CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Open Digital Archive at Oslo and Akershus University College

UNIVERSITY OF OSLO
Department of Informatics

Exporting IP flows using IPFIX

Master Thesis

Per Juvhaugen
Oslo University College

May 23, 2007

Abstract

Todays computer networks are continuously expanding both in size and capacity to accommodate the demands of the traffic they are designed to handle. Depending on the needs of the network operator, different aspects of this traffic needs to be measured and analyzed. Processing the full amount of data on the network would be a daunting task, and to avoid this only certain statistics describing the individual packets are collected. This data is then aggregated into ”flows”, based on criteria from the network operator. IPFIX is a recent IETF effort to standardize a protocol for exporting such flows to a central node for analyzation. But to effectively utilize a system implementing this protocol, one needs to know the impact of the protocol itself on the underlying network and consequently the traffic that flows through it.

This document will explore the performance, capabilities and limitations of the IPFIX protocol. A packet-capture system utilizing the IPFIX protocol for flow export, will be set up in a controlled environment, and traffic will be generated in a predictable manner. Measurements indicate IPFIX to be a fairly flexible protocol for exporting various traffic characteristics, but that it also has scalability issues when deployed in larger, high-capacity networks.

Acknowledgements

I would like to thank my supervisor, Kirsten Ribu, for guidance during this semester, and professor Mark Burgess, for help on deciding the topic for this thesis and for assistance on locating resources in connection to it. Furthermore I would like to thank Luca Deri, for providing a working implementation of the IPFIX protocol, and for his help in getting it up and running. A thanks also goes out to Jørgen Johnsen, for sharing his knowledge about administrating flow information export systems. Lastly I would like to thank all of this years master students at Oslo University College, for fruitful discussions and help throughout the master program.

Preface

The work documented in this thesis, marks the completion of a 2 year master program in *Network and System Administration* at Oslo University College in collaboration with the University of Oslo. The degree has spanned the years 2005 - 2007, and the thesis has been written in the final semester.

Project Background The idea for this project came after working on a project involving flow information export, at a network consulting company (IPnett AS). Not knowing anything about flows, some research had to be done about flow export. It soon became clear that very little information, besides RFCs and Internet-Drafts, was available about IPFIX/NetFlow v10. Being so new and untested, but at the same time being an effort to standardize such a widely popular family of protocols as NetFlow, sparked an interest in finding out more about this protocol. In discussions with professor Mark Burgess, it also became clear that it would be interesting to see how a push-based, centralized protocol would scale in todays ever growing networks.

Target Audience The content of this document, should be easily accessible to most people with a minimum knowledge about basic networking. It is, however, advisable to have a prior understanding of networking concepts and protocols such as IP, TCP and UDP, to fully comprehend the material herein. Other technologies used in the project, will be discussed in the opening chapters. The experiments performed will be discussed, and results presented in a simple and objective way. Configuration files for the applications used, and other periphery information, will be provided in the Appendix. An effort is made to keep the language and terminology in this document as clear as possible, so not to exclude any readers.

Thesis Outline The following is a rough description off each of the chapters in the thesis. For a more complete view of the document structure, please refer to the table of contents.

Chapter 1: Introduction This chapter contains an introduction to the subject of the thesis. It explains the importance of the technology explored in

this document. It identifies areas of interest within the subject, and it explains the motivation behind the project itself.

Chapter 2: Background This chapter gives a presentation of the technology chosen as subject for this thesis, and any other areas directly related to the work. It also discusses any previous research within this area.

Chapter 3: Methodology This chapter explains the methods and ideas behind the test procedures used in this project.

Chapter 4: Experimental Design This chapter documents the specific equipment, tools and applications used to perform the tests. It also identifies the limitations of the test bed.

Chapter 5: Results This chapter presents the results from the tests performed. It explains the individual tests, and comments on the findings. Tables and graphs are used to visualize the results.

Chapter 6: Conclusion, Discussion and Future Work This chapter contains discussion and conclusions deduced from the findings that has surfaced in the course of the project. It also contains sections describing related and future work within this field of research.

Appendix This chapter contains a collection of configuration files, scripts and other information related to the project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter serves as an introduction to the subject of the thesis. It gives an overview of the ideas and motivations behind this project.

1.1 Measuring Network Traffic

Traffic measurements are necessary to operate all types of IP networks, because the network must be provisioned after the type and volume of traffic it hosts. Network operators also need a detailed view of network traffic for security reasons. The composition of the traffic mix must be studied when finding dominant applications, users, or when estimating traffic matrices. All of these measurements could be done by logging the individual packages passing through central points in the network (typically routers and/or switches). But with the increasingly higher volumes of monitoring data, brought about by the ever-growing network capacities, this strategy is no longer feasible. Instead similar packets (packets with a set of common properties) are grouped together in composite *flows*. These flows keep statistical records of the types of traffic they are generated from. This way, similar types of traffic can be stored in a more compact format, without loosing too much information.

Making correct measurements on IP networks is not an easy task. Networks built on IP are not designed to reveal detailed statistics of the traffic between two endpoints. And the functionality for transmitting data between two such points, is divided in layers that only communicate through standard interfaces. Very few measurement capabilities are embedded into the different protocols operating on different layers. Because of all the challenges surrounding the subject of precise measurements in IP networks, a lot of work has been put down into the field. One of the latest technologies developed for use in this area, is the subject of this thesis: IPFIX.

1.1.1 Techniques

Measurement techniques are typically divided up into 2 categories, namely; active and passive.

Active Measurements

When doing active measurements, artificial traffic is being injected into the normal mix of network traffic, to test the network response. Statistics are then generated from the networks reaction to this known traffic. Common methods in this area, are the use of Round Trip Times (RTT) and one-way delay measurements. RTT can e.g. measure the total propagation delay to- and from an observation point. One-way delay measurements, gives an estimation of the time it takes to propagate a signal between 2 points in a network. For this to work, both *sides* in the measurement must have their clocks synchronized. Another type of test, that is definitely active, is the stress test. More intrusive than the other types of tests, caution must be taken when performing it. In networks, the stress test is often used to gain insight into the maximum throughput of a connection, by overloading the link one one side, and listening on the other. Delay, errors and other Quality of Service (QoS) aspects can be measured this way.

Passive Measurements

In contrast to active measurements, passive measurements utilize the existing traffic in the network, monitoring through fixed observation points. Since no test traffic is sent, passive measurements can only be applied in situations where the traffic of interest is already present in the network. Passive measurement is also referred to as non-intrusive measurement or as measurement of observed traffic. It cannot provide the kind of controllable experiments that can be achieved with active measurements. On the other hand it does not suffer from undesired side effects, caused by sending test traffic (e.g., additional load, potential differences in treatment of test traffic and real traffic). Some traffic is often being generated on the network, though, when exporting the collected data into a central database. The network monitoring performed in this project falls under the *passive* category.

1.1.2 Exporting Flows

One of the most common ways of getting network measurements, is by exporting flow information from the network nodes. This is done by aggregating network information into *flow datagrams*. A flow datagram describes, amongst

other things, the source, destination and size of a flow. when collected, this information provides a method for getting a detailed view of the network traffic. The use of flows allows for differentiation of different types of traffic through a selection of traffic properties.

1.2 Applications

Flow information export has become almost a de-facto standard of information retrieval for a number of applications and services. Common to all of them is a need to collect and analyze an ever-growing volume of traffic. This section will present some of the most common usages of flow information export[1].

1.2.1 Usage-based Accounting

Several new business models for selling IP services and IP-based services, have been put into production in recent times. Such services often need accounting based on time or volume, and accounting data can then serve as direct input for various billing systems. With enough detailed data, the accounting can be performed per user or per user group. It can differentiate between basic services and high level services, or even on a per content basis. Advanced filtering on classes of service, per application or per path used in the network, is also possible.

1.2.2 Traffic Profiling

Traffic profiling is the process of characterizing IP flows by using a model that represents key parameters of the flow (e.g. duration, volume, time, etc.). It is considered an indispensable component of network planning, network dimensioning, trend analysis and business model development. Since the objectives of traffic profiling can vary greatly, so can the requirements from the traffic measurements. This means it is in need of great flexibility in the infrastructure, configuration and classification from the measurement facility. Typical information needed for traffic profiling is the distribution of used services and protocols in the network, the amount of packets of a specific type (e.g., percentage of IPv6 packets) and specific flow profiles.

1.2.3 Traffic Engineering

Traffic engineering is a term for measurement, modelling, characterization and control of a network. Its ultimate goal is the optimization of network resource utilization and traffic performance. Traffic engineering comes as a direct reaction to measurements made in network, and requires direct access to the

network nodes, making it a 2-level operation. Level 1; passively obtain measurement results, Level 2; actively use these to tune network parameters. Typical parameters required from measurements are: link utilization, load between specific network nodes, number, size and entry/exit points of the active flows and routing information.

1.2.4 Attack/Intrusion Detection

Capturing flow information to analyze network data is important factor in network security. As a first perimeter defense, flow monitoring can allow for detection of unusual situations or suspicious flows, from e.g. a denial of service attack. As a second perimeter defense, flow analysis can be used to gather information about the offending flows, allowing for the planning of a defense strategy. Intrusion detection requires even more from the flows, as it not only uses specific characteristics of flows, but also stateful packet flow analysis. Locating activities characterized by specific communication patterns.

1.2.5 Quality of Service Monitoring

QoS monitoring is the passive measurement of quality parameters for IP flows. It often requires the correlation of data from multiple observation points (e.g., for measuring one-way metrics), which again demands clock synchronization of the involved metering processes. Since QoS monitoring can lead to a huge amount of measurement result data, it would highly benefit from mechanisms to reduce the measurement data, like aggregation of results and sampling.

1.3 IPFIX

IPFIX is an IETF working group[2]. The IPFIX working group has specified the Information Model (to describe IP flows) and the IPFIX protocol (to transfer IP flow data from IPFIX exporters to collectors). The goal of the IPFIX working group is now to produce *best current practice* and guideline documents concerning implementation, application and usage of the IPFIX protocol. But even if the specifications for the protocol is ready, the protocol still has to be thoroughly tested before network operators will feel confident to implement it in a production environment

1.4 Challenges

The solutions in a specific subject, is seldom what makes the subject interesting. But the challenges it poses, on the other hand, is what makes it intriguing.

And IPFIX has some major challenges to overcome before it will be adopted as an industry standard.

New Standard Being such a recent development, there exists very little material today concerning IPFIX. Both in terms of information and other documentation about the protocol itself (mostly RFCs and Internet-Drafts), as well as actual implementations of it. There are also very few records of networks currently running flow information export through IPFIX. This is possibly a *chicken vs. egg* situation, where one situation is dependant on the other, and vice versa.

Prior Knowledge A flow information export system is often implemented on a network to learn about the traffic already on it. But *blindly* implementing such a system, could have a serious impact on the normal traffic. Unfortunately these are the same systems used to gain that knowledge, working as the typical *Catch 22*. But by documenting tests of these systems under a controlled environment, while doing small-scale passive measurements on central network nodes, should provide operators with enough information to implement a flow information export system without to big of an impact on the normal traffic.

Sampling Generating flow records from *all* traffic seen in a high-capacity network, can be almost impossible. Resource limitations (both in CPU and in available bandwidth) has been overcome by *sampling techniques*. When sampling, one only records a small sample of the traffic seen, and hopes that it will be representative off all the traffic. It is not uncommon to have a sample-ratio of e.g. 1:1000. But this is not good enough for all of the applications of flow information export. For example, when using flows to do intrusion detection, every single packet must be inspected.

Transport Protocol IPFIX specifies that any implementation must support UDP, TCP and SCTP as transport protocols for flow information export. But it also specifies SCTP as the preferred protocol. This might be a hindrance to widespread adoption of the protocol, as SCTP also is fairly new and untested. The protocol is only supported on a few platforms, and while the SCTP seems solid on paper, operators are awaiting confirmation from real world tests.

Scalability Questions are being raised if the *old* NetFlow architecture, which IPFIX is based on, will have the scalability to keep up with the growth of the high-capacity networks of today. Seeing that IPFIX is both push-based and centralized, it is not hard to imagine intolerable volumes of exported traffic. In short terms, the main problem is two folded; Firstly since it is centralized, the bandwidth needed for export will increase together with the normal bandwidth usage on the network. Secondly since

it also is push-based, the central node has no control over the incoming flows, and will probably be the most narrow bottleneck.

Chapter 2

Background

This chapter contains background information on the technologies being used in the project and documented in this thesis.

2.1 Internet Protocol Flow Information eXport - Overview

Internet Protocol Flow Information eXport (IPFIX)[1, 2] is a protocol developed by the IETF IPFIX working group, which aims to standardize the format used for the export of network flow data towards data collection devices and network management systems. Network flow data annotates, here, the aggregation of data packets into composite flows, by characteristics of the data packets. This is a field where Ciscos proprietary *NetFlow* protocol is the dominant. IPFIX is based on the most recent incarnation of NetFlow, namely version 9. This means that the IPFIX protocol is built on a template-based system of information exchange, making it very flexible with regards to changing the default information fields of the exported flow records. Because of the template-based solution of flow records, it is fairly easy to tailor an IPFIX based flow information export system to a network operators specific needs.

2.1.1 Terminology

This section will briefly explain some terminology commonly used when discussing IPFIX. The expressions herein will be used throughout this document.

Flow The IPFIX working group has defined a *flow* as: *A set of IP packets passing an observation point in the network during a certain time interval. All packets belonging to a particular flow have a set of common properties. A packet is defined to belong to a flow if it completely satisfies all the defined properties of the flow.* This definition will cover potentially any traffic seen at the observation point. From a single packet with a specific sequence number, to every single packet on the network.

Observation Point The observation point is a location in the network where IP packets can be observed. Normally this means a central node in the network, typically a router or a switch, or an external *probe* connected to such a point.

Metering Process The metering process is the functionality at the observation point, that generates flow records from the packet headers seen at a Network Interface.The metering process consists of a set of functions that includes packet header capturing, time stamping, sampling, classifying, and maintaining flow records. The maintenance of flow records may include creating new records, updating existing ones, computing flow statistics, deriving further flow properties, detecting flow expiration, passing flow records to the exporting process, and deleting flow records.

Flow Record A flow record contains information about a specific flow that was metered at an observation point. A flow record contains measured properties of the flow (e.g. the total number of bytes of all packets of the flow) and usually characteristic properties of the flow (e.g. source IP address).

Exporting Process The exporting process sends flow records, generated by the metering process, to one or more collecting processes on a collecting node.

Collecting Process The collecting process receives flow records from one or more exporting processes. Normally the flow records are then stored in a database system, but this action is not covered by the IPFIX protocol.

2.1.2 Architecture

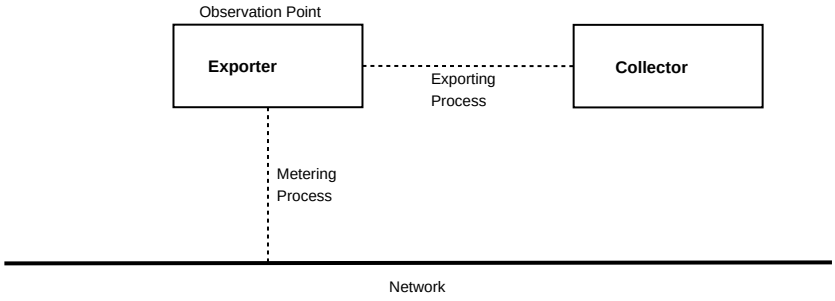

Figure 2.1: **IPFIX Architecture**: *The diagram shows a simplified version of the IPFIX architecture*

Figure 2.1 on the facing page shows the simple schematic of the components needed for flow information export. The component labeled *Exporter* in the diagram, can either be a central node in the network (typically a router or a switch), or an external probe connected to this device. The exporter will have a *metering process* running at all times, capturing the traffic on the network. This process is also responsible for sorting the data packets into flows, and format the flow records. In addition to this, the exporter also has a *exporting process* running. The exporting process does not run at all times, but is called when either the flow buffer is full, or after a certain time period. Whatever comes first (the time period is configurable). When the exporting process is called, it walks the flow cache, and starts sending the flows to a designated *Collector*. When sending flow data, IPFIX utilizes 1 of 3 transport protocols. It is specified that IPFIX must support UDP and TCP, as well as SCTP. The *Collector* has a *collecting process* running at all times, responsible for receiving flows, whenever they come in. Since the IPFIX protocol is push-based, the collector has no influence on when the flows will be coming in. What to do with the flow data after the collector receives it, is out of the scope for the IPFIX protocol, but the most common thing to do is to store the data in a database.

2.1.3 Flow Keys

As explained, the metering process sorts data packets into flows based on a number of properties of the packets, and a time frame. These special properties of the data packet are called *Flow Keys*. and the IPFIX protocol specifies 7 of these flow keys as the basis of which packets belongs together in a flow. These keys are:

- Source IP address
- Destination IP address
- Source port
- Destination port
- Layer 3 protocol type
- Type of Service (ToS) byte
- Logical input interface

Together with a configurable time frame, they are used by the metering process when generating flow records.

2.1.4 Templates

Templates enable the possibility of using specific information fields when exporting flows. The templates are defined by the network administrator, and can be configured to contain more- or less information fields than the default setting. This is a very flexible way of exposing a variety of traffic/flow characteristics. The use of flow templates is a feature that should make possible future enhancements to flows, without simultaneously requiring changes to the basic flow-record format. The templates are sent from the exporter to the collector at the beginning of each export session. They are also periodically resent, to ensure that the collector is aware of the format of the flow records it will receive.

2.1.5 Transport Protocol

When exporting flow data, the connection between the exporter and collector is set up over a transport protocol. IPFIX specifies that all implementations must support UDP, TCP and SCTP as transport protocols. It also specifies the preferred protocol to be SCTP. The most popular transport protocol for flow information export, seems to be UDP. This is probably because it yields by far the least overhead, and most operators do not need the added reliability that TCP and SCTP offers. An overview of the SCTP protocol can be found in section 2.2 on the next page

2.1.6 Message Format

The components and format[3] of an IPFIX message sent from an exporter to a collector is as follows:

Message Header This is the common header, appended to all IPFIX messages. The header contains the protocol version number, the message length, the time of export, a sequence number and the ID number of the observation domain. The header has a total length of 20 bytes.

Template The IPFIX template specifies the format of the flow records sent, and is always present in new export sessions. This is to ensure that the collector is aware of the flow record format.

Option Template Template for specifying optional information fields for the flow-records. Always present in new sessions, and periodically retransmitted (as with the normal template).

Data Set The data set contains the flow-record(s) with the information fields specified in the template or option template.

A correctly formatted IPFIX message must contain the message header, and at least 1 of the other 3 components (template, option template or data set).

2.2 Stream Control Transmission Protocol - Overview

The IETF IPFIX working group requires IPFIX implementations to support 3 different transport protocols for the export of flow information. These are: UDP, TCP and SCTP. Out of these, SCTP is the preferred protocol. But since the protocol has been developed very recently, only a few platforms have SCTP support. People are sceptic towards it, because it is fairly untested, and the protocol consequently suffers from a very low adoption rate. These reasons, along with the fact that it adds plenty of overhead to the transport protocol, are why people are still mostly using UDP for flow information export. But since it is named as the preferred protocol for SCTP, it still warrants an overview of it in this document.

The SCTP (Stream Control Transmission Protocol)[4, 5, 6, 7, 8] is a recently developed, reliable transport protocol for use on top of a potentially unreliable, connectionless packet service such as IP. It is designed as a general purpose, message oriented transport protocol, particularly needed when transporting signalling data. Being message oriented, means it preserves its message boundaries in the same way that UDP does. This means it is operating on whole messages instead of single bytes. So if one message of several related bytes of information is sent in one step, exactly that message is received in one step. Being reliable, means detecting lost, duplicate and out-of-order data in addition to containing flow- and congestion-control mechanisms, in the much the same way as TCP does. It bases these mechanisms on checksums, sequence numbers and selective retransmissions.

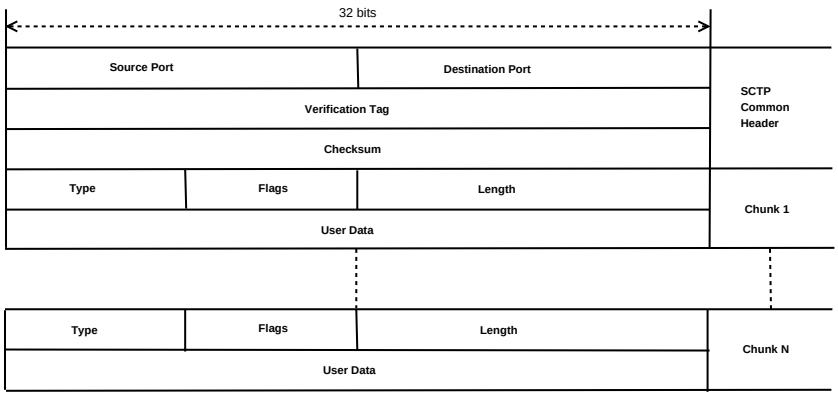

Figure 2.2: **SCTP Packet Format**: *The diagram shows the simplified architecture of a typical SCTP packet*

Figure 2.2 on the preceding page shows the format of a SCTP packet. The first 12 bytes of any given SCTP packet, will contain the common header, consisting of source/destination port, a 32-bit verification tag, and a 32-bit checksum (Adler-32 algorithm) as protection from transmission errors. After the header, comes *N* number of *chunks*, depending on the individual packets/messages. The chunk contains a *Type-field*, that describes the type of chunk being transmitted. This can be a data-chunk, or various control-chunks. It also has a *Flag-field* specific to the type of chunk, and a *Length-field* denoting the length of the chunk. The *User Data-field* contains the payload.

SCTP introduces a new way of setting up connections, where the initialization of a connection is completed after 4 steps. This is known as the SCTP *4-way Handshake*. A simple diagram of this handshake is presented in figure 2.3 on the next page.

In figure 2.3 on the facing page, *Host B* is the passive host of the connection setup (in other words: *Host A* initiates the connection). A passive side in such an association in SCTP, will not allocate any resources until it receives and validates the 3. of the messages. This mechanism will, to a certain degree, help dealing with the issues of *Denial of Service* attacks. But on the other hand, the added overhead of SCTP compared with other transport protocols makes it also a candidate for such attacks. And since the adoption rate of SCTP is fairly low, it is unknown if the 4-way handshake is an effective mechanism against DoS situations.

SCTP operates on 2 distinct levels when transporting datagrams. The first level is responsible for reliable transfers of datagrams. This is achieved by checksums, sequence numbers and selective retransmissions. After a packet has been validated, it continues to a second level, responsible for maintaining the ordering of the received datagrams. This order is maintained within individual streams, but not between different. To enable detection of loss of- and duplicate data packets, as well as reliable datagram delivery, Transport Sequence Numbers (TSN), and Stream Sequence Numbers (SSN) are introduced. The acknowledgements sent by the receiver, are based on these numbers.

SCTP can also do *multi-homed* streaming of data, which refers to SCTPs ability to transmit several independent streams of messages in parallel to nodes which can be reached with more than 1 IP address. If the network hosting these nodes is configured to send data to the node over different paths, the association can become tolerant of physical network failure. SCTP can perform retransmission of data over still available paths, if failures are detected.

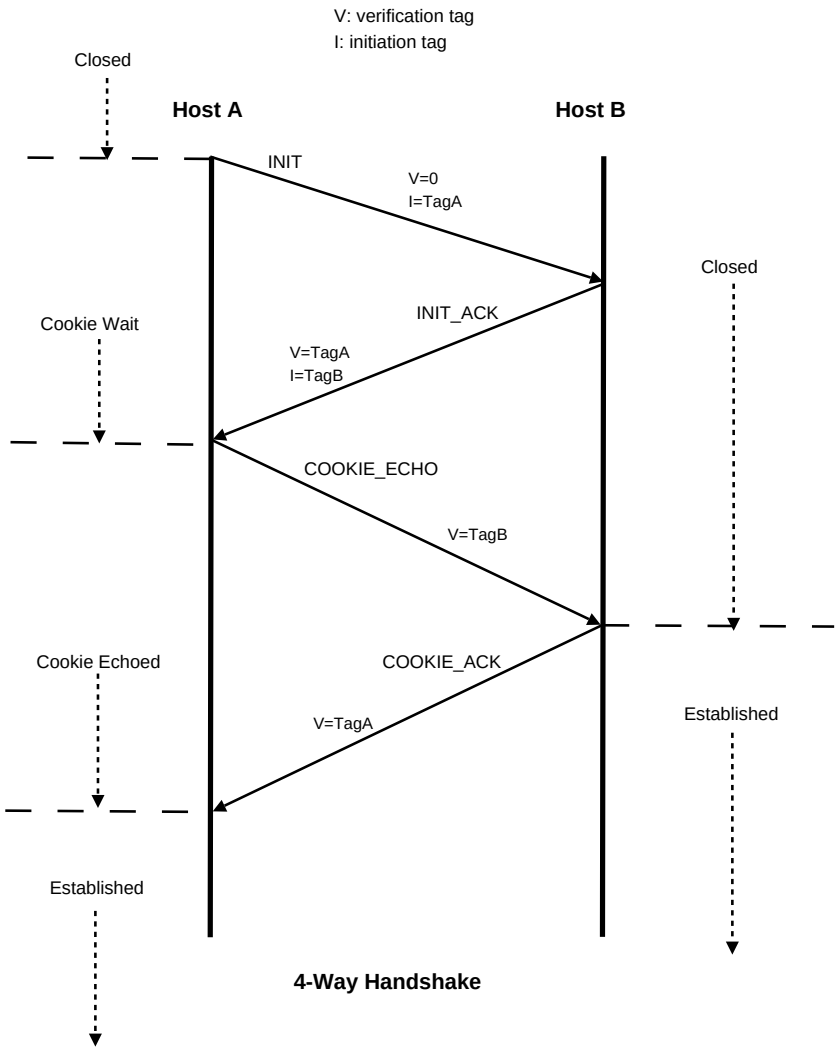

Figure 2.3: **The SCTP 4-Way Handshake**: *The diagram shows the architecture of the typical SCTP 4-Way Handshake*

Chapter 3

Methodology

This chapter will describe the more detailed goals of the project, introduced in chapter 1 on page 13. It will also elaborate on the methods used to reach these goals, and how the reader should evaluate the resulting measurements.

3.1 The Scientific Method

The scientific method is *a body of techniques for investigating phenomena and acquiring new knowledge, as well as for correcting and integrating previous knowledge. It is based on gathering observable, empirical, measurable evidence, subject to specific principles of reasoning*[9, 10]. It is a tool for researchers to propose specific predictions as explanations of natural phenomena, and design experimental models/designs to test the validity of said predictions. One of the most weighted goals of the scientific method, is that the process must be objective, to reduce a biased interpretation of the results. It is expected that all of the data gathered must be documented and shared, so it can be inspected by peers in the scientific field. This will allow for verification of the results by attempts of reproduction, in addition to the establishment of statistical measures of the reliability of the final result. All of the work put into this document and the project which it describes, have been done in accordance with the scientific method to the best of the authors abilities.

3.2 Objectives

The objective of this project is not based on the classical, well formed hypothesis, formulated after observing a phenomenon and trying to rationally explain its behavior. In fact, the motivation stems from quite the opposite, namely the lack of observations of a phenomenon. The IPFIX protocol is based on a very recent Internet-Draft by the IETF IPFIX working group. This means that there

are very few working implementations of the protocol, and equally few observations of its behavior. The lack of documented observations and the fact that the protocol itself is so new, creates some big challenges when evaluating it, as well as an inspiration to do so. Since there seems to be very little documentation of the behavior of a working implementation of IPFIX, this projects focus will be on observing its basic behavior, comment on the protocols design and evaluate its choice of architecture.

Special attention has been given to the task of separating the protocol from its implementation during the testing and evaluation phases. It is important that the evaluation of the protocol does not suffer from eventual faults in the implementation used for testing. The primary objective of the project is to evaluate the functionality of the protocol under normal operations. This means that that it is imperative to measure and follow the test-data, and give less attention to the applications actually running the services. As a secondary objective, however, measuring the resource usage of the implementation can be of interest. Excessive resource usage can be an indication of, either a bad implementation, or a design flaw in the protocol. It can be a difficult task to evaluate this, and it falls outside the scope of this project. In any case; if such anomalies are found, they will be reported. The evaluation of these findings, can then be done by the reader.

3.3 System Model

To put the protocol to test, a system consisting of 5 nodes was chosen (the reason for the number of nodes, was the availability of equipment in the local lab). These nodes would be able to host exporting/collecting processes, as well as generate traffic, monitor resource usage and take measurements, to evaluate the IPFIX protocol. A simple diagram showing the proposed system topology, is presented in figure 3.1 on the facing page.

By running the different modules needed in the test setup as local processes on each node, the total system resource overhead was reduced, compared to alternative model approaches (see section 3.3.1 on the next page). And since there was a lack of devices especially engineered to export IPFIX compatible flows in the local lab, this approach would be the one involving the least amount of work to set up. And less time setting up the test system, meant that more time could be used on actual testing. And time is always a factor when doing a project, such as this. The proposed system topology should also be an adequate analogy to similar flow information export systems in real world scenarios. Here, each node could emulate a subnet, able to generate traffic between local processes (emulating nodes in the same subnet) and between processes on different nodes (emulating nodes in different subnets). Each node would then have one exporting process, emulating an

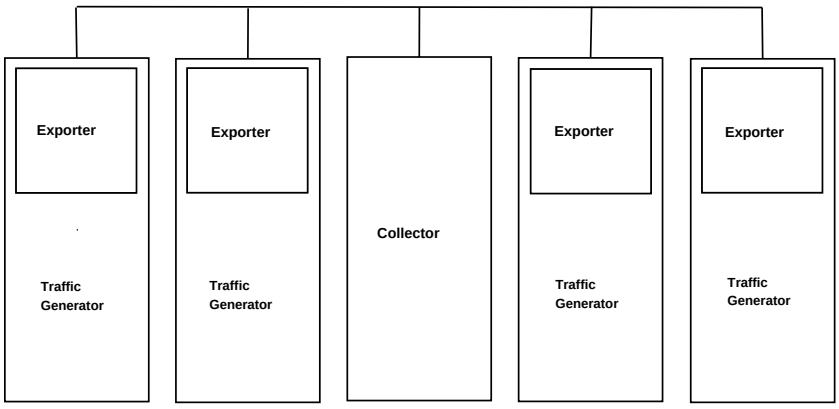

Figure 3.1: **Test System**: *The diagram shows the simplified topology/architecture of the test bed. 5 nodes are interconnected through a switch. 4 of the nodes are generating traffic between each other and exporting all flows seen on their NICs, and 1 of the nodes is acting as a collector.*

exporting node in a subnet. One node in the proposed topology, would be reserved for the collecting process. All traffic seen on the Network Interface Card on each traffic-generating node, would then be exported as a flow to a central collecting node. This enables the monitoring, and consequent evaluation, of the flow information export protocol in use. Both by capturing and analyzing the generated traffic on the network, and also by monitoring the resource usage on the nodes. Tools for this would be set up on each node, and configured to capture and log all activity. These *traffic dumps* and logs would then be used as data for the analyzation process of the test.

3.3.1 Alternative Models

Alternative approaches to evaluate the IPFIX protocol, was also considered. The most obvious one, was to use typical, specialized hardware, such as routers and switches that supported IPFIX, and get measurements from them. This would have a couple of clear advantages. Firstly, the protocol itself would already be implemented, and should not pose as a challenge to configure and get working. Secondly, the nature of the specialized hardware would mean that resource usage on the nodes would not be a cause of concern, as it should not have any impact on the performance of the protocol. The biggest problem with this approach, would be to get access to such equipment. Since the school did not possess IPFIX enabled devices, this approach would rely on the cooperation of external partners. Using external partners would complicate the project work, with respect to the availability of both equipment and external advisors. It would also mean that the tests would not necessarily be carried

out in a controlled environment, but perhaps in a production network. If that was the case, there would be no way to manipulate the base-traffic in the same way as in a laboratory. This approach was fairly quickly discarded.

Yet another approach to a system model that was considered for some time, was the use of virtualization. This could be done in the local laboratory, with the present equipment. The idea was to build virtual network(s), designed to emulate real world network scenarios. This model also had its advantages, and the most noticeable; configurability. With a virtualized network, one has almost complete control over the node- and network parameters. It also comes off as a fairly elegant solution, since all of the nodes and components would have the same mode of operation as in a real world scenario. This in contrast to the chosen model, where instances of exporters, traffic generators and measurement tools only lives as separate processes inside one machine. Without getting to philosophic, an argument could be made that this is the same thing that is happening when virtualizing, only with more overhead. But at least all of the configurations would be done in a way, similar to a real scenario. A couple of points counted against going the virtualization route. Firstly, using virtualization would mean a lot more work when setting up the test bed. And even if this could be done in cooperation with others, using the same system for tests involving virtualization, the increased setup time would be damaging to the already constricted time slot for this project. In addition, the virtualization would add resource overhead to the tests, possibly having an impact on results. One could argue that this overhead would not matter, since the performance of the IPFIX protocol would always stay relative to the data it is exposed to. But in the end, this approach was also abandoned in favor of the simple test design chosen (see section 3.3 on page 28).

3.3.2 Protocol Implementation

The fact that the draft of the IPFIX protocol is fairly recent, meant that very few implementations of the protocol for generic hardware existed. And since the protocol originally was designed for specialized hardware, such implementations are often left up to the individual vendors. Searching for IPFIX enabled NetFlow probes for generic x86 hardware, was a task simplified by the fact that there exists only a handful of these. Choosing an implementation for testing was even easier, as there seemed to be only one IPFIX enabled application ready for deployment, as other implementations only consisted of IPFIX compatible libraries. Due to time restrictions, writing an application that could take advantages of these libraries, was not an option. A complete implementation of the IPFIX protocol, would consist of both an exporter and a collector, as discussed in section 2.1 on page 19. One of the nodes in the test setup would act as the collector, while all the other blades would have an export process

running. The chosen applications for exporting/collecting IPFIX flows, are described in sections 4.1.2 on page 35 and 4.1.2 on page 36 respectively.

3.3.3 Generating Test Data

In a real life scenario, the flow information export protocol would be exposed to a varied mix of network traffic. It is not the goal in the tests described in this document, to recreate this mix of traffic in the strictly controlled test environment. Instead a traffic generator capable of outputting specific types of data, is to be used. By controlling the generated data down to packet level, a clear understanding of the impact of the flow information export protocol can be gained. So while the final results might not seem to be directly comparable to a typical real life situation, the results will be of a kind such that the reader should be able to relate them to a specific scenario. This, of course, demands that the reader already has some knowledge of what kind of traffic is present on the network in question. The traffic generator chosen for the tests described in this document, is presented in section 4.1.2 on page 36. It is based on a client/server architecture, and is configurable on a per-flow basis.

3.3.4 Selected Measurements

When measuring the effects of IPFIX on a network, the interesting data will be the traffic volume emitted when exporting flow records. To a certain degree, it is also interesting to measure the strain on the system providing the flow information export service. But this will not necessarily have any direct impact on the traffic already present on the network, and it will for the most parts be a testament to the specific implementation being tested. In the tests presented in this document, resource usage will be monitored, but not commented on unless they exceed normal levels. This is just to ensure that the IPFIX protocol does not suffer from any possible resource depletion.

The IPFIX protocol specifies 3 possible protocols for transport of flow-data (namely UDP, TCP and SCTP). These are not the main focus of this project, and evaluating them on a specific basis falls outside of the scope of this document. UDP and TCP are considered as *well known* protocols, and the SCTP evaluations referenced to in this document are made by Stewart, et. al.[7] and Rajamani, et. al[8]. Instead the measurements taken in the test will be on the bandwidth usage of the flow information export under various types of baseline traffic, and with various configuration of flow records. Comparisons with similar protocols to IPFIX will also be done. One thing to keep in mind when measuring bandwidth usage, is to present the final results in a way that correctly reflects the strain on the network. Elaboration on this topic can be found in section 5.1.3 on page 45.

All of the tested applications produce logs of the activity, and these logs could potentially be used as measurements in the tests. But to ensure even more objectivity, a separate application will be responsible for all measurements from the separate activities on the system. This application is presented in section 4.1.2 on page 36.

3.3.5 Sources of Errors

There are some possible sources of errors in this test model, but by identifying them early in the process, they can be attended to accordingly.

One of the things to look out for, is excessive resource usage on the nodes running processes pertaining to IPFIX. This could be indicative of problems with running processes, or just normal symptoms of processes with intense resource usage. In either case it could mean that essential processes for the final result could be affected, and the results thereby *tainted*. To ensure that this will not skew any of the results, a lightweight monitoring application will be running on each of the nodes, logging essential system information. This way, any possible errors due to resource depletion, can be identified, and the specific tests redone with new configurations.

Another possible problem, could be errors within the applications running the flow export service itself. Since no comparable results exist, it could be hard to identify any errors from the measurements taken during tests. The applications will be running in *very verbose* mode, and all output will be logged and inspected. But there is no guarantee that any errors in the application itself, would show up on any logs. This is a calculated risk when running recent or otherwise immature applications. It all boils down to how much one trusts the implementation.

The measurements themselves could also contain errors, if the measuring application fails. The application chosen for the task, is a very mature program that has been tried and tested over an extensive period of time. This is, of course, a heavily weighted factor when analyzing the risk of errors in measurements.

And lastly, there could be errors, or at least misrepresentations, when presenting the results from the tests. This especially holds true when presenting bandwidth usage. A more in-depth discussion of this is present in section 5.1.3 on page 45. In particular there is a danger of *averaging* the data over a too large time interval when presenting the discrete measurements on a seemingly continuous timeline.

Chapter 4

Experimental Design

This chapter contains discussion about the simple experimental setup, that allows for inspection and analysis of the IPFIX protocol. It will present the equipment and tools used to carry out the measurements in the experiments, in detail. It will also address some of the shortcomings with the experimental design, and discuss the impact of these on the final results.

4.1 System Configuration

All of the experiments were carried out on the IBM BladeCenter HS20 available in OUCs network lab. The BladeCenter was chosen because of its relatively high capacity in processing power, RAM and internal network bandwidth. An overview of the test setup can be seen in figure 4.1 on the following page. Due to resource limitations, this setup was also used by other master students at OUC for work on their master thesis, and some compromises in the form of time-sharing and choice of operating system had to be made. This was known before the experiments started, and should not be viewed as a hindrance to the work documented in this paper. The BladeCenter in the lab had 6 blades installed.

4.1.1 Hardware Equipment

All of the blades in the IBM HS20 BladeCenter chassis are identical in hardware. However, since one of the blades was malfunctioning at the beginning of the experiments, only 5 blades were actively used.

IBM HS20 BladeCenter Chassis

Table 4.1 on the next page gives an overview of the IBM HS20 BladeCenter Chassis.

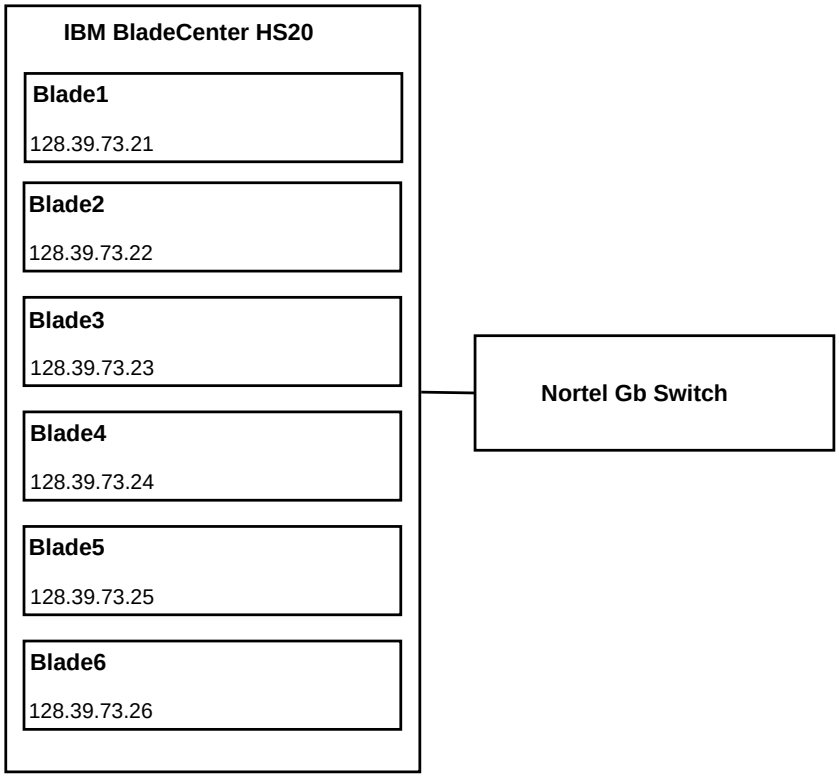

Figure 4.1: **IBM BladeCenter HS20**: *The diagram shows the internal network configuration of the BladeCenter, where the individual blades are connected to a Nortel Gigabit-switch*

HS20 BladeCenter Chassis	
Blade Bays	14 dual-processor blades
Media	DVD-rom and USB port available to all blades
Networking	Nortel Networks Layer 2/3 Copper Gigabit Ethernet
Management Software	IBM Director

Table 4.1: **Description of the IBM HS20 BladeCenter Chassis**: *The BladeCenter chassis supports redundant power supplies and has a separate network interface for management.*

IBM HS20 Blades 1-6

HS20 Blade 1-6	
CPU	Intel Xeon Processors (dual) - 2.8GHz
RAM	1GB PC2-3200 DDR2
Network	Dual Gb NICs
Hard Drive	32GB Ultra320 SCSI
Operating System	Ubuntu 6.04, Kernel 2.6.15-xen (XEN enabled)

Table 4.2: **Description of the IBM HS20 Blades**: *All of the blades are connected internally through a Gigabit switch. They share the same hard drive, USB port, CD-rom, mouse keyboard and screen, which is administered via a KVM switch.*

Table 4.2 gives an overview of the hardware in IBM HS20 Blades 1-6.

4.1.2 Tools

All of the software used in the different tests, is open source and available to everybody. Unless specified otherwise, they are the standard version available from the Internet.

Operating System - Linux

Linux[11] was installed onto all of the Blades. To accommodate for the needs of multiple students using the BladeCenter as a test bed in their master thesis work, a Xen-enabled kernel was chosen. As a result of this, Ubuntu 6.04 (LTS) with kernel 2.6.16-xen is used in all of the tests documented in this paper.

IP flow exporter - nProbe

nProbe[12] is an open source NetFlow/IPFIX probe, able to capture packets flowing on an Ethernet segment, compute the corresponding *flow* and export them to a designated *collector*, using NetFlow/IPFIX. Flow parameters are configurable, and nProbe is able to export flows to both commercial applications and other open source tools such as nTop. nProbes packet capture mechanism is built on the *libpcap* packet capture library. There was initially some problems with getting nProbe to function properly, but with the help of the author (Luca Deri), and a version of nProbe from the development branch, the application was able to capture packets and export them using the the latest IETF draft for IPFIX.

The following is console output from the program, showing the version number:

```
bash 1 \$ ./nprobe -v
Welcome to nprobe v.4.9.1 for i686-pc-linux-gnu
Built on 04/11/07 11:34:14 AM
Copyright 2002-06 by Luca Deri <deri@ntop.org>
```

IP flow collector - nTop

nTop[13] is an open source NetFlow/IPFIX probe and collector, from the same author as nProbe. nTop is able to both capture and export traffic flows as well as collect and analyze them. When used purely as a collector, nTop can be controlled from a relatively simple web interface. Since nProbe uses less resources than nTop when capturing and exporting flows, nTop is used purely as a collector in the experimental setup.

The following is console output from the program, showing the version number:

```
bash 2 Welcome to ntop v.3.2 SourceForge .tgz
[Configured on Nov 30 2005  4:16:33, built on Nov 30 2005 04:17:15]
Copyright 1998-2005 by Luca Deri <deri@ntop.org>.
```

Traffic Generator - Harpoon

Harpoon[14] is a flow level traffic generator. It utilizes a two level design, with a client- and a server-part, to generate flows with specific statistical qualities. It is possible to use harpoon to extract distributional parameters from measured network traces, to replicate the traffic. But it can also be manually configured through input files, to generate traffic from some relatively simple parameters. This is the desired option for all of the tests in this experimental setup. An example of such configuration-files, can be found in appendix A.1 on page 71.

Packet Capture - Tcpdump

Tcpdump[14] is a common tool for capturing traffic from a network interface. It is built upon the thoroughly tested *libpcap* packet capture library, and can both display the captured packets directly on the console, or dump the them into a binary *pcap-file*. In addition it can utilize *BPF-filters* for minimizing the processing load, or rendering more useful output, on networks with a high volume of traffic.

Data Mining - Tcpstat

Tcpstat is a tool for collecting statistics about network interfaces. It can collect information about network usage by either monitoring a specific network interface, or reading from a previously captured dump-file. For the experiments

documented in this paper, it is used for gathering bandwidth information from previously captured *tcpdump-files*.

System Monitor - Atsar

Atsar is a program designed to collect different statistics about the system it is running on. It accomplishes this by reading the files under the *proc* directory in predetermined intervals, and logging the output. Atsar is used as a system monitor on both the exporting and collecting nodes in the experimental setup. Its primary function herein, is to check if the results could be influenced by strain on the components in the system (and not by the design of the protocol).

Plotting - Gnuplot

Gnuplot[15] is a command-driven interactive function plotting program. It is used to plot the graphs of the measured values in the experiments.

Time Synchronization - NTP

The Network time Protocol[16] (NTP) is a protocol for synchronizing the clocks of networked computers. Since the experimental setup utilizes multiple hosts/nodes, it is imperative that they keep the same time, so it is possible to correlate the results on the individual nodes with each other.

Post Processing - Script

A small python script was used for light post processing of the collected network data. Since the *throughput* measurement is a figure relative to the timespan of the collected data, as discussed in section 5.1.3 on page 45, this script provided a flexible way of normalizing the final data. The script can be found in appendix A on page 71.

4.2 Limitations

The IPFIX protocol is designed to be implemented It is heavily based on Ciscos NetFlow protocol version 9. Cisco originally designed the NetFlow protocol for use on networks utilizing the companys own routers. While this doesn't hinder the IPFIX protocol to be implemented on other type of nodes, such as generic x86 computers, it does explain why there is just a few available implementations for such nodes. The fact that IPFIX has its roots from a protocol designed for routers, combined with the fact that it is a fairly new protocol, means that there only exits a very low number of implementations for

standard x86 boxes. There are some applications, both open and proprietary, that are capable of receiving IPFIX formatted flow-data. And there seems to be even fewer applications capable of exporting such flows. When choosing which implementations to use for testing IPFIX, the deciding factor quickly became the flow exporter. There seems to be only 1 readily available, open application that supports the export of IPFIX formatted flows. And that is *nProbe*[12]. In addition there exists libraries, such as libipfix, that has support for IPFIX, but these would require an external application to utilize its functionality.

But even if there is only a limited number of solutions that implement the protocol in question, and only 1 implementation is actually tested, this should not affect the outcome of the tests. It is imperative to keep in mind when looking at the results, that it is how the protocol dictates behavior that produces the results, and not the implementation of it. To make sure that the implementation is not a limiting factor, a monitoring application for system resources was used during all tests.

4.2.1 CPU

The CPU present on the blades, could theoretically become an bottleneck in the experimental design. Both the metering process on the exporting nodes, and the collecting process on the collecting node, requires a certain amount of processing power. The blades are, however, equipped with dual Intel Xeon processors running at 2.8GHz. This is a fairly powerful processor, even by todays standards, and it should not be any weaker than its counterparts found in state of the art routers.

The following is console output from reading *cpuinfo* for one of the CPUs:

```
bash 3 $ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 15
model           : 4
model name      : Intel(R) Xeon(TM) CPU 2.80GHz
stepping        : 3
cpu MHz         : 2800.220
cache size      : 2048 KB
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 5
wp              : yes
flags           : fpu tsc msr pae mce cx8 apic mtrr mca cmov pat
                  pse36 clflush dts acpi mmx fxsr sse sse2 ss ht
                  tm pbe lm constant_tsc pni monitor ds_cpl cid
                  cx16 xtpr
bogomips        : 5602.96
```

4.2.2 RAM

The amount of RAM is also a potential bottleneck in the experimental test setup. This applies especially to the metering process on the exporting nodes, that needs to hold a vast amount of information in buffers before writing to disk, when capturing packets off the network. The blades are each equipped with 1GB of RAM, which on a x86 system, is comparable to many types of routing equipment. The *atsar* application used in the experiments, is able to log both memory and cpu usage during the tests. This will alow for analysis of resource usage, and discussion of the possibility that resource depletion is tainting the protocol test results.

4.2.3 Hard Drive

The hard drive read/write speed could affect the packet-capture operation, when capturing high volume traffic. The blades have a 32GB Ultra320 SCSI hard drive, which should yield decent performance.

The following is concole output from the *hdparm* application, timing the drive speed:

```
bash 4 $ sudo hdparm -tT /dev/sda1

/dev/sda1:
 Timing cached reads:   3348 MB in  2.00 seconds =
 1674.81 MB/sec
 Timing buffered disk reads:  174 MB in  3.01 seconds =
 57.83 MB/sec
```

4.2.4 libpcap

Libpcap[17] is the packet capture library used in both *nProbe* and *tcpdump*. It provides a high level interface to packet capture systems, and is an essential component in the experimental setup. It is both well documented and thoroughly tested, and should be a suitable solution when analyzing the IPFIX protocol. Still, there is one concern that has become evident during the test period; Since both the application being tested, *nProbe*, and the application used for measurement, *tcpdump*, rely on the same component for their core-functionality, errors in the test itself might not be discovered in the measurements. But even if this is true, libpcap is the most widely used component for this type of application. It would seem that choosing any other type of packet-capture mechanism, would mean choosing a more uncertain, and not so well tested solution.

During the tests, the limits of the packet-capture mechanism quickly surfaced. Depending on the size of the packets and the traffic volume, i.e. the number of packets per second, the packet-capture mechanism started dropping packets. This behavior occurred both in nProbe, and in tcpdump. The common denominator here being the libpcap library. This indicates that it is not a individual problem with either of the applications, but rather a common problem with the packet capture library. *Atsar* was used to log both CPU and RAM usage, to discover if the culprit could be heavy hardware resource usage. But when looking at the Atsar-log from the sessions where libpcap started dropping packets, there was no evidence of high CPU or RAM usage.

Since tcpdump was set to write to a binary dump-file, using the *-w* option, the disk write speed could be a bottleneck when capturing a high number of packets. But sending the dumped packets to */dev/null*, and thereby avoiding writing the file to disk, did not reduce packet loss at high speeds. In addition tcpdump was set to not resolve IP addresses and BPF-files (Berkley Packet Filter) were introduced, in an effort to reach larger traffic volumes before packet loss occurred. To no avail.

After spending a fairly long time trying to fix libpcap, a decision to run all tests with speeds that libpcap could keep up with without dropping packets, was made. For the protocol analysis, there is no need to push high volumes of traffic during testing. Since this is not a test of the implementation, but

of the protocol underneath, testing with low traffic volumes should give the same results as with high volumes. The results presented will be relative to the amount of traffic produced. Also, it would be wrong to directly compare a generic software implementation of IPFIX for x86, to IPFIX implementations in dedicated hardware (i.e. routers and switches). On commodity x86 hardware, libpcap is used as the most stable and effective packet capture library. On the other hand, on most routers and switches all packet capture is performed in hardware, and does not suffer from the same limitations as its software counterpart. But evaluating the individual implementations of the IPFIX protocol, is not within the scope of this thesis.

4.2.5 SCTP

The Stream Control Transport Protocol[4, 5] is the preferred transport protocol for IPFIX. But in despite of this, it seems to be very seldom utilized in flow information export systems. This is largely due to the fact that it is a recently developed protocol, and lacks support on many platforms. So on the one hand, administrators might be reluctant to implement such a new and untested protocol, and on the other hand they might not be able to because of lack of support in operating systems. Recent versions of the Linux kernel, should however support SCTP. And together with SCTP libraries (sctplib), compiling programs with SCTP support should be possible.

Unfortunately it was not possible to compile the nProbe application with SCTP support on the machines used for the tests described in this document. This might have something to do with the fact that the machines were not running a *vanilla* kernel, but rather a Xen-enabled one (this situation could not be changed, as the test system was utilized by others throughout the test period). Or it could be incompatibility with the version of nProbe, acquired from Luca Deris development branch. In either case, it meant that testing SCTP as the flow information transport protocol, would not be possible. One could chose to see this as a testament of the continued immaturity of SCTP, and as one of the reasons the protocol is not more widely adopted.

Chapter 5

Results

This chapter will clarify the individual tests, and present and comment on the results. It will also address any encountered anomalies that deviate from the initial methodology.

5.1 Test Procedures

It is imperative to emphasize that all of the tests are designed to highlight aspects of the underlying protocol used, namely IPFIX, and not the implementation of said protocol, namely nProbe and nTop. This means that measurements of e.g. resource usage will not play an important role in the test setup. As long as depletion of resources does not interfere with the execution of the exporting protocol itself, it will not be a factor when analyzing the results. High resource usage on the specific implementation will, however, indicate points in the system where one is likely to run into limitations, even when using other implementations. So even if the exact measurements of said resource measurements, only is relevant for the specific implementation, it should be considered indicative for how a general implementation of the IPFIX protocol will behave. After all, even if IPFIX is designed for specialized hardware, there is nothing in the protocol that limits it to such hardware. And measurements on generic x86 hardware, will apply to the general protocol just the same as tests performed on other systems.

So when testing the IPFIX protocol, the important issues will be how the protocol behaves on the network, independent of the specific implementation. This can roughly be split up into two parts, namely the impact of the IPFIX protocol on the underlying traffic, and the scalability of the protocol itself. This means looking into what happens with the flows after the capturing process, and the traffic characteristics of exporting the flows from an exporting node to a collecting. The capturing process on the exporting node, and the storage facility on the collecting node, can be implemented in different ways on differ-

ent systems, and should not be considered a major factor when analyzing the export protocol.

As every network has its own characteristics, based on the hardware it is built on and the services it provides, the testing should be performed on a simple system in a controlled environment. By having complete control over the different parameters in the tests, the analysis can be as neutral as possible. And the results can be subjectively interpreted, for use in individual scenarios.

5.1.1 Test Details

Common for all of the tests, is the use of Harpoon as a traffic generator. This application provides the possibility to vary e.g. the size of the generated packets, the time between concurrent connections, the type of transport protocol and the total amount of traffic. In most of the tests TCP-traffic with inter-connection-times of 0.1 sec. is used. The size of the data-packets and the total amount of traffic was subject to variations. The specific details on this is provided in the subsection for each test.

Also common to all tests, is the use of nProbe/nTop as exporter/collector. For the most parts, the applications are used with their default setting for IPFIX use (where applicable). This means that UDP is used as the transport protocol for the flows from the exporter to the collector. UDP also seems to be the by far most common transport protocol for flows, used in these types of flow-export systems.

When focusing on the how the protocol handles the flow-data after the metering process on the exporting node, and before handing them off to the storage facility on the collecting node, the remaining phase can essentially be described as transport of flow-data. And since the underlying protocol for this transport is controlled by the exporter/collector, in the following tests, the measurements taken should be purely quantitative, and describe the protocol overhead compared to the normal traffic. In other words: measuring the added bandwidth usage when exporting flow-data. This raw data can then be used as a basis for calculating the impact of IPFIX on more specific situations.

5.1.2 Resource Usage

When analyzing a protocol primarily designed for specialized hardware, only using commodity hardware and software implementations, it is important to keep an eye on the resource usage in the test bed. Experience suggests that a software implementation will almost always have a lower threshold for errors under heavy load, than a similar mechanism implemented in hardware. During tests, the Atsar application was used to monitor the resource usage on the computers used in the tests. The most important factors to monitor

during testing of the IPFIX implementation, are CPU-usage and RAM-usage. These resources will be exploited especially during the packet-capture phase, when walking the hash of flows before exporting, and when receiving flows for buffering and committing to disk.

For most of the tests documented in this paper, the resource usage did not exceed the limits for normal usage, and should consequently not have any impact on the resulting measurements. For tests where the resources on the nodes were depleted more than normal, this will be commented on in the notes for the specific test.

5.1.3 Measuring Bandwidth

The chosen time-interval from which the bandwidth is computed, is an important factor when measuring and calculating bandwidth numbers.

When bandwidth is measured, the resulting numbers are computed from discrete samples of data, and these numbers that describes the bandwidth will vary according to the interval in which they are sampled. In a network with small and rapid bursts of data packets, the bandwidth of the passing data will not be correctly reflected, if the time-interval from where the bandwidth is computed is sufficiently large. This is because the bandwidth will be measured as an average over the chosen time-interval, and is known as the *averaging* effect. And if the time-interval is very short, the resulting bandwidth will be unrealisticly high for very short periods of time.

Because the network interface receives datagrams packet by packet, and not by the packets individual bits, this becomes an issue. When every packet, of multiple bits, is reported being transferred instantaneously, the resulting bandwidth of a sufficiently small time-interval will go towards infinity. In practice making the *true* bandwidth indeterminable. So when thinking of bandwidth in terms of single bits, the notion of bandwidth itself ceases to exist, because the bits are not moving from the network stacks point of view. This dilemma is known as *Zenos Paradox*[18].

In the tests documented in this paper, the time-interval for bandwidth measurements is set to 1 second. This is because the collective amount of flow-export data that travels across the network when the buffers are being flushed, comes in bursts that lasts just under 1 second. So using a 1 second bandwidth average, will reflect the user perceived amount of bandwidth, actually being utilized.

5.2 Tests

When using the Harpoon application to generate traffic, the configuration files is used to fine-tune the parameters. Since the application works as a

client/server pair, one configures the amount of traffic being sent from the server to the client. The application does not take into account that the client needs to request a file transfer from the server, when configuring the bandwidth usage. In other words; the configured bandwidth usage is for uni-directional traffic. But the total amount of traffic generated on the network will be higher, as one measures the bi-directional traffic. The results presented in this section, will take the bi-directional traffic into account, as this will have an impact on normal network usage.

Note therefore that the actual bandwidth usage in the results, will be higher than what is configured in the traffic generator. This just to clear up any confusion about the individual numbers.

5.2.1 Initial Testing

For the first tests, the goal was really just to see how the implementation of the IPFIX protocol behaved when exposed to real data. The server part of the Harpoon application was set up to serve the client data files of 1500 bytes each. The client was set up with interconnection times of 0.1 seconds between each request. All of the traffic was sent using the TCP protocol. The configuration files was tuned so to give a total uni-directional bandwidth usage ranging from 0.5Mb/s to 3Mb/s. As discussed in section 5.2 on the preceding page, this would always yield a slightly higher bi-directional bandwidth usage. This can be seen from the resulting graphs. The test period was set to 300 seconds of actively.

The nProbe application was configured to use the default configuration, exporting data over the UDP transport protocol to the nTop collector.

Table 5.1 on the next page shows the information fields in use when exporting flows with the default configuration in nProbe. The size of each information field is given in bytes, and if one adds all of the fields together, one will find the total size of each exported flow. The table shows that each flow, using the default configuration in nProbe, will weigh in at 45 bytes. This is also the deciding factor, together with the current flow specification, when looking at the network overhead from the exporting process.

The graphs in figures 5.1 on page 48, 5.2 on page 48 and 5.3 on page 49 shows the results from 3 of the initial tests. The complete test results can be seen in table 5.2 on page 50. The first thing to mention about these tests, is that libpcap started dropping some packets in the two tests configured for highest bandwidth usage. That is; the tests configured for 2.5Mb/s and 3Mb/s experienced a 0.1% and 0.2% drop in packets (from libpcap) respectively. And since libpcap is utilized in both the exporting application, as well as in the measurements, the total error margin is actually doubled. This occurs when the libpcap instance in the exporter catches all of the packets which tcpdump

Size	Flow Label	Description
4	%IPV4_SRC_ADDR	IPv4 Source Address
4	%IPV4_DST_ADDR	IPv4 Destination Address
4	%IPV4_NEXT_HOP	IPv4 Next Hop Address
2	%INPUT_SNMP	Input Interface SNMP Idx
2	%OUTPUT_SNMP	Output Interface SNMP Idx
4	%IN_PKTS	Incoming Flow Packets
4	%IN_BYTES	Incoming Flow Bytes
4	%FIRST_SWITCHED	SysUptime (msec) of the First Flow Packet
4	%LAST_SWITCHED	SysUptime (msec) of the Last Flow Packet
2	%L4_SRC_PORT	IPv4 Source Port
2	%L4_DST_PORT	IPv4 Destination Port
1	%TCP_FLAGS	Cumulative of All Flow TCP Flags
1	%PROTOCOL	IP Protocol Byte
1	%SRC_TOS	Type of Service Byte
2	%SRC_AS	Source BGP AS
2	%DST_AS	Destination BGP AS
1	%SRC_MASK	Source Subnet Mask
1	%DST_MASK	Destination Subnet Mask

Table 5.1: **Description of the default nProbe information fields**: *The size of the individual information fields is given in bytes*

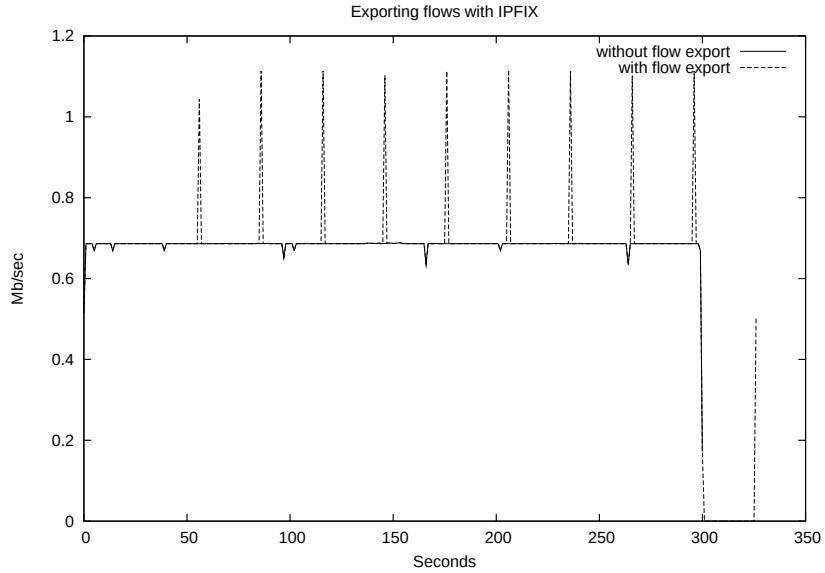

Figure 5.1: **Initial Testing**: *The graph shows the bandwidth usage both with, and without flow exportation in a system generating appr. 0.5Mb/s of uni-directional traffic*

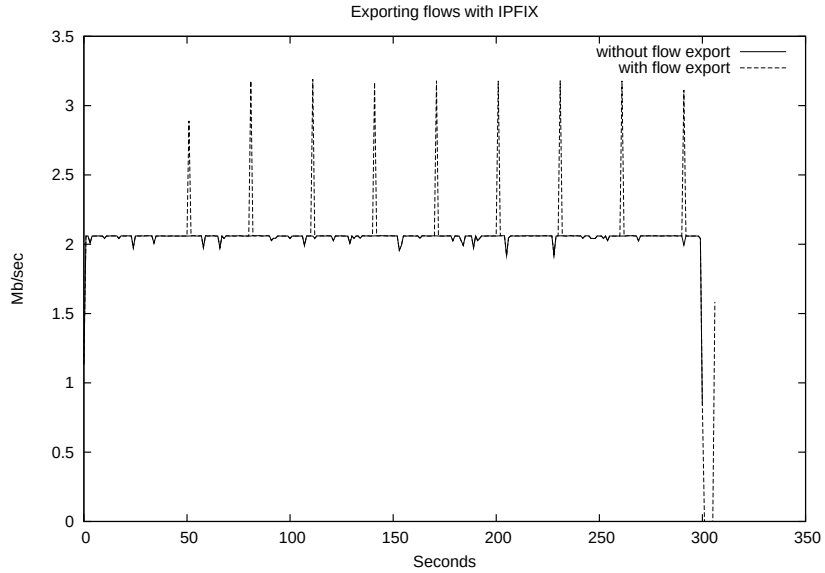

Figure 5.2: **Initial Testing**: *The graph shows the bandwidth usage both with, and without flow exportation in a system generating appr. 1.5Mb/s of uni-directional traffic*

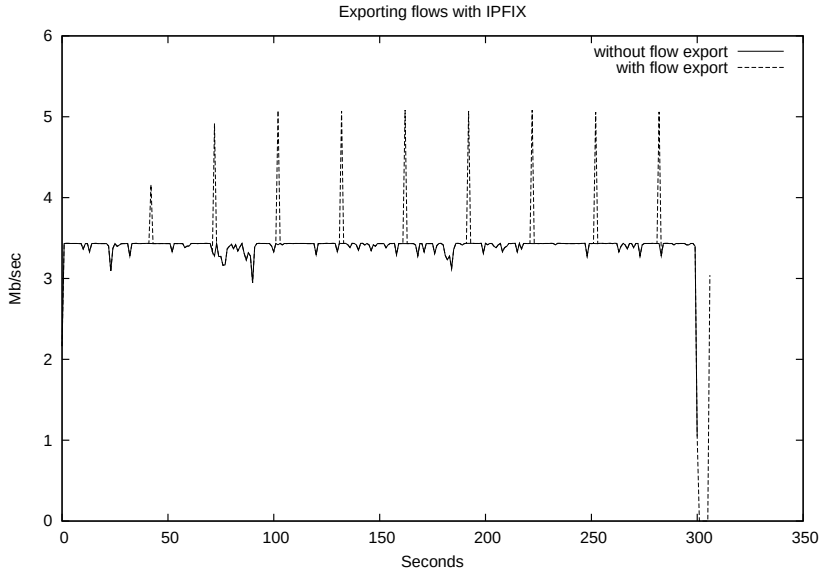

Figure 5.3: **Initial Testing**: *The graph shows the bandwidth usage both with, and without flow exportation in a system generating appr. 2.5Mb/s of uni-directional traffic*

drops, and vice versa. So there exists an error margin in those two tests, albeit a small one. In figure 5.3 this drop in packets is visualized in the normal network traffic baseline. The graph clearly shows a more *deteriorating* behavior than the other resulting graphs with no packet loss.

It is also apparent that the flow-data behaves in a very predictable manner, exporting the first flow after 60 seconds, and then every 30 seconds. The last burst of flow-data should not be considered in these intervals, as it only emitted when the nProbe application manually shuts down. This is the flow data left in the cache, that is being flushed on exit. The IPFIX protocol specifies 2 options for when the exporters cache should be flushed: either when the cache/buffer is full, or after a certain time. The reasons for this seems evident; when the cache/buffer is full, it needs to be flushed, and if a data connection is abruptly terminated, it needs to be marked as a complete flow sooner or later. Both the size of the buffer in the exporter, and the time interval between each flush, is configurable in the exporter.

Table 5.2 on the next page shows the amount of traffic sent over the network throughout the test, both by the traffic generator, and by the flow exporter. It also shows how much traffic overhead, using IPFIX to export flows would yield. The numbers for the tests with a configured traffic volume of 2.5Mb/s and 3.0Mb/s has an error margin of 0.2%and 0.4% respectively. Still the trend seems very indicative on how the flow export overhead behaves un-

	Traffic Data	Traffic & Flow Data	Flow Data	Overhead
0.5Mb/s	25712696	26245042	532364	2.07%
1Mb/s	51376676	52379346	1002670	1.95%
1.5Mb/s	77016182	78437548	1421366	1.85%
2Mb/s	102593210	104375966	1782756	1.74%
2.5Mb/s	127778403	129889569	2111166	1.65%
3Mb/s	153215883	155600621	2384738	1.56%

Table 5.2: **Volume of the data sent over the network**: *All of the data volumes are given in bytes, and the overhead is given as a percental increase over the normal traffic volume*

der increasing traffic loads. A simple graph showing this evolution, can be seen in figure 5.4 on the facing page. The main factor deciding the export overhead, is the underlying traffic. Since each flow being exported is 45 bytes, the overhead will depend on how much data that specific flow describes. The more data that is, the less the overhead becomes. Example: A flow of 45 bytes could theoretically describe 1GB of data on the network. The reason the flow overhead decreases with increasing traffic volumes, is two-folded.

Firstly: The transport protocol for the traffic generator is, in this case, TCP. While the transport protocol for flow export is UDP. The network overhead for TCP is larger than for UDP, so with an increasing volume of data being sent over TCP, the resulting flow-data being sent over UDP generates less overhead through the transport protocol.

Secondly: The timing of the max interval for a flow, is not always optimal in terms of efficiency. A flow that otherwise could have *fitted* within the other parameters, can be split up into two flows if it is not started at the same time as the flow time-interval. Resulting in twice as much flow-data being exported. This is, of course, not a trivial task to accomplish.

Figure 5.4 on the next page shows a fairly linear evolution of the IPFIX overhead, under increasing traffic volumes. This is to be expected, as the difference in traffic volume is 0.5Mb/S in each test. The small variances from a truly linear progression, comes from the time-interval deciding the individual flows.

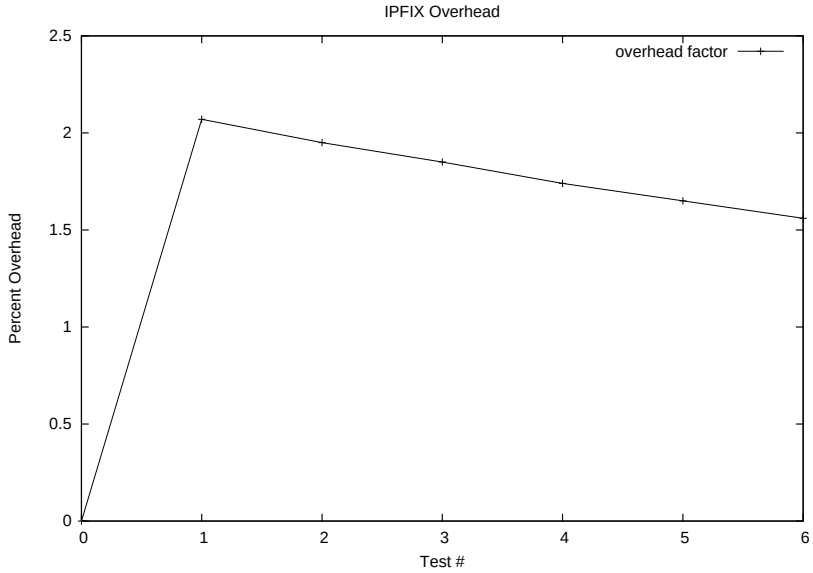

Figure 5.4: **Initial Testing**: *This simple graph shows the evolution of the overhead factor in the initial tests. Test nr. 1 has the lowest configured throughput (0.5Mb/s), gradually increasing to test nr. 6 which has the highest configured throughput (3Mb/S)*

5.2.2 Protocol Overhead

The second batch of tests, looks more closely at the IPFIX overhead compared to the network traffic. The Harpoon traffic generator was configured to generate traffic over the UDP protocol, as to reduce the total overhead generated by the use of the TCP transport protocol in the previous tests. The server part of the Harpoon application was configured to serve UDP packets with a payload of 45, 90 and 450 bytes, for the three individual tests respectively. The client part was configured for 50 concurrent sessions with interconnection times of 0.1 seconds between each request. The test period was set to 180 seconds. Examples of the configuration files for the UDP traffic, can be seen in the appendix section A.1 on page 71

The nProbe application was configured to use the default configuration, exporting data over the UDP transport protocol to the nTop collector.

Looking at the graphs in figures 5.5 on the next page, 5.6 on the following page and 5.7 on page 53, the first thing to notice is that the normal traffic baseline, does not appear to be as smooth as with the initial tests over TCP. This should not affect the results at all, since there is plenty of bandwidth left for the flow export. The low export itself seems just as predictable as the previous tests. At least when it comes to the intervals between the bursts of flow-data being exported. But more interesting, is looking at the amount of data being

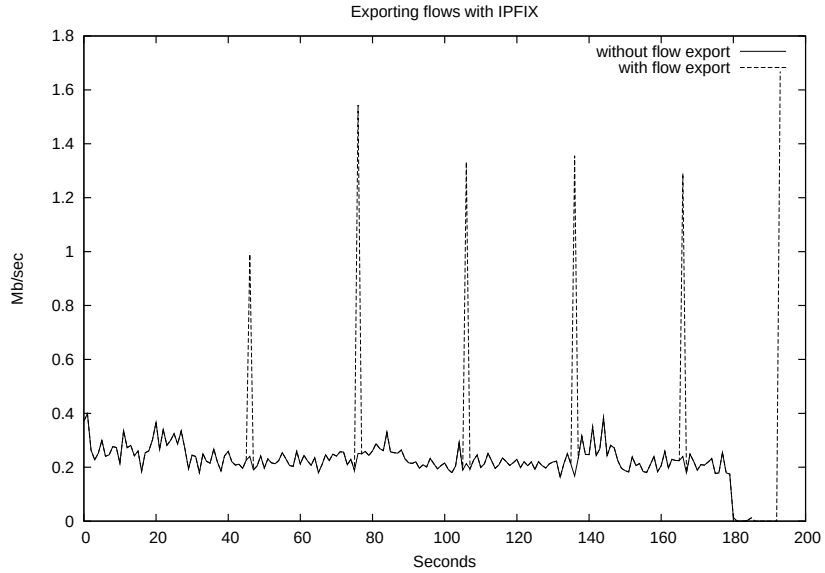

Figure 5.5: **Protocol Overhead**: *The graph shows the bandwidth usage both with, and without flow exportation, in a system generating traffic over UDP. 50 concurrent clients are being served UDP packets with a payload of 45 bytes*

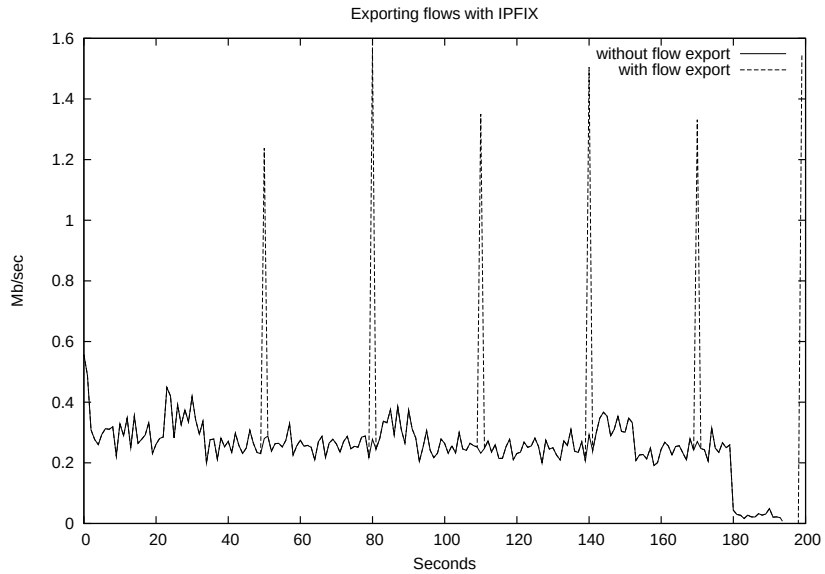

Figure 5.6: **Protocol Overhead**: *The graph shows the bandwidth usage both with, and without flow exportation, in a system generating traffic over UDP. 50 concurrent clients are being served UDP packets with a payload of 90 bytes*

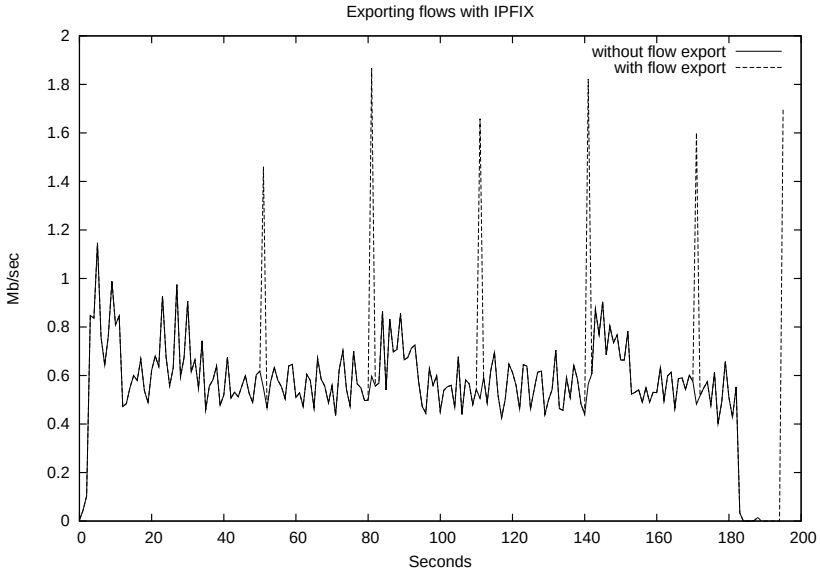

Figure 5.7: **Protocol Overhead**: *The graph shows the bandwidth usage both with, and without flow exportation, in a system generating traffic over UDP. 50 concurrent clients are being served UDP packets with a payload of 450 bytes*

exported, compared to the amount of data in the generated traffic. Since now both types of data are being sent over the UDP transport protocol, the overhead of the transport protocol itself, should be eliminated.

	Traffic Data	**Traffic & Flow Data**	**Flow Data**	**Overhead**
Test # 1	42188256	49258960	7070704	16.76%
Test # 2	49798480	56990944	7192464	14.44%
Test # 3	107888272	115304576	7416304	6.87%

Table 5.3: **Volume of the data sent over the network**: *All of the data volumes are given in bytes, and the overhead is given as a percental increase over the normal traffic volume. For the generated traffic, payloads of 45, 90 and 450 bytes have been served over UDP*

Table 5.3 shows how much data that was sent over the network in the 3 tests, and the calculated overhead on the traffic data when when exporting flows. 1 parameter varied in the individual tests, namely the size of the file being served to the clients in the traffic generation phase. In test nr. 1 (figure 5.5 on the preceding page) the payload was set to 45 bytes, in test nr 2 (figure 5.6 on the facing page) the payload was 90 bytes, and finally in test nr. 3 (figure 5.7) the payload was 450 bytes. Theoretically, an exported flow will always have

the same size. In this case, the default settings implies that the size on 1 flow will be 45 bytes (see table 5.1 on page 47). But the actual data it describes is not limited to 45 bytes, but can in theory be any size, depending on what type of data is being sent. The limiting factor is often the maximum time interval for the flow data. This means that the more data that actually goes into each flow, the less overhead it will be when exporting that flow.

There is only a slight increase in the volume of the flow-data in the 3 separate test, despite the fact that the volume of the generated data increases significantly. This is reflected in the column with the calculated amount of overhead created by the flow-export process in the individual tests. Under optimal conditions, the volume of the flow-data would not increase. The reason it does so in this batch of tests, is because of timing issues, and maybe also because of imperfections in the traffic generator.

The timing issues mentioned in the initial tests, of course remain the same for this batch. The *problem* is that a connection that normally could fit into one flows *timeslot*, will *spill* over into a new flows *timeslot*, if connection starts late enough into one such *timeslot*. When this happens, the volume of the flow(s) describing the connection, will double. As more network data is generated, more data will *spill* over into 2 *timeslots*, and the volume of the flow-data will increase.

There is also a possibility that the traffic generator itself does not produce the exact same number of connections, even if it is configured to do so. It is however unlikely that this should have any significant impact on the results, as the traffic generator is just as likely to produce less connections as it is to produce more.

5.2.3 Varying Information Fields

The underlying traffic on the network, is not the only factor deciding the overhead when exporting flows. Since NetFlow v9, the flow format has been template based, and can as such be configured to export flows of very little size. IPFIX is heavily based on NetFlow v9, and is even known as NetFlow v10, and can thus be configured in a similar manner. In the following tests, the nProbe application was configured to use 3 different templates, which again yielded 3 different sizes on each flow exported. The first test used the default template described in table 5.1 on page 47, where each flow-record exported was 45 bytes in size. The next test utilized the template described in table 5.5, giving flow-records with the size of 16 bytes. The final test used a very small template, described in table 5.4, resulting in flow-records with a size of 6 bytes.

Size	Flow Label	Description
4	%IPV4_SRC_ADDR	IPv4 Source Address
2	%L4_SRC_PORT	IPv4 Source Port

Table 5.4: **Description of the 2 configured nProbe information fields**: *The size of the individual information fields is given in bytes*

Size	Flow Label	Description
4	%IPV4_SRC_ADDR	IPv4 Source Address
4	%IPV4_NEXT_HOP	IPv4 Next Hop Address
4	%IN_BYTES	Incoming Flow Bytes
2	%L4_SRC_PORT	IPv4 Source Port
1	%TCP_FLAGS	Cumulative of All Flow TCP Flags
1	%SRC_TOS	Type of Service Byte

Table 5.5: **Description of the 6 configured nProbe information fields**: *The size of the individual information fields is given in bytes*

The Harpoon traffic generator was configured to use the TCP transport protocol, serving files of 1500 bytes each, and tuned to generate approximately 1.2Mb/s of uni-directional traffic. The same settings was used in all of the tests, and each test was set to last 180 seconds.

The graphs in figures 5.8 on the following page, 5.9 on the next page and 5.10 on page 57 all show a familiar pattern, comparable to the result graphs from the previous tests. Just by looking at the graphs, one can see that the total volume of the emitted flows, is decreasing in size as the number of exported

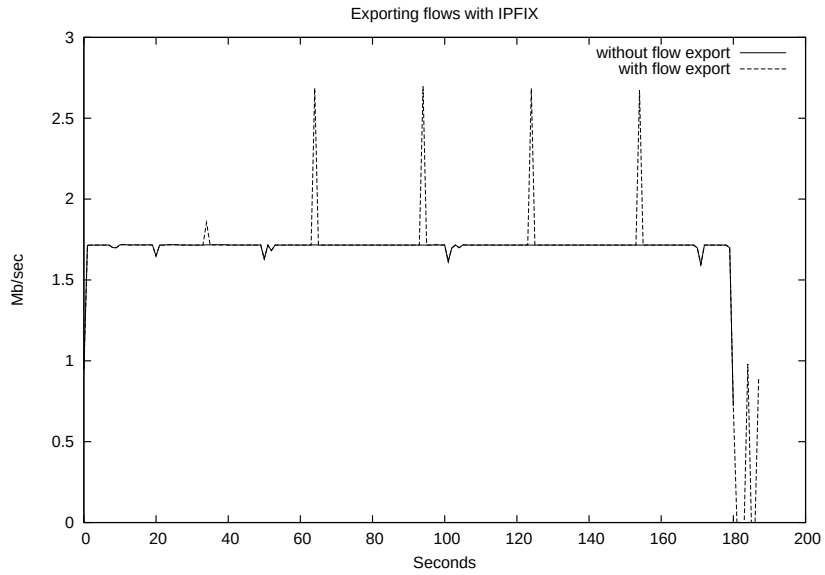

Figure 5.8: **Varying Information Fields**: *The graph shows the bandwidth usage both with, and without flow exportation in a system generating appr. 1.2Mb/s of unidirectional traffic, using the default amount of information fields for export, totalling 45 bytes per flow*

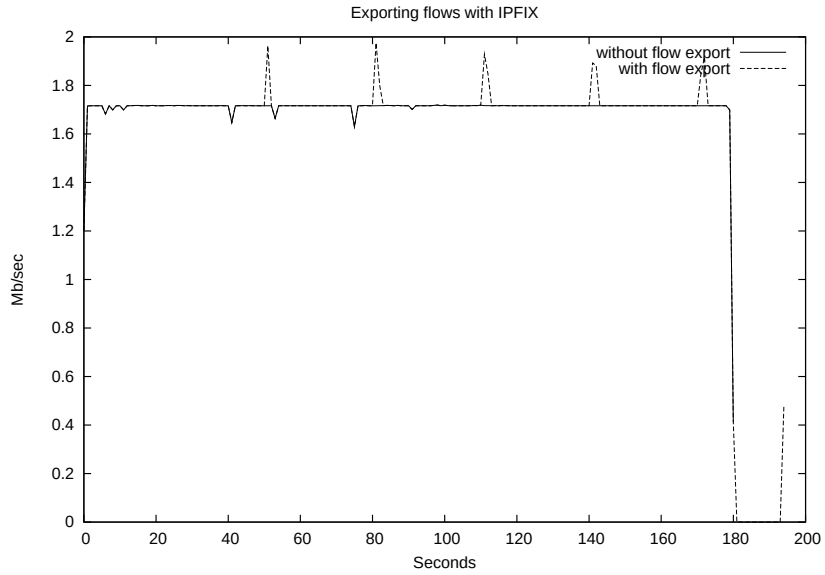

Figure 5.9: **Varying Information Fields**: *The graph shows the bandwidth usage both with, and without flow exportation in a system generating appr. 1.2Mb/s of unidirectional traffic, using 6 information fields for export, totalling 16 bytes per flow*

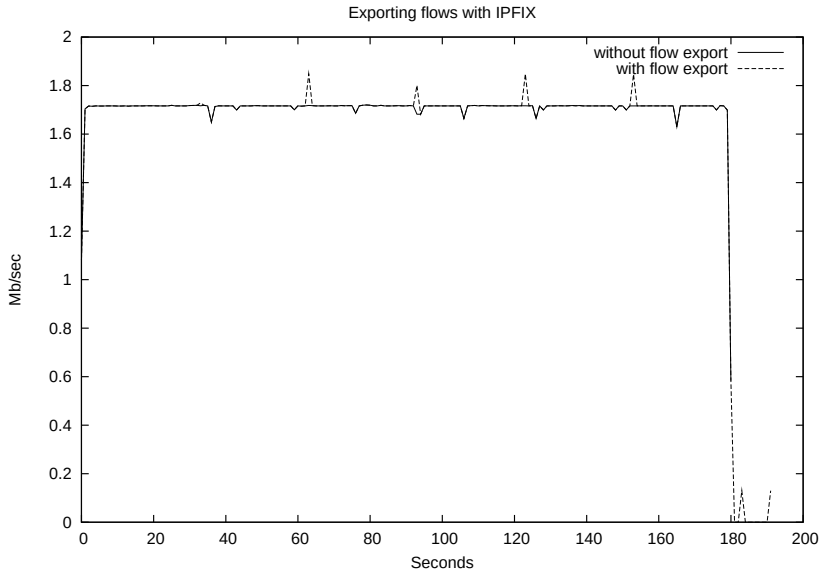

Figure 5.10: **Varying Information Fields**: *The graph shows the bandwidth usage both with, and without flow exportation, in a system generating appr. 1.2Mb/s of uni-directional traffic, using a 2 information fields, totalling 6 bytes per flow*

	Traffic Data	**Traffic & Flow Data**	**Flow Data**	**Overhead**
Test # 1	38543033	39278999	735966	1.91%
Test # 2	38565072	38825170	260098	0.67%
Test # 3	38551500	38649710	98210	0.25%

Table 5.6: **Volume of the data sent over the network**: *All of the data volumes are given in bytes, and the overhead is given as a percental increase over the normal traffic volume*

information fields in the templates goes down. Table 5.6 on the previous page presents the raw data gathered from the tests.

As expected, the volume of the exported flows, will drastically decrease with the number (and size) of the information fields in the current template. Not only that, but since the size of each exported flow is known, along with the total size of all flows, the number of flows in each test can be calculated by dividing the total size with the size of one flow. The number of flows should theoretically be the same for each test, since the same configuration in the traffic generator was used. But for reasons earlier mentioned (flow time-interval and imperfections in the traffic generator), there will always be small differences.

Test nr 1 exported flows with the size of 45 bytes, and had a total export volume of 735966 bytes. This means that it exported **16355 flows**.

Test nr 2 exported flows with the size of 16 bytes, and had a total export volume of 260098 bytes. This means that it exported **16256 flows**.

Test nr 3 exported flows with the size of 6 bytes, and had a total export volume of 98210 bytes. This means that it exported **16368 flows**.

These numbers seem consistent enough to indicate that the exporter functions as expected, and after the IPFIX specifications.

5.2.4 Comparing Export Protocols

As mentioned, there exists several protocols and formats for exporting flow-data. The predominant protocol seems to have been Ciscos NetFlow v5, which is in the process of being superseded by NetFlow v9. IPFIX is heavily based on NetFlow v9, and is often referred to as NetFlow v10. It therefore seems natural to make a quick comparison of these protocols. The nProbe application supports all 3 protocols, and was configured to use these in the separate tests. The test procedure was similar to the one described in section 5.2.3 on page 55, and the Harpoon traffic generator was configured to use the TCP transport protocol, serving files of 1500 bytes each, and tuned to generate approximately 1.2Mb/s of uni-directional traffic. The same settings was used in all of the tests, and each test was set to last 180 seconds.

The nProbe application was set to use the default configuration for all protocols, exporting data over the UDP transport protocol to the nTop collector. For the template based protocols, NetFlow v9 and IPFIX, this meant that each flow record would be 45 bytes in size with a header of 20 bytes. NetFlow v5 has a fixed format, and consists of a 24-byte header and a 48-byte payload, totalling 72 bytes.

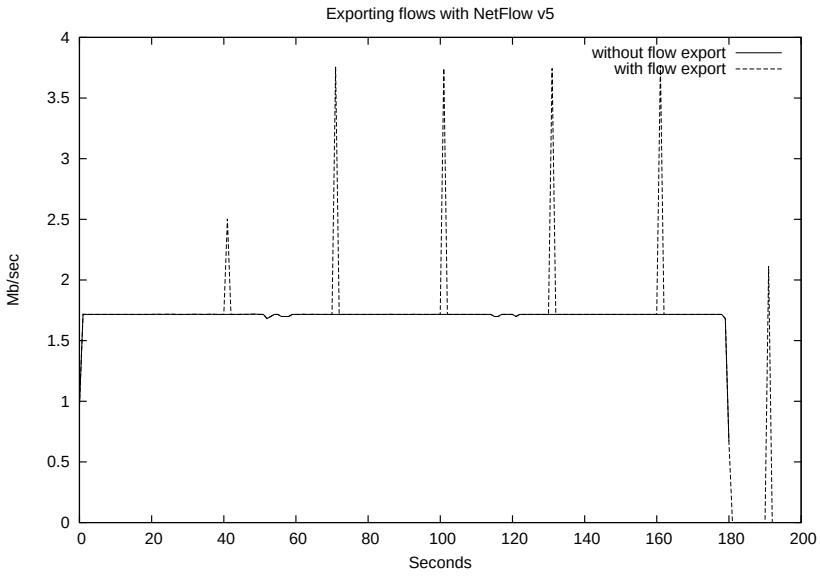

Figure 5.11: **Comparing Export Protocols**: *The graph shows the bandwidth usage both with, and without flow exportation in a system generating appr. 2.5Mb of unidirectional traffic, using the NetFlow v5 protocol*

From the graphs in figures 5.11, 5.12 on the following page and 5.13 on the next page, it seems clear that all the protocols show similarities. But from

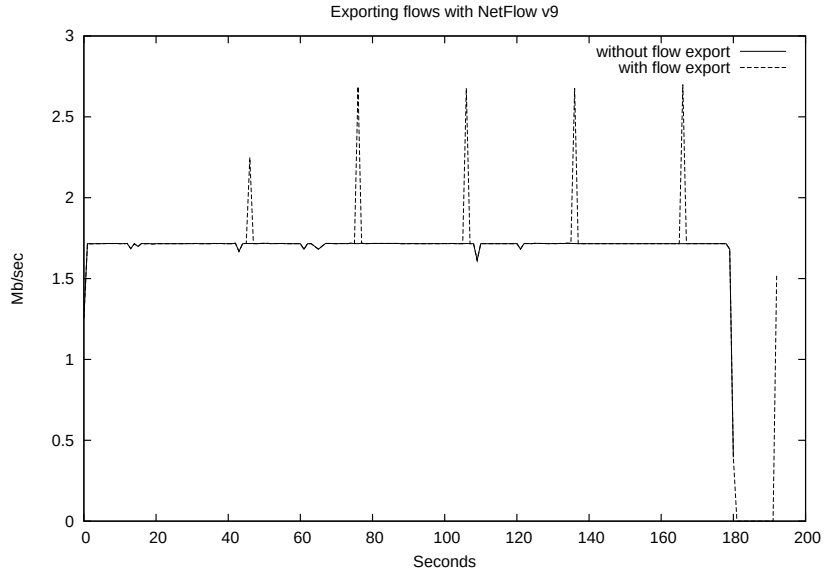

Figure 5.12: **Comparing Export Protocols**: *The graph shows the bandwidth usage both with, and without flow exportation in a system generating appr. 2.5Mb of unidirectional traffic, using the NetFlow v9 protocol*

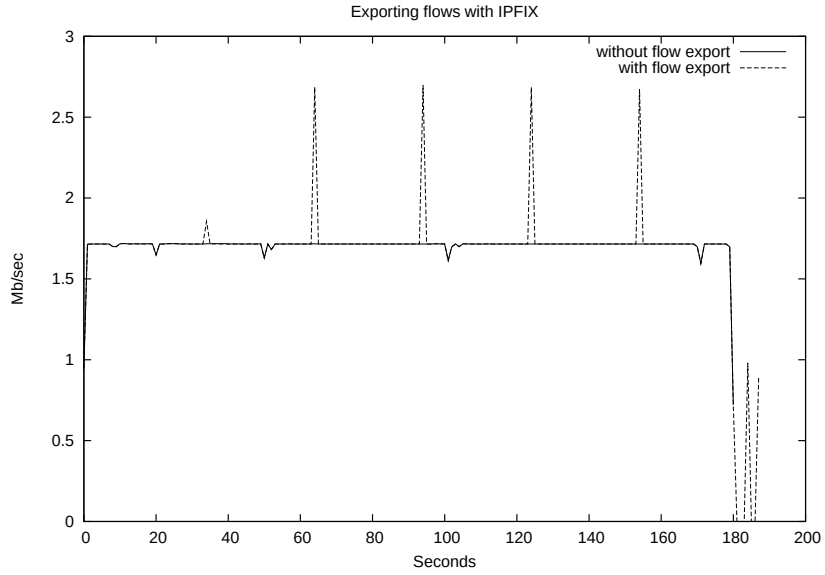

Figure 5.13: **Comparing Export Protocols**: *The graph shows the bandwidth usage both with, and without flow exportation in a system generating appr. 2.5Mb of unidirectional traffic, using the IPFIX protocol*

the looks of it, it seems that NetFlow v5 yields a higher overhead than NetFlow v9 and IPFIX, which again seems to perform on par with each other. More information on NetFlow v5 and NetFlow v9 can be obtained from Ciscos website[19] and RFC3954[20]. The fact that it is the nProbe application that is utilizing the different protocols with default settings, is also a reason to why the three protocols shows such similar behavior in the tests.

	Traffic Data	**Traffic & Flow Data**	**Flow Data**	**Overhead**
NetFlow v5	38579476	40141608	1562132	4.05%
NetFlow v9	38561094	39302088	740994	1.92%
IPFIX	38543033	39278999	735966	1.91%

Table 5.7: **Volume of the data sent over the network**: *All of the data volumes are given in bytes, and the overhead is given as a percental increase over the normal traffic volume*

Looking at the resulting volumes of traffic in table 5.7, confirms the findings from the graphs. The overhead from the NetFlow v5 protocol is almost double of what is found in NetFlow v9 and IPFIX. NetFlow v9 and IPFIX, on the other hand, produce almost the exact same overhead, relative to the traffic they are exposed to. This may at first seem a bit surprising, since the size of each flow exported in NetFlow v5 almost matches the default sizes for NetFlow v9 and IPFIX, and should accordingly produce almost the same overhead from the equal number of flows. And the volume of the generated data in the 3 tests should not have any impact on the relative overhead in each protocol. The answer to this problem is found when looking at the export-logs for the three tests. While IPFIX and NetFlow v9 finds 15670 and 15734 flows respectively, the NetFlow v5 exporter finds 31404 flows in the generated traffic.

The *export process* in NetFlow v5 itself, only yields a slightly higher overhead than NetFlow v9 and IPFIX. This is confirmed in the whitepapers for the protocols. But the *metering process* divides the traffic up into just about double the number of flows. This, of course, has a huge impact on the calculated overhead.

5.2.5 Summary

To evaluate the IPFIX protocol, it is necessary to focus on the transport phase, between the exporter and the collector. The metering process in the collector, and the data storage in the collector, are subject to the individual implementations of the protocol. They are, however, a necessary part of a monitoring system based on flow export, and should not be completely overlooked. The tests in this chapter has been designed to shed light on the issues common to the transport phase, and the architecture of the system. The results from the tests, have been presented in an objective way, and should be individually evaluated relative to the needs and limitations of the reader.

Most of the results are presented as graphs of network behavior, and tables containing the total traffic volumes. These two models should be carefully compared to each other, and also to the transport protocol used for flow export. E.g. the total calculated overhead in one test, while giving the correct volume of traffic needed when implementing a flow-exporter, does not give a true picture of what is needed of available bandwidth for that same implementation. The graphs, on the other hand, gives a better understanding of the instant network usage, but a more poorly measure of the specific volumes. The choice of transport protocol for the flow export, will decide the criteria for available network resources. In all of the tests described in this document, and for many real life implementations, UDP is the choice of transport protocol. This because it is supported on most platforms, and has very little overhead compared with TCP (and even less overhead compared to SCTP). Using any of the two other supported transport protocols, would mean more overhead for flow export, but also a more reliable means of transportations. In other words, there are both positive and negative effects of using the different transport protocols, especially on networks with limited resources. Testing all possible scenarios, is out of the scope for this document. A reasonably good understanding of UDP and TCP (and preferably SCTP), is therefore needed to evaluate the results in this chapter.

Chapter 6

Conclusions, Discussion and Future Work

This chapter will try to summarize the results of the completed tests, and comment on the protocol and its applications.

6.1 IPFIX

One of the major challenges when writing this document, was that working with such a new and untested protocol meant that sources of information, related work and working implementations was hard to find. On the other hand, this was perhaps also the greatest motivation to choose such a subject. But because of this, the choice of implementation to test, was in practise limited to one application; namely nProbe. And concern has been raised, that great care must be taken, so not to evaluate the implementation of a protocol, but rather the protocol itself. This has been handled by, amongst other things, ensuring that the system resources on the nodes running the application, never was a limiting factor during tests. This is also why focus has been on the actual transport/export of data between exporter and collector, and not the efficiency of the specific nodes when doing so.

While being a fairly new proposed standard, IPFIX is an evolution of the tried and tested generations of the NetFlow protocol. It is heavily based on NetFlow v9, and does not bring any radical changes to the table. One of the goals of the IPFIX working group is to standardize existing practice in the area of IP Flow Information Export, and thus securing the interoperability between vendor specific products. This is by no means an easy task, but one that seems to be within reach of the project. This is perhaps the most *weighted* goal as well, since it does not seem that IPFIX comes with any other significant advantage (performance, usability or otherwise), over comparable protocols such as NetFlow v9. But even without any obvious significant advantages, the IPFIX

protocol still is a solid performer compared to other flow export protocols. And the introduction of template based flow information specifications, not found in earlier incarnations of NetFlow (before v10), means IPFIX provides a highly flexible way of specifying and exporting flow information. But to utilize the flexibility to its full potential, a lot of manual configuration, and prior knowledge of network usage, is needed.

The tests described in this document clearly identifies 3 parameters that affects the overhead, and thereby the efficiency, of the IPFIX protocol. Namely: the characteristics of the underlying traffic, the specific flow-template in use, and lastly the choice of transport protocol. Starting with the first parameter, the underlying traffic, it comes as no surprise that it has an impact on the produced overhead of the flow export. Using the definition of a flow as done by IPFIX, a flow could describe GB worth of traffic on the network, or just a few bytes. This characteristic is common to all export protocols dealing with flows, and should be considered a mutual phenomena, and not something specific for the IPFIX protocol. The second parameter, the templates used for flow information retrieval, is where IPFIX (along with NetFlow v9) really shines. This mechanism makes the protocol extremely flexible, when compared to similar protocols. This flexibility can be used to tune the export settings to a level where the generated overhead is minimal. The challenge here, is to find the right balance between the flow information needed, and the generated overhead. And depending on the type of analysis, the need is often present for a large number of information fields. This again means that there seldom is room for large reductions in the templates. And all changes to the template, must be made manually, so this quickly becomes a labor of *trial and error*. Especially if the present traffic is somewhat dynamical in nature. The third parameter, is the type of transport protocol chosen for the flow export. Here IPFIX specifies that any implementation needs to support the use of both the traditional UDP and TCP protocols, as well as the relatively new SCTP protocol. The choice of transport protocol, has an effect on the bandwidth utilization when exporting, but also on the reliability of the export itself. It seems that UDP is currently the predominant choice, because it yields the least overhead on the network. It is also the least reliable protocol, but that does not seem to be a deciding factor for most network operators. This is probably due to the fact that the type of analysis being made, only requires an estimate of the traffic, and not an exact measurement for any given point in time. Because of this, *sampling* is also a very common strategy to limit resource usage on the network when exporting flows. The IPFIX working group recommends SCTP as the protocol for transporting flows. And while it produces an even greater overhead than both UDP and TCP, it also should provide an additional layer of reliability, by utilizing a mechanism for congestion control on the network. Perhaps the biggest problem with SCTP, is the fact that it is a fairly new and untested protocol. Firstly this means that only a select few platforms actually

supports it, making it difficult to implement. An example of which actually occurred in the experimental setup for the tests in this document. Secondly, since the protocol is fairly untested, there might be security risks or unknown problems involved, when using it in a production environment. One thing worth noting, is that the nature of the protocol itself, makes it a candidate for exploitation through a *Denial of Service* attack. Generally speaking, a low adoption rate of SCTP is probably one of the biggest obstacles for IPFIX to pass, on its way to becoming the leading industry standard.

Since the measurements taken in the tests are of quantitative data captured between the exporter and the collector, the specific implementation of the IPFIX protocol should not have any impact on the final results. Given that the implementation follows the protocol specifications, of course.

One issue that has yet to be properly addressed, is knowing the impact on the existing traffic on the network, when implementing IPFIX (or any other flow information export protocol, for that matter). One of the main reasons to export flow information, is to analyze the network usage. In other other words; to effectively implement flow information export, one needs to know a lot about the type of traffic present on the network, in advance. The status of the current protocols for flow export, suggests that if one does not have the proper knowledge of the existing network usage, the implementation of such protocols could have consequences for the existing traffic. This dilemma is to a certain degree addressed by C. Estan et. al. in the paper *Building a Better NetFlow*[21], discussed in subsection 6.2.1 on the next page. The solution presented therein is not ideal, as it handles data-sampling as a prerequisite for adapting the flow information export. But perhaps some of the *adaptive ideas* could be incorporated into a protocol that does not rely on data-sampling, but instead adjusts the information fields in the existing flow-templates used for exportation.

IPFIX is a *push-based* protocol on top of a *centralized* architecture. It is a natural evolution from the early NetFlow protocols, adding valuable mechanisms, such as templates and wider options for the choice of transport protocol. But no matter how much refinement that has been put into the development of the protocol, it can not escape its push-based/centralized nature. In a world where the evolution is pointing to more complex network topologies, higher capacity links with larger volumes of traffic flowing through them, the question remaining is if a protocol based on the NetFlow design will be able to scale up with the increasing demands of the public. The NetFlow protocols was designed to be push-based, because they were designed to be run on nodes (switches, routers) with very limited hardware capabilities. These nodes had no resources for storing and processing vast amounts of information. It therefore makes perfect sense to export this data through a push-based protocol. That way, the node can flush its buffer of data when it reaches a level near its maximum capacity. The NetFlow protocols were also designed to be used in

a centralized architecture, where the exporting nodes would push all of their data onto a central node. This central node was not a switch or a router, but rather a generic computer. It did not have the same limitations of resources as the exporting nodes. It therefor made sense to let one central node do all of the heavy processing of the data, to offload any additional stress on the exporting nodes. As the complexity of the topology, volume of traffic and capacity of the networks has grown, so has the capacity of the routing and switching nodes. And the idea of a push-based/centralized protocol, might not be the best option any more. Especially when it comes to scalability. When scaling upwards, this architecture will hit two barriers. Number one is network capacity; as the load on the network increases, the available bandwidth for flow export will decrease. Unfortunately the need for bandwidth to export flows will grow proportionally to the original network load. And if multiple exporters are experiencing the same increase in network traffic, this effect will be further strengthen when these exporters are pushing traffic to one central point. Number two is collector capacity; one central node receiving flow information from several collectors will experience the collective load from all of the exporters. Its capacity to process all this data will probably be lower than the collective capacity from all the exporters. And since IPFIX is push-based, the collecting node has no influence on when to request incoming flow data, so to distribute this over time.

In the authors opinion, the introduction of SCTP as the preferred transport protocol, is an effort to push back the *scalability boundary* of the protocol. It is note a *cure* of the problem, but rather a *treatment* to prolong the protocols lifespan. With the rising demands of modern networks, a need for a more scalable flow information export protocol will surely surface. Specialized hardware such as routers and switches have increased processing capacities, making different types of architecture a more feasible option. Either making the protocol pull-based from the central collectors point of view, or maybe even more radical changes such as making the whole protocol decentralized, would probably be a more solid foundation for a scalable flow information export protocol.

6.2 Future Work

6.2.1 Adaptive NetFlow

Some of the essence learned from the work put into this document, is that utilizing NetFlow to analyze network behavior, is somewhat of a *Catch 22*; To effectively know what impact implementing flow export on a network will have, one needs to know about the amount and type of traffic already on the network. That might be tricky since that is also the reason to implement export of flow data; to gain information about the amount and type of traffic utilizing

the network.

C. Estan et. al. proposes a modification to the traditional NetFlow protocol, called *Adaptive NetFlow*[21]. This modification addresses some of the shortcomings present in older NetFlow versions, hindering the collection and subsequent analysis of data from the network. One of the more common problems with NetFlow protocols, is the vulnerability to *Denial of Service* attacks; During flooding attacks, both routing-node memory and network bandwidth can be consumed by generated flow-data, exceeding normal operating boundaries. And while this can be countered by the introduction of data-sampling in the flow-exporter, a single, static sample rate will not accommodate the plethora of underlying traffic mixes, resulting in a skewed resource/accuracy relationship for certain types of traffic. Another issue with traditional NetFlow, is the routing-nodes inability to report, without bias, the number of active flows for aggregates with non-TCP traffic.

To help solve some of these problems, Estan et. al. introduces some new concepts for the NetFlow protocol. Firstly they propose a mechanism to automatically adjust the sampling rate for the network data, to a level that does not deplete the resources in the exporting, node or the network as a whole. This is done by starting at the maximum sampling rate sustainable in the node, and adjusting it down as the traffic mix increases in volume. This mechanism will continuously calculate the sampling rate needed for not exhausting the memory in the exporting nodes. This will, of course, also affect the produced volume of flow data. In order to keep the the final result of flow data consistent during such a procedure, the byte and packet count of the existing entries must be adjusted accordingly. This process is dubbed *renormalization*. Secondly they propose a mechanism called *Flow Counting Extension*, capable of estimating the number of active flows in various aggregates, including non-TCP traffic. This is based on an algorithm called *adaptive sampling*, and comes in addition to the mechanism for counting active TCP flows. This estimated number of active flows, is better suited to detect anomalies in non-TCP traffic, and can then be used as a parameter when adapting the sampling-rate to accommodate the current traffic-mix. Anomalies in non-TCP traffic can be indicators of malicious scans, and worms.

While the focus when developing *Adaptive NetFlow*, was to provide a robust layer to traditional NetFlow during flooding attacks or surges of traffic. It is easy to see the advantages of an *adaptive approach* when implementing a system for flow information export on a network, without the prior knowledge of the composition of traffic present on the network. And also on network with large fluctuations in network utilization, an adaptive, rather than static, strategy of data collection would be preferable.

6.2.2 IPFIX Aggregation

The *IPFIX Aggregation* Internet-Draft[22], is working document for the IETF, describing the aggregation of IPFIX flows. It addresses issues concerning the build up of high volumes of flow-data to be exported to a central collecting node. There are situations where the level of flow information, can be heightened without loss to the analyzation process. Using aggregation techniques, measurement information from several similar flows can be aggregated into one flow aggregate. The rules for choosing similar flows and grouping subsets of flows, can be customized. By extending the original IPFIX protocol and information model to include new abstract data types and template sets, efficient transactions of both aggregated flows and the rules of which they are produced, can be achieved between exporting and collecting nodes.

Utilizing a centralized architecture, the collector in a flow information export system can often become a bottleneck. Especially in typical high-speed, large-scale network, capable of sustaining large volumes of data transfers. Flow measurements in theses types of environments, will produce huge amounts of flow-data. But at the same time, many applications for processing flow measurement data does not require detailed flow-level information, but rather collected information from flow aggregates. The level of flow information, are application-specific. The Internet-Draft presents a scheme for such flow aggregation, reducing the number and size of exported flow records, and adapting to the application-specific needs for flow details. This is achieved by discarding unnecessary flow information, and aggregating similar flows into composite flow aggregates before exportation.

The Internet-Draft specifies 2 possible architectures for flow aggregation. An internal aggregator can be implemented as a process, running on an IPFIX enabled device. Aggregating flows captured by the metering processes, and exporting them as one flow aggregate. An external aggregator can be deployed as an extra layer of hardware in the network topology. Such external devices, capable of aggregating IPFIX flows, are called *concentrators*. The use of external aggregators/concentrators, will enable cascading, multi-level aggregation of flows. In such a topology, the concentrators can be arranged in a hierarchical manner, aggregating child-flows to a parent concentrator or a final collector for analysis.

The configuration of the aggregators, is proposed handled in a rule-based approach. A list of aggregation rules, each consisting of instructions for each flow information field, must be supplied to the aggregator. each instruction should consist of 3 elements. The first element should identify which information field the instruction is intended for. The second element should inform the aggregator what to do with the flow information field; discard, keep, mask or aggregate. the third field should have an optional parameter that must be matched for the flow to be aggregated; e.g. 10.10.0.0/16. This way, each rule

will define the content of the flow record as well as the template to export the flow aggregate information. Fields not present in the aggregation instructions are not part of the flow record.

Appendix A

Configuration Files

A.1 Harpoon Configuration Files

A typical configuration file for the server part of a Harpoon traffic generation session will look like the following. Here TCP-traffic is generated, with the server serving packets with the size of 1500 bytes. The server will serve up to 10 clients at a time, and have the IP address of 192.168.0.2

```
<harpoon_plugins>

  <plugin name="TcpServer" objfile="tcp_plugin.so"
          maxthreads="50" personality="server">
      <file_sizes> 1500 </file_sizes>
      <active_sessions> 10 </active_sessions>

      <address_pool name="server_pool">
          <address ipv4="192.168.0.2/32" port="10000" />
      </address_pool>
  </plugin>

</harpoon_plugins>
```

The client configuration file from the same session, will look slightly different. Once again TCP is chosen as the protocol, and there is a pause of 0.1 seconds between each connection. The client will spawn 10 parallel sessions to the server, and the IP addresses of both server and client is defined.

```
<harpoon_plugins>

  <plugin name="TcpClient" objfile="tcp_plugin.so"
          maxthreads="50" personality="client">
```

```
      <interconnection_times> 0.1 </interconnection_times>
      <active_sessions> 10 </active_sessions>

      <address_pool name="client_source_pool">
            <address ipv4="192.168.0.1/32" port="0" />
      </address_pool>

      <address_pool name="client_destination_pool">
            <address ipv4="192.168.0.2/32" port="10000" />
      </address_pool>
  </plugin>

</harpoon_plugins>
```

This is the server part of a configuration for generating UDP traffic. The server will serve up to 50 concurrent clients packets with a payload of 450 bytes.

```
<harpoon_plugins>

  <!-- constant bit-rate UDP sources -->
  <plugin name="UDPCBRserver" objfile="udpcbr_plugin.so"
          maxthreads="50" personality="server">

      <file_sizes>
         450
      </file_sizes>

      <active_sessions> 50 </active_sessions>

      <address_pool name="server_pool">
            <address ipv4="128.39.73.22/32" port="10001" />
      </address_pool>
  </plugin>

</harpoon_plugins>
```

The client part of the UDP configuration will specify the interconnection times between each request and the number of concurrent sessions.

```
<harpoon_plugins>

  <!-- constant bit-rate UDP sources -->
  <plugin name="UDPCBRClient" objfile="udpcbr_plugin.so"
```

```
          maxthreads="50" personality="client">

      <active_sessions> 50 </active_sessions>
      <interconnection_times>
         0.1
      </interconnection_times>
      <datagram_size> 10000 </datagram_size>
      <bitrate> 1000000 </bitrate> <!-- bits per second -->

      <address_pool name="client_destination_pool">
            <address ipv4="128.39.73.22/32" port="10001" />
      </address_pool>

      <address_pool name="client_source_pool">
            <address ipv4="128.39.73.23/32" port="0" />
      </address_pool>
  </plugin>

</harpoon_plugins>
```

Appendix B

Scripts

B.1 Python Normalization Script

The following python script is used for post processing of the measured bandwidth results. It just converts b/s to Mb/s and normalizes timeline of the measurements with regards to the sample-interval. The *factor* variable should equal the number of seconds used for the sample-interval (which again should be a positive integer).

```
#!/usr/bin/env python
import sys, string

try:
        datafile = sys.argv[1]; outfilename = sys.argv[2];
        factor = sys.argv[3];
except:
        print "usage: " + sys.argv[0] + " <datafile> \
        <outfilename> <factor>"; sys.exit(1)

ifile = open(datafile, 'r')

ofile = open(outfilename, 'w')


for l in ifile.readlines():

                counter = 1
                value = l.split()[0]
                value = float(value) / 1000000
                while counter <= int(factor):
                        ofile.write('%s\n' % (value))
```

```
                        counter = counter + 1

ofile.close()
```

Appendix C

Tables

C.1 IPFIX Template Values

Table C.1 shows an overview of the different fields available in a template that can be used when exporting flows with nProbe in the IPFIX format. In other words, these are the values that can be extracted from the packets flowing on a network, and exported to a central collector for analysis.

Table C.1: **Description of IPFIX Export Format Options**: *As a template based export format, IPFIX can be very flexible with regards to the needs of the monitoring application*

ID	Flow Label	Description
[1]	%IN_BYTES	Incoming flow bytes
[2]	%IN_PKTS	Incoming flow packets
[3]	%FLOWS	Number of flows
[4]	%PROTOCOL	IP protocol byte
[5]	%SRC_TOS	Type of service byte
[6]	%TCP_FLAGS	Cumulative of all flow TCP flags
[7]	%L4_SRC_PORT	IPv4 source port
[8]	%IPV4_SRC_ADDR	IPv4 source address
[9]	%SRC_MASK	Source subnet mask (bits)
[10]	%INPUT_SNMP	Input interface SNMP idx
[11]	%L4_DST_PORT	IPv4 destination port
[12]	%IPV4_DST_ADDR	IPv4 destination address
[13]	%DST_MASK	Dest subnet mask (bits)
[14]	%OUTPUT_SNMP	Output interface SNMP idx
[15]	%IPV4_NEXT_HOP	IPv4 next hop address

Continued on next page

Table C.1 – continued from previous page

ID	Flow Label	Description
[16]	%SRC_AS	Source BGP AS
[17]	%DST_AS	Destination BGP AS
[21]	%LAST_SWITCHED	SysUptime (msec) of the last flow pkt
[22]	%FIRST_SWITCHED	SysUptime (msec) of the first flow pkt
[23]	%OUT_BYTES	Outgoing flow byte
[24]	%OUT_PKTS	Outgoing flow packets
[27]	%IPV6_SRC_ADDR	IPv6 source address
[28]	%IPV6_DST_ADDR	IPv6 destination address
[29]	%IPV6_SRC_MASK	IPv4 source mask
[30]	%IPV6_DST_MASK	IPv4 destination mask
[32]	%ICMP_TYPE	ICMP Type * 256 + ICMP code
[34]	%SAMPLING_INTERVAL	Sampling rate
[35]	%SAMPLING_ALGORITHM	Sampling type (deterministic/random)
[36]	%FLOW_ACTIVE_TIMEOUT	Activity timeout of flow cache entries
[37]	%FLOW_INACTIVE_TIMEOUT	Inactivity timeout of flow cache entries
[38]	%ENGINE_TYPE	Flow switching engine
[39]	%ENGINE_ID	Id of the flow switching engine
[40]	%TOTAL_BYTES_EXP	Total bytes exported
[41]	%TOTAL_PKTS_EXP	Total flow packets exported
[42]	%TOTAL_FLOWS_EXP	Total number of exported flows
[56]	%IN_SRC_MAC	Source MAC Address
[57]	%OUT_DST_MAC	Destination MAC Address
[58]	%SRC_VLAN	Source VLAN
[59]	%DST_VLAN	Destination VLAN
[60]	%IP_PROTOCOL_VERSION	[4=IPv4][6=IPv6]
[61]	%DIRECTION	[0=ingress][1=egress] flow
[70]	%MPLS_LABEL_1	MPLS label at position 1
[71]	%MPLS_LABEL_2	MPLS label at position 2
[72]	%MPLS_LABEL_3	MPLS label at position 3
[73]	%MPLS_LABEL_4	MPLS label at position 4
[74]	%MPLS_LABEL_5	MPLS label at position 5
[75]	%MPLS_LABEL_6	MPLS label at position 6
[76]	%MPLS_LABEL_7	MPLS label at position 7
[77]	%MPLS_LABEL_8	MPLS label at position 8
[78]	%MPLS_LABEL_9	MPLS label at position 9
[79]	%MPLS_LABEL_10	MPLS label at position 10
[90]	%FRAGMENTED	1=some flow packets are fragmented
[91]	%FINGERPRINT	TCP fingerprint

Continued on next page

Table C.1 – continued from previous page

ID	Flow Label	Description
[92]	%NW_LATENCY_SEC	Network latency (sec)
[93]	%NW_LATENCY_USEC	Network latency (usec)
[94]	%APPL_LATENCY_SEC	Application latency (sec)
[95]	%APPL_LATENCY_USEC	Application latency (sec)
[96]	%IN_PAYLOAD	Initial payload bytes
[97]	%OUT_PAYLOAD	Initial payload bytes
[98]	%ICMP_FLAGS	Cumulative of all flow ICMP types

Bibliography

[1] e. a. J. Quittek, "Requirements for ip flow information export (ipfix)," *RFC3917*, 2004.

[2] (2007) The ipfix charter. [Online]. Available: http://www.ietf.org/html.charters/ipfix-charter.html

[3] (2006) Specification of the ipfix protocol for the exchange of ip traffic flow information. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-ipfix-protocol-24.txt

[4] e. a. R. Stewart, "Stream control transmission protocol," *RFC2960*, 2000.

[5] (2007) The sctp website. [Online]. Available: http://www.sctp.org/

[6] Wikipedia, "Stream control transmission protocol — wikipedia, the free encyclopedia," 2007. [Online]. Available: \url{http://en.wikipedia.org/w/index.php?title=Stream_Control_Transmission_Protocol&oldid=117536755}[Online;accessed11-May-2007]

[7] R. Stewart and C. Metz, "Sctp: New transport protocol for tcp/ip," *IEEE Internet Computing*, vol. 5, no. 6, pp. 64–69, 2001.

[8] R. Rajamani, S. Kumar, and N. Gupta, "Sctp versus tcp: Comparing the performance of transport protocols for web traffic," 2002.

[9] I. Newton, *Philosophiae Naturalis Principia Mathematica*, 1726.

[10] Wikipedia, "Scientific method — wikipedia, the free encyclopedia," 2007. [Online]. Available: \url{http://en.wikipedia.org/w/index.php?title=Scientific_method&oldid=132335924}[Online;accessed21-May-2007]

[11] (2007) The linux website. [Online]. Available: htp://www.linux.org/

[12] (2007) The nprobe website. [Online]. Available: http://www.ntop.org/nProbe.html

[13] (2007) The ntop website. [Online]. Available: http://www.ntop.org/

[14] (2007) The harpoon website. [Online]. Available: http://www.cs.wisc.edu/~jsommers/harpoon/

[15] (2007) The gnuplot website. [Online]. Available: htp://www.gnuplot.info/

[16] (2007) The ntp website. [Online]. Available: htp://www.ntp.org/

[17] (2007) The tcpdump/libpcap website. [Online]. Available: http://www.courier-mta.org/imap/

[18] Wikipedia, "Zeno's paradoxes — wikipedia, the free encyclopedia," 2007. [Online]. Available: \url{http://en.wikipedia.org/w/index.php?title=Zeno%27s_paradoxes&oldid=126911871}[Online;accessed6-May-2007]

[19] (2007) The cisco website - netflow whitepaper. [Online]. Available: http://www.cisco.com/univercd/cc/td/doc/cisintwk/intsolns/netflsol/nfwhite.htm

[20] B. Claise, "Cisco systems netflow services export version 9," *RFC3954*, 2004.

[21] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better netflow," 2004. [Online]. Available: citeseer.ist.psu.edu/article/estan04building.html

[22] (2006) Ipfix aggregation internet-draft 3. [Online]. Available: http://tools.ietf.org/id/draft-dressler-ipfix-aggregation-03.txt