Abstract: This document describes the SCAMPI Architecture and Component Design. It is based on the findings of the deliverable D0.1, “Description and analysis of the state-of-the-art”, on deliverable D0.2 “Measurement-based application requirements” and on deliverable D1.1 “SCAMPI architecture and component design”.

<table>
<thead>
<tr>
<th>Contractual Date of Delivery</th>
<th>31 March 2003</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual Date of Delivery</td>
<td>27 March 2003</td>
</tr>
<tr>
<td>Deliverable Security Class</td>
<td>Internal</td>
</tr>
<tr>
<td>Editor</td>
<td>Evangelos Markatos</td>
</tr>
</tbody>
</table>

The SCAMPI Consortium consists of:

TERENA Coordinator The Netherlands
IMEC Principal Contractor Belgium
FORTH Principal Contractor Greece
LIACS Principal Contractor The Netherlands
NETikos Principal Contractor Italy
Uninett Principal Contractor Norway
CESNET Principal Contractor Czech Republic
FORTHnet Principal Contractor Greece
4Plus Principal Contractor Greece
Siemens Principal Contractor Germany
## Contents

1 Executive summary 9

2 Introduction 11
   2.1 The need for Network Monitoring ................................. 11
   2.2 The Challenges of Network Monitoring ............................ 15
      2.2.1 A Babel of Monitoring Tools and Languages ................. 15
      2.2.2 The network speed challenge ............................... 16
      2.2.3 Addressing the challenges .................................. 18
   2.3 Roadmap .......................................................... 19

3 Network Monitoring Applications 21
   3.1 Packet capture .................................................. 21
   3.2 Intrusion and DoS detection ....................................... 21
   3.3 QoS monitoring ................................................... 22
   3.4 Network flow statistics .......................................... 22

4 Overall System Architecture 23

5 Hardware Architecture 25
   5.1 Special-purpose Network Interfaces ............................... 25
      5.1.1 The DAG card .................................................. 25
      5.1.2 The COMBO6 card .............................................. 26
   5.2 Intelligent Routers ................................................ 30
      5.2.1 Introduction to Juniper Routers ............................... 30
      5.2.2 Why Juniper? .................................................. 31
      5.2.3 Hardware Architecture ....................................... 31
      5.2.4 Software Architecture ....................................... 33
   5.3 IXP1200-based interfaces ......................................... 40
      5.3.1 Introduction to Network Processors ............................ 40
      5.3.2 Network Processors in the SCAMPI project .................... 41
      5.3.3 Using IXP1200 in SCAMPI ..................................... 46
   5.4 Commodity Interfaces ............................................. 48
      5.4.1 Commodity Interfaces in Promiscuous Mode ................... 48
      5.4.2 A Hierarchical Monitoring System based on Commodity Adaptors 49
## 6 Software Structure

6.1 SNMP .................................................. 53
   6.1.1 NET-SNMP ........................................... 54
   6.1.2 SCAMPI MIB ......................................... 54
   6.1.3 Using the SCAMPI MIB ............................... 58
   6.1.4 MAPI management interface ....................... 59

6.2 The Click Approach ................................. 59

## 7 User-level Software

7.1 Translation Tools ......................................... 61
   7.1.1 pcap2MAPI: running legacy pcap applications on MAPI .......................... 61

7.2 Job Control ............................................... 61
   7.2.1 Trust management ..................................... 62
   7.2.2 Operation authorisation .............................. 63
   7.2.3 Custom code loading and compilation ............... 64

7.3 Resource Management and Protection .................. 65

7.4 The Click Approach ....................................... 66
   7.4.1 The Click Model ...................................... 66

7.5 Distributed MAPI .......................................... 66

7.6 MAPI ...................................................... 66
   7.6.1 Passive Monitoring ................................... 66
   7.6.2 Active Monitoring .................................... 67

7.7 MAPI Implementations ..................................... 67
   7.7.1 MAPI on top of pcap/BPF/LSF ......................... 67
   7.7.2 MAPI on top of Click .................................. 67
   7.7.3 Module Creator ........................................ 68
   7.7.4 Module Optimizer ..................................... 69
   7.7.5 Module Mapper ......................................... 69

7.8 Monitoring applications .................................. 70
   7.8.1 Intrusion and DoS detection ....................... 70
   7.8.2 QoS and TE monitoring ............................... 70
   7.8.3 Quality of Service Monitoring Applications ....... 71
   7.8.4 Traffic Engineering Monitoring Applications .... 72
   7.8.5 Threshold Alerting Application .................... 74
   7.8.6 Network debugging ................................... 74
   7.8.7 Flow-based reporting ................................ 75

7.9 Flow record exporter ...................................... 82
   7.9.1 Flow record specification ........................... 82
   7.9.2 Transport Protocols .................................. 83
   7.9.3 Usage .................................................. 83

## 8 Kernel Software

8.1 The Click module ........................................... 85
8.2 The Back-End Modules ..................................... 85
   8.2.1 Commodity Interfaces ................................. 85
   8.2.2 Special-purpose Adapters ............................ 88
CONTENTS

9 MAPI Definition 91
  9.1 Distributed MAPI .................................................. 91
  9.2 MAPI ................................................................. 91
  9.3 Passive Monitoring MAPI ........................................... 93
    9.3.1 Creating and Terminating Network Flows ................. 94
    9.3.2 Reading packets from a flow .............................. 95
    9.3.3 Applying functions to Network Flows .................. 96
    9.3.4 Packet formats ............................................. 97
    9.3.5 Flow Records ............................................... 97
    9.3.6 IPFIX flow record generation ............................ 98
    9.3.7 Some Simple Examples .................................. 102
    9.3.8 Some More Complex Monitoring Examples ............... 105
    9.3.9 Implementation issues for MAPI ......................... 106
    9.3.10 Options for MAPI Interfaces ........................... 108
  9.4 MAPI vs. Other Approaches .................................. 108

10 Summary and Conclusions 111

A Predefined Click Modules 113

B Language for Network Flow Conditions 115

C Predefined Functions for Network Flows 117

D Man pages for MAPI functions 121
  D.1 MAPI functions ................................................. 121
  D.2 MAPI Variables ................................................ 124
  D.3 Variables for Hierarchical Flows .......................... 128

E SCAMPI MIB 131
  E.1 mapi.h ........................................................ 146

F Timestamp issues 149
## List of Figures

2.1 **Real-time monitoring of the traffic on Abilene’s links.** The photograph is courtesy of Abilene. .................................................. 12  
2.2 **The geographic coverage of the code-red worm on July 20, 2000.** The photograph is courtesy of www.caida.org ............................. 13  
2.3 **The response of Microsoft’s servers during a DDoS attack.** The photograph is courtesy of news.com.com ..................................... 14  
2.4 **The geographic coverage of the sapphire worm on January 25th 2003.** The photograph is courtesy of www.caida.org ............... 14  
2.5 **Universities that participate in NLANR’s network trace gathering program.** The photograph is courtesy of pma.nlanr.net ................ 15  
2.6 **Connectivity and speed of GEANT: the European Research Network.** The photograph is courtesy of GEANT ............................ 17  

4.1 **The SCAMPI network monitoring system.** .................................. 24  

5.1 **The DAG card at 10Gbps.** ...................................................... 26  
5.2 **DAG architecture** ................................................................. 27  
5.3 **COMBO6 motherboard.** ....................................................... 28  
5.4 **Functional blocks of COMBO6.** ............................................. 28  
5.5 **Scheme of the COMBO6 card driver.** ...................................... 30  
5.6 **The Juniper Architecture.** ....................................................... 32  
5.7 **The Juniper’s filtering process.** .............................................. 34  
5.8 **The Juniper’s filtering benchmark.** ......................................... 34  
5.9 **IXP1200 Architecture** ........................................................... 42  
5.10 **The Workbench graphical environment.** Provides a common interface to several tools. ....................................................... 44  
5.11 **Use case of IXP1200 in network monitoring.** ............................ 46  
5.12 **SCAMPI monitoring system based on commodity-only network interfaces.** .......................................................... 48  
5.13 **SCAMPI monitoring system based on load balancer and commodity-only network interfaces.** ........................................... 50  

6.1 **The structure of the SCAMPI Software.** ..................................... 52  
6.2 **SCAMPI MIB** ................................................................. 55
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Monitoring Job Representation</td>
<td>69</td>
</tr>
<tr>
<td>7.2</td>
<td>The geographic coverage of the code-red worm. The photograph is courtesy of <a href="http://www.caida.org">www.caida.org</a></td>
<td>70</td>
</tr>
<tr>
<td>7.3</td>
<td>Components</td>
<td>75</td>
</tr>
<tr>
<td>7.4</td>
<td>Database design</td>
<td>76</td>
</tr>
<tr>
<td>7.5</td>
<td>ER diagram</td>
<td>76</td>
</tr>
<tr>
<td>7.6</td>
<td>Report generator</td>
<td>80</td>
</tr>
<tr>
<td>8.1</td>
<td>Example layout of user application, MAPI, BPF and functions.</td>
<td>86</td>
</tr>
<tr>
<td>8.2</td>
<td>Example of cooked network flows.</td>
<td>87</td>
</tr>
<tr>
<td>8.3</td>
<td>Data flow of packets destined to user space.</td>
<td>87</td>
</tr>
<tr>
<td>9.1</td>
<td>Remote MAPI Architecture</td>
<td>92</td>
</tr>
</tbody>
</table>
Chapter 1

Executive summary

This document presents the system architecture and the components of the SCAMPI platform for network monitoring. The need for improved network monitoring arises from the widespread consensus among network operators, engineers and researchers that effective network monitoring is vital for performing informed network management decisions as well as for supporting the growing set of automated control mechanisms needed to make the IP-based Internet more efficient, robust and secure.

The rationale for our work is the increased pressure on existing architectures that has exposed limitations that are believed to be rooted in the basic abstractions used, in addition to increasing link speeds and the growing number and diversity of monitoring applications. As a response, we have designed a new architecture that has the following key characteristics:

- **an expressive programming interface:** we have designed the SCAMPI Monitoring API (MAPI) so that users can clearly express their needs to the underlying platform. In this direction, the main feature of MAPI is a generalized flow abstraction that allows users to tailor measurements to their own needs, recognizing that the information provided by the existing flow model is often either too specific or not specific enough for monitoring applications. Where necessary and feasible, MAPI also allows the user to trigger custom processing routines not only on summarized data but also the packets themselves. The expressiveness of MAPI also allows the underlying monitoring system to make informed decisions in choosing the most efficient implementation.

- **use of intelligent hardware:** SCAMPI provides a coherent interface on top of different lower-level elements; our current design considers the use of intelligent switches, high-performance network processors, and special-purpose network interface cards. Except for high-performance this also reduces the cost of cross-platform compatibility.

- **scalability through parallelism:** SCAMPI acknowledges that scalable network monitoring needs to explicitly honor parallelism and factor it into the architecture design. SCAMPI components exploit parallelism at several levels: from multiple
processing units in hardware (as in the IXP1200 network processor), up to the use of several sensors in hierarchical intrusion detection systems.

This document is structured as follows. We first describe the current need for network monitoring along with the major monitoring applications that are useful in high-speed networks. We then describe the overall architecture of the system, including hardware, kernel-level software, and user-level software. We then provide an extensive description of the SCAMPI Monitoring API (MAPI) that will be used as the interface between user applications and the SCAMPI monitoring system.
Chapter 2

Introduction

2.1 The need for Network Monitoring

With the widespread use and deployment of the Internet, network monitoring has been increasingly used as the major mechanism to improve the performance and security of computing and communication systems. Indeed, the Committee on Research Horizons in Networking of the U.S. National Research Council, in a recent DARPA-sponsored workshop, suggested that one of the three major challenges of Networking Research in the next few years is the measurement and monitoring of the Internet [26]. Actually, the Committee stated that the network community should strive to record “a day in the life of the Internet”, that is, to “develop and deploy the technology to make it possible to gather a data set containing the complete traffic, topology, and state across the Internet Infrastructure”. Obviously, the Committee realized that the only way to improve any system (including the Internet) is to understand it through detailed monitoring and observation.

SCAMPI represents a step towards this goal by building affordable high-performance network monitoring systems. Besides understanding and improving the Internet as stated in [26], network monitoring systems in general, and SCAMPI in particular, have several other goals as well including:

- to help Internet Service Providers and Application service Providers in giving better service to their customers through improved billing mechanisms.
- to improve the end-user experience through better network performance achieved in part by informed network management and traffic engineering methods
- to enhance the security of computer systems connected to the Internet by providing a better defense against cyberattacks, such as Denial-of-Service attacks and Intrusion attempts.

Each of the above goals of SCAMPI is particularly important for a different set of users and organizations, including Internet Service Providers, Applications Service
CHAPTER 2. INTRODUCTION

Figure 2.1: **Real-time monitoring of the traffic on Abilene’s links.** The photograph is courtesy of Abilene.

Providers, Corporations with computers connected to the Internet, and even, home-based users. For example, Internet Service Providers, are interested in monitoring the amount of traffic in their systems in order to provide their customers with the best possible service. Internet Service Providers are also interested in monitoring the traffic patterns on their networks in order to be able to make the best use of their infrastructure by redirecting traffic through idle communication lines and balancing the communication load whenever possible. For example, in Figure 2.1 we see that the traffic in Abilene’s network is unevenly distributed on the horizontal links across the United States. Actually, the horizontal link that crosses the middle States is about 50% loaded, while at the same time the horizontal link that crosses the south States is less than 5% loaded. Frequent monitoring and traffic balancing would lead to better network link utilization and improved overall end-user experience. Besides providing the best possible service to their customers, ISPs and ASPs are frequently bound by agreements (usually called Service Level Agreements) to provide a minimum level of service to their customers. Thus, they need to monitor their network in order to make sure (and to prove if needed) that they do provide levels of service required by the signed contracts.

Besides ISPs and ASPs, most organizations with an Internet presence are interested in monitoring their networks against cyberattacks. There are two major types of attacks that can be launched against an organization connected to the Internet: Distributed Denial-of-Service (DDoS) attacks, and Intrusion attempts. In Intrusion attempts, the intruder tries to exploit a known vulnerability or bug of the victim system by sending

---

1 Abilene is an advanced backbone network that connects regional network aggregation points, called gigaPoPs, to support the work of Internet2 Universities backbone network.

2 To relieve this situation, modern network protocols like MPLS [35] may perform traffic rerouting on-the-fly, depending on traffic problems.
2.1. THE NEED FOR NETWORK MONITORING

Figure 2.2: The geographic coverage of the code-red worm on July 20, 2000. The photograph is courtesy of www.caida.org.

to the victim a carefully crafted stream of packets. These packets are received by an application running on the victim computer and force the application to execute code which will trigger a bug that will crash or compromise the system. Intrusion attempts started very early in the life of the Internet [36] and are currently widespread. Indeed, Figure 2.2 shows the geographical distribution of the computers that were infected by an intrusion in the summer of 2001: the code-red worm that infected personal computers running Microsoft’s Internet Information server. We see that the worm spread and covered almost every part of the globe that had computers connected to the Internet. Contrary to intrusion attempts, that launch the intrusion through a small stream of packets to each victim computer, DDoS attacks send a flood of fake packets to the victim computer trying to overload it or crash it. In both cases, the victim computer will not be able to respond to legitimate requests. In this way, the victim computer at best appears slow, or even completely unreachable. For example, Figure 2.3 shows the performance of Microsoft’s web servers at a two-day period where they were subjected to a Denial of Service attack. We see that although the servers were able to serve about 100% of their requests during normal operation, during the DDoS attack interval, they were able to serve no more than 20% of the input requests.

Even though the code-red worm managed to cripple around half a million computers, and one would expect that computer users would be alert for such events, recently, a similar worm (the sapphire worm) managed to gain similar geographic spread as can be seen from Figure 2.4. Indeed, the worm infected close to 75,000 computers running SQL servers in less than 30 minutes. The worm was able to spread at exponential rates, doubling the number of computers it infected every 8.5 seconds or so. Clearly, our cyberinfrastructure faces a serious threat, which is larger than ever before. To respond to this threat DARPA has recently announced a program for the “dynamic quarantine of computer worm-based attacks against military enterprise networks” \(^3\). The goal of

\(^3\)http://www.eps.gov/spg/ODA/DARPA/CMO/BAA03-18/Attachments.html

scampi@ist-scampi.org

March 27th, 2003
CHAPTER 2. INTRODUCTION

Figure 2.3: The response of Microsoft’s servers during a DDoS attack. The photograph is courtesy of news.com.com.

Figure 2.4: The geographic coverage of the sapphire worm on January 25th 2003. The photograph is courtesy of www.caida.org
2.2. THE CHALLENGES OF NETWORK MONITORING

Figure 2.5: **Universities that participate in NLANR’s network trace gathering program.** The photograph is courtesy of pma.nlanr.net.

The program is to conduct research that will “develop devices capable of detecting and containing worm infections to 1 percent of vulnerable machines, with a false alarm rate of less than or equal to 1 per day, while minimizing the time to recovery for mission-critical applications running on a test-bed network infected with a worm will be less than or equal to 6 minutes after infection.” These worm-detection devices need to be able to do detailed network monitoring at wire speed.

Besides ISPs, ASPs, Organizations with Internet points of presence, and the military, scientists from Research Institutions and Universities are also interested in monitoring the network traffic in order to gather data that will help them enhance their understanding of the current state of the Internet and will boost their research towards designing next generation systems and networks. Indeed in the United States, more than 20 universities participate in a passive monitoring program coordinated by NLANR (as shown in Figure 2.5). Within this program they capture, store, and subsequently process network packets captured from their network probes. One of the results of the NLANR studies that was widely publicized by the media \(^4\) was the discovery that each week more than 4,000 Denial-of-Service attacks occur throughout the world.

### 2.2 The Challenges of Network Monitoring

#### 2.2.1 A Babel of Monitoring Tools and Languages

From the above examples, it can be easily seen that network monitoring is needed by several different kinds of users that would like to monitor different aspects of the

---

network traffic. For example, some of the users may be interested in viewing aggregate traffic statistics only, while others may be interested in monitoring (and maybe storing on persistent storage) each and every byte that travels through their network (e.g. in order to detect cyberattacks). Currently, state-of-the-art tools specialize in solving only one subclass of network monitoring problems. For example, there exist several traffic monitoring tools currently being used by ISPs and ASPs that monitor the status of their networks in order to alert administrators of possible malfunctions. These tools can also monitor the amount of traffic on the various segments of the network in order to inform administrators of the network usage and enable them to make informed traffic engineering decisions. Some of those tools are sophisticated enough to work both as traffic generators and as traffic analyzers at the same time in order to detect possible network problems.

Besides the above mentioned tools, there exist monitoring environments specialized in discovering Intrusion attempts. These environments, also known as Intrusion Detection Systems (IDSs) [1] examine every packet they observe on the network, trying to detect one of the known intrusion threats. Similar to IDSs, Denial-of-Service attack detection tools monitor the network traffic in order to detect generalized network attacks. Denial-of-Service attack detection systems are frequently integrated into firewalls that examine and classify packet headers in order to detect any abnormal increases the network packets, which may indicate the start of an attack.

Finally, besides the traffic-engineering-related and the security-related monitoring tools, there exists a variety of traffic capture tools that focus on capturing (and maybe storing) all network packets (possibly along with their payloads) in real-time. Such captured packets are subsequently being used by network researchers to drive their own research.

Currently, most of the above monitoring tools and environments are based on different sets of primitives and functions. For example, traffic monitoring tools are based on the primitives provided by network routers such as NetFlow [38]. On the other hand, traffic capture tools, such as those used by NLANR, are based on top of a custom-made hardware-software infrastructure, collectively known as OC3MON [13]. Intrusion detection systems, such as snort [34], have been implemented on top of the libpcap [20] packet capture library, while Denial-of-Service detection systems are being implemented on top of firewalls, which on Linux-based systems are usually implemented on top of netfilter and iptables. To make matters worse, commercial vendors frequently use their own libraries and standards, contributing even more to the Babel of network monitoring tools and environments, making it even more difficult to write portable network monitoring applications.

2.2.2 The network speed challenge

Computer networks continue to get faster at ever-increasing rates. For example, published results suggest that network bandwidth increases at alarming rates doubling every 9-18 months or so [3], an observation which is usually referred to as “Gilder’s Law”. At the time of this writing the Internet backbone links of the European research network GEANT run at 2.5 to 10 Gbps (as shown in Figure 2.6). As if this exponential increase were not enough, network monitoring applications continue to become more
2.2. THE CHALLENGES OF NETWORK MONITORING

Figure 2.6: Connectivity and speed of GEANT: the European Research Network. The photograph is courtesy of GEANT.
complex and more demanding. For example, the first network monitoring applications required very little information from the monitored network, such as aggregated traffic statistics or aggregated flow statistics. On the contrary, recent monitoring applications demand a significant amount of information, that includes both header and payload data for each and every packet of the network. To make matters worse, monitoring applications add an ever-increasing amount of processing on the captured data, such as compute-intensive string matching used for the detection of Internet worms and various other forms of cyberattacks.

2.2.3 Addressing the challenges

Having realized the challenges posed by the state-of-the-art tools in network monitoring, SCAMPI represents a bold step towards building an affordable network monitoring system for high-speed networks that will enable the development of portable applications.

SCAMPI employs a three-pronged approach in order to achieve its goals:

- **Standard Monitoring API (MAPI).** SCAMPI will define a set of monitoring calls/primitives that will be collectively called as MAPI. Monitoring applications will be written using this MAPI. The MAPI will be implemented on top of several platforms, decoupling the development of the monitoring applications from the monitoring environment on top of which the applications will be executed. Thus, monitoring applications will be written once, and will be able to run on top of any monitoring environment without needing to be re-written.

- **Expressive power.** Current monitoring application programming interfaces provide little (if any) expressive power to application programmers. Thus, application programmers are not able to communicate their monitoring requirements to the underlying network monitoring system. As a result, frustrated application programmers end up receiving all network packets in the address space of their application where they perform the operations they need. As a simple example of the poor expressive power of current network monitoring systems, consider a user that wants to sample one out of every 100 packets in order to find the most popular applications that use his/her network. Current network monitoring systems (like libpcap [20], Berkeley Packet Filters [21], and Linux Socket Filters [12]) do not enable users to express such simple sampling requirements. Therefore, users that are interested in receiving just one out of every 100 packets are required to read all packets, and just discard 99 out of every 100 of them. To overcome these limitations, SCAMPI’s MAPI will enable monitoring application programmers to express their requirements to the underlying monitoring system, which in turn will decide how will these requirements will be more efficiently implemented.

- **Scalability through special-purpose hardware and parallelism.** Although network monitoring can be performed on top of traditional network adapters, SCAMPI, wherever possible, will exploit specialized network adapters that provide some monitoring functionalities in hardware [13, 22] These adapters contain on-board processors and FPGAs that can be programmed to perform monitoring
functions and off-load the host processor, memory system, and I/O buss from much of their load.

2.3 Roadmap

The rest of the deliverable is organized as follows: Chapter 3 outlines the network monitoring applications SCAMPI will support. Chapter 4 presents the overall software and hardware structure of the SCAMPI monitoring system. Chapter 5 presents the alternatives for the hardware infrastructure ranging from special-purpose interfaces, to commodity interfaces, to intelligent routers. Chapter 6 presents the overall software structure of the system. Chapter 7 focuses on software that resides in user-level and Chapter 8 focuses on kernel-level software. Chapter 9 presents the API that monitoring applications will use and finally, Chapter 10 summarizes and concludes the deliverable. Specific details about components of the SCAMPI monitoring system are contained in the appendices. Appendix A presents the predefined modules of the Click environment. Appendices B, C, and D present a detailed treatment of various API functions. Finally, appendix E presents the MIB for the SCAMPI system, and appendix F presents requirements for the accuracy of the timestamps.
Chapter 3

Network Monitoring Applications

In this section we briefly describe network monitoring applications, identifying their particular characteristics and explaining how the SCAMPI platform can satisfy them.

3.1 Packet capture

Packet capture involves the collection of packet-level measurements. Such measurements can include the packet arrival time, size, header, and full payload. Measurements can refer to packets belonging to individual flows or packets belonging to aggregate flows. Capturing detailed measurements for all packets flowing through a high speed network link is the most demanding monitoring application, hence would stress the limits of the SCAMPI architecture.

On top of the packet capture functionality, we can have applications for network debugging. Such applications would typically need detailed analysis of each packet header, or even the packet payload; the latter is required in the case higher level or tunneled protocols are being debugged.

3.2 Intrusion and DoS detection

Intrusion detection is the problem of identifying individuals who are trying to use a computer system’s resources without authorization. An intrusion attempt can start with a port scan of the targeted host. This port scan can be detected by simply analyzing flow records containing the destination ports scanned by the intruder. However, to obtain detailed information about the intrusion attempt itself, such as who initiated it and what its target was, it would be necessary to analyze each packet separately, possibly also examining the contents of packets to search for certain substrings.

Detecting Denial of Service (DoS) attacks can be done by analyzing aggregate statistics contained in flow records. If the total number of flows to individual or an
aggregate number of hosts are available, DoS attacks can be detected when some statistic exhibits abrupt and significant changes. When a host has been identified as being the subject of a DoS attack, individual flow records can be studied in more detail and a defense strategy can be derived.

3.3 QoS monitoring

Quality of Service (QoS) monitoring is the passive observation of transmission quality in the network. QoS monitoring can also serve for validating SLAs (Service Level Agreements). Often this kind of monitoring requires multiple observation points in the network, and synchronization of the clock is important. Some common QoS parameters that are often monitored include the following:

- Round-trip time: requires packet matching techniques to match requests and responses so that the RTT time can be calculated.
- One-way delay: requires collection of monitoring data at two observation points.
- Packet loss: can be done with either two observations points or by one observation point where sequence numbers in the packets are analyzed.
- Delay variation: this is defined as the variation of the one-way-delay and is calculated based on one-way delay measurements.

3.4 Network flow statistics

Monitoring of network flow statistics can be used for traffic profiling and network accounting.

Traffic profiling characterizes IP flows according to key parameters such as duration, volume, time, and burstiness. It is heavily used for network planning and dimensioning, and trend analysis. Traffic profiling is also used by traffic engineering which is concerned with performance optimization of operational networks. Traffic profiling typically involves flow records, aggregate statistics, and packet headers.

Accounting typically involves measuring the amount of transferred traffic on a network. For some uses, only the total traffic is counted while others might divide the traffic into categories according to type of traffic, time of day, etc. This means that the underlying platform that collects the information must have the capabilities of differentiating between types of traffic and be configurable as to how this is done.
Chapter 4

Overall System Architecture

The overall system architecture of the SCAMPI network monitoring system is shown in Figure 4.1. The system will be composed of a regular PC (personal computer) coupled with a Hardware Monitor connected to the system’s I/O (e.g. PCI) bus. We envision that different instantiations of the SCAMPI system will probably require different hardware monitors. Low-end monitoring systems will probably use a regular network interface as a hardware monitor. Most network interfaces can be put in a special promiscuous mode in which they receive and propagate to their operating system all the packets that pass though their network interface, independently of whether the packets were destined to this network interface.

High-end systems may employ a special-purpose adaptor, that has the computing capacity to not only capture packets, but also to process them and perform some simple (but fast) monitoring functionalities. Endace\(^1\), for example, a New-Zealand-based company, has a complete series of such adapters for network monitoring at 1Gbps\(^2\), 2.5 Gbps, and 10 Gbps\(^2\).

Other systems may employ various other forms of commodity monitoring hardware. For example, Juniper routers\(^3\) allow users to filter the traffic that passes through the routers based on header fields. The filtered traffic, in turn, can be mirrored to a router’s port, on which we may connect a PC that has its interface in promiscuous mode. Thus, Juniper routers can be used to make an effective first-pass monitor, and delegate the rest of functions to a general-purpose computer (e.g. a PC).\(^3\)

The SCAMPI monitoring system may either be isolated, or connected to the Internet. An isolated passive monitoring system, although it can receive Internet traffic, it does not have an IP address, it does not reply to requests for connections, and it does not send any packets to the Internet. The only way to interact with such an isolated system is through its console. Isolated monitoring systems provide excellent security, because

---

\(^1\)http://www.endace.com

\(^2\)A more accurate way of characterizing the capabilities of network monitoring devices would be to express them in packets per second, instead of Gigabits per second. Since the first performance metric (packets per second) may not always be available, we will report it only when possible.

\(^3\)We should point out that in the case where we use Juniper switches, we do not need the splitter shown in Figure 4.1.
they can not be compromised through network attacks and intrusions. On the other hand, they have a limited amount of flexibility, since remote users are not able to directly download and execute monitoring applications. On the other hand, to avoid total isolation, SCAMPI can be (briefly) connected to the Internet in order to allow users interact directly with it.

**SCAMPI Subsystems**

The SCAMPI system will be composed of the following three subsystems, which will be further explained in subsequent chapters:

- The SCAMPI monitoring hardware
- The SCAMPI kernel-level software
- The SCAMPI user-level software
Chapter 5

Hardware Architecture

5.1 Special-purpose Network Interfaces

5.1.1 The DAG card

The DAG cards from Endace Measurement Systems are high performance PCI based passive monitoring cards capable of capturing whole packets or a user defined portion of packets. They are available in various models ranging from speeds of 10Mbit/s and up to 10Gbit/s. A GPS receiver can be connected directly to the cards and a high precision 64-bit timestamp is added to each captured packet.

The SCAMPI consortium currently has two single-interface OC3 DAG cards and two dual interface GigE DAG cards for use in the project.

Figure 5.1 shows a picture and figure 5.2 shows the general architecture of a DAG card. Most of the functionality of the cards is in an FPGA that can easily be reconfigured. Currently the cards can only be used for packet capture, however the design of the cards allows them to be used as packet generator as well with an appropriate FPGA design.

When a packet is captured it is processed by the FPGA where a DAG header is prepended to it. This header contains various information including the high precision time stamp. The packet is then transferred directly to a large circular buffer in the host system. If the PCI bus is busy the packet can be stored in a small FIFO. If the PCI bus is too congested or the memory buffer is full the packet will be dropped and a loss counter will be incremented. The next packet that can be written to the buffer will have this loss counter in the DAG header so that user applications can detect packet loss. The circular memory buffer in the host system is created at boot time by the DAG driver.

Users can read packets from the DAG cards through the DAG API. The main function for reading packets is `dag_offset` which is a blocking function that returns a pointer to the last packet written to the circular buffer by the DAG card. There is also an inline function, `dag_nextpkt`, that can be used to process individual packets in the buffer. The technique used by DAG is very efficient as it is a zero-copy memory-mapped method for users to access captured packets.

For DAG cards with multiple interfaces, packets from all interfaces are copied to the same memory buffer. To distinguish between the interfaces, each packet has a flag
Figure 5.1: The DAG card at 10Gbps.

in the DAG header indicating which interface the packet was captured by.

**Timestamp format**  The 32 most significant bits of the DAG timestamp are the same as a Unix time and is set from the host’s system clock. The least significant 32 bits represents the fractional part of the times-tamp in the specified second. This gives a resolution of $2^{-32} = 0.23\text{ns}$ and time between Jan 1 1970 and Jan 19 2038 can be represented. The current DAG cards only have a precision of around 100ns so the higher resolution of the timestamp is meant for forward compatibility. The DAG cards also have an RS422 interface where a GPS receiver can be attached. This allows for a synchronization to UTC with an accuracy of 100ns. It is also possible to synchronize two DAG cards by using a normal Ethernet crossover cable.

### 5.1.2 The COMBO6 card

Card COMBO6 (COmmunication Multiport BOard for IPv6) has been developed as activity of 6NET project (IST-2001-32603). The card is aimed as hardware accelerator for IPv6 routing, however for its modular design and high flexibility (FPGA technology) can be simply adopted for network monitoring.

The design of the card is declared as an open source project.

The COMBO6 motherboard consists of following blocks:

- FPGA VIRTEX II (produced by Xilinx)
- memory modules SRAM, CAM and DDRAM
- PCI bus interface
- daughter card interface

---

1. [http://www.liberouter.org](http://www.liberouter.org)

scampi@ist-scampi.org  
26  
March 27th, 2003
5.1. SPECIAL-PURPOSE NETWORK INTERFACES

All link interfaces are located at daughter card. There are two daughter cards under development now. The first one has 4 metallic ports (10/100/1000), while the second one has 4 optical ports (1Gbps) in SFP modules which allow to use wide spectrum of optical transceivers (multimodal, monomodal CWDM, DWDM).

The designed firmware for packet switching in the COMBO6 card (shown in Figure 5.4) includes the following functional blocks:

- input packet buffer memory (IPB)
- L2 and L3 header field extractor (HFE)
- look-up processor (LUP)
- packet replicator and block of output queues (RQU)
- output packet editor (OPE)
- output packet buffer (OPB)
- PCI interface (PCI)
- dynamic memory controller (DRAM)

The driver of the COMBO6 card (which is now being developed for NetBSD) is designed so that it appears to the operating system as a standard multi-port communications card. Block diagram of the driver is shown in Figure 5.5.
CHAPTER 5. HARDWARE ARCHITECTURE

Figure 5.3: COMBO6 motherboard.

Figure 5.4: Functional blocks of COMBO6.
Using COMBO6 as SCAMPI adapter

COMBO6 card is considered to be adopted as a base of SCAMPI hardware monitoring adapter for speeds up to 10 Gbps.

Hardware issues

Significant portion of work is already done, as the hardware of SCAMPI adapter is based on universal COMBO6 card. However it will be necessary to design new daughter card with following features:

- Clock: there should be designed precision clock circuit with frequency 100 MHz, which will provide timestamps with resolution 10 ns. We need such fine resolution to assign unique timestamp to every packet. We have to achieve high accuracy of clock, too. It implies stable, temperature compensated oscillator (TCXO) with external synchronization. We assume to use GPS receiver as a reference source of exact time but adapter must not be dependent on GPS. Alternative supported method of adapter clock synchronization is usage of NTP server.

- Network interface: the adapter based on COMBO6 card is suitable for monitoring and measurement at Ethernet and POS (Packet over SONET) lines from 100 Mbps up to 10 Gbps. The aim is to develop daughter cards with Ethernet link modules 1 Gbps and 10 Gbps in scope of the SCAMPI project. The first prototype will have 1 Gbps link module. This is the highest speed suitable for testing and debugging, as it is not complicated to generate such data flow. In further stage we will focus on 10 Gbps interface.

Firmware issues

The original firmware is designed for routing and not for monitoring. However, as the firmware consists of modules we can use some of them:

- PCI bus interface
- memory management
- link module interface
- layer 2 header parsing
- IP header parsing
- TCP/UDP header parsing
- packet filtering

Other modules have to be developed:
5.2 Intelligent Routers

5.2.1 Introduction to Juniper Routers

Aim of this section is to briefly describe the capabilities of Juniper routers and in particular how they can be useful in network monitoring.

In order to keep up with Gb speeds, network manufactures had to replace their CPU-centric architecture to new ones based ones still based on CPUs that take the advantage of specialized chips (ASIC) capable to capture and manage network packets. Therefore the architecture of modern network equipment is divided in two parts:

- Packet handling: performed by ASIC chips.
- Router monitoring, configuration, and management: performed by a CPU-based embedded PC (usually based on x86 CPUs) running a Unix-based OS, that instrument what the ASIC chips have to do.
5.2. INTELLIGENT ROUTERS

This new architecture is characterized by:

- High efficiency as packets never reach the CPU/OS (unless the user specifies this), network equipment can easily operate at Gb speeds with no performance loss.

- Ability to filter (with a basically unlimited number of ACLs per port), discard, modify packets on the fly.

- On the fly traffic conversion across different media (e.g. IP-over-Ethernet to/from IP-over-ATM).

- Traffic mirror either fully or sampled of full packets or headers only.

- Ability to run standard applications (e.g. packet sniffers, performance tools, network probes) on the OS.

5.2.2 Why Juniper?

Juniper routers sport this new architecture, and have been used and preferred by NETIKOS with respect to other vendors because:

- Price: an entry level router with 4 ports costs as little as 15,000 Euro.

- OS: The JunOS (Juniper Operating System) is based on FreeBSD and it allows us to easily run on the router applications freely available on the Internet.

- Traffic filtering and mirroring: traffic can be filtered and mirrored in a very flexible way.

- Configuration: Juniper’s CLI is very simple and well structured.

- Management: routers can be managed from remote using SNMP or JunoScript, an XML-based configuration language.

- Monitoring: JunOS allows network administrators to define complex traffic filtering expressions for the purpose of traffic control and accounting (SNMP can be used to read counters associated to the specified filters) that are handled directly by ASICs hence able to cope with Gb speeds.

5.2.3 Hardware Architecture

Juniper’s architecture consists of two main components:

1. Packet Forwarding Engine (PFE) that includes all the hardware responsible of the “real” packet forwarding.

2. Routing Engine (RE) that includes all the hardware and software parts responsible of the routing protocol management and the router management at all.
In order to maximise both throughput and router performance, both the PFE and RE don’t share any memory or CPU. As shown in the figure 5.6, for a better analysis, PFE could be further divided into:

- Forwarding Engine Board (FEB) with its own ASICs;
- Physical Interface Card (PIC)

The use of ASICs allows Juniper routers to satisfy many SCAMPI platform requirements. This is because ASICs sport several interesting features including, but not limited to:

- Packet forwarding
- Route lookup
- Filtering
- Sampling
- Rate limiting
- Load Balancing
- Switching
- Buffer Management
- Encapsulation/De-encapsulation
All the channels among ASICs are actually dedicated and oversized in order to avoid problems during the packet forwarding process (the main router purpose).

The routing engine “contains” all processes of the router operating system JunOS and it takes care of general router configuration and management access, alarms notification and the exchanging of the Forwarding Table with the PFE.

The list below contains some of the main router features:

- **FEB**
  - M5 throughput of 6.4 Gbps half duplex
  - M10 throughput of 12.8 Gbps half duplex
  - 266-Mhz CPU and supporting logic
  - Internet Processor II ASIC (40Mpps forwarding performance)
  - Two Distributed Buffer Manager ASICs for coordinating pooled, single-stage buffering
  - One (M5 router) or two (M10 router) I/O Manager ASICs for wire-rate parsing, prioritizing and queuing of packets.
  - Four banks of 2-Mb SRAM for forwarding tables associated with ASICs
  - 64-Mb DRAM storage for microkernel
  - Two 512-Kb boot flash EPROM (programmable on the board)

- **Routing Engine**
  - 333-Mhz Pentium II with integrated 256-Kb level 2 cache
  - 256-Mb/768-Mb of SDRAM memory
  - 80-Mb compact flash drive for primary stage
  - 6.4-Gb-IDE hard drive for secondary stage
  - 110-Mb flash PC card for tertiary storage
  - 10/100Base-T autosensing RJ-45 Ethernet port for out-of-band management
  - Two RS-232 (DB9 connector) serial asynchronous ports for console and remote management

### 5.2.4 Software Architecture

The JunOS (Juniper Operating System) has been designed for satisfying requirements of ISPs and large communities such as universities or large companies. JunOS has been developed with modular concepts that means software in divided in several processes running in separated and protected memory areas to provide fault tolerance (a process fault is not propagated to all the processes of the router operating system).

As JunOS shares the architecture of the UNIX BSD operating system, where processes run independently on protected memory, operations such as upgrade or restart...
CHAPTER 5. HARDWARE ARCHITECTURE

Figure 5.7: The Juniper’s filtering process.

Figure 5.8: The Juniper’s filtering benchmark.
of a software module does not require a router reboot (as it happens on other vendor’s OSs) but just a process reboot.

The routing engine “contains” all processes of the router operating system JunOS and it takes care of general

**Advanced Packet Filtering: IP Service Delivery**

Juniper routers support IP service delivery that can be configured using a simple and flexible syntax. A service is identified by one or more conditions (e.g. source-port, dest-port, etc) and one or more actions to perform on the packets (as an if-then-else structure). Users can apply complex filters by creating easier ones and applying them on the same network traffic; for each filter (or filter term) users could define a counter incremented every time the filter is applied; these counters could be read by SNMP requests (see section 5.2).

As discussed above, the ASIC architecture allows the Juniper Routers to maintain high performances even in case of many filters defined and applied: the figure below describes this feature.

The large variety of filters could be created permits to perform security tasks because all the traffic monitored could be summarized in counters and, analyzing these ones with all the available tools, user could limit DoS, perform Source Address Verification (to prevent spoofing), and perform traffic accounting. The packet filter can classify packets based on any of the fields that can be examined by the JUNOS Internet software filter definition language. These fields include:

- Source and/or destination IP addresses
- Protocol number
- Source and/or destination port numbers
- IP precedence value
- DSCP value
- IP options
- TCP flags
- Packet length
- ICMP type
- Incoming and/or outgoing logical or physical interface

If a packet satisfies the conditions of the filter, you can specify the filter action known as a routing-instance. This filter action allows you to specify the routing table instance that is used to forward traffic that matches the filter’s conditions. Once the routing table is identified, traditional destination-based routing occurs. In addition to the routing-instance action, you can also specify the following action modifiers in the filter.
Filter rules are specified as follows:

```plaintext
[edit]
firewall {
    family family-name {
        filter filter-name {
            accounting-profile name
            interface-specific
            term term-name {
                from {
                    match-conditions ;
                }
                then {
                    action ;
                    action-modifiers ;
                }
            }
        }
    }
}
```

The following match conditions can be specified:

- destination-port number
- dscp number
- fragment-offset number
- icmp-code number
- icmp-type number
- interface-group group-number
- packet-length bytes
The following example blocks telnet and secure shell (ssh) access to all but the 192.168.1.0/24 subnet. This filter also logs any ssh or telnet traffic attempts from other subnets to the firewall log buffer:

```
scampi@ist-scampi.org 37 March 27th, 2003
```
Sampling - Network Traffic Analysis

One of the actions could be performed by Juniper Router is packet sampling. While performing its activities, the router can sample packets according to specified filters and perform actions on them that include:

1. send packets on a specified interface (traffic mirror);

2. count packets according to some specified filters;

3. log packets on disk.

For better traffic analysis the router can “copy” the selected - sampled (by the filter) packets and send them to an analysis console; this feature is called Port Mirroring and it’s one of the possible actions could be done with the filters.
Instead of forward all the traffic sampled, if the analysis console isn’t able to collect all the data, the traffic could be exported using the NetFlow format that can be collected using tools such as eflowd and ntop.

In term of traffic analysis and traffic management, in order to satisfy SCAMPI architecture accounting features, Juniper Router can perform Traffic Rate Limiting in order to identify/apply billing policies, bandwidth consumption.

As stated above, packets can be counted according to a set of specified filter terms. For each packet received matching a filter with a count clause, the router increments a counter. For instance assume the following filter that counts fragmented ICMP traffic:

```plaintext
term anti-fragment {
    from {
        is-fragment;
        protocol icmp;
    }
    then {
        count FragmentAttack;
        syslog;
        discard;
    }
}
```

For each fragmented ICMP packet, the FragmentAttack counter is incremented. Counter values can be accessed by means of:

- **CLI**
  ```plaintext
  > show firewall
  Filter: <filter name>
  FragmentAttack 41822 packets 41645450 bytes
  >
  ```

- **SNMP**
  ```plaintext
  ```

- **JunoScript**
  ```plaintext
  <rpc-reply xmlns:junos="http://xml.juniper.net/junos/5.3R2/junos">
  <output>
  Filter: <filter name>
  FragmentAttack 41822 packets 41645450 bytes
  </output>
  </rpc-reply>
  ```

scampi@ist-scampi.org 39 March 27th, 2003
Security Services - DOS-Attack monitoring/mitigation

A particular use of the filter terms could mitigate the DoS attacks. This is because all the main types of Dos attacks involve ICMP packet than applying several filter to log all the ICMP traffic and monitoring its counter should be possible to identify/prevent attacks (if the counter increases too fast). Other DoS attacks, involving the IP fragment (Fraggle, Boink, Teardrop), could be detected and logged by the Router. In almost cases, if correctly analysed and applying the right filters, the router could log all the DoS attacks. And none of these features involve the main purpose, the packet forwarding.

For instance suppose to handle smurf attacks using a Juniper router by limiting the number of ICMP packets that are/admitted into a network at all the edges, thereby diminishing the effect of a smurf attack. This is because limiting all peers to sending ICMP packets, a smurf attack may temporarily devastate its target, but the network resources, including bandwidth, would not be nearly as stressed.

The following code fragment specifies how to limit ICMP packet to 200 Kb.

```plaintext
[edit firewall]
filter limit-icmp {
    policer pl{
        if-exceeding {
            bandwidth-limit 200k;
            burst-size-limit 20k;
        }
        then {
            discard;
        }
    }
    term one {
        from {
            protocol icmp;
        }
        then {
            policer limit-icmp;
            accept;
            count count-icmp;
        }
    }
}
```

5.3 IXP1200-based interfaces

5.3.1 Introduction to Network Processors

Network processors bridge the gap between ASIC-based and General Purpose Processor based solutions in packet processing systems [8]. ASIC-based packet processing is ex-
tremely fast but it is rather expensive to develop and has a rather fixed functionality. On
the other hand, General Purpose Processors, by being programmable, offer significant
flexibility but, due to their general-purpose nature, they may sacrifice performance.

Network processors attempt to fill the gap between special-purpose ASIC modules
and general-purpose processors, by offering adequate flexibility accompanied with
adequate performance to operate at wire speeds. This combination is achieved by
exploiting packet processing parallelism.

Commonly, packet processing parallelism is supported by redundancy of processing
resources. For example, the IXP1200 Network Processor provides six simple processors
(usually called micro-engines) that can operate independently one from another [9].
On-chip integration of these as well as other units (small memories, other specialized
hardware) make an NP fast enough to operate at link speeds.

In the context of network monitoring, network processors can be used so as to
perform advanced processing near the wire and therefore significantly off loading a
hosting system. They can be used in numerous ways including:

- **filtering**: first level filtering of packets in order to isolate the link speed from a
  monitoring application.

- **sampling**: selectively passing packets based on sampling functions.

- **load balancing**: splitting traffic among monitoring systems.

- **primitive function implementations**: implementation of primitives that when
  used in various combinations they compose complex functions.

### 5.3.2 Network Processors in the SCAMPI project

Network processors (NPs) allow for full programmability of the network card. They
provide processing at line-speed as close to the hardware as possible. Prime examples
are Intel’s IXP/IXA family and IBM’s PowerNP family. The SCAMPI consortium
currently has about a dozen of the former for use in the project. Packet handling
is commonly done by multiple parallel processing elements (PEs) under control of
a general-purpose control processors. Other hardware-assisted processing functions
commonly found in modern NPs include features like hashing functions. Due to
extensive pipelining (both functional and context-based), the hardware is able to keep up
with network speeds while also allowing for extensive programmability. NPs may also
include such aspects of packet processing as buffering hardware, enqueuing, dequeuing,
frame forwarding, scheduling, and chip egress (such as PCI). Programming NPs is
fairly complex and by necessity supported by a variety of tools. Commonly provided
development tools include debuggers, assemblers/compilers, and simulators. PEs are
programmed using special low-level (assembly-like) instructions. Such code is called
microcode in Intel IXPs.

The most widely-used network processor in academic environment is the Intel
IXP1200, shown in Figure 5.9. For this reason and because a number of IXP1200
evaluation boards are currently available to SCAMPI, we will focus on this architecture
in the remainder of this section.
CHAPTER 5. HARDWARE ARCHITECTURE

Figure 5.9: IXP1200 Architecture

The figure illustrates explicitly the various processing levels ($L_0 - L_4$) that exist in a system containing an IXP1200 NP. Both on the host CPU and on the control processor (StrongARM), we may benefit from general-purpose operating system support (e.g. Linux), but the microengines do not run operating systems at all. The IXP1200 processor has the following features:

- One general-purpose StrongARM control processor.
- Six special-purpose 200 MHz multithreaded microengines (4 threads each). Each microengine is associated with a 1KB instruction store which is loaded by the StrongARM.
- A pair of FIFOs (16 slots x 64 bytes each) to transfer packets top and from the network.
- 4 KB of on-chip scratch memory.
- Integrated memory controllers for SRAM and SDRAM.
- PCI bus interface.
- 4-6 Gbps IX databus to connect to MACs.

The IXP1200 is commonly sold as an evaluation board. Two different models of IXP1200 evaluation boards are available to SCAMPI: the ENP2505 and the IXP12EB which differ slightly in the resources present (memory, ports, etc.). Summarising, they provide the following:

scampi@ist-scampi.org
5.3. IXP1200-BASED INTERFACES

- IX bus connect to the MAC ports: the ENP2505 has 2x1Gbps, while the IXP12EB has 2x1Gbps plus 8x100Mbps.
- SRAM to store the routing table and per-flow state (IXP12EB: 2MB, ENP2505: 8MB).
- PCI controller (for 32 or 64 PCI bus, depending on the model).
- Software Developer’s Kit and simulator including a C compiler for the micro-engines.

In SCAMPI we try to perform as much of the processing as possible at the lowest level: i.e. in the microengines. The control processor will be used mostly for control and management. Considering the relatively slow increase in speed of modern PCI buses (compared to that of the networklink), the fewer packets that need to be sent over the bus the better. If packets need to be sent to the host processor, we will do this with the minimum amount of copying, i.e. by mapping the memory of the device to the host as much as possible. Application will run mostly in the kernel and user-space of the host processors (although it is possible that they have some code, e.g. filters, executing at lower levels).

LIACS is currently evaluating various drivers that implement the memory mapping. One of these is a homegrown mapping module, which maps all SDRAM of the IXP all the way to user space. The other is known as the Assan driver, implemented at Georgia Tech. [18] and has a separate UP channel (from IXP to host) and DOWN channel (from host to IXP) which differ in where the actual buffers are located. While the key issue is to minimise data copies, it is not clear what is the fastest way to do so. For SCAMPI applications the precise details of the memory mapping is transparent. They just need to call the memory map operation.

The benefits of IXP1200 for the SCAMPI project

The choice of the IXP1200 was based on several factors including:

- The already active research around routing applications and their implementation over the IXP1200 [37].
- The availability of development tools provided by Intel, to support the development of applications for the IXP1200[14].
- The intuitive architecture of the IXP1200 which will remain mainly the same in the forthcoming new IXP models.

Since the IXP1200 has received wide acceptance, a society that can support the development of systems based on IXP1200 has been formed. Moreover, the new IXP series will be having similar architecture to the IXP1200. This will make easier the transition of already developed applications.

Next we present in brief the architecture of the IXP1200 emphasizing on the features that make it ideal for high speed links. Moreover we will outline the programming process and the development tools that accompany the chip.

scampi@ist-scampi.org 43 March 27th, 2003
Figure 5.10: The Workbench graphical environment. Provides a common interface to several tools.

Software Development Process

The process of developing applications for network processors is far from simple. The network processor programmer will have to exploit the architectural advantages of the design, which in most cases reassembles a small parallel machine. Moreover, debugging and code verification is expected to be quite difficult since the traditional tools do not apply to these custom made architectures.

The Intel IXP1200 processor is accompanied with powerful tools for developing and testing code. From a cycle accurate simulator of the IXP1200 chip up to real hardware development platforms, the tool chain is quite complete. In this section we present the main components supplied by Intel to support the developing of code for its network processor.

Software Tools

Intel’s Software Development Kit for the IXP1200 is composed from the following parts.
5.3. IXP1200-BASED INTERFACES

- The Transactor. A cycle accurate simulator of the IXP1200 chip.
- The Assembler.
- The Linker.
- The microC compiler.
- Ready build software components (ACEs)
- The Workbench. A Graphical interface that incorporates all the above.

The Transactor is a cycle accurate simulator of the IXP1200 chip. It has the ability to simulate all the components of the chip. Moreover, it can simulate network interface cards, SDRAM modules as well as SRAM modules. In addition it can be configured to communicate with external dynamically linked libraries or even remote machines, which are expected to simulate other entities (such as traffic generators, VHDL behavior descriptions, etc.). Therefore, the transactor is a powerful tool to help test and debug code prior downloading it to the real chip.

The assembler and the linker help compile programs written in micro-code. That is in low level assembly code for the micro-engines. (For the StrongArm, there are traditional tool chains for building code.)

In order to help the programmer produce faster code, Intel provides ready software components (micro-ACEs) which can be chained together to build new functionality.

Moreover, Intel provides the microC compiler. MicroC is ANSI-C with intrinsic functions to support the special features of the IXP1200. Using C instead of assembly, greatly reduces complexity of software development.

Finally, all of these tools are incorporated in a single graphical environment, the Workbench (See Figure 5.10). The workbench exports all the functionalities of the above tools in a graphical, easy to use environment. Moreover it has the ability to connect to real hardware (see also next section) in order to monitor code that is running on a real chip.

Hardware Development Platforms

To further facilitate the use of the IXP1200, several developing boards exist. Intel and Radisys are two major suppliers of such boards. Commonly, these boards incorporate:

- The IXP1200 chip,
- Memory resources (SDRAM chips, SRAM chips),
- Network Interface Controllers,
- Network Physical Interfaces (ports),
- PCI interfaces.
The exact specifications of each board vary depending on the developers needs. Commonly, these boards come with support to commercial embedded operating systems such as VxWorks. However, Linux is widely used as the operating system running on top of the IXP1200. This greatly facilitates the development of custom code running on the StrongArm.

To summarize, the Software Development Kit for the IXP1200 is extensive. Moreover, there has been significant research activity around the world which has resulted in significant knowledge resources being available on line. In addition, several companies have developed supplementary tools to further support the fast development of applications (See www.teja.com). All these together form a strong environment to efficiently make use of the IXP1200.

### 5.3.3 Using IXP1200 in SCAMPI

Figure 5.11 shows one scenario of using the IXP1200 in SCAMPI. In this scenario, the purpose of the IXP1200 board is to implement functionality near the wire speed. Therefore, it offloads the main host processor and, even more, it can sustain computation of any sized packets at wire speed.

In the example architecture the following entities exist:

- **IXP1200 Ambassador.** Kernel module providing an API to users wishing to download functions on the IXP1200. Resides on the host processor.

- **Programmer.** Kernel module running on StrongArm. Handles requests of ambassador for downloading code onto the micro-engines and moving data around.
A use case is described using the Trajectory Sampling as the running example [10]. Trajectory Sampling selects packets by applying a hash function on them and keeping only the packets that their resulting hash value is included in a set. In the example, we suppose the user wishes to perform a trajectory sampling with some specific attributes (hash functions/parameters).

To assist function downloading to the IXP1200, a process (IXP1200-ambassador) is running on the host processor offering an API to users wishing to use the IXP1200. This process can be implemented as a kernel module. The ambassador has build-in functions that can be downloaded on the IXP1200. The user specifies which functions wishes to use and supplies possible parameters. The ambassador, based on IXP1200’s current state(load) and on user permissions will accept or decline the request. In the current example, the ambassador should have predefined some hashing functions, and perhaps some of them may be parameterized. It will also have predefined the Trajectory Sampling function, which will take as parameter the hashing function to use. Therefore the user will indicate “Trajectory Sampling” as the desired function and will pass the required parameters: which hash functions to use and with which parameters.

Communication with the ambassador process is accomplished with an API. Specific function calls specify the functions to download and they also indicate the reverse communication path. That is, how results or alerts will be delivered to the end user. In the trajectory sampling example, the user must specify a call back function along with an adequate buffer to hold (perhaps bundles of) sampled packets. In addition, the user may require a threshold of bytes to be gathered before the callback function is called.

Having accepted the request, the ambassador will prepare the necessary code that will program the micro-engines so as to perform the requested task. Preparing the code includes incorporating the user supplied parameters and producing micro-engine object code. That is putting the blocks together and recompiling the code. In the context of this example, the ambassador will put together the specifications of the trajectory sampling with that of the hashing function and the parameters and will produce the final code.

The resulting object file, along with meta-data are passed to a process (the Programmer) running constantly on the StrongArm of the IXP1200. Communication between the ambassador and the programmer is accomplished over the PCI bus. Kernel modules running on both ends provide a NIC abstraction of the PCI interface. That is each end communicates with each other using the network stack.

The Programmer has the responsibility of actually programming the micro-engines with the newly arrived code. Description of the detailed way the programming should be performed is passed in the metadata. Therefore the Programmer is rather simple and does not perform any more sanity checks. It does however move data across the two processors, the StrongArm and the Host Processor. In the case of the trajectory sampling, the Programmer will download the code to all micro-engines. Therefore packets assigned to a micro-engine will have the hashing function performed. Selected packets will be placed in a queue along with a descriptor specifying the matching event (in this case “Trajectory Sampling”).

Code running on the micro-engines has access to all resources available on the IXP1200 board. That is SDRAM and SRAM which are build on board. The results of computations performed should be polled in regular intervals by the ambassador via

scampi@ist-scampi.org     47     March 27th, 2003
the programmer, or the later should push the results back. Assuming that the user has not specified any notification threshold, the programmer will push the sampled packets over the PCI bus to the ambassador, which in turn will notify the end user.

5.4 Commodity Interfaces

5.4.1 Commodity Interfaces in Promiscuous Mode

Although special purpose hardware based on network processors, DAG cards, or intelligent routers can be used for network monitoring, commodity network adapters provide a simple and inexpensive alternative. For example, several commodity network adapters can be set to function in a special mode operation called "promiscuous mode". In this mode, the adapter receives all packets that travel through the network, and not only packets destined for that particular adapter, as is the case in a regular mode of operation. This (promiscuous) mode of operation has been traditionally used to monitor shared communication media, like shared Ethernet, token ring, and wireless media. However, this promiscuous mode can also be used to monitor any network independently of whether it is shared or not. This can be done if we capture a mirror of the link’s traffic (through an optical traffic splitter or a mirroring port within a switch) and send this mirror to a (commodity) network interface which has been set in promiscuous mode (as shown in Figure 5.12). Commodity network interfaces put in promiscuous mode is the simplest form of network monitoring hardware. Although financially appealing, this solution suffers in performance, because the network adapter (through the optical splitter) receives and forwards to the host processor all the network
traffic, even in the case where most of it is not needed. Thus, the host processor is burdened with the task of processing all traffic and filtering the interesting subsets of it. In addition, the host processor needs to copy all packets from the adaptor to the main memory, and may need to process them using some high-level protocol (like IP or even TCP/IP).

5.4.2 A Hierarchical Monitoring System based on Commodity Adaptors

Although commodity adaptors put in promiscuous mode usually have low performance, they can be a valuable part of a high-performance network monitoring system that is based on a load balancer [6]. The load balancer may receive traffic at a high-speed (e.g. at 10 Gbps), may split it at 10 streams (e.g. of 1 Gbps on the average each). Each stream may be sent to a system equipped with a commodity adapter in promiscuous mode [17]. Indeed, figure 5.13 shows the design of such a system. The monitored link is mirrored by an optical splitter, which as previously splits a small portion of the link’s light. This mirror is then sent to a load balancer which splits the traffic (network packets) into a number of different monitoring systems, which in turn receive a subset of the traffic to monitor 2. The balancer must make sure that this division of network packets among individual monitoring systems will not jeopardize the correctness of monitoring applications. For example, intrusion detection applications sometimes do not search individual packets for signs of intrusions, but first concatenate individual network packets into data streams, and then search for signs of intrusions. In these cases, network packets are sent to the TCP/IP protocol, where they may be rearranged. Moreover, duplicate packets will be eliminated, and eventually, the (possibly redundant and out-of-order) sequence of network packets will be transformed into a data stream. Then, the intrusion detection application searches for signs of intrusion in this data stream. To make sure that the data stream is correctly reconstructed, packets that belong in the same flow, should always sent to the same sensor. Such load balancers have already started to appear in the market 3. However, their internal details of operation, have not been disclosed yet. We believe that such hierarchical monitoring systems can be used to address the sophisticated monitoring needs of modern and future high-speed networks.

---

2 At this point we should stress the difference between the splitter and the balancer of figure 5.13. The splitter divides the light in the optical fiber of the monitored link into two identical light beams. The one light beam continues its way to its destination, while the second light beam goes into the monitoring system. We should emphasize that the (optical) splitter has no knowledge of the network packets that this light beam carries. On the contrary, the balancer transforms the light beam into electric signals, and eventually into network packets, and then sends each packet to one of the individual monitoring systems.

3 http://www.toplayer.com/content/products/intrusion_detection/ids_balancer.jsp
SCAMPI monitoring based on load balancer

Figure 5.13: SCAMPI monitoring system based on load balancer and commodity-only network interfaces.
Chapter 6

Software Structure

The overall structure of the SCAMPI software is shown in Figure 6.1.

In the SCAMPI monitoring system, software will run on the host processor (a PC), both at kernel-level and at user-level, as well as on the SCAMPI monitoring hardware. Actually, from the available hardware options described in chapter 5, adapter/router software will be running only on two options: (i) on the option based on special-purpose adapters, and (ii) on the option based on intelligent network routers. Monitoring solutions based commodity network interfaces put in promiscuous mode (as described in section 5.4) will probably not run any software on the adapter, because these network interfaces either do not provide a general-purpose processor, or when they do, they do not provide a development environment that will enable programmers to download and execute their programs on this processor.

One of the most important user-level components, and one of the contributions of this project will be the Monitoring Application Programming Interface (or MAPI). The MAPI will separate the monitoring applications from the monitoring infrastructure. Independent of the underlying infrastructure, monitoring applications will be written using the function calls provided by the MAPI. Thus, the MAPI will enable programmers to write their application once, and then run it on top of different monitoring infrastructures without any changes.

All the boxes of Figure 6.1 that are above the “MAPI” box represent monitoring applications. SCAMPI will be able to run both legacy applications and new monitoring applications. We envision that legacy applications written on top of traditional APIs (like libcpap [20]) will be able to benefit from SCAMPI’s performance through conversion utilities, like pcap2MAPI, which will be a module that will translate legacy pcap monitoring applications into MAPI-based applications, for example, by mapping libcpap calls into MAPI library calls.

Under the MAPI box in Figure 6.1, there are three major horizontal sections that correspond to user-level software, kernel-level software, and software that runs on the monitoring hardware.

User-level software will be composed of following major modules: the job control system, the protection subsystem, and the system that will translate MAPI calls into the underlying operating system calls (shown in the Figure 6.1 as
Figure 6.1: The structure of the SCAMPI Software.
The job control system’s function is to receive submitted monitoring applications, and check whether there exist enough resources to accommodate these applications in the system. Once an application is accepted and starts to run and issue monitoring calls to the MAPI, the MAPI2OS system will translate the MAPI calls into the underlying operating system calls. For example, if an application would like to monitor all incoming traffic to the local domain’s web servers, on a FreeBSD-based system that has only a commodity adapter, the MAPI2OS will create a Berkeley Packet Filter [21]. Similarly, if the application is to run on a Linux-based system, MAPI2OS will create a Linux Socket Filter [12]. Similarly, if the application will run on a PC that will monitor traffic passing through a Juniper router, MAPI2OS will create and install the appropriate filter on the Juniper router [15]. In addition, user-level code will also have code to translate user monitoring requirements into higher-level structures, e.g. graph. We envision that the module constructor will transform monitoring requests into an abstract module-based graph which will represent the monitoring needs of the application. The graph from the module constructor will then be sent to a module optimizer in order to be optimized (if possible). Then the graph will be sent to the module mapper which decides how is the graph going to be partitioned on the underlying computing architecture. Actually, this subsystem may decide on which computing unit (i.e. host processor, special-purpose adapter, or router) each module is going to be implemented. For example, if the application will run on top of a system equipped with a DAG card, MAPI2OS will translate all the MAPI calls into the appropriate calls provided by the DAG interface.  

Finally, the user-level code will have a protection subsystem whose purpose is to enforce protection and resource sharing on the system. It will make sure that ordinary monitoring applications will not be able to crash the system, will not be able to tamper with each other’s data, and will be restricted to the resources it was allocated.

Besides user-level code, a significant part of the SCAMPI software will be implemented as part of the kernel. Most of it will be embedded within a device driver or a loadable module. We will make every effort to incorporate all kernel-level software into drivers or other loadable modules and avoid making any changes to the operating system kernel. In this way our code will achieve maximum deployment because it will be used with different versions of the operating system without the need for (usually conflicting) patches. The driver (i) will provide access to the monitoring hardware and (ii) will implement monitoring-related like filters and aggregation functions.

Finally, a part of the SCAMPI software will be running on the monitoring hardware processor, both in the form of monitoring filters and functions, as well as in the form of protection enforcement code.

6.1 SNMP

The SCAMPI platform will be a complex system that will need some form of management. This management will be done through SNMP and a SCAMPI MIB containing

1 Actually, for performance reasons, the MAPI2OS module may sometimes choose to bypass the underlying operating system abstractions and access the monitoring hardware directly, much like it does on the DAG card, but this performance enhancement is beyond the scope of this discussion.
the management information needed for the management of the SCAMPI platform. There are already many MIBs available that provide useful information like CPU, memory and disk usage. The SCAMPI MIB will not redefine any management objects already available in other MIBs, but will complement these by defining necessary management objects not already existing.

### 6.1.1 NET-SNMP

The SCAMPI MIB will be implemented using NET-SNMP[25]. NET-SNMP is a collection of various tools relating to the Simple Network Management Protocol released under a BSD like licensing. Some of the components in this tool are:

- **Extensible agent** - an SNMP agent that provides the necessary mechanisms for easy extensibility. New MIB modules can be added and removed dynamically without restarting the agent.
- **SNMP library** - a relatively easy to use SNMP library that developers can use to implement new MIBs and SNMP utilities.
- **tools to request or set information from SNMP agents**
- **tools to generate and handle SNMP traps**

The SCAMPI MIB will be implemented in C as an AgentX sub-agent.

### 6.1.2 SCAMPI MIB

The managed objects available in the SCAMPI MIB will be organized into four main groups as shown in figure 6.2. A full definition of the MIB is included in Appendix E.

- **scampiDevices** This group provides information about each SCAMPI adapter and its interfaces that are available through MAPI to do measurements with. It is organized into two tables, one for devices and one for the interfaces available on devices.

  - **scampiDeviceTable** This table provides various information about devices.
    - **scampiDeviceIndex** A unique value, greater than zero, for each device available for monitoring through MAPI.
    - **scampiDeviceName** A textual string containing the name of the device. The name should uniquely identify the device in the host system. An example of a device name is ‘/dev/eth1’
    - **scampiDeviceDescr** A textual string containing information about the device. The string should include the name of the manufacturer, the product name and the version of the device hardware/software.
    - **scampiDeviceAlias** This object is an ‘alias’ name for the device as specified by a network manager, and provides a non-volatile ‘handle’ for the device.
6.1. SNMP

Figure 6.2: SCAMPI MIB
scampiDeviceIfNum  An integer representing the number of interfaces on the device.

scampiDeviceGPSSync  A boolean value used for signaling GPS time synchronization problems.

**scampiDeviceIfTable**  This table provides various information about interfaces on devices.

scampiDevIfIndex  A unique value, greater than zero, for each interface available for monitoring through MAPI.

scampiDevIfDeviceIndex  A reference to scampiDeviceIndex identifying which device this interface belongs to.

scampiDevIfType  The type of interface.

scampiDevIfSpeed  The supported bandwidth of the interface in units of 1,000,000 bits per second.

scampiDevIfAlias  This object is an ‘alias’ name for the interface as specified by a network manager, and provides a non-volatile ‘handle’ for the interface.

scampiDevIfStatus  The current status of the interface. The status can be: active, ready, unavailable, linkLost or unknown.

scampiDevIfPkts  The total number of packets captured by the interface.

scampiDevIfOctets  The total number of octets captured by the interface.

scampiDevIfDroppedPkts  The total number of dropped packets during packet capture by the interface.

scampiDevIfCounterDiscontinuityTime  The value of sysUpTime on the most recent occasion at which any one or more of this interface’s counters suffered a discontinuity.

**MAPI group**  This group provides general information about the MAPI resources.

scampiMapiUsers  The total number of unique users that are currently using MAPI.

scanouMapiFlows  The total number of active flows in MAPI.

scampiMapiFunctions  The total number of predefined functions in MAPI that are currently in use.

scampiMapiUserFunctions  The total number of user defined functions registered with MAPI that are currently in use.
6.1. SNMP

Flow group This group provides information about open flows in MAPI as well as a history of past flows.

scampiFlowMaxHistLength Specifies the maximum number of finished flows that are displayed in the scampiFlowTable.

scampiFlowMaxTime Specifies the maximum age in seconds of entries in scampiFlowTable. This is the number of seconds since scampiFlowStop.

scampiFlowTable This table provides information and statistics about flows that have been created through MAPI.

scampiFlowIndex A unique integer value used for identifying the flow.

scampiFlowKID The ID of the authentication key that was used by the process that initiated this flow.

scampiFlowIfIndex The scampiIfIndex of the interface that the flow uses to capture packets.

scampiFlowCondition The flow condition used when creating the flow.

scampiFlowPkts The total number of packets captured by the flow.

scampiFlowOctets The total number of octets captured by the flow.

scampiFlowDroppedPkts The total number of dropped packets during packet capture by the flow.

scampiFlowGPPSSync A boolean value that indicates GPS synchronization problems during the lifetime of the flow.

scampiFlowStart The sysUpTime of when the flow started.

scampiFlowStop The sysUpTime of when the flow finished. A value of 0 indicates that the flow is still active.

Measurement group The measurement group provides detailed statistics about number of packets and bytes in a time interval, usually in the sub second domain.

scampiMesCfgTable This table is used for configuring measurement jobs.

scampiMesCfgKID The ID of the authentication key belonging to the user who created and controls this measurement.

scampiMesCfgIndex The index of the measurement. Should be unique for the the for the corresponding scampiMesCfgKID value.

scampiMesCfgIfIndex Reference to scampiDevIfIndex and specifies which interface that should be used for measurements.
scampiMesCfgIntervalSec, scampiCfgMesIntervalSec, and scampiCfgMesIntervalFrac together form the time interval for measurements. scampiCfgMesIntervalSec specifies the number of whole seconds of the interval.

scampiMesCfgIntervalFrac, scampiMesCfgIntervalSec, and scampiCfgMesIntervalFrac together form the time interval for measurements. scampiMesStartFrac is the sub-second part of the interval in units of 2⁻³² seconds.

scampiMesCfgMaxLength The maximum number of entries for this measurement in the scampiMesTable.

scampiMesCfgActive If set to 1 the measurement is active and results are stored in scampiMesTable. A value of 0 indicates that the measurement is inactive and no results are put into scampiMesTable.

scampiMesCfgStorageType The storage type of this conceptual row.

scampiMesCfgRowStatus The status of this conceptual row.

**scampiMesTable** The results from measurement jobs defined in scampiMesCfgTable are stored in this table.

scampiMesKID The ID of the authentication key belonging user who created and controls this measurement.

scampiMesIndex The index of the measurement.

scampiMesIntervalId A unique ID for the time interval.

scampiMesStartSec, scampiMesStartSec, and scampiMesStartFrac together form a timestamp for when the interval started. scampiMesStartSec contains number of second since midnight January 1 1970.

scampiMesStartFrac, scampiMesStartSec, and scampiMesStartFrac together form a timestamp for when the interval started. scampiMesStartFrac is the sub-second part of the timestamp in units of 2⁻³² seconds.

scampiMesPkts Total number of packets captured during the interval.

scampiMesOctets Total number of bytes captured during the interval.

### 6.1.3 Using the SCAMPI MIB

Managers will use the scampiDevices and scampiMapi groups to monitor the status of a SCAMPI platform. By polling the scampiDevices group at regular intervals, managers can detect problems with network connections and GPS synchronization. The scampiDevIfDroppedPkts reports dropped packets by an interface. Dropped packets will usually be a sign indicating that the SCAMPI platform is overloaded and is trying to run too many monitoring jobs. However, since a SCAMPI platform is a normal PC other processes that are not related to SCAMPI can also cause CPU congestion.
resulting in dropped packets. By combining the counter of dropped packets with the usage statistics of MAPI from scampiMapi, a manager will usually be able to detect when the cause of dropped packets is measurement jobs.

The scampiFlows group can be used by managers to get a detailed overview of which users are running measurement jobs and who are using the SCAMPI platform the most. By polling this group at regular intervals, managers can generate a usage history of the SCAMPI platform.

The scampiMeasurement group is not related to monitoring the status of the SCAMPI platform but is an SNMP interface to MAPI for counting packets and octets at user defined intervals. A user can set up one or more measurement jobs by creating rows in the scampiMesCfgTable. A row in this table specifies which interfaces that should be monitored and the length of the interval packets and octets are counted over. This interval can be in the sub second domain and the results are stored in the scampiMesTable. This table can contain the results for several intervals so that the user is not required to poll the results too frequently.

### 6.1.4 MAPI management interface

The SCAMPI MIB will collect all the information it needs through the MAPI interface. The following management functions will be used:

- `mapi_get_num_users` returns the number of active users.
- `mapi_get_num_flows` returns the number of active flows.
- `mapi_get_num_funct` returns the number of active functions.
- `mapi_get_device_list` returns a list of devices that are available for measurements.
- `mapi_get_device_info` returns detailed information about a device.
- `mapi_get_interface_info` returns detailed information about an interface including usage statistics.
- `mapi_get_flow_list` returns a list of active flows
- `mapi_get_flow_info` returns detailed information and statistics about a flow

The C header file defining these functions and related data structures is included in Appendix E.1.

### 6.2 The Click Approach

One way of implementing the SCAMPI monitoring functionality is to do it on top of the Click router. Click is a modular software router originally developed at MIT, and is very suited for monitoring purposes [24]. To configure a Click router, modules or elements are linked together. Elements control every aspect of the router’s behavior e.g. communication with devices, packet handling and modification, queuing and
scheduling. New elements, e.g. patternMatcher or TCPCooker, can be written in C++. A Click configuration can run in user- and kernel-mode as a kernel module. At the time of writing, Linux (2.2, 2.4) and FreeBSD are supported.
Chapter 7

User-level Software

7.1 Translation Tools

7.1.1 pcap2MAPI: running legacy pcap applications on MAPI

Although novel applications will be able to reap the performance benefits of the MAPI implementation, legacy applications should be able to run on top of SCAMPI monitoring systems as well. This can be done by providing a tool that translates the API of traditional monitoring libraries into MAPI. pcap2MAPI is a library that translates calls made using the pcap API into MAPI calls. Some of these translations are straightforward, while others need more support. For example, the `pcap_open_live()` call can be rather easily mapped into the MAPI `create_flow()` call. Similarly, the `pcap_next()` call can be mapped into the MAPI `get_next_packet()` call. On the other hand, we should not imply that all calls can be mapped one-to-one. For example, the `pcap_lookupdev()` call does not have a matching call in MAPI and needs to be implemented (or propagated to the lower software layers). Similarly, the `pcap_dispatch()` call that is used to process several packets will need to be implemented using a repeated execution of the `get_next_packet()` MAPI call, or a non-trivial invocation of the `mapi_loop()` call.

7.2 Job Control

If SCAMPI is to support multiple applications that may be active simultaneously, it needs to provide some form of resource control, both for safety (in terms of not being able to bring the system in a corrupt or inconsistent state), sharing (i.e. ensuring applications only use those resources that are allocated to them) and security (in terms of not being able to touch other applications’ sensitive data/packets). We consider monitoring applications to be the clients of the SCAMPI platform, which acts as a server. In this section, we focus on admission control (AC), which deals with questions like:
1. does the client have the required privileges to perform a specific operation at the server?

2. given the total resource capacity and the resources currently in use, can we accommodate the client’s request?

The former aspect is handled by explicit trust management and in essence depends only on the credentials presented by a client. As we shall see, the latter is dealt with in a similar fashion, but with the subtle difference that per-request state may be required at the server side. Besides AC, we also need to enforce the resource control. This is the topic of Section 7.3.

### 7.2.1 Trust management

All authorization in SCAMPI will be based on delegated trust management via explicit credentials as implemented by Keynote [4]. In such a scheme, a server initially contains a policy, stating explicitly which clients are allowed to do what. An example of such a policy is shown below:

```
KeyNote-Version: 2
Comment: This is a trivial policy: authorize 2 licensees to perform a specific operation 'createFlow' in the 'SCAMPI.MAPI' application domain. The licensees are identified by their public keys.
Authorizer: "POLICY"
Licensees: "rsa-base64:MEgCQQDMcZukqn3Wa4Z3y3Wk1jb/eoFnDrfNN\B72OLJsfl65SnFRLbxrgEnEP7LevQRI0KsUq86NgQntx1btq\lqyETdAgMBAEE=" &
"rsa-base64:MEgCQQCUzaqp20cx8YFJBPqGUs8g2OJfghkU\3cUSBnTYqPqV/L4ZrjD28Hb2p1RPacTI1IJPtEdhMg2M2LqfAqMBAAA="
Conditions: app_domain == "SCAMPI.MAPI" && operation = "createFlow" -> "true";
```

The policy explicitly allows two clients to perform the `createFlow()` operation, provided this is taking place in the SCAMPI.MAPI application domain. Many other conditions may be specified, e.g. concerning the parameters of the `createFlow()` operation. The name/value pairs in the conditions are not interpreted by KeyNote, and may be assigned any semantics appropriate to the SCAMPI project.

Whenever a client wants to perform the `createFlow()` operation, it has to provide the corresponding `credential`, proving that it was authorised to perform this operation. As an example, the following credential might serve as sufficient authorisation for a client with public key `rsa-base64:ABCDE12345`, provided it is trying to call the operation `createFlow()` with `10.0.0.1` as the second parameter:

```
Comment: OKE CL credential authorising client to load code of this type
Authorizer: KEY1
Local-Constants:
  KEY1 = "rsa-base64:MEgCQQDMcZukqn3Wa4Z3y3Wk1jb/eoFnDrfNN\B72OLJsfl65SnFRLbxrgEnEP7LevQRI0KsUq86NgQntx1btq\lqyETdAgMBAEE="
Licensees: "rsa-base64:ABCDE12345"
Conditions: app_domain == "SCAMPI.MAPI" && operation = "createFlow" -> "true";
Signature: "sig-rsa-shal-base64:DulhVTvNV8sAnhjnI/9Unz1H+/VM\9GnSm/pspE0gW0g/QzXHY2gwaWebFIP1zAF/qnBGOwusxk1KElz"
```

scampi@ist-scampi.org  62  March 27th, 2003
In the above fashion we see that trust may be delegated, i.e. an authorising party may delegate part of its privileges to another party (the licensee). In this case, the licensee that was mentioned in the policy, explicitly grants part of its trust to the client with public key rsa-base64:ABCDE12345 with an extra constraint on the second parameter of the call. Note that the credential is unforgeable, because it was signed (with the private key of the authoriser). It may be necessary to submit multiple credentials, which together authorise the request, e.g. when the policy authorises A, A authorises B, and B authorises C, C will have to provide both the AB and the BC credentials.

The credential scheme in KeyNote is extremely powerful and the expressions permissible in the conditions are fairly flexible (for example, we may specify that values X and Y combined should be less than Z, etc.).

7.2.2 Operation authorisation

For security reasons, each operation in the MAPI should be authorised. In this way, it is possible to restrict the access to resources (including packets) to specific clients only. However, we do not want to send a specific set of credentials for each and every request. We therefore use an explicit authorisation operation which provides the appropriate authorisation settings:\footnote{scampi\_set\_authorisation\_creds may actually be implemented with a set of mapi\_set\_flow\_option calls that will be checked with at the time of mapi\_connect call that will be defined in chapter 9.}

\begin{verbatim}
int scampi\_set\_authorisation\_creds (char *privkey, char *pubkey, char *creds);
\end{verbatim}

The reason why clients need to provide their private keys in addition to the public keys is that they need to be able to prove that they really are who they say they are (as indicated by their public key). Normally this is implemented by signing a nonce value with the private key (challenge-response). This is also the approach taken in SCAMPI.

There may be some concern about the security of an operation that requires clients to submit their private keys, but we stress that this operation is a local library call only and the private key will not actually be sent through the MAPI (or anywhere else for that matter). The reason for using the above interface is that it relieves the clients from implementing the fairly tedious challenge-response interaction themselves. At any rate, in case this is needed, clients may still choose to do so.

Every subsequent operation will be automatically authorised with the credentials and keys provided. However, this is completely transparent to the clients. Also, if no authorisation is required (for example, if the client ‘owns’ the network), the credentials and policy, etc., may be NULL.

At admission-control time, we may also check whether or not sufficient resources are available in the system to accommodate the new request. For example, it may be specified in a policy that the maximum number of flows that can be created by a client should not exceed \( N \), or the number of functions applied to a flow should not exceed \( M \). This also means that when a request is accepted by admission control, the available resource capacity changes and should be updated.

In other words, whichever resources are taken into consideration (simple examples could be number of flows, or data rates in the flows), some accounting of resources is

scampi@ist-scampi.org

63

March 27th, 2003
needed, e.g. to keep track of the reserved resources (on a system-wide or per-request basis). This is also needed for the admission control component, which needs to keep state about the currently available resource capacities.

For this purpose, a resource database component is added to the SCAMPI server. The resource database is responsible for keeping track of the resources in the server’s domain. Each request that leads to an explicit resource reservation (e.g. loading code that instantiates a queue of specific size) may be entered in the database, together with the resources reserved/allocated on its behalf and dependencies between requests. All of this is transparent to the clients. It is only relevant to the admission control procedure.

### 7.2.3 Custom code loading and compilation

One of the goals of SCAMPI is to allow new functionality to be loaded below the kernel-userspace boundaries by clients with the appropriate privileges. There may be two ways in which this is done. If security is not an issue, clients should be able to insert new modules that automatically link with the predefined functions. For example, in a Click-like software model, it may be possible to link a new component in between two existing elements. In case security and strict resource control is needed, the components that will be linked will be OKE components.

Note that the question whether or not it makes sense to execute code in the kernel depends partly on the hardware. On a device with large mmap-able packet buffers, it may be possible to run all the application code in user space. For devices where this is not possible, kernel code is constantly active to pull the packet off the card. In this case, it would be expensive to context switch to user space all the time to process the packets. Instead, it may be better to execute as much of the code as possible in the kernel.

We distinguish between code loading and code application. When loading a new function, we only add (and link) some object code that was previously unknown to the system, without actually executing it (beyond what is needed for registration). Applying the code means that the function is added to a flow that was previously created. Both operations need to be appropriately authorised. In an extreme case, one may be able to use code that one is not permitted to load, and vice versa.

Code loading means different things in different contexts. For example, in userspace, it may simply be a matter of opening a shared library (dlopen()), while a kernel load requires inserting a kernel module insmod). There is even more involved in programming a network processor and this is certainly the case if reprogramming an FPGA is allowed to inject new functions at this level. Moreover, the binary formats for most of these architectures vary widely. While we do not claim that within the timeframe of the SCAMPI we will support code loading for all of these architectures, we do aim to provide the mechanisms to do so. For this purpose, the load operation takes two parameters that define what sort of code this is and where it is supposed to be running: (1) the type of code to be loaded, which can be OKE code or plain code, and (2) the place where this code will be run (e.g. in the kernel on an i686 host processor, or in the kernel of the StrongArm control processor on an IXP). Whether or not one is permitted to load such code at this location is a separate issue and handled by the
7.3. RESOURCE MANAGEMENT AND PROTECTION

Two more parameters are passed to the load function. First, a unique identifier in the form of a null-terminated string. This allows us to refer to the function at a later stage, e.g., to apply it, or to remove the code. Second, we pass a set of parameters that allows us to be more specific in applying the function, or to set limits on the application of this function. For example, when creating a queue function, it may be useful to pass as parameter the size of the queue. These parameters are used for many purposes: code instantiation (e.g., \texttt{kmalloc} the appropriate amount of memory), resource accounting (e.g., keep track of how much buffer space was allocated on behalf of this client), admission control (e.g., reject the request if there are insufficient buffers available, or reject all requests for queues exceeding a specific size). For OKE modules, the parameters also contains a pointer to the environment setup code (ESC) and may further contain upperbounds on CPU time, stack space, etc.

\begin{verbatim}
enum code_location { HOST_USER_i686, HOST_KERNEL_i686,
                    IXP1200_USER_STRONGARM, IXP1200_KERNEL_STRONGARM,
                    IXP1200_ME, JUNIPER_X_Y_Z, ... };
enum code_type { PLAIN, OKE },
int scampi_load_code (char *id, // unique identifier
                      scampi_code_t code,
                      enum code_location code_location,
                      enum code_type code_type,
                      param_t *param);
int scampi_unload_code (char *id);
\end{verbatim}

Applying the code should be as simple as possible. We intend to use the same MAPI function for both predefined functions and functions that were loaded by clients. The system should be aware of where the functions are located and hence, regardless of privilege considerations, whether or not it makes sense to apply a function. For example, suppose a flow is created and the level at which it is created is the kernel of the operating system (i.e., there is no notion of that flow at lower levels). In that case, it makes no sense to try and apply a function that was loaded in the IXP1200 to that flow, as it would require data to travel up and down the PCI bus several times.

### 7.3 Resource Management and Protection

OKE modules are protected by OKE resource and access control [5]. They may be restricted in their use of the processor, memory, API, etc. For ‘plain’ object code such restrictions are not possible. However, this does not mean that no resource control is possible for vanilla functions at all. In particular, through the use of adding parameters and KeyNote-based authorisation, we may limit, for instance, the amount of traffic generated by a function. Consider the following two scenario’s:

**Scenario 1.** The client owns the network and simply wants to get the packets as fast possible, without any limitations whatsoever.

**Scenario 2.** The clients want to get as many packets as possible, but they are not the owners of the network and their flows are restricted to only 100Mbps. This is specified in their credentials, e.g., in the form of leaky bucket parameters.

scampi@ist-scampi.org 65 March 27th, 2003
Scenario 1 is straightforward. All packets enter the flow as quickly as possible to be processed by the functions corresponding to the flow. Scenario 2 is more interesting from a resource control point of view. Whenever we create a flow in scenario 2, we should also explicitly state that we want a leaky bucket applied to the traffic entering (or exiting) our flow. This should be specified in the parameter list, supplied by the client. If we fail to do so, the \texttt{mapi\_connect\_flow()} operation will return with an error message, as the request is rejected by the admission control procedure.

### 7.4 The Click Approach

#### 7.4.1 The Click Model

A Click router is an interconnected collection of modules called elements [24]. Elements are processing components that control every aspect of the router’s behavior, from communicating with devices to packet modification to queuing, dropping policies and packet scheduling. Individual elements can have very powerful behavior, and it’s easy to write new ones in C++. The programmer simply writes a router configuration by gluing elements together with a simple language.

The Click language describes the configuration of a Click router (or monitor, or firewall, or any other kind of network system). It has two directives: (i) \textit{declarations}, which declare new elements and (ii) \textit{connections}, which connect those elements together. Click router configurations are directed graphs of elements, declarations specify the vertices of the graph and connections specify the edges.

### 7.5 Distributed MAPI

Every monitoring application addresses the monitoring platform by calling MAPI functions. An application uses the functions defined in the MAPI interface to configure the monitoring agent and retrieve results from it. Whether this agent is running locally or remotely is of no concern to the application. The MAPI should be completely transparent to the end user.

### 7.6 MAPI

#### 7.6.1 Passive Monitoring

User applications will be able to invoke the SCAMPI’s monitoring API and receive network packets as well as aggregated statistics in user space. A detailed treatment about MAPI, its functions, variables and the format that it uses can be found in chapter 9.
7.6.2 Active Monitoring

Active Monitoring deals with the measurement and monitoring of network characteristics by sending test-packets through the network and analyze their treatment upon reception. Active monitoring is performed between two end-points: a sending node, and a receiving node. The sender sends a stream of packets according to a well-defined pattern, and annotates them with sequence and timing information. The receiver uses these annotations to calculate network characteristics (like one-way loss ratio and one-way delay).

An important observation here is that the receiving part reassembles the passive measurement functionality a lot. So at the receiving side, we can create a network flow specifically for analyzing the packets injected at the sending side. Specific functions must be applied to the passive network flow in order to analyze the sequence and timing information.

7.7 MAPI Implementations

7.7.1 MAPI on top of pcap/BPF/LSF

The most fundamental operation that is necessary in order to implement MAPI is a way to receive raw packets. In libpcap this can be done by using the function pcap_next. So, one possible way of delivering packets to MAPI applications running on top of pcap is to have a function (say mapi_rcv), which in an infinite loop will call pcap_next. Note that besides delivering raw packets to end applications, MAPI enables them to apply functions to packets. Thus, when mapi_rcv gets the packet it will iterate over the list of installed functions, calling every function and passing as an argument the received packet. The function mapi_rcv will be called every time an application calls the function create_flow which will be similar with the libpcap function pcap_open_live. The function create_flow will create a structure (say mapi_t, which will enclose the structure pcap_t and the list of functions to be called. This list is a list of structures (say predef_func) which contains among other things a local BPF filter and a function pointer. To be accurate, every time create_flow is called a new thread or process must be created which will run the function mapi_rcv.

7.7.2 MAPI on top of Click

The Monitoring API is a library that can be compiled into any application to encapsulate the communication with the centralized SCAMPI components. The MAPI can either run co-located with the SCAMPI monitoring device or can run remotely and use a specifically designed protocol. On top of Click, a C++ version of the MAPI is being implemented. An application can configure a monitoring job by creating an instance of the MAPI class. This instance represents the monitoring job in an Object Oriented fashion. The following functions form the basic infrastructure for the Object Oriented MAPI.

- Network flow creation: adds a new network flow to the system.
CHAPTER 7. USER-LEVEL SOFTWARE

- Apply function: append the functions defined below, or User-defined functions added to the system, to the chain of operations to be executed on the netflow.

- Network flow removal: releases the resources related to the netflow.

- Information retrieval: gets information from a given netflow, including statistics, packet traces or flowrecords.

- Events: get asynchronous information from the given netflow.

The functions can be subdivided in the following categories:

**Cooking:** re-order and select packets in such a way that a transport or application-level flow can be analyzed.

**Flow selection:** this category classifies incoming packets into a network flow. This can be based on IP/TCP/UDP header fields, a fixed bit pattern or a pattern to be matched in a cooked stream. Also a set of sampling selectors is available (deterministic, random or based on hashing of packet header fields). Flow selections can be concatenated to refine the selection. The resulting selected packets are sent to the following steps in the chain.

**Flow analysis:** get analysis results from a network flow. This include byte and packet counters, bandwidth and packets per second analysis.

**Flow export:** export a resulting packets or statistics into a flow record, a tcpdump file or proprietary format.

### 7.7.3 Module Creator

The module creator is a component that translates a monitoring job (created by calling MAPI functions) to a similar module representation. A first step in creating a monitoring job is obtaining the required network flow, i.e. all packets are selected that are needed in the monitoring job. Modules such as classifiers, samplers and patternmatchers will be used to create the target network flow. Once the monitor obtains the target network flow, several functions will operate on that subflow. These functions will process the flow in order to obtain the required monitoring information. Functions such as return the entire network flow, return only packet headers, count packets or bytes and measure bandwidth of the flow will be supported.

Figure 7.1 depicts the module representation of a generic monitoring job. It consists out of different packet processing components that are linked together. If a packet conforms the selection criterion in a certain component, it is sent to the next component in the chain.

Because a SCAMPI monitoring agent will process multiple monitoring jobs simultaneously, a global monitoring graph will be created. The global monitoring graph will be a tree with the Raw Network Flow as root and consists out of all merged monitoring
jobs. This tree has to be optimized in order to support as many monitoring jobs as the
monitor (hardware and software) can handle. Every node in the graph has an associated
cost or processing time, which conforms the number of processing cycles needed to
process a packet. A patternmatcher for example has a higher cost than a simple counter,
a filter will have a higher cost than a periodic sampler. Each packet will traverse the
tree top-down. It’s possible that a packet will branch in a certain node and travel on
multiple paths simultaneously. The ultimate optimization goal is to minimize the total
processing time for each packet.

7.7.4 Module Optimizer
A straightforward solution to create the global monitoring graph is to "tee" the raw
network flow for each monitoring job. In this case each monitoring job will operate on
this raw network flow. This approach is very easy to implement, but will definitely not
be the most efficient solution. Other techniques and algorithms will be used to optimize
the monitoring graph.

7.7.5 Module Mapper
The module mapper will map a monitoring graph to the corresponding Click repre-
sentation. The Click kernel module can be configured by writing the configuration
to the “proc” filesystem (“proc/click/config” or “proc/click/hotconfig”). A Click con-
figuration is described in “Click language”. Due to the fact that both the monitoring
graph and Click configurations are directed graphs of elements, the conversion is very
straightforward.
7.8 Monitoring applications

7.8.1 Intrusion and DoS detection

Over the last few years, the Internet has been repeatedly used as a medium to launch attacks against computer and communication subsystems. Such attacks, which are usually called cyber-attacks may disable a large number of computers, which may in turn paralyze critical infrastructures including telecommunications, provision of electric power, transportation, water supplies, athletic infrastructure, and commerce. Such cyber-attacks propagate rapidly and may have profound impact. For example, in 2001 a computer worm/exploit named CODE-RED was released on the Internet and infected more than 340,000 computer systems in less than 24 hours. Indeed, Figure 7.2 shows that CODE-READ spread very rapidly even during business hours, and within a day it infected computers in practically every corner of the earth. This CODE-RED incident was not an isolated case. Actually, the frequency of such events is currently on the rise. For example, the number of computer-related vulnerabilities reported to CERT (the Carnegie Mellon University Computer Emergency Response Team) is currently increasing exponentially, doubling every year or so for the last 2-3 years. To reduce the number, spread, and impact of Internet-related attacks, Intrusion Detection Systems and Denial-of-Service Attack detection systems are currently being employed on an increasing number of strategic Internet points. Within SCAMPI we plan to take an open-source Intrusion Detection System (like snort [34] or Bro [28]), and run it on top of the monitoring infrastructure.

7.8.2 QoS and TE monitoring

This paragraph depicts the communication process/needs between the QoS and the Traffic Engineering Applications implemented at the User Level of the SCAMPI plat-
form and the underlying MAPI platform. More specifically, the user needs for QoS and TE Monitoring applications are described and how can these needs be "translated" into a format that is understood from the underlying MAPI.

Due to abundance of bandwidth in current networks, the attention in Quality of Service (QoS) in computer networks has recently shifted from what we can call “protective” QoS to so called “proactive” QoS. Instead of providing priority treatment for certain class of traffic or guaranteed sharing of available capacity we care about configuring the network and the end-hosts so that we can get an optimum performance even over long-distance networks.

However, the requirements for QoS monitoring have not changed too much. We still need to know how well the network performs, whether it satisfies required SLA (Service Level Agreement). Primary network QoS characteristics are one-way packet loss rate, one-way delay, delay variation and throughput. All characteristics can be measured by both active and passive monitoring. Recent studies show [2] that relationship between active and passive one-way packet loss rate measurement is still not clear. Therefore, it is better to support both types of measurement and carefully compare the results. Passive monitoring of one-way delay and delay variation requires correlating packets captured at two monitoring points. Additionally, one-way delay measurement (both passive and active) requires time synchronization between monitoring points. Throughput is traditionally measured with active monitoring, although it can also be monitored passively by counting packets or segments.

Resolution of complex performance problems related to transport protocol issues requires detailed monitoring of protocol behavior. For this purpose we need smart monitoring architecture capable of programmable monitoring of specific patterns in packet headers. SCAMPI will provide both hardware support for counters of specific patterns and software application for QoS and performance monitoring.

7.8.3 Quality of Service Monitoring Applications

Quality of Service (QoS) monitoring applications should monitor the underlying network in order to measure the quality parameters. This is achieved through:

- Passive observation of packets
- Active measurements

QoS applications might need data/metrics from more than one nodes and therefore multiple observation points should be installed on the network.

The Quality of Service parameters that the SCAMPI platform should provide to the end users are:

1. **One way packet loss**
   
   An end user (e.g. an ISP, a network administrator) may want to know how many packets are lost and re-transmitted during a session.

   **Possible SCAMPI “implementation”:**
   
   Count the TCP sessions in each packet flow and monitor the sequence numbers (SN) field of the TCP header in each session.
2. **One-way delay**
   An end user may want to know how much time did it took for a packet to travel from one point of the network to another.

   *Possible SCAMPI “implementation”:*
   Utilization of: a) two monitoring probes, one at each endpoint and b) timestamp mechanism. The first probe will inject traffic to the network and the second probe will receive it. The one-way latency between the two end-probes can be calculated by subtracting the timestamp registered at the receiving node from the timestamp registered at the sender node, provided that the computer clocks in both sides are synchronized.

3. **Packet delay variation (Jitter)**
   An end user may want to measure the inter-packet delay variance during flows.

   *Possible SCAMPI “implementation”:*
   During a flow, the arrival time of each packet will be stored. Arrival time of consecutive packets are subtracted and the result of this subtraction is given as an input to the QoS Applications, that compute the distribution of inter-packet delay variance. Real time applications like voice or video are very sensitive in this parameter, which is desirable to be as stable as possible.

4. **Throughput**
   An end user may want to know how many packets have passed from a specific network link at a specific time interval.

   *Possible SCAMPI “implementation”:*
   Count packets and their size in a time period between destination and source, then add the size of the packets and divide the outcome with the time duration, this number gives the total throughput between two points. If we want to measure the throughput per application SCAMPI must check the packet payload in order to find the specific application to which the packet belongs.

5. **Round Trip Time**
   An end user may want to know the overall time that will be needed in order for a packet to travel from the source to the destination plus the time needed for the destination’s response to reach the initial source.

   *Possible SCAMPI “implementation”:*
   Utilization of: a) two monitoring probes, one at each endpoint and b) timestamp mechanism. The first probe will inject traffic to the network and the second probe will receive it and will reply to it. The round trip time between the two end-points can be calculated by subtracting the timestamp inserted into the packet by the sender, from the time that the same packet is received back from the source.

### 7.8.4 Traffic Engineering Monitoring Applications

Traffic Engineering Applications can monitor the underlying network in order to be able to perform a variety of functions like:
7.8. MONITORING APPLICATIONS

- planning
- dimensioning
- adjustment
- admission control
- diagnostic monitoring
- performance optimization
- trend analysis
- creation of business models

Traffic engineering applications in general need flow-based aggregate statistics, that report the amount of traffic per flow.

1. **Aggregate statistics per host**
   The User (e.g. the network administrator) often needs to know the traffic generated by a specific host.
   
   *Possible SCAMPI “implementation”*:
   Count traffic volume per IP

2. **Aggregate statistics per Autonomous System**
   The User (e.g. an ISP) often needs to know the traffic volume that is exchanged between Autonomous Systems (AS).
   
   *Possible SCAMPI “implementation”*:
   Count traffic volume of each IP that belongs to a specific AS

3. **Aggregate statistics per DNS domain**
   The User (e.g. an ISP) often needs to know the traffic volume that is exchanged between DNS domains.
   
   *Possible SCAMPI “implementation”*:
   Count traffic volume of each IP that belongs to a specific DNS domain

4. **Aggregate statistics per application**
   The User (e.g. an ISP) often needs to know the traffic amount generated from each one of his applications (e.g. FTP, HTTP, NNTP).
   
   *Possible SCAMPI “implementation”*:
   Receive network packets and partition them in application categories according to the TCP port number (e.g. for TCP applications), or according to further information retrieved from the packet payload.

5. **Aggregate statistics per priority**
   The User (e.g. an ISP) often has different traffic classes, with different priorities for each class (e.g. VoIP, Video, Internet Traffic) and needs to know the traffic amount generated from each class.

scampi@ist-scampi.org 73 March 27th, 2003
CHAPTER 7. USER-LEVEL SOFTWARE

Possible SCAMPI “implementation”:
Count packets according to the Type of Service (ToS) IP header field.

6. Aggregate statistics per time period
The User (e.g. the network administrator) often needs to know the traffic generated during: a) different time zones during a day (morning, noon, night), b) specific days within a week (e.g. during the weekend), etc.

Possible SCAMPI “implementation”:
Count all traffic volume passing through the network during the specified time interval.

7.8.5 Threshold Alerting Application
Next to the ‘observational’ applications, which observe the networks, e.g. to derive QoS-statistics, network management applications (and more specifically TE-applications) tend to use monitoring for adding an automated feedback loop to the network management applications. As an example, support for this feedback mechanism is being developed in the IETF RAP working group [32]. For each provisioned (configured) part of a network device, we can request to receive a notification related to a monitored value of this “object”:

- at regular intervals.
- if the value changes.
- if a threshold value is crossed.

In the context of scampi, the threshold alerting application can link “events” (as a function) to the monitoring job specification. Some examples are: notify if measured delay becomes too big (notification linked with an active monitoring analysis function) or notify if a user takes up too much capacity (notification linked with bandwidth metering). As a result, the network management system might take automated actions (optionally while still keeping a record for diagnostics).

In terms of implementation, events can be expressed using a call-back function linked to a blocking mapi_read_results call.

7.8.6 Network debugging
Network debugging is used to detect anomalous behavior on the network. Some problems can be detected from flow records. However, a detailed analysis of the header of each packet is often required. If higher level protocols or tunneled protocols are being debugged, some part of the packet payload also needs analyzing.

Network problems can sometimes be related to timing. To detect this it is important that the underlying monitoring platform provides an accurate timestamp with high resolution.

Sampling can cause problems with network debugging since important packets that can identify problems may be left out by the sampling process.
7.8.7 **Flow-based reporting**

This application will generate flow based reports based on the input from NetFlow/IPFIX records. The application will take advantage of the SCAMPI IPFIX exporter which can export extra information not normally found in flow records to generate novel reports not available by other existing applications.

Figure 7.3 shows the various components of this application. All the components will be implemented as part of the SCAMPI project although some of them will be heavily based on already existing software like FlowTools\(^2\).

**SCAMPI IPFIX exporter** collects flow data from MAPI and exports it using the IPFIX protocol.

**NetFlow Collector** collects raw NetFlow/IPFIX data and stores it to disk in flat files.

**Report Generator** analyzes the raw flow data and generates reports that are stored in an SQL database.

**CLI Interface** collection of small command line based applications for analyzing and printing information stored in raw flow files.

**Maintenance scripts** aggregates reports, deletes old records and updates information about available observation points, name of ports and AS numbers etc.

**SQL Web interface** allows users to browse the reports stored in the SQL database through a web interface.

**Database design**

All generated reports will be stored in a PostgreSQL database. Figure 7.4 shows the basic principals of the database design. There will be three tables that contains common information about devices, observation points and network links for all report types. Reports will provide statistics for a certain time period and resolution. The first implementation will provide statistics for an hour, a day and a month. It will however be easy to add other time resolutions. For each time resolution there will be several report types and each report type has its own table.

---

\(^2\)http://www.splintered.net/sw/flow-tools/

scampi@ist-scampi.org
CHAPTER 7. USER-LEVEL SOFTWARE

Figure 7.4: Database design

Figure 7.5: ER diagram
Figure 7.5 shows the database design for a report about IP protocol usage. For each new report type that is added, new tables will be created to store the information. The tables with a dark background are common for all reports.

One advantage of having separate tables for time resolutions like hours, days and months is easier management. If cascading deletes are used in the SQL definitions, deleting old records in the time resolution tables will delete the corresponding information in all report tables as well. This means that the maintenance script that deletes old records do not have to be updated when extra report types are added.

Storing reports in separate tables provide a good isolation between the various reports so that it is easier to add and delete reports without influencing the performance of other reports.

The tables storing information about hours, days and months, uses smallint to store the information about specific year, month, day and hour. This is not as efficient as using the data type date, however it makes it easier to implement a generic web interface that does not have to know all the details about time resolution.

**Device**  This table that contains information about available devices that are used to collect flow data. This can be a router or a SCAMPI PC with monitoring adapters.

**Attributes**

- id  unique id for the device
- ip  ip address
- name  name of device
- path  path to where on the disk the raw flow data from this device is stored. This is used by the report generator to locate the data.

**observation_point**  Information about a single observation point.

**Attributes**

- id  unique id for the observation point
- device  reference to unique id of a device. Specifies which device the observation point belongs to.
- ifindex  used for representing the ifindex value in NetFlow data
- ip  IP address of the observation point if applicable
- name  name of observation point
- sampling_interval  sampling interval used by the observation point
- sampling_method  sampling method used by the observation point
 CHAPTER 7. USER-LEVEL SOFTWARE

**hour**  List of all hours for which reports are available.

**Attributes**
- **id**: unique id
- **year**: specifies the year
- **month**: specifies the month of the year
- **day**: specifies the day of the month
- **hour**: specifies the hour of the day

**day**  List of all full days for which reports are available.

**Attributes**
- **id**: unique id
- **year**: specifies the year
- **month**: specifies the month of the year
- **day**: specifies the day of the month

**month**  List of all months for which reports are available

**Attributes**
- **id**: unique id
- **year**: specifies the year
- **month**: specifies the month of the year

**protocol_hour**  This table contains statistics about protocol usage for one hour intervals. Other report types will have similar tables although the exact type and number of attributes may vary.

**Attributes**
- **id**: unique id
- **hour**: reference to an hour in the hours table and specifies the hour this report belongs to.
- **obs_point**: reference to an observation point and specifies the observation point this report belongs to.

scampi@ist-scampi.org  78  March 27th, 2003
flows number of flows in this time interval
octets number of octets in this time interval
pkts number of packets in this time interval

**protocol\_day** Same as the protocol\_hour table except that the data is for one whole day.

**protocol\_month** Same as the protocol\_hour table except that the data is for one whole month.

**protocol\_names** Table used for translating protocol numbers into names.

**Attributes**
- protnr protocol number
- name name of protocol

**NetFlow/IPFIX Collector**

To capture NetFlow/IPFIX records, flow-capture from Flow-tools will be used without any changes. This is a small application that can receive various versions of NetFlow records and store them to disk. The current version supports NetFlow versions 1,4,5,7 and 8, however the main developers of Flow-tools have stated that NetFlow version 9 and/or IPFIX support will be added as soon as a stable specification is ready.

**CLI interface**

The CLI interface will use the existing applications that are part of Flow-tools. There are already tools for filtering flows based on information stored in flow records, printing flow records and generating reports. The tool for generating statistical reports is flow-stat. As part of the SCAMPI project flow-stat will be extended to provide new reports that use the extra information that is available from MAPI.

**Report Generator**

The basic design principal for this application is shown in figure 7.6. The application will be implemented in Perl and DBI will be used as interface to the SQL database.

**Report types** One important design criteria for this application is that it should be easy to extend it with future report types. This is solved by keeping the code for each report type in a separate file. The main program finds all of these files and runs them one at a time. This is done by requiring that all files have a function with the same name as the file.
CHAPTER 7. USER-LEVEL SOFTWARE

Figure 7.6: Report generator

scampi@ist-scampi.org 80 March 27th, 2003
7.8. MONITORING APPLICATIONS

\[ \text{<filename> (dbhandler, resid, interface, file)} \]
where:

- **<filename>** name of file and name of the function.
- **dbhandler** SQL database handler
- **resid** id of the entry in the highest resolution table for which the report should be generated
- **interface** id of interface for which this report should be generated
- **file** name of file that contains the raw netflow data

**Flow-stat based report scripts** Many report scripts will simply use flow-stat to generate the reports and then insert the values into the database. For these kinds of reports a function will be available that provides an easy method for inserting flow-stat reports into the SQL table. The function will be:

\[ \text{flow-stat (dbhandler, sql-cmd, flow-stat-opt, limit)} \]
where:

- **dbhandler** SQL database handler
- **sql-cmd** SQL command used for inserting reports. The following format is used:

\[
'\text{insert into <table> (<scol1>,<scol2>....) select}$ \\$
\text{fcol1,$fcol2... where not exists(select <scol? from <table> where <scol?>=$fcol? and ...)')}'.
\]

where **scol**=SQL column name, **fcol**=flow-stat column

- **flow-stat-opt** options passed to the flow-stat applications
- **limit** used for limiting number of entries that should be stored in the database. A value of 0 indicates that all entries are stored.

**Config file** There will be a configuration file for configuring parameters for the report generator and for individual reports. This config file will be stored in the same directory as the main script and will contain a list of parameters with one parameter per line. The format will be:

\[ \text{<parameter name>=<parameter value>} \]

Parameters that are specific for a certain reports should use the following naming scheme for parameter names:

\[ \text{<parameter name>=<report name>_<parameter>} \]

There will be a function available that scripts can use to read configuration parameters:

\[ \text{readCfg(''<parameter name>'')} \]

scampi@ist-scampi.org 81 March 27th, 2003
CHAPTER 7. USER-LEVEL SOFTWARE

Maintenance scripts
The maintenance script will be a collection of perl scripts that performs various task for maintaining and aggregating the information stored in the database.

Aggregating reports  The report generator generates reports for the highest time resolution available. At regular intervals these reports must be aggregated to lower time resolutions. This will be done by a collection of SQL commands that calculates the aggregated values.

In the first implementation there will be SQL commands that aggregates information from hourly reports to daily reports and from daily reports to monthly reports.

Deleting old reports  It is not possible to store an infinite number of reports at the highest time resolution. Limited hard disk space and performance issues makes it necessary to delete older reports while keeping the information available at lower time resolution.

Deletions of old reports will be done by SQL commands that delete entries in the time resolution tables. All report tables will be created with cascading delete so entries in these tables will automatically be deleted at the same time.

Collecting external information  Many of the reports will need external information like name of ports, name of AS numbers etc. There will be several small scripts that collects this information on a regular basis and updates the database.

SQL Web interface
Implemented in PHP using FastTemplates to separate HTML from PHP code for easier translation to other languages. One important goal of the web interface is that it should be possible to add new report types without having to change the source code of the interface.

7.9 Flow record exporter
The flow exporter is a user level program that uses MAPI to produce flow records and that will act like a filter distributing the flow records to recipients like files or standard output as well as providing netflow/ipfix export.

7.9.1 Flow record specification
The flow record transport format through MAPI is the netflow version 9 format as documented in the relevant IETF draft.

This program must support the default format and may support the the specification of the Flow Key and the Flow report fields via a Netflow v 9 record containing a Options Template specifying flow measurement parameters and a Template Record specifying the fields to be reported. This is done via the mapi commands:

scampi@ist-scampi.org     82     March 27th, 2003
mapi_apply_function(flow, FLOW_KEY, *template)

    specify the fields to be the flow key with a Template Record and the
    flow measurements parameters with a Options Template.

mapi_apply_function(flow, FLOW_REPORT, *template)

    specify the fields that are to be reported with a Templated record

7.9.2 Transport Protocols

Netflow v5-8 uses UDP to transport flow records to a collector. This will result in
packets being dropped by intermittent network failures and not being retransmitted,
whereas the burden on the exporter for buffering is modest. There is a proposal to
transport Netflow v9 with PR-SCTP a partly reliable transport protocol where you
could send templates on a reliable stream and the data on a partly reliable stream
both belonging to the same source ID will be merged by the collector. SCTP is still
under development on the linux and BSD platforms and TCP was recommended as
a temporary solution by the last IETF ipx-meeting. This program MUST therefore
support UDP and SHOULD support TCP and MAY support SCTP.

When using UDP, the template packets must be resent periodically in case of loss.
The collector should buffer data packets until the first successfully transmitted template
is received. A new template should be resent every 10 until 100 and then per 100
packets.

7.9.3 Usage

The syntax of the netflow export command is as follows:

mapi-flow-export <option>...

-trace <selection> specifies a filter for traffic
to be include in this trace
default selection is all

-type <type> type of records to be produced (netflow5,
netflow9, ipfix)

-format <format> format of output (text, raw)
text format will print one record per line separating
fields with a separator
show format IP-addresses and portnumbers as expected and
other fields in hex

/fs <chars> This fieldseparator is used when dumping
records in text format.
Default value is ",,",.
-dest <url> where to place output: file://filename
    or protocol://address:port
  protocol = (TCP, UDP, SCTP). Default port (2055). Default
destination is standard output.

-flow-key <field-list> a list of fields that
  comprise the key for a flow-record
  field-list = { name | offset } / length, ....
  offset is from the start of the IP-packet in bits,
  name is from the Netflow v9 template, length in bits
  default flow-key is netflow v9 standard 7-tuple

-flow-fields <field-list> a list of fields
  to be included in the flow record (ref flow-key)
  default field-list is as netflow v9

The program will reformat flow records from the version 9 template format to other
formats as given by the type option.
Chapter 8

Kernel Software

8.1 The Click module

Middleware allows to perform complex parts of the monitoring chain in software, which aren’t supported in the hardware. The Click modular router can be compiled as a Linux kernel module, called `click.o`. It can capture packets from network devices before Linux gets a chance to handle them, send packets directly to devices, and send packets to Linux for normal processing.

The module uses the `/proc` filesystem for its API. It creates a number of files under `/proc/click`, some read-only and some read/write (writable by the superuser). You control the module by writing to these files. A router configuration can for example be installed by writing it to `/proc/click/config` or `/proc/click/hotconfig`. The configuration can use most of Click’s element classes. Several element classes control how the module receives and transmits packets: `FromDevice(n)` and `PollDevice(n)` steal packets from devices before Linux processes them, `ToDevice(n)` sends packets directly to devices, and `ToLinux(n)` sends packets to Linux for normal processing. Removing the module or installing a null configuration will restore your machine’s default networking behavior.

8.2 The Back-End Modules

8.2.1 Commodity Interfaces

General description

The fundamental difference between MAPI and existing APIs is that MAPI provides an efficient way of applying functions to packets. These functions can be used for several cases ranging from the trivial case of simply counting packets, up to the advanced case of reassembling a TCP/IP stream. Initially, MAPI will provide a number of predefined functions which is more frequently used by applications willing to monitor traffic. Such functions are the functions mentioned above and some others responsible for counting
total bytes, sampling packets, searching the packets for a pattern, organizing packets into flows, logging packets, trajectory sampling, reassembling IP, UDP/IP, as outlined in appendix C.

In addition to functions, MAPI also supports BPF filters. A BPF filter can be applied before any function is called. In this case if a packet does not pass the filter then none of the functions will be called. So, suppose an application wants to count all packets for a specific port (say port 80) coming from a specific source IP address (say 192.168.0.1). It has to do two things: first, apply the BPF filter which will pass only the requested packets and then apply the function responsible for counting packets as shown in figure 8.1.

Functions will have the ability to change the contents of a packet or even drop a packet. An example is the following: suppose an application wants to see if anyone is trying to gain root access to the machine and if so, log these intruding network packets and analyze them later. Initially, the application will apply the function which is responsible for TCP/IP reassembly to incoming packets. This function will drop retransmitted packets, defrag IP fragments etc. Then it will apply a function which is responsible to search all packets for a pattern su root and drop the packets that mismatch. Lastly, it will apply the function which logs all packets coming to it. This function will receive only the packets that passed the previous function as shown in figure 8.2.

To continue with, it is time to see the route that a packet arriving at NIC follows, until it reaches user space. First, it is transferred from NIC memory to kernel space through DMA or memory map. Then the BPF filter is applied and if the packet is not rejected it eventually reaches MAPI. There, all functions are applied and if the packet is not dropped it is sent through the socket interface to user space as shown in figure 8.3.

MAPI will provide an easy way of adding new functions without the need of restarting computer or even the need of recompiling the whole MAPI. The only things that the user will have to do in order to load his/her function into MAPI is to compile

scampi@ist-scampi.org

86

March 27th, 2003
8.2. THE BACK-END MODULES

Figure 8.2: Example of cooked network flows.

Figure 8.3: Data flow of packets destined to user space.
her/his function and execute a command. In the Linux operating system this function­
ality will be implemented with the use of modutils package (insmod, modprobe ,
). Of course, modutils will have to be appropriately adapted because currently only
the superuser (root) can use them. The modified modutils will let only authenticated
users to add/remove functions and of course one user could not remove or overwrite
the functions of another user. Also, predefined functions will be loaded and un­loaded
automatically from the operating system kernel. So, when an application requires a
function from MAPI, MAPI will check to see if this function is already loaded, if
not MAPI will ask kernel to load the function. On the other hand if MAPI observes
that a function has not been used for a long time it will unload the function and free
the memory. It is obvious that this functionality in order to be implemented requires
support from operating system kernel (Linux, supports this functionality).

MAPI functions

The functions that will be applied to packets are nothing more than common C functions
that treat packets as arrays of bytes. These functions are organized as a linked list. Every
time a packet is received by the network interface (NIC), is handed to a function that
iterates over the list of functions and passes the packet as an argument to every function
of this list, provided that the packet passed the BPF filter (if anyone). As mentioned
above the function can drop/modify the packet if it wants to. In case the functions
following in the list are not called. The effect is that the position of the function inside
the list does matter. Until know we have seen how functions will collect statistics
e tc but we have not discuss how applications will register their willingness to apply a
function to MAPI. This will be done through the ioctl system call. With this system call
a function will be able to inform MAPI which function/BPF filter wants to be applied
to incoming packets and specify a number of configuration options. Also, it can inform
MAPI that it wants the results of a function or that it is not interested anymore and it
wants the function to be removed. A simple scenario for an application that uses MAPI
to search packets for a specific pattern is the following

- The user process instructs the kernel to search the received packets for a pattern
  and hand to it anyone that matches the pattern using an ioctl (optionally it can
  use a BPF filter).

- The process blocks in a read system call (waiting for such packets to arrive).

- The MAPI searches the packets that pass the BPF filter (if anyone) and hands to
  the process the matching ones.

- The process can use an ioctl to see some statistics kept by the kernel and say
  the kernel to stop searching for the pattern because it is not interested any more.

8.2.2 Special-purpose Adapters

Special-purpose adapters are different from commodity interfaces in that they have
been specially designed to capture packets at high speeds and transfer them to the host
8.2. THE BACK-END MODULES

system with as little overhead as possible. Adapters like the DAG cards and COMBO6 both use memory mapped zero-copy techniques to transfer packets directly to the user. Special-purpose adapters can usually capture only the header of packets, a user specified part or whole packets. As speed of the network increases most special-purpose adapters will have some built in intelligence that can be used for processing packets and reduce the number of packets transferred to the host system. Exactly what type of processing that can be done by an adapter will vary greatly depending on the type of adapter. Some functions that might be available are sampling, filtering, string searches and if the adapter is more advanced and have enough memory it might also be able to generate flow records.

All of these features must be controlled by a kernel driver. The kernel driver tells the adapter where in memory to write packets, configures how much of packets that should be captured and also configures the available functions for packet processing. For adapters that support memory mapped zero-copy techniques for transferring packets to the user, there is little gain in doing processing in the kernel.
Chapter 9

MAPI Definition

9.1 Distributed MAPI

Every application addresses the monitoring platform throughout the MAPI. An application uses the functions defined in the MAPI interface to configure the monitoring agent and retrieve results from it. Whether this agent is running locally or remotely is of no concern to the functionality of the application. ¹ The MAPI should be completely transparent to the end user. To accomplish this, the MAPI (client-side) has to communicate transparently with the monitoring agent (server-side).

Figure 9.1 illustrates the remote MAPI process. An application configures a monitoring job by using MAPI functions. When calling these functions on the client-side, a graph that represents this monitoring job is built. Next, this graph is sent to the monitoring agent over Linux sockets. Linux sockets only transfer text, so prior to transmitting the graph, it has to be serialized. This serialization process takes a graph and translates it to the corresponding XML representation. On the server-side, the monitoring agent parses the XML representation of the monitoring job and rebuilds the original graph representation. Then, it merges the monitoring job with the already configured jobs.

9.2 MAPI

Most applications will interact with SCAMPI through its Monitoring API (MAPI). The MAPI provides applications with the following functionality:

- Passive monitoring:
  - access to network packets (passive monitoring)
  - ability to install filters to reduce the number of packets delivered to end applications

¹ Note, however, that if an agent is running remotely, the application will probably notice a performance penalty. The penalty will increase with the amount of monitoring data that the application would like to receive.
- ability to invoke pre-defined functions to operate on packets
- ability to define and install new functions that will operate on packets

- Active Monitoring
  - ability to initiate active measurements
  - ability to receive results/packets of active measurement

- Job control: ability to submit jobs to be executed

- Remote Access:
  - ability to invoke monitoring functions on remote computers

Active Measurements

As observed in 7.6.2 the active measurements can be split in two distinct actions:

- creating an active flow at the sending side
- creating the corresponding passive capturing of this flow at the receiving side and add the functions required for analyzing the annotations in the active probe packets (or the flow can be exported and analyzed more in detail by the applications).
The sending side should configure an active flow with the following call:

```c
am_flow fd = mapi_create_am_flow (am_t flowspec);
```

where the flow specification is a structure containing the source IP-address, destination IP address, source UDP port and destination UDP port to be used in the active measurement packets. Observe that the same specification can be used as a classification rule for the receiving side. Extra options for the active flow, like average packet length and inter-arrival times of the flows can be set using:

```
the option values? Steven?
mapi_set_am_flow_option(am_flow fd, int option_name,
int option_value);
```

In order to start and stop sending packets, the following functions are used:

```c
mapi_start_am_flow (am_flow fd)
mapi_stop_am_flow (am_flow fd)
```

It is up to the application running on top of the MAPI to make sure that sending and receiving side are configured correctly, and that they don’t interfere with the normal network operations (e.g. by using IP-addresses and ports of hosts and connections active in the network). The resource allocated to the flow at the sending side can be released with:

```c
mapi_close_am_flow (am_flow fd)
```

Functions to be implemented for this at the receiving sides are:

- Minimum, maximum and average one-way delay of the captured flow.
- Minimum, maximum and average one-way loss of the captured flow.
- Minimum, maximum and average one-way delay variation of the captured flow.

### 9.3 Passive Monitoring MAPI

Each Application Programming Interface should provide its users with a suitable abstraction which, on the one hand, is simple enough for humans to use and understand, and on the other hand, is powerful enough to express complex user requirements. If we look at successful APIs over the last decades, we will see that they provide simple, yet powerful abstractions that decouple programmers from the underlying hardware that is being used to implement these abstractions. For example, consider the file system programming interface, which was originally defined for the UNIX operating system [33]. The file system interface provided the `file` as the single most fundamental abstraction. A file was defined to be nothing more than a (potentially) long sequential stream of bytes, something like a large non-ended array. Users are able to create, open, close, read, and write files. By storing data in files as a long sequence of characters, users decouple the
data processing (which is a high-level operation) from their underlying storage media (which is a low-level operation). Indeed, over the last 30 years, the file system API has changed very little since its original conception, while the media used to store the data have changed more than four orders of magnitude. For example, the state-of-the-art magnetic storage in the early 70’s were 5-Mbyte disks (in the original PDP-11 where UNIX was initially deployed) and magnetic tapes. Currently, commodity storage systems are composed of 100-Gbyte disks which are expected to grow significantly larger in the near future. ²

We think that the MAPI should provide an equally simple, yet powerful abstraction: the network flow. A network flow is defined to be a sequence of packets that satisfy a given condition. For example, the simplest flow one can think of, is the flow that is composed of all network packets. Another flow may be composed of all packets directed to a particular web server. Another more complex flow may be composed of all packets sent from a given source IP subnet to a given destination IP subnet that use the TCP/IP protocol and have the SYN flag set. Note that this definition of flow differs from the traditional that usually refers to the traffic between two hosts, or even between two hosts using specific ports.

9.3.1 Creating and Terminating Network Flows

Each flow will be identified by a unique id: the flow descriptor. Users will be able to define flows, and operate on flows. For example, the following call creates a new flow:

```c
flow_descriptor fd =
    mapi_create_flow(char *device_d, condition *c, mode m)
```

The flow, defined by the `flow descriptor` `fd` consists of all packets that arrive in the device `device_d` (which can be either a monitoring device or a trace file), and satisfy condition `c`.³

Network flows come in three modes as defined by argument `mode m`: raw, cooked, and hierarchical. In the raw mode, the flow consists of all network packets that satisfy the condition `c`. These may include fragmented packets, retransmitted packets, out-of-order packets, etc. The packets in the raw mode are stored (and transmitted to the monitoring application) unmodified in the order of their arrival. In the cooked mode, on the other hand, the incoming packets (that satisfy the condition `c`) are processed according to the protocol stated in the packet header. Such protocols maybe TCP/IP, UDP/IP, etc. For example when processing TCP/IP packets in cooked mode, fragmented IP packets are combined, retransmitted packets are filtered out, and in general packets are reassembled into a data stream. That is, in the cooked mode the incoming network packets are turned into the data stream that would normally be presented to the socket layer. In the cooked mode, users also define a block size, which is the size of the chunk of the data they want to receive. For example, if the user defines a flow in cooked mode

²To demonstrate how the UNIX file system API survived over the decades, we should cite Ken Thompson, A.M Turing Award recipient and co-inventor of UNIX, who when was asked what he would do differently if starting UNIX over again he said “I would spell the ‘creat’ system call with an e”.

³The language for expressing conditions is explained in Appendix B.

scampi@ist-scampi.org

94

March 27th, 2003
and a block size of 64 Kbytes, the system will reassemble all the received packets into a data stream and chop the data stream into 64-Kbyte-large chunks. These chunks will be returned to the monitoring applications when they request them. If the user does not define a block size, the system will use the default block size. The block size along with all options of a network flow can be adjusted using the following call:

\[ \text{mapi_set_flow_option(flow_descriptor fd, int option, void * value)} \]

One the monitoring application set all the options, it will connect to the flow in order to start receiving network packets and/or network statistics:

\[ \text{int mapi_connect(flow_descriptor fd)} \]

Besides creating a network flow, monitoring applications may also close the flow when they are no longer interested in monitoring this flow:

\[ \text{mapi_close_flow(flow_descriptor fd)} \]

After closing a flow, the system releases all the structures that have been allocated for the flow.

Network flows allow users to organize the packets they are interested in monitoring into separate streams, and thus be able to treat them differently. For example, in most cases, users are interested in monitoring several sources of packets, and for each source of packet they are probably interested in monitoring different properties. Assume for the moment network administrators who may be interested in several flows at-a-time: they may be interested in observing the bandwidth consumed by peer-to-peer file sharing systems that may be running, while at the same time they may be interested in monitoring for Denial of Service attacks on their web server. On top of that, their site may also participate in a trajectory sampling experiment that samples and records a small percentage of packets [10]. Organizing these three different monitoring activities as separate flows, allows users (i.e. the administrators) to identify them, to isolate them, and to treat them differently.

Even more important than neatly separating different monitoring activities, network flows allow users to focus on different activities at different times. For example, during a DDoS attack, an administrator may decide to ignore the applications that measure the bandwidth usage of peer-to-peer systems and launch more fine-grain DDoS attack monitoring activities in order to pinpoint and isolate the attack. When the DDoS attack is over, the administrator may decide to stop some of these fine-grain DDoS attack monitoring activities and resume its usual peer-to-peer bandwidth usage monitoring.

### 9.3.2 Reading packets from a flow

Once a flow is established, the user will probably want to read packets from the flow. Packets can be read one-at-a-time using the following blocking call:

\[ \text{packet *p = mapi_get_next_packet(flow_descriptor fd)} \]

---

4 More information about network flow options can be found in appendix D.2.
If the user does not want to read one packet at-a-time and possibly block, (s)he may register a callback function that will be called when a packet to the specific flow is available. The following call invokes the callback handler for each packet that arrives in the network flow, and for the next packets.

```c
int mapi_loop(flow_descriptor fd, int cnt, mapi_handler callback)
```

### 9.3.3 Applying functions to Network Flows

Besides the neat arrangement of packets, network flows allow users to treat packets that belong to separate flows in different ways. For example, a user may be interested in *logging* all packets of one flow (e.g. to record an intrusion attempt), in just *counting* the packets and their lengths of a second flow (e.g. to count the bandwidth usage of an application), and in *sampling* the packets of a third flow (e.g. to find the most frequent network destinations). The abstraction of the network flow allows the user to clearly communicate to the underlying monitoring system these different operations. To enable users to communicate these different requirements, MAPI will enable users to associate functions with flows. This association implies that the functions will be applied to each packet of a flow. For example, a user may only want to count the packets that belong to one particular flow. In this case, the user will associate a counter function with this flow. Each packet that arrives in the flow will invoke this function which will just increment a counter. As another example consider a user who wants to sample every tenth packet of a network flow. Then, (s)he will be able to associate a sampling function with this flow. Each arriving packet will invoke this function, which will discard nine out of every 10 packets. The following is an example of asking the system to apply function to all packets of network flow.

```c
mapi_apply_function(flow_descriptor fd, function f,...)
```

The SCAMPI monitoring system will provide several *predefined* functions that will probably cover most of the network monitoring needs of ordinary users. For example, there will be a function that counts all packets in a flow. Another function will sample one out of every packets. The sampling may be *probabilistic* or *deterministic*, depending on the value of argument mode. There will also be functions that count various traffic metrics, like bandwidth, fragmented packets, etc. There will also be parameterized hashing functions that will take arguments that the user may define. Based on the value of the hashing function, the packet may be dumped or not. Although these functions will enable users to process packets, and compute the network traffic metrics they are interested in, without receiving the packets in their own address space, they must somehow communicate their results to the interested users. For example, a user that will define that the function will be applied to all packets of a flow, will be interested in reading what is the number of packets that have been counted so far. This can be achieved by allocating a small amount of

---

5 inspired from the libpcap library
6 For a current list of these functions have a look at appendix C.
7 Functions can be applied in *cooked* mode as well, after the re-assembly (i.e. cooking) of packets is done.

scampi@ist-scampi.org 96 March 27th, 2003
memory, or a data structure to each network flow. The functions that will be applied
the packets of the network flow will write their results into this data structure. The
user who is interested in reading the results will read the data structure through the
following call:

\[
\text{mapi\_read\_results}(\text{flow fd, int function, void * result})
\]

After the return of this call the result structure will be populated with the appropriate
values computed by the functions. \(^8\) If users apply more than one function to the
packets of a flow, they may choose to read the results of only one function at-a-time
distinguishing them with the int function argument. For example, if the users
apply both the PACKET\_COUNT, and the BYTE\_COUNT functions to a network flow,
the following call read the results of the PACKET\_COUNT function only:

```c
struct packet\_count\_results pr ;
mapi\_read\_results(fd, PACKET\_COUNT, &pr) ;
printf("Number of packets counted for flow %d is %llu \n",
    fd, pr.packets)
```

Similarly the following code prints the the number of bytes of the packets received
by this flow:

```c
struct byte\_count\_results pr ;
mapi\_read\_results(fd, BYTE\_COUNT, &pr) ;
printf("Number of packets counted for flow %d is %llu \n",
    fd, pr.bytes)
```

### 9.3.4 Packet formats

Function get\_next\_packet returns a pointer to a packet. The exact contents of
this pointer and the format of the data it points to, may vary depending on the end-user
preferences. For example, the user may want to receive the data in simple ASCII form
much like the pcap library does. On the other hand, the user may choose to receive
the data encoded according to some compression protocol that the capture mechanism
may use. To allow for all these (and possibly future) options, packet is nothing more
than an unsigned char. The format of the data it points to depends on the encoding
scheme that has been used. The actual format of the monitoring records can be defined
using the option monitoring\_record\_format as explained in appendix D.

### 9.3.5 Flow Records

MAPI can collect statistics on a flow using Flow Records. This is achieved with
two variables. FLOW\_RECORD is a logical variable which when set to TRUE enables

---

\(^8\) Alternatively, applications can configure the network flow by writing to it, and get information back by
a read operation. The statistical information can be encapsulated in a sort of "pseudo" stream coming from
the socket. The idea is that we can then use reads to "poll" the stream, or use select to be notified when
information is available. Instead of the synchronous read operation in both approaches, a callback from the
MAPI to the requesting applications is possible as well.

scampi@ist-scampi.org 97 March 27th, 2003
the collection of Flow Records. `FLOW_RECORD_TYPE` can take values from the set `{IPFIX, NETFLOW_V5, NETFLOW_V9}`, selecting the format of the records that MAPI will maintain.

To read the collected flow records, an application can use the `read_flow_record` function:

```c
void * mapi_read_flow_record(flow_descriptor fd)
```

that returns a flow record for the specified flow.

### IPFIX flow record generation

There are several cases where users are interested in viewing traffic statistics without focusing on any specific previously known network flow. In addition, users may be interested in viewing traffic data from several different points of view. To cater to these user needs, MAPI defines **hierarchical** network flows. A hierarchical flow differs from regular network flows in that it is composed of several **sub-flows**.

The notion of a sub-flow corresponds to the IPFIX \(^9\) and NetFlow notion of flow. A sub-flow is defined to contain all packets (within the parent flow) that have a common 5 or 7-tuple of protocol number, source IP address, destination IP address, source port, and destination port. If SCAMPI is able to see input and output interfaces, these are also included in the tuple.

The MAPI implementation is able to report to statistics about sub-flows to MAPI clients. It does this by returning IPFIX flow records. IPFIX standardisation is still in progress,\(^{10}\) but the record format is expected to be Cisco NetFlow v9 with minor changes.

Clients may collect IPFIX flow records directly over the MAPI. One particular client is the Flow Record Exporter, which can distribute flow records to recipients like files or standard output as well as provide NetFlow/IPFIX export via the network.

MAPI provides flow records in IPFIX format only. Clients which need a different format must go via the Flow Record Exporter.

### Flow key specification

The following properties may be used as part of the key for distinguishing flows:

- Incoming interface
- Outgoing interface
- Source IP address
- Destination IP address
- IP Protocol type (TCP, UDP, ICMP,...)

---


\(^{10}\)It is likely that this section will have to be revised when the IPFIX standard is finalized.

scampi@ist-scampi.org 98 March 27th, 2003
9.3. PASSIVE MONITORING MAPI

- IP version number (IPv4, IPv6)
- Source port number
- Destination port number

For source address and destination address, separating by full match is supported as well as separation by prefix match.

Port numbers are only relevant for TCP and UDP flows.

Interfaces are not relevant where SCAMPI is deployed as a probe.

All meaningful combinations of the above may be used to distinguish IPFIX flows.

The client chooses the flow key by calling

```c
mapi_apply_function(flow, FLOW_KEY, *template)
```

There is a discrepancy between how the IPFIX draft and NetFlow V9 handle IP version number. NetFlow defines separate fields for IPv4 and IPv6 addresses. IPFIX defines address and version fields. We use the IPFIX terminology until it is known how this issue is resolved.

**Flow record specification**  
MAPI is able to include any combination of the following information fields in flow records returned to the client:

- Source IP address
- Destination IP address
- IP Protocol type (TCP, UDP, ICMP,...)
- IP version number (IPv4, IPv6)
- Source port number
- Destination port number
- Number of packets
- Number of bytes
- Type of service (IPv4)
- Traffic class octet (IPv6)
- Flow label (IPv6)
- Timestamp of the first packet of the flow
- Timestamp of the last packet of the flow
- Unique identifier of the observation point (the SCAMPI probe)
- ICMP type and code
CHAPTER 9. MAPI DEFINITION

- input interface
- output interface
- multicast replication factor
- average time between packet arrivals
- standard deviation of time between packet arrivals
- average packet size
- standard deviation of packet size
- a serial number

Input interface, output interface and multicast replication factor are not applicable where SCAMPI is deployed as a probe.

The client may choose to have all meaningful combinations of the above included in the flow records returned. The client chooses what to include by calling

```c
mapi_apply_function(flow, FLOW_REPORT, *template)
```

MAPI only maintains those statistics which some client has requested. Thus, administrators may avoid doing costly computations in a high data rate situation.

When the set of information fields being monitored is increased, because a new client asks for fields which were so far not monitored, all existing sub-flows are expired.

There are a number of differences between how the IPFIX draft and NetFlow V9 express the same information, e.g. differences in handling IPv4/IPv6. We use the IPFIX terminology until it is known how this issue is resolved.

**Implementation issues** The design has to be able to keep up with high data rates.

**Software structure** The IPFIX record generation task can be broken down to the following:

- Packet classification
- Statistics updating
- IPFIX record output

Conceptually, this can be grouped into one process which does classification and statistics updating, and another which periodically exports records for completed and expired flows. Both operate on a data structure consisting of

- IpfixFlowTable - a table of IpfixFlowRecords, hashed by IpfixFlowKey
- CreatedTimeList - a list of pointers to IpfixFlowRecords, ordered by time the flow record was created
- ActiveTimeList - a list of pointers to IpfixFlowRecords, ordered by time the flow record was last active
9.3. PASSIVE MONITORING MAPI

**Synchronization**  Implementing these two conceptual processes as separate operating system processes or threads is likely to degrade performance. This is because access to the shared data structures has to be protected by a mutual exclusion mechanism, and acquiring the lock will normally be a more expensive operation than actually classifying a packet and updating a record.

Classification/updating and record output should therefore be done in the same thread. The IPFIX flow record generator will present an interface to the rest of the MAPI implementation more or less as follows:

```
ipfix_process (scampi_flow, timestamp, packet, action)
```

where action can be one of UPDATE and OUTPUT. The environment invokes ipfix_process with UPDATE for each packet and periodically with OUTPUT.

**Classification**  Packet classification is done with a hash function. The function has to be cheap to compute and distribute flows well over the hash buckets. We propose the following function to compute a 32 bit hash:

```
for each possible key field:

    if field is a key field for the current scampi flow
        xor field value with hash
        left shift hash value 8 bits
```

IPv6 addresses are treated as 4 32 bit fields. The number of hash buckets needs to be high enough to avoid many different IPFIX flows hashing to the same bucket.

**Expiry**  sub-flows expire and become ready for output when any of the following happens:

- a TCP session is closed.
- it has been inactive for subflow_timeout seconds.
- it has existed for max_subflow_duration seconds.
- When a new client asks for fields which were so far not monitored, all existing sub-flows are expired.
- Finally, each hierarchical flow has a predefined maximum number for sub-flows: max_number_of_subflows. When a new sub-flow is created but there are already max_number_of_subflows active sub-flows, the one which has been inactive for the longest time is selected and considered expired.

scampi@ist-scampi.org  101  March 27th, 2003
Reading Dynamically Generated Flows

Once a hierarchical flow is created, the user is able to read the expired sub-flows using the following calls:

```c
sub_flow *p = mapi_get_next_subflow(flow_descriptor fd)
```

If the user does not want to read one expired flow at a time and possibly block, (s)he may register a callback function that will be called when a sub-flow expires. The following call invokes the callback handler for each packet that arrives in the network flow `fd`, and for the next `cnt` packets.

```c
mapi_subflow_loop(flow_descriptor fd, int cnt, subflow_mapi_handler callback)
```

### 9.3.6 Some Simple Examples

Once we have defined the functionality of the MAPI, let's see how we could use it to implement some simple monitoring examples. 11

In our first example we are interested in receiving the first 100 packets that are destined to port 80 (usually the port used for web servers):

```c
#include "mapi.h"
char * dev = "/dev/scampi"
packet * p ;
flow_descriptor fd ;

filter = "dst port 80" ;

// create a flow that consists of all packets
// destined to port 80
fd = mapi_create_flow(dev, filter, RAW) ;
mapi_connect(fd) ;
for (i = 0 ; i < 100 ; i++) {
    p = mapi_get_next_packet(fd) ;
    print_header(p);
}
```

In the following example we count the number of packets (destined to port 80) that have been seen by the system over a period of 10 seconds:

```c
#include "mapi.h"
char * dev = "/dev/scampi"
flow_descriptor fd ;
packet_count_results r ;
```

We should stress that our examples are being used for illustration purposes only. Our goal is to demonstrate the expressive power and simplicity of the MAPI, and not to propose optimal solutions. For the same reason, in the examples we have chosen to ignore possible errors and exceptions.

scampi@ist-scampi.org 102 March 27th, 2003
condition = "dst port 80";

fd = mapi_create_flow(dev, condition, RAW);
mapi_apply_function(fd, PACKET_COUNT);
mapi_connect(fd);
sleep(10); // sleep for 10 seconds
mapi_read_results(fd, PACKET_COUNT, (void *) &r);
printf("Read %d packets\n", r.packets);

In the following example we print the bandwidth usage per second for our web server (whose IP address is 139.91.191.150):

char * dev = "/dev/scampi"
flow_descriptor fd;
byte_count_results r;
int previous_byte_count = 0;

condition = "ip 139.91.191.150";

fd = mapi_create_flow(dev, condition, RAW);
mapi_apply_function(fd, BYTE_COUNT);
mapi_connect(fd);

while(1) { //forever
    sleep(1); // sleep for one second
    mapi_read_results(fd, BYTE_COUNT, (void *) &r);
    printf("%d bytes/sec\n", r.bytes-previous_byte_count);
    previous_byte_count = r.bytes;
}

Suppose now that we would like to print separately the incoming traffic to our web server from the outgoing traffic from it. We will just create two network flows: one for the incoming traffic and one for the outgoing traffic.

char * dev = "/dev/scampi"
flow_descriptor in_fd, out_fd;
byte_count_results in_r, out_r;
int in_previous_byte_count = 0, out_previous_byte_count = 0;

in_condition = "dst port 80 and dst ip 139.91.191.150 ";
out_condition = "src port 80 and src ip 139.91.191.150 ";

in_fd = mapi_create_flow(dev, in_condition, RAW);
out_fd = mapi_create_flow(dev, out_condition, RAW);
mapi_apply_function(in_fd, BYTE_COUNT);
mapi_apply_function(out_fd, BYTE_COUNT);
CHAPTER 9. MAPI DEFINITION

mapi_apply_function(out_fd, BYTE_COUNT) ;
mapi_connect(in_fd) ;
mapi_connect(out_fd) ;

while(1) { //forever
    sleep(1) ; // sleep for one second
    mapi_read_results(in_fd, BYTE_COUNT, (void*) &in_r) ;
    mapi_read_results(out_fd, BYTE_COUNT, (void*) &out_r) ;
    printf("%d incoming bytes/sec\n",
           in_r.bytes-in_previous_byte_count);
    in_previous_byte_count = in_r.bytes ;
    printf("%d outgoing bytes/sec\n",
           out_r.bytes-out_previous_byte_count);
    out_previous_byte_count = out_r.bytes ;
}

The reader will notice that the user application just tells the MAPI what is interested
in measuring, and after that it sleeps most of the time. It just wakes up every second to
read and print the number of bytes that have been observed.

In the next example we write the code for trajectory sampling [10]. Trajectory sam­
pling, is applied to all packets in a set of routers in order to determine the trajectories
that the packets follow through those routers. To do so, first the packets are deter­
ministically sampled by applying a given hash function to all of them and choosing those
packets in which the result of the hash function is a given value. These selected packets
are then sent to a central point which determines the route of the packets through the
routers. To reduce the information sent to this central point, a second hash function is
applied to the sampled packets, and the result of this second hash function is sent to
the central point. The purpose of this second hash function is to create a signature that
(almost) uniquely identifies each packet.

#include "mapi.h"
char * dev = "/dev/scampi"
flow_descriptor fd ;

fd = mapi_create_flow(dev, NULL, RAW) ;
//100 and 2 are arguments to the TRAJECTORY_SAMPLING_H function
mapi_apply_function(fd, TRAJECTORY_SAMPLING_H, 100, 2) ;
mapi_connect(fd) ;
while(1) {
    // returns only sampled packets
    packet = mapi_get_next_packet(fd) ;
    signature = hash(packet);
    sent_to_central_node(signature) ;
}

As we wan see, the application defines a ow consisting of all packets. Then, it
tells the system to apply a hash function to those packets and to discard all packets that do not match the value. Then the application just reads the (non-discarded) packets of the flow, computes a second hash function on them, and sends the resulting signature to a central node for processing.

9.3.7 Some More Complex Monitoring Examples

Find the 10 most traffic-intensive applications (first approach)

Let’s consider a somewhat more complicated monitoring example. Suppose that a system administrator would like to find the top-10 applications that generate the most traffic, as well as the exact amount of traffic per such application. In a traditional libpcap-based system, the administrator would have to construct an application that receives all network packets in user space, partitions them in bins according to their port number and find the 10 bins with the largest number of packets. We believe that this task can be simplified using our MAPI as follows: the administrator will define a network flow that consists of all packets, and will define a function that samples one out of every 100 (for example) packets. The application will receive these packets and based on their port number will find the 10 applications that generate the most traffic (with high probability). Then the application will create one flow for each of those applications and measure the amount of traffic for each such flow.

Find the 10 most traffic-intensive applications (second approach)

The MAPI provides a predefined function called `packet_distribution_mask(((int16_t *) func (void *packet)))`, which can be used to find distributions of packets based on specific header/payload field values which are the result of function `func`. Thus, if we assume that we have defined the function `dst_port((void *packet))` which returns the destination port of IP packet `packet`, then, if we apply function `packet_distribution_mask(dst_port)` to all packets of the monitored traffic, then the distribution of the outgoing traffic based on ports will be easily found on the field `distribution[65536]` of the structure `struct packet_distribution_results`. The user can find the complete distribution of the (outgoing traffic) per port by reading the values of array `distribution[65536]`. The 10 highest values represent the top-10 applications.

Which files did a user sent using the Gnutella file sharing application

Peer-to-peer traffic is getting increasingly larger. Actually, it has been reported that in several education institutions p2p traffic is larger than web traffic [19]. Therefore, several administrators are interested in finding which users send the most files, or which files are sent by which users. Suppose that an administrator is interested in finding out which files were sent by the user using the computer with IP address 192.89.93.27 in the Gnutella p2p system. To do so, the administrator will create a cooked network flow...
consisting of all Gnutella packets (i.e. all packets send and received over the Gnutella port send by IP address 192.98.93.27). Then the administrator will do a substring search to find what are the names of the files sent over this connection.

Find the average duration of a UDP session

Suppose that we are interested in finding the average duration of a UDP session. Contrary to a TCP session, a UDP “session” does not have a clearly labeled start and end packet. Thus, a network flow is usually defined as a train of packets that arrive within a short time interval of each other [7]. In this case, we can define a hierarchical flow composed of all UDP packets. Each UDP session is nothing more than a sub-flow. The monitoring application will enter a loop receiving one expired flow after the other. For each flow it will find the duration of the flow (i.e. duration of the UDP session) by subtracting subflow start time from the subflow end time. Calculating the average of such durations is then straightforward.

Find the average duration of a TCP session

Suppose that the user would like to find the average duration of a TCP connection. As a first cut, this can be done in the same way as described previously in the UDP sessions. However, in the paragraph we will describe on more way. The user may create one flow with the starting packets of a connection (i.e. those that have the SYN flag set), and one flow with the closing packets (i.e. those that have the SYN/FIN flag set). The user application in turn will correlate packets from these two flows in order to find the TCP sessions and their duration.

9.3.8 Implementation issues for MAPI

The described MAPI can be easily implemented on top of today’s systems at various levels. It can be implemented in user-space (on top of libpcap), in kernel-space (in cooperation with BPF or LSF), or (some subsystems of MAPI) at specialized hardware. For example, MAPI can be implemented on top of libpcap as follows: libpcap will provide all packets, which will be later categorized (by the MAPI implementation) into network flows.

MAPI can also be implemented inside the operating system kernel as a add-on to the Berkeley Packet Filters [21] or Linux Socket Filters [12]. In this case, the operating system will be able to do a significant amount of packet processing in kernel space without ever delivering the packets in user level. The MAPI can be also implemented in a system with special-purpose hardware, like a special-purpose network interface. The special-purpose network interface will implement some of the MAPI’s functionality, probably the filtering and the categorization of packets into flows. Finally, the MAPI

\[^{13}\text{Gnutella (and clones) use the port number 6346. Similarly, several p2p applications use various available ports. For example, WinMX uses ports 6257 and 6999, Morpheus/KaZaa use port 1214, eDonkey uses port 4662 and 4665, and Direct Connect uses port 412. Although the method proposed in this example applies only to peer-to-peer applications that use widely known ports, peer-to-peer applications that use dynamic ports can still be tracked by analyzing both port numbers and packet payloads.}\]

scampi@ist-scampi.org

106

March 27th, 2003
can also be (partially) implemented on top of traditional systems that can communicate with intelligent routers that keep statistics about the incoming traffic. In this case, the MAPI will provide only the functions already implemented by the router. It will simply not provide the rest of the functions.

MAPI for High-Speed Monitoring

The MAPI described has been especially designed for easy, and high-speed network monitoring. For example, user applications are allowed to define network flows which clearly separate the various sources of traffic. Thus, the user will be able to treat each flow of traffic differently, and focus of different flows at different times. In addition, users are able to define functions to be executed on each packet. Thus, the system may choose to execute these functions in the best possible place: the user-space, the kernel-space, a co-processor, or some special-purpose hardware.

MAPI for clock control

MAPI has to deal with clock for timestamp generation, whether the clock is implemented in software (kernel) or in hardware (on the adapter). A hardware clock on the adapter can be accessed via a mapi_ioctl call.

Timestamps are represented by the timespec structure adopted from nanokernel implementation:

```c
struct timespec {
    long tv_sec; /* # of UTC seconds since 0:0:00, 1970, January 1 */
    long tv_nsec; /* nanoseconds since beginning of current second */
}
```

Functions

mapi_gettime (struct timespec *ts)

Function returns current time of the clock. Function is not used for timestamp generation, as it is done earlier and automatically in adapter or by driver.

mapi_settime (struct timespec *ts)

Function sets the adapter clock to the given value.

mapi_adjtime (long offset, long skew)

Function adjusts clock rate and offset in case clock is not disciplined by NTP process or external time source.

mapi_discipline (int par)

Function start and stops disciplining of the adapter clock by external signal (for instance PPS -Pulse Per Second- signal from GPS receiver)
Management

For management purposes MAPI will provide information about the status of the monitoring hardware that is used and basic usage statistics. All management information will be available through a single call:

management_info *result = get_management_info(int info_type)

where info_type specifies the type of management information that should be returned.

9.3.9 Options for MAPI Interfaces

A SCAMPI application can be implemented as an SNMP client, which contacts a SCAMPI SNMP agent. This SNMP agent offers exactly the same functionality as the MAPI (configuration of the click building blocks to receive the desired outputs). Results can be sent to the SNMP client by either SNMP traps (asynchronous) or as a response to an SNMP GET_REQUEST. However, SNMP has a few drawbacks in this context: (i) since SNMP communication makes implicit use of the UDP protocol, communication is not reliable, (ii) SNMP does not provide any support for transactional operations, (iii) a lot of implementations which suffer from buffer overflow have been reported, (iv) the SNMP agents can only contact the clients by means of traps, and (v) SNMP only provides very limited support for authentication. Authentication is available in SNMPv3. However, very few implementations of SNMPv3 exist. A SCAMPI application can also be implemented as a PDP (Policy Decision Point), which contacts the SCAMPI PEP (Policy Enforcement Point) via COPS (Common Open Policy Service). However, the deployment of a COPS interface to the MAPI poses a few problems: (i) there is currently no support in e.g. HPOpenview for managing COPS PEPs., (ii) there is only a small COPS codebase in the Internet community and (iii) COPS implementations are rather complex.

9.4 MAPI vs. Other Approaches

Current network monitoring tools use a wide variety of languages and environments, including libpcap [20], Berkeley Packet Filters [21], CoralReef [16], Linux Socket Filters [12], IPFIX [30], and NETFLOW [38]. The notion of network flow, has been introduced in monitoring systems several years ago. For example, a flow has been defined as a “sequence of related packets sent from a source to a unicast, anycast, or multicast destination(s)” [31]. The notion of a flow has also been widely used by CISCO in its various network monitoring products including NetFlow [38]. A similar notion of a flow is currently being standardized in IPFIX. However, contrary to other definitions, MAPI gave the “network flow” a first-class status. For the first time and contrary to previous proposals and implementations, flows in MAPI have a name, they have an identification. Users can access their flows by their id. Users can create flows, they can destroy (close) flows, they can read packets from a flow. Users can also apply functions to flows; they can sample packets from flows; they can count packets, bytes

scampi@ist-scampi.org 108 March 27th, 2003
and various traffic statistics in a flow; in short they can operate on flows, much like they can operate in other programming abstractions like sockets, pipes, files, etc. Using these first-class flows users are able to perform a wide variety of new monitoring operations. For example, MAPI network flows enables users to develop intrusion detection systems that are based on full packet payload inspection. On the contrary, NetFlow, IPFIX, and related systems and proposals (to the best of our understanding) do not provide full content inspection, and thus they cannot be used to implement signature-based intrusion detection.

Besides the notion of the flow, MAPI shares functionality with previously defined network monitoring systems. For example, Berkeley Packet Filters [21] can filter monitored traffic much like the conditions of the network flows do. As another example, Linux Socket Filters [12] associate a socket with each filter, much like MAPI associates a network flow with one condition. However, no previous such filtering environment allowed users to fully express their real monitoring needs. Users with somewhat complex monitoring requirements usually completely bypassed the underlying filtering mechanism and re-implemented their filtering inside their applications much like the snort intrusion detection system does [34]. MAPI is the first network monitoring abstraction that uses a language rich enough to allow users express complex monitoring needs and enables the system to implement them as efficiently as possible. For the first time, users are able to tell the monitoring system that they want only those packets that match a virus signature; or that they just want a head count of the packets to their web server, achieving both expressive power, and increased efficiency. This expressiveness enables MAPI-based applications to efficiently perform a wide variety of applications that is not possible with systems such as pcap and LSF. For example, lets assume that we have an intelligent network adapter equipped with a programmable processor. Assume also that the monitoring application would like to sample the packets and receive one out of every 100 packets of the entire network traffic. User applications on top of MAPI have the ability to express this sampling requirement, which, in turn, MAPI may implement it in the programmable processor on the network adapter. Thus, only one percent of the network traffic will be transferred from the adapter to the host processor. On the contrary, applications running on top of pcap or LSF have no way of stating this sampling requirement. As a result, they will be forced to receive the entire 100% of the traffic, only to throw away 99% of it. Summarizing, by letting users express their needs, and by giving network flows a first-class status, MAPI enables the efficient implementation of monitoring applications on top of a wide variety of platforms.
Chapter 10

Summary and Conclusions

This document presented the system architecture and the components of the SCAMPI platform for network monitoring. The need for improved network monitoring arises from the widespread consensus among network operators, engineers and researchers that effective network monitoring is vital for performing informed network management decisions as well as for supporting the growing set of automated control mechanisms needed to make the IP-based Internet more efficient, robust and secure.

The rationale for our work is the increased pressure on existing architectures that has exposed limitations that are believed to be rooted in the basic abstractions used, in addition to increasing link speeds and the growing number and diversity of monitoring applications. As a response, we have designed a new architecture that has the following key characteristics:

- **an expressive programming interface**: we have designed the SCAMPI Monitoring API (MAPI) so that users can clearly express their needs to the underlying platform. In this direction, the main feature of MAPI is a generalized flow abstraction that allows users to tailor measurements to their own needs, recognizing that the information provided by the existing flow model is often either too specific or not specific enough for monitoring applications. Where necessary and feasible, MAPI also allows the user to trigger custom processing routines not only on summarized data but also the packets themselves. The expressiveness of MAPI also allows the underlying monitoring system to make informed decisions in choosing the most efficient implementation.

- **use of intelligent hardware**: SCAMPI provides a coherent interface on top of different lower-level elements; our current design considers the use of intelligent switches, high-performance network processors, and special-purpose network interface cards. Except for high-performance this also reduces the cost of cross-platform compatibility.

- **scalability through parallelism**: SCAMPI acknowledges that scalable network monitoring needs to explicitly honor parallelism and factor it into the architecture design. SCAMPI components exploit parallelism at several levels: from multiple
processing units in hardware (as in the IXP1200 network processor), up to the use of several sensors in hierarchical intrusion detection systems.

As networks continue to get faster at exponential speeds, we believe that the above dimensions will be of significant help in making a sense out of the colorful chaos of information that is currently flowing in our networks.
Appendix A

Predefined Click Modules

At the time of writing, the latest release of Click is version 1.2.4. This version comes standard with a huge library of elements that can be used in the SCAMPI project. Standard elements that are currently being used in the first prototype are:

- **FromDevice(DEVNAME, OPTIONS):** Captures packets from a certain network device.
- **MarkIPHeader(14):** Determines whether a packet is an IP-packet.
- **Tee(N):** Create N flows out of a single flow.
- **IPClassifier(PATTERN 1, ..., PATTERN N):** Classifies IP packets according to tcpdump-like patterns.
- **Counter(KEYWORDS):** Maintains statistics information about packet count and packet rate.
- **RandomSample(P, KEYWORDS):** Samples packets with probability P.
- **ToHostSniffers(DEVNAME):** Hands packets to any packet sniffers registered with Linux, such as packet sockets.

Other predefined elements can be found on the Click website at: [http://www.pdos.lcs.mit.edu/click/](http://www.pdos.lcs.mit.edu/click/).
Appendix B

Language for Network Flow Conditions

When a new flow is created, a condition is given among the arguments needed to create the flow. The condition is nothing more than a filter that enables users to express to the system the subset of traffic they are interested in monitoring. In this way, (i) the users receive only the information they need, and (ii) the system is able to reduce the amount of traffic sent in user-space and thus to improve its performance. The language that users use to express their conditions should be powerful enough to express most frequently occurring conditions, and at the same time it should be simple enough so as to be efficiently implemented. At the same time, it should be extensible so that users will be able to express requirements that we do not currently know.

Within SCAMPI we have chosen to use a dual approach to expressing the above conditions:

- We will use a known and frequently used language to express such filters, and
- We will enable users to further restrict the packets they are interested in observing through the application of functions to network flows.

Thus, the language that we will use to express conditions will be the one used by Berkeley Packet Filters [21]. It is interesting to note that the same language is being used in tcpdump [39], in Linux Socket Filters [12], and in libpcap [20].

The language allows users to test for equality, inequality, and range conditions for all known IP header fields. For example

```
host calliope.ics.forth.gr
```

returns all network packets sent to/from calliope.ics.forth.gr. Or,

```
dst port 80
```

returns all packets whose destination port is 80 (which usually corresponds to web traffic). If a user would like to trace all packets between local hosts and the hosts at the University of California at Berkeley the following command returns just them:

```
```
Actually, the user is also able to access bytes within packets using a array-like notation. For example,

\[ \text{ip}[2:2] > 100 \]

returns all IP packets whole length is greater than 100. The \( \text{ip}[2:2] \) returns the 2 bytes starting at the third byte of the IP header (i.e. the \( \text{ip}[2] \)). These two bytes represent the length of the IP packet.

Note that although the above language can test for quality of inequality all fields of an IP packet, it may still not be enough for very sophisticated users that would like to perform more complicated conditions. For example, some users may also want to test whether packets contain specific substrings (like in intrusion detection systems [34]), or they may want to perform sophisticated sampling, like in trajectory sampling [10]. We believe that this functionality can be implemented in SCAMPI on top of the above condition language by defining appropriate functions and by applying the functions to the packets of the network flow.
Appendix C

Predefined Functions for Network Flows

The MAPI implementation will provide a large set of predefined functions that will be applied to network packets. Such functions include:

PACKET_COUNT

structure of results:
struct packet_count_results {
  uint64_t packets ;
};

It keeps the number of packets seen by the flow so far.

PACKETS_IN_INTERVAL(unsigned long us)

structure of results:
struct packets_in_interval_count_results {
  struct timeval start ;
  uint64_t packets ;
};

It keeps the number of packets passed in a time interval. The length of the interval in microseconds is the input parameter \texttt{us}. The start of the interval is return in the results structure, along with the number of the packets observed.

PACKET_DISTRIBUTION_MASK(int offset, int mask)

structure of results:
APPENDIX C. PREDEFINED FUNCTIONS FOR NETWORK FLOWS

```c
struct packet_distribution_results {
    uint64_t distribution[65536];
};
```

This is a rather elaborate counting function that can be used to find the distribution of the length of the packets, the ports of the packets, etc. The function takes as arguments an offset and a mask. For each incoming packet the function takes a sequential number of bits starting at byte offset. This number of bits which is the value of mask should not be larger than 16. The resulting 16-bit number is then used as an index to the array `distribution` to increase the respected position.

```c
 PACKET_DISTRIBUTION_FUNCTION((int16_t*)func (void * packet))
```

This is an even more elaborate counting function that can be used to find the distribution of the length of the packets, the ports of the packets, etc. The function takes as arguments a function that operates on the packet. For each incoming packet the function computes a 16-bit value which is then used as an index to the array `distribution` to increase the respected position.

```c
struct byte_count_results {
    uint64_t bytes;
};
```

This structure keeps the sum of the lengths of all packets seen by the flow so far.

```c
BYTE_COUNT
```

```c
struct packets_in_interval_count_results {
    struct timeval start;
    uint64_t packets;
};
```

This structure keeps the number of octets observed in a time interval. The length of the interval in microseconds is the input parameter `us`. The start of the interval is stored as a `timeval`.
interval is return in the results structure, along with the number of the packets observed.

DISCARD

structure of results:
struct discard_results {
  uint64_t discarded_packets;
};

It discards the current packet. It is being used when users are not really interested in receiving packets, but are only interested in calculateing statistics in network flows.

SAMPLE_PACKETS(int sample_period, int mode)

structure of results: none

It samples packets from the network flow. All but one out of every sample_period packets are discarded and never sent to the monitoring application. If the mode is DETERMINISTIC, the sampling is deterministic. If mode is PROBABILISTIC, the sampling is probabilistic with samples taken from a uniform distribution. Sampling functions represent the most fundamental functionality of several monitoring environments, including sflow [29].

SUBSTRING_SEARCH (int offset, char *s, int len)

structure of results:
struct substring_search_results {
  long packets_matched;
};

It searches the body of the network packet to see if it contains string s. If not, the packet is discarded. The search starts from byte offset and lasts for len bytes.

ADDITIVE_HASH (char *packet, int offset, int len, int prime, int low, int high)
{
  int hash, i;
  for (hash=len, i=offset; i<len; ++i)
    hash = hash+packet[i];
  return (hash % prime);
}

structure of results: none

It computes the additive hash function. It keeps the packet if the resulting value is between low and high. Otherwise, it discards the packet.

scampi@ist-scampi.org 119 March 27th, 2003
APPENDIX C. PREDEFINED FUNCTIONS FOR NETWORK FLOWS

TRAJECTORY_SAMPLING_H
(char *packet, int len, int A, int r)

structure of results: none

It computes the $h(\phi(x))$ function as described in [10], and then takes the $h(\phi(x)) \mod A$. If the result is larger than or equal to $r$ the packet is discarded.

TRAJECTORY_SAMPLING_G
(char *packet, int len, int B)

It computes the $g(\phi(x))$ function as described in [10].

PACKET_SAVE
(int startbyte, int endbyte)

structure of results: none

It defines the portion of the packet that is to be reported to the application. It is useful to trace specific portions of the packets (such as application-level header structures). This functionality is similar to the tcpdump syntax for specifying bytes ranges to be traced (for example: tcp[20:10]). The tracing of disjoint portions is not allowed. If endbyte is 0, the function saves up to the end of the packet.
Appendix D

Man pages for MAPI functions

D.1 MAPI functions

```c
flow_descriptor fd =
    mapi_create_flow(char *device_d, condition *c, mode m)
```

`create_flow` is used for creating a new network flow of packets that match condition `c`. The flow of packets will be read from device `device_d`. Mode `m` defines the type of the flow: RAW, COOKED, and HIERARCHICAL. If successful, the function returns a non-negative integer `flow_descriptor fd`. If unsuccessful, the function returns a negative integer.

```c
int mapi_close_flow(flow_descriptor fd)
```

`mapi_close_flow` is used to close the flow defined by descriptor `fd` and deallocate resources. If successful it returns a non-negative integer. Otherwise, it returns a negative one.

```c
int mapi_connect(flow_descriptor fd)
```

`mapi_connect` is used to connect to the flow `fd` and start receiving information. The call may fail depending on the requestor’s privileges and requirements. If successful it returns a non-negative integer. Otherwise, it returns a negative one.

```c
mapi_set_flow_option(flow_descriptor fd, int option, void * value)
```

`set_flow_option` is used to configure flow `fd`, by assigning option `value` to `option`.

```c
void * mapi_get_flow_option(flow_descriptor fd, int option)
```
APPENDIX D. MAN PAGES FOR MAPI FUNCTIONS

get_flow_option is used to gather information about the configuration of a flow fd, by returning a pointer to the value of option option.

packet *p = mapi_get_next_packet(flow_descriptor fd)

get_next_packet returns a pointer to the next available packet for flow fd. If no such packet exists, the call blocks until such a packet is received. The packet p is a sequence of bytes. This sequence starts with a header and continues with the network packet itself. The actual format of this sequence can be found later in this appendix when talking about the monitoring_record format option.

int mapi_loop(flow_descriptor fd,
int cnt, mapi_handler callback)

If users do not want to block in the process of receiving network packets from a flow using the get_next_packet call, they may invoke the mapi_loop call which invokes handler callback after it has received a packet of flow fd. The handler callback is invoked for the next cnt packets of flow fd. If cnt is -1, callback is invoked for all future packets of flow fd.

int mapi_apply_function(flow_descriptor fd, function f,...)

apply_function applies function f to all packets of the network flow fd. The function may compute statistics, hash functions, or anything else. If the application is successful, it returns a non-negative integer. Otherwise, it returns a negative one.

int mapi_remove_function(flow_descriptor fd, function f,...)

remove_function removes function f from being applied to packets of network flow fd. If no such function exists, a negative value is returned.

int mapi_save_to_disk(flow_descriptor fd, int file_descriptor,...)

mapi_save_to_disk saves all packets of network flow fd to the file file_descriptor. If the function fails, a negative value is returned.

int mapi_read_results(flow_descriptor fd, void * results)

read_results will receive statistics or any kind of results that have been computed by application of functions in the packets of flow fd. The results will be returned in a structure pointed to by pointer results. The function in case of successful completion, returns a non-negative integer.

MAPI can collect statistics on a flow using Flow Records. This is achieved with two variables. FLOW_RECORD is a logical variable which when set to TRUE enables the collection of Flow Records. FLOW_RECORD_TYPE can take values from the set

scampi@ist-scampi.org

122

March 27th, 2003
D.1. MAPI FUNCTIONS

{IPFIX, NETFLOW V5 NETFLOW V9}, selecting the format of the records that MAPI will maintain.

To read the collected flow records, an application can use the read_flow_record function:

```c
void* mapi_read_flow_record(flow_descriptor fd)
```

read_flow_record returns a flow record of the flow fd. The type of the record is IPFIX, NETFLOW V5, or NETFLOW V9, depending on the value of FLOW_RECORD_TYPE.

```c
sub_flow *p = mapi_get_next_subflow(flow_descriptor fd)
```

get_next_subflow finds the next expired subflow of the hierarchical flow fd. The function returns a pointer to a structure of type sub_flow. This call is blocking.

```c
int mapi_subflow_loop(flow_descriptor fd,
    int cnt, subflow_mapi_handler callback)
```

If the users do not want to block while waiting for sub-flows to expire, subflow_loop enables them to install a handler callback that will be invoked for the next cnt expired subflows. If cnt equals -1, then callback will be invoked for all sub-flows that will expire in the future.

```c
int mapi_settime (struct timespec *ts)
```

mapi_settime sets the adapter clock to the given value.

```c
int mapi_adjtime(long offset, long skew)
```

mapi_adjtime adjusts clock rate and offset in case clock is not disciplined by NTP process or external time source.

```c
int mapi_discipline(int par)
```

mapi_discipline start and stops disciplining of the adapter clock by external signal (for instance PPS -Pulse Per Second- signal from GPS receiver)

```c
int mapi_discipline(int par)
```

mapi_discipline start and stops disciplining of the adapter clock by external signal (for instance PPS -Pulse Per Second- signal from GPS receiver)

```c
management_info *result = get_management_info(int info_type)
```

get_management_info will provide information about the status of the monitoring hardware that is used and basic usage statistics.
D.2 MAPI Variables

The options available for network flows are 1:

packet_size

The size of packets in cooked network flows. The default value is 64 Kbytes.

packet_length

The maximum number of bytes of each packet that the monitoring environment will return to user applications. It can be adjusted to improve privacy and performance. Its default value is 64 bytes.

number_of_sub_flows

The number of bytes of sub-flows for each hierarchical flow. Its default value is 32K.

subflow_timeout

If no packet arrives for a subflow within subflow_timeout milliseconds, the flow is considered expired. The default value is one second.

max_flow_duration

The maximum duration of a network subflow. When the “life” of the flow exceeds max_flow_duration (measured in milliseconds), the flow is considered expired. The default value is 15 minutes.

interface_number

When a device has more than one interfaces, interface_number will be used to distinguish among them. Actually, interface_number is a bitmask. When its i'th bit is set, the flow receives packets from the i'th interface. If more than one bits are set, the flow receives packets from all these interfaces. Suppose for example that a DAG card (“/dev/dag”) has two interfaces. Using the following code, all packets from interface number 0 will arrive in flow fd1, and all packets from interface number 1 will arrive in flow fd2:

```
int if1 = 1 ; /* 0...0001 */
int if2 = 2 ; /* 0...0010 */

packet * p1, p2 ;
```

1Recall that these options may be modified with the set_flow_option(flow_descriptor, fd, int option, int value) function.

scampi@ist-scampi.org 124 March 27th, 2003
D.2. MAPI VARIABLES

```c
fd1 = mapi_create_flow("/dev/dag");
mapi_set_flow_option(fd1, interface_number, &if1);
p1 = mapi_get_next_packet(fd1);

fd2 = mapi_create_flow("/dev/dag");
mapi_set_flow_option(fd2, interface_number, &if2);
p2 = mapi_get_next_packet(fd2);
```

**no_copy**

If `no_copy` is TRUE, the application declares that it does not want to receive any packets in user space. It created the network flow just to apply functions to its packets and gather statistics. Thus, when packets arrive, the MAPI implementation does not keep a copy of them around for possible delivery to user applications. The default value is FALSE.

**condition**

It returns a string containing the condition `c` that was initially used to create this flow using the `create_flow(device_d, c, m)` MAPI call.

**total_number_of_packets**

The total number of packets captured by the flow

**total_number_of_octets**

The total number of octets captured by the flow

**total_number_of_dropped_packets**

The total number of dropped packets during the packet capture by the flow.

**flow_record**

If `flow_record` is TRUE, then the system gathers statistics for this network flow.

**flow_record_type**

The value of `flow_record_type` denotes the format in which applications would like to receive the collected statistics for this network flow. It can get values `IPFIX`, `NETFLOW_V5`, and `NETFLOW_V9`.

**monitoring_record_format**

scampi@ist-scampi.org

125

March 27th, 2003
This option defines that format of the packet records that will be returned to applications interested in passive monitoring. The `get_next_packet` function returns a pointer to structure that has some metadata about the packet followed by the IP network packet itself. It can take several values including:

- **PLAIN**: it returns just a timestamp and the length of the IP packet that immediately follows the structure:

  ```c
  struct plain_data {
    struct timeval ts ;
    int16_t ip_packet_len ;
  }
  ```

  Thus, if the user executes the following code:

  ```c
  packet * p = mapi_get_next_packet(fd) ;
  ```

  then the timestamp is at

  ```c
  ((struct plain_data *) p)->ts,
  ```

  the packet length is at

  ```c
  ((struct plain_data *) p)->ip_packet_len,
  ```

  and the IP packet starts from

  ```c
  p+sizeof(struct plain_data),
  ```

  We should probably stress that all the above pointer operations do not necessarily need to copy data between kernel space and user space. Indeed the can be implemented using the `mmap` system call in a transparent way. For example, the user-level MAPI library may decide to `mmap` a portion of memory that will be shared between user space and kernel space. In this way, the SCAMPI monitoring system that will reside in kernel space will write all network packets in this spared space, and the user-level MAPI implementation in response to the `get_next_packet` call will return pointers that reside within this memory mapped range. This memory mapped range may be as large as the monitoring library wants, so that wrapping of the pointers will not turn out to be a significant problem. On the other hand, we must stress that this memory mapped region can not be infinitely sized. Thus, if a user application continues to receive packets the `get_next_packet` will eventually need to reuse pointer values. This implies that `get_next_packet` returns the next packet in a portion of memory that may be overwritten in some (possibly distant) future call of `get_next_packet`. Thus, if users would like to have several outstanding packets in main memory, they will eventually need to allocate memory for them.
• **DAG_ERF/GLR**: this is the generic variable length record of the extensible record format supported by the DAG cards. We expect it to be used when the monitoring system is implemented on top of DAG cards.

```c
struct dag_record {
    unsigned long long ts;
    unsigned char type;
    flags_t flags;
    unsigned short rlen;
    unsigned short lctr;
    unsigned short wlen;
    union {
        pos_rec_t pos;
        eth_rec_t eth;
        atm_rec_t atm;
    } rec;
};
```

• **ETHERNET_HEADER**: it returns the packet along with its Ethernet headers and the length of the IP packet:

```c
struct ethernet_header_data {
    struct timeval ts ;
    struct sniff_ethernet sf ;
    int16_t ip_packet_len ;
}
```

```c
struct sniff_ethernet {
    /* Destination host address */
    u_char ether_dhost[ETHER_ADDR_LEN];
    /* Source host address */
    u_char ether_shost[ETHER_ADDR_LEN];
    u_short ether_type; /* IP? ARP? RARP? etc */
};
```

**FLOW_PACKET_RECORDS**: it returns flow/packet records as described in [11]. A “packet” in this format is actually a sequence of packet records headed by a flow record as described below:

```c
struct flow_record {
    uint8_t protocol ;
    uint8_t flags ;
    uint16_t record_number ;
    uint64_t timestamp ;
```
uint32_t source_address;
uint32_t destination_address;
uint16_t source_port;
uint16_t destination_port;
uint32_t initial_sequence_number;
uint32_t initial_ack_number;
uint32_t number_of_packet_records_that_follow;
}

struct packet_record {
  uint64_t timestamp;
  uint16_t total_length;
  uint16_t identification;
  uint8_t type_of_service;
  uint8_t time_to_live;
  uint8_t tcp_flags;
  uint8_t flags;
  uint16_t sequence_number_offset;
  uint16_t ack_number_offset;
  uint16_t ip_packet_len;
}

flow_packet_records_length

flow_packet_records_length is used with the FLOW_PACKET_RECORDS format to set the maximum number of bytes that the user is willing to accept in the future invocations of get_next_packet call. flow_packet_records_length is necessary because in the FLOW_PACKET_RECORDS there is no pre-set limit on how much information the system is going to return to the user, because the system does not return one network packet at-a-time, but a potentially large sequence of them. Thus, the users should tell the monitoring system how much bytes they have allocated, and based on this information the system should decide how many packets it should send them.

D.3 Variables for Hierarchical Flows

For each sub-flow kept within a hierarchical flow, the system keeps the following variables:

subflow_number_of_packets

The number of packets seen in the sub-flow

subflow_number_of_bytes

The number of bytes in the packets of this sub-flow.
D.3. VARIABLES FOR HIERARCHICAL FLOWS

subflow_start_time

The timestamp of the first observed packet of the sub-flow.

subflow_end_time

The timestamp of the last observed packet of the sub-flow.

avg_time_between_packet_arrivals

The average time between successive packet arrivals in microseconds.

std_dev_time_between_packet_arrivals

The standard deviation of the time between successive packet arrivals.

avg_packet_size

The average packet size.

std_dev_packet_size

The standard deviation of the packet sizes.

All the above variables are packed in a structure as follows:

struct sub_flow {
  long subflow_number_of_packets ;
  long subflow_number_of_bytes ;
  long subflow_start_time ;
  long avg_time_between_packet_arrivals ;
  double std_dev_time_between_packet_arrivals ;
  int avg_packet_size ;
  double std_dev_packet_size ;
}
Appendix E

SCAMPI MIB

SCAMPI-MIB DEFINITIONS ::= BEGIN

IMPORTS
   MODULE-IDENTITY, OBJECT-TYPE, NOTIFICATION-TYPE, Counter32, Counter64,
   Gauge32, enterprises FROM SNMPv2-SMI
   DisplayString, TimeStamp
   FROM SNMPv2-TC
   IANAifType FROM IANAifType-MIB;

uninett OBJECT IDENTIFIER ::= { enterprises 2428 }
uninettExperiment OBJECT IDENTIFIER ::= { uninett 2428 }

scampiMIB MODULE-IDENTITY
   LAST-UPDATED "0302260000Z"
   ORGANIZATION "SCAMPI Consortium"
   CONTACT-INFO
   *URL: http://www.ist-scampi.org
   Email: info@ist-scampi.org

   Editor: Arne Oslebo
   UNINETT
   Postal: N-7465 Trondheim
   Norway
   Email: Arne.Oslebo@uninett.no*

   DESCRIPTION
   "The MIB module to describe SCAMPI platform related objects."

   ::= { uninettExperiment 123 }

scampiMIBObjects OBJECT IDENTIFIER ::= { scampiMIB 1 }
scampiTraps OBJECT IDENTIFIER ::= { scampiMIB 2 }
scampiMIBConformance OBJECT IDENTIFIER ::= { scampiMIB 3 }

-- Devices group ******************************************
-- The devices group provides information about devices and interfaces which
APPENDIX E. SCAMPI MIB

-- are available for measurement through MAPI

scampiDevices OBJECT IDENTIFIER ::= { scampiMIBObjects 1 }

scampiDeviceTable OBJECT-TYPE
SYNTAX SEQUENCE OF ScampiDeviceEntry
MAX-ACCESS not-accessible
STATUS current
DESCRIPTION "Information about each available device"
::= { scampiDevices 1 }

scampiDeviceEntry OBJECT-TYPE
SYNTAX ScampiDeviceEntry
MAX-ACCESS not-accessible
STATUS current
DESCRIPTION "An entry in this table provides information about a specific device."
INDEX { scampiDeviceIndex }
::= { scampiDeviceTable 1 }

ScampiDeviceEntry ::= SEQUENCE
{
scampiDeviceIndex Integer32,
scampiDeviceName DisplayString,
scampiDeviceDescr DisplayString,
scampiDeviceAlias DisplayString,
scampiDeviceIFNum Integer32,
scampiDeviceGPSync TruthValue
}

scampiDeviceIndex OBJECT-TYPE
SYNTAX Integer32
MAX-ACCESS not-accessible
STATUS current
DESCRIPTION "A unique value, greater than zero, for each device available for monitoring through MAPI. It is recommended that the values are assigned contiguously starting from one and remain constant from one re-initialization of the system to the next re-initialization."
::={ scampiDeviceEntry 1 }

scampiDeviceName OBJECT-TYPE
SYNTAX DisplayString (SIZE (0..64))
MAX-ACCESS read-only
STATUS current
DESCRIPTION "A textual string containing the name of the device. The name should uniquely identify the device in the host system. An example of a device name is '/dev/eth1'"
::={ scampiDeviceEntry 2 }

scampiDeviceDescr OBJECT-TYPE
SYNTAX DisplayString (SIZE (0..255))
MAX-ACCESS read-only
STATUS current
DESCRIPTION "A textual string containing information about the device. The
string should include the name of the manufacturer, the product
name and the version of the device hardware/software.

::={ scampiDeviceEntry 3 }

scampiDeviceAlias OBJECT-TYPE
SYNTAX DisplayString (SIZE (0..64))
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"This object is an ‘alias’ name for the device as
specified by a network manager, and provides a non-volatile
‘handle’ for the device.

On the first instantiation of a device, the value of
scampiDeviceAlias associated with that device is the zero-length
string. As and when a value is written into an instance of
scampiDeviceAlias through a network management set operation, then the
agent must retain the supplied value in the scampiDeviceAlias instance
associated with the same device for as long as that device remains
instantiated, including across all re-initializations/reboots of the
network management system, including those which result in a change of
the device’s scampiDeviceIndex value."

::={ scampiDeviceEntry 4 }

scampiDeviceIfNum OBJECT-TYPE
SYNTAX INTEGER32
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"An integer representing the number of interfaces on the device"

::={ scampiDeviceEntry 5 }

scampiDeviceGPSSync OBJECT-TYPE
SYNTAX TruthValue
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"A boolean value used for signalling GPS time synchronization problems.
Many devices will support GPS time synchronization. A true value signals
that synchronization with GPS is working. A false value signals that the
device is not accurately synchronized with GPS. Devices that does not
support GPS time synchronization should set this value to false."

::={ scampiDeviceEntry 6 }

-- scampiDeviceIfTable **********************************************

scampiDeviceIfTable OBJECT-TYPE
SYNTAX SEQUENCE OF ScampiDeviceIfEntry
MAX-ACCESS not-accessible
STATUS current
DESCRIPTION
"Information about each available interface."

::={ scampiDevices 2 }

scampiDeviceIfEntry OBJECT-TYPE
SYNTAX ScampiIfEntry
MAX-ACCESS not-accessible
STATUS current

scampi@ist-scampi.org 133 March 27th, 2003
APPENDIX E. SCAMPI MIB

DESCRIPTION  "An entry in this table provides information about a specific interface"
INDEX  { scampiDevIfIndex }
 ::= { scampiDeviceIfTable 1 }

ScampiDeviceIfEntry ::= SEQUENCE
{
scampiDevIfIndex Integer32,
scampiDevIfDeviceIndex Integer32,
scampiDevIfType IANAifType,
scampiDevIfSpeed Gauge32,
scampiDevIfAlias DisplayString,
scampiDevIfStatus Integer32,
scampiDevIfPkts Counter64,
scampiDevIfOctets Counter64,
scampiDevIfDroppedPkts Counter64,
scampiDevIfCounterDiscontinuityTime TimeStamp
}

scampiDevIfIndex OBJECT-TYPE
SYNTAX Integer32
MAX-ACCESS not-accessible
STATUS current
DESCRIPTION
"A unique value, greater than zero, for each interface available for monitoring through MAPI. It is recommended that the values are assigned contiguously starting from one and remain constant from one re-initialization of the system to the next re-initialization."
 ::= { scampiDeviceIfEntry 1 }

scampiDevIfDeviceIndex OBJECT-TYPE
SYNTAX Integer32
MAX-ACCESS not-accessible
STATUS current
DESCRIPTION
"A reference to scampiDeviceIndex identifying which device this interface belongs to."
 ::= { scampiDeviceIfEntry 2 }

scampiDevIfType OBJECT-TYPE
SYNTAX IANAifType
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"The type of interface. Additional values for ifType are assigned by the Internet Assigned Numbers Authority (IANA), through updating the syntax of the IANAifType textual convention."
 ::= { scampiDeviceIfEntry 3 }

scampiDevIfSpeed OBJECT-TYPE
SYNTAX Gauge32
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"The supported bandwidth of the interface in units of 1,000,000 bits per second. If this object reports a
value of 'n' then the speed of the interface is somewhere in
the range of 'n-500,000' to 'n+499,999'."
::= { scampiDeviceIfEntry 4 }

scampiDevIfAlias OBJECT-TYPE
SYNTAX DisplayString (SIZE (0..64))
MAX-ACCESS read-write
STATUS current
DESCRIPTION
"This object is an 'alias' name for the interface as
specified by a network manager, and provides a non-volatile
'handle' for the interface.

On the first instantiation of an interface, the value of
scampiDevIfAlias associated with that interface is the zero-length
string. As and when a value is written into an instance of
scampiDevIfAlias through a network management set operation, then the
agent must retain the supplied value in the scampiDevIfAlias instance
associated with the same interface for as long as that
interface remains instantiated, including across all re-
initializations/reboots of the network management system,
including those which result in a change of the interface’s
scampiDevIfIndex value."
::= { scampiDeviceIfEntry 5 }

scampiDevIfStatus OBJECT-TYPE
SYNTAX INTEGER32 { active(1), -- currently being used for measurements
ready(2), -- ready to be used for measurements
unavailable(3), -- unavailable for measurements
linkLost(4), -- network link is down
unknown(5) -- status of interface can not be determined }
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"The current status of the interface."
::= { scampiDeviceIfEntry 6 }

scampiDevIfPkts OBJECT-TYPE
SYNTAX Counter64
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"The total number of packets captured by the interface.
Discontinuities in the value of this counter can occur at
re-initialization of the management system, and at other times as
indicated by the value of scampiIfCounterDiscontinuityTime."
::= { scampiDeviceIfEntry 7 }

scampiDevIfOctets OBJECT-TYPE
SYNTAX Counter64
MAX-ACCESS read-only
STATUS current

scampi@ist-scampi.org 135 March 27th, 2003
APPENDIX E. SCAMPI MIB

DESCRIPTION
"The total number of octets captured by the interface.

Discontinuities in the value of this counter can occur at
re-initialization of the management system, and at other times as
indicated by the value of scampiDevIfCounterDiscontinuityTime."
 ::= { scampiDeviceIfEntry 8 }

scampiDevIfDroppedPkts OBJECT-TYPE
SYNTAX Counter64
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"The total number of dropped packets during packet capture by the
interface.

Discontinuities in the value of this counter can occur at
re-initialization of the management system, and at other times as
indicated by the value of scampiIfCounterDiscontinuityTime."
 ::= { scampiDeviceIfEntry 9 }

scampiDevIfCounterDiscontinuityTime OBJECT-TYPE
SYNTAX TimeStamp
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"The value of sysUpTime on the most recent occasion at which
any one or more of this interface’s counters suffered a
discontinuity."
 ::= { scampiDeviceIfEntry 10 }

-- General information about MAPI ************************************

scampiMapi OBJECT IDENTIFIER ::= { scampiMIBObjects 2 }

scampiMapiUsers OBJECT-TYPE
SYNTAX Gauge32
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"The total number of unique users that are currently using MAPI"
 ::= { scampiMapi 1 }

scampiMapiFlows OBJECT-TYPE
SYNTAX Gauge32
MAX-ACCESS read-only
STATUS current
DESCRIPTION
"The total number of active flows in MAPI"
 ::= { scampiMapi 2 }

scampiMapiFunctions OBJECT-TYPE
SYNTAX Gauge32
MAX-ACCESS read-only
STATUS current
DESCRIPTION

scampi@ist-scampi.org 136 March 27th, 2003
"The total number of predefined functions in MAPI that are currently in use"
 ::= { scampiMapi 3 }

scampiMapiUserFunctions OBJECT-TYPE
 SYNTAX Gauge32
 MAX-ACCESS read-only
 STATUS current
 DESCRIPTION
 "The total number of user defined functions registered with MAPI that are currently in use"
 ::= { scampiMapi 4 }

-- Information about MAPI flows *******************************************
-- This group provides information about open flows in MAPI as well as a history of past flows.

scampiFlows OBJECT IDENTIFIER ::= { scampiMIBObjects 3 }

scampiFlowCfgMaxHistLength OBJECT-TYPE
 SYNTAX Integer32
 MAX-ACCESS read-write
 STATUS current
 DESCRIPTION
 "Specifies the maximum number of finished flows that are displayed in the scampiFlowTable.
 A value of 0 means that there are no limit."
 ::= { scampiFlows 1 }

scampiFlowCfgMaxTime OBJECT-TYPE
 SYNTAX Integer32
 MAX-ACCESS read-write
 STATUS current
 DESCRIPTION
 "Specifies the maximum age in seconds of entries in scampiFlowTable.
 This is the number of seconds since scampiFlowStop.
 A value of 0 means that there are no limit"
 ::= { scampiFlows 2 }

scampiFlowTable OBJECT-TYPE
 SYNTAX SEQUENCE OF ScampiFlowEntry
 MAX-ACCESS not-accessible
 STATUS current
 DESCRIPTION
 "Information about active and closed flows created through MAPI"
 ::= { scampiFlows 3 }

scampiFlowEntry OBJECT-TYPE
 SYNTAX ScampiFlowEntry
 MAX-ACCESS not-accessible
 STATUS current
 DESCRIPTION
 "An entry in this table provides information about a specific flow in MAPI"
 INDEX { scampiFlowIndex scampiFlowKID }
 ::= { scampiFlowTable 1 }

scampi@ist-scampi.org 137 March 27th, 2003
APPENDIX E. SCAMPI MIB

ScampiFlowEntry ::= SEQUENCE
{ scampiFlowIndex Integer32, scampiFlowKID Integer32, scampiFlowIfIndex Integer32, scampiFlowCondition DisplayString, scampiFlowPkts Counter64, scampiFlowOctets Counter64, scampiFlowDroppedPkts Counter64, scampiFlowGPSSync TruthValue, scampiFlowStart TimeStamp, scampiFlowStop TimeStamp }

scampiFlowIndex OBJECT-TYPE
SYNTAX Integer32
MAX-ACCESS not-accessible
STATUS current
DESCRIPTION "A unique integer value used for identifying the flow."
 ::= { scampiFlowEntry 1 }

scampiFlowKID OBJECT-TYPE
SYNTAX Integer32
MAX-ACCESS not-accessible
STATUS current
DESCRIPTION "The ID of the authentication key that was used by the process that initiated this flow."
 ::= { scampiFlowEntry 2 }

scampiFlowIfIndex OBJECT-TYPE
SYNTAX Integer32
MAX-ACCESS read-only
STATUS current
DESCRIPTION "The scampiIfIndex of the interface that the flow uses to capture packets"
 ::= { scampiFlowEntry 3 }

scampiFlowCondition OBJECT-TYPE
SYNTAX DisplayString (SIZE (0..255))
MAX-ACCESS read-only
STATUS current
DESCRIPTION "The flow condition used when creating the flow"
 ::= { scampiFlowEntry 4 }

scampiFlowPkts OBJECT-TYPE
SYNTAX Counter64
MAX-ACCESS read-only
STATUS current
DESCRIPTION "The total number of packets captured by the flow."
 ::= { scampiFlowEntry 5 }

scampi@ist-scampi.org 138 March 27th, 2003
scampiFlowOctets OBJECT-TYPE
  SYNTAX Counter64
  MAX-ACCESS read-only
  STATUS current
  DESCRIPTION
  "The total number of octets captured by the flow."
  ::= { scampiFlowEntry 6 }

scampiFlowDroppedPkts OBJECT-TYPE
  SYNTAX Counter64
  MAX-ACCESS read-only
  STATUS current
  DESCRIPTION
  "The total number of dropped packets during packet capture by the
the flow"
  ::= { scampiFlowEntry 7 }

scampiFlowGPSSync OBJECT-TYPE
  SYNTAX TruthValue
  MAX-ACCESS read-only
  STATUS current
  DESCRIPTION
  "A boolean value that indicates GPS synchronization problems during
the lifetime of the flow. A value of 1 indicates that a problem
occurred while a value of 0 indicates no problems."
  ::= { scampiFlowEntry 8 }

scampiFlowStart OBJECT-TYPE
  SYNTAX TimeStamp
  MAX-ACCESS read-only
  STATUS current
  DESCRIPTION
  "The sysUpTime of when the flow started."
  ::= { scampiFlowEntry 9 }

scampiFlowStop OBJECT-TYPE
  SYNTAX TimeStamp
  MAX-ACCESS read-only
  STATUS current
  DESCRIPTION
  "The sysUpTime of when the flow finished. A value of 0 indicates
that the flow is still active."
  ::= { scampiFlowEntry 10 }

-- Measurement group ***********************************************
-- The measurement group provides an SNMP interface for doing measurments
-- of traffic in specific intervals including sub second intervals.

scampiMeasurements OBJECT IDENTIFIER ::= { scampiMIBObjects 4 }

scampiMesCfgTable OBJECT-TYPE
  SYNTAX SEQUENCE OF ScampiMesCfgEntry
  MAX-ACCESS not-accessible
  STATUS current
  DESCRIPTION
  "Configuration parameters for measurements where results are shown
in scampiMesTable"
APPENDIX E. SCAMPI MIB

::={ scampiMeasurements 1 }

scampiMesCfgEntry OBJECT-TYPE
SYNTAX ScampiMesCfgEntry
MAX-ACCESS not-accessible
STATUS current
DESCRIPTION "An entry in this table provides configuration parameters for the corresponding measurement in scampiMesTable"
INDEX { scampiMesCfgKID scampiMesCfgIndex }
 ::= { scampiMesCfgTable 1 }

ScampiMesCfgEntry ::= SEQUENCE
{
  scampiMesCfgKID Integer32,
  scampiMesCfgIndex Integer32,
  scampiMesCfgIfIndex Integer32,
  scampiMesCfgIntervalSec Integer32,
  scampiMesCfgIntervalFrac Integer32,
  scampiMesCfgMaxLength Integer32,
  scampiMesCfgActive TruthValue,
  scampiMesCfgStorageType StorageType,
  scampiMesCfgRowStatus RowStatus
}

scampiMesCfgKID OBJECT-TYPE
SYNTAX Integer32
MAX-ACCESS not-accessible
STATUS current
DESCRIPTION "The ID of the authentication key belonging to the user who created and controls this measurement"
 ::= { scampiMesCfgEntry 1 }

scampiMesCfgIndex OBJECT-TYPE
SYNTAX Integer32
MAX-ACCESS not-accessible
STATUS current
DESCRIPTION "The index of the measurement. Should be unique for the corresponding scampiMesCfgKID value."
 ::= { scampiMesCfgEntry 2 }

scampiMesCfgIfIndex OBJECT-TYPE
SYNTAX Integer32
MAX-ACCESS not-accessible
STATUS current
DESCRIPTION "Reference to scampiDevIfIndex and specifies which interface that should be used for measurements."
 ::= { scampiMesCfgEntry 3 }

scampiMesCfgIntervalSec OBJECT-TYPE
SYNTAX Integer32
MAX-ACCESS read-create
STATUS current
DESCRIPTION
"scampiMesCfgIntervalSec and scampiMesCfgIntervalFrac together form the
time interval for measurements. scampiMesCfgIntervalSec specifies the number
of whole seconds of the interval."
::={ scampiMesCfgEntry 4 }

scampiMesCfgIntervalFrac OBJECT-TYPE
SYNTAX Integer32
MAX-ACCESS read-create
STATUS current
DESCRIPTION
"scampiMesCfgIntervalSec and scampiMesCfgIntervalFrac together form the
time interval for measurements. scampiMesStartFrac is the sub-second
part of the interval in units of 2^-32 seconds."
::={ scampiMesCfgEntry 5 }

scampiMesCfgMaxLength OBJECT-TYPE
SYNTAX Integer32
MAX-ACCESS read-create
STATUS current
DESCRIPTION
"The maximum number of entries for this measurement in the
scampiMesTable."
::={ scampiMesCfgEntry 6 }

scampiMesCfgActive OBJECT-TYPE
SYNTAX TruthValue
MAX-ACCESS read-create
STATUS current
DESCRIPTION
"If set to 1 the measurement is active and results are stored in
scampiMesTable. A value of 0 indicates that the measurement is
inactive and no results are put into scampiMesTable."
::={ scampiMesCfgEntry 7 }

scampiMesCfgStorageType OBJECT-TYPE
SYNTAX StorageType
MAX-ACCESS read-create
STATUS current
DESCRIPTION
"The storage type of this conceptual row."
::={ scampiMesCfgEntry 8 }

scampiMesCfgRowStatus OBJECT-TYPE
SYNTAX RowStatus
MAX-ACCESS read-create
STATUS current
DESCRIPTION
"The status of this conceptual row.

The RowStatus TC [RFC2579] requires that this
DESCRIPTION clause states under which circumstances
other objects in this row can be modified:

The value of this object has no effect on whether
other objects in this conceptual row can be modified."
scampiMesTable OBJECT-TYPE
  SYNTAX SEQUENCE OF ScampiMesEntry
  MAX-ACCESS not-accessible
  STATUS current
  DESCRIPTION
  "Measurement results"
  ::= { scampiMeasurements 2 }

scampiMesEntry OBJECT-TYPE
  SYNTAX ScampiMesEntry
  MAX-ACCESS not-accessible
  STATUS current
  DESCRIPTION
  "An entry in this table provides information about the number
  of packets and bytes captured in a certain time interval"
  INDEX { scampiMesKID scampiMesIndex scampiMesIntervalId }
  ::= { scampiMesTable 1 }

ScampiMesEntry ::= SEQUENCE
  {
    scampiMesKID       Integer32,
    scampiMesIndex     Counter32,
    scampiMesIntervalId Counter32,
    scampiMesStartSec  Integer32,
    scampiMesStartFrac Integer32,
    scampiMesPkts      Counter64,
    scampiMesOctets    Counter64
  }

scampiMesUID OBJECT-TYPE
  SYNTAX Integer32
  MAX-ACCESS not-accessible
  STATUS current
  DESCRIPTION
  "The UID of the user who created and controls this measurement"
  ::= { scampiMesEntry 1 }

scampiMesIndex OBJECT-TYPE
  SYNTAX Counter32
  MAX-ACCESS not-accessible
  STATUS current
  DESCRIPTION
  "The index of the measurement."
  ::= { scampiMesEntry 2 }

scampiMesIntervalId OBJECT-TYPE
  SYNTAX Integer32
  MAX-ACCESS not-accessible
  STATUS current
  DESCRIPTION
  "A unique ID for the time interval. If the corresponding
  scampiMesCfgMaxHistLength is too large and scampiMesIntervalId
  wraps, old intervals should be overwritten."
  ::= { scampiMesEntry 3 }
scampiMesStartSec OBJECT-TYPE
SYNTAX     Integer32
MAX-ACCESS read-only
STATUS     current
DESCRIPTION
"scampiMesStartSec and scampiMesStartFrac together form a timestamp
for when the interval started. scampiMesStartSec contains number
of second since midnight January 1 1970."
 ::= { scampiMesEntry 4 }

scampiMesStartFrac OBJECT-TYPE
SYNTAX     Integer32
MAX-ACCESS read-only
STATUS     current
DESCRIPTION
"scampiMesStartSec and scampiMesStartFrac together form a timestamp
for when the interval started. scampiMesStartFrac is the sub-second
part of the timestamp in units of 2^-32 seconds."
 ::= { scampiMesEntry 5 }

scampiMesPkts OBJECT-TYPE
SYNTAX     Counter32
MAX-ACCESS read-only
STATUS     current
DESCRIPTION
"Total number of packets captured during the interval"
 ::= { scampiMesEntry 6 }

scampiMesOctets OBJECT-TYPE
SYNTAX     Counter32
MAX-ACCESS read-only
STATUS     current
DESCRIPTION
"Total number of bytes captured during the interval"
 ::= { scampiMesEntry 7 }

-- Notifications ***********************************************

scampiIfStatus NOTIFICATION-TYPE
STATUS     current
DESCRIPTION "This trap is sent when an interface changes the value of
scampiDevIfStatus."
 ::= { scampiTraps 1 }

scampiGPSSyncLost NOTIFICATION-TYPE
STATUS     current
DESCRIPTION "This trap is sent when GPS synchronization is lost."
 ::= { scampiTraps 2 }

-- Conformance information ***************************************

scampiMIBCompliances OBJECT IDENTIFIER ::= { scampiMIBConformance 1 }
scampiMIBGroups OBJECT IDENTIFIER ::= { scampiMIBConformance 2 }

-- Compliance statements ******************************************
APPENDIX E. SCAMPI MIB

scampiBasicMIBCompliance MODULE-COMPLIANCE
  STATUS current
  DESCRIPTION "The compliance statement for SNMP engines which implements
  the basic SCAMPI MIB"

  MODULE -- this module
  MANDATORY-GROUPS { scampiBasicGroup }

  OBJECT scampiDeviceAlias
  MIN-ACCESS read-only
  DESCRIPTION "Write access not required."

  OBJECT scampiDevIfAlias
  MIN-ACCESS read-only
  DESCRIPTION "Write access not required."

::= { scampiMIBCompliances 1 }

scampiExtendedMIBCompliance MODULE-COMPLIANCE
  STATUS current
  DESCRIPTION "The compliance statement for SNMP engines which implements
  the extended SCAMPI MIB"

  MODULE -- this module
  MANDATORY-GROUPS { scampiBasicGroup, scampiExtendedGroup }

  OBJECT scampiDeviceAlias
  MIN-ACCESS read-only
  DESCRIPTION "Write access not required."

  OBJECT scampiDevIfAlias
  MIN-ACCESS read-only
  DESCRIPTION "Write access not required."

  OBJECT scampiFlowCfgMaxHistLength
  MIN-ACCESS read-only
  DESCRIPTION "Write access not required."

  OBJECT scampiFlowCfgMaxTime
  MIN-ACCESS read-only
  DESCRIPTION "Write access not required."

::= { scampiMIBCompliances 2 }

scampiFullMIBCompliance MODULE-COMPLIANCE
  STATUS current
  DESCRIPTION "The compliance statement for SNMP engines which implements
  the full SCAMPI MIB"

  MODULE -- this module
  MANDATORY-GROUPS { scampiBasicGroup, scampiExtendedGroup, scampiAdvancedGroup }

  OBJECT scampiDeviceAlias
  MIN-ACCESS read-only
  DESCRIPTION "Write access not required."
OBJECT scampiDevIfAlias
MIN-ACCESS read-only
DESCRIPTION "Write access not required."

OBJECT scampiFlowCfgMaxHistLength
MIN-ACCESS read-only
DESCRIPTION "Write access not required."

OBJECT scampiFlowCfgMaxTime
MIN-ACCESS read-only
DESCRIPTION "Write access not required."

::= { scampiMIBCompliances 3 }

-- Units of conformance

scampiBasicGroup OBJECT-GROUP
OBJECTS {
  scampiDeviceName,  
  scampiDeviceDescr,  
  scampiDeviceAlias,  
  scampiDeviceIfNum,  
  scampiDeviceGPSSync,  
  scampiDevIfDeviceIndex,  
  scampiDevIfType,  
  scampiDevIfSpeed,  
  scampiDevIfAlias,  
  scampiDevIfStatus,  
  scampiDevIfPktS,  
  scampiDevIfOctets,  
  scampiDevIfDroppedPkts,  
  scampiDevIfCounterDiscontinuityTime,  
  scampiMapiUsers,  
  scampiMapiFlows,  
  scampiMapiFunctions,  
  scampiMapiUserFunctions
}
STATUS current
DESCRIPTION "Collection of mandatory objects for basic compliance"
::= { scampiMIBGroups 1 }

scampiExtendedGroup OBJECT-GROUP
OBJECTS {
  scampiFlowCfgMaxHistLength,  
  scampiFlowCfgMaxTime,  
  scampiFlowIfIndex,  
  scampiFlowPktS,  
  scampiFlowOctets,  
  scampiFlowDroppedPkts,  
  scampiFlowStart,  
  scampiFlowStop
}
STATUS current
DESCRIPTION "Collection of objects for extended compliance"
APPENDIX E. SCAMPI MIB

::={scampiMIBGroups 2}

scampiAdvancedGroup OBJECT-GROUP
OBJECTS {
  scampiMesCfgIfIndex,
  scampiMesCfgIntervalSec,
  scampiMesCfgIntervalFrac,
  scampiMesCfgMaxLength,
  scampiMesCfgActive,
  scampiMesCfgStorageType,
  scampiMesCfgRowStatus,
  scampiMesIntervalId,
  scampiMesStartSec,
  scampiMesStartFrac,
  scampiMesPkts,
  scampiMesOctets
}

STATUS current
DESCRIPTION "Collection of advanced objects for full compliance"
::={scampiMIBGroups 3}

END

E.1 mapi.h

//Management structures

enum mapiFunctType { ALL, PREDEFINED, USERDEFINED };
enum mapiIfStatus {ACTIVE, READY, UNAVAILABLE, LINKLOST, UNKNOWN };

typedef struct mapiDeviceList {
  char **devices; //Array of device names
  unsigned length; //Length of array
} mapiDeviceList_t;

typedef struct mapiDeviceInfo {
  char *name; //Name of device
  char *description; //Description of device
  unsigned numInterfaces; //Number of interfaces on device
  int gpsSync; //Positive value if device is synchronized with GPS receiver
} mapiDeviceInfo_t;

typedef struct mapiInterfaceInfo {
  unsigned ifType; //Type of interface
  unsigned ifSpeed; //Speed of interface
  enum mapiIfStatus ifStatus; //Status of interface
  unsigned long long pkts; //Total captured packets
  unsigned long long octets; //Total captured octets
  unsigned long long droppedPkts; //Total dropped packets
} mapiInterfaceInfo_t;

typedef struct mapiFlowList {
  flow_descriptor **flows; //Array of flow descriptors
  unsigned length; //Length of array
} mapiFlowList_t;
typedef struct mapiFlowInfo {
    unsigned KID; //ID of the authentication key used by the process that owns the flow
    char* dev; //Device this flow capture packets from
    unsigned interface; //Interface on the device this flow captures packets from
    unsigned long long pkts; //Total captured packets
    unsigned long long octets; //Total captured octets
    unsigned long long droppedPkts; //Total dropped packets
} mapiFlowInfo_t;

//Management functions

unsigned mapi_get_num_users();
//Returns the number of active users

unsigned mapi_get_num_flows();
//Returns the number of active flows

unsigned mapi_get_num_funct(enum mapiFunctType type);
//Returns the number of active functions.
//Parameters:
// type = type of function

mapiDeviceList_t* mapi_get_device_list();
//Returns a list of available devices

mapiDeviceInfo_t* mapi_get_device_info(char *device);
//Returns information about a device
//Parameters:
// device = name of device

mapiInterfaceInfo_t* mapi_get_interface_info(char *device, unsigned interface);
//Returns information about an interface
//Parameters:
// device = name of device
// interface = interface number on the device

mapiFlowList_t* mapi_get_flow_list();
//Returns a list of active flows

mapiFlowInfo_t* mapi_get_flow_info(flow_descriptor* fd);
//Returns information about an open flow
//Parameters:
// fd = flow descriptor
Appendix F

Timestamp issues

Each packed passed the filter has to get timestamp in order to evaluate time related parameters and characteristics of a single packet or a particular flow.

We use terms resolution, accuracy, precision, offset and skew according to RFC-2330 [27] and with respect to RFC-1305 [23]:

**Resolution** The smallest unit by which the clock is advanced (sometime called granularity).

**Accuracy** The offset between local and some other (usually UTC) clock.

**Precision** The smallest unit in which time is reported.

**Skew** The difference in rate of local clock to some reference (usually UTC). It can be expressed as first derivation of offset, too.

The resolution of timestamps should be fine enough to measure offset of two successive packets or cells on the interface. This implies sub-microsecond granularity (e.g. distance of two smallest, 640 bytes long packets, at Gigabit Ethernet is 5 us, while ATM cells at STM16 come every 160 ns).

Beside this requirement of the timestamp resolution, the highest possible absolute accuracy is also important. For instance, the one-way delay measurement depends on timestamp accuracy as it requires to know both timestamp of packet sending and receiving. The accuracy should not be worse then 1 millisecond, however it is preferred to be two orders of magnitude better (i.e. about ten microseconds)
Bibliography


[22] Endance measurement systems. DAGH 4.2GE dual gigabit ethernet network interface card, 2002.


scampi@ist-scampi.org  152  March 27th, 2003


scampi@ist-scampi.org 153 March 27th, 2003