hash table is used by the modules as a memory for session setups – that is, after detecting a session setup, they will insert a new item in to the hash table for later use by itself or other modules.

```
typedef struct _app_dyn {
  uint32_t addr;
  uint16_t port;
  uint16_t application;
  uint16_t protocol;
  time_t   expires;
  uint16_t ttl;
  struct _app_dyn *prev;
  struct _app_dyn *next;
} DynamicApp;
```

Figure 11 - DynamicApp data structure

Using the FTP example from the Literature Survey, should our module spot the string “227 Entering Passive Mode …” over an FTP stream we would know that the string immediately following this will identify the host and port that the next *passive* FTP stream will operate over. In this example the module would decode this host and port and then store it within the DynamicApp hash table, tagged with the FTP application ID.

Finally, the detection function in each module will return a single 16-bit unsigned integer identifying the application they have classified the packet as belonging to. If a module cannot identify the traffic, it simply returns the application ID that was passed in as the second argument (a zero is passed in on the first call, signifying "Unknown").

The original application classification flow-chart in Figure 4 of the high level design incorporated an optimisation that statically mapped known ports to applications (for example, 80 to HTTP). This was only performed for ports less than 1024, as these are regulated by the IANA and are reserved for usage by specific applications (IANA, 2007). The observant reader will note that this is notably absent from the description above. Indeed, whilst early prototypes did include this feature, it was actually found to cause frequent erroneous classifications on larger networks. For example, one static mapping was port 80 to HTTP, but some peer-to-peer clients use this port to overcome firewalls and proxy server restrictions. In this case, the traffic would be immediately misidentified as HTTP, without even touching the relevant peer-to-peer modules that would have classified it correctly.

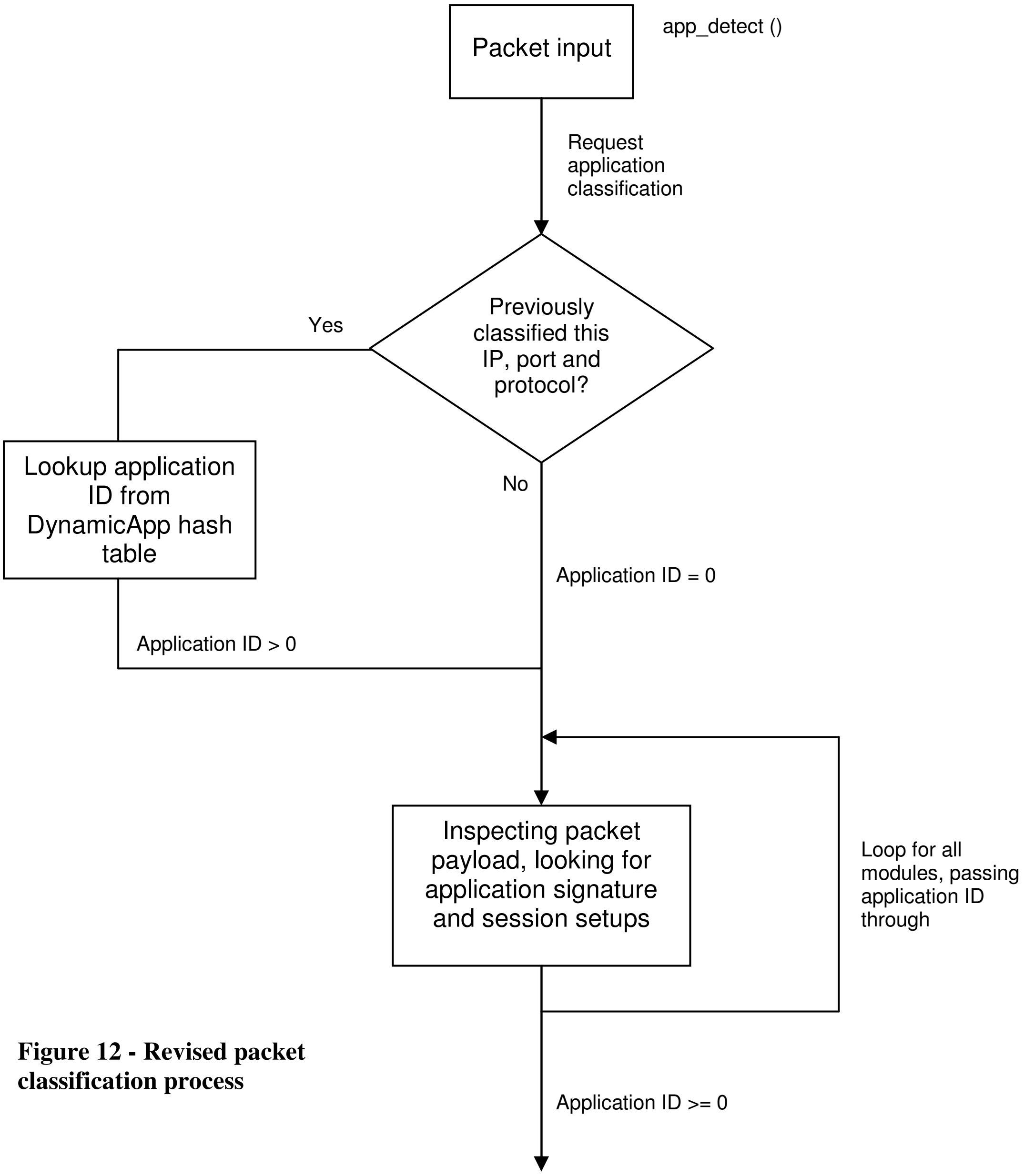

Packet input
app_detect ()
Request application classification
Previously classified this IP, port and protocol?
Yes
No
Lookup application ID from DynamicApp hash table
Application ID = 0
Application ID > 0
Inspecting packet payload, looking for application signature and session setups
Loop for all modules, passing application ID through
Application ID >= 0

Figure 12 - Revised packet classification process

This revised (and arguably simplified) traffic classification diagram represents increased computational overhead over the previous one in Figure 4. Whereas the previous one did not necessarily have to scan *every* packet to identify the traffic, this new approach does. But it does so with noticeably increased accuracy. Furthermore, whilst the performance hit could, in theory, be non-negligible on a high traffic network, some initial and informal benchmarks on a 630MB block of captured traffic show the system to be capable of processing 522Mbps using this new technique. That is well above the required 100Mbps, and this is using a relatively old Pentium 3 1.4Ghz machine. Further benchmarks will be detailed later in the testing phase.

In order to demonstrate the use of the classification modules before moving on to detailing the use of traffic flows, the construction of two of the more complex modules will now be described. Development of the remaining modules (of which there are over twenty) followed in a similar vein, referring to the RFCs where possible, and the well known port numbers provided by IANA where not.

BitTorrent module

A key motivation involved in the undertaking of this project has been the proliferation of BitTorrent in recent years. This is traditionally a difficult protocol to classify, partly because it does not have an IANA assigned port number, but also because the clients often randomise their listening port to avoid detection. The lack of a formal and universally accepted specification of the BitTorrent protocol has also hampered research efforts.

However, regardless of these issues, enough is known of the protocol to know that BitTorrent communication between two hosts will always begin with a handshake. It is precisely this handshake that is sought in order to identify the BitTorrent stream. According to numerous specifications available on the Internet (Theory.org, 2006; BitTorrent.org, 2006), the handshake begins with the following 20 bytes (the \19 represents the number 19, whilst the rest of the bytes are shown in their ASCII representation:

```
\19BitTorrent protocol
```

So, a packet from HostA:PortA to HostB:PortB beginning with the above string signifies the setup of a BitTorrent stream between those two host/port pairs. Note also that BitTorrent has evolved to use not only TCP but UDP as well, primarily because of the reduced overhead in sending acknowledgements and responses, so both protocols need to be watched on the same pair of hosts and ports.

This is precisely how the BitTorrent module operates (the source code for this module is included in Appendix R, bittorrent.c). For each packet passed to the module, the first 20-bytes are compared to the handshake message detailed earlier. Should this match, the IP addresses and port numbers are extracted from the packet, and inserted in to the DynamicApp hash table having been flagged with the BitTorrent application ID. A 30-minute expiry time (named “TTL” – Time To Live – in the structure) is used. For each handshake up to four inserts may be made – one for TCP and one for UDP, for both source and destination hosts. Should a host and port be seen that already exist in the hash table, they will not be re-added to the hash table, but instead the TTL will be reset.

FTP and Passive FTP

FTP has long been a very easy protocol to identify, primarily because it almost always sits on port 21 for commands, and 20 for data. However, that is "active" FTP, which actually results in the FTP server connecting back to the client from port 20. This may seem a bizarre concept nowadays, but like the X11 protocol that operates in a similar fashion, it was constructed long before the days when firewalls were necessary and NAT (Network Address Translation) routers began causing issues with such connection mechanisms.

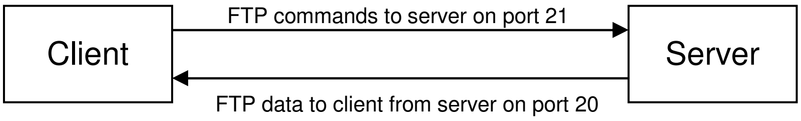

Client
FTP commands to server on port 21
Server
FTP data to client from server on port 20

Figure 13 - Active (PORT) FTP connection mechanism

As discussed in the Literature Survey, the inherent flaws within this mechanism were recognised and a modified form of the FTP protocol was devised. This allowed the client to initiate the connection to the server on some randomly selected unprivileged port for the FTP data transfer. This method is known as passive FTP.

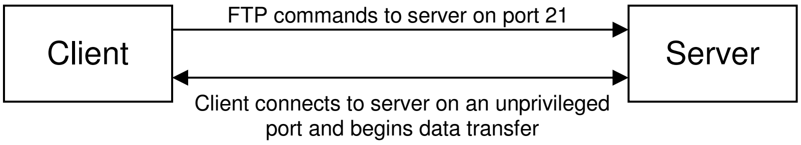

Client
FTP commands to server on port 21
Server
Client connects to server on an unprivileged
port and begins data transfer

Figure 14 - Passive (PASV) FTP connection mechanism

The host and port that the client should connect to the server on for data transfer would be described in the initial command stream (via port 21). The communication involved in setting this stream up is as follows:

```
[… FTP authentication …]
> PASV
< 227 Entering Passive Mode (80,77,246,42,195,149).
```

The numbers immediately following the string "`227 Entering Passive Mode`" are an encoding of the host and port number the client should connect to. This is encoded simply with the first four components representing the four octets of an IP address, and the last two representing the port number (multiply the first by 256 and add the second one – so port 50069 in the example above).

This newly discovered host and port is then added to the DynamicApp hash table and flagged with an application ID tagging it as FTP. This ensures that future streams seen using this host and port will correctly be identified as FTP traffic – even if the data is binary (i.e. unidentifiable) and flows between two arbitrary ports.

Traffic flows and reporting

Thus far this design document has covered the process of packet classification. What has yet to be discussed is what is done with these classifications.

The high-level design overview made mention of Cisco's NetFlow protocol. For each traffic flow, identified by source address and port, destination address and port, and protocol, the number of packets and number of total bytes is recorded. This constitutes a single "flow" of network traffic, and once this flow expires it is sent back to the flow recorder. Ultimately this process means that only a very small block of information need be sent back to the flow recorder to adequately describe a flow of thousands of packets (Feldman, 2004).

A similar process is used here, although the format of the flow structure is slightly different, to allow for some more advanced functionality. The structure of the flows used within this system is shown below in Figure 15. Note the principle additions to the flow structure are the application field, the expires field, and the sent field. The rest of the fields can be found, in one form or another, in the NetFlow protocol.

```
typedef struct _flow {
  uint32_t saddr;       /* Source address */
  uint32_t daddr;       /* Destination address */
  uint16_t sport;       /* Source port */
  uint16_t dport;       /* Destination port */
  uint32_t len;         /* Total bytes in flow */
  uint16_t packets;     /* Total packets in flow */
  uint16_t protocol;    /* Protocol ID */
  uint16_t application; /* Application ID */
  uint16_t reserved;    /* Reserved for future use */
  time_t   started;     /* Time flow began */
  time_t   expires;     /* Expiry time of flow */
  struct  _flow *next;
  struct  _flow *prev;
  uint8_t  sent;        /* Has the flow been sent already? */
} IPFlow;
```

Figure 15 - IPFlow data structure used within client application

Upon receiving a packet in to the main libpcap callback function, netmon_callback, the system will attempt to determine a classification for the type of traffic in the packet using the application detection methods described earlier. Regardless of the result, an attempt is made to find a previous flow that matches this one. As in NetFlow, a matching flow is said to be one where the source address and port, destination address and port, and protocol all match those of another flow. This is implemented using a hash table to ensure fast lookups. Should a flow not be found, one is created with the associated details and the application ID discovered earlier. If an existing flow is found, the number of bytes and packets are incremented accordingly, and the application ID is set if one was not discovered previously. This is process is shown in detail in the source code of main.c (Appendix R) in the function netmon_callback.

The concept of a flow expiring is an important one. In NetFlow, a flow expires when a TCP RST or FIN flag is seen in TCP connections, or when the flow is dormant for a period of time (typically five minutes). This alone is inadequate for the near real-time purposes of the project, so some additional measures have been introduced. By default, a flow is set to expire at any of the following:

(1) One minute after its creation
(2) Over 1MB of data are recorded for the flow
(3) Over 1000 packets are recorded for the flow

Once the flow has expired it is guaranteed to no longer be used. Any new data matching that flow will be recorded in a new flow, and this expired one will simply be ignored by all of the hashing functions.

However, the expired flows are not merely discarded just yet. At least once per minute a process begins that finds all expired flows. Should any exist, a TCP connection on port 5000 (by default) is established to the aggregation server and the expired flow information is transferred. Once these flows have been sent to the server, they are removed from the hash table and freed from memory. It is worth noting here that because this hash table is expected to fill up very quickly on a high traffic network, we do not wait for the conservative garbage collector (GC.h) to determine that the memory is no longer in use and free it. Instead, the memory is freed directly using the provided GC_free function.

It is important to note that only the first 32 bytes of the flow are sent to the server, as the remaining fields (next, prev and sent) would be irrelevant and simply a waste of bandwidth. This will become more significant later in the decoding at the server-side of the communication. So on a standard network with an MTU (Maximum Transmission Unit) of 1500 bytes, 45 different flows may be held within one packet (1500 – 20 (IPheader) – 20 (TCPheader) = 1460. 1460 / 32 = 45.625).

Changes required following implementation on a high traffic network

Upon debugging the client system installed in Fluidata’s network it soon became apparent that much CPU time was being wasted by hash table lookups and scans of the flow table to see if expired flows needed to be sent. In the low traffic environments of the Bath and Plymouth networks that early development and testing was performed upon, this issue did not arise.

Fortunately these issues had been anticipated as a potential bottleneck of the client earlier in development, and as a result constants defining the size of hash tables and how often checks of the hash table should be made were defined in .h files. By increasing the default size of the hash tables from 10000 records to 40000, there was four times as much space available within the table before spilling had to be performed to linked-lists (which trail off each node within the table). Whilst this resulted in an increased memory footprint, the size was still trivial when one considers the system in question had 6GB RAM. There is still significant head-room for increasing this hash table size further.

The frequency at which the flow table was checked for expired flows was increased from once in every 1000 packets to once in every 20000 packets. Fluidata's network, on average, handled some 10000 packets per second, so even in the worst case a scan of the flow table would still be performed every few seconds. The only requirement here is that the client report flows in to the server at least once per minute. To do so more regularly is perfectly acceptable, but would mean valuable processing time would be wasted establishing a connection to the server. Again, there is further room for growth here, as the reporting frequency could be lowered further, allowing more time for data processing.

Aggregation server

The high-level design overview identified that the aggregation server's function was a very simple one – receive data from the clients, decode it and insert it into the database. This is almost precisely what the resulting server does.

In anticipation that the server may have to cope with large volumes of connections and operate very quickly; C has again been chosen as the programming language. Listening on port 5000 by default, the server waits for new connections and performs synchronous I/O multiplexing using the select() function from the C library. This allows multiple clients to report in to the server (seemingly) simultaneously. Usage of the select() statement was adapted from an example found in Beej (2007).

Upon receiving a connection, the data from the packet is fetched and the individual flows are decoded one at a time (each being 32 bytes long, as detailed earlier). Decoding a flow is trivial because the precise structure of the flows memory layout is already known. Thus the data is simply mapped (from each 32-byte offset) to a flow structure. The structure of each flow is identical to that of the client, with the exception of the final missing three fields – next, prev and sent – which are neither required nor received by the server.

In one small difference from the high-level design presented earlier, the server application itself does not insert in to a database. In fact, the application knows nothing of a database. Instead, the application simply outputs SQL-standard INSERT statements to STDOUT, which can then be piped into any database of the user's choice. For example, one could run the application with a MySQL database using:

```
./netmond | mysql -u [username] -p -D netmon
```

Or with a PostgreSQL database:

```
./netmond | psql -U [username] -W -d netmon
```

The point here is that the server does not care about the database the user uses it with – it simply has to conform to SQL standards. The client performs all of the hard and complex work, so programmers wishing to use the collected data in a different way need only redirect the server to a different database and begin using that.

There is a very small performance penalty for using this method of inserting into the database, but this is irrelevant when one considers that clients will only be reporting back to the server a few times per minute and that most databases can handle hundreds, if not thousands, of inserts per second.

The INSERT statements output by the program do assume a structure for the database table though. The structure of the table, using MySQL formatted type names and declarations, is presented in Figure 16.

```
CREATE TABLE flow (
  id int(11) NOT NULL auto_increment,
  saddr varchar(15) NOT NULL,
  daddr varchar(15) NOT NULL,
  sport int(10) unsigned NOT NULL,
  dport int(10) unsigned NOT NULL,
  len int(10) unsigned NOT NULL,
  packets int(10) unsigned NOT NULL,
  protocol smallint(5) unsigned NOT NULL,
  application smallint(5) unsigned NOT NULL,
  started datetime default NULL,
  expires datetime default NULL,
  PRIMARY KEY  (id),
  KEY saddr (saddr,daddr)
)
```

Figure 16 - Database table structure for flow summaries

Network architecture

Having now described in depth the design and implementation of both the client and server applications, it would be useful to provide a short discussion of the network architecture that must be employed to operate the system. The Literature Survey outlined a number of possible architectures, including the use of network “taps” and port mirroring. Ultimately, port mirroring has been chosen here as it is both simple to use and free (providing the switch in use already supports it). Conversely, network taps are very expensive, putting them out of reach of the testing phase of this project. The system architecture could be very easily adapted to work with taps by simply feeding a tapped output into the mirror input on the capture/analysis client.

The port mirroring method relies upon the network switch allowing an administrator to mirror (clone) all inbound and outbound traffic from one or more ports to a single target port. A diagrammatic explanation of this can be seen in Figure 17.

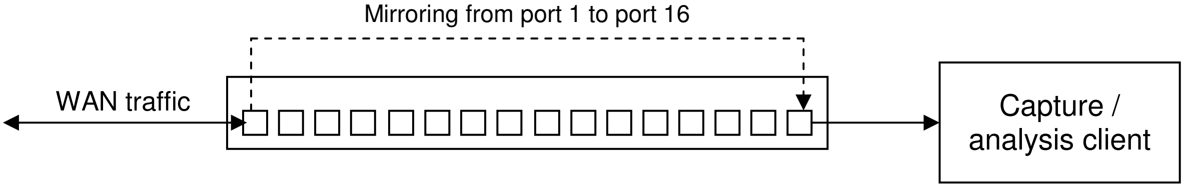


Figure 17 - Traffic capture using port mirroring

In Figure 17 the switch's first port is connected to the WAN (probably via a router of some form), its last port is connected to a capture/analysis client (as described earlier), and it has fourteen remaining ports for individual hosts (or perhaps other switches and routers). Port 1 is mirrored to port 16, allowing the capture/analysis client visibility of all WAN traffic passing through this switch. Please note that it is not necessary to capture the remaining fourteen ports, as all of their WAN traffic must flow through the first port anyway. In fact, if all fourteen other ports were to be mirrored to the last port as well, visibility of internal network traffic would also be available. This is certainly another possible use for the system, although not the intended one.

Figure 18 presents a high-level view of the architecture after bringing multiple switches, capture/analysis clients, and an aggregation server into the equation.

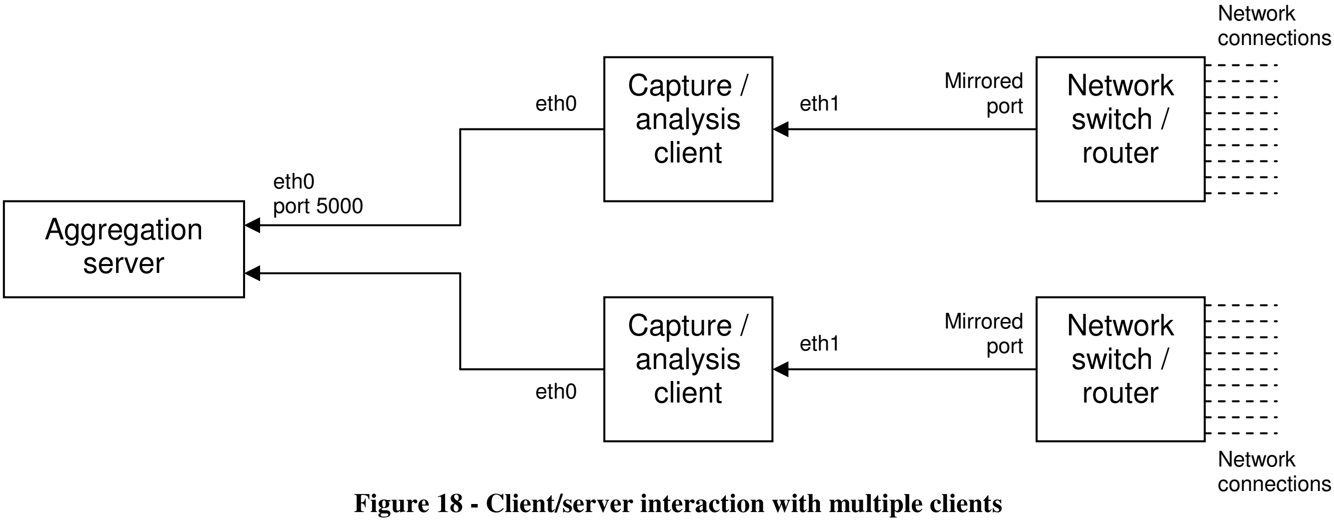


Figure 18 - Client/server interaction with multiple clients

Both the capture/analysis client and the aggregation server have been developed using the standard C libraries, as well as the aforementioned GC.h and libpcap. Therefore, whilst the system was developed on Linux machines, it should in theory work on any compatible platform – including Solaris, BSD and Windows.

The database and associated processes

At this point in the design a single database table “flow” exists that stores each network flow, along with its application classification. This raw data now needs to be transformed in to something more usable.

The plan to split this data in to two databases has remained unchanged from the original high-level design. Detailed graphing data is stored in RRDs (Round Robin Databases) suitable for use with the graphing application RRDTool, whilst summary data for use on the front end is stored in a relational database using MySQL.

RRDs

RRDs are an excellent means of storing time-series data, as they allow the user to define different periods of time that they want data stored at different resolutions. The RRDTool application itself handles the process of averaging the data across these intervals – the user simply has to insert data into the database at predetermined regular intervals. The command in Figure 19 was used to generate each of the RRDs.

```
1. rrdtool create \
2.   --start now-43200 [filename] \
3.    --step 60 \
4.    DS:InOctets:GAUGE:120:0:12500000 \
5.    DS:OutOctets:GAUGE:120:0:12500000 \
6.    RRA:AVERAGE:0.5:1:360 \
7.    RRA:AVERAGE:0.5:5:288 \
8.    RRA:AVERAGE:0.5:60:168 \
9.    RRA:AVERAGE:0.5:180:248 \
10.   RRA:AVERAGE:0.5:1440:365 \
11.   RRA:MIN:0.5:1:360 \
12.   RRA:MAX:0.5:1:360
```

Figure 19 - Command used to generate RRDs

Breaking this command down will provide the reader with further insight into the purpose and use of RRDS. The string “—start now-43200” on the second line tells us that the database began 12 hours ago (all times are presented in seconds here). The “—step 60” on the third line means that the RRD expects new data to be input once every 60 seconds. The fourth and five lines define the two datasets – InOctets and OutOctets (essentially “Bytes in” and “Bytes out”). The “120” in lines four and five indicate that data may arrive between 0 and 120 seconds per primary data point – this is done to allow for variances in the reporting time, as data might not be available precisely every 60 seconds.

The remaining lines (six through twelve) define the RRAs. These are where the data is actually stored, averaged according to different functions, over different intervals and at different resolutions. The first RRA on line six is an average data set, averaging over every single data point (in effect creating no average at all – it is just a clone of the input data) for six hours (360 data points * 60 seconds per data point = 21600 seconds = six hours). The second RRA averages five data points and stores 288 such averages, thus we have 5-minute resolution data for the past day (5 * 288 * 60 = 86400 = 1 day). The third RRA creates 60-minute resolution data for the past week, the fourth creates three-hourly resolution data for the past month, and finally we have 1-day resolution data for the past year. The final two RRAs store minimum and maximum values seen in the past six hours.

Inserting data into the database is achieved using a command of the form:

```
rrdtool update [filename] -t InOctets:OutOctets \
        [timestamp]:[bytesIn]:[bytesOut]
```

Note though that once data is inserted it cannot be modified or deleted. It is a database in the most primitive sense – you can store data in it, but it provides very little additional functionality you would expect from a modern database engine. But this is precisely what it was intended for – to store time-series data that would not change, and of course, to operate very quickly.

Ultimately this has left us with a suitable means of storing time-series data over a range of different intervals. The different resolutions also meet requirement 8, which defines the minimum resolutions that data must be available at. However, deciding how to split these RRDs and how to populate them remains to be done.

Referring back to requirement 10, it can be observed that data needs to be split in the following ways:

(1) Across all groups, i.e. in total
(2) Per group
(3) Per application
(4) Per host

Therefore the following RRD files have been proposed and implemented, all generated according to the same command earlier (thus they all have the same data storage capability):

(1) total.rrd – Totals across all groups
(2) group*X*.rrd – Totals for each group X
(3) group*X*_app_*Y* – Totals for each group X, using application Y
(4) total_app_*X*.rrd – Totals for each application X, across all groups
(5) *IPAddress*.rrd – Totals per application for IP *IPAddress*
(6) *IPAddress*_app_*Y*.rrd – Totals for application *Y* for IP *IPAddress*

These RRDs provide the required data storage, although it may be necessary to use multiple RRDs in tandem to produce some of the more complex graphs. Fortunately, this is a feature that RRDTool supports.

The data is inserted in to the RRDs by way of a process starting each minute on the database server. This process reads over the contents of the flow database table and groups the data according to the various different groupings required (i.e. in total, per group, per application, per host) and then writes the results out to the RRD files.

Early in development it was noted that launching each RRDTool command consecutively (i.e. one after the other) resulted in the process taking considerable time to execute with only relatively few hosts. However, it was realised that because all of the RRD files were completely independent of one another, they could be written to *to what?* simultaneously without conflict. Therefore the RRDTool commands were all launched as background threads and left to complete in their own time (typically less than 0.1 seconds).

It is also worth noting at this point that the process that harvests the flow database table and populates the RRD files is written in the PHP scripting language. PHP was chosen here due to its extremely flexible array support and ease of use. Its speed was of secondary consideration, as the complexity remains in the RRDTool application, which the PHP script is capable of launching multiple instances of simultaneously.

Changes required following implementation on a high traffic network

Despite the changes described above, once testing began on Fluidata's network, problems were soon observed. With some 2000 hosts on the network, each typically using 5-10 applications at once, one would have to launch between 10000 and 20000 processes per minute to record the data in to the RRD files. Testing soon demonstrated this to be completely infeasible.

Experimentation was performed using UNIX pipes to pipe the commands directly into a single instance of RRDTool (thus saving on the start-up costs of launching each process individually). This yielded greatly improved results, and has remained the method of choice for both storing data in RRDs and retrieving it.

Furthermore, it was also found that the process of traffic averaging using MySQL was too slow. Traffic data was being written in to two places – the RRDs (for graph generation) and a flat database table known as "host_traffic", which would later be summarised into the 1-hour, 6-hour and 24-hour averaged tables. It was these summarising processes that consumed far too much time to run once per minute. As a result of this, a change in design was instigated.

Instead of having the aggregation script write to both the RRDs and the MySQL database, the script was modified just to write to the RRDs. RRDTool would automatically average the data over those time periods anyway, so it was reasoned that one could gain speed by having RRDTool perform the CPU intensive work of averaging, and then simply extract the averaged data afterwards via another script. Changing the system to run in this manner instead resulted in significant gains that allowed the system to better handle the large volumes of data being captured from Fluidata's network.

Finally, problems were later observed when trying to administer the vast number of RRD files created. After operating for five days on Fluidata's network, some 3GB of RRD files had been generated. The size on disk was not an issue (as RRD files are constant in size once created), but instead the sheer number of files in one directory was causing a file system bottleneck. For this reason the RRDs were reorganised into a series of sub-directories of the form shown below in Figure 20.

```
/rrds
      /totals
            /appXX.rrd   (An RRD per application)
      /groups
            /[Group ID]
                  /appXX.rrd   (An RRD per application, per group
      /hosts
            /OctetA
                  /OctetB
                        /OctetC
                              /OctetD
                                    /appXX.rrd   (An RRD per app, per IP)
```

Figure 20 - Directory structure of RRD files

For example, the RRDs for 89.105.120.5 would be stored in /rrds/hosts/89/105/120/5/. This process reduced the number of files that could be present in any one directory to be limited by the number of applications defined (currently 26).

Graph generation

In the preliminary high-level design it was stated that graphs would be generated at every level, every minute. Moving past the figure of only 50 hosts on the network, it soon became apparent that this was infeasible. Instead, only the most visible graphs (i.e. the totals per group, application and host) are generated automatically every minute. The remaining graphs (a much larger figure) would be generated as and when users attempted to access them.

Graph generation is achieved using the application RRDTool, which utilises the RRD databases created and populated earlier. The commands to generate the graphs are built dynamically too. Whilst they could be constructed statically, that would require the inclusion of every possible data series on the graph to ensure that nothing was missed. Instead, only the data series that have transferred data within the specified time frame are used as data sources for the generated graph. This makes for a shorter generation time and a less confusing and cluttered graph. The graphs are generated using a command similar to the one in Figure 21.

```
rrdtool graph /home/netmon/html/192.168.254.211_both_app.png \
    --start now-21600 \
    DEF:DP1in=rrds/192.168.254.211_app_5060.rrd:InOctets:AVERAGE \
    DEF:DP1out=rrds/192.168.254.211_app_5060.rrd:OutOctets:AVERAGE \
    CDEF:DP1=DP1in,DP1out,+ \
    CDEF:Ln1=DP1,DP1,UNKN,IF \
    CDEF:KB1=DP1,1024,/ \
    AREA:DP1#EC9D48:"SIP         " \
    'GPRINT:KB1:MIN:  Min\: %6.1lf' \
    'GPRINT:KB1:MAX:  Max\: %6.1lf' \
    'GPRINT:KB1:AVERAGE:  Avg\: %6.1lf' \
    'GPRINT:KB1:LAST:  Current\: %6.1lf KB/s \\n' \
    LINE1:Ln1#CC7016 \
    --title "Traffic by application for 192.168.254.211" \
    -v "Bytes per second" \
    -w 600 -h 150
```

Figure 21 - Sample RRDTool graph generation command

It can be seen that this will generate a graph for the past six hours (now - 21600 seconds = 6 hours ago), using the data from the file “192.168.254.211_app_5060.rrd”, which is the SIP (VoIP) RRD for the host 192.168.254.211. Note that only the SIP data point is being plotted. This is because the host 192.168.254.211 is a VoIP phone and is therefore transferring SIP traffic only. Far larger examples with up to ten data series exist, but these would be unlikely to fit on a single page, and thus be very hard to explain. Ultimately, this will produce a graph similar to the one in Figure 22.

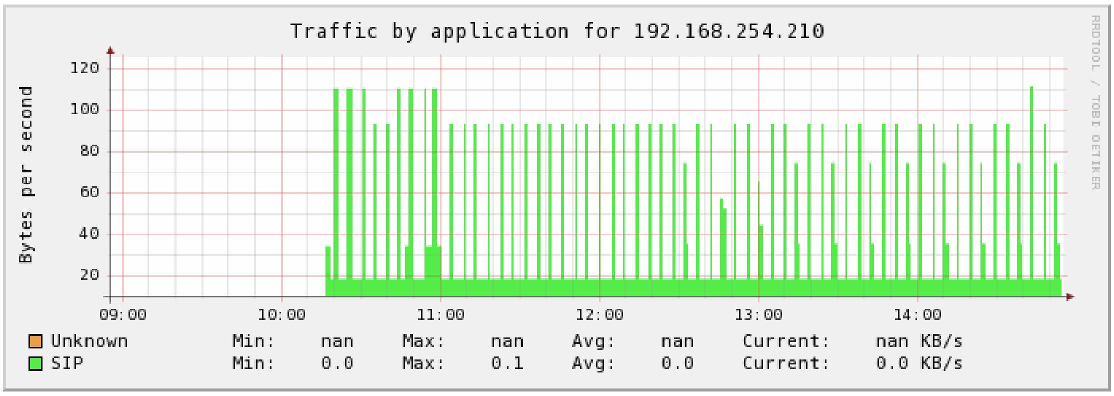


Figure 22 - Sample SIP traffic graph for 192.168.254.210

Changes required following implementation on a high traffic network

Graph generation on the small home networks in Bath and Plymouth (as discussed in the testing section) could be performed once every minute without any significant system load. However, once the feed from Fluidata was activated, it became clear that there were often well over 1,000 hosts active at a time. This warranted the inclusion of the method discussed briefly before – generation of graphs only upon request.

Since the graph generation commands were all dynamic (i.e. one would only graph the applications that a specific host was using), these commands could not be hard-coded into the front end. Instead, a series of processes runs each minute that creates the commands to generate all of the graphs, and stores these commands in the database (Note that it does not *execute* the commands – only generates them). Upon loading a page in the front end, the graph’s name (which would be constant based upon what the user was looking at – e.g. “ip192.168.0.1_app25.png”) is used to lookup the command to generate the graph. This command is then executed, the graph is saved to disk, and the page continues loading. An individual graph takes approximately 0.1 seconds to generate using RRDTool, so the slowdown on page load is minimal.

MySQL database

The other main function performed by the database processes that run every minute is to extrapolate the contents of the raw flow table to produce summary data that is inserted in to the other tables. The primary tables that are populated from this method are as follows:

Table name	Stores
app_usage_curr	Current bytes in/out per second for each application in use per IP address and/or group
app_usage_1hr	Total bytes in/out per application per IP address/group averaged over the past hour
app_usage_6hr	Total bytes in/out per application per IP address/group averaged over the past 6 hours
app_usage_24hr	Total bytes in/out per application per IP address/group averaged over the past 24 hours
ip	A log of all IP addresses seen, and when they were last seen
host_totals	A breakdown at 1-day resolution of the application and bandwidth usage per IP address. Used primarily in the front end for summaries per host.

Table 1 - Database tables used for traffic averages

An experienced database user will soon realise that, even from this level, it is clear that the database is not properly normalised. This results in data duplication and wasted storage space. The primary reason for not fully normalising a database is performance, and this is precisely why the database has not been fully normalised in this instance. There is the potential for many of these tables to grow to hundreds of thousands (if not millions) of rows, and joining these tables dynamically and querying upon these joins would result in some very slow queries that would likely delay the front end noticeably.

For this reason the decision was made to split the data out into a range of different tables. Some informal testing conducted on the database has shown that even with ten million rows in the database, the performance remains strong and the database size is just under 2GB. Whilst this is rather large for a database of this type, it is certainly still well within limits, especially given the decreasing cost of hard disk storage.

All of the above database tables are updated once per minute by the extrapolating processes that operates on the flow table. When updating the data, a dirty bit is set on each old row in the table, the new data inserted, and then all data with the dirty bit set deleted. This ensures that the table will always be non-empty, thus ensuring that the viewers on the front end do not find themselves presented with an empty screen should they visit at the precise moment the data is being updated.

It is worth noting at this point that there are three other database tables that the back-end processes rely upon, but which are not modified by them. These are as follows:

Table name	Stores
application	This table maintains a mapping of all application IDs (as used in the back-end processes) and their names. For example, the application ID 80 maps to HTTP. It is common for the application ID to match the commonly used port number for the application in question.
group	This table is the authorative store for all of the groups. This will include a group ID, name and a compound ruleset that describes the networks that belong to the group.
ip_group	This table contains the individual IP address to group

	mappings, as extrapolated from the group table. Again, it would be possible to find an IP address from a groups rule, but this would require some very complex SQL and would also be very slow. The group and ip_group table should always be kept in synchronisation.

Table 2 - Static database tables

A complete database schema can be found in Appendix G.

Front end

The front end has been through several design iterations since the early mock-up used for requirements validation. Ultimately, in conjunction with user feedback, the final design has come around full circle to be very similar to the preliminary design presented in Appendix E. Some additional features have been incorporated as a result of user feedback. These are detailed in the sub-section titled “Changes as a result of increased scale and user feedback”.

As was highlighted in the high-level design, usability should be the key consideration when designing the front-end. This has certainly been attempted here. Cascading style sheets have been used throughout to ensure consistency in terms of font sizes, faces, colours, hyperlinks, borders, table sizes, and so on. Wherever a table is presented (as they are on most pages), it is limited to displaying 10 rows of data by default, with an option at the bottom to expand it to display all of the rows. This prevents the page filling up with hundreds of rows when the user does not even need to see them. Furthermore, all tables presenting tabulated data can be sorted by any column heading, simply by clicking on the column heading. Given that a large number of websites use similar functionality, this should be intuitive to the users.

As discussed in the database processes section, frequently used graphs will be pre-generated every minute. This means that the user visiting the page will simply be served a static image, which is noticeably faster than building the graph dynamically from the database. These frequently used graphs are deemed to be those visible immediately after clicking on group, application or hostname hyperlink. Additional graphs, such as the separate inbound/outbound view of the default graph and the historical view of the same graph (split over the past day, week, month and year), are all generated on-the-fly when the user visits the page.

A small and simple menu system has also been incorporated into the front end. This provides immediate access to the front page, each individual group, and the various configuration options that will be touched upon later. User feedback from initial designs prompted the inclusion of a host search box on the menu too. Users noted that they often had the need to monitor the traffic of one or two key users, and that it was tiresome to find the group they belonged to, click on it, locate the host from the groups host list, and then click on it. Instead, they noted that a search box would likely be the quickest and most convenient means of jumping to the correct page.

The finished design for the statistics pages consists of a series of inter-connected pages, each one drilling down one level further in detail than the last. Figure 23 illustrates how these pages are interconnected.

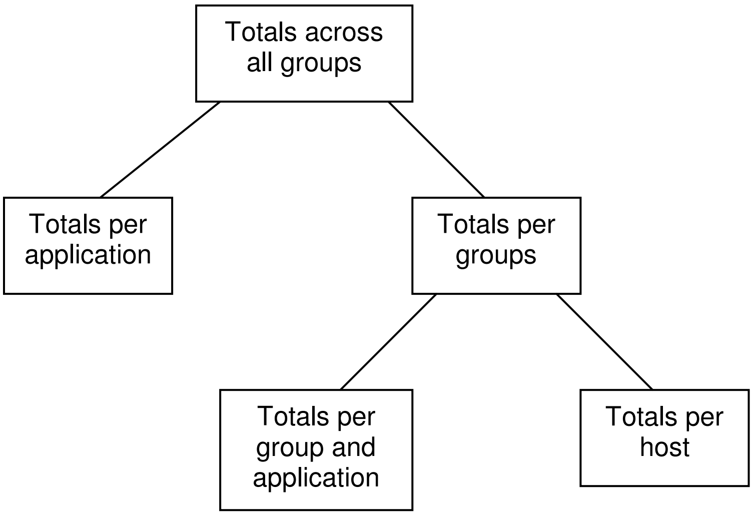


Figure 23 - Front end page hierarchy

The entry page to the system represents the top of the tree in Figure 21. This page incorporates an at-a-glance graphical overview of the traffic for each group currently monitored, as well as tabular data to back this up with firm figures. Additional tables representing the most traffic-intensive applications and hosts are also included on the front page. These can of course be reordered by the user to display the least used applications and hosts too.

The first of the two application-centric pages presents data for a specific application across all of the groups. One could consider it a view of the front page, specialised with respect to one specific application. Again, this includes a graph depicting the different groups' usage of the application in question over the past six hours. As with every major graph in the system, the user can opt to view graphs going back up to one year by clicking on the "View historical" link beneath the graph. Also included on the page is a quick at-a-glance list of the heaviest users of that particular application across all the groups. User feedback has noted the usefulness of this table in particular.

The next page in the front end focuses upon the traffic in specific groups. Here, a graph of application usage across the group is presented, complete with the standard options of splitting the graph in to individual inbound/outbound displays and the historical view. Again, a tabular breakdown of the heaviest users and most used applications within the group is displayed. However, as a result of user feedback in to the initial designs, a table presenting the total bandwidth usage across the group is presented as well, detailing bandwidth usage for the current day, the previous day, the past 30 days and the current month. Users cited that this would be useful in helping them monitor their users' traffic patterns and detecting significant over-usage.

The second of the application-centric pages is accessible from the groups page only. This presents a view of a specific applications usage within a specific group. The graph presented on this page is the first (and only) within the system that splits the data down by IP address. This allows the user to see visually which hosts are the heaviest users of specific applications. As on the group page, a tabular traffic summary of the current day, previous day, current month and past 30 days is also included.

The final page represents the lowest level view a user can see – individual hosts. This page includes a summary graph of the traffic for different applications for that host for the past six hours, as well as a tabular summary of the same data. A traffic summary is also included, again detailing the host's total bandwidth usage for the current day, previous day, current month and past 30 days. Unlike all other pages though, a series of graphs is then presented at the bottom of the page. Each of these graphs represents the individual traffic usage for that host for a specific application over the past day. This has been included again due to user request, primarily because users cited that the summary graph could be too confusing when a large amount of traffic spread over a large amount of applications is presented simultaneously.

One item that was not really covered in the preliminary high-level design was that of defining the groups themselves. Requirement 15 stated that the user needs to be able to define named groupings of hosts by single IP address, a range of IPs and an IP/subnet mask. This functionality has been designed and implemented using two simple pages – one which lists all current groups, providing links to add, edit and delete them, and another page that allows for the addition and editing of groups. It is important to note that as soon as any changes are made to the groups (be it adding, deleting, or editing), the previously mentioned extrapolation script is called that repopulates the ip_group database table based upon the contents of the group table.

Security was also highlighted in requirement 19 as being a concern. However, rather than spend many hours designing and developing a security sub-system (which is far from the focus of the project), it has instead been decided to simply utilise the Apache webservers built in security mechanisms. Using .htaccess and .htpasswd files, a server administrator can easily restrict certain web directories to not only specific usernames and passwords, but also specific IP addresses or subnets. The access control files need not be flat files either – modules such as mod_auth_mysql for Apache allow access control to be specified in a database. One could also configure Apache to operate over SSL, ensuring that all communications between client and host are encrypted. In summary though, user access control will not be built into the system directly – instead external systems will be used to provide the necessary security, and users would be required to configure them accordingly.

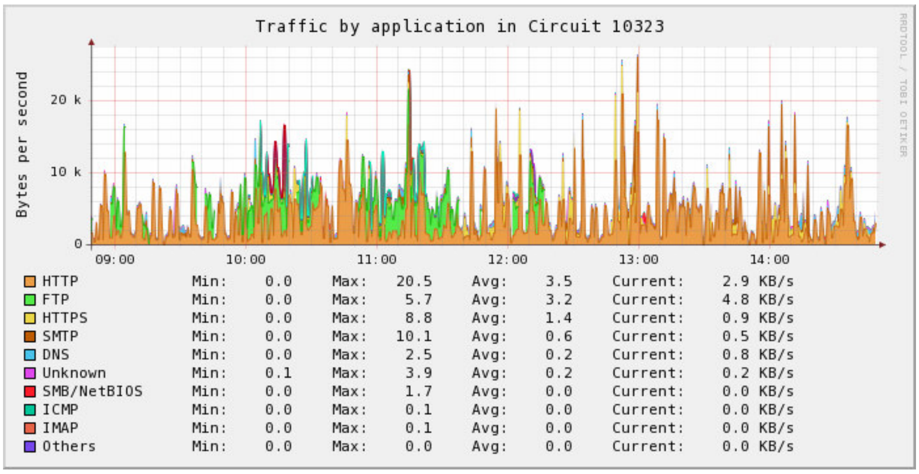
Changes as a result of increased scale and user feedback

The front end of the system was originally designed and tested on the small networks of Bath and Plymouth. Moving it to the far larger production network of Fluidata posed numerous design and usability challenges that had to be overcome. The following changes are based upon the preliminary feedback from Fluidata, of which notes can be found in Appendix M and N.

Graphing appearance

Preliminary feedback from Fluidata noted that whilst the stacked graphs provided a very detailed breakdown of application usage within a group or by a specific host, they often became cluttered and hard to visualise. With over twenty different data series on some graphs, the colours used to represent each often overlapped or blurred in to one, making visualising the data harder still.

The first solution to this problem, discussed at length with Fluidata, involved representing only the top ten (or any other suitably small number) data series distinctly on the graph, and then grouping the rest together. This led to graphs such as the following appearing (note the final data series in the table is labelled "Others").



```
🟧 HTTP        Min:   0.0   Max:  20.5   Avg:   3.5   Current:   2.9 KB/s
🟩 FTP         Min:   0.0   Max:   5.7   Avg:   3.2   Current:   4.8 KB/s
🟨 HTTPS       Min:   0.0   Max:   8.8   Avg:   1.4   Current:   0.9 KB/s
🟫 SMTP        Min:   0.0   Max:  10.1   Avg:   0.6   Current:   0.5 KB/s
🟦 DNS         Min:   0.0   Max:   2.5   Avg:   0.2   Current:   0.8 KB/s
🟪 Unknown     Min:   0.1   Max:   3.9   Avg:   0.2   Current:   0.2 KB/s
🟥 SMB/NetBIOS Min:   0.0   Max:   1.7   Avg:   0.0   Current:   0.0 KB/s
🟩 ICMP        Min:   0.0   Max:   0.1   Avg:   0.0   Current:   0.0 KB/s
🟧 IMAP        Min:   0.0   Max:   0.1   Avg:   0.0   Current:   0.0 KB/s
🟪 Others      Min:   0.0   Max:   0.0   Avg:   0.0   Current:   0.0 KB/s
```

Figure 24 - Traffic graph with minority traffic grouped together

However, despite this significant improvement, problems were still experienced. Fluidata noted that when a few pixels on the graph could represent tens or hundreds of kilobytes per seconds, it made it very hard to distinguish between data series. This led to the suggestion of zoom functionality.

Research conducted into the possibility of zooming into RRDTool generated graphs soon found a GPL-licensed script known as "Bonsai" that performed precisely this purpose (Steffen, 2004). However, testing of the script discovered a flaw – it only allowed for horizontal zooming (i.e. one could zoom in to a time span, but could not zoom to 50KB/s to 100KB/s traffic rate). The script was modified to support vertical zooming as well, and this modified version is included in the source code in the appendix.

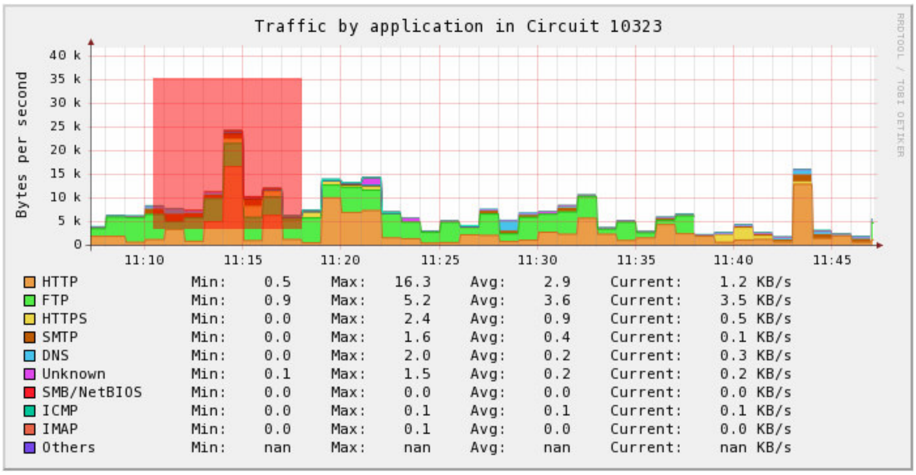


Figure 25 - Zooming in to a traffic graph

Grouping functionality

Fluidata were also keen to use the grouping functionality of the system to its full extent. Early into the development they supplied a list of some 361 different groups and IP allocations. Whilst this importing of these groups and the additional data collection overhead did not pose a problem, issues did appear on the front end.

Firstly, the number of data series on the front page graph was far too many, even with both of the improvements discussed earlier in this section. Furthermore, generating a graph with 361 different data series took upwards of ten seconds – far too slow for real-time usage. For these reasons the front page graph was modified to show total inbound and outbound traffic rather than every single group.

Secondly, the menu at the left of each page previously held a basic list of all of the groups to allow users quick access to any group. Whilst this worked well with only two groups (Bath and Plymouth), listing 361 groups caused the page to become far too long and take too long to render. For this reason the list was modified to use an HTML drop down list instead.

Unknown traffic

Despite the best efforts to classify as much traffic as possible, a portion of Fluidata's traffic could never adequately be classified. As a business ISP, some of the traffic they carry is likely to be sensitive and warrant encryption. Fluidata confirmed that this was the likely source of the unknown traffic (See notes in Appendix N). Nonetheless, upon observing such traffic Fluidata suggested that an option be included to allow users to breakdown this "Unknown" traffic to the host/port level, such that they could more easily spot trends in unclassified traffic and perhaps incorporate new classification modules.

In the interests of improving future traffic classification rates, a page achieving this functionality was created and could be accessed from the "Breakdown Unknown traffic" link from an individual hosts' page.

Alerting

The final part of the front end is the alerting system. A number of small and simple pages allow the user to view a summary of existing alerts, via logs of previous alerts that have been sent, and add, edit and delete hosts. The majority of the work surrounding the alerting system has been focussed upon the rule generation wizard. This has followed very closely from the original design, and thus includes the following sub-page:

(1) The first page allows the user to define which applications the rule will monitor (or instead, which applications it will specifically *not* monitor). Selecting none will assume all applications should be monitored.
(2) The second page allows the user to define specifically which hosts will be monitored as a part of the rule. Selecting none will assume all hosts are monitored.
(3) The third page lets users give other users the ability to set the different traffic levels upon which the rule should trigger. This can be based upon current traffic exceeding or falling below a certain threshold, or indeed hourly-averaged or six-hourly-average traffic exceeding or falling below a threshold. Whilst previous pages have allowed for nothing to be entered, at least *one* option must be selected here (otherwise, alerts would be sent whenever any traffic was passed).
(4) The fourth page allows the user to define when the rule should run. This may be either continuously, or only at different times of the day. User feedback also suggested that an option to exclude alerting at the weekend be included.
(5) The fifth page allows the user to define a name for the alert, and also select which contacts should receive the alerts from a list box, populated with contacts created back on the main page.
(6) The sixth and final page asks the user to confirm all details for the alert and indicate whether it should be activated following confirmation. Links back to the previous sections are also provided directly from this page, so users may return and edit their previous selections. This is also the page users are directed to when editing alerts.

The high-level design briefly discussed how the alerting wizard builds up a set of constraints stored in a PHP session as the user progresses through the stages. Once the wizard finishes, these constraints are transformed in to an SQL statement that is itself then stored inside a field in the “alert” table in the database. The backend alerting process, which runs every minute (the same frequency at which data is received), then walks over all of the entries in the alert table, executing the SQL statements dynamically. Should the execution of the SQL statement return any rows, then the process knows that the alert has been triggered and it begins alerting the appropriate contacts.

The following three database tables are used within the alerting system, both on the front end and back end.

Table name	Purpose
alert	This table stores the name, description and raw SQL statement for each individual alert, along with a flag indicating whether or not this alert is currently active.
alert_contact	This table stores the individual contacts, complete with email addresses and SMS details, that may be attached to any alert.
alert_contact_link	This table maintains the many-to-many relationship between the alert and alert_contact table. This is required in order to keep the data normalised. Note that in this case the data is fully normalised, as the table size is expected to be relatively small (never more than a few hundred or at most thousand rows).

Table 3 - Database tables utilised by the alerting system

User documentation

Early on in the design and implementation process it became clear that the wide range of different components of the system and large number of options available when configuring each would likely warrant a user manual. Indeed, following the completion of the implementation, an installation and usage manual has been created. The focus is clearly upon the system architecture and the installation process, as the actual usage of the system once operational should be relatively self-explanatory to those familiar with existing traffic analysers. This user manual can be found in Appendix K.

In addition to this, and in traditional UNIX programming style, “man” pages have also been created for the client and server applications. These are installed into /usr/share/man/ as a part of the installation process, and describe how to configure and start the client and server processes. These can be accessed using “man netmonc” and “man netmond” on a system that has the application installed.

It is worth noting here that the user manual makes reference to several optional installation components. The fact that database choice is optional has already been discussed, but so far there has been no mention of Dan Bertstein’s daemontools package. daemontools (Bernstein, 2001) is used to manage and control Unix services – namely, start them, stop them, and ensure they are running. Given that it is essential that both the client and server applications be operating continually in order for the system to work correctly, using a service manager such as this is only logical. daemontools claims many advantages over the traditional init.d method of service control in UNIX, but the most important difference that has instigated its selection here is its ability to monitor a service and restart it if its hangs or aborts prematurely.

The installation of additional components such as daemontools and database applications is not described in the user documentation. Users are instead instructed to refer to online installation instructions for such packages.

Chapter 5

Testing

The testing document aims to show that each of the individual requirements have been met successfully. The majority of the functional requirements can be validated with simple yes/no tests, but a small number (most notably requirements 3 and 7) require empirical testing that will be covered later. The non-functional requirements, which are often subjective, are far harder to test and will require direct user testing to ascertain success or failure.

Functional testing

Requirement	Justification for success
1	Separate modules for HTTP, HTTPS, FTP, DNS, IMAP, POP, SMTP, POPS, IMAPS, SSH, Telnet, RDP have all been implemented and tested. In addition to this, modules for IPSec, Samba/NetBIOS, MSN Messenger, IceCast, RTSP and SOCKS have also been implemented. Each module is stored as a separate .c file, which can be found in the client/libs sub-directory. Source code for these can be found in the appendix.
2	Separate modules for the dynamic protocols BitTorrent, SIP (Voice over IP) and Passive FTP have been implemented. In addition to this, a module for the Gnutella peer-to-peer application has also been crated. Again, each module is stored as a separate .c file, which can be found in the client/libs sub-directory. Source code for these can be found in the appendix.
3	See empirical testing section titled “Accuracy and Reliability”
4	The netmonc client application begins by assuming an Unknown classification for every packet, and attempts to infer a correct classification using the modules. If none is found, the Unknown classification is retained. This is shown in more detail in Figure 11 of the design document.
5	New modules can be included using the methodology discussed in the design. The dl (dynamic linking) library in C has been used to allow for a separation of modules and core code, so that new modules can be installed without recompilation of the main application. Modules may also be enabled or disabled by the user by editing the modules configuration file (typically installed to /etc/netmon_modules).
6	The detailed design clearly shows a logical distinction between client and server. Multiple clients can interact with a single server. That said, the system is flexible enough to allow a server and client to operate on

	the same physical machine. The empirical testing section also demonstrates that multiple clients may report in to a single aggregation server successfully.
6.1	Clients capture and analyse the traffic and report summaries of the flow information back to the server. As shown on the diagram in Figure 16 of the design and implementation, each client is designed to operate with a copy of the WAN traffic as an input. This methodology has been used throughout the empirical testing.
6.2	The aggregation server has been shown to receive data from multiple clients and record the information accordingly. A later empirical test of the systems performance uses two networks (Bath and Plymouth) reporting in to a single server, which demonstrates this functioning correctly.
7	See empirical testing section titled “Performance”
8	The following data resolutions have been implemented: - 1 minute averages over the past 6 hours - 5 minute averages over the past day - 3 hour averages over the past week - 6 hour averages over the past month - 24 hour averages over the past year These resolutions are visible in the historical graphs available for any host, group or application. The RRDs that store the data at these resolutions are generated using the command in Figure 19 of the design document.
9	A web based front end has been created using a combination of PHP, MySQL, CSS and HTML. A detailed design of this is available in the design and implementation section, whilst the source code is available as a part of the attached code package. The same front end has been used by users of the system to verify its functionality.
10	See below:
10.1	The front page presents a graph that accumulates traffic across all applications, split by individual groups. An option underneath this graph provides the user with the ability to view the inbound and outbound traffic levels individually.
10.2	The group level pages present graphs that summarise the traffic across the entire group, split by the different applications used. Again, an option underneath this graph provides the user with the ability to view the inbound and outbound traffic levels individually.
10.3	The application level pages present graphs that summarise the traffic for all hosts that use that application, split by the hosts themselves. Again, an option underneath this graph provides the user with the ability to view the inbound and outbound traffic levels individually.
10.4	The host level pages provide this graphing functionality by presenting all the different applications each host uses as a separate data series. Again, an option underneath this graph provides the user with the ability to view the inbound and outbound traffic levels individually.
10.5	For each application that a host uses, an individual graph depicting inbound and outbound traffic is generated. Such graphs are visible on the host level pages.
11	Below each major graph within the system an option “View historical”

	is present that allows users to view the graph over the following historical timespans: - Past day (using 5 minute averages) - Past week (using 3 hour averages) - Past month (using 6 hour averages) - Past year (using 24 hour averages)
12	See below:
12.1	All graphs include a title accurately indicting their contents. For example, the title for a specific host might be "Traffic by application for 192.168.254.200". This is specified in the command used to generate the graph images (as shown in Figure 21).
12.2	The Y-axis of each graph is labelled in bits per second (See Figure 21), as is the standard convention with network monitoring. When the numbers exceed 1024, each data point is appended with a K, and when they exceed 1024*1024 each data point is appended with an M, and so on.
12.3	The X-axis of each graph is labelled with the time in 24 hour format, with the right-most point of the axis being the current time. This view is again generated by the command in Figure 21, and can be seen on any of the graphs included in this document.
12.4	Each data series on every graph is labelled. Groups are labelled by their group name, hosts are labelled by their IP address, and applications are labelled by their application name. Labels are attached to the data series using the command in Figure 21, and the results can again be seen in any of the graphs in this document.
12.5	For each graph a minimum, maximum, average and current transfer rate is displayed for each data series. This is specified using the command in Figure 21 and can be seen in any of the graphs in this document.
13	Wherever tabular traffic data is presented, it includes at least the following different resolutions: - Current inbound traffic rate (13.1) - Current outbound traffic rate (13.2) - Past hours averaged inbound traffic rate (13.3) - Past hours averaged outbound traffic rate (13.4) - Past six-hours averaged inbound traffic rate (13.5) - Past six-hours averaged outbound traffic rate (13.6)
14	Tabular traffic data has been generated at the following levels: - Accumulated total traffic across all groups of hosts, split by application – on the front page, home.php (14.1) - Accumulated traffic per group, across all hosts, split by application – on the group level pages, group.php and group_app.php (14.2) - Accumulated traffic per application, split by host – on the application level pages, app.php (14.3) - Accumulated traffic per host, split by application – on the host level pages, host.php (14.4)
15	Groups can be defined using the "Define groups" option on the web interface. This allows the user to set, per group, a: - Name (15) - Series of IPs, IP ranges or CIDR subnet masks (15.1, 15.2, 15.3) The system supports an arbitrary combination of the above for

	maximum flexibility. Group-to-host mappings are automatically regenerated by the script “extrapolate.php” when a user modifies the group assignments
16	The alerting wizard supports all of the following requirements. A detailed description of the alerting wizards functionality is available in the design and implementation section. The alerting wizard front end is available in “alert_create_rule.php” and the cronjob that monitors the alerts is available in “alerts.php”.
16.1	The alerting wizard allows the user to define one or more applications that the rule should match, or alternatively, one of more applications that the rule should explicitly not match.
16.2	The alerting wizard allows the user to define an arbitrary number of hosts that the rule should operate for.
16.3 and 16.4	The user may specify to alert upon the following traffic levels within the alerting wizard: - Current traffic falling below or exceeding a certain rate - Past hours averaged traffic falling below or exceeding a certain rate - Past six-hours averaged traffic falling below or exceeding a certain rate - Past 24-hours averaged traffic falling below or exceeding a certain rate
16.5	The user may select a time range for which the rule is active. Furthermore, they can also disable the rule from operating at the weekend. This information is stored within the “alert” database table.
16.6	The alerting wizard stores each of the inputs the user makes, and constructs an SQL query from these (placing conjunctions between each separate term). The SQL query is then stored in the “query” field in the “alert” database table. This SQL query is, in effect, the rule. By doing this the user is able to arbitrarily combine any of the stages of the wizard to create a highly flexible rule.
17	Each alerting contact is created with the following: - A name (17.1) - An email address or SMS phone number (17.2) - A flag indicating which of the above was entered (17.2) - A flag indicating whether or not the contact is active (17.3) The contacts for the alerts are saved within the “alert_contact” database table, and tied to the alerts via the “alert_contact_link” database table.
18	Alerts are monitored by the alerting daemon (alerts.php) that checks their status every minute. Should an alert be triggered, contacts may be alerted either via email (18.1) or by SMS (18.2)
19	As discussed in the design, security has not been built in to the system as a core feature. Instead, the traditional method of securing Apache webservers (using .htaccess files) has been used. Users can optionally integrate with any security system they like – be it flat file, database or something more complex, such as Kerberos. Flat file .htaccess authentication was used and demonstrated as working in the Fluidata installation.
20	The network architecture in use (port mirroring from a switch) itself is uni-directional only, so this makes it impossible for the capture and analysis clients to interfere with the networks they are monitoring.

21	Each host level page provides the user with an option of viewing a detailed breakdown of any Unclassified traffic to and from that host. This includes the source address, destination address, source port, destination port, transport layer protocol, number of packets and total number of bytes per flow. This is handled by “Unknown.php”, which can be found in the appendix.

Empirical testing

A number of the functional requirements require empirical testing – that is, tests that measure the system against some known criteria of performance metrics. Most notably, functional requirements 3 and 7 require precisely this treatment. Empirical testing has been conducted on three distinct networks. These are as follows:

Fluidata's network

Fluidata kindly agreed to provide a live test bed for the application on their network. This was a mirrored port of a single core network feed, giving visibility of around 300 individual customers connected continuously, averaging around 35Mbps continuous data throughput. A high-level diagram of their network depicting the major switches, routers, WAN links and speeds can be found in Appendix H.

A single capture and analysis client will sit on their network, taking a port mirrored feed from their primary Ethernet switch (sw0). The aggregation server will run on the same server as well. Given that their network has a significantly higher load than the other test networks, a more powerful server has been chosen to complete this task. The specifications of this machine can be found in Appendix O. The machine was manufactured in mid-2006, so can be considered a modern machine. For reference purposes, this system will be known as "System 3" from this point forward.

Bath network

The Bath network comprises of a small student house in Bath, operating around eight active hosts simultaneously. A single 6Mbps ADSL WAN link is present, and average traffic throughput is around 0.5Mbps (predominantly HTTP and Peer-to-Peer traffic) – although it will occasionally burst to the 6Mbps limit. A diagram of the network layout can be found in Appendix I.

A single capture client sits on the network, taking a mirrored feed of WAN traffic from the PowerConnect gigabit switch. The aggregation server application (netmond) sits on the same server. The specifications of the server in question, which will be known as "System 2" from this point forward, can be found in Appendix O. This system was manufactured in 2002, so cannot be considered very modern.

Plymouth network

The Plymouth network comprises a larger student house in the centre of Plymouth. With around eighteen active hosts at any one time, an average traffic rate of 2.5Mbps flows across the network (predominantly HTTP and Peer-to-Peer traffic). This is supported by a 10Mbps cable broadband WAN link. A diagram depicting the network layout can be found in Appendix J.

A single capture client sits on their network, being fed by a mirrored port from a 16-port 100Mbps Intel switch. Unlike the other networks, this capture client does not also run as a server. Instead, this client reports back to the Bath aggregation server (the two networks are divided in to two different groups on the front end for observability purposes). The specification of the capture and analysis client in use on this network can be found in Appendix O. This system was manufactured around 1998, so could be considered a rather old system nowadays. This system will be known as "System 1" from this point forward.

Performance testing

Functional requirement 7 stated that the system must "*Be able to capture and analyse traffic at up to 100Mbps continuous data throughput per WAN link on commodity computing hardware*". Essentially, this means that each capture and analysis client needed to be able to handle 100Mbps continuous data throughput (CDT). Unfortunately, none of the capture clients in place on the three available networks involved in testing were handling 100Mbps CDT, so an alternative testing method had to be devised.

Tcpdump, produced by the authors of libpcap (which has played an important role in the developed application), has been used to capture 1GB of raw network traffic[1]. This captured traffic can then be replayed through the netmonc application all at once[2]. The time that the netmonc application takes to process 1GB of data will tell us the maximum CDT that it can handle[3].

The tests were performed on the three computer systems discussed earlier, and the results are visible in Table 4. For each test all of the modules outlined in requirement 1 and requirement 2 were enabled.

	System 1	System 2	System 3
Test 1	54.513s (140Mbps)	15.543s (491Mbps)	1.666s (4579Mbps)
Test 2	52.653s (145Mbps)	14.827s (514Mbps)	1.641s (4649Mbps)
Test 3	53.190s (143Mbps)	14.295s (533Mbps)	1.653s (4615Mbps)
Average	53.452s (143Mbps)	14.889s (512Mbps)	1.653s (4615Mbps)

Table 4 - Processing speed for a 1GB capture with all modules enabled

It can be seen that even on the slowest computer (System 1) a sustained rate of circa-140Mbps is possible, which is safely above the 100Mbps target threshold. The fastest of the three systems was able to process the packets at over 4.5Gbps – over 45 times the required 100Mbps CDT.

It was noted earlier in the design that one may opt to disable or exclude certain classification modules to increase performance. Testing this statement with the same testing methodology as before, except this time with all modules disabled, we find ourselves with the results table presented in Table 5.

[1] tcpdump was called with the following parameters:

```
tcpdump -i eth1 -s 1500 -w /tmp/capture1G -W 1 -C 1024M
```

[2] Captured traffic was replayed using libpcaps pcap_open method to open a static file rather than the usual pcap_open_live (which is used for monitoring network interfaces)

[3] The time taken was calculated using the Unix "time" command. Namely: `time ./netmonc`

	System 1	System 2	System 3
Test 1	55.690s (137Mbps)	14.386s (530Mbps)	1.630s (4681Mbps)
Test 2	52.203s (146Mbps)	14.269s (535Mbps)	1.632s (4675Mbps)
Test 3	53.190s (143Mbps)	14.339s (532Mbps)	1.633s (4672Mbps)
Average	53.694s (142Mbps)	14.331s (532Mbps)	1.631s (4678Mbps)

Table 5 - Processing speed for a 1GB capture with modules disabled

These results are at first surprising, as they suggest that passing the packets through the modules adds no significant overhead to processing time. In fact, the average processing rate decreased slightly on System 1 when all modules were disabled (although this can almost certainly be put down to an anomalous first result).

However, some thought leads us to the conclusion that with results this similar, there must still be significant CPU head-room. This in turn suggests that the CPU is not the limiting factor. Indeed, the most likely culprit is memory at some point – be it the hard disk drive, system RAM or CPU cache.

Although results for the fastest and most modern of the systems (System 3) suggest that gigabit packet monitoring is certainly viable, it is worth reminding the reader of the research conducted earlier in the Literature Survey and the fact that this test is not performing live packet capture from a network interface. The research conducted by Deri (Deri, 2004) showed that modern copper gigabit cards were not capable of transferring traffic to libpcap much beyond 600Mbps without severe packet loss. So even though the netmonc application and CPU may be able to handle it, the network hardware and kernel driver support is not quite developed enough yet. This problem was hidden from view in the tests because the methodology used a static capture file sitting on the hard disk drive, rather than capturing from a gigabit network interface that is prone to large bursts of traffic.

One final point that is worth considering is that the tests conducted here used only a single 1GB traffic sample, which was largely composed of HTTP and BitTorrent traffic (which typically involved larger packets). Should other protocols utilising smaller packet sizes have been better represented, there would have been a far higher number of packets to process. This would likely have reduced the maximum CDT considerably.

Accuracy and reliability

Requirement 3 stated that the system must be able to "*Accurately classify at least 99% of traffic, where the traffic type should be classifiable according to the protocols supported*". Essentially, this means that where it is possible for the system to classify traffic, it should be able to do it at least 99% of the time.

This is a difficult metric to test against. Traditionally an existing system known to measure traffic correctly would be used as a basis to perform tests against, but because no such system exists (within the realms of availability to us anyway), a different method will have to be used.

Instead, knowledge of the test networks has been used to devise suitable tests. For example, it is known that on the Bath network hosts with IP addresses between 192.168.254.210 and 192.168.254.219 are IP phones. Similarly, on the Plymouth network, 192.168.8.15 is known to be a BitTorrent server and thus communicate almost exclusively BitTorrent traffic.

Using this information a number of tests can be constructed that will adequately test the accuracy of the system with respect to certain protocols. These tests will take the form of raw packet captures of hosts and ports that are known to be engaging in a certain protocol. These captures will then be replayed to the netmonc capture and analysis program, and the results of its analysis will be analysed.

It is important to note that only the BitTorrent, SSH, HTTP and SIP (VoIP) will be tested using this technique. With twenty-six classification modules present in total, it would be infeasible to test each one individually in this manner. Given their similarity in construction, and the fact that they have undergone two months of real-world testing on Fluidata’s deployment, it is also unnecessary.

BitTorrent

As has been highlighted earlier in the document, BitTorrent and the growth of other peer-to-peer protocols were one of the earliest driving forces behind the project. Therefore it is of the utmost importance such traffic is correctly classified!

On the Plymouth network it is known that the host 192.168.8.15 operates with the sole purpose of being a BitTorrent server. This means that, excluding DNS queries, the only WAN traffic this host transmits and receives is BitTorrent based. 100MB of this hosts traffic was captured using the command below. It is worth noting that this traffic was captured mid-flow – i.e. after most of the handshakes have already been made. Based upon this information, it is likely that the initial results will be poor, but will show dramatic improvement over time.

```
tcpdump -i eth0 -s 1500 -w /tmp/bt.cap -W 1 -C 100M "host 192.168.8.15 and
port not 80 and port not 53 and port not 443"
```

On the Bath network the host 192.168.254.200 has been configured to communicate BitTorrent traffic solely on port 35258 (both TCP and UDP). Capturing 100MB of traffic on this host and port should result in purely BitTorrent traffic being captured. This time the traffic capture was started first, and then a BitTorrent transfer was started. This should have ensured that all handshakes were captured. This capture was setup with the command below.

```
tcpdump -i eth1 -s 1500 -w /tmp/tests/bittorrent.cap -W 1 -C 100M “host
192.168.254.200 and port 35258“
```

Results

The capture files were analysed by the netmonc client application and fed in to the database. The following results table was produced by performing a simple SQL query on the database tables.

	Correctly classified		Total transferred		Percentage	
	Packets	Bytes	Packets	Bytes	Of packets	Of bytes
Plymouth	191484	92282540	192872	93209774	99.280%	99.005%
Bath	173582	94516810	173587	94517465	99.997%	99.999%

Table 6 - BitTorrent detection rate

The first thing to notice from the above results is that in all cases the 99% accuracy goal is exceeded. However, the observant reader will notice that in the Plymouth case this target was only just met. The reason for this was hidden away in the testing description – the captures on the Plymouth network were started mid-transfer, well after the BitTorrent handshakes had taken place. Because it is the handshakes that are used to identify flows, it is to be expected that any sessions already underway would not be identified. Fortunately, BitTorrent sessions are relatively short-lived (typically two hosts will not communicate without re-handshaking for more than an hour), so the percentage would have risen slowly over time.

The Bath network was not affected by this issue because the capture was started before the BitTorrent transfer. Thus all handshakes were captured, meaning that a far greater percentage of packets were correctly classified.

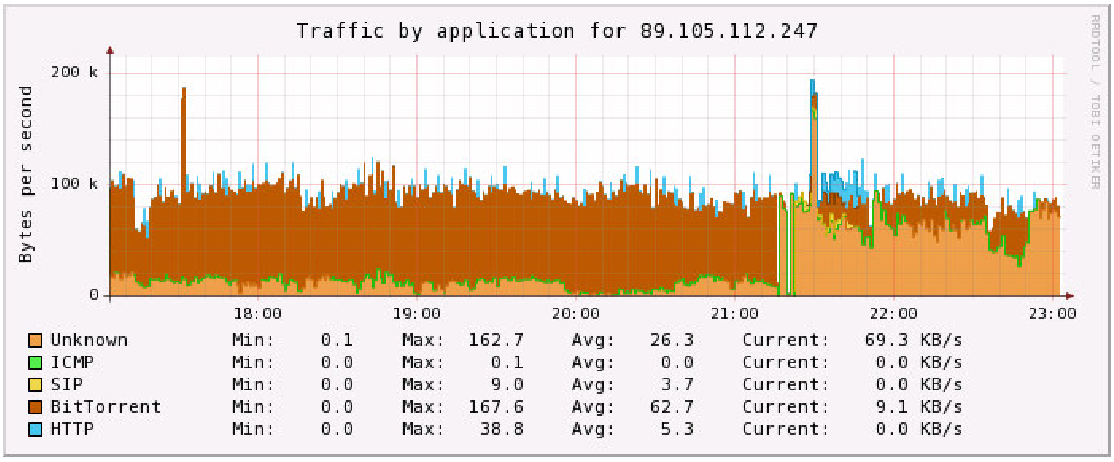


Figure 26 - Traffic graph depicting loss of flow information upon system restart

The above traffic graph from the system demonstrates this issue well. A large amount of BitTorrent traffic is being monitored on the network until around 21:15, at which point the client application was restarted (thus the flow information was lost). After this point a great deal more Unknown traffic was recorded, which is most likely BitTorrent traffic that could not be immediately identified due to no handshakes being available. Slowly, the ratio of BitTorrent to Unknown traffic begins to swing in favour of BitTorrent again, until the client is restarted once more shortly before 23:00.

SIP (Voice over IP)

SIP is a VoIP (Voice over IP) protocol that has grown massively in popularity in recent years, a fact that can most likely be attributed to a similar growth in broadband take-up. It has already been said that on the Bath network it is known that all hosts between 192.168.254.210 and 192.168.254.219 are IP phones running the SIP protocol. Therefore, these devices should *only* ever communicate SIP (In fact, this is not quite true; they will occasionally use TFTP to check for firmware upgrades, but this will be a negligible amount of traffic).

Capturing the traffic for three IP phones on the Bath network was conducted using the command detailed below. A phone call was then placed between two hosts for a short period to accumulate enough traffic to fill 10MB.

```
tcpdump –i eth1 –s 1500 –w /tmp/tests/sip.cap –W 1 –C 10M “host
192.168.254.210 or host 192.168.254.211 or host 192.168.254.213”
```

Results

The following results were generated by running the capture file through the client application:

	Correctly classified		Total transferred		Percentage	
	Packets	Bytes	Packets	Bytes	Of packets	Of bytes
Bath	35582	7202392	35584	7202464	99.994%	99.999%

Table 7 - SIP detection rate

With only two packets communicated by the IP phones that were not classified as SIP, this test can clearly be marked as successful. These two packets appear to be TFTP packets too, as they used the common TFTP port number. If a classification module was written for TFTP, the classification figure could easily be driven to 100% in this case.

HTTP

HTTP, the protocol most commonly associated with web traffic, plays an important role in the majority of modern networks. Although HTTP can operate on other ports, it is most commonly associated with port 80, and for the purpose of these tests, capturing 100MB of port 80 traffic will almost certainly be exclusively HTTP traffic.

Capturing 100MB of port 80 traffic on the Bath network was achieved using the following tcpdump command:

```
tcpdump –i eth1 –s 1500 –w /tmp/tests/http.cap –W 1 –C 100M “port 80”
```

Results

The following results were generated by running the capture file through the client application:

	Correctly classified		Total transferred		Percentage	
	Packets	Bytes	Packets	Bytes	Of packets	Of bytes
Bath	110538	96262374	110538	96262374	100.000%	100.000%

Table 8 - HTTP detection rate

The above results show that the system was correctly able to classify every HTTP packet from the 100MB sample passing through the system.

It should be noted that the capture also detected 4 other packets operating on port 80. However, these were found to be communicating with 192.168.254.200 on port 35258 – which you will remember as being the test configuration for BitTorrent transfers in Bath. Clearly some end user has configured their BitTorrent client to operate on port 80 in an attempt to defeat firewalls and/or proxy servers. The system was able to correctly identify this though, and classified the packets accordingly.

SSH

SSH was invented as a secure replacement for the plain-text (and thus insecure) Telnet protocol. All communications are encrypted, and it is the principle means for remotely controlling servers, switches, routers and other sensitive devices. Furthermore, SSH also allows you to tunnel other applications and file transfers via it. This is commonly used within the Bath network to work remotely with the University of Bath's network.

Capturing 100MB of SSH traffic between the Bath network and the University of Bath was achieved using the following tcpdump command:

```
tcpdump –i eth1 –s 1500 –w /tmp/tests/ssh.cap –W 1 –C 100M "host
sshgate.bath.ac.uk"
```

Results

The following results were generated by running the capture file through the client application:

	Correctly classified		Total transferred		Percentage	
	Packets	Bytes	Packets	Bytes	Of packets	Of bytes
Bath	117348	96462336	117348	96462336	100.000%	100.000%

Table 9 - SSH detection rate

As with HTTP, every single SSH packet to sshgate.bath.ac.uk was correctly classified. There were no anomalies present in this test.

Summary of empirical testing

The two core requirements (3 and 7) that required empirical testing have been exhaustively tested and have been found to pass in all cases. In nearly all cases the targets were met by a very large margin, suggesting that there is room for significant growth within the system. Indeed, a modern server (System 3) was found to be able to process 45 times the 100Mbps requirement, and averaged in excess of a 99.99% correct classification rate – far better than the required 99%.

The system has also demonstrated resistance against end-users attempting to disguise their activities as being that of another application by simply modifying the port numbers in use. This was most notable in the HTTP test case where one user had configured their BitTorrent client to operate on port 80, imitating an HTTP server. The netmonc analysis application correctly recognised this though and classified the packets as BitTorrent (by using knowledge of an earlier stream).

Non-functional testing

Testing the non-functional requirements is a harder process, as these are precisely the requirements that cannot be measured by simple metrics – they are often subjective. For this reason, testing in real-world environments and user feedback often plays a key part in the validation of the non-functional requirements. This was one of the key motivations behind seeking the involvement of Fluidata.

Usability testing

Traditional methods for examining the usability of a system would include sending out questionnaires or conducting observations of user interaction. However, because the system is designed for specialist use, such general purpose methods do not really apply. Instead the focus will be placed upon users that have a stake in the system, with the primary attention being given to Fluidata Limited, who have been using the system on their live network for approximately two months.

User feedback has been key to the ongoing development of the application. As can be seen from the latter stages of the design document, user feedback prompted numerous changes to the front end and other components of the system. Fluidata have been the most vocal with such change requests, but contributions have also come from users on the Bath and Plymouth networks too.

Fluidata’s final feedback, as presented in Appendix Q, states that the system is “simple and intuitive to use”. The grouping and zooming features, which Fluidata had been a strong proponent of earlier in the project, were also triumphed as being a great help to usability in their feedback.

However, they did raise one serious usability concern that hampered them testing the alerting component of the system. Earlier in the project Fluidata had been keen to see automatic trending of network data to adjust alerting rules. This, as stated in the requirements and design, has been left for future work and is discussed at length in the Future Work section. Ultimately, Fluidata felt that the lack of this feature meant that too much effort would be required to provide significant test coverage of their network. Going in to the design stages it had already been decided to pay less attention to the alerting system, given that it was not the focus of the project. To have this flagged as the only significant usability issue was therefore quite pleasing.

Operation on commodity hardware

The performance testing for functional requirement 7 was performed on a range of computer systems, ranging from nearly 10 years old to only 6 months old. All of these systems use the standard x86 architecture and no specialised components that would not normally be available off-the-shelf. Given that all of these systems were able to satisfy all of the functional requirements, it can be said that the system does indeed operate on commodity hardware. Thus non-functional requirement 25 has been fulfilled.

Stability and reliability

The bulk of the programming of the system was conducted between the beginning of December 2006 and the end of February 2007. The nature of the system requires it to run constantly in order to gather traffic information correctly. Since early January 2007, the system has been running 24 hours per day, seven days a week on the servers in both Bath and Plymouth. In early March 2007 the server in Fluidata's London datacenter came online and, as of 25 April 2007, has been running continually without error.

The reliability of the system is harder to test. At the time of writing, no reliability issues have been reported by the users of the system (at present there are approximately 6 users spread throughout Bath, Plymouth and Fluidata in London). A reliability issue would be defined here as meaning an inaccessible page, the server timing out, incorrect data being reported, alerts not being triggered when they should be, servers crashing, and similar issues in the same vein.

Whilst it is impossible to say whether a system is completely reliable or stable, over the past few months the system has been used by an experienced user-base who have had no issues with its reliability, or stability. This is sufficient for the purposes of this test. Thus it can be said that non-functional requirement 27 has been met.

Accounting for future improvements

Accounting for future improvements to the system is clearly an important requirement, as demonstrated by the amount of future features specified in requirement 28.

The first such desired feature (requirement 28.1) would be support for gigabit links. This is discussed in depth in the future work section of this document, so it will suffice to say that testing has already shown that gigabit speeds are possible; the limiting factor is now the Linux kernel and network card drivers (as discussed in the Literature Survey, the Linux kernel is currently incapable of capturing at wire-speed gigabit). There is no limit whatsoever imposed on the operating speed of the application itself. In theory, given a fast enough system, it would even be able to support 10Gbps network links.

The trending feature (requirement 28.2), as suggested by questionnaire respondents, is also discussed at length in the future work section. Ultimately though, the data required for the trending process to be carried out (presumably using standard deviation techniques) is already available within the MySQL database. It would of course need to be sampled and averaged, but this is all feasible given the functions supported by MySQL or indeed any other database system. The alerting system has also been constructed almost completely separately from the main traffic reporting application, so it should be very easy for a future developer to drastically change how alerts are generated without having to impact the operation of the main system.

Support for additional alerting mechanisms, as specified in requirement 28.3, is certainly something that a competent programmer would be able to implement with relative ease. At present only SMS and email have been implemented (as per requirement 18). Support for additional alerting mechanisms would require changes in only two places:

(1) On the front end where contacts are defined – an additional alerting method would have to be added to the drop down options.
(2) A function to handle this new alerting type in the alerts.php cron job that runs each minute.

Supporting different databases on the back-end is something that was originally envisaged as being a challenging task, but has turned out to be quite trivial. The method of achieving this, as discussed in the server design already, simply involves having the server program output SQL INSERT statements to STDOUT and then using UNIX pipes to redirect the SQL queries to a database of the user's choice. Instructions have already been provided in the design for using MySQL and PostgreSQL, although any database should be possible.

Modelling traffic outside of the local network was another future feature that one questionnaire respondent hinted at. Technically this is already possible, as one could simply define an external range of IPs as being one of the groups that should be monitored. As was stated previously, this is not the intended use of the grouping feature, but it is certainly possible as the system uses only this to differentiate between external and internal hosts. What is really meant by this requirement though is the ability to report on the most popular traffic destinations. Fluidata stated in their questionnaire feedback that being able to do this (by AS number in their example) would allow them to arrange peering agreements with the most popular destinations, thus saving on bandwidth bills.

A competent programmer could certainly implement this additional functionality. The destination hosts can already be found in the database, so any additional functionality and front-end work could be built around this. This would most likely involve performing WHOIS queries on the destination IPs and caching their AS numbers. Of course, with a very busy network this would generate hundreds of thousands of WHOIS queries, so it may be easier for the future developer to seek a replicated copy of the WHOIS database.

The final future requirement that had to be accounted for was the possibility that ISPs may wish to present certain graphs and data from the system to their end-users via their own web-based portal. This requirement came directly from a discussion with

Fluidata, who already provide traffic graphs and usage data to their end-users via an online portal. Indeed, this requirement has certainly been met for the following two reasons:

(1) The data required to display application usage and total traffic data is already available in the database for every host that is monitored. Another application could easily connect to this database and redisplay the data in any format that the programmer chose.

(2) The graphs could also be easily redisplayed on other websites. Each graph is stored using a logical naming convention. For example, the total application summary graph for 192.168.254.200 that accounted for traffic in both directions would be called "192.168.254.200_both_app.png". This could simply be referenced by the other web portal to provide graphing capabilities too.

In summary, the use of readily available programming languages and database engines (such as C, PHP and MySQL) has enabled the addition of future functionality and subsequent fulfilment of non-functional requirement 28 with relative ease and minimal effort. Furthermore, no components of the existing system would have to be redesigned or re-implemented to account for such changes, meaning that new components could be added and removed at will – without impacting any current functionality.

Testing optional requirements

Three optional requirements were originally specified in the requirements document (See requirements 22, 23 and 24).

Trending of traffic data (Requirement 22)

Requirement 22, the trending of network traffic to produce a more accurate alerting system, has been left for future work. Whilst this was clearly a very useful feature, it was deemed to be beyond the focus of the projects main goals. Coupled with the fact that it would require considerable extra design, development and testing, this was left as future work. A more detailed discussion of how this could be achieved can be found in the “Future work” section.

Handling of non-TCP/UDP protocols (Requirement 23)

Requirement 23 focussed on the capture and analysis of protocols that did not sit atop TCP or UDP. As noted in later sections of the client application design, this became a core requirement when it became clear that much of the traffic on Fluidata’s network fell in to this category. Support has been built in to the application for the following additional network protocols:

- ICMP (Used in Ping, Traceroute, and so on)
- IPSec (Typically used by VPNs)
- GRE (Generic Routing Encapsulation, as used by some Cisco VPNs)
- IEEE 802.11Q (As used by VLANs)
- PPP (Point to Point Protocol)

Support for all of the above was required in order to capture all of the traffic on Fluidata’s network. GRE and VLAN support is clearly demonstrated as working, as all of Fluidata’s traffic uses these protocols above the Ethernet layer. To not support these would have resulted in no traffic being captured at all! Functioning support for ICMP, IPSec and PPP is demonstrated by the systems ability to record traffic for these protocols, which is depicted in the graphs shown in Figures 27, 28 and 29:

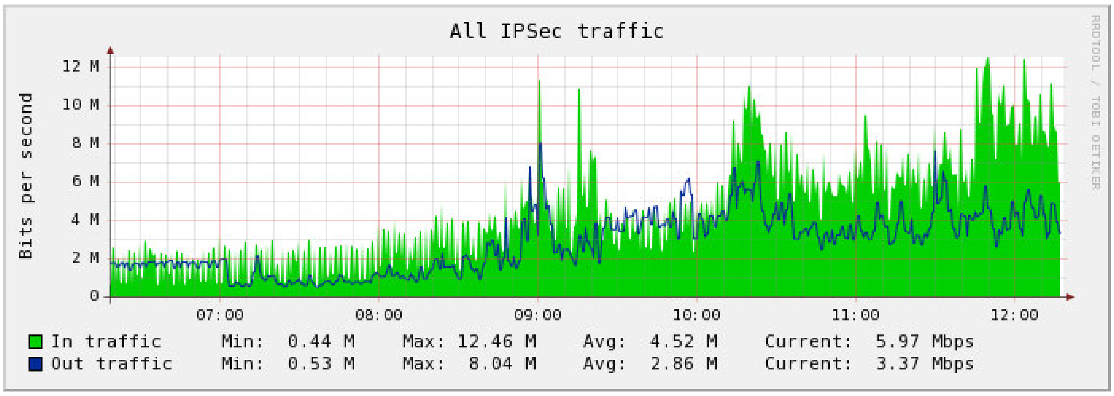


Figure 27 - Traffic graph depicting presence of IPSec traffic

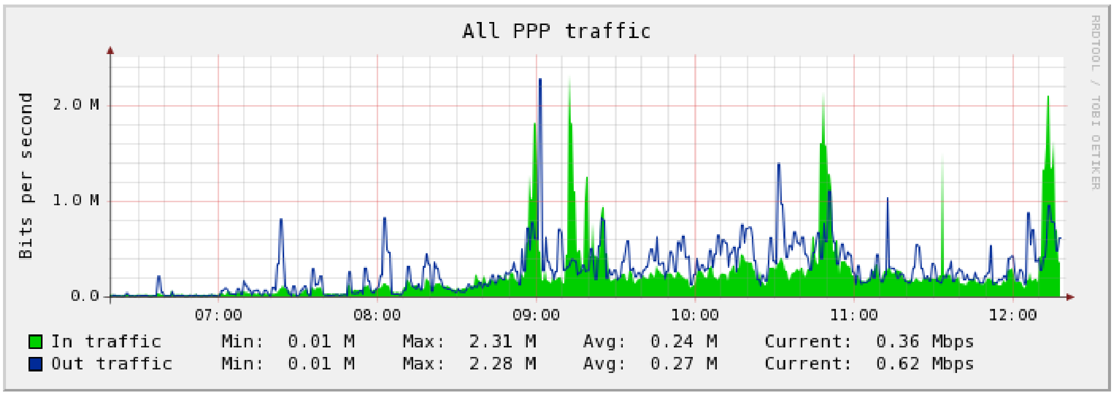


Figure 28 - Traffic graph depicting presence of PPP traffic

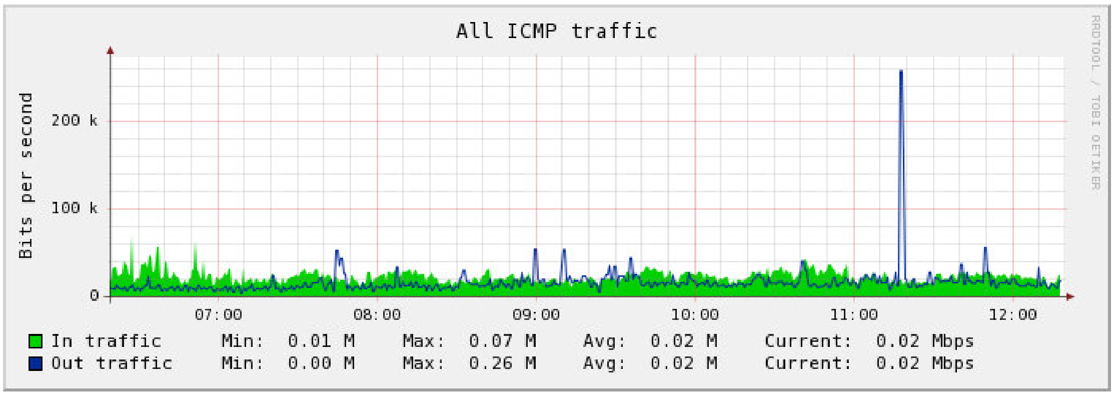


Figure 29 - Traffic graph depicting presence of ICMP traffic

Guaranteed alerting mechanism (Requirement 24)

The final optional requirement (requirement 24) dealt with a request of Chris Rogers from Fluidata in his questionnaire response. He requested a guaranteed method of alerting support contacts, primarily because SMS and email were not guaranteed forms of communications. This requirement was entered as optional because it was clearly not in line with the projects main goals. Indeed, this has not changed during development, so the requirement has remained unfulfilled. This requirement will now be left as future work.

Real-world testing

Fluidata Limited provided a suitable platform upon which it was possible to test the system in a real-world environment. This consisted of a gigabit feed from their core switch (as depicted in Appendix H) to a capture and analysis server hosted in their data centre (system 1, as described in Appendix O). This system housed both the client and server applications, as well as the database and front end. A deployment with more resources would likely seek to separate these entities further.

The feed from Fluidata consisted of a single gigabit link, connected to the secondary Ethernet port of the server. During working hours on a weekday, traffic over this connection averaged around 100Mbps continuous data throughput (circa 20,000 packets per second). During the evenings and weekends this figure dropped significantly to around 35Mbps.

The original system was unable to detect any packets on the network when first deployed. This was due to it not being built with support for VLANs or GRE (Generic Routing Encapsulation). As discussed in the design, support for these protocols was soon incorporated and testing was able to resume.

Unfortunately two further issues prevented the system from operating correctly straight away. Firstly, the processes that average the data in the database over the desired resolutions (1-hour, 6-hour and 24-hour) were unable to handle the load within the time-frame required. These processes took in excess of nine minutes to execute – clearly unacceptable when they had to be run once every minute. As discussed in the design, modifications were made to better utilise the storage and functionality of the RRD format. This saw the execution time fall to within twenty seconds. Secondly, graph generation could no longer be conducted for every host on the network every minute; there were simply too many. Instead, the graph generation scripts were modified to merely record the commands that *would* be used to generate the graphs, and then only call those commands via the front end when a user requested the graph.

It should be noted that it was not the volume of data flowing over Fluidata's network that the early versions of the system had trouble handling. Instead, it was the amount of unique hosts connected identified on their network. Previous testing on the Bath and Plymouth networks had involved no more than 25 hosts at a time. Fluidata's network had over 1,000 hosts communicating on a normal day. This massive increase was the biggest contributor to the load-problems experienced.

Once these changes had been completed, and a few of the front-end database queries had been optimised, the system operated as specified.

A number of modifications to the detection modules were made to account for the type of traffic flowing over Fluidata's network. Whereas the Bath and Plymouth networks that had been the subject of previous tests were dominated by peer-to-peer traffic, Fluidata's network was primarily in use by businesses that had no call for this kind of application. Instead, their traffic was found to contain predominantly web and IPSec (VPN) traffic. As discussed in the design, this necessitated the implementation of support for VPN traffic classification; something which until this point had been considered an optional extra (See the previous section discussing Optional Requirements). In addition to this, support for PPP (Point to Point Protocol), IceCast, SMB/NetBIOS and SOCKS application protocols were implemented. Support for each one of these was simply implemented as a module, requiring no recompilation of the main application. This is precisely how the module system was intended to be used, and the fact that it could be used in this manner in a real-world environment validated the additional design and implementation time taken.

However, it should be noted that despite the best efforts to classify as many protocols on Fluidata's network as possible, there still remained an elusive 5-10% of traffic that could not be classified by the system. This traffic operated predominantly on seemingly arbitrary ports, but originated from only a relatively small number of hosts and almost always used UDP as a transport method. This suggested that the data was most likely encrypted VPN traffic - an assumption which has since been backed up by Chris Rogers, Operations Director at Fluidata Limited (See Appendix N).

Throughout the two months that Fluidata were testing the live system, Chris provided considerable feedback (summarised in Appendix M and N) that introduced new requirements and helped shape the finished implementation. In some cases they also cited instances where the system was useful in helping them diagnose and resolve a problem that their existing systems could not help with.

Perhaps the single most interesting problem that was resolved occurred on Easter weekend 2007 (See appendices P and Q). In this instance a suspicious outbound traffic spike was observed, which contrasted sharply against the background of very-low traffic over the long Easter weekend. Fluidata's traditional network monitoring system, Cacti, could not identify the true source of the traffic. However, they were able to use the system developed here to observe that one server in particular (89.105.96.125) was sending vast amounts of FTP data (peaking at 14Mbps) to external hosts. As a result of this, Chris Rogers was able to identify that the server in question had an unsecured anonymous FTP server running (which was being used by people to dump and serve large files). This was soon secured and the traffic ceased. Although Chris almost jokingly adds that they "would never have tracked that down" without the system, this is in fact quite plausible. An account of the event can be found in an email from Chris Rogers, in Appendix P. Similar reports of other instances where the system was used to help with a real-world problem can be found in Appendix Q.

This event backs up the claim made early in the Literature Survey that SNMP based network monitoring (as Cacti uses) is insufficient for tracking down problems such as this. More importantly though, it highlights the fact that the system developed here has a real-world use that is not already satisfied by many network monitoring systems (looking back at the questionnaire response in Appendix B, Fluidata were using Cacti, MRTG, and a custom RTG/RRDTool development). To have problems like this exist in the network for any length of time would not only cost them financially (in terms of bandwidth), but it may also cause to damage to their reputation.

Testing summary – Referring back to the specification

The original goal of the project was to create a distributed traffic analyser, capable of operating on high speed networks using only commodity hardware. The functional requirements specification firmly grounded these parameters (and many others). All of these requirements have been demonstrated as having been met in the sections titled “Functional Testing” and “Empirical Testing”. The majority of the functional requirements were met simply by referring back to the design and implementation, and showing what part had satisfied the requirement. The few functional requirements that required empirical study were tested thoroughly against pre-defined metrics in the “Empirical Testing” section. Here it was shown that the system was capable of exceeding both accuracy and speed requirements by a large margin, even on older hardware. The testing conducted on Fluidata’s network (which, at times, pushed the system beyond specification) has shown that even under real-world conditions the system is still able to perform within parameters.

Extensive user involvement from Fluidata throughout development has helped ensure that the system both met their needs and is usable by specialists in the field, such as themselves. In numerous instances the system proved its worth by helping determine the root cause of network issues (see the section Real World Testing, and appendices P and Q). The system was described by them as “simple and intuitive” to use, and only one significant flaw was identified (the alerting systems lack of trending).

Whilst being harder to test than their functional counterparts, the non-functional requirements have also been met. This has been discussed in depth in the “Non-Functional Testing” section. Having successfully operated on hardware nearly 10 years old the system was deemed to have met its target of working on commodity hardware with ease. Furthermore, its stability and reliability were demonstrated through its continued operation and use without error for over two months.

One of the three optional requirements has been implemented and tested too, as demonstrated in the section “Testing optional requirements”. This requirement, which involved capturing and analysing non-TCP/UDP traffic, ultimately proved essential on Fluidata’s network. To not do it would have meant missing a sizeable quantity of their network traffic, thus invalidating much of the real-world testing.

In summary, the completed system has passed all of the tests, met the original specification, and performed well in a real world environment. Therefore it can be said that the system has tested successfully against its original goals.

Chapter 6

Future work

Encrypted peer-to-peer traffic

Encrypted protocols have been operating on our networks for almost as long as the networks have existed themselves. HTTPS, SSH and all other SSL traffic is encrypted, with the net effect that packet contents cannot be inspected. The headers, however, remain unencrypted. These packets can still be classified though, as they almost always operate on their default ports (e.g. port 443 for HTTPS, 22 for SSH, and so on) and often include an unencrypted handshake that details their protocol name and version number.

However, some newer encrypted protocols are going out of their way to avoid exactly the classification that this system aims to provide. The BitTorrent protocol has had an unofficial addition to its specification since late 2005 that specifies an encryption scheme for the protocol (Azureus Project, 2005). This scheme, developed jointly by the major BitTorrent clients, encrypts all traffic between client and server using RC4 encryption – including the handshake messages that are used to classify the stream as BitTorrent.

Fortunately, the packet headers are still available, thus rendering the encryption scheme less than foolproof. For example, the client could capture an un-encrypted handshake between two BitTorrent clients – hostA:portA and hostB:portB (by the process discussed earlier in the design, we have now flagged hostA:portA as BitTorrent and hostB:portB as BitTorrent). Now suppose that hostC:portC initiates an encrypted session with hostA. This encrypted session would still be handled by hostA on portA – which we have already classified as being BitTorrent. Thus it can be inferred that the entire stream is BitTorrent, without ever having to decrypt the encrypted messages from hostC. With BitTorrent clients now set to use encryption where possible by default, it is clear that this method works, as over 99.999% of traffic was successfully classified during testing of BitTorrent traffic.

However, this flaw in the client's communication design can easily be overcome by randomising the ports for encrypted transfers. This would render the entire method of tagging hosts and ports with a specific application useless, as the data could not be examined to find the next host:port pair (because the traffic would be encrypted).

There is an alternative solution though. Madhukar et al (2006) present their "Transport-Layer Method". Essentially this uses heuristics and knowledge of how the peer-to-peer streams operate to classify them. Madhukar et al highlights three key steps the Transport-Layer classification process:

1. If a source-destination IP pair is concurrently using TCP and UDP on non-well-known ports (>1024), classify the stream as peer-to-peer. BitTorrent follows precisely this characteristic, with control messages and older clients using TCP, and newer clients using UDP for raw data transfer.
2. Look for IP-port pairs where the number of distinct connected ports matches the number of distinct connected IPs. Classify such IP-port pairs as transferring peer-to-peer. Use simple heuristics to aid in this process (e.g. if the number is less than 3, then it is far less likely to be peer-to-peer than if the number was over 20).
3. Do not flag up well-known-ports as peer-to-peer, or packets where the sizes do not fit well with known peer-to-peer protocol designs.

This method has a number of advantages. Firstly, it is able to classify peer-to-peer traffic without requiring or inspecting packet payload. This not only gets around the encryption problem, but it also circumvents any legal issues on networks that do not permit payload inspection, and also reduces computational overhead (as only the header needs to be inspected). Secondly, it should be able to correctly classify future peer-to-peer protocols without any changes. This is because it merely relies upon the behaviour of the protocols acting in a similar way – i.e. a single client communicating with lots of separate hosts on different ports all simultaneously.

However, there are also a few concerns with this method. Firstly, it cannot distinguish between different peer-to-peer protocols – it simply classifies everything as peer-to-peer. Secondly, whilst the results of testing by Madhukar et al (2006) are very positive, its accuracy under real-world conditions with users disguising their peer-to-peer clients with well-known ports and other such tricks remains to be seen. It is the suspicion of the author that accuracy would be significantly less than that of the signature analysis method with larger and more real-world datasets, but that suitable heuristics could be applied to improve detection rates considerably.

Given more time and resources, the intention would be to implement a modified version of the Transport-Layer Method. This would not replace the existing signature analysis method in use already, but would be used in conjunction with and as a failover for the signature analysis method. The three core processes highlighted earlier that form the method would all be implemented as a part of the application detection scheme (appdetect.c), most likely using additional hash tables to provide fast lookups from a given host: port pair to all other host:port pairs that they are communicating with.

However, the transport layer method would not simply be used as a failover method if the signature analysis method failed to determine an application. Suppose that hostA was already known to be communicating a large amount of BitTorrent data, and then a new stream was detected that could not be classified by the signature analysis method, but the transport layer method was able to classify it as peer-to-peer. Using the knowledge that hostA was already active in BitTorrent, one could infer that the generic peer-to-peer traffic that the transport layer classified was in fact most likely BitTorrent.

Because this new method can only make a guess at the applications that generated the packets, it would be wise to introduce some simple probability metrics too. So, for example, port based analysis may infer that a packet flowing through 80 is HTTP with 75% probability, but the transport layer method may determine with 90% probability that the packet is actually BitTorrent (using the processes and heuristics described earlier). Ultimately, all modules and classification techniques would have to generate a probability, and the one with highest value would 'win' and classify the traffic.

Monitoring VoIP call quality within a distributed network

After years of promise, Voice of IP services (commonly known as "VoIP") have finally reached the mainstream, with large ISPs such as Orange and BT providing such services to consumers. A common complaint with VoIP services is the "jitter" one may receive on the line as call quality degrades. This in turn is often caused by packet loss or high latency at some point in the path between the two end-points (i.e. the phones). Whilst this is largely a non-issue for home users at the moment, businesses will demand a higher call quality.

Following a brief discussion via email, a meeting was arranged with Peter Gradwell, Managing Director of Gradwell dot com Limited. One of Gradwell's key business areas is providing VoIP (Voice over IP) services to upwards of 4,000 customers (predominantly businesses) (Gradwell, 2006). Handling support requests relating to call quality issues is a commonplace problem, and at present is a very hard problem to track down. Whilst it is usually the customer's ISP or their own internal network at fault, there is always the possibility that it could be a fault or capacity problem on the Gradwell network itself. To make matters worse, Gradwell's network is far from small – it is composed of multiple Asterisk servers (the open-source VoIP server), and a plethora of switches and routers. Tracking a single VoIP call through the network is a formidable challenge itself. With multiple entry and exit points, there are a wide number of routes a call may take; a fact which makes tracking a fault down on one particular link even harder.

The system developed as a part of this project is already distributed and can already recognise SIP traffic and the resulting RTP (UDP-based) audio streams. However, a number of significant modifications would be required in order to make the system capable of monitoring call quality. These include:

- Specialising the system to handle VoIP calls (SIP in particular) only. This would increase potential throughput the system could handle.
- Monitor each call separately and for each produce an estimate of call quality based upon packet loss, errors, latency, bursts and the resulting jitter.
- Correlate call quality information from multiple sources to produce a graph of how call quality degraded (if at all) whilst traversing the network.
- Support the recognition of network schematics, thus allowing the system to understand call routing and therefore estimate the location of faults within the network.

Much research has already been conducted on measuring call quality (Cole et al, 2001; TIA, 2006), but so far this has not yet been applied to monitoring calls from multiple points in the network and then correlating the results. The main challenge here would involve the tracking of hundreds, possibly thousands, of simultaneous calls, each taking different routes through the network and Internet. Although each could, in theory, be uniquely identified and tracked by the "Call ID" (an MD5 hash of all of the call parameters), correlating all of this data in real-time to produce accurate measures of call quality would likely require far more computing resources than used within this project.

Installation process

The sheer size of the installation section of the user manual (Appendix K) highlights the need for a more streamlined installation process. At present the user has to unpack the source code to the machines, compile it, install it and then optionally configure the database, Apache and any start-up scripts. For a novice user this process would be daunting to say the least, and may even result in them deciding not to continue with installation.

Ideally, a simpler method of installation would be included. This could take the form of an RPM (Red Hat Package Manager) file that would install the compiled components to their standard locations, and then prompt the user for the following configuration options:

- Database type, host and associated login details
- Which network interface to capture
- Which server and port to report to
- Which port to operate the server on
- Whether or not to run the system on start-up

It should be noted though that the more advanced users may object to an over-simplification of the installation process. For them, a more manual process may be preferable, as they could opt to leave out certain components, use a different database, install to different locations on the drive, and so on. Therefore the simplified and automated installation process would be made available alongside the present one - not instead of.

Trending to improve alerting system

The ability to learn from and trend existing traffic data to improve the reliability and accuracy of the alerting system was suggested by all of the original questionnaire respondents in the requirements elicitation process. Rather than setting fixed throughput limits, as is currently done within the alerting system, one would simply define tolerances for each rule. Providing the rule in question fell within tolerances (deviated from the norm), the user would not be alerted.

It is the definition of this "norm" that is the interesting and challenging thing here. Traffic varies according to the times of the day (9am-5pm typically being busiest for business ISPs, 5pm-10pm for consumer ISPs), and traffic will likely increase steadily over time (assuming the ISP in question is growing and taking on more customers). Despite all these variations in traffic, one would still want this kind of traffic usage to be defined by the norm. For example, if you owned an ISP with a handful of customers, but were taking on a steady few extra customers each month, you would want the system to account for this – and not have to redefine the rule each month.

The following traffic graph, taken from the first 10Gbps interface of Lark – one of the core switches of the University of Bath, demonstrates the regularity of traffic usage over the period of one week.

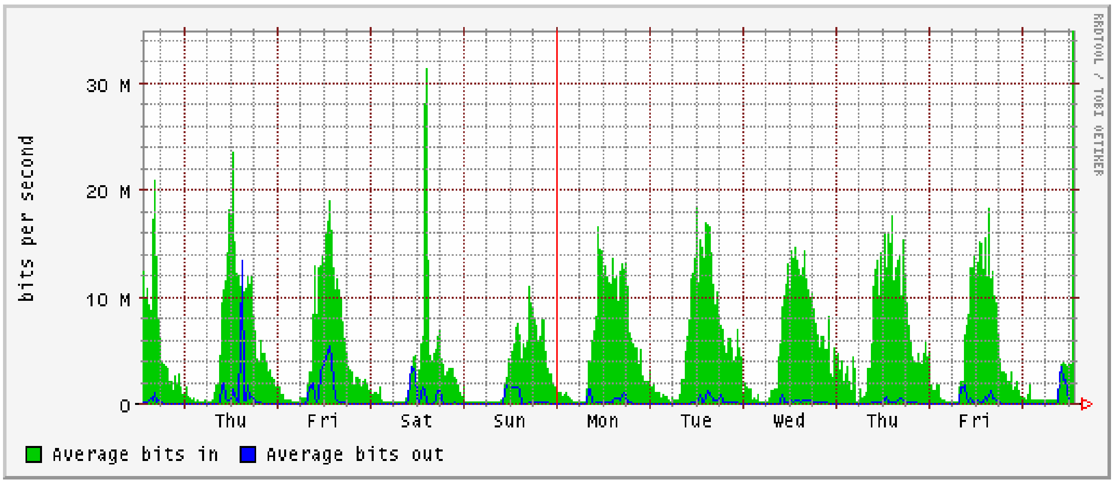


Figure 30 - Traffic graph from 10Gbps interface on lark.bath.ac.uk

Note that the traffic dips considerably during the weekend, except for a massive spike for a short period on Saturday (the latest spike on Saturday is visible on the far-right of the graph). Applying trending to this data, we might find a diagram similar to the following (Please note that the following is a sketch only and by no means accurate).

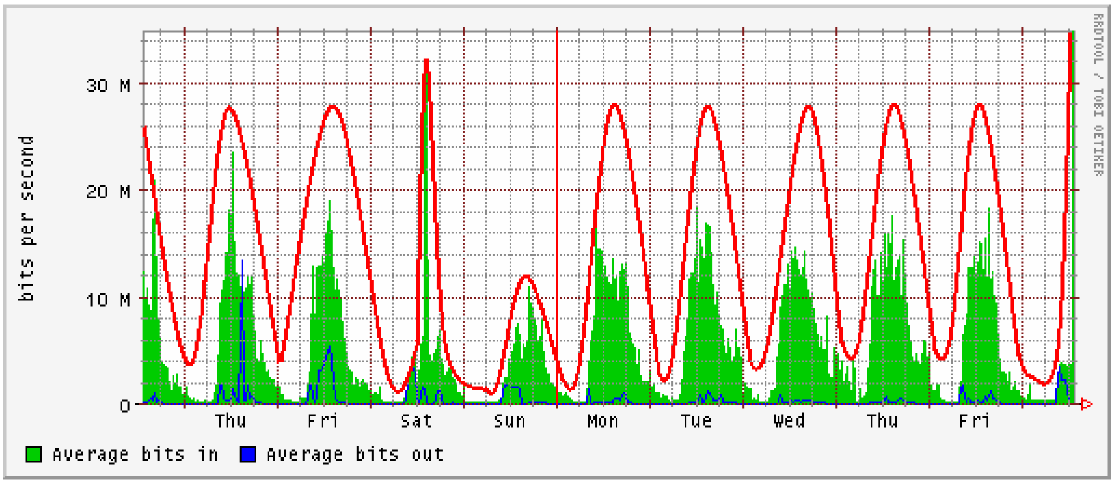


Figure 31 - Trended data with thresholds for alerting

Note the red line – depicting the trend line for the inbound data rate, with an artificial threshold applied (Outbound trend line has not been drawn for clarity). Should the inbound traffic exceed the limits defined by the red line, an alert would be triggered.

The alerting system can therefore be split out in to two separate processes: (1) the alerting process (which has already been seen) and (2) the learning/trending process that defines the norms. Both processes would be run in tandem – so the trending process would be constantly adjusting the norm to account for slight changes in traffic. Of course, large changes should continue to trigger alerts and not be recorded as a new norm.

It is suggested that norms would be stored as a block of time-series data. Traffic data would be sampled from the existing database, averaged and then recorded at specific intervals within the time series. These intervals would require some consideration – previously 1 minute intervals have been used for the alerting system, but if this were to be done here and norms were recorded for *every* host for a full day, there would be 1440 entries in the database per day per host. With a thousand hosts, over a million entries per day would be generated. This also assumes that only a single day's data is recorded – we may wish to trend over an entire week. This would allow users to watch for any abnormal traffic at the weekend – something a business ISP would almost certainly want to do. Of course, one optimisation would be to only trend for the hosts that alerts had been configured for. This would dramatically reduce data storage requirements, but it would also mean that we would incur a lead-time (for learning in this case) before the alert could be made active.

The actual learning process would be relatively simple though. One could sample the data at this interval, and if a new value fell within the standard deviation of the rest of the data, the new value would be averaged with the previous value for that point in time and stored. If the value fell out of range though, then an alert should be generated. It should be noted that calculating the standard deviation over thousands of values, for hundreds or thousands of hosts, every minute, would also likely generate a computational problem that required attention. An alternative to using mean averages combined with standard deviation would be to employ a technique such as wavelet analysis (Joseph et al, 2004), normally only found in graphics applications. This is well suited to approximating complex-signals with sharp discontinuities and would likely yield significant data storage savings, but at the cost of increased complexity and processing time.

Clearly there are data storage and computational issues at the heart of this work. Just as clear though is the fact that users of the system would find this feature incredibly useful – Fluidata even described it as "vital" in a phone conversation when reviewing an early version of the system. Given more time and resources, it is certainly something that would be implemented in the future.

However, it should also be noted that in Fluidata's final feedback (Appendix Q), discussion is made of detecting "viruses, SPAM and DoS attacks". Whilst this would certainly be a feasible addition to the system, it is perhaps a task more suited to a dedicated network IDS (Intrusion Detection System) such as Snort. Snort uses rules (which are updated daily) to provide coverage and protection from the very latest network attacks. Since this is leaving the realms of network traffic monitoring, it has not been discussed further as potential future work.

Capture/analysis client support for gigabit links

The requirements specification stated that each client only needed to support 100Mbps continuous data throughput. This rather conservative figure came about from two key factors:

(1) None of the respondents of the questionnaires used in excess of 100Mbps per link (all less than 50Mbps in fact)
(2) Capturing packets at full-speed 1Gbps (the next step above 100Mbps in Ethernet terms) is very hard on current commodity hardware using a readily available operating system such as Linux

This second issue was discussed in depth by Schneider et al (2005). Their test hardware is similar in specification to the fastest of the machines that have been used for testing here, so it stands to reason that wire-speed gigabit packet capture is still not feasible on the hardware available to us.

That said, should a dramatic improvement be made to the Linux kernel or network drivers that did allow gigabit packet capture, the system would already be able to cope with the increased speed. This is evidenced by the performance testing conducted earlier that saw the fastest of the three systems able to analyse traffic at 4.6Gbps – well in excess of 1Gbps. The tests by Deri (2004) demonstrated that capture at around 600Mbps was possible, so even now, users could use the system in a gigabit configuration and see positive results providing they were not using more than 600Mbps.

Future work in this area would focus upon testing new hardware to see if gigabit capture became possible. DAG cards (highlighted by Graham (2006) as being highly specialised cards providing wire-speed gigabit capture, but at a very high price) could also be validated with the system, providing they could be sourced.

Chapter 7

Conclusion

The original aim of the project was to create a distributed network traffic analyser that was able to classify even the most difficult of protocols, whilst operating at high speed on commodity hardware. Traditional techniques such as SNMP (used by MRTG and Cacti) were shown to provide too little detail, whilst more involved techniques such as (stateless) port analysis (as used in nTop) were too inaccurate when dealing with modern protocols. Signature analysis was soon identified as the most likely candidate for achieving the task, although its use in real-world applications (and thus its performance on commodity hardware) remained largely unknown.

Commercial traffic analysers from the likes of Cisco Systems and Packeteer are able to monitor and classify vast amounts of this "difficult" traffic, but require the use of ASIC hardware and come at a very high price (typically tens of thousands of pounds, as discussed by Fluidata in Appendix Q). Such a cost often puts them beyond the reach of many of the smaller-to-medium sized ISPs, which have been highlighted as the target users of this system. Unfortunately, no such hardware was available for comparison against the finished solution created here.

The project's goals and requirements were validated against feedback from three small-to-medium UK-based ISPs, all of whom felt that the system would make a useful addition to their business. One ISP in particular, Fluidata Limited, went to greater lengths by providing a live test-bed for the system on their core network, as well as regular feedback about its accuracy, performance and usability. This proved to be a great help to the ongoing system development, with respect to both functionality and the personal satisfaction gained in knowing that it had a real-world application. Fluidata are continuing to use the system even now (having negotiated a non-exclusive royalty-free license agreement with the University of Bath), and are looking to integrate it with more of their systems (see last paragraph, Appendix Q).

The original project plan (Appendix S), developed as a part of the project proposal, has been loosely followed. Whilst the bulk of the documentation began later than scheduled, the system development was completed ahead of time. Furthermore, the original project plan did not account for any large scale deployment to a third party (as happened with Fluidata in February/March 2007). Despite this, all of the key milestones in the project plan were met, with little or no overrun in to the contingency time.

Early performance concerns were raised when the Literature Survey found that even modern Linux kernels could not capture packets at wire-speed 1Gbps. This would present a problem when the system was used in very high traffic environments, and perhaps partly explains why commercial vendors tend to build their solutions on ASIC hardware (which they can optimise for precisely this environment). Research in to how the target audience (the ISPs) maintained their network links revealed that sub-50Mbps links were used exclusively; indicating that supporting a mere 100Mbps continuous data throughput would be adequate for the purposes of this project. Tests of the finished system later revealed that even 10-year old hardware could handle over 100Mbps, whilst modern hardware would achieve some 4.6Gpbs (~4600Mbps), with the limiting factor being the speed of the kernel.

The design and implementation section of this document has shown how real-world testing on Fluidata's network instigated significant changes to the systems construction following some early issues. Whilst the traffic capture and analysis component of the system performed very well throughout, it was soon discovered that writing such a large volume of statistics to the hard disk in a meaningful way (such that they could easily be accessed for display on the front end) was considerably more difficult. This type of problem would have been very difficult to simulate in a controlled environment, so it was very beneficial to have access to a live real-world network to test the system against.

The accuracy and reliability of the system performed well under testing, always exceeding the 99% goal and often surpassing 99.999%. The modularity with which the client application was designed and built allows support for future protocols to be incorporated with relative ease and very little risk. Concerns over future protocols use of encryption to mask their true contents (thus defeating the signature analysis method of classification used here) led to the investigation of alternative methods, such as the Transport Layer method presented in Madhukar et al (2006). Whilst there is no known real-world implementation of this method to compare against, what little research there is suggests that this would make an excellent addition to the system. That said, at this time there is relatively little call for this as few protocols employ encryption as standard (the commonly-used exception being HTTPS and VPNs). The Transport Layer method alone would be insufficient to provide the granularity required here (it can merely infer what class of traffic a packet belongs to, based upon its characteristics), so it would be supplemental and not a replacement for the (stateful) signature analysis method used.

Functional testing of the system has also yielded very positive results, with every test passing and supporting evidence being found for each non-functional test too. Fluidata's feedback was also very positive, demonstrating not only that it met their needs, but also that it had a real world application. Again, their continued use of the system following the completion of this dissertation is a testament to this.

From an academic viewpoint, the signature analysis method developed here has been shown to be both highly accurate and a strong performer. Whilst signature analysis has been studied in previous research, it has been extended here to maintain state and infer classifications based upon transitive relationships (e.g. If A is communicating with B and is known to be communicating HTTP, and B is communicating with C, then the communication between B and C must also be HTTP). Supporting such inferences provides dramatically improved classification rates and reduces overhead on the system. There is certainly room for improvement though, most likely in the form of an extension to support the Transport Layer method (discussed in depth in the Future Work section), which would improve classification rates by reducing the amount of “Unknown” traffic found on the network. However, as highlighted in the Literature Survey, signature analysis may not always be viable due to it requiring access to the raw packets on the network. This was identified as a potential cause for concern with respect to privacy and legality.

From all perspectives, the project has clearly been a success. All of the original goals have been met, and often surpassed. The interest in the project from Fluidata and other ISPs has also demonstrated the products appeal to potential users. There is clearly room for future work into the areas explored by this dissertation; particularly with respect to scalability and handling the increasing volumes of encrypted traffic.

Bibliography

1. Author Unknown. Active FTP vs. Passive FTP, a Definitive Explanation. Available at: http://slacksite.com/other/ftp.html Accessed 03/12/2006)
2. Azureus Project. 2005. Azureus Message Stream Encryption. Available at: http://www.azureuswiki.com/index.php/Message_Stream_Encryption (Accessed: 24/02/2007)
3. Barford, P., Kline, J., Plonka, D., and Ron, A. 2002. A signal analysis of network traffic anomalies. ACM Press, New York, NY, 71-82. Available at: http://doi.acm.org/10.1145/637201.637210 (Accessed 03/12/2006)
4. "Beej". 2007. A cheesy multiperson chat server. Available at: http://beej.us/guide/bgnet/examples/selectserver.c (Accessed: 12/01/2007)
5. Bernstein, D. 2001. daemontools. Available at: http://cr.yp.to/daemontools.html (Accessed: 04/03/2007)
6. Beverly, R. RTG: A Scalable SNMP Statistics Architecture for Service Providers. Available at: http://www.usenix.org/events/lisa02/tech/beverly/beverly.pdf (Accessed 03/12/2006)
7. Boehm, H. 2007. A garbage collector for C and C++. Available at: http://www.hpl.hp.com/personal/Hans_Boehm/gc/ (Accessed 01/03/2007)
8. BitTorrent.org, 2006. BitTorrent Protocol Specification. Available at: http://www.bittorrent.org/protocol.html (Accessed 01/03/2007)
9. BT Plc. 2003. Press release: BT in Broadband Breakthrough. Available at: http://www.btplc.com/News/Articles/Showarticle.cfm?ArticleID=b79eadae-2d42-4f01-8375-7680997c3b44 (Accessed 07/03/2007)
10. BT Wholesale. 2007. BT Wholesale ADSL Coverage. Available at: http://www.btwholesale.com/application;?pageid=editorial_two_column&nodeId=navigation/node/data/Broadband_Community/Coverage/navNode_Coverage (Accessed 07/03/2007)
11. Caceres R, 1989. Measurements of Wide Area Internet Traffic. Available at: http://digitalassets.lib.berkeley.edu/techreports/ucb/text/CSD-89-550.pdf (Accessed 03/12/2006)
12. CacheLogic. 2006. CacheLogic Research: P2P in 2005. Available at: http://www.cachelogic.com/home/pages/studies/2005_06.php (Accessed 03/12/2006)
13. CERT. 2001. Denial of Service Attacks. Available at: http://www.cert.org/tech_tips/denial_of_service.html (Accessed: 01/03/2007)
14. Cisco Systems, Inc. 2006. Introduction to Cisco IOS NetFlow – A Technical Overview. Available at: http://www.cisco.com/application/pdf/en/us/guest/products/ps6601/c1244/cdccont_0900aecd80406232.pdf (Accessed 03/12/2006)
15. Cisco Systems. 2006. Internet Protocols (IP): TCP Packet Format. Available at: http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/ip.htm#wp3664 (Accessed 03/12/2006)

16. Clickatel, 2007. SMS Gateway API. Available at: http://www.clickatell.com/brochure/products/developer_solutions.php (Accessed 27/02/2007)
17. Cole, R. G. and Rosenbluth, J. H. 2001. Voice over IP performance monitoring. SIGCOMM Comput. Commun. Rev. 31, 2 (Apr. 2001), 9-24. Available at: http://doi.acm.org/10.1145/505666.505669 (Accessed 23/04/2007)
18. ComScore Networks. 2001. Press release regarding Napsters decline. Available at: http://www.comscore.com/press/release.asp?id=249 (Accessed 01/03/2007)
19. Deri, L. 2004. Improving Passive Packet Capture: Beyond Device Polling. Available at: http://luca.ntop.org/Ring.pdf (Accessed 03/12/2006)
20. Dreger, H., Feldmann, A., Paxson, V., and Sommer, R. 2004. Operational experiences with high-volume network intrusion detection. ACM Press, New York, NY, 2-11. Available at: http://doi.acm.org/10.1145/1030083.1030086 (Accessed 03/12/2006)
21. Estan, C., Keys, K., Moore, D., and Varghese, G. 2004. Building a better NetFlow. ACM Press, New York, NY, 245-256. Available at: http://doi.acm.org/10.1145/1015467.1015495 (Accessed 03/12/2006)
22. Feldman, J. March 2004. NTop, NetFlow and Cisco Routers. Available at: http://www.fr.ntop.org/mirrors/networking/ntop/ntop-netflow-cisco.pdf (Accessed: 27/02/2007)
23. Fisk, M. and Varghese, G. 2002. Agile and scalable analysis of network events. ACM Press, New York, NY, 285-290. Available at: http://doi.acm.org/10.1145/637201.637245 (Accessed 03/12/2006)
24. Fukuda, K., Cho, K., and Esaki, H. 2005. The impact of residential broadband traffic on Japanese ISP backbones. Available at: http://doi.acm.org/10.1145/1052812.1052820 (Accessed 03/12/2006)
25. Guerrero, J. H. and Cárdenas, R. G. 2005. An example of communication between security tools: iptables - snort. *SIGOPS Oper. Syst. Rev.* 39, 3 (Jul. 2005), 34-43. Available at: http://doi.acm.org/10.1145/1075395.1075398 (Accessed 17/04/2007)
26. Gradwell, P. 2006. Curriculum Vitae. Available at: http://peter.gradwell.com/cv/cv.pdf (Accessed 23/04/2007)
27. Graham, I. Endace Limited. Achieving Zero-loss Multi-gigabit IDS. Available at: http://www.touchbriefings.com/pdf/2259/graham.pdf (Accessed 27/02/2007)
28. IANA, 2007. Well Known Port Numbers. Available at: http://www.iana.org/assignments/port-numbers (Accessed: 27/02/2007)
29. IEEE Standards Association. 2003. 802.1Q Virtual LANs. Available at http://standards.ieee.org/getieee802/download/802.1Q-2003.pdf (Accessed: 12/03/2007)
30. Joseph R., Hu Z., Martonosi M. 2004. Wavelet Analysis for Microprocessor Design: Experiences with Wavelet-Based dI/dt Characterization (Summary section). Available at http://www.ece.northwestern.edu/~rjoseph/publications/wavelet-hpca2004.pdf (Accessed 07/03/2007)
31. Minshall, M. 2005. TCPDPRIV. Available at: http://ita.ee.lbl.gov/html/contrib/tcpdpriv.html (Accessed 03/12/2006)

32. Hiremagalur, B. and Kar, D. C. 2005. WLAN traffic graphing application using simple network management protocol. Available at: http://portal.acm.org/citation.cfm?id=1047871&coll=Portal&dl=ACM&CFID=2463885&CFTOKEN=94789903&ret=1 (Accessed 03/12/2006)
33. Hussain, A., Bartlett, G., Pryadkin, Y., Heidemann, J., Papadopoulos, C., and Bannister, J. 2005. Experiences with a continuous network tracing infrastructure. ACM Press, New York, NY, 185-190. Available at: http://doi.acm.org/10.1145/1080173.1080181 (Accessed 03/12/2006)
34. Lawrence Berkeley National Laboratory Network Research Group. 2006. Libpcap and TCPDump. Available at: http://www.tcpdump.org/ (Accessed 03/12/2006)
35. Lawrence Berkeley National Laboratory Network Research Group. 2006. Related projects to Libpcap. Available at: http://www.tcpdump.org/related.html (Accessed 03/12/2006)
36. Madhukar, A and Williamson, C. 2006. A Longitudinal Study of P2P Traffic Classification. Available at: http://pages.cpsc.ucalgary.ca/~carey/papers/2006/Alok-MASCOTS2006.pdf (Accessed: 27/02/2006)
37. MySQL AB. 2006. MySQL C API. Available at: http://dev.mysql.com/doc/refman/5.0/en/c.html (Accessed 03/12/2006)
38. Network Instruments LLC. 2004. Analysing Full-Duplex Networks Through SPAN's, Port Aggregators, and TAPs. Available at: http://www.dna.com.au/pdf/netinst/TAPs_NInetwork_WP.pdf (Accessed 03/12/2006)
39. Oetiker, T. 2006. RRDTool. Available at: http://oss.oetiker.ch/rrdtool/ (Accessed 03/12/2006)
40. Peuhkuri, M. 2001. A method to compress and anonymize packet traces. ACM Press, New York, NY, 257-261. Available at: http://doi.acm.org/10.1145/505202.505233 (Accessed 03/12/2006)
41. Point Topic. 2007. Report of broadband take-up in 2003 and 2006. Included in Appendix L. Provided by Oliver Johnson, General Manager, Point Topic Ltd.
42. Pras, A. 2000. NTOP – Network TOP. An Overview. Available at: http://www.ntop.org/ntop-overview.pdf (Accessed 03/12/2006)
43. Samknows.com. 2007. Figures compiled from data available on the website. Available at: http://www.samknows.com/broadband/ (Accessed 07/03/2007)
44. Shalunov, S. 2003. Internet2 NetFlow Weekly Reports. Available at: http://www.Internet2.edu/~shalunov/talks/20031013-Indianapolis-MM-NetFlow.pdf (Accessed 03/12/2006)
45. Shankar, A. Random Code Samples. Available at: http://floatingsun.net/asimshankar/code-samples/user/pcap/ (Accessed: 05/01/2007)
46. Schneider, F. and Wallerich, J. 2005. Performance evaluation of packet capturing systems for high-speed networks. ACM Press, New York, NY, 284-285. Available at: http://doi.acm.org/10.1145/1095921.1095982 (Accessed 03/12/2006)
47. Steffen, E. 2004. Bonsai: An other cool zoom/history feature for Cacti. Available at http://forums.cacti.net/about3295.html (Accessed: 16/03/2007)
48. Symantec Corporation. 2006. Security Response: W32.SQLExp.Worm. Available at:

http://www.symantec.com/security_response/writeup.jsp?docid=2003-012502-3306-99 (Accessed 03/12/2006)

49. Network Working Group. 2000. RFC 2784, Generic Routing Encapsulation. Available at: http://tools.ietf.org/html/rfc2784 (Accessed (12/03/2007)
50. The Cacti Group. 2006. Cacti: The Complete RRDTool- based Graphing Solution. Available at: http://cacti.net/ (Accessed 03/12/2006)
51. The PHP Group. 2006. PHP Manual: MySQL Functions. Available at: http://uk.php.net/mysql (Accessed 03/12/2006)
52. The Register. February 2007. BitTorrent Inc offers legal downloads. Available at: http://www.theregister.co.uk/2007/02/26/bittorrent_legal_downloads/ (Accessed 05/03/2007)
53. Theory.org, 2006. BitTorrent Specification. Available at: http://wiki.theory.org/BitTorrentSpecification (Accessed: 27/02/2007)
54. TIA (Telecommunications Industry Association). March 2006. Voice Quality Recommendations for IP Telephony. Available at http://www.tiaonline.org/standards/technology/voip/documents/TSB116-Afinalforglobal.pdf (Accessed 23/04/2007)
55. UPnP Forum. 2006. Universal Plug and Play. Available at: http://www.upnp.org/ (Accessed 03/12/2006)

Appendices

Appendix A – Blank questionnaire

Questionnaire

The following questionnaire is intended to aide in the requirements elicitation process of the development of a new network traffic monitoring system. Please answer all the questions as truthfully and fully as possible. For each question, please tick, highlight or circle the answer(s) that are appropriate.

---

1. How would you classify the business you represent? Please select one of the following:

   □ Non-IT related small business
   □ Small-medium ISP
   □ Large ISP

2. What is your role within that business? Please select one of the following:

   □ Owner
   □ Director
   □ Manager
   □ Other: __________________________________

3. Is the business reliant upon Internet connectivity, either for its own operations or its customers?

   □ Yes □ No

4. Do you currently employ any network traffic monitoring software to monitor the state of the network? Please select all of the ones you use.

   □ None
   □ Cacti
   □ MRTG
   □ Cricket
   □ nTop
   □ HP OpenView
   □ Custom RTG/RRDTool development

□ Other: ______________________________

5. Would you find the ability to monitor traffic per application a useful feature of a new traffic analyser?

   □ Yes □ No

6. If you answered Yes to question 5, which of the following protocols would you like to monitor? Please select all of the ones applicable.

   □ HTTP □ IMAP □ POP □ DNS □ SSH

   □ SMTP □ Telnet □ SIP □ Kazaa □ Gnutella

   □ BitTorrent

   □ Others: ______________________________

7. Do you currently maintain multiple disparate Internet connections, or subnets within your network that you would like to be able to analyse traffic for separately?

   □ Yes □ No

8. If you answered Yes to question 7, what is the average traffic per Internet connection you currently operate? Please select one of the following:

   □ Less than 10Mbps □ Less than 50Mbps

   □ Less than 100Mbps □ Less than 1Gbps

   □ Over 1Gbps

9. What graphing intervals would you consider useful for a traffic analyser to employ? Please select all of the ones applicable.

   □ Sub-daily □ Daily □ Weekly

   □ Monthly □ Yearly

10. Do you currently employ an automated alerting system to monitor the state of your network, and alert you if an issue arises?

    □ Yes □ No

11. If you answered Yes to question 10, what means of alerting do you use, or would find useful in a future system? Please select all of the ones applicable.

    □ Email □ SMS □ Pager

□ Other: ______________________________

12. Would you find the ability to alert upon different classifications of traffic useful in a future system that allowed application-level traffic monitoring?

□ Yes □ No

13. Do you have any further comments or suggestions surrounding the development of a new network traffic monitoring system? If so, please enter them below:

Appendix B – Questionnaire responses

Respondent: Alexander Fossa, Managing Director, Xifos Limited

Questionnaire

The following questionnaire is intended to aide in the requirements elicitation process of the development of a new network traffic monitoring system. Please answer all the questions as truthfully and fully as possible. For each question, please tick, highlight or circle the answer(s) that are appropriate.

---

1. How would you classify the business you represent? Please select one of the following:

   □ Non-IT related small business
   ■ Small-medium ISP
   □ Large ISP

2. What is your role within that business? Please select one of the following:

   ■ Owner
   □ Director
   □ Manager
   □ Other: ____________________

3. Is the business reliant upon Internet connectivity, either for its own operations or its customers?

   ■ Yes □ No

4. Do you currently employ any network traffic monitoring software to monitor the state of the network? Please select all of the ones you use.

   □ None
   ■ Cacti
   □ MRTG
   □ Cricket
   □ nTop
   □ HP OpenView
   □ Custom RTG/RRDTool development

   □ Other: ____________________

5. Would you find the ability to monitor traffic per application a useful feature of a new traffic analyser?
   ■ Yes □ No

6. If you answered Yes to question 5, which of the following protocols would you like to monitor? Please select all of the ones applicable.

   ■ HTTP   □ IMAP   □ POP   □ DNS   □ SSH

   □ SMTP   □ Telnet   ■ SIP   ■ Kazaa   ■ Gnutella

   ■ BitTorrent

   □ Others: ______________________________

7. Do you currently maintain multiple disparate Internet connections, or subnets within your network that you would like to be able to analyse traffic for separately?

   ■ Yes   □ No

8. If you answered Yes to question 7, what is the average traffic per Internet connection you currently operate? Please select one of the following:

   □ Less than 10Mbps   ■ Less than 50Mbps

   □ Less than 100Mbps   □ Less than 1Gbps

   □ Over 1Gbps

9. What graphing intervals would you consider useful for a traffic analyser to employ? Please select all of the ones applicable.

   ■ Sub-daily   □ Daily   □ Weekly

   □ Monthly   □ Yearly

10. Do you currently employ an automated alerting system to monitor the state of your network, and alert you if an issue arises?

   ■ Yes   □ No

11. If you answered Yes to question 10, what means of alerting do you use, or would find useful in a future system? Please select all of the ones applicable.

   ■ Email   ■ SMS   □ Pager

   □ Other: ______________________________

12. Would you find the ability to alert upon different classifications of traffic useful in a future system that allowed application-level traffic monitoring?

   ■ Yes   □ No

13. Do you have any further comments or suggestions surrounding the development of a new network traffic monitoring system? If so, please enter them below:

    Ability to identify trends in traffic, to reduce the number of false alerts.
    Packet Signature detection, Layer 4 -7 Inspection

Respondent: Chris Rogers, Operations Director, Fluidata Limited

Questionnaire

The following questionnaire is intended to aide in the requirements elicitation process of the development of a new network traffic monitoring system. Please answer all the questions as truthfully and fully as possible. For each question, please tick, highlight or circle the answer(s) that are appropriate.

---

1. How would you classify the business you represent? Please select one of the following:

   □ Non-IT related small business
   ■ Small-medium ISP
   □ Large ISP

2. What is your role within that business? Please select one of the following:

   □ Owner
   ■ Director
   □ Manager
   □ Other: ______________________________

3. Is the business reliant upon Internet connectivity, either for its own operations or its customers?

   ■ Yes □ No

4. Do you currently employ any network traffic monitoring software to monitor the state of the network? Please select all of the ones you use.

   □ None
   ■ Cacti
   ■ MRTG
   □ Cricket
   □ nTop
   □ HP OpenView
   ■ Custom RTG/RRDTool development

   □ Other: ______________________________

5. Would you find the ability to monitor traffic per application a useful feature of a new traffic analyser?

   ■ Yes □ No

6. If you answered Yes to question 5, which of the following protocols would you like to monitor? Please select all of the ones applicable.

■ HTTP ■ IMAP ■ POP ■ DNS ■ SSH

■ SMTP ■ Telnet ■ SIP ■ Kazaa ■ Gnutella

■ BitTorrent

■ Others: IP-Sec, FTP, HTTPS, ICMP echo

7. Do you currently maintain multiple disparate Internet connections, or subnets within your network that you would like to be able to analyse traffic for separately?

■ Yes □ No

8. If you answered Yes to question 7, what is the average traffic per Internet connection you currently operate? Please select one of the following:

□ Less than 10Mbps ■ Less than 50Mbps

□ Less than 100Mbps □ Less than 1Gbps

□ Over 1Gbps

9. What graphing intervals would you consider useful for a traffic analyser to employ? Please select all of the ones applicable.

■ Sub-daily ■ Daily ■ Weekly

■ Monthly ■ Yearly

10. Do you currently employ an automated alerting system to monitor the state of your network, and alert you if an issue arises?

■ Yes □ No

11. If you answered Yes to question 10, what means of alerting do you use, or would find useful in a future system? Please select all of the ones applicable.

■ Email ■ SMS □ Pager

□ Other: Automated dial plan for critical alerts to call set list of mobiles in turn until it gets an answer, as SMS delivery cannot be guaranteed, with service providers offering up to 24 hour delivery guarantee, so cannot be 100% relied upon for the most serious emergency

12. Would you find the ability to alert upon different classifications of traffic useful in a future system that allowed application-level traffic monitoring?

■ Yes □ No

13. Do you have any further comments or suggestions surrounding the development of a new network traffic monitoring system? If so, please enter them below:

- The ability to monitor the AS numbers traffic is going to, so that direct peering links can be established with the most common ones and reduce transit costs.
- Ability to learn data these trends, as configuring levels across multiple nodes for different traffic types is going to be a lengthy task. Average traffic levels at set times of the day could be recorded for different types of traffic. Historical data could then be used to set a variance margin to work out what is within normal limits.

Respondent: Mark Barber, Owner, Brightstar Associates Limited

Questionnaire

The following questionnaire is intended to aide in the requirements elicitation process of the development of a new network traffic monitoring system. Please answer all the questions as truthfully and fully as possible. For each question, please tick, highlight or circle the answer(s) that are appropriate.

---

1. How would you classify the business you represent? Please select one of the following:

   Small-medium ISP

2. What is your role within that business? Please select one of the following:

   □ Other: Proprietor MD Owner

3. Is the business reliant upon Internet connectivity, either for its own operations or its customers?

   Yes

4. Do you currently employ any network traffic monitoring software to monitor the state of the network? Please select all of the ones you use.

   □ MRTG

5. Would you find the ability to monitor traffic per application a useful feature of a new traffic analyser?

   □ Yes

6. If you answered Yes to question 5, which of the following protocols would you like to monitor? Please select all of the ones applicable.

   □ HTTP □ IMAP □ POP □ DNS □ SSH

   □ SMTP □ Telnet □ SIP □ Kazaa □ Gnutella

   □ BitTorrent

   □ Others: All of the above plus Limewire & L2TP, BGP Headers

7. Do you currently maintain multiple disparate Internet connections, or subnets within your network that you would like to be able to analyse traffic for separately?

   □ Yes

8. If you answered Yes to question 7, what is the average traffic per Internet connection you currently operate? Please select one of the following:

   □ Less than 10Mbps

9. What graphing intervals would you consider useful for a traffic analyser to employ? Please select all of the ones applicable.

   □ Sub-daily

10. Do you currently employ an automated alerting system to monitor the state of your network, and alert you if an issue arises?

    □ Yes

11. If you answered Yes to question 10, what means of alerting do you use, or would find useful in a future system? Please select all of the ones applicable.

    □ Email □ SMS

12. Would you find the ability to alert upon different classifications of traffic useful in a future system that allowed application-level traffic monitoring?

    □ Yes

13. Do you have any further comments or suggestions surrounding the development of a new network traffic monitoring system? If so, please enter them below:

    I would like to be able track and alert on L2TP traffic to allow me to detect a BGP failover condition whereby a BGP L2TP tunnel is built over transit in a failed state.

Appendix C – Fluidata's feedback from project proposal

Fluidata

1st February, 2007

Hi Sam,

Thank you for inviting me to ready a copy of your project proposal and provide feedback. I think you have a very good understanding of what you're doing and what is required, and I'm confident that this would be an invaluable tool for your target market.

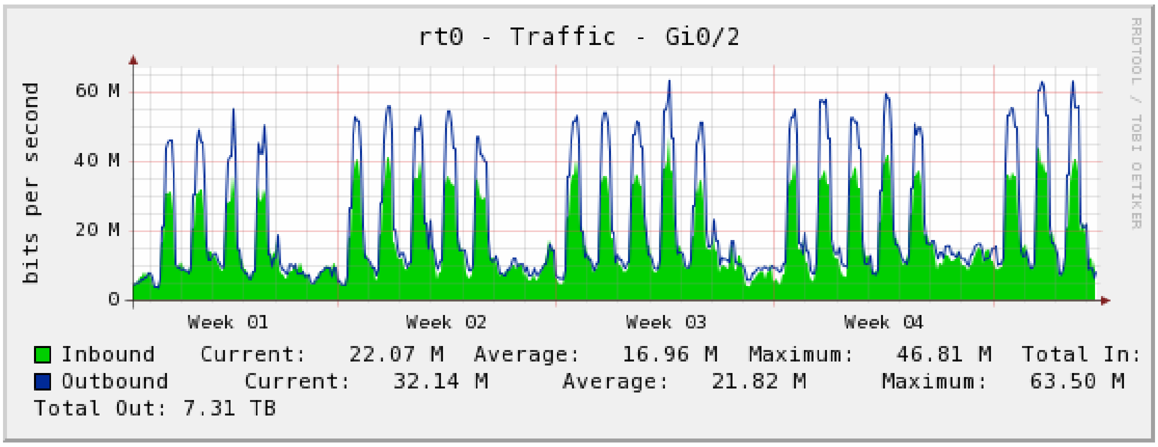
I have feedback on a couple of specific points, which are detailed below. Additionally, on the subject of testing I feel that this is something we can accommodate if it would be of value to you. I would be prepared to grant access to core network devices, but I would like to be able to operate the final version of your software on the network into the future for our own purposes.

Feedback

The ability to drill down to traffic usage according to type is extremely useful for an ISP, or anyone administering a large LAN or WAN. I think the application will be equally as useful for corporate network administrators as for ISPs.

In particular the ability to detect significant changes in types of traffic would be excellent, but I think it presents some challenges, because the system is going to need to adapt to - or be taught about - the network.

For example, as a business ISP, Fluidata's traffic levels are not constant, but they are consistent and predictable.

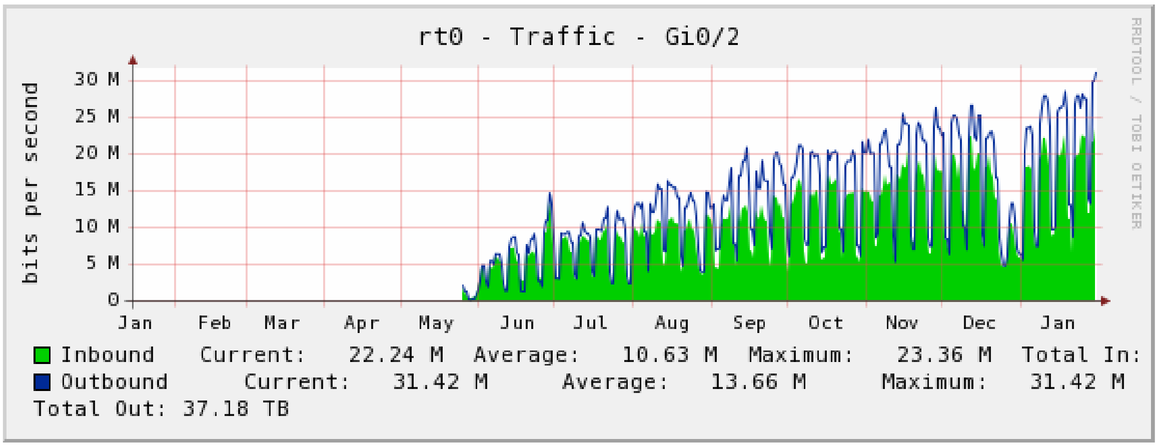


Monthly (2 Hour Average)

In this example, traffic flowing over this router interface follows a regular pattern – traffic drops right down outside of business working hours. The system would need to be configurable to be aware of normal levels of traffic at certain times, otherwise it is going to raise false alerts. So I would want the system to be able to be aware of expected levels of traffic according to days of the week and times of the day.

What would be really clever is if the system could learn these trends itself – I envisage a simple process could be run to record average traffic levels at set times of the day for different types of traffic. Historical data could then be used to set a variance margin, say using standard deviations, to work out what is within normal limits. You may consider this beyond the realms of this project, but for a systems administrator, configuring levels across multiple nodes for different traffic types is going to be a lengthy task.

Also, if levels are manually set by an administrator across nodes then as traffic levels grow over time this will also lead to false alerts being raised. If you look at the same interface as above over a longer period you will see a steady growth:



Yearly (1 Day Average)

If the administrator has programmed set thresholds into the system then there will be a constant task of revising these over time as usage grows or shrinks. But if the system uses an algorithm to teach itself normal levels using historical data, it can then keep track of trends by updating itself regularly, say by always working from the last 30 days worth of data.

Aside from this, the only additional comment I have is regarding the supported total bandwidth of 100 Mbps. I do not know if this is simply due to computer processing resources, or an inherent limitation of the software. But I would imagine that 100 Mbps would not be sufficient bandwidth for most corporate LANs and WANs or for ISPs.

My perception is that those that would have the need and could afford to deploy such a system such as yours would have a greater aggregated bandwidth than this. However, I appreciate that this is a concern relating to commercial deployment, and that for the purposes of demonstrating the software for this project 100 Mbps would suffice.

Testing

Fluidata would be prepared to let you test this system on the network. I do not believe there would be any privacy or legal issues with this for the following reasons:

- All of our clients, without exception, have signed a contract with us which contains the following clause:

"Fluidata shall have the right to examine, from time to time, the use to which you put the Services and the nature of the data/information you are transmitting or receiving via the Services where such examination is necessary: (i) to protect / safeguard the integrity, operation and functionality of Fluidata's (and neighbouring) networks; or (ii) to comply with police, judicial, regulatory or governmental orders, notices, directives or requests."

- Your system would allow us to examine traffic so that we can protect and safeguard the operation of the network. And, as it would not actually allow us to packet sniff, we would not be able to see actual client data, which would be more of a legal issue.

- You would be acting as an agent appointed by Fluidata and a non-disclosure agreement would exist between us and you to protect any confidential information you may discover.

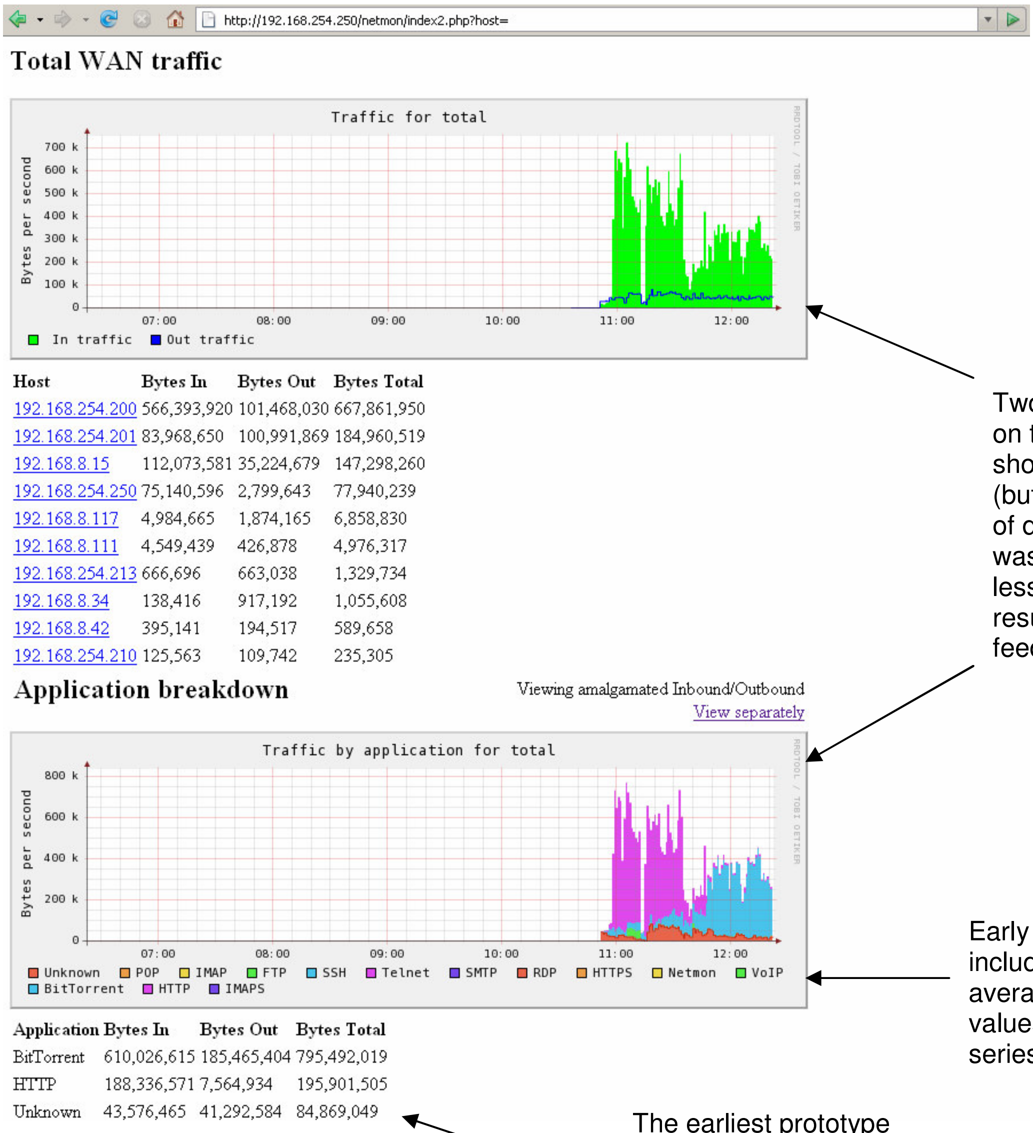
I hope that this is of use to you, and wish you success with your project. If we can be of any further assistance please don't hesitate to get in touch

Yours sincerely,

Chris Rogers

Operations Director
Fluidata

Appendix D – Early design of front-end prototype



Total WAN traffic

Host	Bytes In	Bytes Out	Bytes Total
192.168.254.200	566,393,920	101,468,030	667,861,950
192.168.254.201	83,968,650	100,991,869	184,960,519
192.168.8.15	112,073,581	35,224,679	147,298,260
192.168.254.250	75,140,596	2,799,643	77,940,239
192.168.8.117	4,984,665	1,874,165	6,858,830
192.168.8.111	4,549,439	426,878	4,976,317
192.168.254.213	666,696	663,038	1,329,734
192.168.8.34	138,416	917,192	1,055,608
192.168.8.42	395,141	194,517	589,658
192.168.254.210	125,563	109,742	235,305

Application breakdown

Viewing amalgamated Inbound/Outbound
View separately

Application	Bytes In	Bytes Out	Bytes Total
BitTorrent	610,026,615	185,465,404	795,492,019
HTTP	188,336,571	7,564,934	195,901,505
Unknown	43,576,465	41,292,584	84,869,049

Two graphs presented on the front page, both showing similar data (but at different levels of detail). A decision was made to drop the less detailed one, as a result of user feedback.

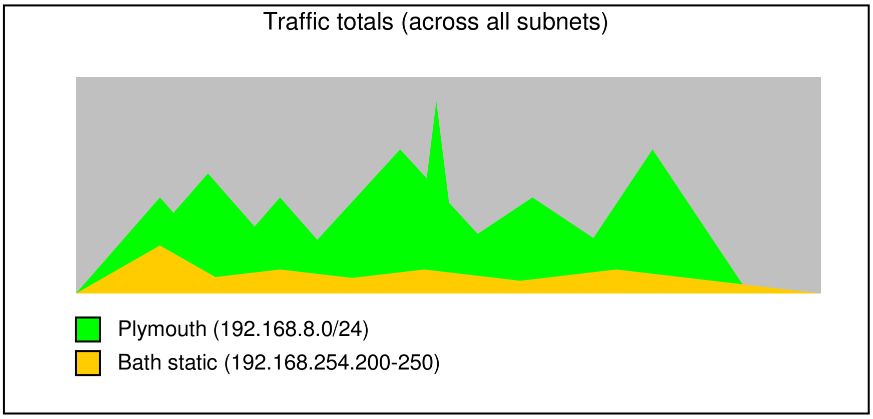
Early graphs did not include the min, max, average, and current values for each data series.

The earliest prototype simply presented data for the past day. Later versions included statistics for current traffic, traffic from the past hour, past six hours and past day.

Appendix E – Later design of front-end

Netmon

Groups



	Current (Up/Down)	Active hosts	Data in 6 hrs (Up/Down)
Plymouth	220 / 30 KB/s	14	3.1 / 0.7 GB
Bath static	40 / 2 KB/s	8	774 / 67 MB

Most used applications

(Showing Top 10 →View all)

	Current (Up/Down)	Data in 6 hrs (Up/Down)
BitTorrent	200 / 20 KB/s	3.1 / 0.7 GB
HTTP	15 / 2 KB/s	774 / 67 MB
Unknown	3 / 1 KB/s	500 / 45 MB
FTP	0 / 0 KB/s	176 / 1 MB
IMAP	0 / 0 KB/s	5 / 0.1 MB

Most active hosts

(Showing Top 10 →View all)

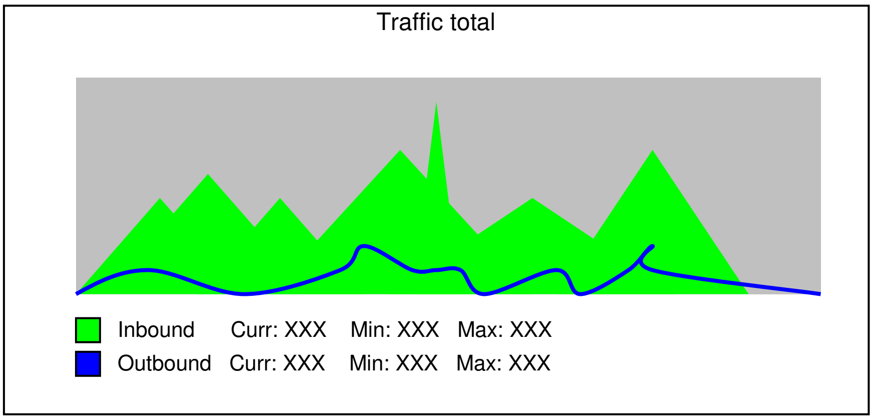
	Current (Up/Down)	Data in 6 hrs (Up/Down)
192.168.8.15	220 / 30 KB/s	3.1 / 0.7 GB
192.168.254.200	40 / 2 KB/s	774 / 67 MB
192.168.254.200	40 / 2 KB/s	774 / 67 MB
192.168.254.200	40 / 2 KB/s	774 / 67 MB
192.168.254.200	40 / 2 KB/s	774 / 67 MB
192.168.254.200	40 / 2 KB/s	774 / 67 MB

Plymouth

> Edit group settings

Total traffic

(View historical)



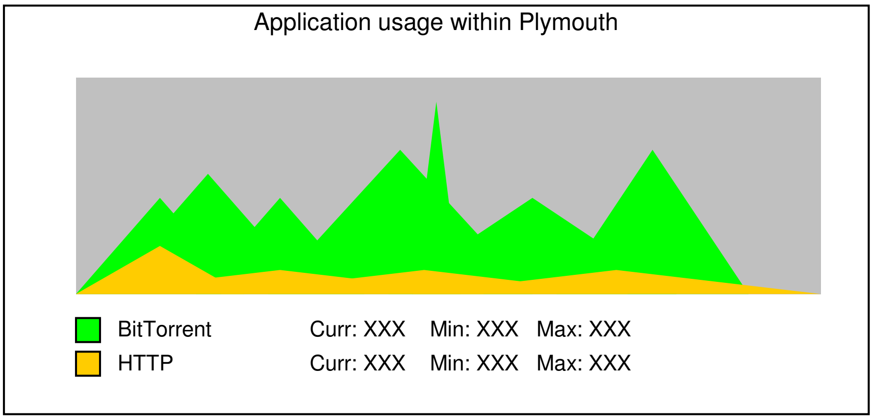
Most active hosts

(Showing Top 10 → View all 14)

IP	Current (Up/Down)	1 hour	6 hours	24 hours
192.168.8.15	220 / 30 KB/s	3.1 / 0.7 GB	3.1 / 0.7 GB	3.1 / 0.7 GB
192.168.8.15	40 / 2 KB/s	774 / 67 MB	774 / 67 MB	774 / 67 MB
192.168.8.15	40 / 2 KB/s	774 / 67 MB	774 / 67 MB	774 / 67 MB
192.168.8.15	40 / 2 KB/s	774 / 67 MB	774 / 67 MB	774 / 67 MB

Most used applications

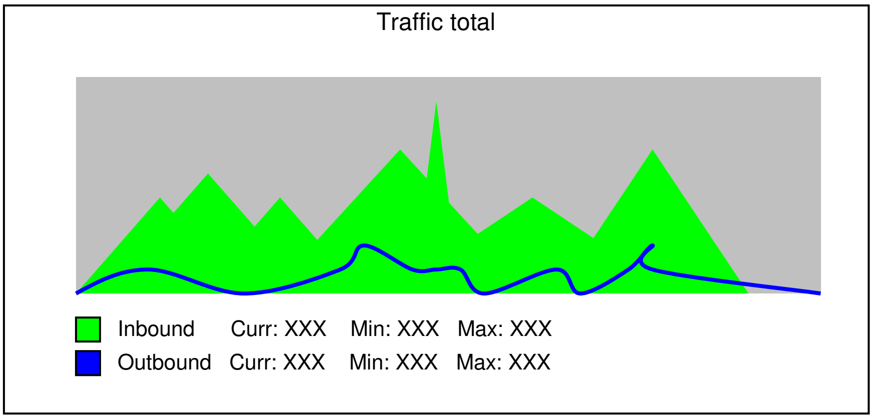
(Switch to pie-chart view)



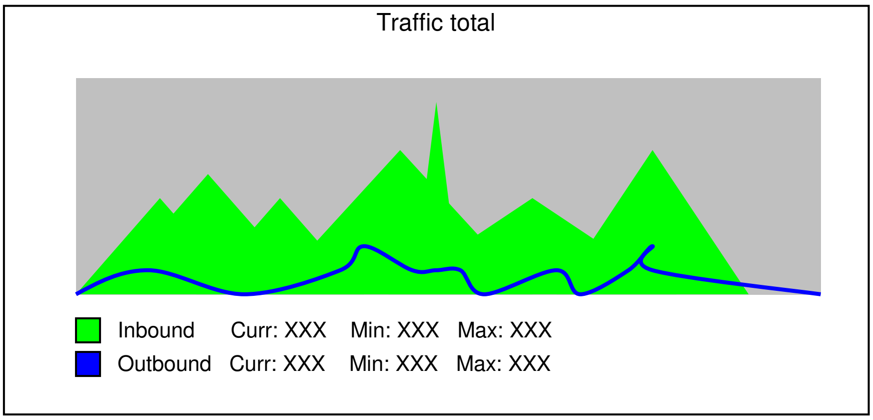
	Current (Up/Down)	Data in 6 hrs (Up/Down)
BitTorrent	200 / 20 KB/s	3.1 / 0.7 GB
HTTP	15 / 2 KB/s	774 / 67 MB
Unknown	3 / 1 KB/s	500 / 45 MB
FTP	0 / 0 KB/s	176 / 1 MB
IMAP	0 / 0 KB/s	5 / 0.1 MB

Plymouth – Historical Traffic

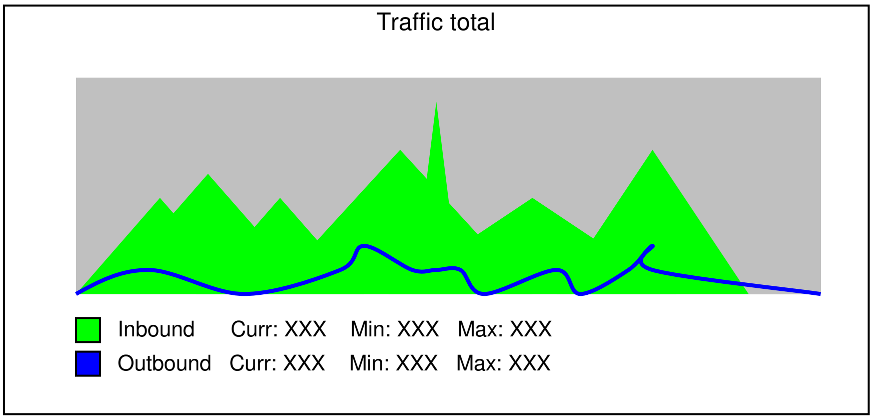
Past 24 hours



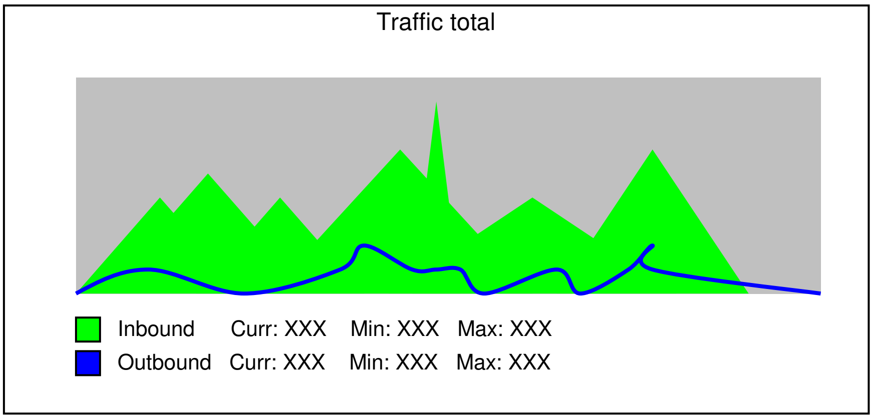
Past week



Past month



Past year



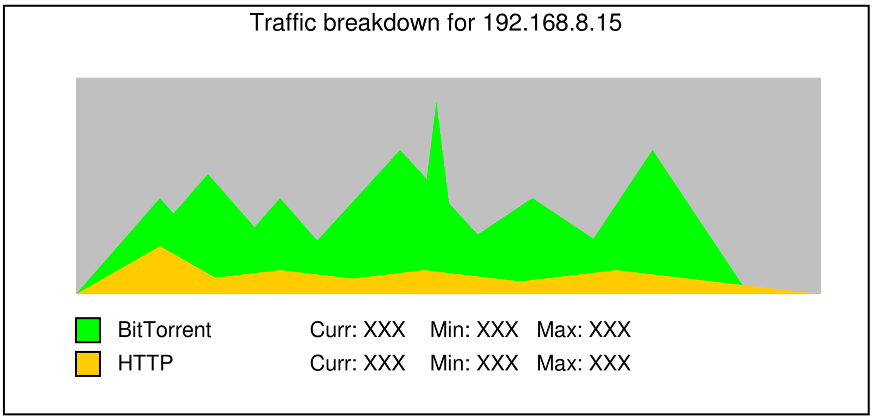
192.168.8.15

Information

Member of:	Plymouth	
Role:	BitTorrent server [Edit]	
First seen:	2000-12-16 01:24:12	
Last seen:	2007-01-16 01:24:12 (16 seconds ago)	
Traffic:	Past minute:	1.6MB in, 0.4MB out
	Past 10 minutes:	1.6MB in, 0.4MB out
	Past hour:	1.6MB in, 0.4MB out
	Past day:	1.6MB in, 0.4MB out
	Past month:	1.6MB in, 0.4MB out
	All:	1.6MB in, 0.4MB out

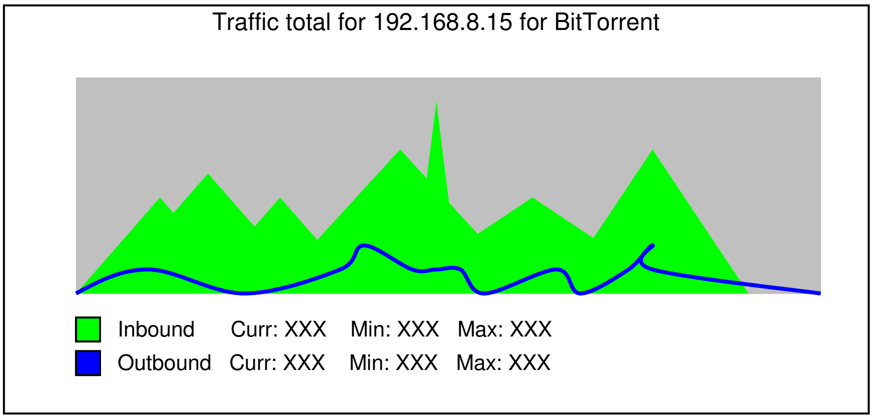
Traffic breakdown

(View historical)



	Current (Up/Down)	**Data in 6 hrs (Up/Down)**
BitTorrent	200 / 20 KB/s	3.1 / 0.7 GB
HTTP	15 / 2 KB/s	774 / 67 MB
Unknown	3 / 1 KB/s	500 / 45 MB
FTP	0 / 0 KB/s	176 / 1 MB
IMAP	0 / 0 KB/s	5 / 0.1 MB

Per application



[And all the other applications too ….]

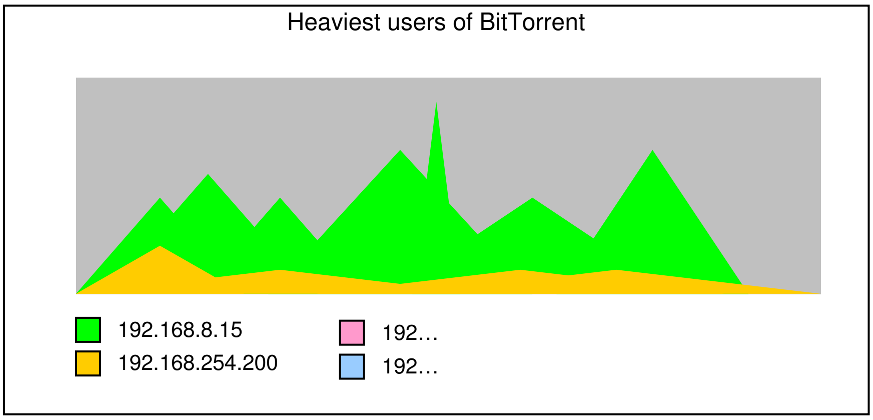
BitTorrent

Information

Traffic:	Past minute:	1.6MB in, 0.4MB out
	Past 10 minutes:	1.6MB in, 0.4MB out
	Past hour:	1.6MB in, 0.4MB out
	Past day:	1.6MB in, 0.4MB out
	Past month:	1.6MB in, 0.4MB out
	All:	1.6MB in, 0.4MB out

Heaviest users

(View historical)



IP	Current (Up/Down)	1 hour	6 hours	24 hours
192.168.8.15	220 / 30 KB/s	3.1 / 0.7 GB	3.1 / 0.7 GB	3.1 / 0.7 GB
192.168.8.15	40 / 2 KB/s	774 / 67 MB	774 / 67 MB	774 / 67 MB
192.168.8.15	40 / 2 KB/s	774 / 67 MB	774 / 67 MB	774 / 67 MB
192.168.8.15	40 / 2 KB/s	774 / 67 MB	774 / 67 MB	774 / 67 MB

Alerting Configuration

Alerting rules

Name	Action
Mail server flood protection	Edit / Delete
Mail from non mail-server	Edit / Delete
Peer2Peer file-sharing	Edit / Delete
WAN downtime	Edit / Delete

> Add a new alert

Alerting contacts

Name	Email	SMS	Action
Sam Crawford	sam@sa...	07977926…	Edit / Delete
Joe Bloggs	joe@....	072334…	Edit / Delete

> Add a new contact

Configure alert: Mail from non mail-server

Name: Mail from non mail-server

Trigger on:

Existing rules

SMTP from hosts not in “mail-servers” group, passing any amount of traffic, for any period of time, at any time of the day	Edit / Delete

> Add a new rule (via wizard)

##########

Wizard steps:

1. Define protocol (and whether or not you want to match on this)
2. Define hosts (either by single IP, by group, by range, or by NOT specific hosts)
3. Define amount of traffic (if any), and over what time period
4. Define the hours of the day for which the rule should be active

Appendix F – Packet formats

IP packet format

<table>
<tr><td>Version</td><td colspan="2">Header length</td><td>Type of service</td><td colspan="2">Total length</td></tr>
<tr><td colspan="4">Identification</td><td>Flags</td><td>Fragment offset</td></tr>
<tr><td colspan="2">TTL</td><td colspan="2">Protocol</td><td colspan="2">Checksum</td></tr>
<tr><td colspan="6">Source IP address</td></tr>
<tr><td colspan="6">Destination IP address</td></tr>
<tr><td colspan="6">Options (if any)</td></tr>
<tr><td colspan="6">Data</td></tr>
</table>

TCP packet format

<table>
<tr><td colspan="3">Source port</td><td>Destination port</td></tr>
<tr><td colspan="4">Sequence number</td></tr>
<tr><td colspan="4">Acknowledgement number</td></tr>
<tr><td>Offset</td><td>Reserved</td><td>Flags</td><td>Window</td></tr>
<tr><td colspan="3">Checksum</td><td>Urgent pointer</td></tr>
<tr><td colspan="4">Options (and padding, if applicable)</td></tr>
<tr><td colspan="4">Data (variable length)</td></tr>
</table>

UDP packet format

<table>
<tr><td>Source port</td><td>Destination port</td></tr>
<tr><td>Length</td><td>Checksum</td></tr>
<tr><td colspan="2">Data (if any)</td></tr>
</table>

Ethernet packet

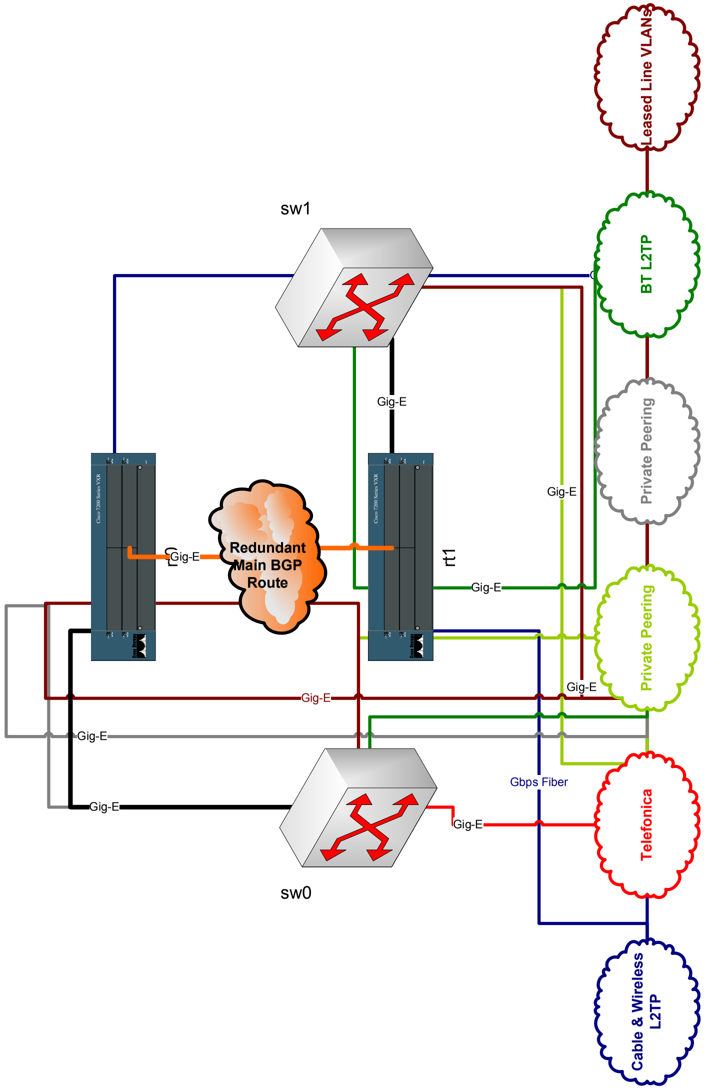
<table>
<tr><td>Preamble</td></tr>
<tr><td>Frame offset</td></tr>
<tr><td>Destination address</td></tr>
<tr><td>Source address</td></tr>
<tr><td>Length</td></tr>
<tr><td>Data</td></tr>
<tr><td>Trailing CRC</td></tr>
</table>

Appendix G – Complete database schema

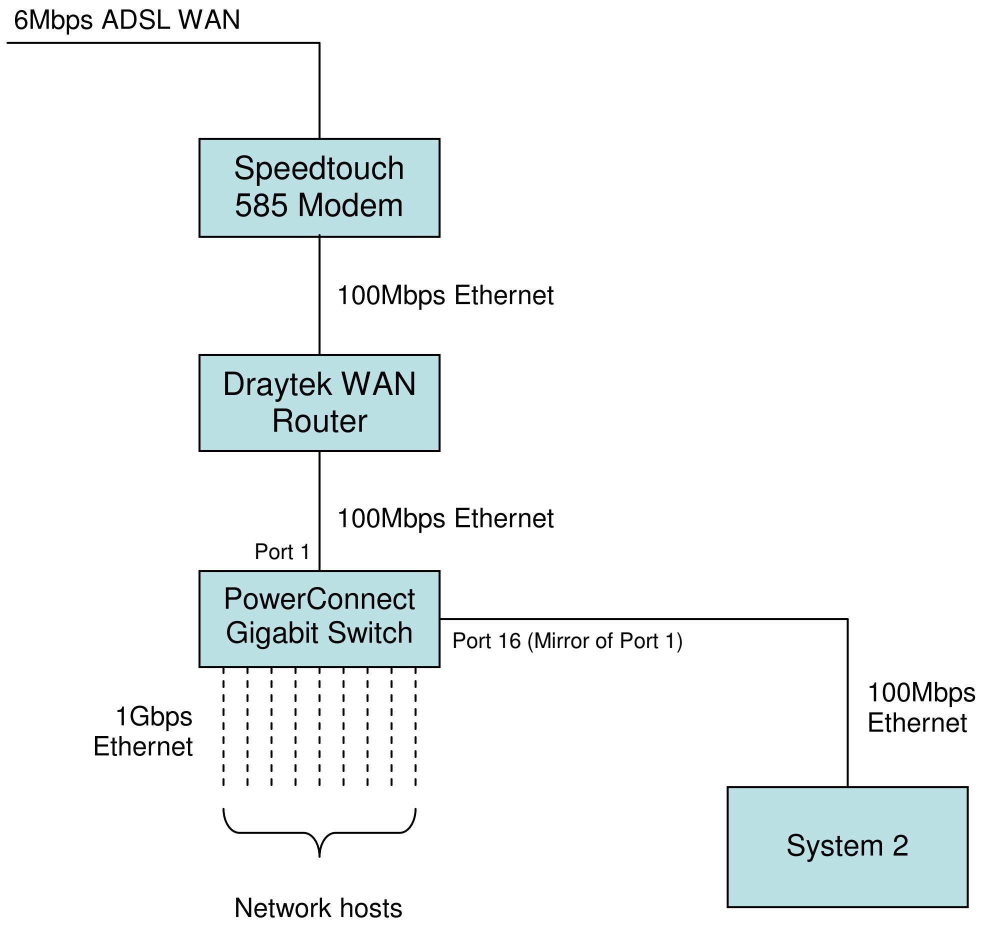
```
CREATE TABLE alert (
  alert_id int(11) NOT NULL auto_increment,
  name varchar(255) NOT NULL default '',
  phparray text NOT NULL,
  `query` text NOT NULL,
  created datetime NOT NULL default '0000-00-00 00:00:00',
  condition tinyint(1) NOT NULL default '0',
  active tinyint(1) NOT NULL default '0',
  PRIMARY KEY  (alert_id),
  UNIQUE KEY name (name)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE TABLE alert_contact (
  contact_id int(11) NOT NULL auto_increment,
  name varchar(255) NOT NULL default '',
  `type` varchar(10) NOT NULL default '',
  `value` varchar(255) NOT NULL default '',
  active tinyint(1) NOT NULL default '0',
  PRIMARY KEY  (contact_id)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE TABLE alert_contact_link (
  alert_id int(11) NOT NULL default '0',
  contact_id int(11) NOT NULL default '0',
  PRIMARY KEY  (alert_id,contact_id)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE TABLE alert_log (
  log_id int(11) NOT NULL auto_increment,
  alert_id int(11) NOT NULL default '0',
  recipient varchar(255) NOT NULL default '',
  message varchar(255) NOT NULL default '',
  `datetime` datetime NOT NULL default '0000-00-00 00:00:00',
  PRIMARY KEY  (log_id),
  KEY alert_id (alert_id)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE TABLE app_usage_1hr (
  app_id varchar(15) NOT NULL default '',
  user_type varchar(10) NOT NULL default '',
  user_value varchar(50) NOT NULL default '',
  bytes_in bigint(20) NOT NULL default '0',
  bytes_out bigint(20) NOT NULL default '0',
  dirty tinyint(1) NOT NULL default '0',
  PRIMARY KEY  (app_id,user_type,user_value),
  KEY dirty (dirty)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE TABLE app_usage_24hr (
  app_id varchar(15) NOT NULL default '',
  user_type varchar(10) NOT NULL default '',
  user_value varchar(50) NOT NULL default '',
  bytes_in bigint(20) NOT NULL default '0',
  bytes_out bigint(20) NOT NULL default '0',
  dirty tinyint(1) NOT NULL default '0',
  PRIMARY KEY  (app_id,user_type,user_value),
  KEY dirty (dirty)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE TABLE app_usage_6hr (
  app_id varchar(15) NOT NULL default '',
  user_type varchar(10) NOT NULL default '',
  user_value varchar(50) NOT NULL default '',
  bytes_in bigint(20) NOT NULL default '0',
  bytes_out bigint(20) NOT NULL default '0',
  dirty tinyint(1) NOT NULL default '0',
  PRIMARY KEY  (app_id,user_type,user_value),
  KEY dirty (dirty)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE TABLE app_usage_curr (
  app_id varchar(15) NOT NULL default '',
  user_type varchar(10) NOT NULL default '',
  user_value varchar(50) NOT NULL default '',
  speed_in int(11) NOT NULL default '0',
  speed_out int(11) NOT NULL default '0',
  dirty tinyint(1) NOT NULL default '0',
  PRIMARY KEY  (app_id,user_type,user_value),
  KEY dirty (dirty)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE TABLE application (
  app_id int(11) NOT NULL default '0',
  name varchar(255) NOT NULL default '',
  clight varchar(6) NOT NULL default '',
  cdark varchar(6) NOT NULL default '',
  PRIMARY KEY  (app_id)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE TABLE flow (
  id int(11) NOT NULL auto_increment,
  saddr varchar(15) NOT NULL default '',
  daddr varchar(15) NOT NULL default '',
  sport int(10) unsigned NOT NULL default '0',
  dport int(10) unsigned NOT NULL default '0',
  len int(10) unsigned NOT NULL default '0',
  packets int(10) unsigned NOT NULL default '0',
  protocol smallint(5) unsigned NOT NULL default '0',
  application smallint(5) unsigned NOT NULL default '0',
  started datetime default NULL,
  expires datetime default NULL,
  PRIMARY KEY  (id),
  KEY saddr (saddr,daddr)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE TABLE graph (
  filename varchar(255) NOT NULL default '',
  cmd text NOT NULL,
  `datetime` datetime NOT NULL default '0000-00-00 00:00:00',
  PRIMARY KEY  (filename)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE TABLE `group` (
  group_id int(11) NOT NULL auto_increment,
  name varchar(255) NOT NULL default '',
  rule text NOT NULL,
  PRIMARY KEY  (group_id)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE TABLE group_app_totals (
  period varchar(50) NOT NULL default '',
  application int(11) NOT NULL default '0',
  group_id int(11) NOT NULL default '0',
  bytes_in bigint(20) NOT NULL default '0',
  bytes_out bigint(20) NOT NULL default '0',
  PRIMARY KEY  (period,application,group_id)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE TABLE group_totals (
  period varchar(50) NOT NULL default '',
  group_id int(11) NOT NULL default '0',
  bytes_in bigint(20) NOT NULL default '0',
  bytes_out bigint(20) NOT NULL default '0',
  PRIMARY KEY  (period,group_id)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE TABLE host_totals (
  addr varchar(15) NOT NULL default '',
  group_id int(11) NOT NULL default '0',
  application int(11) NOT NULL default '0',
  bytes_in bigint(20) NOT NULL default '0',
  bytes_out bigint(20) NOT NULL default '0',
  `date` date NOT NULL default '0000-00-00',
  PRIMARY KEY  (addr,application,`date`),
  KEY group_id (group_id)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE TABLE host_traffic (
  id int(11) NOT NULL auto_increment,
  addr varchar(15) NOT NULL default '',
  application int(11) NOT NULL default '0',
  protocol int(11) NOT NULL default '0',
  bytes bigint(20) NOT NULL default '0',
  direction tinyint(4) NOT NULL default '0',
  `datetime` datetime NOT NULL default '0000-00-00 00:00:00',
  PRIMARY KEY  (id),
  KEY `datetime` (`datetime`),
  KEY addr (addr,application)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE TABLE host_traffic_unknown (
  saddr varchar(15) NOT NULL default '',
  daddr varchar(15) NOT NULL default '',
  sport int(11) NOT NULL default '0',
  dport int(11) NOT NULL default '0',
  bytes int(11) NOT NULL default '0',
  packets int(11) NOT NULL default '0',
  protocol smallint(6) NOT NULL default '0',
  updated int(11) NOT NULL default '0',
  PRIMARY KEY  (saddr,daddr,sport,dport,protocol)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE TABLE ip (
  id int(11) NOT NULL auto_increment,
  addr varchar(30) NOT NULL default '',
  lastseen datetime NOT NULL default '0000-00-00 00:00:00',
  PRIMARY KEY  (id),
  UNIQUE KEY addr (addr)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

CREATE TABLE ip_group (
  ipaddr varchar(30) NOT NULL default '',
  group_id int(11) NOT NULL default '0',
  PRIMARY KEY  (ipaddr)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
```

Appendix H – Fluidata network diagram

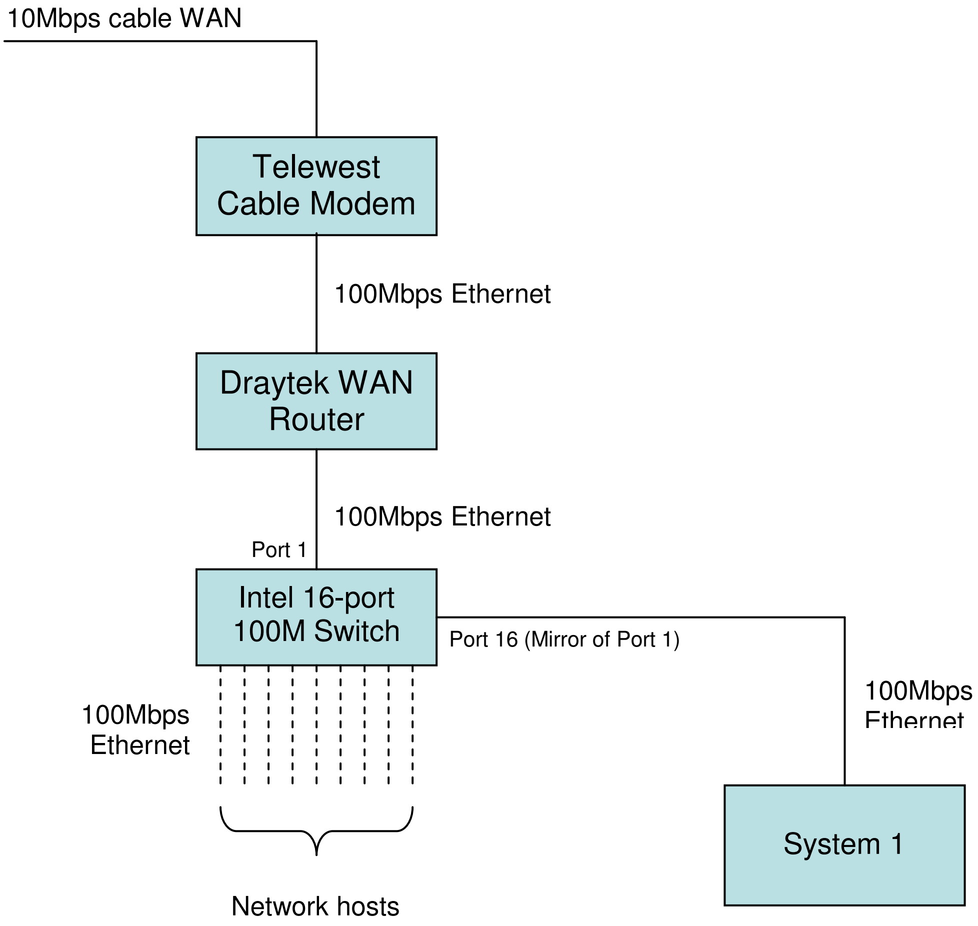
Fluidata Core Network Diagram



Appendix I – Bath network diagram



Appendix J – Plymouth network diagram



Appendix K – User manual

NETMON – USER MANUAL

Installation

The netmon application consists of four main components, each of which needs to be installed and configured separately.

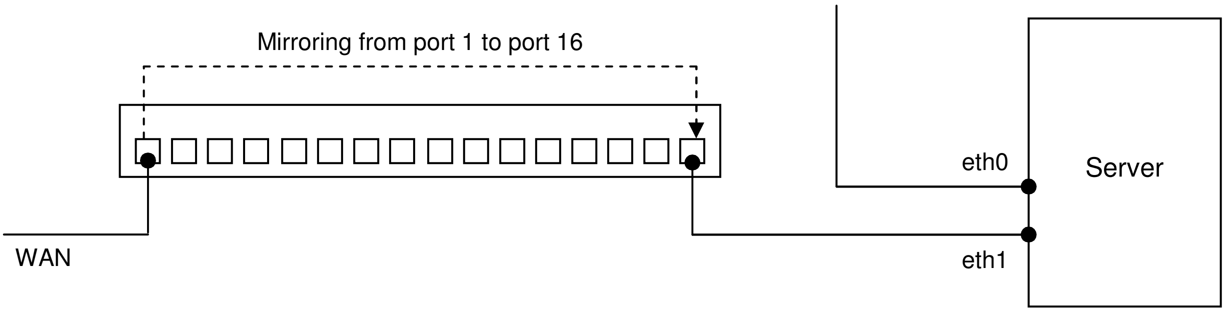
Capture and analysis client

Pre-requisites:

1. A network switch capable of port mirroring
2. A server with at least two network cards
3. The Linux operating system, with gcc, Hans Boehm's GC and libpcap/tcpdump installed
4. The source distribution for netmonc (netmonc-src.tar.gz)
5. Optional: Dan Bernstein's daemontools

Configuring your network and server

Before continuing, you must ensure that your network switch has been configured to mirror WAN traffic to a port that your servers second Ethernet card is connected to. The first Ethernet card in the server needs to be able to route to the netmon server, so will likely require a WAN connection.



Of course, the first Ethernet device on the server (eth0 in the above diagram) could be connected to the same switch.

Checklist:

1. Verify that your switch is configured port mirroring
2. Verify that both Ethernet cards on the server are active and that links are detected. You can check that the card is configured and active using "ifconfig ethX", and that the link is detected using "ethtool ethX". For example (for eth1):

```
[root@dev netmon]# ifconfig eth1
```

```
eth1      Link encap:Ethernet  HWaddr 00:03:47:A5:16:F9
          inet addr:10.0.0.10  Bcast:10.255.255.255  Mask:255.0.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:280737 errors:0 dropped:0 overruns:0 frame:0
          TX packets:239 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:138147350 (131.7 MiB)  TX bytes:20554 (20.0 KiB)

[root@dev netmon]# ethtool eth1
Settings for eth1:
        Supported ports: [ TP MII ]
        Advertised auto-negotiation: Yes
        Speed: 100Mb/s
        Duplex: Full
        Port: MII
        PHYAD: 1
        Transceiver: internal
        Auto-negotiation: on
        Supports Wake-on: g
        Wake-on: g
        Current message level: 0x00000007 (7)
        Link detected: yes
```

3. Verify that traffic from others hosts is now visible to you using tcpdump (Note that you will need to be root to use this). For example:

```
[root@dev netmon]# tcpdump -i eth1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 96 bytes
12:16:03.845202 IP 192.168.254.200.1118 > firewall.domain:  35093+ A?
www.theregister.co.uk. (39)
12:16:03.861207 IP firewall.domain > 192.168.254.200.1118:  35093 1/0/0
A[|domain]
12:16:03.863231 IP 192.168.254.200.55676 > firewall.domain:  34562+ A?
www.theregister.co.uk. (39)
12:16:03.863554 IP firewall.domain > 192.168.254.200.55676:  34562 1/0/0
A[|domain]
12:16:03.879526 IP 192.168.254.200.55677 > 212.100.234.54.http: . ack 1 win
65535
12:16:03.879697 IP 192.168.254.200.55677 > 212.100.234.54.http: P 1:478(477)
ack 1 win 65535
12:16:03.904479 IP 212.100.234.54.http > 192.168.254.200.55677: . ack 478 win
6432
```

Tip: If you only see traffic for your servers IP address, then you are probably capturing the wrong port!

Installing the software

1. Change to /tmp, unpack the netmonc source code and change to the netmonc directory

```
[root@dev ~]# cd /tmp
[root@dev tmp]# tar -zxf netmonc.tar.gz
[root@dev tmp]# cd netmonc
[root@dev netmonc]#
```

2. Compile the source code using “make”

```
[root@dev netmonc]# make
gcc -fPIC -c main.c
gcc -fPIC -c appdetect.c
…
gcc -shared bittorrent.o http.o https.o dns.o smtp.o ssh.o sip.o
imap.o ftp.o netmon.o kazaa.o ../appdetect.o -o libnetmon.so
```

```
make[1]: Leaving directory `/tmp/netmonc/libs'
```

Note: If you see any errors during the compilation process, do not ignore them! Ensure that you have gcc and make installed, as well as libpcap and Hans Boehm’s Garbage Collector

3. Install the application using “make install”

   ```
   [root@dev netmonc]# make install
   ```

4. Verify that installation was successful using the following command. If any errors are reported upon start-up, please go back and check that each of the previous steps completed successfully. If a line like the following one appears, then the installation was successful. Terminate the process using Ctrl+C and continue.

   ```
   [root@dev client]# ./netmonc –i eth1
   netmonc v0.2 starting up (Built: 2007-02-11). Monitoring eth1,
   reporting to localhost:5000...
   Loading module bittorrent... ok
   ...
   netmonc v0.2 started successfully
   ```

   Note: You will almost certainly need to edit some of the parameters to netmonc. Use –i to specify the capture interface, -h to specify the server hostname, -p to specify the server port number. See “netmonc –help” or “man netmonc” for full details.

5. OPTIONAL

   You can now configure netmonc to run at system start-up and also get automatically restarted should the process ever be killed or terminate unexpectedly.

   The best way of achieving this is to utilise Dan Bernstein’s daemontools application. Begin by installing this following the instructions at http://cr.yp.to/daemontools/install.html

   You can now configure the service to start-up automatically using the following commands:

   ```
   [root@dev ~]# mkdir /service/netmonc
   [root@dev ~]# echo “#!/bin/sh” >> /service/netmonc/run
   [root@dev ~]# echo “exec /usr/bin/netmonc –i eth1 2>&1 > /dev/null” >>
   /service/netmonc/run
   ```

   Note: Again, you will almost certainly need to modify the parameters to the netmonc command.

   You can then start the netmonc service using “svc –u /service/netmonc” and stop it using “svc –d /service/netmonc”

Netmon server

Pre-requisites:

1. The Linux operating system, with gcc and Hans Boehm's Garbage Collector
2. The source distribution for netmond (netmond-src.tar.gz)
3. Optional: Dan Bernstein's daemontools
4. A database engine such as MySQL

Installing the software

1. Change to /tmp, unpack the netmond source code and change to the netmond directory

   ```
   [root@dev ~]# cd /tmp
   [root@dev tmp]# tar -zxf netmond.tar.gz
   [root@dev tmp]# cd netmond
   [root@dev netmond]#
   ```

2. Compile the source code using "make"

   ```
   [root@dev netmond]# make
   gcc -lgc server.o -o netmond
   ```

   Note: If you see any errors during the compilation process, do not ignore them! Ensure that you have gcc and make installed, as well as Hans Boehm's Garbage Collector

3. Install the application using "make install"

   ```
   [root@dev netmond]# make install
   ```

4. You can now test the link between the client(s) and server. First start the netmond server using "netmond", and then start the netmonc clients. After a short time on the server you should begin to see SQL INSERT statements being echoed. Quit this using CTRL+C.

5. You now need to create the database and install the schema. Use the following, replacing your MySQL username and password where appropriate.

   ```
   [root@dev netmond]# mysql -u root -p < sql/schema.mysql
   ```

   You can now start the application running with database support using Unix pipes as follows:

   ```
   [root@dev netmond]# netmond | mysql -u root -p -D netmon
   ```

6. <u>OPTIONAL</u>

   You can now configure netmond to run at system start-up and also get automatically restarted should the process ever be killed or terminate unexpectedly.

The best way of achieving this is to utilise Dan Bernstein's daemontools application. Begin by installing this following the instructions at http://cr.yp.to/daemontools/install.html

You can now configure the service to start-up automatically using the following commands:

```
[root@dev ~]# mkdir /service/netmond
[root@dev ~]# echo “#!/bin/sh” >> /service/netmonc/run
[root@dev ~]# echo “exec /usr/bin/netmond 2>&1 > /dev/null” >>
/service/netmond/run
```

You can then start the netmond service using “svc –u /service/netmond” and stop it using “svc –d /service/netmond”

Configuring cron jobs

A series of cron jobs are required in order process and extrapolate the raw flow data fed to the server by the clients. The following crontab entries should be installed, assuming that the PHP scripts are located in /home/netmon/cronjobs/

```
* * * * * /usr/bin/php -q /home/netmon/cronjobs/generate.php > /tmp/gen6 2>&1
* * * * * /usr/bin/php -q /home/netmon/cronjobs/host_graphs.php > /tmp/graph 2>&1
* * * * * /usr/bin/php -q /home/netmon/cronjobs/group_graphs.php > /tmp/group 2>&1
* * * * * /usr/bin/php -q /home/netmon/cronjobs/group_graph_totals.php >
          /tmp/group_totals 2>&1
* * * * * /usr/bin/php -q /home/netmon/cronjobs/app_graphs.php > /tmp/app 2>&1
* * * * * /usr/bin/php -q /home/netmon/cronjobs/app_graph_totals.php >
          /tmp/app_totals 2>&1
* * * * * /usr/bin/php -q /home/netmon/cronjobs/alerts.php > /tmp/alerts 2>&1
```

Please note that these cronjobs require PHP and MySQL to be installed on the server in question.

Web-based front end

The final stage of the installation process involves installing a web-based front end to the database system. This front end has been built using PHP and MySQL, and has been designed to run under Apache – thus all three of these applications are pre-requisites.

Installation instructions

1. Unpack front-end.tar.gz in to /home/netmon/html/

   ```
   [root@dev html]# tar -zxf front-end.tar.gz
   ```

2. Ensure that the directory /home/netmon/html and /home/netmon have been configured with executable privileges:

   ```
   [root@dev html]# chmod 755 /home/netmon
   [root@dev html]# chmod 755 /home/netmon/html
   ```

3. Create a Directory Alias in Apache's /etc/httpd/httpd.conf file for allowing web access:

   ```
   Alias /netmon/ "/home/netmon/html/"
   <Directory "/home/netmon/html">
       Options All
       AllowOverride All
       Order allow,deny
       Allow from all
   </Directory>
   ```

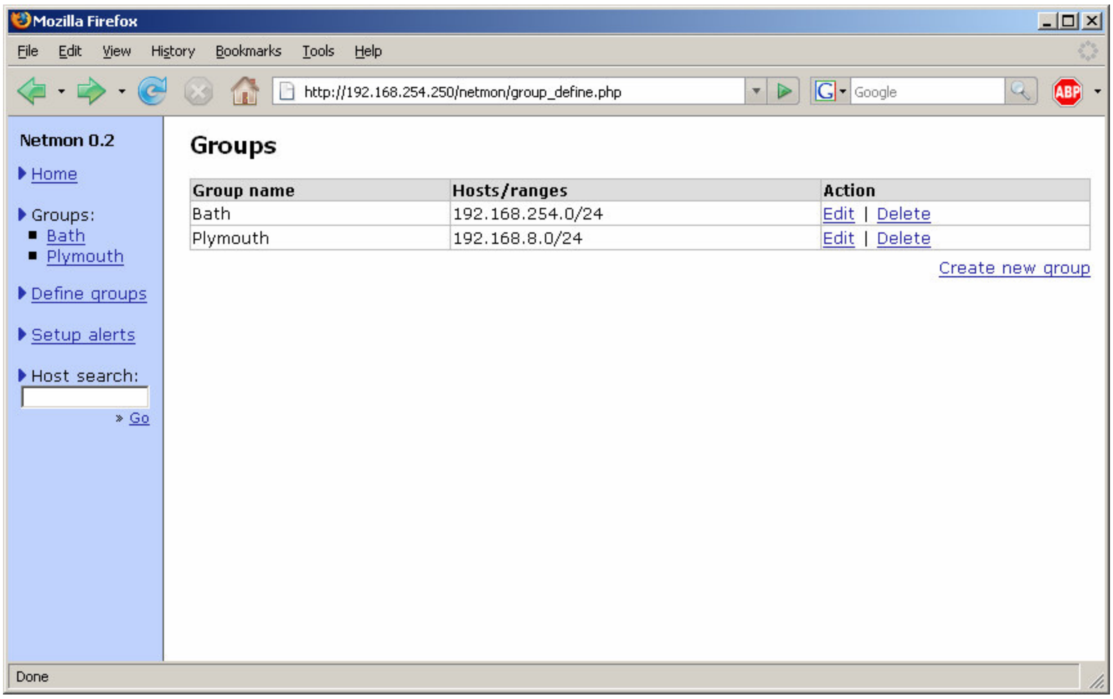
4. Restart Apache using "apachectl restart". The netmon home page should now be available at http://hostname/netmon/

Note: It may be necessary to change the database username, password and hostname in the front end. This can be set in the file common.php in the root of the /html/ directory.

Group configuration

Upon first visiting the netmon front page you will be presented with a near empty display. All tables will be empty, and no graphs will be displayed. This is because the application does not know the boundary of your network – this is, it does not know which parts of the network are yours. Whilst it would be feasible to map in detail every host you communicate with, it is more likely that you will be instead interested only in your own network.

Therefore, the first task that you should conduct following installation is the definition of groups. Each group defines the boundary of each sub-network that you wish to monitor distinctly. Navigate to the "Define Groups" option from the front page to begin the group definition process.



Groups

Group name	Hosts/ranges	Action
Bath	192.168.254.0/24	Edit | Delete
Plymouth	192.168.8.0/24	Edit | Delete

Create new group

In the above screenshot, two sub-networks are defined in the CIDR format:

1. 192.168.254.0/24 (Same as 192.168.254.0 – 192.168.254.254)
2. 192.168.8.0/24 (Same as 192.168.8.0 – 192.168.8.254)

Multiple ranges can be defined per group too. So, for example, the Bath group could be defined by 192.168.254.0/24 and 80.77.250.232/29 (a public IP range running from 80.77.250.232 to 80.77.250.240).

As soon as a group is defined, data and graphs will start being collected and generated.

Modules configuration

It may be undesirable in certain circumstances to monitor a specific protocol on your network or part of your network. For example, you may run a business network and therefore not expect users to engage in BitTorrent and other peer-to-peer application usage. Therefore, in order to save on capture overhead and clutter on the front end, it would be best to disable this classification module altogether on the capture client in question.

This can be achieved by commenting out the relevant line in /etc/netmon_modules and restarting the netmonc application. For example, if one no longer wanted to monitor BitTorrent traffic distinctly on a specific network that the host clienta was monitoring, one would do the following:

1. Open /etc/netmon_modules in a text editor such as vi and place a # in front of "bittorrent". The file should then look something like:

   ```
   [root@clienta etc]# cat /etc/netmon_modules
   http
   #bittorrent
   https
   dns
   smtp
   ```

2. Restart netmonc. Assuming the optional installation instructions were followed, this can be achieved simply by executing the following:

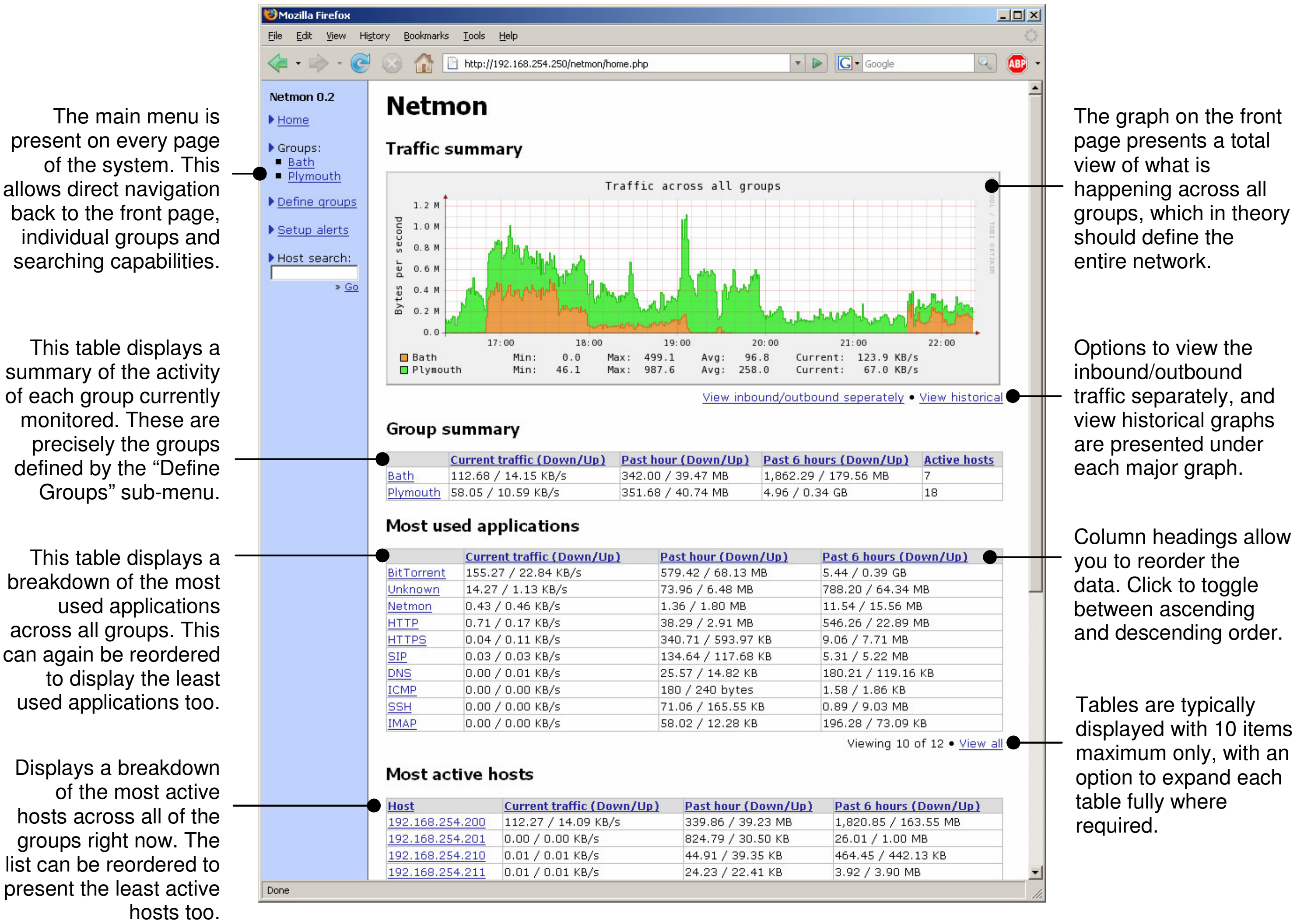
   ```
   [root@clienta etc]# svc -t /service/netmonc
   ```

Note: Disabling a module does not mean that the traffic is simply ignored. It is instead grouped together as "Unknown" (Unclassified) traffic. Too much "Unknown" traffic on your graphs may indicate that a new protocol is emerging and it is time to consider installing new classification modules.

Using the front end

Front page

The system has been designed to be simple enough such that little or no explanation is required for many of the functions. The following image is a screen capture of the front page of the system, complete with annotations of each major function.



Netmon

Traffic summary

```
🟧 Bath        Min:   0.0   Max:  499.1   Avg:   96.8   Current:  123.9 KB/s
🟩 Plymouth    Min:  46.1   Max:  987.6   Avg:  258.0   Current:   67.0 KB/s
```

View inbound/outbound seperately • View historical

Group summary

	Current traffic (Down/Up)	Past hour (Down/Up)	Past 6 hours (Down/Up)	Active hosts
Bath	112.68 / 14.15 KB/s	342.00 / 39.47 MB	1,862.29 / 179.56 MB	7
Plymouth	58.05 / 10.59 KB/s	351.68 / 40.74 MB	4.96 / 0.34 GB	18

Most used applications

	Current traffic (Down/Up)	Past hour (Down/Up)	Past 6 hours (Down/Up)
BitTorrent	155.27 / 22.84 KB/s	579.42 / 68.13 MB	5.44 / 0.39 GB
Unknown	14.27 / 1.13 KB/s	73.96 / 6.48 MB	788.20 / 64.34 MB
Netmon	0.43 / 0.46 KB/s	1.36 / 1.80 MB	11.54 / 15.56 MB
HTTP	0.71 / 0.17 KB/s	38.29 / 2.91 MB	546.26 / 22.89 MB
HTTPS	0.04 / 0.11 KB/s	340.71 / 593.97 KB	9.06 / 7.71 MB
SIP	0.03 / 0.03 KB/s	134.64 / 117.68 KB	5.31 / 5.22 MB
DNS	0.00 / 0.01 KB/s	25.57 / 14.82 KB	180.21 / 119.16 KB
ICMP	0.00 / 0.00 KB/s	180 / 240 bytes	1.58 / 1.86 KB
SSH	0.00 / 0.00 KB/s	71.06 / 165.55 KB	0.89 / 9.03 MB
IMAP	0.00 / 0.00 KB/s	58.02 / 12.28 KB	196.28 / 73.09 KB

Viewing 10 of 12 • View all

Most active hosts

Host	Current traffic (Down/Up)	Past hour (Down/Up)	Past 6 hours (Down/Up)
192.168.254.200	112.27 / 14.09 KB/s	339.86 / 39.23 MB	1,820.85 / 163.55 MB
192.168.254.201	0.00 / 0.00 KB/s	824.79 / 30.50 KB	26.01 / 1.00 MB
192.168.254.210	0.01 / 0.01 KB/s	44.91 / 39.35 KB	464.45 / 442.13 KB
192.168.254.211	0.01 / 0.01 KB/s	24.23 / 22.41 KB	3.92 / 3.90 MB

The main menu is present on every page of the system. This allows direct navigation back to the front page, individual groups and searching capabilities.

This table displays a summary of the activity of each group currently monitored. These are precisely the groups defined by the "Define Groups" sub-menu.

This table displays a breakdown of the most used applications across all groups. This can again be reordered to display the least used applications too.

Displays a breakdown of the most active hosts across all of the groups right now. The list can be reordered to present the least active hosts too.

The graph on the front page presents a total view of what is happening across all groups, which in theory should define the entire network.

Options to view the inbound/outbound traffic separately, and view historical graphs are presented under each major graph.

Column headings allow you to reorder the data. Click to toggle between ascending and descending order.

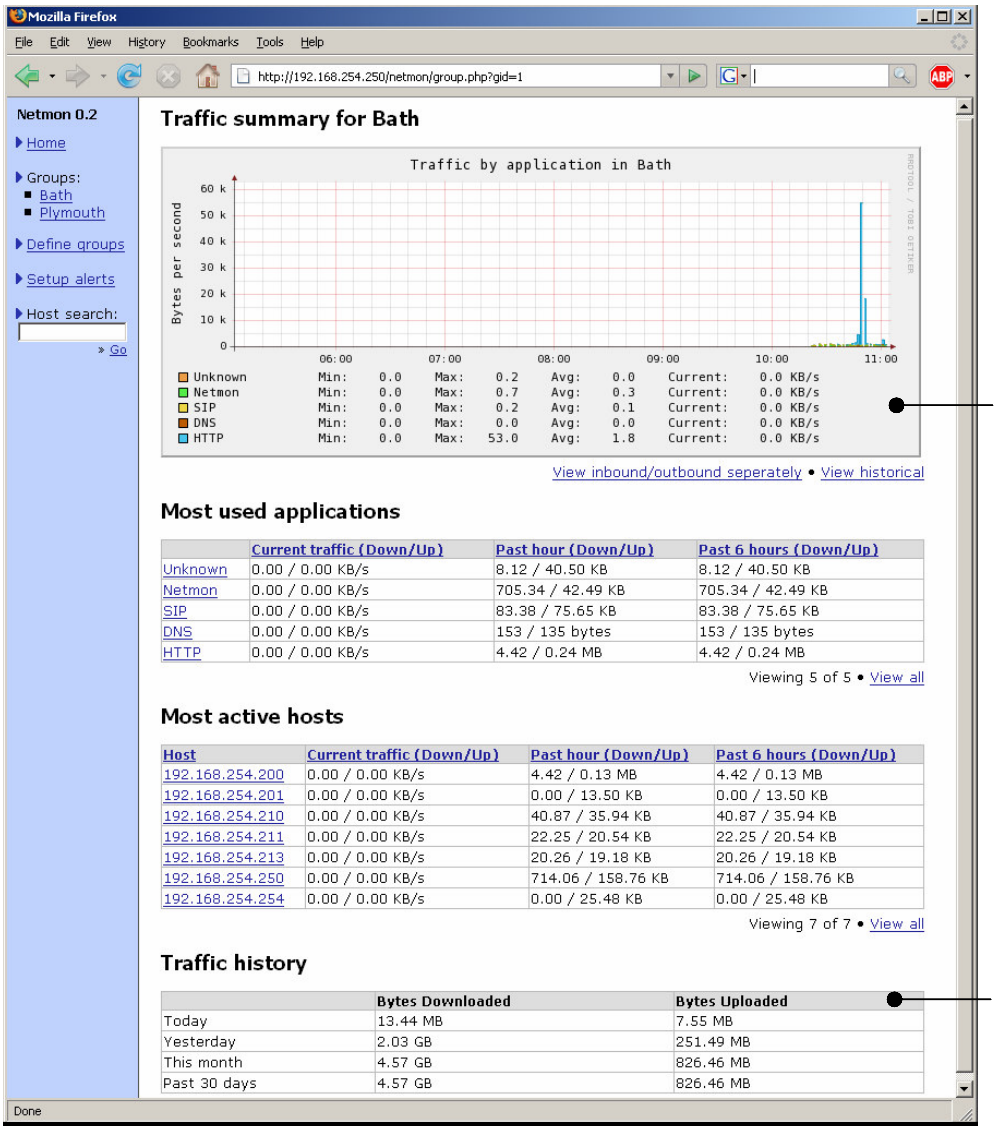
Tables are typically displayed with 10 items maximum only, with an option to expand each table fully where required.

As can be seen from the above, from the front page you can immediately access information about specific groups, applications and even individual hosts.

Each data table only provides detailed statistics for the current traffic, the past hours traffic and the past 6-hours traffic. Older information can be found by clicking on the item you are interested in, and either (a) viewing the historical traffic graphs, or (b) viewing the traffic history table at the bottom of the page.

Group level information

The group information page provides data pertaining to a specific group, as defined earlier under the “Define groups” menu option. On these pages, only information about hosts belonging to that group will be displayed. This allows you, the user, with sufficient granularity to focus your attention on specific groups, rather than having to look at all of the data globally. As discussed earlier, grouping may also follow intuitively from the logical layout of your network.



Traffic summary for Bath

```
🟧 Unknown    Min:   0.0    Max:   0.2    Avg:   0.0    Current:   0.0 KB/s
🟩 Netmon     Min:   0.0    Max:   0.7    Avg:   0.3    Current:   0.0 KB/s
🟨 SIP        Min:   0.0    Max:   0.2    Avg:   0.1    Current:   0.0 KB/s
🟫 DNS        Min:   0.0    Max:   0.0    Avg:   0.0    Current:   0.0 KB/s
🟦 HTTP       Min:   0.0    Max:  53.0    Avg:   1.8    Current:   0.0 KB/s
```

View inbound/outbound seperately • View historical

Most used applications

	Current traffic (Down/Up)	Past hour (Down/Up)	Past 6 hours (Down/Up)
Unknown	0.00 / 0.00 KB/s	8.12 / 40.50 KB	8.12 / 40.50 KB
Netmon	0.00 / 0.00 KB/s	705.34 / 42.49 KB	705.34 / 42.49 KB
SIP	0.00 / 0.00 KB/s	83.38 / 75.65 KB	83.38 / 75.65 KB
DNS	0.00 / 0.00 KB/s	153 / 135 bytes	153 / 135 bytes
HTTP	0.00 / 0.00 KB/s	4.42 / 0.24 MB	4.42 / 0.24 MB

Viewing 5 of 5 • View all

Most active hosts

Host	Current traffic (Down/Up)	Past hour (Down/Up)	Past 6 hours (Down/Up)
192.168.254.200	0.00 / 0.00 KB/s	4.42 / 0.13 MB	4.42 / 0.13 MB
192.168.254.201	0.00 / 0.00 KB/s	0.00 / 13.50 KB	0.00 / 13.50 KB
192.168.254.210	0.00 / 0.00 KB/s	40.87 / 35.94 KB	40.87 / 35.94 KB
192.168.254.211	0.00 / 0.00 KB/s	22.25 / 20.54 KB	22.25 / 20.54 KB
192.168.254.213	0.00 / 0.00 KB/s	20.26 / 19.18 KB	20.26 / 19.18 KB
192.168.254.250	0.00 / 0.00 KB/s	714.06 / 158.76 KB	714.06 / 158.76 KB
192.168.254.254	0.00 / 0.00 KB/s	0.00 / 25.48 KB	0.00 / 25.48 KB

Viewing 7 of 7 • View all

Traffic history

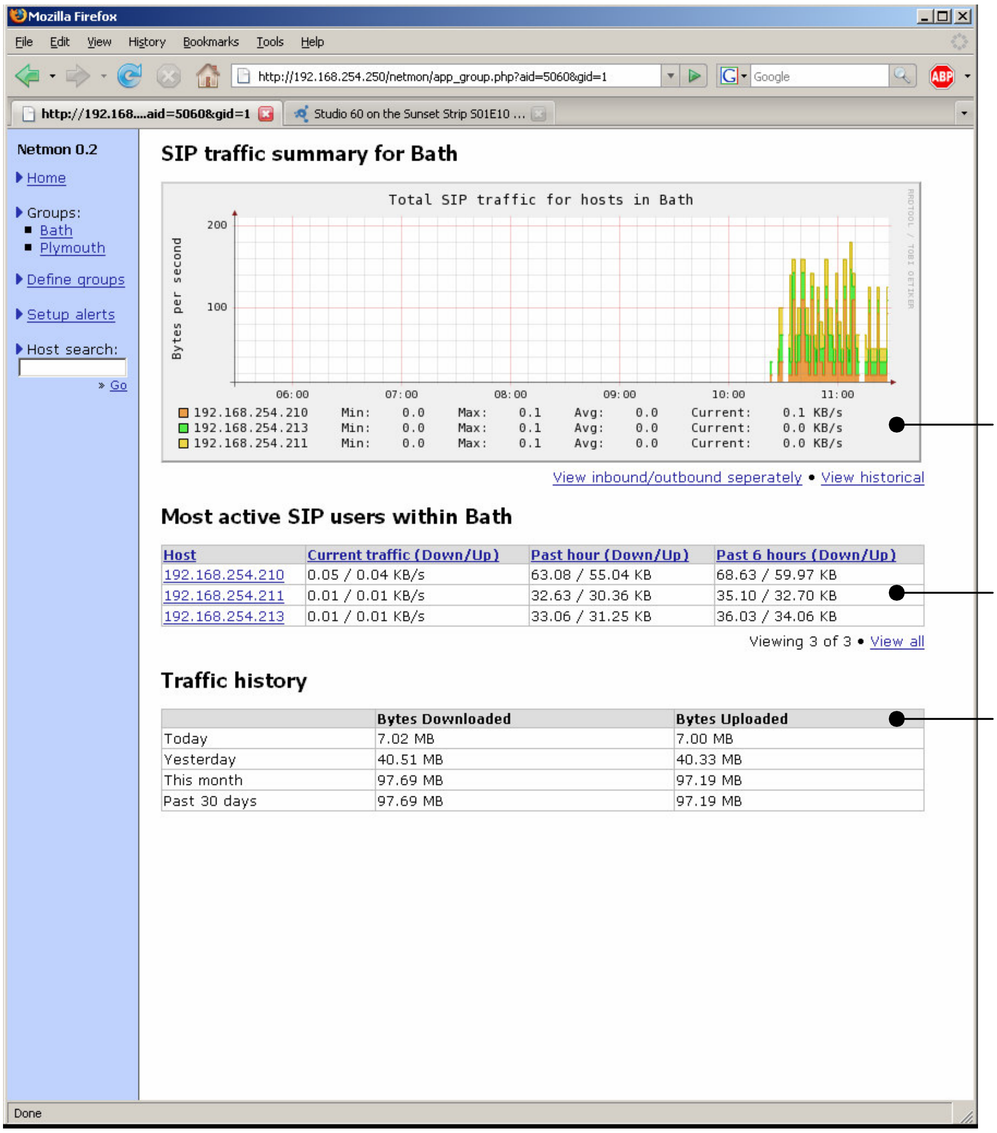
	Bytes Downloaded	Bytes Uploaded
Today	13.44 MB	7.55 MB
Yesterday	2.03 GB	251.49 MB
This month	4.57 GB	826.46 MB
Past 30 days	4.57 GB	826.46 MB

Graphs on the group level pages are specific to the group in question, and are split by the application usage. Only applications used by the group are displayed.

The traffic history table provides an at-a-glance view of the groups traffic usage today, yesterday and for the past month.

Application level information

Taking a different view of the data and looking at it by application may provide you with a new viewpoint on which applications are increasing in popularity and which are on the decline. The system supports viewing data at the application level, as demonstrated by the following screenshot of SIP (A standard protocol for Voice over IP) traffic in the Bath group.



Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.254.250/netmon/app_group.php?aid=5060&gid=1
Google
http://192.168....aid=5060&gid=1
Studio 60 on the Sunset Strip S01E10 ...
Netmon 0.2
Home
Groups:
Bath
Plymouth
Define groups
Setup alerts
Host search:
» Go
Total SIP traffic for hosts in Bath
Bytes per second
200
100
06:00
07:00
08:00
09:00
10:00
11:00
RRDTOOL / TOBI OETIKER
Done

SIP traffic summary for Bath

```
🟧 192.168.254.210   Min:   0.0   Max:   0.1   Avg:   0.0   Current:   0.1 KB/s
🟩 192.168.254.213   Min:   0.0   Max:   0.1   Avg:   0.0   Current:   0.0 KB/s
🟨 192.168.254.211   Min:   0.0   Max:   0.1   Avg:   0.0   Current:   0.0 KB/s
```

View inbound/outbound seperately • View historical

Most active SIP users within Bath

Host	Current traffic (Down/Up)	Past hour (Down/Up)	Past 6 hours (Down/Up)
192.168.254.210	0.05 / 0.04 KB/s	63.08 / 55.04 KB	68.63 / 59.97 KB
192.168.254.211	0.01 / 0.01 KB/s	32.63 / 30.36 KB	35.10 / 32.70 KB
192.168.254.213	0.01 / 0.01 KB/s	33.06 / 31.25 KB	36.03 / 34.06 KB

Viewing 3 of 3 • View all

Traffic history

	Bytes Downloaded	Bytes Uploaded
Today	7.02 MB	7.00 MB
Yesterday	40.51 MB	40.33 MB
This month	97.69 MB	97.19 MB
Past 30 days	97.69 MB	97.19 MB

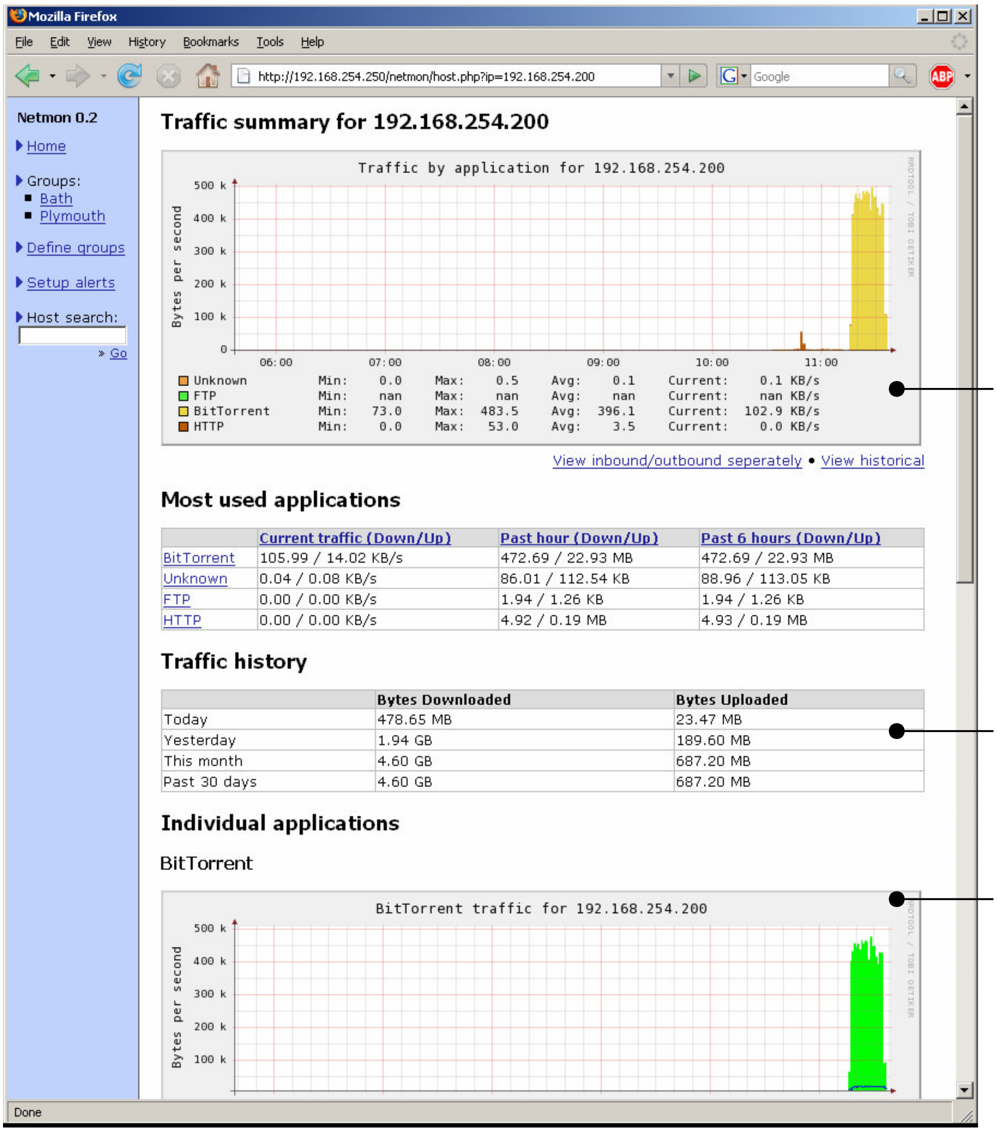
The graph here displays only hosts within Bath that have been actively involved in SIP traffic.

As before, only hosts involved in SIP communications are displayed in the data tables

Historical traffic information here is limited to the SIP protocol

Host level information

The lowest of the available levels to view data from is the host level. This provides application usage information pertaining to a single host, along with detail historical information. This will be particularly useful when diagnosing a problem with a certain host, or determining the source of traffic spikes.



Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.254.250/netmon/host.php?ip=192.168.254.200
Google
ABP
Done

Netmon 0.2

- Home
- Groups:
  - Bath
  - Plymouth
- Define groups
- Setup alerts
- Host search:

» Go

Traffic summary for 192.168.254.200

Traffic by application for 192.168.254.200
Bytes per second
500 k
400 k
300 k
200 k
100 k
0
06:00
07:00
08:00
09:00
10:00
11:00

```
🟧 Unknown       Min:   0.0   Max:   0.5   Avg:   0.1   Current:   0.1 KB/s
🟩 FTP           Min:   nan   Max:   nan   Avg:   nan   Current:   nan KB/s
🟨 BitTorrent    Min:  73.0   Max: 483.5   Avg: 396.1   Current: 102.9 KB/s
🟫 HTTP          Min:   0.0   Max:  53.0   Avg:   3.5   Current:   0.0 KB/s
```

View inbound/outbound seperately • View historical

Most used applications

	Current traffic (Down/Up)	Past hour (Down/Up)	Past 6 hours (Down/Up)
BitTorrent	105.99 / 14.02 KB/s	472.69 / 22.93 MB	472.69 / 22.93 MB
Unknown	0.04 / 0.08 KB/s	86.01 / 112.54 KB	88.96 / 113.05 KB
FTP	0.00 / 0.00 KB/s	1.94 / 1.26 KB	1.94 / 1.26 KB
HTTP	0.00 / 0.00 KB/s	4.92 / 0.19 MB	4.93 / 0.19 MB

Traffic history

	Bytes Downloaded	Bytes Uploaded
Today	478.65 MB	23.47 MB
Yesterday	1.94 GB	189.60 MB
This month	4.60 GB	687.20 MB
Past 30 days	4.60 GB	687.20 MB

Individual applications

BitTorrent

This graph displays data relating to the application usage of 192.168.254.200. Only applications that this host has used are displayed.

Historical traffic information here is limited to the host in question

A more granular view of each applications usage by this host is also available, with solid green areas representing bytes in, and the blue line representing bytes out.

Alerting

The alerting sub-system provides you with the ability to have the system monitor specific parts of the network for you, and alert you should they deviate from within your defined parameters.

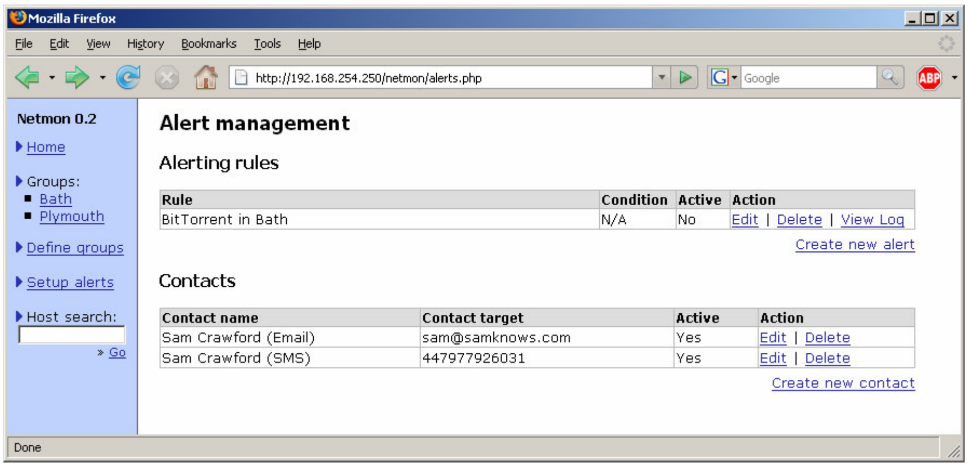
Configuring alerts

Alerts may be defined on any single one of, or a combination of, the following:

- Specific hosts or ranges of hosts
- Specific application traffic usage
- Traffic exceeding or falling below defined thresholds
- Traffic averages being based upon current, past-hour, past-six-hours or past-24-hour averages (Choosing an average rather than current traffic should help reduce false alerts)
- Specific times of the day, with the option of disabling the alert at the weekend

The user must also configure contacts within the alerting system. Each of these contacts may take the form of an email address or mobile phone number (to which SMS messages will be sent). With the advent of email-to-pager and even email-to-fax systems, users of the alerting system may easily extend alerts to send to devices other than email clients and mobile phones too.

To configure alerts, click on the "Setup alerts" link on the main menu. You will be presented with a page similar to the one in the following screenshot.



Alert management

Alerting rules

Rule	Condition	Active	Action
BitTorrent in Bath	N/A	No	Edit | Delete | View Log

Create new alert

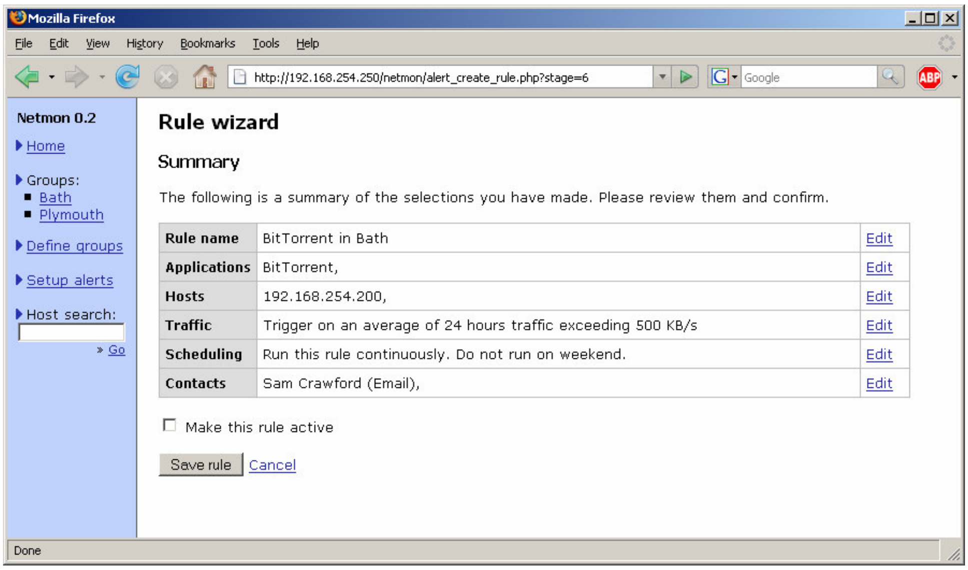
Contacts

Contact name	Contact target	Active	Action
Sam Crawford (Email)	sam@samknows.com	Yes	Edit | Delete
Sam Crawford (SMS)	447977926031	Yes	Edit | Delete

Create new contact

In the above, a single "alerting rule" has been created as well as two contacts – one SMS and one email based. Note that the single rule "BitTorrent in Bath" is inactive, but the two contacts are active. Setting a rule to be inactive simply means that it is not checked, and thus no alerts are ever dispatched for it.

Choosing to edit the rule “BitTorrent in Bath” results in you being presented with the following screen.


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.254.250/netmon/alert_create_rule.php?stage=6
Google
Netmon 0.2
Home
Groups:
Bath
Plymouth
Define groups
Setup alerts
Host search:
» Go
Done

Rule wizard

Summary

The following is a summary of the selections you have made. Please review them and confirm.

Rule name	BitTorrent in Bath	Edit
Applications	BitTorrent,	Edit
Hosts	192.168.254.200,	Edit
Traffic	Trigger on an average of 24 hours traffic exceeding 500 KB/s	Edit
Scheduling	Run this rule continuously. Do not run on weekend.	Edit
Contacts	Sam Crawford (Email),	Edit

☐ Make this rule active

Save rule Cancel

From this central screen you can edit each individual part of the rule and also activate or deactivate the entire rule. This rule can be read in the following way:

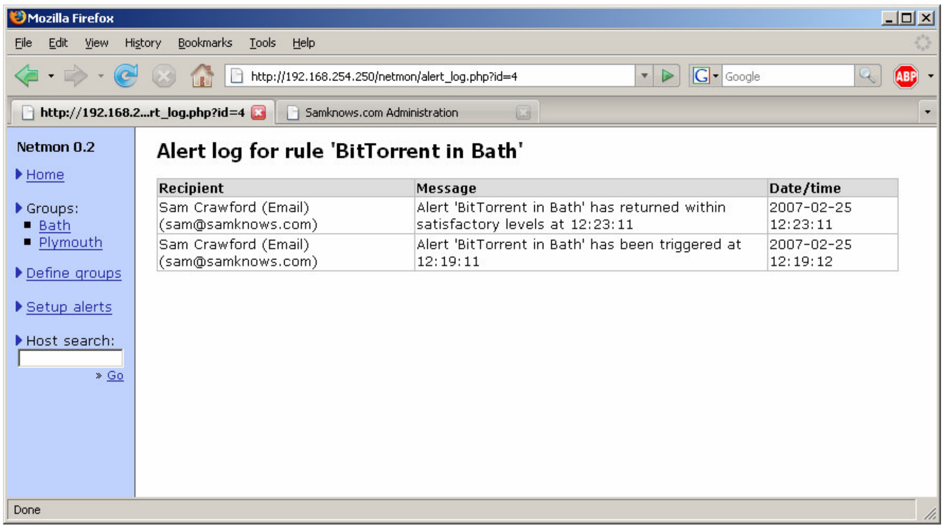
“Send alerts to Sam Crawford via Email whenever BitTorrent traffic to/from 192.168.254.200 exceeds an average of 500KB/s over the past 24 hours. But do not send any alerts over the weekend”

Clicking edit on any individual section provides the user with a web-form (pre-populated with the currently selected information) that allows them to alter their selections. Creating a new rule follows precisely the same process, except that all of the sections are initially empty.

Note: When an alert is triggered, it is sent only once to all recipients. It does not continually send for the duration of the issue. When traffic does eventually return to within normal levels, another alert is sent to all parties stating that this fact. This has been designed to prevent flooding users with emails and/or SMS messages.

Alerting logs

Whenever an alert is triggered, an entry is recorded in an alerting log. This is viewable by clicking on the “Alert log” link next to each alert in the main table. This will present you with a screen similar to the following:


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://192.168.254.250/netmon/alert_log.php?id=4
Google
http://192.168.2...rt_log.php?id=4
Samknows.com Administration
Netmon 0.2
Home
Groups:
Bath
Plymouth
Define groups
Setup alerts
Host search:
» Go
Done

Alert log for rule 'BitTorrent in Bath'

Recipient	Message	Date/time
Sam Crawford (Email) (sam@samknows.com)	Alert 'BitTorrent in Bath' has returned within satisfactory levels at 12:23:11	2007-02-25 12:23:11
Sam Crawford (Email) (sam@samknows.com)	Alert 'BitTorrent in Bath' has been triggered at 12:19:11	2007-02-25 12:19:12

In this example, a single alert was sent at 12:19:12 to sam@samknows.com following the rule “BitTorrent in Bath” triggering. Almost precisely four minutes later the rule returned to within normal levels, so another message was sent.

Note: The alerting system checks the alerting rules once per minute. Therefore, it is possible, when monitoring current traffic and it spikes for only a few seconds, that an alert will not be generated.

Appendix L – Broadband take-up data provided by Point Topic Limited

Country	Population	Households	2003Q3			2006Q3		
			Total	**%Pop**	**%HH**	**Total**	**%Pop**	**%HH**
UK	59,668,000	25,104,000	**1,247,820**	2.09%	4.97%	**12,331,251**	20.67%	49.12%

Data supplied by Oliver Johnson, General Manager, Point Topic Limited. Email provided below.

-----Original Message-----
From: Oliver Johnson [mailto:oliver.johnson@point-topic.com]
Sent: 15 March 2007 07:30
To: Sam Crawford
Subject: Re: Latest invoice

Sam,

Sorry completely forgot about this. Very rude, hope it's not too late.

Let me know if you need more data/detail

Ollie

[Attachments: Broadband total UK.xls, Broadband by operator.xls]

Appendix M – Preliminary feedback from Fluidata

The following are notes taken from a phone discussion with Chris Rogers, Operations Director at Fluidata, on 28th February 2007. Note that this was based on his direct interaction with the system monitoring the test networks (Bath and Plymouth). At this point the system had not gone live on the Fluidata network.

```
Initial feedback from Chris Rogers (28 Feb 2007)

* Too many colours / data series on one graph.
  - Perhaps limit the number of data series and group the smallest
    series together in to an "Others" series?
  - Colours blend in to one another too easily
  - Perhaps some kind of zooming functionality might be possible?

* Unknown traffic can spike to consume large amounts
  - What is this unknown traffic?
  - Requested: option to drill down to see src/dst host/port pairs

* Grouping discussions
  - Chris wants to be able to group data by each individual circuit
    on his network. (Each has varying number of IP addresses
    attached)
  - Estimated ~350 groups to import.
  - Impact on back end RRD/databases? Visual changes to front end?
```

Appendix N – Further feedback from Fluidata

The following are notes taken from a phone discussion with Chris Rogers, Operations Director at Fluidata, on 19th March 2007. Note that this was based on his direct interaction with the system monitoring his own network (Fluidata).

```
Later feedback from Chris Rogers (19 March 2007)
(After system went live on Fluidata's network)

* Reviewed changes since last discussion
  - Detailed break of Unknown traffic now available
  - Insignificant data series grouped in to "Others" on graphs
  - Users can now zoom in to graphs
  - All of Fluidata's groups have been imported (361 at present)

* Chris noted that the system had been used that day to aide in
  providing usage evidence in a customer billing issue

* Chris also confirmed that traffic from 89.105.117.69 (which sees
  large volumes of unknown traffic) is owned by Encrypted Software
  Solutions, and is therefore almost certainly carrying encrypted
  traffic that cannot be classified.

* Chris requested that transfer rates be displayed in bits/sec
  rather than bytes/sec. Cited that this was the industry standard
  when representing network traffic.
  (Changed trivially - Just involved multiplying everything by 8 and
  changing some textual labels)

* Fluidata stated that they were very impressed with the level of
  detail one could drill down to
```

Appendix O – Test system specifications

The following systems were used to develop and test the system.

System 3 – Installed on Fluidata's core network (1Gbps connection)

HP/Compaq Proliant DL360 G4p
Dual 3.0Ghz Intel Xeon CPUs, with 2MB cache each
4GB DDR2 ECC memory
Two 73GB 10,000RPM U320 SCSI disks in RAID-1 (mirror)
Hardware RAID controller
Dual gigabit network interface cards
Running CentOS 4.4 Linux (64-bit SMP kernel)
Kernel: 2.6.9-42.0.8.ELsmp x86_64

System 2 – Installed on a student network in Bath (100Mbps connection)

Dual 1.4Ghz Intel Pentium 3 "Tualatin" CPUs, with 512KB cache each
1.5GB PC133 ECC SDRAM
Two 18GB 10,000RPM U160 SCSI disks in RAID-1 (mirror)
Hardware RAID controller
Dual 100Mbps network interface cards
Running CentOS 4.4 Linux (32-bit SMP kernel)
Kernel: 2.6.9-42.0.10.plus.c4smp i386/i686

System 1 – Installed on a student network in Plymouth (100Mbps connection)

Intel Pentium 2 400Mhz CPU
128MB PC100 SDRAM
40GB IDE hard disk drive
Dual 100Mbps network interface cards
Running CentOS 4.4 Linux (32-bit uni-processor kernel)
Kernel: 2.6.9-42.0.10.plus.c4 i386/i686

Appendix P – Demonstrating a real-world use

```
-------- Original Message --------
Subject:    thanks :)
Date:       Tue, 10 Apr 2007 12:37:53 +0100
From:       Chris Rogers | Fluidata <ChrisRogers@fluidata.co.uk>
To:         Sam Crawford <sam@samknows.com>
```

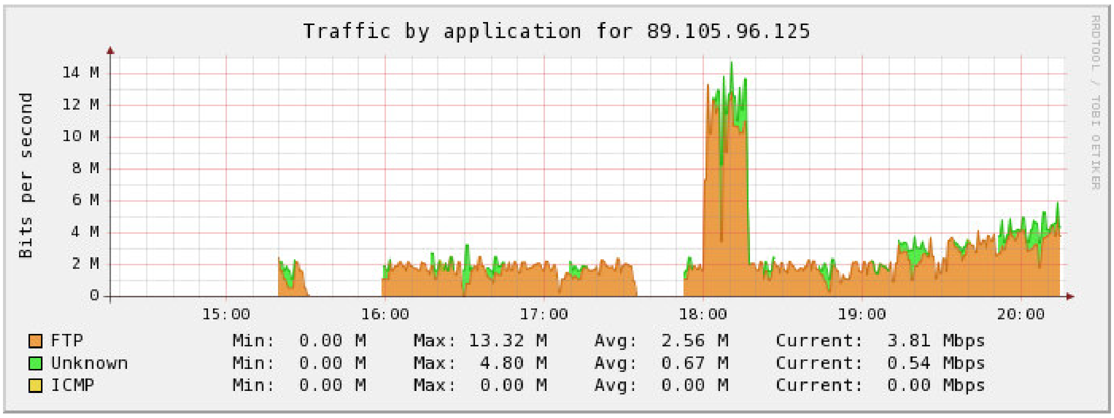
You’ll be pleased to hear that thanks to your project I was able to isolate suspicious spikes in upload over the weekend, track it down to one of our servers, and see that it was ftp traffic. Was quickly able to close off anonymous ftp service that someone here had left open, who shall remain anonymous!

Would probably never have tracked that down without it!

Cheers,

Chris

-------- Attached Image --------



Appendix Q – Fluidata's final feedback

Fluidata

18th April, 2007

Hi Sam,

I've really enjoyed testing your project over the past weeks, and as requested I'm laying out my feedback from this experience.

There's no question that this is an invaluable tool certainly for an ISP, and I also think any SME large enough to have dedicated IT staff would find this useful. It affords a visibility on network activity that would normally be very costly to achieve, and completely beyond the reach of small businesses.

We have previously evaluated Packeteer, which is arguably the best in its class, but the cost of deploying it stretched into many tens of thousands of pounds. We couldn't make the commercial case for that cost given that merely want to observe rather than shape traffic.

The ability to keep a close eye on both core network devices and clients usage has already proven its worth to me personally on a number of occasions. For example, when a client called in for support complaining that they were suffering from degraded VoIP calls we were able to inspect their traffic and inform them that there was a large amount of http traffic hogging the bandwidth. They went away satisfied and everyone was relived and better informed.

That type of problem is normally very tiresome to troubleshoot, and often ends up with a standoff between ISP vs client, with the ISP simply insisting that the fault is at the client end but unable to prove it. Both parties suffer, as the problem remains unresolved and increases the likelihood that the client will move to another ISP. To be able to rapidly pull off concrete evidence of what the traffic is actually doing and present it to the client is phenomenally useful.

On another occasion a client was complaining that their bandwidth was maxed out, but said they were not using the traffic. This was a 4 Mbps leased line, and we were rapidly able to inform them that they were pushing out 2 Mbps of smtp traffic, suggesting a seriously compromised host.

But both of these examples were prompted by client complaint first, rather than any alert from the software. To be frank I haven't experimented with the alerting feature of your software, and so am not placed to give feedback on it. However, the fact that I haven't bothered is a reflection of the sheer amount of effort I know it would involve. This is definitely my strongest criticism of the software, as I feel it limits the usefulness of the alerting feature.

In the client sample used for this test there's simply no way I would sit down and configure alerts for hundreds of clients for different traffic types. If we were to roll this software out to clients it's possible they would configure individual alerts for themselves, but not likely from my experience. And more to the point, compromised hosts sitting on our clients' internet connections are issues we need to keep an eye on rather than them: viruses, SPAM, and DoS attacks originating from our network will result in damage to our online standing, and likely blocked IPs, which would cause us huge issues and ruin our reputation.

During our testing of your software there was one occasion when I spotted a suspicious traffic spike on one of our transit feeds on Cacti. This was a spike of about 15 Mbps of upload above background traffic, and it was a bank holiday weekend which is always our quietest time. So this which was very irregular for us, especially given that most clients don't have the capacity to generate that much upload. Using your software I was able to track down the host responsible very quickly, which turned out to be one of our own servers on our network, and then identify straight away that it was ftp traffic. Jumping onto the server a quick netstat showed me some interesting ftp sessions open, and I promptly shut down ftp. A more thorough sift through the server revealed an anonymous ftp vulnerability which was closed off.

Alerting for this kind of activity would have been very useful, as it was more by luck that I came across it on Cacti. But configuring alerting for each core network device would be very time consuming, and some levels would need to be revised regularly.

The need for an automated way of setting up alerts is definitely there, and is missing from the market place to the best of my knowledge. This is a shame, because the benefit of informative alerting is invaluable. Also, it is not an impossible task – whilst traffic levels are almost never constant they often change fairly predictably over time.

Whilst this would be a very useful addition I cannot fault the software as it is. It is simple and intuitive to use. The grouping feature has allowed us to group together hosts by connection, making it easy to drill down into individual clients and see what is going on. And the zooming function makes it much clearer to see exactly what is happening. The individual graphs per host and per type of traffic also bring about useful clarity.

The reporting of most active hosts and most used applications is useful, as we can keep a close eye on potential troublemakers. Also, being able to see what ports are being used for the "unknown" traffic would undoubtedly be vital for network administrators needing to keep up with the latest peer-to-peer software etc. As mentioned with the alerting I feel some useful heuristics could be deployed here, keeping watch for trends in unknown ports to define new applications, although the rules would be tricky to define.

Of course, I should also congratulate you on the very ability to track different types of data. I have to confess I had never fully appreciated the difficulty in tracking ftp and bit torrent data. But, of course, bit torrent is engineered to avoid detection by port hopping, so to keep track of it so effectively is a real achievement. Even though we are a business only ISP, with not a single home user on the network, I was still amused to see bit torrent featuring in the top 10 of the most used applications on the network every single day!

This brings me to my final comment, which is that we have been suitably impressed by your software to want to integrate it into our customer portal. At the moment we give clients access to graphs of their bandwidth usage as a free service, and we regularly get positive feedback about how useful IT managers find this. But to be able to throw in free breakdown of that traffic by type would, I have no doubt, blow the minds of many of our clients. Although I’m not sure if this would work to our benefit in all cases – we have one client who currently uses a solid 8 Mbps of traffic and is looking to upgrade to a LES 100 tail, he might well think twice if he could see just how much of his office traffic is peer-to-peer!

Yours sincerely,

Chris Rogers

Operations Director
Fluidata

Appendix R – Code index and samples

The source code is split in to four distinct parts (shown below), which are presented on the attached CD in directories of the same name.

1. **Client (also known as "netmonc")**
   This is where the source code for the capture and analysis client resides. Included with this are the individual application detection modules (under /client/libs/). The code in these directories is the primary focus of the project, as it deals with the bulk of the traffic capture, signature analysis and flow reporting back to the server.

2. **Server (also known as "netmond")**
   This is where the source code for the server resides. This is comparatively very simple to the client, as it simply reads data from a connection, interprets it, and outputs SQL.

3. **HTML**
   This is where the HTML/PHP/CSS source code resides for the front end.

4. **Cronjobs**
   The system relies upon a series of PHP-scripted cronjobs that run every minute or five minutes. This directory contains each of the cronjobs and a listing of how and when they should be run.

Attached below in the rest of this appendix is a ***sample*** of some of the client code.

Please note that the following code is only a sample of the client code. The full code can be found on the attached source code CD.

main.c

Please note that the following code is only a sample of the client code. The full code can be found on the attached source code CD.

```
/**
 * main.c
 * Version: 0.3
 * Author:  Sam Crawford
 *
 * Main packet capture, analysis and reporting program.
 * Captures packets through libpcaps callback method,
 * determines application type using signature analysis &
 * previous flows, records new traffic flows and periodically
 * phones home to report flow statistics.
 */

#include<stdlib.h>
#include<net/ethernet.h>
#include<netinet/in.h>
#include<netinet/tcp.h>
#include<netinet/udp.h>
#include<linux/if_arp.h>
#include<linux/ip.h>
#include<stdio.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netdb.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <signal.h>
#include <gc.h>

#include "gc.h"
#include "capture.h"
#include "flow.h"
#include "hash.h"
#include "netmon.h"
#include "appdetect.h"

#define NETMONC_VER "0.3"
#define NETMONC_BUILT "2007-02-25"

char* netmond_server;        /* Server hostname */
int   netmond_port;          /* Server port */
uint32_t total_packets = 0;  /* Total packets handled */
time_t lastsend = 0;         /* Time of last send to server */
int mode;                    /* Operation mode - live or file */

/**
 * Netmon callback function. Function header defined by libpcaps routines.
 */
void netmon_callback(u_char *args, const struct pcap_pkthdr* header,
                     const u_char* raw)
{
   /* Decode incoming packet */
   struct packet pkt;
   decode_packet(&pkt, raw);

   /* Only handle IP packets */
   if (IS_IP_PACKET(&pkt))
   {
      /* Fetch addresses, protocol, etc */
      IPFlow* flow   = NULL;
      uint32_t saddr = (uint32_t) (&pkt)->nh.iph->saddr;
      uint32_t daddr = (uint32_t) (&pkt)->nh.iph->daddr;
      uint16_t sport = 0;
      uint16_t dport = 0;
      uint16_t app   = 0;
      uint16_t proto = (&pkt)->nh.iph->protocol;

      /* Fetch src and dst ports */
      if (IS_TCPIP_PACKET(&pkt)) {
         sport = (&pkt)->h.th->source;
         dport = (&pkt)->h.th->dest;
      }
      if (IS_UDPIP_PACKET(&pkt)) {
         sport = (&pkt)->h.uh->source;
         dport = (&pkt)->h.uh->dest;
      }

      /* Find application */
      app = app_detect(&pkt);

      /* Lookup existing flow */
      flow = hash_lookup_flow(saddr,daddr,sport,dport,proto);
      total_packets++;

      /* If previous flow exists... */
      if (flow) {
         /* Set application type if not already done */
         if (flow->application == 0 && app > 0)
            flow->application = app;
         flow->len += (&pkt)->nh.iph->tot_len;   /* Increment counters */
         flow->packets++;
      } else {
         /* Create a new flow and record it */
         uint16_t len = (&pkt)->nh.iph->tot_len;
         flow = flow_create(saddr,daddr,sport,dport,len,1,proto,app);
         hash_add_flow(flow);
      }

      /* If we've reached capacity in our flow table, or haven't phoned home
          * in over 60 seconds, then look to see if we need to perform a push
       */
      if (mode == MODE_LIVE &&
          (total_packets == PACKET_CHUNK_LIMIT || (total_packets %10000 == 0
                  && time(NULL) > lastsend+60))) {
         total_packets = 0;
         lastsend = time(NULL);
         netmon_pushdata();      /* Push data to server */
      }
   }
}

/**
 * Connects to netmond server
 * TODO: Add failed connections counter
 */
int netmon_server_connect()
{
   int sockfd;
   struct hostent *he;
   struct sockaddr_in remote_addr;

   /* Lookup hostname */
   if ((he=gethostbyname(netmond_server)) == NULL) {
       fprintf(stderr,"Failed to determine hostname\n");
       return -1;
   }

   /* Create a socket */
   if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
       fprintf(stderr,"Could not create socket\n");
       return -1;
   }

   /* Set socket properties */
   remote_addr.sin_family = AF_INET;
   remote_addr.sin_port = htons(netmond_port);
   remote_addr.sin_addr = *((struct in_addr *)he->h_addr);
   memset(&(remote_addr.sin_zero), '\0', 8);

   /* Connect to server */
   if (connect(sockfd, (struct sockaddr *)&remote_addr,
                            sizeof(struct sockaddr)) == -1) {
       fprintf(stderr,"Could not connect to server!\n");
       return -1;
   }
   return sockfd;
}
```

```
/**
 * Debug method for writing flow information directly to database query
 * (thus bypassing the server). May be more efficient in a *very* high
 * traffic environment, as there is no overhead in communicating flows to
 * the server
 */
void writedb(IPFlow *flow)
{
    fprintf(stdout,"INSERT INTO flow (saddr,daddr,sport,dport,len,packets,
                       protocol,application,started,expires) VALUES (");
    fprintf(stdout,"'%s',",inet_ntoa(htonl(flow->saddr)));
    fprintf(stdout,"'%s',",inet_ntoa(htonl(flow->daddr)));
    fprintf(stdout,"%u,%u,%u,%u,%u,%u,FROM_UNIXTIME(%d),
               FROM_UNIXTIME(%d));\n",
            flow->sport,flow->dport,flow->len,flow->packets,flow->protocol,
                   flow->application,flow->started,flow->expires);
    fflush(stdout);
}

/**
 * Push flow data to the server if necessary
 */
void netmon_pushdata()
{
   int connected = 0;
   int sockfd;
   int size=0,n;
   char buf[BUFF_LEN];
   time_t currtime = time(NULL);
   int i;
   if (!USE_STDOUT) bzero(buf,BUFF_LEN);

   /* Loop over hash table */
   for (i=0; i<HASH_SIZE; i++)
   {
       IPFlow *t = hash_table[i];

       /* Loop along linked list from this branch of the hash table */
       while(t)
       {
          IPFlow *toFree = NULL;
          /* Has this flow expired? */
          /* If so, remove it from our cache and send it! */
          if (mode == MODE_FILE || t->expires <= currtime ||
                          t->len >= FLOW_EXPIRY_BYTES)
          {
             /* If we're not connected, connect to the server */
             if (!USE_STDOUT && !connected) {
                sockfd = netmon_server_connect();
                if (sockfd != -1) connected = 1;
             }
             /* If flow has already been sent, don't resend
              * should never be hit since v0.2
    */
   if (BE_VERBOSE && t->sent == 0) {
      fprintf(stdout,"Sending expired flow: ");
      flow_print(t);
   }

   /* Force a send if required */
   if (connected == 1 && t->sent == 0)
   {
      /* Perform the *actual* send iff our buffer is full */
      if (size >= BUFF_LEN) {
         n = send(sockfd,buf,size,0);
         if (n < 0) {
            fprintf(stderr,"Error writing to socket!\n");
            connected = 0;
         }
         size = 0;
         bzero(buf,BUFF_LEN);
      }
      /* Fudge expiry time to now */
      if (t->expires > currtime) t->expires = currtime;

      /* Add the current flow to the buffer */
      memcpy(&buf[size], t, 32);
      size += 32; // 32 bytes
      t->sent = 1;

   /* If in debug mode, print the flow directly to stdout */
   } else if (USE_STDOUT && t->sent == 0) {
      if (t->expires > currtime) t->expires = currtime;
      writedb(t);
   }

   /* Remove the current flow from the
    * hash table and linked list
    */
   toFree = t;
   if (t->prev == NULL) {
      hash_table[i] = t->next;
      if (hash_table[i] != NULL)
         hash_table[i]->prev = NULL;
   } else {
      t->prev->next = t->next;
   }
   if (t->next != NULL) {
      t->next->prev = t->prev;
   }

}
t = t->next;

/* Free node from memory if we're done */
if(toFree != NULL) free(toFree);
```

```
       }
   }
   /* Send any remaining data in the buffer */
   if (connected == 1 && size > 0) {
      n = send(sockfd,buf,size,0);
      if (n < 0) {
         fprintf(stderr,"Error writing to socket!\n");
         connected = 0;
      }
   }
   if (!USE_STDOUT) close(sockfd);
}

/**
 * main() function for the netmon client application
 */
int main(int argc, char **argv)
{
   char *file = NULL;
   char *dev = "eth0";
   pcap_t *pcap;
   netmond_server = DEFAULT_SERVER;
   netmond_port = DEFAULT_PORT;

   /* Check arguments */
   if (argc == 2 && strcmp(argv[1],"--help") == 0) {
      fprintf(stderr,"Usage: netmonc [-i interface] [-h server] [-p port]\n");
      exit(1);
   } else {
      int i;
      for(i=1; i<argc; i++) {
         if (strcmp(argv[i],"-i") == 0) {           /* Interface selection */
            dev = argv[++i];
         } else if (strcmp(argv[i],"-h") == 0) {   /* Server hostname */
            netmond_server = argv[++i];
         } else if (strcmp(argv[i],"-p") == 0) {   /* Server port (TCP) */
            netmond_port = atoi(argv[++i]);
         } else if (strcmp(argv[i],"-f") == 0) {   /* Use a pcap file */
            file = argv[++i];
         } else if (strcmp(argv[i],"-s") == 0) {   /* Debug - use stdout */
            netmond_port = -1;
         }else if (strcmp(argv[i],"-v") == 0) {    /* Be verbose */
            verbosity = 2;
         }
      }
   }

   if (BE_VERBOSE) {
      /* Startup message */
      fprintf(stdout,"netmonc v%s starting up (Built: %s).
                          Monitoring %s, reporting to %s:%d...\n",
              NETMONC_VER,NETMONC_BUILT,dev,netmond_server,netmond_port);
   }
   /* Ignore SIGPIPE and SIGCLD error signals */
   signal(SIGPIPE, SIG_IGN);
   signal(SIGCLD, SIG_IGN);

   init_modules();      /* Initialise dynamic linking modules */
   hash_appdyn_init();  /* Initialise DynamicApp hash tables */
   hash_init();         /* Initialise IP flow hash tables */

   if (BE_VERBOSE) fprintf(stdout,"netmonc v%s started successfully\n",
                              NETMONC_VER);

   /* Create pcap handle */
   if (file == NULL) {
      mode = MODE_LIVE;
      pcap = pcap_open_live(dev,BUFSIZ,1,-1,errbuf);
   } else {
      mode = MODE_FILE;
      pcap = pcap_open_offline(file,errbuf);
   }
   if (!pcap) {
      fprintf(stderr,"Could not open pcap handler: %s\n",errbuf);
      exit(1);
   }

   /* Watch out! This function loops infinitely, calling netmon_callback */
   pcap_loop(pcap,-1,netmon_callback,NULL);
   pcap_close(pcap);

   /* Debug code... don't usually see this being called */
   if (mode == MODE_FILE) {
      printf("Forcing offline data push:\n");
      netmon_pushdata();
   }
   if (BE_VERBOSE) {
      printf("\nRemaining (unpushed) data:\n");
      hash_print();
      printf("DynamicApp hash table:\n");
      hash_appdyn_print(hash_app_dyn);
   }
   return 0;
}
```

capture.c

Please note that the following code is only a sample of the client code. The full code can be found on the attached source code CD.

```c
/**
 * capture.c
 * Version: 0.1
 * Author:  Sam Crawford
 *
 * Packet capture and decoding functionality using pcap.
 *
 * Based in part on example found at:
 * http://floatingsun.net/asimshankar/code-samples/user/pcap/pcap.c
 * Extended to support TCP (so that it works!), ICMP, VLANs, IPSEC, PPP, GRE
 */

#include<pcap.h>
#include<stdlib.h>
#include<net/ethernet.h>
#include<netinet/in.h>
#include<netinet/tcp.h>
#include<netinet/udp.h>
#include<linux/if_arp.h>
#include<linux/ip.h>
#include<netdb.h>
#include<time.h>
#include<unistd.h>
#include<getopt.h>
#include<stdio.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netdb.h>

#include "gc.h"
#include "capture.h"

/**
 * Print the first 20 bytes of a binary blob as hex
 * Surprisingly useful for debugging
 */
void print_string_as_hex(char *str)
{
   if (str) {
      int i;
      const uint8_t *ch = str;
      for(i = 0; i < 16; i++) {
         fprintf(stderr,"%02x ", *ch);
         ch++;
      }
   } else {
      fprintf(stderr,"(null)");
   }
   fprintf(stderr,"\n");
}

/**
 * Debugging function for printing packet information
 */
void decode_debug(struct packet *p)
{
   if (IS_TCPIP_PACKET(p)) {
       fprintf(stderr,"tcp ");
       fprintf(stderr,"%s:%d",inet_ntoa(htonl(p->nh.iph->saddr)),
                  p->h.th->source);
       fprintf(stderr," > ");
       fprintf(stderr,"%s:%d",inet_ntoa(htonl(p->nh.iph->daddr)),
                  p->h.th->dest);
       fprintf(stderr," [tot_len=%d datalen=%d]\n",p->nh.iph->tot_len,
                  p->data_len);
   } else if (IS_UDPIP_PACKET(p)) {
       fprintf(stderr,"udp ");
       fprintf(stderr,"%s:%d",inet_ntoa(htonl(p->nh.iph->saddr)),
                  p->h.uh->source);
       fprintf(stderr," > ");
       fprintf(stderr,"%s:%d",inet_ntoa(htonl(p->nh.iph->daddr)),
                  p->h.uh->dest);
       fprintf(stderr," [tot_len=%d udp_len=%d datalen=%d]\n",
                  p->nh.iph->tot_len,
                         p->h.uh->len,p->data_len);
   } else if (IS_IPSEC_PACKET(p)) {
       fprintf(stderr,"ipsec ");
       fprintf(stderr,"%s",inet_ntoa(htonl(p->nh.iph->saddr)));
       fprintf(stderr," > ");
       fprintf(stderr,"%s",inet_ntoa(htonl(p->nh.iph->daddr)));
       fprintf(stderr," [tot_len=%d datalen=%d]\n",p->nh.iph->tot_len,
                  p->data_len);
   } else if (IS_ICMP_PACKET(p)) {
       fprintf(stderr,"icmp ");
       fprintf(stderr,"%s",inet_ntoa(htonl(p->nh.iph->saddr)));
       fprintf(stderr," > ");
       fprintf(stderr,"%s",inet_ntoa(htonl(p->nh.iph->daddr)));
       fprintf(stderr," [tot_len=%d datalen=%d]\n",p->nh.iph->tot_len,
                  p->data_len);
   } else if (IS_PPP_PACKET(p)) {
       fprintf(stderr,"ppp ");
       fprintf(stderr,"%s",inet_ntoa(htonl(p->nh.iph->saddr)));
       fprintf(stderr," > ");
       fprintf(stderr,"%s",inet_ntoa(htonl(p->nh.iph->daddr)));
       fprintf(stderr," [tot_len=%d datalen=%d]\n",
                  p->nh.iph->tot_len,p->data_len);
   } else {
       fprintf(stderr,"Unknown packet IP type = %d\n",
                  p->nh.iph->protocol);
   }
}

/**
 * Decode a UDP packet
 */
void decode_udp(struct packet *p, const u_char *raw)
{
   /* Pull out UDP header, and convert ints from byte to host order */
   struct udphdr *uh = p->h.uh;
   uh->source = ntohs(uh->source);
   uh->dest = ntohs(uh->dest);
   uh->len = ntohs(uh->len);

   /* Data is offset at 42 bytes in to the raw packet */
   p->data = (u_char *) &(raw[42]);
   p->data_len = uh->len-8;
}

/**
 * Decode a TCP packet
 */
void decode_tcp(struct packet *p, const u_char *raw)
{
   /* Pull out TCP header */
   struct tcphdr *th = p->h.th;
   th->source = ntohs(th->source);
   th->dest = ntohs(th->dest);
   th->seq = ntohl(th->seq);
   th->ack_seq = ntohl(th->ack_seq);
   th->window = ntohs(th->window);
   th->urg_ptr = ntohs(th->urg_ptr);

   /* Data is at TCP header offset, + 4xDataOffset */
   p->data = (void*)th + (4 * th->doff);
   p->data_len = p->nh.iph->tot_len - (4*th->doff) - (4*p->nh.iph->ihl);
}

/**
 * Decode GRE (Generic Routing Encapsulation)
 */
void decode_gre(struct packet *p, const u_char *raw)
{
   /* Magic to get the IP protocol */
   u_int16_t proto = (*((char*)p->h.raw+2) << 2) + (*((char*)p->h.raw+3));
   switch(proto)
   {
      case PROTO_GRE_IP:
          /* IP over GRE - Shift ptr on by GRE len to get next ip hdr */
          decode_ip(p, raw + 4 * (p->nh.iph->ihl) + 4);
```

```
          break;
      case PROTO_GRE_PPP:
          /* PPP over Ethernet */
          p->nh.iph->protocol = PROTO_PPP;
          break;
   }
   //print_string_as_hex((char*) raw + 4 * (p->nh.iph->ihl) + 6);
   //print_string_as_hex((char*) p->h.raw);
}

/**
 * Decode an IP packet
 * May call decode_udp or decode_tcp as well
 */
void decode_ip(struct packet *p, const u_char *raw)
{
   /* Pull out IP packet header */
   //struct iphdr *iph = p->nh.iph;
   p->nh.iph = (struct iphdr*) (raw + sizeof(struct ethhdr));
   p->nh.iph->tot_len = ntohs(p->nh.iph->tot_len);
   p->nh.iph->id = ntohs(p->nh.iph->id);
   p->nh.iph->frag_off = ntohs(p->nh.iph->frag_off);
   p->nh.iph->saddr = ntohl(p->nh.iph->saddr);
   p->nh.iph->daddr = ntohl(p->nh.iph->daddr);
   p->h.raw = (void*)p->nh.iph + 4*(p->nh.iph->ihl);

//fprintf(stderr,"Proto: %d\n",p->nh.iph->protocol);
   /* Switch on actual transport level protocol and decode */
   switch(p->nh.iph->protocol)
   {
      /* Recurse and get the REAL IP header */
      case IPPROTO_GRE:
         decode_gre(p, raw);
         break;
      /* Decode TCP */
      case IPPROTO_TCP:
         decode_tcp(p, raw);
         break;
      /* Decode UDP */
      case IPPROTO_UDP:
         decode_udp(p, raw);
         break;
      /* ICMP not decoded here. No point. */
      case IPPROTO_ICMP:
      default:
         p->data = p->h.raw;
         p->data_len = p->nh.iph->tot_len;
   }
}

/**
 * Just set the raw fields if we don't know how to decode
 */
```

```
void decode_unknown(struct packet *p, const u_char *raw)
{
   p->h.raw = p->data = p->nh.raw;
}

/**
 * Main packet decoding function.
 * Will call decode_ip (and subsequently TCP and UDP)
 * if the packet is ethernet based
 */
void decode_packet(struct packet *p, const u_char *raw)
{
   p->eh = (struct ethhdr*)raw;
   p->eh->h_proto = ntohs(p->eh->h_proto);
   p->nh.raw = (void*)raw + sizeof(struct ethhdr);

   /* Only decode ethernet header */
   switch(p->eh->h_proto)
   {
          /* Standard IP packet */
      case ETH_P_IP:
         decode_ip(p, raw);
         break;
          /* We're using VLANs. Ignore. Munch 4 bytes */
      case ETH_VLAN:
         decode_ip(p, raw+4);
         p->eh->h_proto = ETH_P_IP;
         break;
      default:
         decode_unknown(p, raw);
   }
   //decode_debug(p);
}
```

appdetect.c

Please note that the following code is only a sample of the client code. The full code can be found on the attached source code CD.

```c
/**
 * appdetect.c
 * Version: 0.2
 * Author:  Sam Crawford
 *
 * Application detection functionality for individual packets.
 * Basic idea is:
 * (1) Receive packet in to app_detect()
 * (2) Lookup in hash table, have we previously classified this host/port?
 *     If so, take that application ID initially
 * (3) Pass through all of the modules so they can look for dynamic session
 *     setups, and perhaps decide that they have a better match for the
 *      packet
 * (4) Return application ID
 */

#include<stdlib.h>
#include<net/ethernet.h>
#include<netinet/in.h>
#include<netinet/tcp.h>
#include<netinet/udp.h>
#include<linux/if_arp.h>
#include<linux/ip.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netdb.h>
#include<stdio.h>
#include<stdint.h>
#include<dlfcn.h>
#include<gc.h>
#include<string.h>

#include "gc.h"
#include "capture.h"
#include "flow.h"
#include "appdetect.h"
#include "netmon.h"

/* Array of function pointers */
u_int16_t (**modules)(struct packet *p, u_int16_t app,
                      DynamicApp **hashtable);
void* lib_handle;   /* Handle to our shared library of modules */
int modules_num;    /* Number of modules we have loaded */
char* module_names[100];   /* Name of the modules we have loaded */
const char* error_msg;

/**
 * Scans the file "/etc/netmon_modules" or "modules" in the current dir,
 * populating the global module_names array with their contents
 */
int load_module_names()
{
   FILE * fp;
   char name[100];
   modules_num = 0;

   fp = fopen("/etc/netmon_modules","r");
   if (fp == NULL) {
      fprintf(stderr, "Could not open \"/etc/netmon_modules\"...\n");
   }
   if (fp == NULL) {
      fp = fopen("modules","r");
      if (fp == NULL) fprintf(stderr, "Could not open \"./modules\"...\n");
   }
   if (fp == NULL) {
      fprintf(stderr, "No modules configuration file found!\n");
      exit(1);
   }

   /* Loop for all lines */
   while(fscanf(fp,"%s\n",name) != EOF) {
      /* Read over comments and blank lines */
      if (name[0] != '#' && strlen(name) > 0) {
         module_names[modules_num] = (char*) calloc(sizeof(char),100);
         strcpy(module_names[modules_num++],name);
      }
   }
   fclose(fp);
   return 0;
}

/**
 * Load the modules library, read in the modules file,
 * attempt to load the modules to memory for use later.
 * Terminates process if any modules not found!
 */
int init_modules()
{
   int i;

   /* Find module names */
   load_module_names();
   modules = (void*) calloc(sizeof(int*),modules_num);
   /* Open the library */
   lib_handle = dlopen("/usr/lib/libnetmon.so", RTLD_LAZY);
   if (!lib_handle) {
      fprintf(stderr, "Error during dlopen(): %s\n", dlerror());
      exit(1);
   }

   /* Loop for all modules */
   for(i=0; i<modules_num; i++) {
      char tmpname[100] = "detect_";
      strcat(tmpname,module_names[i]);
      if (BE_VERBOSE) fprintf(stdout,"Loading module %s... ",
                                     module_names[i]);

      /* Attempt to find the module in the library and load it */
      modules[i] = dlsym(lib_handle, tmpname);
      error_msg = dlerror();
      if (error_msg) {
         fprintf(stderr, "failed!\n");
         fprintf(stderr, "Error locating dynamic function '%s' - %s\n",
                            tmpname, error_msg);
         exit(1);
      } else if (BE_VERBOSE) {
         fprintf(stdout,"ok\n");
      }
   }

   /* If we got here, all was good and we loaded all the modules */
   return 1;
}

/**
 * Initialise DynamicApp hash table for storing host/port/protocol ->
 * application mappings */
void hash_appdyn_init()
{
   hash_app_dyn = (DynamicApp **) calloc(HASH_APP_SIZE,sizeof(DynamicApp*));
}

/* Hash function: host x port x protocol -> int */
int hash_appdyn_hash(uint32_t addr, uint16_t port, uint16_t protocol)
{
   return (addr + port + protocol) % HASH_APP_SIZE;
}

/* Creates a new DynamicApp structure */
DynamicApp * hash_appdyn_create(uint32_t addr, uint16_t port,
                 uint16_t protocol, uint16_t appid, uint16_t expire_secs)
{
   DynamicApp *app = (DynamicApp*) malloc(sizeof(DynamicApp));
   app->addr = addr;
   app->port = port;
   app->protocol = protocol;
```

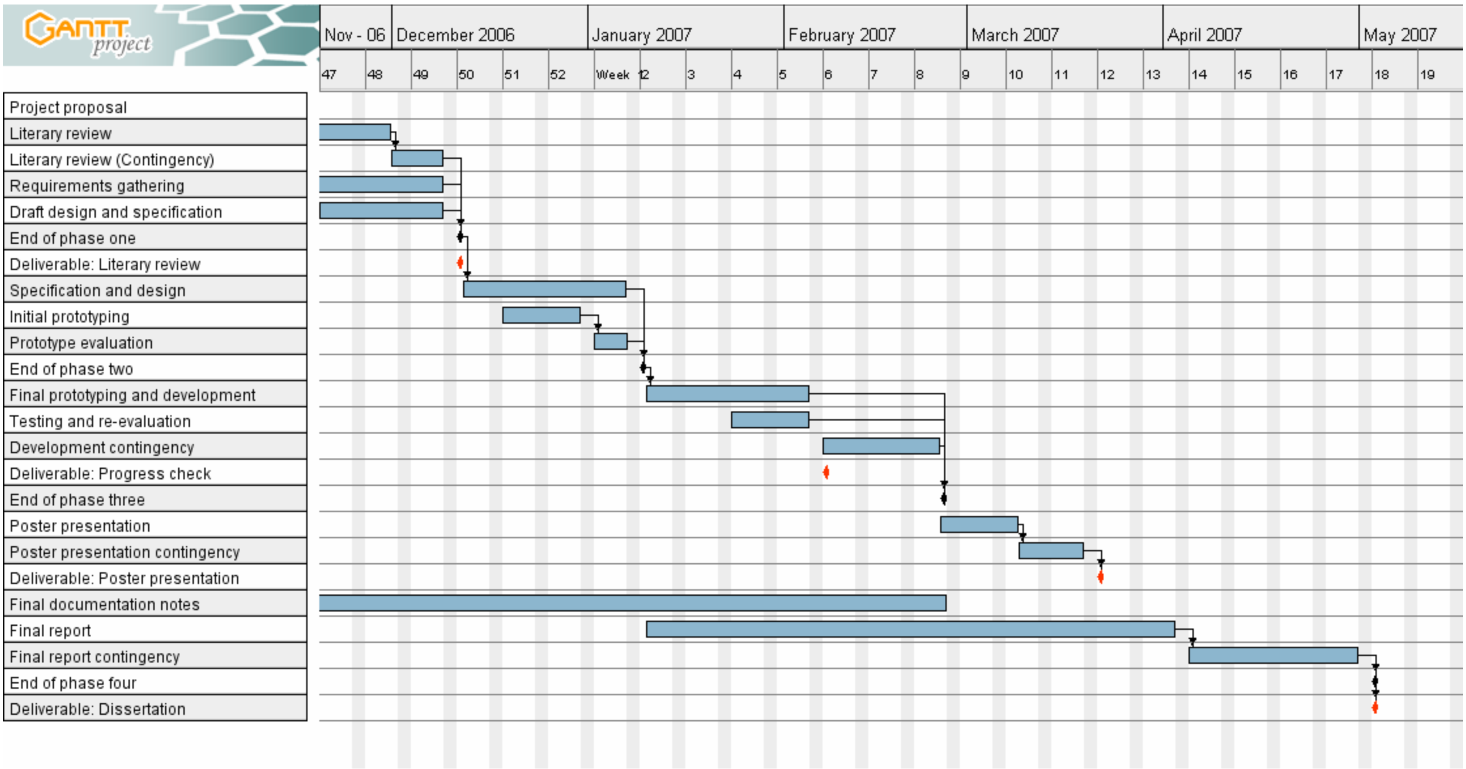
```
   app->application = appid;
   app->expires = time(NULL) + expire_secs;
   app->ttl = expire_secs;
   app->prev = NULL;
   app->next = NULL;
   return app;
}

/* Adds a DynamicApp in to the hashtable */
void hash_appdyn_add(DynamicApp *app, DynamicApp **hashtable)
{
   int hashval = hash_appdyn_hash(app->addr,app->port,app->protocol);
   if (hashtable[hashval] != NULL)
      hashtable[hashval]->prev = app;
   app->next = hashtable[hashval];
   hashtable[hashval] = app;
}

/**
 * Prints the dynamicApp hash table. Used only in debug
 */
void hash_appdyn_print(DynamicApp **hashtable)
{
   int i;
   for (i=0; i<HASH_APP_SIZE; i++) {
       int j = 0;
       DynamicApp *t = hashtable[i];
       while(t != NULL) {
          printf("[%5d][%2d] => ",i,j++);
          printf("%s:%d, [app: %d, proto: %d, expires: %d]\n",
                            inet_ntoa(htonl(t->addr)),t->port,t->application,
                            t->protocol,t->expires);
          //printf("%d\n",t);
          t = t->next;
       }
   }
}

/* Checks to see if the app is in the hash table */
DynamicApp * hash_appdyn_lookup(uint32_t addr, uint16_t port,
                                uint16_t protocol, DynamicApp **hashtable)
{
   int hashval = hash_appdyn_hash(addr,port,protocol);
   DynamicApp *t = hashtable[hashval];  /* Find flow at correct position */
   time_t currtime = time(NULL);

   /* Loop over the linked list, trying to find exact matching flow */
   while (t != NULL)
   {
      DynamicApp *toFree = NULL;

      /* If the current node matches, return it */
      if (t->addr == addr && t->port == port && t->protocol == protocol
                 && t->expires >= currtime)
         return t;

      /* Remove expired items */
      if (t->expires < currtime) {
         toFree = t;
         if (t->prev != NULL) {
            t->prev->next = t->next;
         } else {
            hashtable[hashval] = t->next;
            if (hashtable[hashval] != NULL)
               hashtable[hashval]->prev = NULL;
         }
         if (t->next != NULL) {
            t->next->prev = t->prev;
         }
      }
      t = t->next;
      if (toFree != NULL) free(toFree);
   }
   return NULL; /* Return null if flow does not exist */
}

/**
 * Clean the hash table of expired items. Used only in debug -
 * is a very intrusive function!
 */
void hash_appdyn_clean(int hashval, DynamicApp **hashtable)
{
   DynamicApp *t = hashtable[hashval];
   time_t currtime = time(NULL);
   while (t != NULL) {
      if (t->expires < currtime) {
         fprintf(stdout,"Expired DynamicApp found... removing...\n");
         if (t->prev != NULL) {
            t->prev->next = t->next;
         } else {
            hashtable[hashval] = t->next;
         }
         if (t->next != NULL) {
            t->next->prev = t->prev;
         }

      }
      t = t->next;
   }
}

/**
 * Main function for application detection process
 */
uint16_t app_detect(struct packet* p)
{
   uint16_t app = 0;
   uint32_t saddr = (uint32_t) p->nh.iph->saddr;
   uint32_t daddr = (uint32_t) p->nh.iph->daddr;
   uint16_t sport = 0;
   uint16_t dport = 0;
   int i=0;

   /* Find src and dst ports from packet type */
   if (IS_ICMP_PACKET(p)) {
      return 1;
   } else if (IS_IPSEC_PACKET(p)) {  /* Handle IPSEC */
      return 50;
   } else if (IS_PPP_PACKET(p)) {    /* Handle PPP */
      return PROTO_PPP;
   } else if (IS_TCPIP_PACKET(p)) {
      sport = p->h.th->source;
      dport = p->h.th->dest;
   } else if (IS_UDPIP_PACKET(p)) {
      sport = p->h.uh->source;
      dport = p->h.uh->dest;
   } else {
      return 0;
   }

   /* Attempt to lookup source and destination addresses in the hash
    * table to see if we can match against a prior application flow
    */
   DynamicApp *da = hash_appdyn_lookup(saddr,sport,p->nh.iph->protocol,
                                           hash_app_dyn);
   if (da != NULL) {
      da->expires = time(NULL) + da->ttl; /* Reset expiry */
      app = da->application;
   } else {
      da = hash_appdyn_lookup(daddr,dport,p->nh.iph->protocol,
                                     hash_app_dyn);
      if (da != NULL) {
         da->expires = time(NULL) + da->ttl; /* Reset expiry */
         app = da->application;
      }
   }

   /* Traverse module list using the array of function pointers
    * obtained earlier using init_modules
    * Function signatures:  module( packet, app )
    */
   for(i=0; i<modules_num; i++)
   {
      app = (*modules[i])(p, app, hash_app_dyn);
   }

   /* Return the application we ultimately settled on */
   return app;
}
```

libs/bittorrent.c

Please note that the following code is only a sample of the client code. The full code can be found on the attached source code CD.

```
/**
 * bittorrent.c
 * Version: 0.1
 * Author:  Sam Crawford
 *
 * Classifies BitTorrent traffic flows
 */

#include <stdio.h>
#include<net/ethernet.h>
#include<netinet/in.h>
#include<netinet/tcp.h>
#include<netinet/udp.h>
#include<linux/if_arp.h>
#include<linux/ip.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netdb.h>
#include "../capture.h"
#include "../appdetect.h"

char btheader[] = {
  0x13, 0x42, 0x69, 0x74, 0x54, 0x6f, 0x72, 0x72,
  0x65, 0x6e, 0x74, 0x20, 0x70, 0x72, 0x6f, 0x74,
  0x6f, 0x63, 0x6f, 0x6c };

u_int16_t detect_bittorrent(struct packet *p, u_int16_t app,
                            DynamicApp **hashtable)
{
   if (p->data_len >= 20 && p->data_len < 512 && memcmp((char*)p->data,
                                                            btheader,20) == 0)
   {
      uint32_t saddr = (uint32_t) p->nh.iph->saddr;
      uint32_t daddr = (uint32_t) p->nh.iph->daddr;
      uint16_t sport = 0;
      uint16_t dport = 0;
      if (IS_TCPIP_PACKET(p)) {
         sport = p->h.th->source;
         dport = p->h.th->dest;
      } else if (IS_UDPIP_PACKET(p)) {
         sport = p->h.uh->source;
         dport = p->h.uh->dest;
      }

      /* Record dynamicApp structures for TCP/UDP on both host/port pairs.
       * Enables future identification of flows
       */
      DynamicApp *dd, *ds;
      ds = hash_appdyn_lookup(saddr,sport,p->nh.iph->protocol,hashtable);
      dd = hash_appdyn_lookup(daddr,dport,p->nh.iph->protocol,hashtable);
      if (ds == NULL) {
         ds = hash_appdyn_create(saddr,sport,IPPROTO_TCP,6883,3600); // 1hr
         hash_appdyn_add(ds,hashtable);
         ds = hash_appdyn_create(saddr,sport,IPPROTO_UDP,6883,3600); // 1hr
         hash_appdyn_add(ds,hashtable);
      } else {
         ds->expires = time(NULL)+ds->ttl;
      }
      if (dd == NULL) {
         dd = hash_appdyn_create(daddr,dport,IPPROTO_TCP,6883,3600);
         hash_appdyn_add(dd,hashtable);
         dd = hash_appdyn_create(daddr,dport,IPPROTO_UDP,6883,3600);
         hash_appdyn_add(dd,hashtable);
      } else {
         dd->expires = time(NULL)+dd->ttl;
      }

      return 6883;
   }
   return app;
}
```

Appendix S – Original Gantt chart


GANTT project
Nov - 06
December 2006
January 2007
February 2007
March 2007
April 2007
May 2007
Week 1
Project proposal
Literary review
Literary review (Contingency)
Requirements gathering
Draft design and specification
End of phase one
Deliverable: Literary review
Specification and design
Initial prototyping
Prototype evaluation
End of phase two
Final prototyping and development
Testing and re-evaluation
Development contingency
Deliverable: Progress check
End of phase three
Poster presentation
Poster presentation contingency
Deliverable: Poster presentation
Final documentation notes
Final report
Final report contingency
End of phase four
Deliverable: Dissertation